



PRAKTIKUM MEDIZININFORMATIK MIT BACHELORARBEIT

Titel

Medication Management App with FHIR

Author

Anreiter Simon

Vienna, March 2016

Studienkennzahl lt. Studienblatt: A 521
Studienrichtung lt. Studienblatt: Informatik
Erstbetreuer: Dipl.-Ing. Dr. Christoph Rinner

Contents

1	Project Setup	4
1.1	Prerequisites	4
1.2	Getting Started	4
1.3	Conclusion	6
2	Managing Dependencies with CocoaPods[1]	6
2.1	Prerequisites	6
2.2	Getting Started	7
2.2.1	Adding CocoaPods to the project	7
2.2.2	Specifying the dependencies	7
2.2.3	Installing the dependencies	7
2.3	Conclusion	7
3	Overview and Explanation of used FHIR-Resources	8
3.1	Goals	8
3.2	Overview	8
3.3	Resources	8
3.3.1	Patient	8
3.3.2	Medication	10
3.3.3	MedicationOrder	10
3.3.4	DosageInstructions	11
3.3.5	Timing	12
3.3.6	MedicationAdministration	13
4	Building the Application Structure	13
4.1	Prerequisites	13
4.2	Goals	13
4.3	Getting started	14
4.4	Adding a TabBarController	14
4.5	Building the Navigation-flow	14
4.6	Adding the SignInViewController	15
4.7	Adding the PatientDetailViewController	15
4.8	Conclusion	16
5	Implementing SignIn	16
5.1	Prerequisites	16
5.2	Goals	16
5.3	Getting started	17
5.4	Implementing the SessionManager	17
5.4.1	Creating the SessionManager	17
5.4.2	Creating the LogInCredentials	18
5.4.3	Modeling the result of the Log-In task	19
5.5	PatientSignInViewController Interface	20
5.5.1	Building the Interface	21

5.5.2	Adding outlets to the class implementation	21
5.5.3	Adding the UI-Elements in the storyboard	22
5.6	PatientSignInViewController functionality	22
5.6.1	Adding a completionHandler	22
5.6.2	Implement <code>submitButtonPressed</code> action	23
5.7	Conclusion	25
6	Implementing PatientMedicationsViewController	25
6.1	Prerequisites	25
6.2	Goals	25
6.3	Getting started	26
6.4	PatientMedicationsViewController Interface	26
6.4.1	Creating the StatusView	26
6.4.2	Defining outlets in the implementation class	26
6.5	Ensuring the user is logged in	30
6.5.1	Showing the PatientSignInViewController	30
6.6	Generating Instructions for Prescriptions	31
6.6.1	Overview	31
6.6.2	TimingUnit	32
6.6.3	EventTiming	32
6.6.4	Duration	34
6.6.5	Frequency	34
6.6.6	Period	34
6.6.7	TimingBounds	35
6.6.8	Bringing it together	35
6.7	Loading and displaying all prescriptions for the logged-in user . .	37
6.7.1	Creating the MedicationOrderTableViewCell	38
6.7.2	Configuring the MedicationOrderTableViewCell	38
6.7.3	Preparing the display of MedicationOrderTableViewCell	38
6.7.4	Loading the results	39
7	Implementing the MedicationDetailViewController	40
7.1	Prerequisites	40
7.2	Getting Started	40
7.2.1	Adding the ResearchKit dependency	41
7.3	MedicationDetailViewController Interface	41
7.3.1	Create the implementation class	41
7.3.2	Adding the HeaderView	41
7.3.3	Defining the outlets and actions	41
7.3.4	Create the MedicationAdministrationTableViewCell	43
7.4	Loading and displaying all administrations for a prescription . .	43
7.4.1	Defining the properties	43
7.4.2	Implementing UITableViewDataSource methods	44
7.4.3	Configuring the MedicationAdministrationTableViewCell . . .	44
7.4.4	Implement loading of administrations	45
7.5	Rendering a chart visualizing a timeline of administrations . . .	46

7.5.1	Configuring the view for state	46
7.5.2	Configuring the view for the prescription	47
7.5.3	Implementing the SegmentedControl functionality	48
7.6	Implementing the Chart	49
7.6.1	Generating identifiers	50
7.6.2	Creating the Charts Data-Model	51
7.6.3	Updating the Chart	53
7.6.4	Implementing the ORKGraphChartViewDataSource protocol methods	53
7.7	Connecting MedicationDetailViewController and PatientMedicationsViewController	54
7.8	Conclusion	55
8	Implementing the CreateMedicationAdministrationViewController	55
8.1	Prerequisites	55
8.2	Goals	55
8.3	Building the Interface	56
8.4	Defining the outlets and actions	56
8.5	Creating MedicationAdministration on the server	59
8.5.1	Implementing the actions	59

1 Project Setup

1.1 Prerequisites

- Installed XCode 7.3 (Current Version as of writing this)

1.2 Getting Started

1. Create your **working directory** for example run `mkdir medbrain` in the Terminal
2. Open XCode and create a new Project

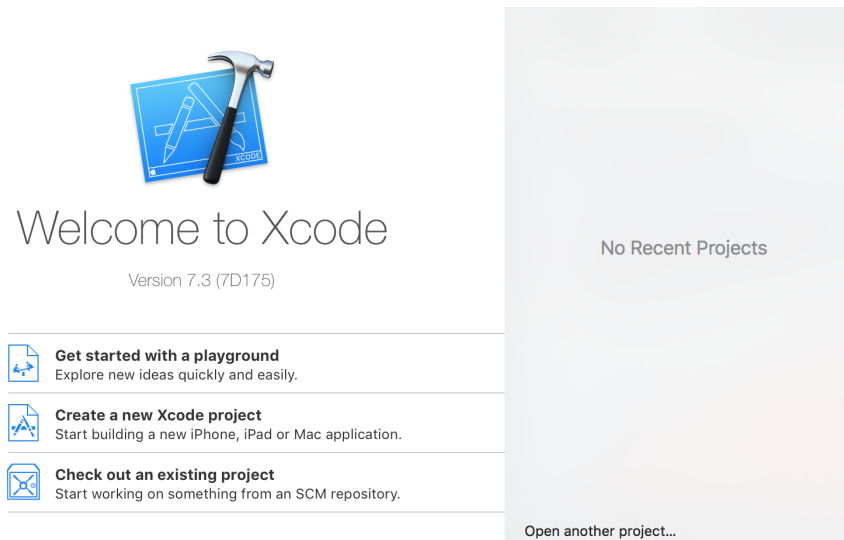


Figure 1: Initial prompt shown when opening XCode

3. In the following screen select **Single View Application**

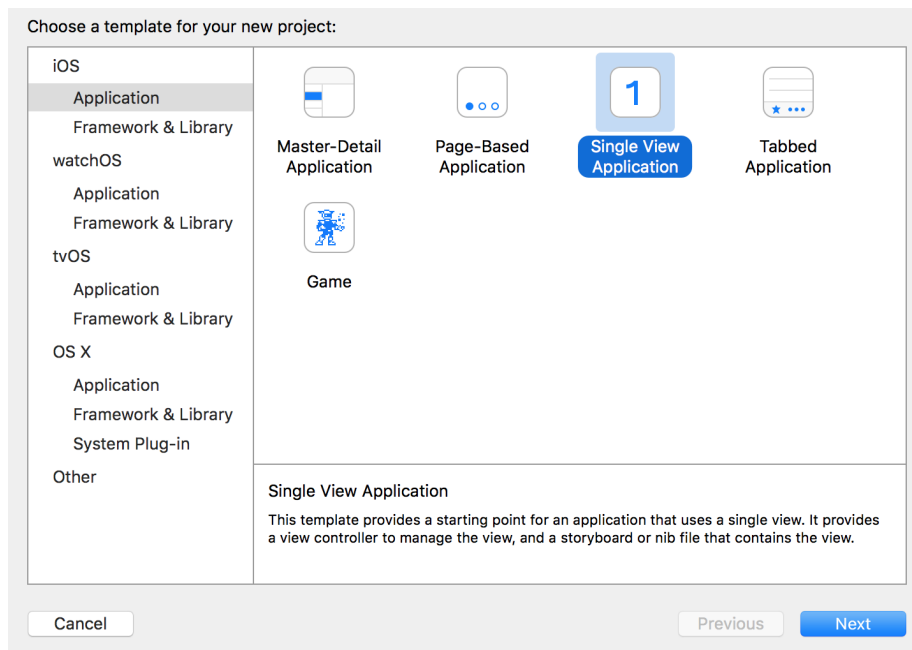


Figure 2: Selecting a project template

This template will create an iOS-App with a single empty screen.

4. Enter a Product Name for the project (In this case **Medbrain**)

Choose options for your new project:

Product Name:

Organization Name:

Organization Identifier:

Bundle Identifier:

Language:

Devices:

☐ Use Core Data

☒ Include Unit Tests

☒ Include UI Tests

Figure 3: Specifying basic project information

Note: the `Organization Identifier` and `Bundle Identifier` typically follow a reverse-DNS-format. The `Bundle Identifier` is used to uniquely identify your app (i.e.: in the iTunes AppStore) >By Default this is set by the following schema

`{Organization Identifier}`.

`{Product Name}` but can be set manually at a later point.

5. When prompted where to create the project select the previously created **working directory**.

1.3 Conclusion

You completed the basic setup for an iOS-App

2 Managing Dependencies with CocoaPods[1]

2.1 Prerequisites

Before continuing ensure CocoaPods is installed. To install CocoaPods run `sudo gem install cocoapods` in the terminal.

If you want to start here you just run `git checkout step2`.

The Project is located at `project/` of the git repository root.

2.2 Getting Started

CocoaPods is a tool for managing dependencies in iOS and Mac Applications.

A possible alternative is Apples Swift Package-Manager[3].

Since CocoaPods is already very established and commonly seen as Best Practice, and the Swift Package Manager is still in beta stage CocoaPods is preferred.

2.2.1 Adding CocoaPods to the project

Open a terminal and navigate to the **working directory**. Run the command `pod init`. This will create a file named Podfile.

2.2.2 Specifying the dependencies

Open the Podfile with any text-editor and change its contents to:

```
platform :ios, '8.0'
use_frameworks!

target 'Medbrain' do
  pod 'SMART', '~> 2.1'
end

target 'MedbrainTests' do
end

target 'MedbrainUITests' do
end
```

The pod 'SMART' installs the dependency **Swift-SMART**[4].

This is a library simplifying the usage of FHIR-Resources within Swift Projects.

2.2.3 Installing the dependencies

Now that it is specified which dependencies are to be used in the project they have to be installed.

run `pod install` in the Terminal.

This downloads all libraries specified in the Podfile and integrates them into the XCode project.

2.3 Conclusion

You learned how to setup CocoaPods in a iOS-project and install dependencies.

Next-up is a overview of the used FHIR-Resources

3 Overview and Explanation of used FHIR-Resources

3.1 Goals

- Get a understanding of the resources used in this app.
- Specify assumptions about the dataset.

3.2 Overview

Before starting to work with FHIR its important to understand what the resources represent and the relationships between them.

FHIR-Resources have very few required properties since it tries to support many healthcare-standards which may have different requirements.

It is up to the developer to refine the definition of a sufficiently specified resource for the context it is used in.

As an orientation the **ELGA Implementierungsleitfaden** was used.

The main resources used are:

- Patient 3.3.1
- Medication 3.3.2
- MedicationOrder 3.3.3
- DosageInstructions 3.3.4
- Timing 3.3.5
- MedicationAdministration 3.3.6

These will be explained more in depth in the corresponding section below.

3.3 Resources

3.3.1 Patient

The patient-resource has a central role in this app. It contains basic information about the person and is often referenced by other resources.

Note: a patient in FHIR is not necessarily human. but in the context of the app it is assumed that the patient is human.

UML Diagram

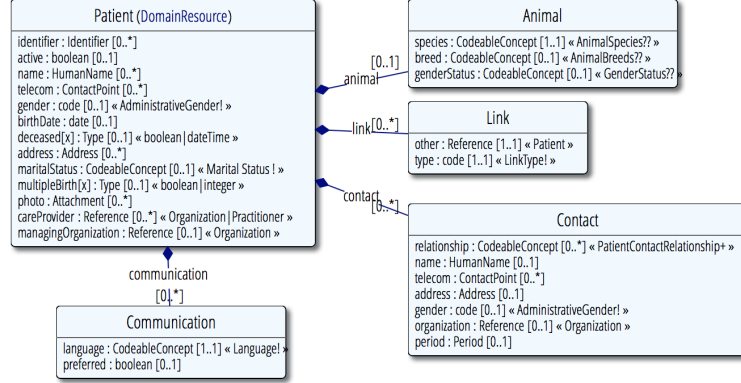


Figure 4: Structure of the Patient resource

Requirements There are no explicit requirements in order for the app to work. These requirements are strictly technical and additional constraints may make sense. It is assumed all information contained in the Patient resource is optional except for the id. The id must be present in order to search for medications which were prescribed for this specific patient.

These requirements can be formalized like:

requirement	expression	required
has id	<code>patient.id != null</code>	yes
is human	<code>patient.animal == null</code>	yes

Account, AllergyIntolerance, Appointment, AppointmentResponse, AuditEvent, Basic, BodySite, CarePlan, Claim, ClinicalImpression, Communication, CommunicationRequest, Composition, Condition, Contract, Coverage, DetectedIssue, Device, DeviceUseRequest, DeviceUseStatement, DiagnosticOrder, DiagnosticReport, DocumentManifest, DocumentReference, Encounter, EnrollmentRequest, EpisodeOfCare, FamilyMemberHistory, Flag, Goal, Group, ImagingObjectSelection, ImagingStudy, Immunization, ImmunizationRecommendation, List, Media, MedicationAdministration, MedicationDispense, MedicationOrder, MedicationStatement, NutritionOrder, Observation, Order, Person, Procedure, ProcedureRequest, Provenance, QuestionnaireResponse, ReferralRequest,

RelatedPerson , RiskAssessment , Schedule , Specimen ,
SupplyDelivery , SupplyRequest , VisionPrescription

3.3.2 Medication

Represents a medication.

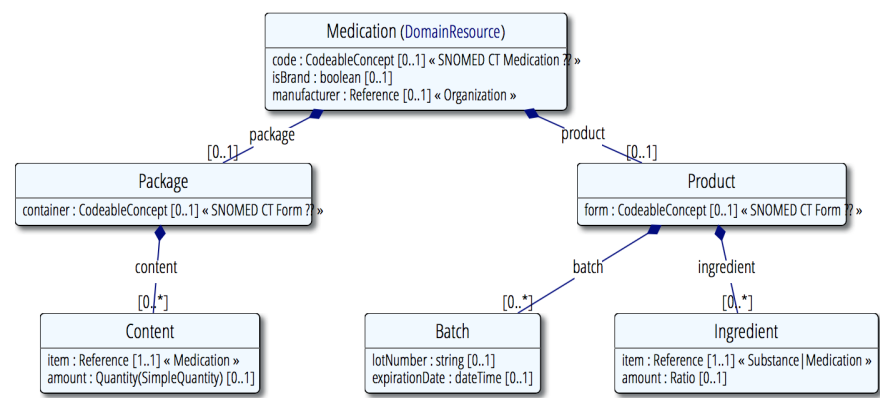


Figure 5: Structure of the Medication resource

Requirements In the context of the app only the medications name is displayed, therefore its only requirement is its name. This should not imply it describes a medication sufficiently.

requirement	expression	required
has display-name	<code>medication.code.coding[0].display != null</code>	yes

CarePlan , Group , MedicationAdministration ,
MedicationDispense , MedicationOrder ,
MedicationStatement , Procedure , SupplyDelivery ,
SupplyRequest

3.3.3 MedicationOrder

<https://www.hl7.org/fhir/medicationorder.html>
An order for supply and administration of the medication to a patient.

A `MedicationOrder` can only be created by a `Practitioner` and never by a `Patient`.

Therefore the app only requires read-only access to a patients medication-orders

requirement	expression	required
at least one dosageInstruction prescribed for a patient	<code>medicationOrder.dosageInstructions[50] != null</code> <code>medicationOrder.patient != null</code>	yes
has medication	<code>medicationOrder.medication != null</code>	yes

Requirements

This resource is referenced by:

CarePlan, Claim, ClinicalImpression, MedicationAdministration, MedicationDispense

3.3.4 DosageInstructions

Is a substructure of `MedicationOrder`. Contains information about timing and dosageInstructions

Note: `DosageInstruction` includes a text property which describes its content. This property is ignored and descriptions are generated from the structured data. This provides the possibility to localize the description (generate descriptions in different languages)

dosageInstruction	Σ	0..*	BackboneElement	How medication should be taken
text	Σ	0..1	string	Dosage instructions expressed as text
additionalInstructions	Σ	0..1	CodeableConcept	Supplemental instructions - e.g. "with meals"
timing	Σ	0..1	Timing	When medication should be administered
asNeeded[x]	Σ	0..1		Take "as needed" (for x)
asNeededBoolean			boolean	
asNeededCodeableConcept			CodeableConcept	
site[x]	Σ	0..1		Body site to administer to SNOMED CT Anatomical Structure for Administration Site Codes (Example)
siteCodeableConcept			CodeableConcept	
siteReference			Reference(BodySite)	
route	Σ	0..1	CodeableConcept	How drug should enter body SNOMED CT Route Codes (Example)
method	Σ	0..1	CodeableConcept	Technique for administering medication
dose[x]	Σ	0..1		Amount of medication per dose
doseRange			Range	
doseQuantity			SimpleQuantity	
rate[x]	Σ	0..1		Amount of medication per unit of time
rateRatio			Ratio	
rateRange			Range	
maxDosePerPeriod	Σ	0..1	Ratio	Upper limit on medication per unit of time

Figure 6: Structure of the DosageInstructions resource

3.3.5 Timing

Name	Flags	Card.	Type	Description & Constraints
Timing	Σ		Element	A timing schedule that specifies an event that may occur multiple times
event	Σ	0..*	dateTime	When the event occurs
repeat	Σ I	0..1	Element	When the event is to occur <i>Either frequency or when can exist, not both</i> <i>If there's a duration, there needs to be duration units</i> <i>If there's a period, there needs to be period units</i> <i>If there's a periodMax, there must be a period</i> <i>If there's a durationMax, there must be a duration</i> Length/Range of lengths, or (Start and/or end) limits
bounds[x]	Σ	0..1		
boundsQuantity			Duration	
boundsRange			Range	
boundsPeriod			Period	
count	Σ	0..1	integer	Number of times to repeat
duration	Σ I	0..1	decimal	How long when it happens <i>duration SHALL be a non-negative value</i>
durationMax	Σ	0..1	decimal	How long when it happens (Max)
durationUnits	Σ	0..1	code	s min h d wk mo a - unit of time (UCUM) UnitsOfTime (Required)
frequency	Σ	0..1	integer	Event occurs frequency times per period
frequencyMax	Σ	0..1	integer	Event occurs up to frequencyMax times per period
period	Σ I	0..1	decimal	Event occurs frequency times per period <i>period SHALL be a non-negative value</i>
periodMax	Σ	0..1	decimal	Upper limit of period (3-4 hours)
periodUnits	Σ	0..1	code	s min h d wk mo a - unit of time (UCUM) UnitsOfTime (Required)
when	Σ	0..1	code	Regular life events the event is tied to EventTiming (Required)
code	Σ	0..1	CodeableConcept	QD QOD Q4H Q6H BID TID QID AM PM + TimingAbbreviation (Preferred)

Figure 7: Structure of the Timing resource

3.3.6 MedicationAdministration

Represents a medications administration or lack thereof.

The patient creates a **MedicationAdministration** when he/she takes/doesnt take a **Medication**

the **MedicationAdministration** must reference a **MedicationOrder**

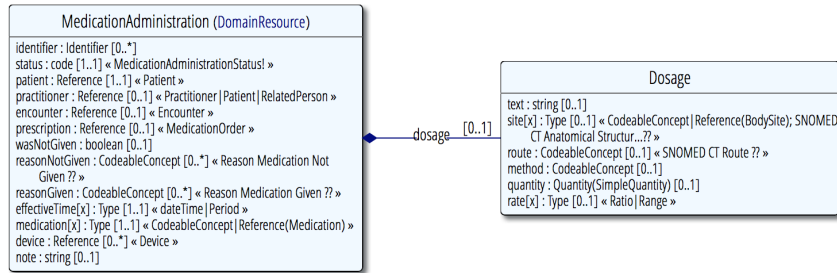


Figure 8: Structure of the MedicationAdministration resource

requirement	expression
references patient	<code>administration.patient != null</code>
references medication-order	<code>administration.prescription != null</code>
references medication	<code>administration.medication != null</code>
effectiveTime set	<code>patient.effectiveTime != null</code>

Requirements

4 Building the Application Structure

4.1 Prerequisites

Finished Step 2 of this tutorial described in section 2 of this paper. To start here run `git checkout step4`. The Project is located at `project/` of the git repository root.

4.2 Goals

In this step each of the apps screens will be added as a placeholder implementation. Each screens functionality will be implemented later on.

Learn about building User Interfaces utilizing X-Codes Interface-Builder and Storyboards.

4.3 Getting started

By default X-Code creates a storyboard name `Main.storyboard`. It contains a single `ViewController`. We start of by deleting the `ViewController` and its corresponding `ViewController` implementation.

1. Delete the file `ViewController.swift`
2. Select file `Main.storyboard` in the project navigator.
3. Select the empty `ViewController` and delete it.

4.4 Adding a `TabBarController`

1. From the Interface-Builders object-library drag a `Tab Bar Controller` object onto the storyboard.
2. Delete the two `ChildControllers`.
3. Select the newly added controller and in the `Attributes Inspector` select the checkbox `Is Initial ViewController`

4.5 Building the Navigation-flow

1. Select a `Navigation Controller` from the object-library and drag it onto the storyboard.

Note: This also adds an empty `Table View Controller` > >
The added `Table View Controller` will later show a list of the patients medications and will be referred to as `PatientMedicationsViewController`.

2. Drag a `Table View Controller` from the object-library onto the storyboard.

Note: This controller will show a single medication in more detail and will be referred to as `MedicationDetailViewController`.

3. `ctrl+drag` from the first `PatientMedicationsViewController` to the newly added `MedicationDetailViewController`
4. select `show` in the `Manual-Segue` section
5. `ctrl+drag` from the `Tab Bar Controller` to the `Navigation Controller`
6. select `viewcontrollers` in the `Relationship-Segue` section

4.6 Adding the SignInViewController

To be able to determine which medications to show the patient has to sign-in.

In order to sign-in a new **View Controller** named **PatientSignInViewController** will be introduced.

This controller will be shown every time the user reaches the **PatientMedicationsViewController** and is not signed-in.

1. From the **object-library** drag a **View Controller** object on to the storyboard.
2. **ctrl+drag** from the first **PatientMedicationsViewController** to the newly added **PatientSignInViewController**.
3. select **Present Modally** in the **Manual-Segue** section.

4.7 Adding the PatientDetailViewController

After signing in the user should have the option to review his information (i.e.: email-address, phone-number, etc.)

This functionality will be implemented in the **PatientDetailViewController**.

1. From the **object-library** drag a **View Controller** object on to the storyboard.
2. **ctrl+drag** from the **Tab Bar Controller** to the **PatientDetailViewController**
3. select **viewcontrollers** in the **Relationship-Segue** section

When you are finished the end result should look similar to this:

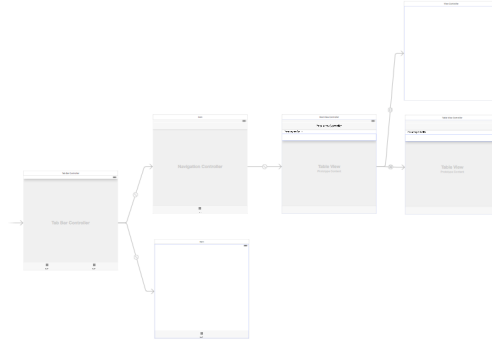


Figure 9: Finished Design in the Storyboard

4.8 Conclusion

You setup a storyboard showing the apps intended flow.

5 Implementing SignIn

5.1 Prerequisites

Finished Step 4 of this tutorial described in section 4 of this paper. To start here run `git checkout step5`. The Project is located at `project/` of the git repository root.

5.2 Goals

The data displayed in this app is very specific to the currently signed in patient.

So before building the rest of the app it is the first priority to implement the sign-in.

5.3 Getting started

For the implementation we will introduce following classes:

- **SessionManager**
- **PatientSignInViewController**

SessionManager

- Implemented as a singleton.
- Its responsibility is to manage the currently signed in patient.
- Handle log-in and log-out

PatientSignInViewController

- Displays a screen in which the user can enter his credentials. (i.e. username and password)
- Displays loading indicator while the log-in is in progress
- Displays a message if a error occurred

5.4 Implementing the SessionManager

5.4.1 Creating the SessionManager

1. Create a new file named `SessionManager.swift` (File->New->File)
2. When prompted for the template for the file select **Swift File**
3. Create a singleton class and import the **SMART** framework

```
import Foundation
import SMART

class SessionManager {
    ///singleton instance
    static let shared = SessionManager()

    private init() {
        ///private initializer to avoid this class being instantiated anywhere else than
    }

    ///The server against which requests are executed
}
```

```

var server = Server(base: "http://fhir2.healthintersections.com.au/open/")

///The currently logged in patient (defaults to nil)
var patient: Patient?

///convenience property is 'true' if a patient is signed in
var signedIn: Bool {
    return patient != nil
}
}

```

4. Define Notification posted when the patient changes

```

///MARK: - Definitions
extension SessionManager {

    ///Name of the notification sent to observers when the patient changes
    static let PatientChangedNotification = "SessionManager.PatientChangedNotification"

    ///User-info dictionary key for the old patient value
    static let PatientChangedNotificationOldKey = "SessionManager.PatientChangedNotificationOldKey"

    ///User-info dictionary key for the new patient value
    static let PatientChangedNotificationNewKey = "SessionManager.PatientChangedNotificationNewKey"

    ///When the patient changes notify all observers about this change
    func patientDidChange(old: Patient?, new: Patient?) {
        var info = [NSObject:AnyObject]()
        info[SessionManager.PatientChangedNotificationOldKey] = old
        info[SessionManager.PatientChangedNotificationNewKey] = new

        ///Notify obusersers that the current patient did change
        NotificationCenter.defaultCenter().postNotificationName(SessionManager.PatientChangedNotification, object: nil, userInfo: info)
    }
}

```

5.4.2 Creating the LogInCredentials

This object is passed to the SessionManger when initiating the login.

```

///credentials should provide enough data to uniquely identify a patient
struct LogInCredentials {

    ///credentials which identify a test_user on the server
    static var defaultCredentials: LogInCredentials {
        return LogInCredentials(queryParamters: ["_id": ["\"$exact\": \"f001\"]])
    }
}

```

```

    var queryParamters: [NSObject:AnyObject] = [:]
}

```

5.4.3 Modeling the result of the Log-In task

1. Create an accompanying enum Result. This object represents the result of a task which might fail.

```

//when performing a task which might fail the result of this task can be modelled like this
enum Result<ExpectedResultType> {
    //the task completed successfully and produced a result
    //if no actual result is produced use Void as ExpectedResultType
    case Success(_: ExpectedResultType)

    //the task failed and optionally provides an error which contains more information what
    case Error(_: ErrorType?)
}

```

//MARK: - Methods

```

extension SessionManager {

    //closure called after the log-in completes
    //Note: completing the log-in does NOT mean it was successful
    typealias LogInCompletionHandler = (result: Result<Patient>) -> Void

    ///attempts to log in the user with the specified credentials asynchronously
    ///calls completion handler with the result (success/failure)
    func login(credentials: LogInCredentials, completion: LogInCompletionHandler) {

        Patient.search(credentials.queryParamters).perform(server) {(bundle, error) in

            let result: Result<Patient>

            if let patients = bundle?.entry?.flatMap({ \$.resource as? Patient }) where !patients.isEmpty {
                if patients.count > 1 {
                    print("warning: multiple patients found. using first")
                }

                let oldPatient = self.patient
                self.patient = patients.first
                self.patientDidChange(oldPatient, new: self.patient)

                result = .Success(patients.first!)
            } else {
                result = .Error(error)
            }
        }
    }
}

```

```

        }

        completion(result: result)
    }
}

func logout() {
    let oldPatient = self.patient
    patient = nil
    patientDidChange(oldPatient, new: nil)
}
}

```

The finished `SessionManager` implementation can be found [here](#)

5.5 PatientSignInViewController Interface

1. create a new file named `PatientSignInViewController.swift` (File->New->File)
 - When prompted to choose a template for the file choose iOS->Cocoa Touch Class
 - in the next step choose `PatientSignInViewController` as class
 - make it a subclass of `UIViewController`
2. in the `Main.storyboard` we created the `PatientSignInViewController` but it is not linked to its implementation.
 - go to the `Main.storyboard` and select the `PatientSignInViewController`
 - in the `Identity-Inspector` under the `Custom-Class` enter `PatientSignInViewController`

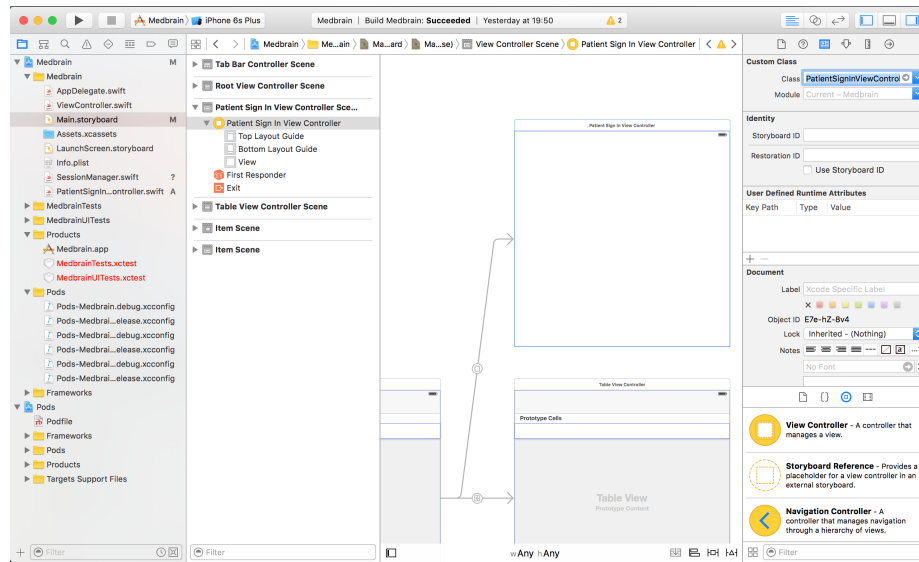


Figure 10: Setting the PatientSignInViewControllers class

5.5.1 Building the Interface

The `PatientSignInViewController` requires the following elements:

- TextField for entering e-mail
- TextField for entering password
- Button for submitting the entered information
- Activity indicator being displayed while the log-in is in process

For this purpose `outlets` have to be defined in the class-implementation.

Note: Outlets are a way of connecting elements in the Interface-Builder to the implementation. The creation and configuration could also be done programmatically.

5.5.2 Adding outlets to the class implementation

add the following to the `PatientSignInViewController` implementation:

```
@IBOutlet var emailTextfield: UITextField!
@IBOutlet var passwordTextfield: UITextField!
@IBOutlet var submitButton: UIButton!
@IBOutlet var activityIndicator: UIActivityIndicatorView!

@IBAction func submitButtonPressed(sender: AnyObject?) {
    //will be implemented later
}
```

5.5.3 Adding the UI-Elements in the storyboard

Note: In this tutorial the design will not be handled in depth. Instead the required UI-elements will be defined and shown how they need to be linked.

1. Drag the required elements from the object-library into the `PatientSignInViewController`. The end result should look like this:

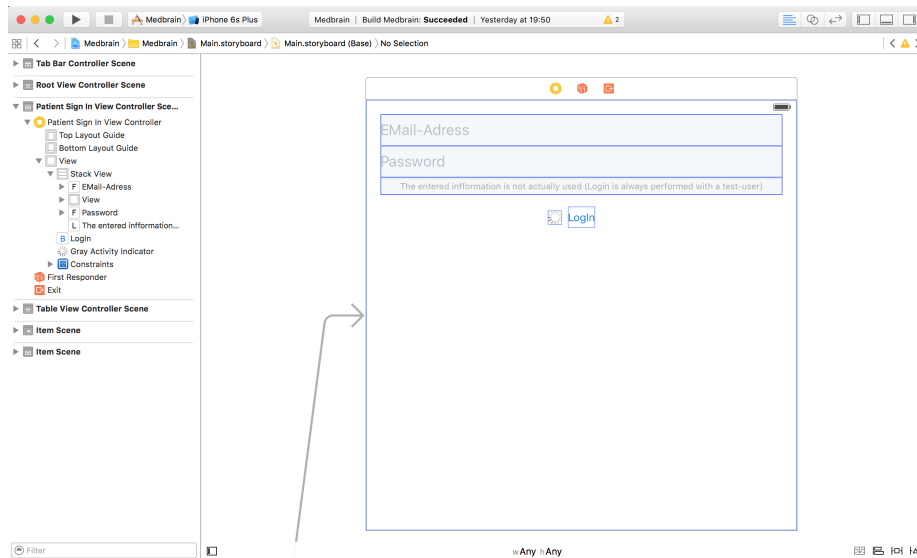


Figure 11: Completed design of the `PatientSignInViewController`

2. Connect the outlets and actions to the corresponding elements.

5.6 PatientSignInViewController functionality

Import SMART framework add `import SMART` statement to the top of the `PatientSignInViewController`

5.6.1 Adding a completionHandler

To inform the initiator of the Sign-In that the task was completed a `completionHandler` is added. This is a new property on the `PatientSignInViewController`.

Note: The `completionHandler` is implemented as a closure. A closure is a self-contained block of code to be executed which can be passed as a parameter.

First define a typealias for the `completionHandler`

```
typealias SignInCompletionHandler = (patient: Patient) -> Void
```

This will be called after the user has successfully signed in.

Add a optional property named `completionHandler` of type `SignInCompletionHandler`:

```
///Will be called after the Sign-In successfully finished
///Note: user cannot leave screen without successfully signing in
var completionHandler: SignInCompletionHandler?
```

5.6.2 Implement `submitButtonPressed` action

This method gets called when the `submitButton` is pressed and will perform 3 steps:

1. Generate `LogInCredentials` from the entered information

Note: In this implementation credentials which identify a test-user are always used.
2. Log-in using `SessionManager`
3. Verify result and if successful call `completionHandler` if failed show error

```
@IBAction func submitButtonPressed(sender: AnyObject?) {
    //1. generate 'LogInCredentials' from the entered information
    //2. log-in using 'SessionManager'
    //3. Verify result and if successful call 'completionHandler' if failed show error
}
```

1. Generating and verifying credentials

Two helper-methods are introduced

```
///generate 'LogInCredentials' from the information entered
///- returns: nil if the entered credentials are incomplete or invalid
func generateCredentials() -> LogInCredentials? {
    //when sign-in is implemented properly this has to be adjusted

    //Always return the default credentials for the test-user
    return LogInCredentials.defaultCredentials
}

///displays an alert with the specified message
func showError(message message: String) {
    let alertController = UIAlertController(title: message, message: nil, preferredStyle: .Alert)
    alertController.addAction(UIAlertAction(title: "Ok", style: .Default, handler: nil))

    presentViewController(alertController, animated: true, completion: nil)
}
```


In the `submitButtonPressed` method the above methods will be used

```
@IBAction func submitButtonPressed(sender: AnyObject?) {
    //verify that the entered information is valid
    guard let credentials = generateCredentials() else {
        showError(message: "Entered Information is incomplete or invalid")
        return
    }

    //2. log-in using 'SessionManager'
    //3. Verify result and if successful call 'completionHandler' if failed show error
}
```

2. Performing the log-in

```
@IBAction func submitButtonPressed(sender: AnyObject?) {
    //verify that the entered information is valid
    guard let credentials = generateCredentials() else {
        showError(message: "Entered Information is incomplete or invalid")
        return
    }

    //disable textFields so user cannot edit information while it is being submitted
    emailTextField.userInteractionEnabled = false
    passwordTextField.userInteractionEnabled = false
    submitButton.enabled = false

    //attempt sign-in using the verified credentials
    SessionManager.shared.logIn(credentials) { (result) in
        //3. Verify result and if successful call 'completionHandler' if failed show error
    }
}
```

3. Handling the result

```
@IBAction func submitButtonPressed(sender: AnyObject?) {
    //verify that the entered information is valid
    guard let credentials = generateCredentials() else {
        showError(message: "Entered Information is incomplete or invalid")
        return
    }

    //disable controls so user cannot edit information while it is being submitted
    emailTextField.userInteractionEnabled = false
    passwordTextField.userInteractionEnabled = false
    submitButton.enabled = false
    activityIndicator.startAnimating()
}
```

```

//attempt sign-in using the verified credentials
SessionManager.shared.logIn(credentials) { (result) in

    //reenable controls
    self.emailTextfield.userInteractionEnabled = true
    self.passwordTextfield.userInteractionEnabled = true
    self.submitButton.enabled = true
    self.activityIndicator.stopAnimating()

    //check result
    switch result {
    case .Success(let patient):
        //Was successful call completionHandler
        self.completionHandler?(patient: patient)
    case .Error(_):
        //Failed show Error Message
        self.showError(message: "Something went wrong")
    }
}
}

```

The finished PatientSignInViewController implementation can be found at `project/Medbrain/Medbrain/PatientSignInViewController.swift`

5.7 Conclusion

You built a Log-In Interface and functionality and implemented the Session-Manager.

6 Implementing PatientMedicationsViewController

6.1 Prerequisites

Finished Step 5 of this tutorial described in section 5 of this paper. To start here run `git checkout step6`. The Project is located at `project/` of the git repository root.

6.2 Goals

Build a View Controller which shows a list of prescriptions for a specific patient. Additionally if the user is not signed in the Log-in Screen should be shown.

6.3 Getting started

This Step is split up into 4 parts:

1. Creating the `PatientMedicationsViewController` interface
2. Ensuring the user is logged-in
3. Generating instructions for prescriptions
4. Loading and displaying all `MedicationOrders` for the logged-in user

6.4 PatientMedicationsViewController Interface

Creating the `PatientMedicationsViewController` is very similar to creating the `PatientSignInViewController` interface:

1. Create a `UITableViewController` subclass named `PatientMedicationsViewController`
2. In the `Main.storyboard` select the `PatientMedicationsViewController` and set its class to `PatientMedicationsViewController`

6.4.1 Creating the StatusView

This `View Controller` loads data from a remote source. While performing such a task its common practice to do this asynchronously. (While performing this task the user can still interact with the app.)

To inform the user that the app is fetching data it displays a `StatusView`.

The `StatusView` has responsibility for displaying:

- loading states
- error states
- empty states

6.4.2 Defining outlets in the implementation class

The `StatusView` consists of two elements:

- loading-indicator
- title-label

For these elements and the `StatusView` itself outlets are declared.

Add the following to the `PatientMedicationsViewController`:

```
@IBOutlet var statusView: UIView!
@IBOutlet weak var activityIndicator: UIActivityIndicatorView!
@IBOutlet weak var statusTitleLabel: UILabel!
```

drag a view object from the `object-library` onto the `PatientMedicationsViewController`

Again the implementation of the design will not be explained in depth. The important part is that all elements are added.

The end-result can look like this:

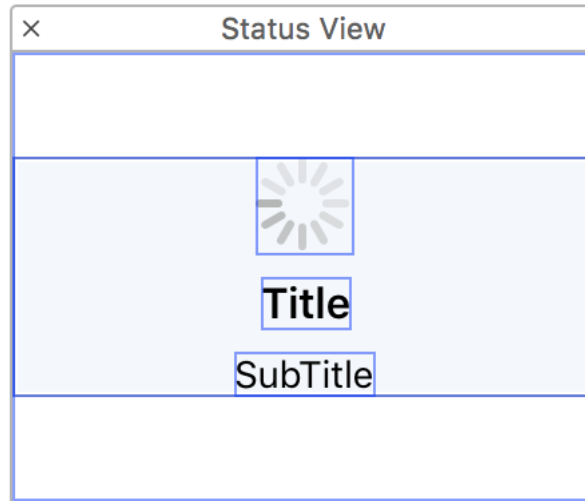


Figure 12: Finished design of the StatusView

Connect the storyboard-elements to the outlets.

To actually show the statusView this has to be set to be the tableViews `backgroundView`. In order to accomplish this the `viewDidLoad()` method is overridden. Once the `ViewControllers` view is created the `viewDidLoad()` method gets called. At this point all storyboard-elements have been linked up.

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    tableView.backgroundView = statusView  
}
```

To keep track of the state a `State` enum and a corresponding property are introduced:

```
enum State {  
    //Starting state show nothing  
    case Initial  
  
    //Fetching medications from the server
```

```

    case LoadingResults

    //Fetching medications failed
    case Error

    //Fetching medications succeeded but list was empty
    case Empty

    //Fetching medications succeeded
    case Loaded
}

```

```

var state: State = .Initial

```

Additionally 3 convenience accessors are added to the State enum:

```

///does the statusView need to be shown
var showsStatusView: Bool {
    switch self {
    case .Initial, .LoadingResults, .Error, .Empty:
        return true
    case .Loaded:
        return false
    }
}

///when the statusView is show should the loading-indicator be displayed
var showsLoadingIndicator: Bool {
    switch self {
    case .LoadingResults:
        return true
    case .Initial, .Error, .Empty, .Loaded:
        return false
    }
}

///The text displayed in the statusView
var title: String? {
    switch self {
    case .Initial:
        return nil
    case .LoadingResults:
        return "Loading..."
    case .Error:
        return "Error"
    case .Empty:

```

```

        return "No Prescriptions"
    case .Loaded:
        return nil
    }
}

```

The View Controller has to be able to display every state which is why a new method `configure(forState state: State)` is added:

```

func configure(forState state: State) {
    //Hide/show the statusView
    statusView.hidden = !state.showsStatusView

    //Hide show the titleLabel
    titleLabel.hidden = state.title?.isEmpty ?? true
    titleLabel.text = state.title ?? " "

    //Start/Stop Animating the activityIndicator
    if activityIndicator.isAnimating() && !state.showsLoadingIndicator {
        activityIndicator.stopAnimating()
    } else if !activityIndicator.isAnimating() && state.showsLoadingIndicator {
        activityIndicator.startAnimating()
    }
}

```

The above mentioned method should be executed every time the controllers state changes. This can be achieved by adding a **property observer** to the state property.

```

var state: State = .Initial {
    didSet {
        configure(forState: state)
    }
}

```

To initially configure the `statusView` the `viewDidLoad()` has to be modified as follows:

```

enum State {
    //Starting state show nothing
    case Initial

    //Fetching medications from the server
    case LoadingResults

    //Fetching medications failed
}

```

```

        case Error

        //Fetching medications succeeded but list was empty
        case Empty

        //Fetching medications succeeded
        case Loaded
    }

    var state: State = .Initial

```

6.5 Ensuring the user is logged in

When the user navigates to the `PatientMedicationsViewController` it is possible that the user hasn't logged in yet.

`UIViewController` (and its subclasses) have methods that are invoked by the framework for specific lifecycle events.

Of particular interest is the `viewDidAppear(animated: Bool)` method.

This method gets called after the View Controller has appeared.

Before overriding this method the `PatientMedicationsViewController` needs a way to show the `PatientSignInViewController`.

This is done via a `StoryboardSegue`.

In a previous step we set up a segue from `PatientMedicationsViewController` to the `PatientSignInViewController`. But to invoke this segue it has to be assigned an identifier.

- In the `Main.storyboard` select the `PatientMedicationsViewController` and in the `Attributes Inspector` assign `showSignIn` as identifier.

6.5.1 Showing the PatientSignInViewController

Add the following to the `PatientMedicationsViewController` implementation.

```

struct Period {
    let value: Double
    let valueMax: Double?
    let unit: TimingUnit

    init?(_ value: NSDecimalNumber?, valueMax: NSDecimalNumber?, unit: String?) {
        guard let duration = value?.doubleValue, unit = TimingUnit(unit) else {
            return nil
        }
        self.value = duration
        self.valueMax = valueMax?.doubleValue
        self.unit = unit
    }
}

```

```

    }
}

```

Now the `PatientSignInViewController` is shown but after the user signs-in successfully the `PatientSignInViewController` is never dismissed. For this purpose the `completionHandler` property on the `PatientSignInViewController` was introduced.

To set this property we need to acquire a reference to the `signInController`.

This can be achieved by overriding `prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)` which gets called before a segue is executed.

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {

    if segue.identifier == "showSignIn" {
        //Get a reference to the signInController
        let signInController = segue.destinationViewController as! PatientSignInViewController

        //set isSigninIn flag to true
        isSigninIn = true

        //Set the completionHandler
        signInController.completionHandler = { (patient) in
            //Dismiss the signInController and wait for the animation to finish
            self.dismissViewControllerAnimated(true) {
                self.isSigninIn = false

                //Now that the patient is signed in it the prescriptions can be loaded
                self.loadContent() //NOTE: This function will be implemented later
            }
        }
    }
}

```

6.6 Generating Instructions for Prescriptions

6.6.1 Overview

When displaying a list-item in the `PatientMedicationsViewController` as well as in the `MedicationDetailViewController` instructions should be shown.

These instructions should be generated from the data contained in the `MedicationOrder`

When generating these instructions its important to make as few assumptions about grammatical structures as possible as these may vary from language to language. To address this issue these descriptions are built using `Localized-Strings`. Strings are not used directly but are stored in a file (`Localizable.stringsdict`) and are referred to by keys. Multiple languages can be supported by providing a file for each desired language. For instance when describing a duration in seconds in english in the `Localizable.stringsdict` file a item is added:


```

<key>for %d second(s)</key>
<dict>
  <key>NSStringLocalizedFormatKey</key>
  <string>%#@value@</string>
  <key>value</key>
  <dict>
    <key>NSStringFormatSpecTypeKey</key>
    <string>NSStringPluralRuleType</string>
    <key>NSStringFormatValueTypeKey</key>
    <string>d</string>
    <key>one</key>
    <string>for one second</string>
    <key>other</key>
    <string>for %d seconds</string>
  </dict>
</dict>

```

In this example the key is "for %d seconds"

To allow the building of plural forms a `pluralization` rule can be defined. So for instance when describing one second the result is "for one second" as in all other cases it is "for x seconds"

To simplify the generation `DosageInstructions` some convenience classes/structs were introduced:

6.6.2 TimingUnit

String enum wrapping the units of time used by FHIR.

More information can be found [here](#).

```

enum TimingUnit: String {
  case Second = "s"
  case Minute = "min"
  case Hour = "h"
  case Day = "d"
  case Week = "wk"
  case Month = "mo"
}

```

6.6.3 EventTiming

String enum wrapping the codes for events used in FHIR

More information can be found [here](#)

```

enum EventTiming: String {
  init?(_ rawValue: String?) {
    guard let value = rawValue else { return nil }
    self.init(rawValue: value)
  }
}

```

```

}

/// before sleep
case BeforeSleep = "HS"

/// upon waking up
case Wake = "WAKE"

/// meal
case Meal = "C"

/// breakfast
case Breakfast = "CM"

/// lunch
case Lunch = "CD"

/// dinner
case Dinner = "CV"

/// before meal
case BeforeMeal = "AC"

/// before breakfast
case BeforeBreakFast = "ACM"

/// before lunch
case BeforeLunch = "ACD"

/// before dinner
case BeforeDinner = "ACV"

/// after meal
case AfterMeal = "PC"

/// after breakfast
case AfterBreakfast = "PCM"

/// after lunch
case AfterLunch = "PCD"

/// after dinner
case AfterDinner = "PCV"
}

```

6.6.4 Duration

Encapsulates a Duration. **Note:** The duration may also be defined as a range.
(valueMax != nil)

```
struct Duration {
  let value: Double
  let valueMax: Double?
  let unit: TimingUnit

  init?(_ value: NSDecimalNumber?, valueMax: NSDecimalNumber?, unit: String?) {
    guard let duration = value?.doubleValue, unit = TimingUnit(unit) else {
      return nil
    }
    self.value = duration
    self.valueMax = valueMax?.doubleValue
    self.unit = unit
  }
}
```

6.6.5 Frequency

Encapsulates a Frequency **Note:** The frequency may also be defined as a range.
(max != nil)

```
struct Frequency {
  let frequency: Int
  let max: Int?
}

extension Frequency {
  ///convenience initializer for creating frequency from a FHIR TimingRepeat
  init?(_ timing: TimingRepeat) {
    guard let frequency = timing.frequency else {
      return nil
    }
    self.frequency = frequency
    self.max = timing.frequencyMax
  }
}
```

6.6.6 Period

Encapsulates a period of time **Note:** The period may also be defined as a range.
(valueMax != nil)

```
struct Period {
  let value: Double
```

```

let valueMax: Double?
let unit: TimingUnit

init?(_ value: NSDecimalNumber?, valueMax: NSDecimalNumber?, unit: String?) {
    guard let duration = value?.doubleValue, unit = TimingUnit(unit) else {
        return nil
    }
    self.value = duration
    self.valueMax = valueMax?.doubleValue
    self.unit = unit
}
}

```

6.6.7 TimingBounds

Describes timing bounds used in FHIR.

This is modeled as an enum which wraps a value because `enums` are mutually exclusive. (Either a `Duration` or a `Period` but never both)

```

func numberOfSectionsInTableView(tableView: UITableView) -> Int

func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITa

```

6.6.8 Bringing it together

The generated instruction is a "sentence" where each of the items described above may contribute a part of that sentence.

Each valid "sentence" should be a key in the `Localizable.stringsdict` file.

To express this in code a protocol named `LocalizedSentenceBuildingSupport` was introduced. >**Note:** A protocol is the swift equivalent of an interface in java

elements which adopt this protocol provide a part of a sentence.

```

protocol LocalizedSentenceBuildingSupport {
    ///return all possibilities
    ///Note: this is not necessarily required for building the sentence but is useful for g
    static var allFormatStrings: [String] { get }

    var formatString: String { get }
    var localizedArguments: [CVarArgType] { get }
}

```

The sentence built by a `MedicationOrder` follows is generated of parts in the following order. **Note:** Each element is optional

1. Frequency
2. Period
3. EventTiming
4. Duration
5. TimingBounds

Since there are many possible combinations a **Swift-Playground** generating all required keys was implemented.

Note: A **Swift-Playground** is a file where specific code snippets can be tested easily in order to see if they are implemented correctly.

This playground can be found at:

`project/Medbrain/Medbrain/MedicationOrderInstructions.playground`

Since the generated instructions will be used in various places in the app it does not make sense to implement this in the `PatientMedicationsViewController`. As a reusable solution the instructions will be exposed as a computed property on the `MedicationOrder` class. Additionally a convenience accessor for the `medicationName` will be implemented.

1. Create a new file named `MedicationOrder+Instructions.swift`

Note: It is Swift Convention when adding a extension to a existing class/struct to place the implementation in a File named `ExtendedClass+ExtensionName.swift`

2. Implement the extension like this:

```
import SMART

extension SMART.MedicationOrder {

    var localizedInstructions: String {
        //No instructions to generate
        guard let repeat_fhir = dosageInstruction?.first?.timing?.repeat_fhir else {
            return "no instructions"
        }

        let optionalSentenceParts: [LocalizedSentenceBuildingSupport?] = [
            Frequency(repeat_fhir),
            Period(repeat_fhir.period, valueMax: repeat_fhir.periodMax, unit: repeat_fhir.periodUnit),
            EventTiming(repeat_fhir.when),
            Duration(repeat_fhir.duration, valueMax: repeat_fhir.durationMax, unit: repeat_fhir.durationUnit),
            TimingBounds(repeat_fhir)
        ]
    }
}
```

```

        //eliminiate all items which are nil
        let sentenceParts = optionalSentenceParts.flatMap { \ $0 }

        //builder string for sentence
        var sentence = "timing"
        //arguments to use
        var arguments = [CVarArgType]()

        //combine all sentence parts and arguments
        sentenceParts.forEach {
            sentence += \ $0.formatString
            arguments.appendContentsOf(\ $0.localizedArguments)
        }

        //Lookup string in Localizable.stringsdict and return result
        return String(format: NSLocalizedString(sentence, comment: ""), locale: NSLocal

    }

    ///The name of the prescribed medication
    var medicationName: String {
        if let medname = medicationCodeableConcept?.coding?.first?.display {
            return medname
        }

        if let display = medicationReference?.display {
            return display
        }

        return "No medicationname"
    }
}

```

6.7 Loading and displaying all prescriptions for the logged-in user

The ViewController needs to display a list of MedicationOrders

Therefore the View Controller needs to store the list of medications somewhere.

A new property called medicationOrders is introduced. Each time the medicationOrders are changed the tableView needs to be reloaded.

```

///medicationOrders to display in the list
var medicationOrders: [MedicationOrder] = [] {
    didSet {
        tableView.reloadData()
    }
}

```

```

    }
}

```

6.7.1 Creating the MedicationOrderTableViewCell

1. Create a new Cocoa Touch Class named `MedicationOrderTableViewCell` and make it a subclass of `UITableViewCell`
2. In the `Main.storyboard` select the `PatientMedicationsViewController`
Note: By default in a `UITableViewController` a single table-view-cell prototype is created.
3. Select the first cell
4. In the `Identity Inspector` change its class to `MedicationTableViewCell`
5. In the `Attributes Inspector` change its style to `Subtitle`
6. In the `Attributes Inspector` assign it the identifier `MedicationOrderTableViewCell`
Note: This identifier will be used later to instantiate a cell of this type

6.7.2 Configuring the MedicationOrderTableViewCell

The `MedicationOrderTableViewCell` represents a single item in the list.

Each cell is configured for a single `MedicationOrder`. In the `MedicationOrderTableViewCell` implementation import the `SMART` framework and add the following method

```

func configure(medicationOrder: MedicationOrder) {
    self.textLabel?.text = medicationOrder.medicationName
    self.detailTextLabel?.text = medicationOrder.localizedInstructions
}

```

6.7.3 Preparing the display of MedicationOrderTableViewCell

`PatientMedicationsViewController` is a subclass of `UITableViewController` the `UITableViewController` conforms to the `UITableViewDataSource` protocol.

The `tableView` requires the `dataSource` to implement the following methods:

```

@IBOutlet var titleLabel: UILabel!
@IBOutlet var subtitleLabel: UILabel!
@IBOutlet var segmentedControl: UISegmentedControl!
@IBOutlet var activityIndicator: UIActivityIndicatorView!
@IBOutlet var chartView: ORKLineGraphChartView!

@IBAction func segmentedControlChanged(sender: AnyObject?) {
}

```

These methods should be implemented as follows:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    //the tableView should display a single section
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
    //in the section the number of items should be the number of medicationOrders
    return medicationOrders.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    //dequeue cell with the identifier specified in the storyboard
    let cell = tableView.dequeueReusableCellWithIdentifier("MedicationOrderTableViewCell", forIndexPath: indexPath) as! MedicationOrderTableViewCell

    //find the medicationOrder at the index
    let medicationOrder = medicationOrders[indexPath.row]

    //configure the cell
    cell.configure(medicationOrder)

    return cell
}
```

Now the tableView is able to display the medicationOrders.

6.7.4 Loading the results

When the sign-in finished the method loadContent() was called. Now this function will be implemented.

```
func loadContent() {
    //transition to the loading state
    state = .LoadingResults

    //ensure a patient is logged in and it has an id
    guard let patient = SessionManager.shared.patient, patientId = patient.id else {
        //transition to the error state
        state = .Error
        return
    }

    //perform a query search all MedicationOrder prescribed for the logged in patient
    MedicationOrder.search(["patient": patientId]).perform(SessionManager.shared
        .server) { [weak self](bundle, error) in
```



```

dispatch_async(dispatch_get_main_queue()) {

    //self was captured weakly
    //if the ViewController was deallocated while the search is in progress don't
    //doing so would crash the app
    guard let strongSelf = self else {
        return
    }

    //if a error occurred reset the medicationOrders and transition to the error
    guard error == nil else {
        strongSelf.medicationOrders = []
        strongSelf.state = .Error
        return
    }

    //extract all medicationOrders from the result
    let meds = bundle?.entry?.flatMap { \$.resource as? MedicationOrder } ?? []

    //if no medicationOrders where found transition to the empty state
    strongSelf.state = meds.isEmpty ? .Empty : .Loaded

    strongSelf.medicationOrders = meds
}
}
}

```

7 Implementing the MedicationDetailViewController

7.1 Prerequisites

Finished Step 6 of this tutorial described in section 6 of this paper. To start here run `git checkout step7`. The Project is located at `project/` of the git repository root.

7.2 Getting Started

When selecting a item in the `PatientMedicationsViewController` a View should be shown showing further details about the medication.

This view will be implemented in the `MedicationDetailViewController`. The view should show a graph with administrations for the specified medication. Additionally a list of all administrations should be displayed.

In order to display the graph a new dependency will be introduced.

`ResearchKit` contains classes which can be used to render graphs:

7.2.1 Adding the ResearchKit dependency

ResearchKit[2] is an open source framework introduced by Apple that allows to create apps for medical research.

Edit your Podfile as follows:

```
CarePlan, Claim, ClinicalImpression, MedicationAdministration, MedicationDispense
```

This Step will be split up into 3 parts

1. Building the interface
2. Loading result and displaying a list of all administrations
3. Rendering a chart visualizing the timeline of the administrations

7.3 MedicationDetailViewController Interface

7.3.1 Create the implementation class

Create a new UITableViewController subclass named MedicationDetailViewController. In the Main.storyboard change the MedicationDetailViewController class to MedicationDetailViewController.

7.3.2 Adding the HeaderView

Add a tableViewHeader to the table view by dragging a View object from the object library to the top edge of the tableView.

This headerView will contain:

- **TitleLabel** displaying the name of the medication.
- **SubtitleLabel** displaying the instructions for the medicationOrder
- **GraphView** displaying a timeline of the administrations
- **SegmentedControl** allowing to customize the grouping of administrations (by year, month, week or day)
- **ActivityIndicator** indicating progress while the administrations are loaded from the server

7.3.3 Defining the outlets and actions

First import ResearchKit

```
@IBOutlet var titleLabel: UILabel!  
@IBOutlet var subtitleLabel: UILabel!  
@IBOutlet var segmentedControl: UISegmentedControl!  
@IBOutlet var activityIndicator: UIActivityIndicatorView!  
@IBOutlet var chartView: ORKLineGraphChartView!
```

```
@IBAction func segmentedControlChanged(sender: AnyObject?) {
}
}
```

Again the design will not be explained in depth. Add the according elements and connect the outlets. For the `segmentedControl` add the `segmentedControlChanged` action for the control-event `ValueChanged`

To add the `chartView`:

- Drag a `View` object from the `object library`.
- Select the view and in the `Identity Inspector` set its class to `ORKLineGraphChartView`

The end-result can look like this:

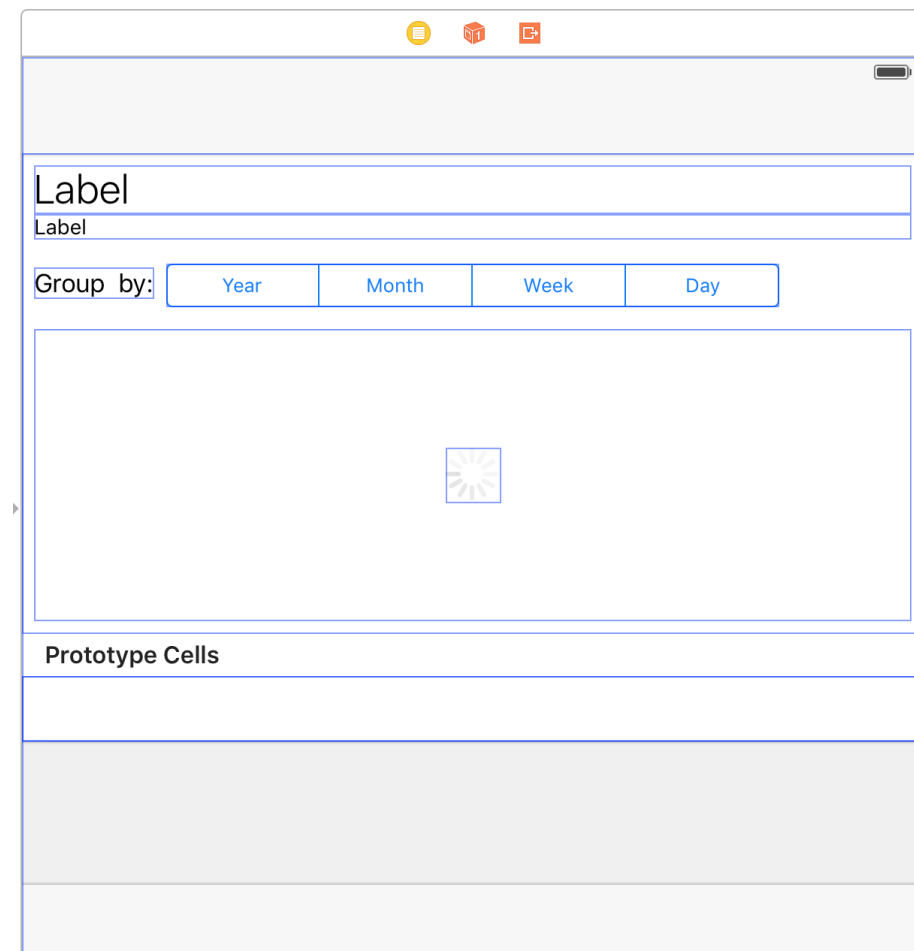


Figure 13: Finished design of `tableHeaderView`

7.3.4 Create the MedicationAdministrationTableViewCell

1. Create a new UITableViewCell subclass named MedicationAdministrationTableViewCell.
2. In the Storyboard set its class to MedicationAdministrationTableViewCell
3. In the Identity Inspector identifier to MedicationAdministrationTableViewCell
4. In the Attributes Inspector change its style to Subtitle

7.4 Loading and displaying all administrations for a prescription

7.4.1 Defining the properties

The administrations in this View are loaded for a specific MedicationOrder

Add a new property called `medicationOrder`

```
///medicationOrder for which details are shown  
///NOTE: - This property has to be set before 'viewDidLoad()' is called  
var medicationOrder: MedicationOrder!
```

As before the results are loaded asynchronously.

The previously declared `State` enum as well as the `configure(forState state: State)` method will be added.

Add the `state` property and a initially empty implementation of `configure(forState state: State)`

```
var state: State = .Initial {  
    didSet {  
        configure(forState: state)  
    }  
}  
  
func configure(forState state: State) {  
    //will be implemented later  
}
```

In this controller a list of MedicationAdministrations will be shown.

Add a property named `administrations`

```
var administrations: [MedicationAdministration] = [] {  
    didSet {  
        tableView.reloadData()  
    }  
}
```

7.4.2 Implementing UITableViewDataSource methods

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    //the tableView should display a single section
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
    //in the section the number of items should be the number of medicationOrders
    return administrations.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    //dequeue cell with the identifier specified in the storyboard
    let cell = tableView.dequeueReusableCellWithIdentifier("MedicationAdministrationTableViewCell")

    //find the medicationOrder at the index
    let medicationAdministration = administrations[indexPath.row]

    //Configuring the cell will be implemented later

    return cell
}
```

7.4.3 Configuring the MedicationAdministrationTableViewCell

In the MedicationAdministrationTableViewCell implementation import the SMART framework add a method called configure(administration: MedicationAdministration) with the following contents.

```
static let dateFormatter: NSDateFormatter = {
    let formatter = NSDateFormatter()
    formatter.dateStyle = .LongStyle

    return formatter
}()

func configure(administration: MedicationAdministration) {
    //set transparent color to the contentView
    contentView.backgroundColor = UIColor.clearColor()

    //try to get the date when the medication was administered / not administered
    if let date = administration.effectiveTimeDateTime?.date?.nsDate ?? administration.effectiveTime
        detailTextLabel?.text = MedicationAdministrationTableViewCell.dateFormatter.stringFromDate(date)
    } else {
        detailTextLabel?.text = "-"
    }
}
```

```

        //if the medication was not taken set red backgroundColor
        if let wasNotGiven = administration.wasNotGiven where wasNotGiven {
            titleLabel?.text = "not taken"

            backgroundColor = UIColor.redColor().colorWithAlphaComponent(0.2)
        } else {
            titleLabel?.text = "taken"
            backgroundColor = UIColor.whiteColor()
        }
    }
}

```

Now that the `configure(administration: MedicationAdministration)` method is implemented it needs to be called.

Navigate to the `MedicationDetailViewController` and add the call to configure the cell.

```

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) {
    //dequeue cell with the identifier specified in the storyboard
    let cell = tableView.dequeueReusableCellWithIdentifier("MedicationAdministrationTableViewCell")

    //find the medicationOrder at the index
    let medicationAdministration = administrations[indexPath.row]

    cell.configure(medicationAdministration)

    return cell
}

```

7.4.4 Implement loading of administrations

A user can only reach this screen if he has signed in already. So the content can be loaded immediately when the screen has appeared.

Next the `loadContent()` method will be implemented and called from `viewDidAppear(animated: Bool)`:

```

override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)
    loadContent()
}

func loadContent() {
    //ensure the medicationOrder has an id (this should always be the case)
    guard let id = medicationOrder.id else {
        state = .Error
        return
    }
}

```

```

    }

    //transition to the loading state
    state = .LoadingResults

    //Perform a search of MedicationAdministrations for the medicationOrder
    MedicationAdministration.search(["prescription": id]).perform(SessionManager.shared.serv

        dispatch_async(dispatch_get_main_queue()) {
            guard let strongSelf = self else {
                return
            }

            //if a error occurred reset the administrations and transition to the error state
            guard error == nil else {
                strongSelf.administrations = []
                strongSelf.state = .Error
                return
            }

            let administrations = bundle?.entry?.flatMap { $0.resource as? MedicationAdmini

            strongSelf.administrations = administrations
            strongSelf.state = administrations.isEmpty ? .Empty : .Loaded

        }
    }
}

```

7.5 Rendering a chart visualizing a timeline of administrations

In a previous step See:7.3.2 the UI-elements were added in the storyboard and connected to the corresponding outlets. But the view was never configured for the corresponding content/state.

7.5.1 Configuring the view for state

Start of by implementing the `configure(forState state: State)` method.

```

func configure(forState state: State) {
    //only show the chart if data has loaded successfully
    chartView.hidden = state != .Loaded
}

```

```

        //start/stop the activityIndicator if needed
        if state.showsLoadingIndicator && !activityIndicator.isAnimating() {
            activityIndicator.startAnimating()
        } else if !state.showsLoadingIndicator && activityIndicator.isAnimating() {
            activityIndicator.stopAnimating()
        }
    }
}

```

7.5.2 Configuring the view for the prescription

The view is correctly showing a list of all administrations for a given prescription. But the `MedicationDetailViewController` should also show for which Medication data is shown. A new function named `configure(forMedicationOrder medicationOrder: MedicationOrder)` is implemented. This function uses the previously declared properties defined in `MedicationOrder+Instructions.swift`.

```

extension MedicationDetailViewController: ORKGraphChartViewDataSource {

    ///Display two lines one for the medications which were taken, one for the medications which were not taken
    func numberOfPlotsInGraphChartView(graphChartView: ORKGraphChartView) -> Int {
        return 2
    }

    func graphChartView(graphChartView: ORKGraphChartView, numberOfPointsForPlotIndex plotIndex: Int) -> Int {
        return graphData.content.count
    }

    ///The line representing medications which were not taken should be red
    ///The other line should use the views tint color
    func graphChartView(graphChartView: ORKGraphChartView, colorForPlotIndex plotIndex: Int) -> UIColor {
        if plotIndex == 1 {
            return view.tintColor
        } else {
            return UIColor.redColor()
        }
    }

    func graphChartView(graphChartView: ORKGraphChartView, pointForPointIndex pointIndex: Int) -> ORKRangedPoint {
        if plotIndex == 1 {
            return ORKRangedPoint(value: CGFloat(graphData.content[pointIndex].countTaken))
        } else {
            return ORKRangedPoint(value: CGFloat(graphData.content[pointIndex].countNotTaken))
        }
    }
}

```



```
    }
}
```

Additionally some setup has to be performed in the `MedicationDetailViewController`'s `viewDidLoad()` method.

```
override func viewDidLoad() {
    super.viewDidLoad()

    //'MedicationDetailViewController' will provide chart data
    chartView.dataSource = self

    configure(forState: state)
    configure(forMedicationOrder: medicationOrder)
}
```

7.5.3 Implementing the SegmentedControl functionality

In the storyboard a `SegmentedControl` was added which allows the user to modify how the chart renders its data. The user can select to group medications by:

- Year
- Month
- Week
- Day

To express this in code a new `enum` named `DateGroupingMode` is introduced.

```
enum DateGroupingMode: Int {
    case Year = 0
    case Month = 1
    case Week = 2
    case Day = 3
}
```

Note: This enum was specified as a `Int` enum this is helpful when used in combination with `SegmentedControl` which exposes its current selection via the `selectedSegmentIndex` property. The `selectedSegmentIndex` corresponds to the `DateGroupingMode` `rawValue`.

Add a property named `groupingMode` to the `MedicationDetailViewController`. Each time this property changes the charts data needs to be updated.

Additionally every time the `administrations` change the `chartData` has to be updated as well.

```

var groupingMode: DateGroupingMode = .Month {
    didSet {
        updateChartData()
    }
}

var administrations: [MedicationAdministration] = [] {
    didSet {
        tableView.reloadData()
        updateChartData()
    }
}

func updateChartData() {
    //will be implemented later
}

```

Add the following to the `segmentedControlChanged(sender: AnyObject?)` implementation:

```

@IBAction func segmentedControlChanged(sender: AnyObject?) {
    groupingMode = DateGroupingMode(rawValue: segmentedControl.selectedSegmentIndex)!
}

```

7.6 Implementing the Chart

The charts data is dependent on the `MedicationDetailViewController`'s `administrations` and `groupingMode` and has to perform some processing to group the administrations. Two helper-functions which extend `NSDate` are introduced.

The first helper-function generates a `identifier` from a `NSDate` object using a specified `DateGroupingMode`. Each administration generates a identifier and all administrations with the same identifier fall into the same group.

Consider the following example:

- Administration1 was administered on Monday-15.08.2016
- Administration2 was administered on Tuesday-16.08.2016
- Administration3 was administered on Friday-20.02.2015

The following identifiers will be generated for these administration-dates:

	Group by			
	Year	Month	Week	Day
Administration1	2016	2016-08	2016-34	2016-08-15
Administration2	2016	2016-08	2016-34	2016-08-16
Administration3	2016	2016-02	2016-08	2016-02-20

Table 5: Table showing generated identifiers

When grouping by `Year` all administration dates generate the same identifier and are therefore in the same group. But when grouping by `Month` only `Administration1` and `Administration2` fall into the same group.

7.6.1 Generating identifiers

Create a new File named `NSDate+DateGrouping.swift`. In it add the following implementation:

```
extension NSDate {
    func identifier(groupingMode mode: DateGroupingMode) -> String {

        //specify which calendar-units to use
        let units: NSCalendarUnit

        switch mode {
        case .Day:
            units = [ .Year, .Month, .Day ]
        case .Week:
            units = [ .Year, .WeekOfYear ]
        case .Month:
            units = [ .Year, .Month]
        case .Year:
            units = [ .Year]
        }

        let calendar = NSCalendar.currentCalendar()

        let components = calendar.components(units, fromDate: self)

        switch mode {
        case .Day:
            return String(format: "%04d-%02d-%02d", components.year, components.month, components.day)
        case .Week:
            return String(format: "%04d-%02d", components.year, components.weekOfYear)
        case .Month:
            return String(format: "%04d-%02d", components.year, components.month)
        case .Year:
            return String(format: "%04d", components.year)
        }
    }
}
```

A second convenience function called `predecessor(groupingMode mode: DateGroupingMode)-NSDate` is added. This method returns a date which lies in the group directly before the current dates group.

```

///returns: - a date which lies in the group directly before this dates group
func predecessor(groupingMode mode: DateGroupingMode)->NSDate {
    let calendar = NSCalendar.currentCalendar()

    let components = NSDateComponents()

    switch mode {
    case .Day:
        components.day = -1
    case .Week:
        components.weekOfYear = -1
    case .Month:
        components.month = -1
    case .Year:
        components.year = -1
    }

    return calendar.dateByAddingComponents(components, toDate: self, options: NSCalendarOptions)
}

```

7.6.2 Creating the Charts Data-Model

The charts Data-Model will be created in a new struct called GraphData

```

struct GraphData {

    ///Represents a group in the graph
    struct GroupData {
        let identifier: String

        ///count of medications which where administered
        let countTaken: Int
        ///count of medications which were not administered
        let countNotTaken: Int
    }

    ///each item represents a group the values represent the count of administrations
    let content: [GroupData]

    init(_ administrations: [MedicationAdministration], groupingMode mode: DateGroupingMode) {
        var takenIdentifierToCounts = [String:Int]()
        var missedIdentifierToCounts = [String:Int]()

        ///Start with the current date
        var current = NSDate()
    }
}

```

```

var identifiers = [String]()

for _ in 0...maxGroups {
    let identifier = current.identifier(groupingMode: mode)

    identifiers.append(identifier)

    current = current.predecessor(groupingMode: mode)
}

for administration in administrations {
    guard let date = administration.effectiveTimeDateTime?.date.nsDate ?? administration.date.nsDate ?? Date()

    let identifier = date.identifier(groupingMode: mode)

    if !(administration.wasNotGiven ?? false) {

        takenIdentifierToCounts[identifier] = (takenIdentifierToCounts[identifier] ?? 0) + 1
    } else {

        missedIdentifierToCounts[identifier] = (missedIdentifierToCounts[identifier] ?? 0) + 1
    }

}

content = identifiers.map { GroupData(identifier: "\$0", countTaken: takenIdentifierToCounts[identifier] ?? 0, countMissed: missedIdentifierToCounts[identifier] ?? 0) }
}

```

Add a property called `graphData` to the `MedicationDetailViewController`. Each time the `graphData` changes the `chartView` should be reloaded.

```

var graphData: GraphData = GraphData([], groupingMode: DateGroupingMode.Day) {
    didSet {
        //WORKAROUND: - ORKLineGraphChartView has no explicit reloadData() method. Setting dataSource manually
        chartView.dataSource = self

        //animated the change
        chartView.animateWithDuration(0.2)
    }
}

```

7.6.3 Updating the Chart

Above the `updateChartData()` method was defined which gets called every time the `administrations` or the `groupingMode` changes. In this method the `graphData` should be recomputed.

```
func updateChartData() {
    graphData = GraphData(administrations, groupingMode: groupingMode)
}
```

7.6.4 Implementing the ORKGraphChartViewDataSource protocol methods

The `chartView` expects the data to be rendered to be provided by a object conforming to the `ORKGraphChartViewDataSource`.

The `MedicationDetailViewController` will implement these methods in a extension:

```
extension MedicationDetailViewController: ORKGraphChartViewDataSource {

    //Display two lines one for the medications which were taken, one for the medications u
    func numberOfPlotsInGraphChartView(graphChartView: ORKGraphChartView) -> Int {
        return 2
    }

    func graphChartView(graphChartView: ORKGraphChartView, numberOfPointsForPlotIndex plotIndex: Int) -> ORKGraphPoint {
        return graphData.content.count
    }

    //The line representing medications which were not taken should be red
    //The other line should use the views tint color
    func graphChartView(graphChartView: ORKGraphChartView, colorForPlotIndex plotIndex: Int) -> UIColor {
        if plotIndex == 1 {
            return view.tintColor
        } else {
            return UIColor.redColor()
        }
    }

    func graphChartView(graphChartView: ORKGraphChartView, pointForPointIndex pointIndex: Int) -> ORKGraphPoint {
        if plotIndex == 1 {
            return ORKRangedPoint(value: CGFloat(graphData.content[pointIndex].countTaken))
        } else {
            return ORKRangedPoint(value: CGFloat(graphData.content[pointIndex].countNotTaken))
        }
    }
}
```

```
    }
}
```

Now the `MedicationDetailViewController` is able to fully display the administrations for the `medicationOrder` that is set.

But the link from the `PatientMedicationsViewController` is not yet established.

7.7 Connecting MedicationDetailViewController and PatientMedicationsViewController

1. Navigate to the `Main.storyboard`
2. Select the Segue going from the `PatientMedicationsViewController` to the `MedicationDetailViewController`.
3. In the `Attributes Inspector` assign it the identifier `showMedication`.

Go to the `PatientMedicationsViewController`.

The `MedicationDetailViewController` should be shown when the user selects a item in the list. The item that should be shown is the one the user tapped on. `UITableView` can be assigned a `delegate` which must conform to the `UITableViewDelegate` protocol. The delegate can be used to modify certain aspects of how the `tableView` displays its content. (for instance: the height of cells). Additionally the delegate is informed when a cell is selected. To receive these notifications the `func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath)` method must be implemented.

When a cell is selected the previously added segue with identifier `showMedication` is executed.

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    performSegueWithIdentifier("showMedication", sender: nil)
}
```

Before performing the Segue the `prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)` function is invoked. In this function the selected `MedicationOrder` is passed to the `MedicationDetailViewController`

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showSignIn" {
        let signInController = segue.destinationViewController as! PatientSignInViewController
        signInController.isSignInIn = true
    }
}
```

```

        signInController.completionHandler = { (patient) in
            self.dismissViewControllerAnimated(true) {
                self.isSignInIn = false

                self.loadContent()
            }
        }
    } else if segue.identifier == "showMedication" {
        let detailController = segue.destinationViewController as! MedicationDetailView

        let selectedMedication = medicationOrders[tableView.indexPathForSelectedRow!.row]

        detailController.medicationOrder = selectedMedication
    }
}

```

7.8 Conclusion

- added a new dependency (ResearchKit)
- implemented a list showing all MedicationAdministrations
- Loaded all MedicationAdministrations for a specific prescription from a remote server
- implemented a graph visualizing the timeline of administrations

8 Implementing the CreateMedicationAdministrationViewController

8.1 Prerequisites

Finished Step 7 of this tutorial described in section ?? of this paper. To start here run `git checkout step8`. The Project is located at `project/` of the git repository root.

8.2 Goals

To allow the user to track when he/she takes a medication a new View Controller named `CreateMedicationAdministrationViewController` will be implemented.

This Step is split up into two parts:

- Building the Interface
- Creating the MedicationAdministration on the server

8.3 Building the Interface

The `CreateMedicationAdministrationViewController` displays a fairly simple interface:

The UI-Elements needed are:

- **Switch** allowing the user to specify whether the medication was administered or not administered
- **DatePicker** allowing the user to specify when the medication was administered/not administered
- **CancelButton** allowing the user to cancel the creation of the administration
- **SaveButton** allowing the user to finally “save” (create the resource on the server) the `MedicationAdministration`

The `CreateMedicationAdministrationViewController` is only reachable from the `MedicationDetailViewController`.

It creates a `MedicationAdministration` for the `MedicationOrder` shown in the `MedicationDetailViewController`

8.4 Defining the outlets and actions

1. Create a `UITableViewController` subclass named `CreateMedicationAdministrationViewController`.
2. In it import the SMART framework by adding `import SMART`
3. Add the following outlets and actions:

```
@IBOutlet weak var switchControl: UISwitch!
@IBOutlet weak var datePicker: UIDatePicker!
@IBOutlet var saveBarButtonItem: UIBarButtonItem!
@IBOutlet var cancelButtonItem: UIBarButtonItem!

@IBAction func cancelPressed(sender: AnyObject?) {
    //will be implemented later
}

@IBAction func saveAdministration(sender: AnyObject?) {
    //will be implemented later
}
```

Setting up the ViewController

1. In the `Main.storyboard` drag a new View Controller object on to the storyboard.

2. Select it and change its class to `CreateMedicationAdministrationViewController`
3. under `Editor->Embed In` select `Navigation Controller`
4. Select the `MedicationDetailViewController` and add a `BarButtonItem`
5. Select the `BarButtonItem` and in the `Attributes Inspector` for `System Item` set its value to `Add`
6. `ctr+drag` from the `BarButtonItem` to the `Navigation Controller` in which the `CreateMedicationAdministrationViewController` is embedded.
7. In the `Action Segue` `Segue Section` select `Present Modally`.
8. Select the `Segue` and in the `Attributes Inspector` assign it the identifier `createAdministration`

Creating the interface

1. Add a `BarButtonItem` to the right side of the `CreateMedicationAdministrationViewController` and set its `System Item` value to `Save`
2. Connect the `BarButtonItem` to the `saveBarButtonItem` outlet.
3. Connect the `BarButtonItem` to the `func saveAdministration(sender: AnyObject?)` action.
4. Add a `BarButtonItem` to the left side of the `CreateMedicationAdministrationViewController` and set its `System Item` value to `Cancel`
5. Connect the `BarButtonItem` to the `cancelBarButtonItem` outlet.
6. Connect the `BarButtonItem` to the `func cancelPressed(sender: AnyObject?)` action.

The rest of the design is not handled in depth. At least add a `Switch` and a `DatePicker` object and connect them to the corresponding outlets.

Cancel

Create Admin

Was Taken:

Date/Time taken:

Date Pick

58

The end-result can look like this.

8.5 Creating MedicationAdministration on the server

Some setup is needed to be able to create the resource on the server.

1. The `MedicationAdministration` is created for a specific `medicationOrder` so create a property `medicationOrder`.

```
var medicationOrder: MedicationOrder!
```

2. Add a `completionHandler` property which is called when the task is finished.

Note: Since the user can cancel the creation of the resource finishing the task can mean it was cancelled. For this purpose the `completionHandler` contains a `Bool` flag named `cancelled`

```
 typealias CreateAdministrationCompletionHandler = (cancelled: Bool)->Void
```

```
 var completionHandler :CreateAdministrationCompletionHandler?
```

3. In the `MedicationDetailViewController` implement the `prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)` method as follows:

```
 override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "createAdministration" {
        let createController = (segue.destinationViewController as! UINavigationController)

        //pass the medicationOrder to the 'CreateMedicationAdministrationViewController'
        createController.medicationOrder = medicationOrder

        //Dismiss the CreateMedicationAdministrationViewController after it is finished
        //Reload the administrations if the creation was successful (not cancelled)
        createController.completionHandler = { (cancelled) in
            self.dismissViewControllerAnimated(true, completion: {
                if !cancelled {
                    self.loadContent()
                }
            })
        }
    }
}
```

8.5.1 Implementing the actions

When the user presses the `cancelBarButtonItem` the `cancelPressed(sender: AnyObject?)` method is invoked. Implement the method as follows:

```

@IBAction func cancelPressed(sender: AnyObject?) {
    //call the completion handler with cancelled=true
    completionHandler?(cancelled: true)
}

```

When the `saveBarButtonItem` is pressed to persist the configured `MedicationAdministration` the `func saveAdministration(sender: AnyObject?)` function should perform the following tasks: - Disable the editable views as well as the `save` and `cancel` buttons. - Create a instance of `MedicationAdministration` using the data entered before. - Perform the Server-Request to create the resource. - If successful call the `completionHandler` - If an error occurred display a error message and reenale all previously disabled elements.

Note: The SMART framework provides no way to create resources other then initializing them with a `FHIRJSON` instance. `FHIRJSON` represents a JSON-Object and is defined as `public typealias FHIRJSON = [String: AnyObject]`

to create a `MedicationAdministration` a convenience initializer for it is introduced in an extension of `MedicationAdministration`.

```

extension MedicationAdministration {
    convenience init(medicationOrder: MedicationOrder, patient: Patient, wasTaken: Bool, time: Date) {
        var json = FHIRJSON()
        //has to have a status
        json["status"] = "completed"
        //create a reference to the medicationOrder
        json["prescription"] = ["reference": "\(MedicationOrder.resourceName)/\(medicationOrder.id)"]
        //reuse the medicationOrders reference to the medication
        json["medicationReference"] = medicationOrder.medicationReference?.asJSON()
        json["patient"] = patient.asJSON()
        json["wasNotGiven"] = !wasTaken
        json["effectiveTimePeriod"] = [
            "end": time.fhir_asDateTime().asJSON(),
            "start": time.fhir_asDateTime().asJSON()
        ]

        //use designated initializer with the json
        self.init(json: json)
    }
}

```

Now that a `MedicationAdministration` can be created locally it has to be created on the server.

Implement the `func saveAdministration(sender: AnyObject?)` function like this:

```

@IBAction func saveAdministration(sender: AnyObject?) {

    //create administration locally
    let administration = MedicationAdministration(medicationOrder: medicationOrder,
                                                    patient: SessionManager.shared.patient!,
                                                    wasTaken: switchControl.on,
                                                    time: datePicker.date)

    //Disable controls
    switchControl.userInteractionEnabled = false
    datePicker.userInteractionEnabled = false
    saveBarButtonItem.enabled = false
    cancelBarButtonItem.enabled = false

    //Attempt to create resource on server
    administration.create(SessionManager.shared.server) { [weak self] (error) in
        dispatch_async(dispatch_get_main_queue()) {

            guard let strongSelf = self else { return }

            if error != nil {
                //Reenable controls
                strongSelf.switchControl.userInteractionEnabled = true
                strongSelf.datePicker.userInteractionEnabled = true
                strongSelf.saveBarButtonItem.enabled = true
                strongSelf.cancelBarButtonItem.enabled = true

                //Show error message
                let alertController = UIAlertController(title: "Something went wrong", message: nil, preferredStyle: .Alert)
                alertController.addAction(UIAlertAction(title: "Ok", style: .Default, handler: nil))

                strongSelf.presentViewController(alertController, animated: true, completion: nil)
            } else {
                strongSelf.completionHandler?(cancelled: false)
            }
        }
    }
}

```

References

- [1] CocoaPods. <https://cocoapods.org>.
- [2] Research Kit. <https://www.hl7.org/fhir/2015May/index.html>.

- [3] Swift Package Manager. <https://github.com/apple/swift-package-manager>.
- [4] Swift-SMART. <https://github.com/smart-on-fhir/Swift-SMART>.