IOS HEALTHKIT APP. – DESIGNDOKUMENT

INHALTSVERZEICHNIS

ÜBERBLICK

HealthkitApp ist eine *iOS* Applikation, die einem Endbenutzer oder einem Experten ermöglicht die Sensorwerte, die von einem *Arduino* oder *Raspbery* Framework durch einen Server zur Verfügung gestellt werden zu visualisieren bzw. das aktuelle Stresslevel des Probanden zu bestimmen und anzuzeigen. Die Sensorwerte werden durch ein mit *Arduino* und *Raspbery* Gruppen vereinbartes Protokoll übermittelt.

Das Grunddesign der Applikation ist basiert auf dem MVC Architekturmuster. Das Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Das MVC Architekturmuster wird auch gut durch die Standardbibliothek von *iOS* unterstützt.

MODELL (MODEL)

Das Modell der Applikation ist zuständig für die Abfrage der Daten aus dem Server, ihre Speicherung und Aktualisierung. Das Modell bietet eine wohldefinierte Schnittstelle um die Daten aus den Controllern und Views abzufragen. Das Modell kapselt auch die Logik der Berechnung des Stresslevels.

PRÄSENTATION (VIEW)

Die Präsentationsschicht der Applikation hauptsächlich in dem *XCode* Storyboard definiert. Das Storyboard definiert die visuellen Komponenten jedes einzelnen Screens, ihr Layout und Eigenschaften.

STEUERUNG (CONTROLLER)

Für jeden Screen des Storyboards wird eine Controller-Klasse implementiert. Der Controller ist zuständig für das Laden der Daten in die Views und ihre Aktualisierung. Außerdem definieren die Controllers Interaktion mit dem Benutzer und die Navigation vom Controller zu Controller.

INTERFACE DER HEALTHKIT IOS APPLIKATION

Das Ziel war eine benutzerfreundliche UI mit einem ansprechenden Design zu entwickeln. Der Benutzer sollte nicht mit vielen Daten und Einstellungen überfordert sein. Die Anzeige jedes Screens muss auf das Wichtigste reduziert werden und immer leicht bedienbar bleiben.

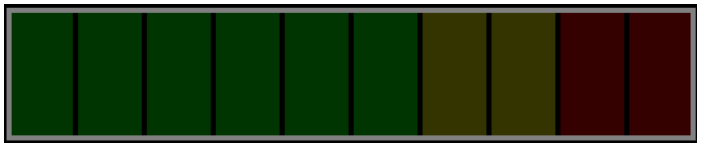
ANZEIGE DER DATEN

Bei der Entwicklung dieser Applikation haben wir uns nicht auf die Standard *iOS* Komponenten beschränkt, sondern für jede Anzeige einen geeigneten Komponenten ausgesucht. Im Folgendem sind einige dieser Komponenten aufgelistet:

- eine zifferblattartiger View für die Geschwindigkeitsanzeige. Siehe Abbildung unten:

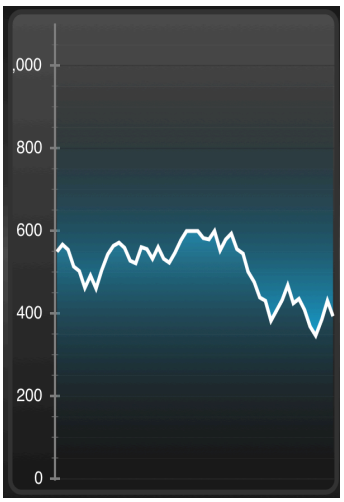
ZIFFERBLATTARTIGER VIEW

- ein *Barview* um das Stresslevel anzuzeigen:



BARVIEW

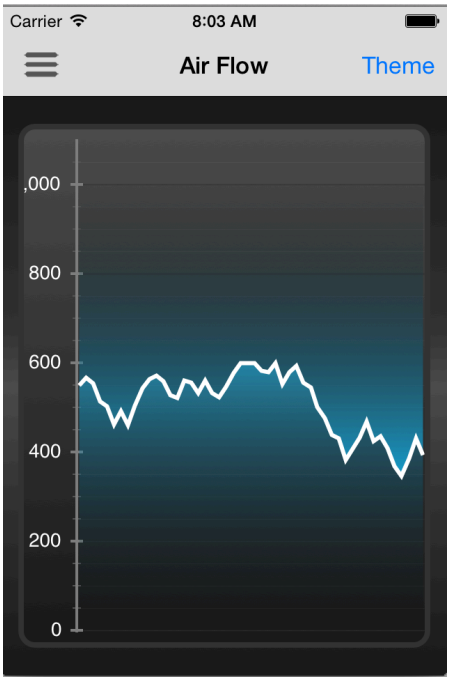
- die *Coreplot* Diagramme für die Anzeige von Verlauf der Sensorwerte:

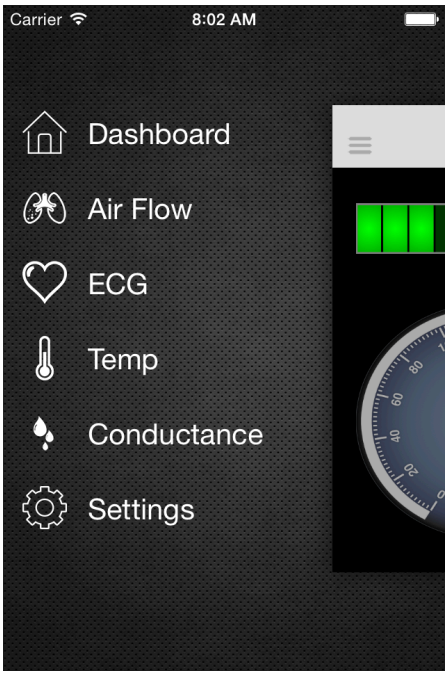


EINSATZ DER COREPLOT BEI DEN CHARTS

NAVIGATION

Für die Navigation haben wir eine spezielle Seitenmenü-Komponente verwendet. Das Seitenmenü wird normallerweise komplett versteckt und wird durch ein Klicken auf einer unauffälliger Taste in der oberen linken Ecke des Screens geöffnet. Dann wird das Menü auf dem Großteil des Screens angezeigt und nach der Auswahl eines Menüpunkts wieder versteckt.





Durch diese Navigationskomponente erreichen wir nicht nur ein ansprechendes UI-Design, sondern auch eine bessere Bedienbarkeit. Das Menü lässt sich gut verstecken, und so werden die Screens nicht mit unnötiger Information belastet. Nicht destotrotz wird bei der Menüauswahl fast der ganze Screen verwendet, was die Auswahl besonders während der Fahrt erleichtert.

Für das Menü haben wir geeignete Icons entworfen und einen passenden Background gewählt.

STARTSCREEN

Das Startscreen wird angezeigt wenn die Applikation gestartet wird. Hier werden dem Benutzer insbesondere der aktuelle Stresslevelwert und die aktuelle Geschwindigkeit des Autos vorgezeigt.

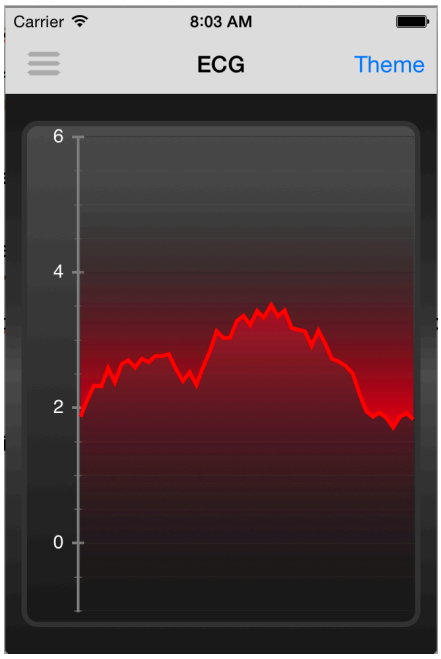
STARTSCREEN

Die Anzeigen des Stresslevels und der Geschwindigkeit werden in regelmäßigen Zeitintervallen durch einen *Timer* aktualisiert. Bei jeder Aktualisierung werden die aktuelle Werte vom Modell abgefragt.

ANZEIGE DER SENSORWERTE

Die Screens für die Anzeige verschiedenen Sensorwerten sind durch die Menü erreichbar. In jedem Screen wir der historische Verlauf eines bestimmten Sensorwerts angezeigt. In der aktuellen Implementierung unterstützen wir die Anzeige von *ECG*, *Airflow*, Temperatur und Hautleitfähigkeit.

Hier ist zum Beispiel das Screen für die Anzeige der *ECG*:



ECG SCREEN

Das Wertebereich der Y-Achse im Chart ist fest und entspricht dem Wertebereich des ECG-Sensors. Die X-Achse entspricht einem Zeitintervall von 10s in der Vergangenheit bis zur aktuellen Zeit. Deswegen muss der Wertebereich der X-Achse ständig aktualisiert werden. Die X-Achse haben wir versteckt da sie dem Benutzer keine nützliche Information liefert.

Das Chart wird durch eine Linie und zwei Gradient-Füllungen – oberhalb und unterhalb der Linie – dargestellt. Die Farben der Linie und der Füllungen sind an das angezeigte Wert abgestimmt:

```
// Setup line style for the plot
CPTMutableLineStyle *plotLineStyle = [plot.dataLineStyle mutableCopy];
plotLineStyle.lineWidth = 3;
plotLineStyle.lineColor = [self getLineColor];
plot.dataLineStyle = plotLineStyle;

// Setup gradient fill for the plot
CPTColor *areaColor      = [self getAreaColorBottom];


CPTGradient *areaGradient = [CPTGradient gradientWithBeginningColor:areaColor
                endingColor:[CPTColor clearColor]];
areaGradient.angle = -90.0;
CPTFill *areaGradientFill = [CPTFill fillWithGradient:areaGradient];
plot.areaFill      = areaGradientFill;
plot.areaBaseValue = [self getAreaBaseValue];

areaColor = [self getAreaColorTop];
```

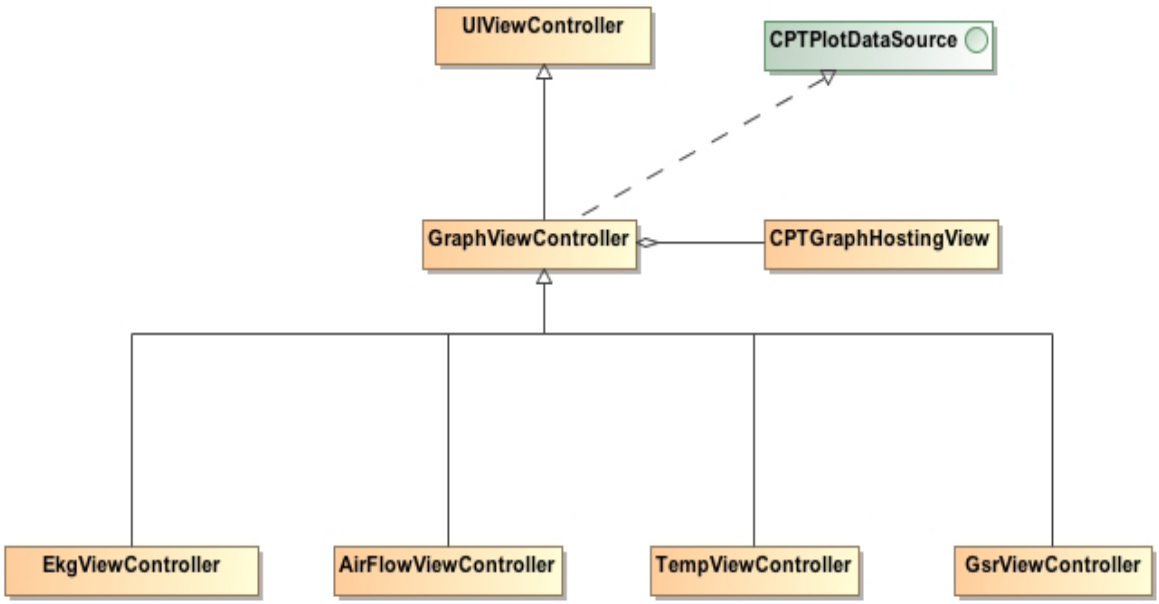
```
areaGradient = [CPTGradient gradientWithBeginningColor:[CPTColor clearColor]
                endingColor:areaColor];
areaGradient.angle  = -90.0;
areaGradientFill = [CPTFill fillWithGradient:areaGradient];
plot.areaFill2 = areaGradientFill;
plot.areaBaseValue2 = [self getAreaBaseValue2];
```

Ähnlich wie im Startscreen werden die Charts durch einen *Timer* in regelmäßigen Zeitintervallen aktualisiert. Dabei wir dass Modell nach den Sensorwerten in dem angezeigten Zeitintervall abgefragt.

Der Code der Controllern für Anzeige der Charts variiert an bei bestimmten Stellen:

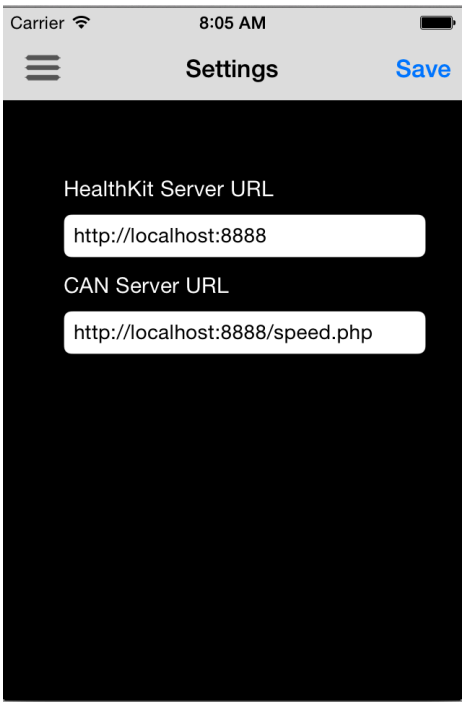
- Die Anbindung an das Datenmodell
- Die Farben für die Chartlinien und Gradientfüllungen
- Wertebereich der Y-Achse

Um den sich wiederholenden Code wiederzuverwenden haben wir eine gemeinsame Superklasse *GraphViewController* definiert, und für den variierenden Code abstrakte Methoden definiert, die in Unterklassen überschrieben werden.



SETTINGS

Der Settings-Screen ist über das Seitenmenü erreichbar und dient zur Einstellung des URL-Adressen der Webservern für *HeathKit* und *CAN*.



SETTING SCREEN

Die Settings werden zuerst mit *Defaultwerten* initialisiert. Nach der Änderung der Settings wird das Datenmodell mit den neuen Server neu konfiguriert und die neue Werte werden auf dem Gerät gespeichert, so dass wenn man die Applikation neu startet werden die Settings wieder geladen.

```
- (IBAction)saveSettings:(id)sender {
   if([self.ipInput.text rangeOfString:@"http://"].location ==   NSNotFound){
      UIAlertView *wrongUrl = [[UIAlertView alloc] initWithTitle:@"Wrong Url"
         message:@"The ip adress should begin with http://" delegate:self
               cancelButtonTitle:@"ok" otherButtonTitles:nil];
      [wrongUrl show];
   } else {
      self.loadingView.hidden = NO;
      [[NSUserDefaults standardUserDefaults]    setValue:self.ipInput.text
               forKey:@"healthKitURL"];
      [[NSUserDefaults standardUserDefaults] setValue:self.canUrl.text forKey:@"speedUrl"];

      [[NSUserDefaults standardUserDefaults] synchronize];
      [UIView animateWithDuration:0.5 delay:0 options:UIViewAnimationCurveEaseOut
               animations:^{
         self.loadingView.alpha = 0;
```

```
        } completion:nil];
        [dataModel loadSettings];
        [canDataModel loadSettings];
    }
}
```

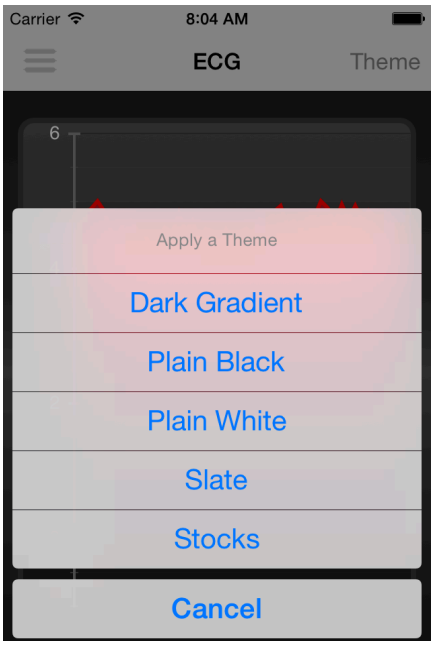
UNTERSTÜTZUNG FÜR ROTATION UND VERSCHIEDENE AUFLÖSUNGEN

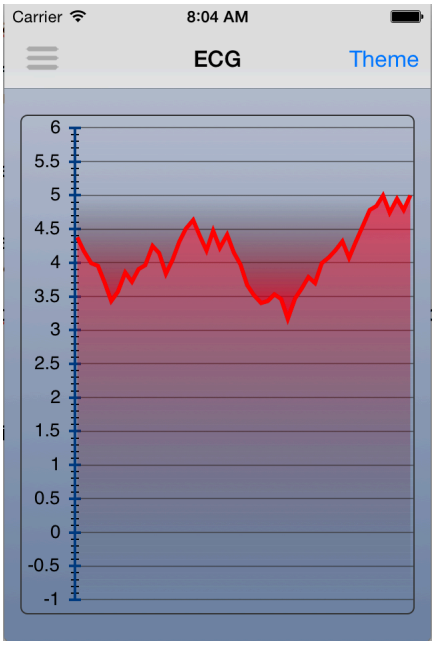
Um *iOS* Geräte mit unterschiedlichen Auflösungen und die Rotation des Geräts zu unterstützen haben wir statt einer fester Positionierung der UI-Elementen dynamische *Constraints* verwendet.

Zum Beispiel das Diagramm im ECG-Screen wird relativ zu den Rändern des umschließenden Views positioniert.

UNTERSTÜZUNG VERSCHIEDENER THEMEN FÜR DIAGRAMME

Der Benutzer hat im jedem Screen die Möglichkeit den aktuell angezeigten Diagramm in gewünschtem Style anzuzeigen lassen. Durch das Themenmenu rechts oben kann man von einer Liste von vordefinierten Stylthemen wählen.





DATENMODELL DER HEALTHKIT APPLIKATION

Datenmodel besteht aus zwei Singleton Objekten, die in *AppDelegate.h* definiert sind. Das erste Singletonobjekt *dataModel* stellt die Sensorwerten und das andere Objekt *canDataModel* stellt den Geschwindigkeitswert für die anderen Klassen bereit.

Die Datenmodell für Sensorwerte holt Daten aus dem Server, speichert sie ab, fügt Zeitstempel hinzu. Die alte Daten werden aufgeräumt um Speicher freizugeben. Es bietet

eine Schnittstelle um Daten im bestimmten Zeitintervall abzufragen, z.B. in letzten 10s. Es beinhaltet die Berechnung vom aktuellen Stresswert.

Wir nutzen unabhängige *Timer* für *Requests* von Server und für Aktualisierung von Views. So machen die beide Prozessen unabhängig von einander. Man kann sie auch mit unterschiedlichen Zeitintervallen takten. Z.B. wenn Verbindung schlechter ist kann man Requests seltener schicken, aber die Views trotzdem häufig aktualisieren um eine fliessende Animation zu erreichen.

Der *Timer* für Kommunikation mit Server läuft kontinuierlich um auch die historische Daten zu sammeln die nicht gerade angezeigt werden. Der *Timer* für Aktualisierung von einem View ist aktiv nur während das View sichtbar ist.

Serverrequest werden asynchron bearbeitet, d.h. man registriert einen *Delegate* der aufgerufen wird, wenn der Response vom Server kommt. Von verschiedenen Kommunikationsmöglichkeiten mit denen wir experimentiert haben, hat diese als stabilste sich erwiesen:

```
NSMutableURLRequest* urlRequest = [[NSMutableURLRequest alloc] init];
[urlRequest setURL: [NSURL URLWithString: self.urlToUse]];
[urlRequest setHTTPMethod: @"GET"];
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection sendAsynchronousRequest:urlRequest queue:queue
            completionHandler: ^(NSURLResponse *response, NSData *responseData,
            NSError *connError)
{
   if ([responseData length] > 0 && connError == nil) {
      NSError* jsonError;
      NSDictionary* json = [NSJSONSerialization JSONObjectWithData:responseData
           options:kNilOptions error:& jsonError];
      if (!json) {
         NSLog(@"Error parsing JSON: %@", jsonError);
      } else {

         // Data was successfully received and parsed
         NSDictionary* sensorsDict = [json valueForKey:@"Sensors"];
         NSMutableDictionary* sensorsNew = [sensorsDict mutableCopy];
         id time = [NSNumber numberWithDouble:[self getTimeInSec]];
         [sensorsNew setValue:time forKey:@"time"];
         [self.allValues addObject: sensorsNew];
       }
   }
   else if (connError!= nil) {
      NSLog(@"Error: %@", connError);
   }
}];
```

STRESSBERECHNUNG

Das ursprüngliche Ziel war Stress als eine Kombination von Herzschlagfrequenz, Atemfrequenz und Hautleitfähigkeit zu berechnen.

Die Sensorwerte die im Stressberechnung verwendet werden, werden zuerst normalisiert, d.h. durch ihren Schwankungsbereich (MAX-MIN) geteilt. So bekommen wir alle Werte im Bereich zwischen 0..1. Dann nehmen wir ihren Mittelwert.

NormHF = (HF – MinHF)= / (MaxHF – MinHF)

NormAF = (AF – MinAF) / (MaxAF – MinAF)

NormHL = (HL – MinAF) / (MaxHL - MinHL)

StressLevel = (NormHF + NormAF + NormHL) / 3

Durch Normalisierung erreichen wir, dass alle Komponenten gleich gewichtet sind, unabhängig von ihrem Wertebereich. Im Folgendem sieht man die Formel zur Berechnung des Stresslevels.

Leider die Daten die wir für den Client bekommen sind nicht genau genug um daraus zuverlässig Herzschlagfrequenz und Atemfrequenz herzuleiten. Außerdem wurde erst spät die Netzwerkkommunikation stabilisiert und so echte Daten ins Gerät bekommen. Deswegen haben wir für Stresslevelberechnung Hautleitfähigkeit und Temperatur verwendet (auch wenn es nicht die geeignetste Werte sind).

VERWENDETE EXTERNEN BIBLIOTHEKEN:

Insgesamt haben wir im Projekt vier Custom Bibliotheken verwendet.

1. *WMGaugeView* zur Visualisierung des Geschwindigkeitswert im Dashboard Scene **https://github.com/Will-tm/WMGaugeView**
2. *F3BarGauge* zur Visualisierung des Stresslevel im Dashboard Scene. **https://github.com/ChiefPilot/F3BarGauge**
3. *RESideMenu* wurde für die Seiten Menüleiste verwendet https://github.com/romaonthego/RESideMenu
4. *CorePlot* von Google um Charts zu zeichnen https://code.google.com/p/core-plot/

Wir wollten kein Rad neu erfinden, sondern von open Source verfügbare Komponenten im Netz das beste aussuchen und für unsere Bedürfnisse anpassen. Diese Komponenten haben wir so konfiguriert dass sie an einander stilistisch gut abgestimmt sind, mit passenden Bilder kombiniert. Das Ziel war dass trotz verschiedenen Komponenten das Design einheitlich aussieht.

WMGAUGEVIEW

WMGaugeView ist ein einfach einsetzbarer Tachometer ähnlicher View, der einstellbare Visualisierungen für *iOS* Applikationen unterstützt.

Voraussetzungen für *WMGaugeView* sind:

- *Xcode5* oder höhere Versionen
- Apple LLVM Compiler
- *iOS7* oder höhere Version
- ARC

Einbindung in das Projekt:

Alles was Sie für die Einbindung dieser Bibliothek brauchen, ist importieren der *WMGaugeView*.h und *WmGaugeView*.m im Projekt. Die Klasse, die den *WMGaugeView* verwendet soll im Header #*include* "*WMGaugeView*.h" deklarieren.

Konfiguration des *WMGaugeView*:

```
self.gaugeView.minValue = canDataModel.minSpeedValue;
self.gaugeView.maxValue = canDataModel.maxSpeedValue;
self.gaugeView.rangeValues = @[ [NSNumber numberWithFloat:
   canDataModel.maxSpeedValue] ];
self.gaugeView.rangeColors = @[ RGB(170, 170, 170) ];
self.gaugeView.showRangeLabels = YES;
self.gaugeView.unitOfMeasurement = @"Speed";
self.gaugeView.showUnitOfMeasurement = YES;
self.gaugeView.scaleDivisionsWidth = 0.008;
self.gaugeView.scaleSubdivisionsWidth = 0.006;
self.gaugeView.rangeLabelsFontColor = [UIColor blackColor];
self.gaugeView.rangeLabelsWidth = 0.04;
self.gaugeView.rangeLabelsFont = [UIFont fontWithName:@"Helvetica" size:0.04];
self.gaugeView.showInnerRim = YES;
self.gaugeView.showScale = YES;
self.gaugeView.showScaleShadow = YES;
```

F3GAUGEBAR

F3GaugeBar ist ein LED ähnlicher View, der für *iOS* 6 und *iOS* 7 verwendet werden kann. Je nach Stresswert ändert sich die Farbe von Grün bis Rot. Diesen View kann man horizontal oder Vertikal einsetzen.

Konfiguration des F3GaugeBar

```
self.verticalBar.minLimit = [dataModel getMinStressValue];
self.verticalBar.maxLimit = [dataModel getMaxStressValue];
```

RESIDEMENU

RESideMenu ist eine entwickelte Bibliothek unter Lizenz von MIT und sie ermöglich ein *Parallaxeffekt* für die Seitenmenü. Wo die verschiedenen Menüpunkte darunter organisiert werden. Unter dem Begriff Bewegungsparallaxe versteht man in der Wahrnehmungspsychologie den Effekt, der sich optisch ergibt, wenn verschiedene Objekte unterschiedlich voneinander entfernt in einer Landschaft verteilt sind und sich der Beobachter parallel zu diesen Objekten seitlich fortbewegt und dabei in Richtung Horizont blickt.

EINSATZ VON RESIDEMEU IM PROJEKT

1. Zuerst muss eine Subklasse von *RESideMenu* erzeugt werden. Im unserem Fall heißt die Subklasse *RootViewController*.
2. Im zweiten Schritt muss man im Storyboard ein View erzeugen und dann die Klasse dieses View muss als *RootViewController* ausgewählt werden.
3. Es ist sicher zu stellen, dass *#import "RESideMenu.h"* im Header von RootViewController geschrieben ist.
4. Fügt man dem Storyboard weitere View hinzu und nennt man diese Views *LeftViewController, ContentViewController*
5. Und schließlich fügt man die Methode *awakeFromNib* in *RootViewController* mit folgendem Code:

```
- (void)awakeFromNib
{
    self.menuPreferredStatusBarStyle = UIStatusBarStyleLightContent;
    self.contentViewShadowColor = [UIColor blackColor];
    self.contentViewShadowOffset = CGSizeMake(0, 0);
    self.contentViewShadowOpacity = 0.6;
    self.contentViewShadowRadius = 12;
    self.contentViewShadowEnabled = YES;

    self.contentViewController = [self.storyboard
            instantiateViewControllerWithIdentifier:@"contentViewController"];
            self.leftMenuViewController = [self.storyboard
            instantiateViewControllerWithIdentifier:@"leftMenuViewController"];
    self.backgroundImage = [UIImage imageNamed:@"BackgroundMenu"];
    if (self.backgroundImage == nil) {
        NSLog(@"Failed to load background image");
    }
    self.delegate = self;
}
```