markus' nerd blog
INTERACTIVE CURIOSITY

# Mobile Accessibility

Perceivable, Operable, Understandable, Robust

markus' nerd blog
INTERACTIVE CURIOSITY

# Erste Analysen und Tools

Grundlagen der Accessibility-Tests

# Erste Tests zur Problemaufdeckung

- Screenreader-Test
- Automatischer Accessibilty-Scan
- Kontrastüberprüfung
- Navigationsprüfung

https://github.com/ICMaurer/Accessibility-Workshop/

# 1. Screenreader-Test

## **Android** (TalkBack)

**Tool**: TalkBack, der integrierte Screenreader für Android.

**Anwendung**: Aktivieren von TalkBack in den Bedienungshilfen und die App durchgehen, um zu hören, wie die Inhalte vorgelesen werden.

**Ziel**: Überprüfen, ob alle Texte korrekt und verständlich vorgelesen werden. Prüfen, ob Buttons, Labels und Felder klar benannt sind.

## **iOS** (VoiceOver)

**Tool**: VoiceOver, der integrierte Screenreader für iOS.

**Anwendung**: Aktivieren von VoiceOver in den Einstellungen (Bedienungshilfen) und die App durchgehen, um zu überprüfen, ob die Inhalte verständlich sind und korrekt benannt werden.

**Ziel**: Sicherstellen, dass VoiceOver die App-Inhalte klar und sinnvoll wiedergibt. Überprüfen, ob alle interaktiven Elemente gut erkennbar sind.

# WCAG

**1.1.1 Nicht-Text-Inhalte (AA):** Inhalte wie Buttons und Bilder benötigen Alternativtexte und klare Beschriftungen, damit Screenreader sie interpretieren können.

**2.4.3 Fokusreihenfolge (AA):** Eine logische Navigationsreihenfolge ist wichtig, damit Nutzer nicht den Überblick verlieren.

**4.1.2 Name, Rolle, Wert (AA):** Stellt sicher, dass alle Bedienelemente Namen und Rollen haben, die für Assistenztechnologien verfügbar sind.

# 2. Automatischer Accessibility-Scan

## **Android** (Accessibility Scanner)

**Tool:** Accessibility Scanner (kostenlos, funktioniert auch im Emulator).

**Anwendung:** Installation der Accessibility Scanner App und Scan der App-Screens, um Barrierefreiheitsprobleme (z. B. Kontrastprobleme, fehlende Beschriftungen) zu identifizieren.

**Ziel:** Eine Liste mit problematischen Bereichen erhalten und erste Hinweise auf notwendige Korrekturen finden.

## **iOS** (Xcode Accessibility Inspector)

**Tool:** Xcode Accessibility Inspector (im Xcode enthalten, kostenlos, funktioniert auch im Simulator).

**Anwendung:** Im Xcode Simulator den Accessibility Inspector öffnen und die App analysieren. Der Inspector zeigt auf, welche Elemente keine ausreichenden Accessibility-Attribute haben.

**Ziel:** Probleme wie fehlende Labels oder zu geringe Kontraste identifizieren, um eine Basis für die Verbesserung zu schaffen.

# WCAG

**1.3.1 Info und Beziehungen (AA)**: Die Struktur der Inhalte sollte klar sein, damit sie auch ohne visuelle Darstellung interpretiert werden kann.

**1.4.3 Kontrast (Minimum) (AA)**: Sicherstellen, dass der Kontrast von Text und Bildern den Mindestanforderungen entspricht.

**4.1.2 Name, Rolle, Wert (AA)**: Stellt sicher, dass alle UI-Elemente programmatisch interpretierbare Namen und Rollen haben.

# 3. Kontrastüberprüfung

## **Android** (Accessibility Scanner)

**Tool**: Accessibility Scanner bietet eine Kontrastüberprüfung.

**Anwendung**: In der gescannten Ansicht die Kontrastwerte der UI-Elemente analysieren. Probleme werden rot markiert und die Scanner-App gibt Empfehlungen.

**Ziel**: Prüfen, ob alle Texte ausreichend Kontrast zum Hintergrund haben, damit sie auch bei visuellen Einschränkungen gut lesbar sind.

## **iOS** (Xcode Accessibility Inspector)

**Tool**: Xcode Accessibility Inspector.

**Anwendung**: Der Accessibility Inspector zeigt automatisch Kontrastprobleme an, wenn ein Text zu wenig Kontrast zum Hintergrund hat.

**Ziel**: Sicherstellen, dass Texte und UI-Elemente genug Kontrast aufweisen, um für alle Nutzergruppen gut lesbar zu sein.

# WCAG

**1.4.3 Kontrast (Minimum) (AA)**: Text und Bilder müssen ausreichenden Kontrast zum Hintergrund aufweisen.

**1.4.6 Kontrast (Erweitert) (AAA)**: Erhöhter Kontrast für Textinhalte verbessert die Zugänglichkeit für Nutzer mit Sehbehinderungen.

# 4. Navigationsprüfung

## **Android** (TalkBack /Accessibility Scanner)

**Tool:** TalkBack und Accessibility Scanner.

**Anwendung:** Mit TalkBack durch die App navigieren, um sicherzustellen, dass alle interaktiven Elemente erreichbar sind und die Reihenfolge der Fokusbewegung sinnvoll ist. Accessibility Scanner kann helfen, fehlende Labels und fokussierbare Elemente zu identifizieren.

**Ziel:** Prüfen, ob die App intuitiv navigierbar ist, ohne dass wichtige Inhalte oder Bedienelemente ausgelassen werden.

## **iOS** (Xcode Accessibility Inspector)

**Tool:** VoiceOver und Accessibility Inspector.

**Anwendung:** VoiceOver aktivieren und die App durchgehen, um sicherzustellen, dass alle relevanten Elemente korrekt fokussierbar sind und die Navigationsreihenfolge Sinn ergibt. Der Accessibility Inspector kann zusätzliche Insights geben.

**Ziel:** Sicherstellen, dass die Navigation für VoiceOver-Nutzer reibungslos und logisch abläuft.

# WCAG

**2.4.3 Fokusreihenfolge (AA)**: Eine logische Reihenfolge stellt sicher, dass Nutzer durch die App geführt werden.

**2.1.1 Tastaturbedienbarkeit (AA)**: Alle Funktionen müssen ohne Maus zugänglich sein.

**2.4.7 Sichtbarer Fokus (AA)**: Der sichtbare Fokus auf dem aktuell ausgewählten Element ist essentiell für die Orientierung.

# Zusammenfassung der ersten Testschritte

- Grundlegende Probleme in
- der **Wahrnehmung** und
- **Bedienbarkeit** aufdecken

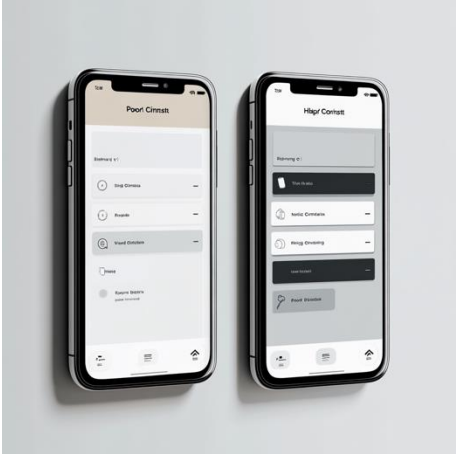
# Beispieldokumentation

# Wahrnehmung

(Perceivable)

Ziel: Sicherstellen, dass alle Inhalte für alle Benutzer wahrnehmbar sind.

# 1. Kontrasteinstellungen (Android)



## Android XML

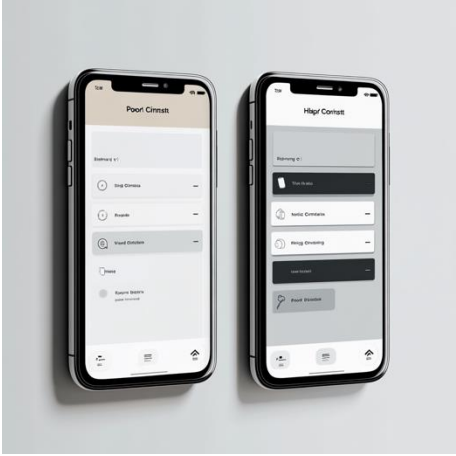
```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Beispieltext"
   android:textColor="#000000"
   android:background="#FFFFFF"
   android:textSize="18sp"
   android:padding="16dp"
/>
```

## Android Jetpack Compose

```
@Composable
fun ContrastedText() {
   Text(
      text = "Beispieltext",
      color = MaterialTheme.colorScheme.onBackground,
// Kontrastfarbe für den Hintergrund
      modifier = Modifier

.background(MaterialTheme.colorScheme.background)
         .padding(16.dp),
      fontSize = 18.sp
   )
}
```

# 1. Kontrasteinstellungen (iOS)



## SwiftUI

```
Text("Beispieltext")
    .font(.system(size: 18))
    .foregroundColor(.black) // Textfarbe: Schwarz für besseren Kontrast
    .padding()
    .background(Color.white) // Hintergrundfarbe: Weiß
    .cornerRadius(8)
```

## UIKit

```
let label = UILabel()
label.text = "Beispieltext"
label.textColor = UIColor.black // Textfarbe: Schwarz für besseren Kontrast
label.backgroundColor = UIColor.white // Hintergrundfarbe: Weiß
label.font = UIFont.systemFont(ofSize: 18)
label.textAlignment = .center
```

# 2. Alternative für Bilder (Android)

## XML

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/beispielbild"
    android:contentDescription="Beschreibung des Bildes"
/>
```

## Jetpack Compose

```
@Composable
fun AccessibleImage() {
    Image(
        painter = painterResource(id =
                                  R.drawable.beispielbild),
        contentDescription = "Beschreibung des Bildes",
        modifier = Modifier.size(128.dp)
    )
}
```

# 2. Alternative für Bilder (iOS)

## SwiftUI

```
Image("beispielbild")
    .resizable()
    .frame(width: 128, height: 128)
    .accessibilityLabel("Beschreibung des Bildes")
```

## UIKit

```
let imageView = UIImageView(image: UIImage(named:
"beispielbild"))

imageView.accessibilityLabel = "Beschreibung des Bildes"
```

# Aufgabe

**iOS (SwiftUI) Projekt**

1. **Mangelnder Farbkontrast bei Texten**

- **Problem:** Der Kontrast für einige Texte wie „Banking App Login“ und Schaltflächen ist möglicherweise unzureichend.
- **Verletzte Richtlinie:** 1.4.3 Kontrast (Minimum) (AA), 1.4.6 Kontrast (Erweitert) (AAA).
- **Lösung im Code:** Verwenden Sie kontrastreichere Farben für Text und Schaltflächen, um die Lesbarkeit zu verbessern.

# Aufgabe

**Android (XML-Layout) Projekt**

1. **Fehlende contentDescription für Bilder und Icons**

- **Problem:** Bilder und Icons haben keine contentDescription und sind für Screenreader unsichtbar.
- **Verletzte Richtlinie:** 1.1.1 Nicht-Text-Inhalt (AA).
- **Lösung im Code:** Stellen Sie sicher, dass jedes wichtige visuelle Element eine contentDescription hat.

# Aufgabe

**Android (Jetpack Compose) Projekt**

1. **Unzureichender Farbkontrast**

- **Problem:** Text- und Hintergrundfarben haben zu wenig Kontrast.
- **Verletzte Richtlinie:** 1.4.3 Kontrast (Minimum) (AA).
- **Lösung im Code:** Verwenden Sie kontrastreichere Farben, um sicherzustellen, dass die Texte auch bei schlechten Lichtverhältnissen gut lesbar sind.

# Bedienbarkeit

(Operable)

Ziel: Sicherstellen, dass alle Funktionen für alle Benutzer bedienbar sind, insbesondere über Tastatur und Screenreader.

# 1. Navigationsreihenfolge (Android)

## XML

```
<Button
   android:id="@+id/button1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Button 1"
   android:nextFocusForward="@id/button2" />

<Button
   android:id="@+id/button2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Button 2"
   android:nextFocusForward="@id/button3" />
```

android:nextFocusForward, android:nextFocusDown, android:nextFocusLeft und android:nextFocusRight

## Compose

```
val (button1, button2, button3) = FocusRequester.createRefs()

Button(
   onClick = { /* Handle click */ },
   modifier = Modifier
      .focusRequester(button1)
      .focusOrder(button1) { next = button2 }
) {
   Text("Button 1")
}

Button(
   onClick = { /* Handle click */ },
   modifier = Modifier
      .focusRequester(button2)
      .focusOrder(button2) { next = button3 }
) {
   Text("Button 2")
}
```

# 1. Navigationsreihenfolge (iOS)

## SwiftUI

```
Button("Button 1") { }
 .accessibilitySortPriority(2)

Button("Button 2") { }
 .accessibilitySortPriority(1)
```

## UIKit

```
UIAccessibility.post(notification:        .layoutChanged, argument:
someView)
```

# 2. Große Bedienelemente / Touch-Targets (Android)

WCGA Konform: mind. 48dp

## XML

```
<Button
   android:id="@+id/large_button"
   android:layout_width="48dp"
   android:layout_height="48dp"
   android:text="Großer Button" />
```

## Compose

```
Button(
   onClick = { /* Handle click */ },
   modifier = Modifier.size(48.dp)
) {
   Text("Großer Button")
}
```

# 2. Große Bedienelemente / Touch-Targets (iOS)

## SwiftUI

```
Button("Großer Button") { }
    .frame(width: 44, height: 44)
```

## UIKit

```
let button = UIButton(type: .system)

button.frame = CGRect(x: 0, y: 0, width: 44,
                      height: 44)

button.setTitle("Großer Button", for: .normal)
```

# Aufgabe

**iOS (SwiftUI) Projekt**

1. **Keine klare Fokus-Reihenfolge für VoiceOver**

- **Problem:** Die Reihenfolge der Elemente ist möglicherweise nicht klar und intuitiv.
- **Verletzte Richtlinie:** 2.4.3 Fokus-Reihenfolge (AA).
- **Lösung im Code:** Nutzen Sie .accessibilitySortPriority() bei Bedarf, um die Reihenfolge der fokussierten Elemente zu steuern.

# Aufgabe

**Android (XML-Layout) Projekt**

1. **Keine logische Fokus-Reihenfolge für TalkBack**

- **Problem:** Die Fokus-Reihenfolge der Elemente ist möglicherweise nicht logisch.
- **Verletzte Richtlinie:** 2.4.3 Fokus-Reihenfolge (AA).
- **Lösung im Code:** Nutzen Sie android:accessibilityTraversalBefore, um die Reihenfolge der fokussierten Elemente anzupassen.

# Aufgabe

**Android (Jetpack Compose) Projekt**

1. **Große Bedienelemente**

- **Problem:** Touch-Targets sind möglicherweise zu klein und schwer zu bedienen.
- **Verletzte Richtlinie:** 2.5.5 Zielgröße (AA).
- **Lösung im Code:** Passen Sie die Größe der Touch-Targets an, sodass sie mindestens 48x48dp betragen, um eine bessere Bedienbarkeit zu gewährleisten.

# Verständlichkeit

(Understandable)

Ziel: Sicherstellen, dass alle Inhalte und Bedienelemente verständlich sind.

# 1. Beschriftungen und Hilfetexte (Android)

WCGA Konform: mind. 48dp

## XML

```
<TextView
   android:id="@+id/usernameLabel"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Benutzername:" />

<EditText
   android:id="@+id/usernameInput"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:hint="Geben Sie Ihren Benutzernamen ein"
   android:labelFor="@id/usernameLabel" />
```

## Compose

```
TextField(
   value = username,
   onValueChange = { newUsername -> username =
                                          newUsername },
   placeholder = { Text("Geben Sie Ihren Benutzernamen
                                                ein") },
   label = { Text("Benutzername") }
)
```

# 1. Beschriftungen und Hilfetexte (iOS)

## SwiftUI

```
TextField("Geben Sie Ihren Benutzernamen ein", text: $username)
  .textFieldStyle(RoundedBorderTextFieldStyle())
  .padding()
```

## UIKit

```
let usernameField = UITextField()

usernameField.placeholder = "Geben Sie Ihren
                             Benutzernamen ein"
```

# 2. Fehlermeldungen (Android)

WCGA Konform: mind. 48dp

## XML

```
val usernameInput: EditText =
        findViewById(R.id.usernameInput)
if (usernameInput.text.toString().isEmpty()) {
        usernameInput.error = "Benutzername ist
                                erforderlich"
}
```

## Compose

```
if (username.isEmpty()) {
   Text("Benutzername ist erforderlich", color = Color.Red)
}
```

# 2. Fehlermeldungen (iOS)

## SwiftUI

```
if username.isEmpty {
  Text("Benutzername ist erforderlich")
      .foregroundColor(.red)
}
```

## UIKit

```
if usernameField.text?.isEmpty ?? true {
   let alert = UIAlertController(title: "Fehler",      message: "Benutzername ist erforderlich",    preferredStyle: .alert)
   alert.addAction(UIAlertAction(title: "OK", style:
                                        .default, handler: nil))
   present(alert, animated: true, completion: nil)
}
```

# Aufgabe

**iOS (SwiftUI) Projekt**

1. **Fehlende Anweisungen für nicht-standardmäßige Bedienelemente**

- **Problem:** Schaltflächen und Felder haben keine ausreichenden Anweisungen.
- **Verletzte Richtlinie:** 3.3.2 Beschriftungen oder Anweisungen (AA).
- **Lösung im Code:** Verwenden Sie .accessibilityHint(…) für zusätzliche Hinweise, um die Funktionalität klarer zu beschreiben.

# Aufgabe

**Android (XML-Layout) Projekt**

1. **Beschriftungen und Hilfetexte**

- **Problem:** Textfelder und Schaltflächen haben keine ausreichenden Beschriftungen oder Hinweise.
- **Verletzte Richtlinie:** 3.3.2 Beschriftungen oder Anweisungen (AA).
- **Lösung im Code:** Fügen Sie android:hint und android:labelFor hinzu, um Benutzern kontextbezogene Hilfen zu geben.

# Aufgabe

**Android (Jetpack Compose) Projekt**

1. **Fehlermeldungen**

- **Problem:** Fehlermeldungen bei Eingabefehlern sind nicht verständlich genug.
- **Verletzte Richtlinie:** 3.3.1 Fehlererkennung (AA).
- **Lösung im Code:** Formulieren Sie klare Fehlermeldungen, die den Benutzer wissen lassen, wie der Fehler behoben werden kann.

# Robustheit

(Robust)

Ziel: Sicherstellen, dass alle Inhalte und Bedienelemente mit verschiedenen Technologien kompatibel und langlebig sind.

e

# 2. Fehlermeldungen (Android)

WCGA Konform: mind. 48dp

## XML

```
val usernameInput: EditText =
        findViewById(R.id.usernameInput)
if (usernameInput.text.toString().isEmpty()) {
        usernameInput.error = "Benutzername ist
                                erforderlich"
}
```

## Compose

```
if (username.isEmpty()) {
   Text("Benutzername ist erforderlich", color = Color.Red)
}
```

# 1. Kompatibilität mit unterschiedlichen Versionen

- Verwenden Sie AppCompat-Komponenten wie AppCompatTextView und AppCompatButton in XML, um sicherzustellen, dass die App abwärtskompatibel ist und auf älteren Android-Versionen robust läuft.
- In Compose wird die Abwärtskompatibilität durch die Nutzung von Material Design-Komponenten wie MaterialTheme unterstützt.

# 2. Dynamische Schriftgrößen

- Ändern Sie die Schriftgröße in den Einstellungen des Geräts oder Emulators und überprüfen Sie, ob alle Texte korrekt skaliert werden, ohne dass das Layout bricht.
- Verwenden Sie **Dynamic Type** in den iOS-Einstellungen und überprüfen Sie, ob Texte sich entsprechend anpassen und gut lesbar bleiben.
- In SwiftUI verwenden Sie .dynamicTypeSize(...), um die dynamische Textgröße zu unterstützen.

# Aufgabe

**iOS (SwiftUI) Projekt**

1. **Dynamische Schriftgrößen**

- **Problem:** Texte passen sich möglicherweise nicht an die Einstellungen für dynamische Schriftgrößen an.
- **Verletzte Richtlinie:** 1.4.4 Textgröße ändern (AA).
- **Lösung im Code:** Verwenden Sie Dynamic Type und aktivieren Sie diese Einstellung in den iOS-Einstellungen.

# Aufgabe

**Android (XML-Layout) Projekt**

1. **Kompatibilität mit unterschiedlichen Versionen**

- **Problem:** Elemente nutzen möglicherweise keine AppCompat-Komponenten, was zu Inkompatibilitäten führen kann.
- **Verletzte Richtlinie:** 4.1.1 Kompatibilität.
- **Lösung im Code:** Verwenden Sie AppCompat-Komponenten, um eine breite Unterstützung für verschiedene Versionen zu gewährleisten.

# Aufgabe

**Android (Jetpack Compose) Projekt**

1. **Dynamische Schriftgrößen testen**

- **Problem:** Texte passen sich möglicherweise nicht den Systemeinstellungen für größere Schriftgrößen an.
- **Verletzte Richtlinie:** 1.4.4 Textgröße ändern (AA).
- **Lösung im Code:** Verwenden Sie MaterialTheme.typography, um die Schriftgrößen dynamisch anpassbar zu machen.

# Deklarativ?