



11. März 2025

Anleitung Noten-App

Schnuppertag Applikationsentwicklung BIT

Beschreibung

Für den Schnuppertag Applikationsentwicklung 2025 haben wir (die Lernenden des zweiten und dritten Lehrjahres) uns dazu entschieden, eine Noten-App umzusetzen. Dabei können Fächer mit Noten erstellt, bearbeitet und gelöscht werden. Durchschnitte mit Gewichtungen können abgelesen und Wunschnoten berechnet werden. Die App verfügt auch über die Möglichkeit, alle Noten, als ein PDF zu exportieren.

Emoji	Bedeutung
⚠	wichtige Informationen, nicht überspringen!
💡	Erklärung
💡	Verweis auf Theorie (optional, wenn's dich interessiert)
ℹ	Tipp
🌟	optional für Schnelle

Vorbereitung

Expo-Go

Verbinde dich mit deinem Smartphone in das WLAN-Netzwerk «**Softwareschmiede**», das Passwort lautet «**work@BIT**». Öffne den App-Store/Play-Store und lade die App “**Expo Go**” herunter.



Development-Server starten

Wechsle mit dem **Terminal** in das «**grades-bit**» Verzeichnis (das innere der beiden) und starte den Development Server. Dies erreichst du mit den folgenden Befehlen.

```
cd grades-bit/application/grades-bit  
npm start
```

App auf Smartphone starten

Sobald der QR-Code erscheint, kannst du diesen mit deiner Smartphonekamera scannen und in Expo Go öffnen. Die App sollte somit laufen.

Sobald du Änderungen an deinem Code vornimmst, musst du «**Ctrl + S**» in Visual Studio Code drücken, um einen Reload zu bewirken (alternativ kannst du auch im cmd-Fenster «**r**» drücken). Die Änderungen werden danach gleich auf deinem Smartphone sichtbar.



Tipp: Mit «Links oben > File > Save All» Kannst du alle Dateien auf einmal speichern.

Wenn dies nicht geht: Starte Expo-Go auf deinem Smartphone neu (bei iPhone hochswippen und die App ganz wegwischen).

Werden immer noch keine Änderungen übernommen: Starte den Development-Server neu, indem du mittels «**Ctrl + C**» im Terminal den laufenden Server stoppst und ihn mit «**npm start**» erneut startest.

Ordner- und Seitenstruktur kennenlernen

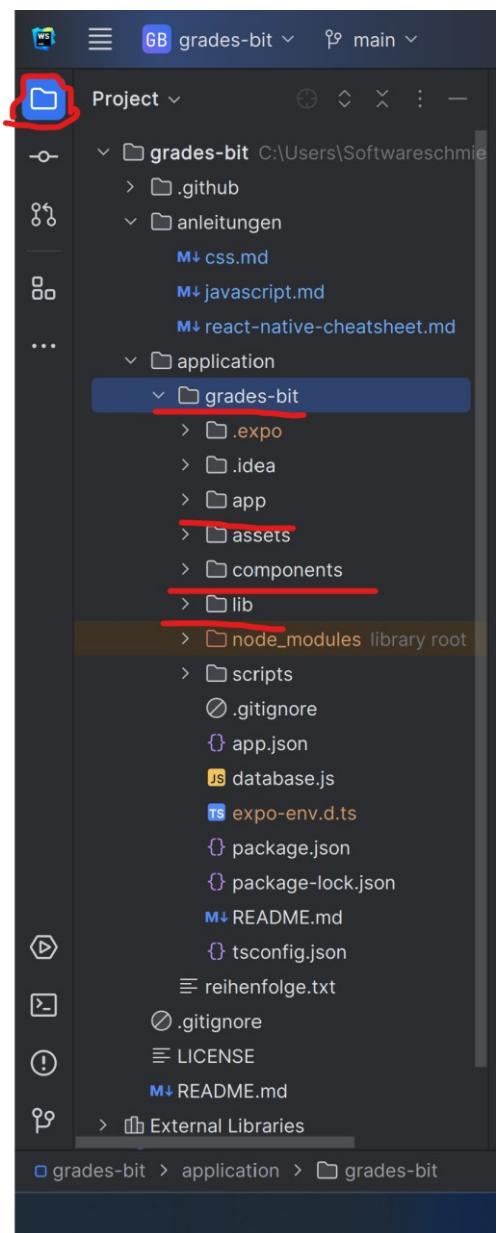
Bevor wir mit dem Programmieren beginnen, schauen wir uns zuerst einmal die Ordner- und Seitenstruktur an.

Ordnerstruktur

Im «app» Ordner gibt es mehrere Dateien und Ordner. «**index.js**» ist der Einstiegspunkt in die App, also «index.js» wird als erstes geladen, wenn wir die App auf dem Smartphone öffnen (Das HTML von «index.js» wird auf dem Bildschirm des Smartphones angezeigt).

Die beiden Ordner «**fach**» und «**note**» spiegeln zwei Unterseiten wider. Warum Ordner? Ein Ordner ist eine Ebene. In den beiden Ordner «**fach**» und «**note**» gibt es je eine Datei namens «**create.js**» und ein Ordner «**[id]**». Die Seite "**create.js**" ist die Seite, die beim Erstellen eines Fachs oder einer Note angezeigt wird.

Wir werden dir immer sagen in welcher Datei du arbeiten sollst, also keine Angst, falls dich das etwas verwirrt. Wenn wir eine Pfadangabe machen, musst du einfach immer vom **grades-bit** Verzeichnis ausgehen.



Anleitung

- ⚠ Bei den Aufgaben ist der Teil, den du machen sollst, immer **Blau** markiert. Der schwarze Code wurde von uns geschrieben, wenn du dich an ihm orientierst, sollte es kein Problem sein deinen Code am richtigen Ort zu schreiben.
- ⚠ Die **drei roten Punkte ...** zeigen dir, dass hier noch mehr Code wäre, aber wir diesen nicht in der Anleitung geschrieben haben, da es sonst zu viel Platz einnehmen würde.
- ⚠ Wenn du bei einer Aufgabe in der Anleitung selbst etwas schreiben musst, gibt es in den jeweiligen Dateien Kommentare, die dir damit helfen.

```
// Kommentare beginnen mit einem doppelten Schrägstrich  
  
/* Das hier ist ebenfalls ein Kommentar, aber im HTML */
```

Schritt 1 – Titel

Wir beginnen mal ganz einfach. Als Erstes fügen wir auf der **Homepage (app/index.js)** einen **Titel** hinzu, so etwas wie «Fächer». Kopiere oder schreibe folgenden Code ab:

```
Datei: app/index.js  
export default function index() {  
  
    useFocusEffect(  
        useCallback(() => {  
            [], []  
        }, []  
    );  
  
    return (  
        <View style={styles.container}>  
            <View style={styles.titel}>  
                <Text style={styles.text}>Fächer</Text>  
            </View>  
        </View>  
    );  
}
```

Speichere deine Änderungen mit «**Ctrl + S**». Nun solltest du auf deinem Smartphone den Text sehen.

Schritt 2 - Knopf

Damit wir Fächer erstellen können, brauchen wir irgendeine Art von Formular, um den Namen eines Fachs eingeben zu können. Dies implementieren wir jetzt. Füge die blauen Zeilen dem Code hinzu:

Datei: app/index.js

```
export default function index() {  
    ...  
    function zumErstellen() {  
        router.push("/fach/create")  
    }  
  
    return (  
        <View style={styles.container}>  
            <Knopf beimKlicken={zumErstellen} text={"Fach hinzufügen"} />  
            <View style={styles.titel}>  
                <Text style={styles.text}>Fächer</Text>  
            </View>  
        </View>  
    );  
}
```

Speichere deine Änderungen wieder mit «**Ctrl + S**».

Öffne nun mal die Datei «**Knopf.js**» im Ordner «**Eingaben**» unter «**components**». Da wirst du sehen, dass diese noch leer ist (fast leer). Füge nun hier auch die blauen Zeilen ein:

Datei: components/Eingaben/Knopf.js

```
export default function Knopf({beimKlicken, text}) {  
    return (  
        <>  
            <TouchableOpacity style={styles.button} onPress={beimKlicken}>  
                <Text style={styles.text}>{text}</Text>  
            </TouchableOpacity>  
        </>  
    );  
}
```

Speichere deine Änderungen mit «**Ctrl + S**».



Verweis auf Theorie: **Components**



Verweis auf Theorie: **TouchableOpacity**

Schritt 3 – Fächer hinzufügen

Nun können wir zur Fachhinzufüge Seite mit einem Knopf navigieren.

Man soll nicht einfach irgendwas eingeben können, deshalb **validieren** wir die Eingabe, die wir mit unserem Textfeld machen. Diesen Code siehst du unten in der Funktion **formularBestätigt()**, sieh sie dir kurz an und schreibe dann den blau markierten Code in die Datei.

Datei: app/fach/create.js

```
export default function FachErstellen() {
    const [name, nameSetzen] = useState("")
    const [error, errorSetzen] = useState(null)

    async function formularBestätigt() {
        if (name.length < 2) {
            errorSetzen("Name ist zu kurz")
        } else if (name.length > 20) {
            errorSetzen("Name ist zu lang")
        } else {
            await insertIntoFach(name)
            router.back()
        }
    }

    return (
        <View style={styles.container}>
            <StatusBar/>
            <Text style={styles.bigText}>...</Text>
            <Textfeld
                ...
                />
                {error ? ... : null}
                <Knopf beimKlicken={formularBestätigt} text={"Bestätigen"} />
        </View>
    )
}
```

Speichere deine Änderungen mit «**Ctrl + S**».



Verweis auf Theorie: **useState**

Schritt 4 – Fächer anzeigen

Grossartig, jetzt können wir Fächer hinzufügen! Damit unsere App Fächer anzeigen kann, benötigen wir eine Liste von Fächern. Dafür gibt es die Komponente **FachList**, die alle Fächer in einer Liste darstellt. Diese bauen wir nun ein.

Die Fächer werden zwar gespeichert, aber nicht angezeigt. Warum? **Sie werden nicht geladen**. Der Code, den wir einfügen, fügt dies hinzu: Ein **Hook** (Details dazu lassen wir aus) und die Funktion **useCallback()**. Die rufen die Fächer aus der Datenbank ab.

Datei: app/index.js

```
export default function index() {
    const [faecher, setFaecher] = useState([])

    useEffect(
        useCallback(() => {
            async function LadeFaecher() {
                const daten = await getAllFaecher()
                setFaecher(daten)
            }
            LadeFaecher()
        }, [])
    );

    ...

    return (
        <View style={styles.container}>
            <Knopf beimKlicken={zumErstellen} text={"Fach hinzufügen"} />
            <View style={styles.list}>
                <View style={styles.titel}>
                    <Text style={styles.text}>Fächer</Text>
                </View>
                <FachList faecher={faecher}/>
            </View>
        </View>
    );
}
```

Speichere deine Änderungen mit «**Ctrl + S**».



Verweis auf Theorie: **useFocusEffect & useCallback**

Schritt 5 – Navigation Fach Detailseite

Nun möchten wir gerne auf die Detailseite eines Faches navigieren, um später Noten für das Fach verwalten zu können. Wir müssen zu «/fach/[id]/index.js» bzw. «/fach/[id]» navigieren. «index.js» Kann hier ignoriert werden, da React-Native automatisch weiß, dass es sich um «index.js» handelt.

Damit wir dieses Verhalten erreichen können, brauchen wir als Erstes eine solche Funktion. Diese Funktion hier navigiert zum entsprechenden Pfad. Wir können den Pfad einfach als Text angeben. Nicht vergessen «[id]» ist nur ein Platzhalt, wir müssen diesen von Hand ersetzen.

Implementiere nun den benötigten Code. Das Text-Tag ist bereits vorhanden.

Datei: components/Listen/FachListItem.js

```
export default function FachListItem({fach}) {  
  
    function navigiereZuDetailAnsicht() {  
        router.push(`fach/${fach.id}`)  
    }  
  
    return (  
        <View>  
            <TouchableOpacity onPress={navigiereZuDetailAnsicht} style={styles.container}>  
                <Text style={[styles.text, styles.titel]} numberOfLines={1} ellipsisMode={"tail"}>{fach.name}</Text>  
            </TouchableOpacity>  
        </View>  
    )  
}
```

Speichere deine Änderungen mit «**Ctrl + S**».

Hier sollte kein Zeilenumbruch sein: `style={styles.container}` (Es hat zu wenig Platz um die Ganze Zeile auf ein A4 Blatt zu schreiben).



Verweis auf Theorie: **String Interpolation**

Schritt 6 – Fach Detailseite

Jetzt wollen wir die Fachdetailseite implementieren, also die Seite, auf welcher man das Fach und die Noten für das Fach verwalten kann.

Datei: app/fach/[id]/index.js

```
export default function Index() {
    const {id} = useLocalSearchParams()
    const [sichtbarkeit, sichtbarkeitSetzen] = useState(null)
    const [fach, setFach] = useState({})

    useFocusEffect(
        useCallback(() => {
            if (!id) {
                return
            }

            async function FachLaden() {
                const daten = await getFachById()
                setFach(daten)
            }

            FachLaden()
        }, [id])
    );

    ...

    return (
        <View style={styles.container}>
            <View style={styles.titleBar}>
                <Text style={styles.titel} numberOfLines={1}>{fach.name}</Text>
            </View>
        </View>
    )
}
```

Schritt 7 – Fach löschen

Nun, da wir auf unsere Detail-Ansicht navigieren können, wollen wir einige Funktionen einbauen, wie z.B. den Namen eines Faches zu ändern oder ein Fach ganz zu löschen. Dazu werden wir einen «**Icon-Knopf**» verwenden. Der IconKnopf ist einer unserer Komponenten,

Im **Index von Fach** müssen wir zuerst eine Funktion schreiben, die unser Fach aus der Liste entfernen kann. In der Funktion werden mehrere Datenbankabfragen gemacht, damit wollen wir uns erstmal nicht beschäftigen. **Füge den Code am richtigen Ort ein.**

Jetzt, da wir unsere Funktion geschrieben haben. Müssen wir diese noch aufrufen können. Jetzt setzen wir den besagten **IconKnopf** ein. Vielleicht hast du beim Anschauen der Komponente schon bemerkt, wofür das **icon** Attribut verwendet wird. Mit diesem Attribut bestimmen wir, wie unser Knopf aussehen soll, wir verwenden hier passend «**trash**».

 **Tipp:** Ersetze «Dein Text» und «Dein Titel» mit einem passenden Text und Titel für das **Frage-Fenster**.

Datei: app/fach/[id]/index.js

```
...
async function fachLöschen() {
  const noten = await getNotenByFachId(id)
  for (const note of noten) {
    await removeNote(note.id)
  }
  await removeFach(id)
  router.back()
}
function frageLöschen() {
  sichtbarkeitSetzen(true)
}
...
return (
  <View style={styles.container}>
    <FrageFenster
      text={"Dein Text"}
      titel={"Dein Titel"}
      istSichtbar={istSichtbar}
      sichtbarkeitSetzen={sichtbarkeitSetzen}
      wennAbbrechenAngeklickt={() => {}}
      wennBesätigenAngeklickt={fachLöschen}>

    <View style={styles.titleBar}>
      <Text style={styles.titel} numberofLines={1}>{fach.name}</Text>
      <View style={styles.icons}>
        <IconKnopf beimKlicken={frageLöschen} icon={"trash"} />
      </View>
    </View>
  </View>
)
```

Schritt 8 – Navigation Fach Edit Seite

Nun können wir Fächer erstellen, anzeigen und löschen. Was ist aber, wenn wir einen Tippfehler beim Erstellen machen? Wir müssen das Fach wieder Löschen und ein neues Erstellen. Das ist noch nicht so toll, deswegen wollen wir jetzt auch noch Fächer bearbeiten können.

Füge noch einen zweiten IconKnopf hinzu. Dieser soll das Icon «pencil» haben. Beim Klicken musst du eine Funktion aufrufen, welche zur Fachbearbeitungsseite wechselt.

Datei: app/fach/[id]/index.js

```
...
    function zumBearbeiten() {
        router.push(`/fach/${id}/edit`)
    }

    return (
        <View style={styles.container}>
            <FrageFenster
                text={"DeinText"}
                titel={"DeinText"}
                istSichtbar={istSichtbar}
                sichtbarkeitSetzen={sichtbarkeitSetzen}
                wennAbbrechenAngeklickt={() => {}}
                wennBesättigenAngeklickt={fachLöschen}/>

            <View style={styles.titleBar}>
                <Text style={styles.titel} numberOfLines={1}>{fach.name}</Text>
                <View style={styles.icons}>
                    <IconKnopf beimKlicken={fachLöschen} icon={"trash"/> Hier musst du einen zweiten IconKnopf einbauen. Schreibe diesen Text hier nicht ab.
                </View>
            </View>
        </View>
    )
...

```

Schritt 9 – Fachbearbeitungsseite

So nun können wir zur Bearbeitungsseite wechseln, jedoch ist sie noch leer. Das wollen wir jetzt ändern:

```
Datei: app/fach/[id]/edit.js
export default function Edit() {
    ...
    useEffect(() => {
        if (!id) {
            return;
        }

        async function FachLaden() {
            const data = await getFachById(id)
            fachSetzen(data)
        }

        FachLaden()
    }, [id]);

    async function formularBestätigt() {
        if (fach.name.length < 2) {
            errorSetzen("Name ist zu kurz")
        } else if (fach.name.length > 20) {
            errorSetzen("Name ist zu lang")
        } else {
            await updateFach(id, fach.name)
            router.back()
        }
    }
    ...

    return (
        <View style={styles.container}>
            <Text style={styles.bigText}>Beispieltext</Text>
            <Textfeld
                titel={"Name des Fachs"}
                inhalt={fach.name}
                wennInhaltVerändertWird={(neuerInhalt) => fachSetzen({...fach,
name: neuerInhalt})}
                platzhalter={"Neuer Name"}
            />

            {error ? <Text style={styles.error}>{error}</Text> : null}
            Hier musst du zwei Knöpfe einfügen: Abbrechen und Speichern, rufe die
            korrekte Funktion auf. Schreibe diesen Text hier wieder nicht ab!
        </View>
    )
}
```



Verweis auf Theorie: **useEffect**

Schritt 10 - Note hinzufügen Knopf

Erstelle ein Knopf auf der Fach Detail Seite, welcher den Text «Note hinzufügen» hat und zur Fachhinzufüge Seite ([«note/create.js»](#)) navigiert.

Datei: app/fach/[id]/index.js

```
export default function Index() {  
    ...  
    return (  
        <View style={styles.container}>  
            ...  
            <View style={styles.titleBar}>  
                ...  
            </View>  
  
            Erstelle hier den Knopf «Note hinzufügen». Text nicht abschreiben...  
        </View>  
    )  
}
```

Verwende diese Zeile Code, um zu Erstellseite zu navigieren:

```
router.push(`/note/create?id=${id}`)
```



Falls du mehr über [«?id=\\${id}»](#) wissen willst: Frage einen Lernenden.

Schritt 11 – Notenhinzufüge Seite

Jetzt können wir zur Hinzufügeseite navigieren, diese ist im Moment aber noch leer. Auf der Notenhinzufüge Seite brauchen wir:

Bedingungen für die Validierung:

- Titel Länge mindestens 2
- Titel Länge maximal 20
- Note mindestens 1
- Note maximum 6
- Gewichtung mindestens 1

 Verwende den Code, welcher schon in der Datei [«app/note/create.js»](#) ist. Du musst ein paar Dinge anpassen, Die Kommentare in der Datei helfen dir.

Schritt 12 – Noten anzeigen

Die Notenübersicht machen wir auf der Detailseite vom Fach, da wir alle Noten nach Fächern geordnet ansehen wollen. Füge nun den notwendigen Code ein, um die Noten anzuzeigen.

Datei: app/fach/[id]/index.js

```
...
  const [istSichtbar, sichtbarkeitSetzen] = useState(false)
  const [noten, setNoten] = useState([])

  useEffect(
    useCallback(() => {
      ...
        async function NotenLaden() {
          const data = await getNotenByFachId(id)
          setNoten(data)
        }
        FachLaden()
        NotenLaden()
      }, [id])
  );
  ...

  return (
    <View style={styles.container}>
      ...
        <Knopf beimKlicken={zumErstellen} text={"Note hinzufügen"} />
        <View style={styles.list}>
          <NotenListe noten={noten}/>
        </View>
      </View>
    )
  }
}
```



Als erstes erstellen wir einen **useState**, um die Noten zu speichern. Im **useEffect** laden wir den Noten in den **useState** und anschliessend zeigen wir sie mithilfe der **NotenListe** an.

Schritt 13 – Navigation Noten Detail-/Editseite

Bevor wir Noten bearbeiten können, müssen wir zuerst auf die Bearbeitungsseite navigieren.



⚠️ Verwende den Code, welcher schon in der Datei «**components/Listen/NoteListeltem.js**» ist.

Schritt 14 – Noten Detailseite

Bei den Detailseiten der Noten wird es etwas anders als bei den Detailseiten der Fächer. Wir wollen die Noten Detailseite so gestalten, dass sie gleichzeitig die Bearbeitungsseite ist.



⚠️ Verwende den Code, welcher schon in der Datei «**app/note/[id]/index.js**» ist.



Tipp: Verwende dieselben Bedingungen für die Validierung der Note beim Speichern, wie beim Erstellen.

Schritt 15 – Durchschnitt anzeigen

Wir wollen nun noch den Notendurchschnitt anzeigen können. Einmal auf der Fachdetail Ansicht und einmal auf der Homepage.

Im Components Ordner gibt es die Komponente «**Durchschnitt.js**», lies dir den Code aufmerksam durch und füge diese beiden Farbstufen ein.

Du darfst auch noch mehr Farbstufen hinzufügen 😊

Datei: components/Durchschnitt.js

```
...
function erhalteFarbeNachNotenSchnitt(durchschnitt) {
    if (durchschnitt >= 4) {
        return "green"
    }

    return "red"
}
...
```

Den Durchschnitt kannst du jetzt auf beiden Seiten (Home, Fach) einfügen:

Datei: app/fach/[id]/index.js

```
...
return (
    <View style={styles.container}>
        <StatusBar/>
        <FrageFenster
            ...
        />

        <View style={styles.titleBar}>
            ...
        </View>

        <Knopf beimKlicken={zumErstellen} text={"Note hinzufügen"} />
        <View style={styles.list}>
            <NotenListe noten={noten}/>
        </View>
        <View style={styles.durchschnitt}>
            <TrennLinie/>
            <Durchschnitt noten={noten}/>
        </View>
    </View>
)
...
```

Datei: app/index.js

```
export default function index() {
    const [faecher, setFaecher] = useState([])
    const [noten, setNoten] = useState([])

    useEffect(
        useCallback(() => {
            ...
            async function LadeNoten() {
                const daten = await getAllNoten()
                setNoten(daten)
            }

            LadeFaecher()
            LadeNoten()
        }, [])
    );
}

...
return (
    <View style={styles.container}>
        <StatusBar/>

        <Knopf beimKlicken={zumErstellen} text={"Fach hinzufügen"} />

        <View style={styles.list}>
            ...
        </View>
        <View>
            <TrennLinie/>
            <Durchschnitt noten={noten} />
        </View>
    </View>
);
}
```

Geschafft! (Fast)

Den Hauptteil der App hast du jetzt geschrieben, das ist schon mal eine sehr starke Leistung. Doch in der Applikationsentwicklung werden Apps und Websites immer **weiterentwickelt und angepasst**. So auch deine App, wir bauen jetzt noch ein paar weitere Features ein, um deiner App einen gewissen **Wow-Effekt** zu geben. Anschliessend haben wir noch spannende Partneraufträge bereit und du kannst deine App anders stylen. Nach dem Abschluss der nächsten Aufgaben meldest du dich einfach bei uns.

Viel Spass 😊

Schritt 16 – Fachdurchschnitt auf der Hauptseite

Wenn wir den Durchschnitt eines Faches sehen wollen, ist es im Moment noch mühsam. Wir müssen jedes Mal auf die Detailansicht wechseln. Wir wollen nun noch den Durchschnitt eines Faches auf der Hauptseite anzeigen können.

Dies bedeutet, wir müssen das «**FachListItem.js**» so bearbeiten, dass es alle Noten des Faches lädt und den Durchschnitt berechnet.

Datei: components/Listen/FachListItem.js

```
export default function FachListItem({fach}) {
  const [noten, notenSetzen] = useState([]);

  useEffect(() => {
    if (!fach) {
      return
    }

    async function LadeNoten() {
      const data = await getNotenByFachId(fach.id)
      notenSetzen(data)
    }
    LadeNoten();
  }, [fach])

  function berechneNotenDurchschnitt() {
    let summe = 0

    for (const note of noten) {
      summe += note.wert * note.gewichtung
    }

    if (summe === 0) {
      return "-"
    }

    let summeGewichtungen = 0
    for (const note of noten) {
      summeGewichtungen += note.gewichtung
    }

    const durchschnitt = summe / summeGewichtungen
    return durchschnitt.toFixed(2)
  }
  ...
  return (
    <View>
      <TouchableOpacity onPress={navigiereZuDetailAnsicht} style={styles.container}>
        <Text style={[styles.text, styles.titel]} numberOfLines={1} ellip-sizeMode="tail">{fach.name}</Text>
          <Text style={styles.schnitt}>{berechneNotenDurchschnitt()}</Text>
        </TouchableOpacity>
      </View>
    )
}
```

Schritt 17 – farbige Markierungen

Bei der vorherigen Aufgabe haben wir die Durchschnitte angezeigt. Diese habe auf der rechten Seite einen farbigen Balken (grün oder rot). Wir wollen jetzt auch noch einen solchen farbigen Balken neben jeder Note platzieren.

Erstelle diese neue Datei im «**components**» Verzeichnis:

```
Datei: components/Balken.js
import {StyleSheet, Text, View} from "react-native";

export default function Balken({wert}) {
    function erhalteFarbeNachNotenWert(note) {
        if (note >= 4) {
            return "green"
        }
        return "red"
    }

    return (
        <View style={[styles.balken, {backgroundColor: erhalteFarbeNachNoten-
Wert(wert)}]}><Text></Text></View>
    )
}

const styles = StyleSheet.create({
    balken: {
        width: "8px",
        height: "100%",
    }
})
```

Verwende die Datei in den beiden anderen Dateien «**NotenListelItem.js**» und «**FachListlItem.js**»

```
Datei: components/Listen/NotenListelItem.js
return (
    <TouchableOpacity onPress={navigiereZuDetailAnsicht} style={styles.con-
tainer}>
    ...
    <Balken wert={note.wert}/>
</TouchableOpacity>
)
```

```
Datei: components/Listen/FachListlItem.js
return (
    <View>
        <TouchableOpacity onPress={navigiereZuDetailAnsicht} style={styles.con-
tainer}>
        ...
        <Balken wert={berechneNotenDurchschnitt()} />
    </TouchableOpacity>
</View>
)
```

Schritt 18 – Wunschnote berechnen

Du kannst nun alle deine Noten festhalten und siehst auch die Durchschnitte der Fächer. Wäre es jetzt nicht auch großartig, wenn du deine Wunschnote berechnen könntest? Nun, zum Glück kann man so etwas einbauen! Wir haben dir dafür bereits eine Komponente namens Wunschnote.js geschrieben.

Die richtige Aufgabe kommt nach diesem Code-Blcok.

Datei: app/fach/[id]/index.js

```
...
    return (
      <View style={styles.container}>
        <StatusBar/>
        <FrageFenster
          ...
        />

        <View style={styles.titleBar}>
          ...
        </View>

        <Knopf beimKlicken={zumErstellen} text={"Note hinzufügen"} />
        <View style={styles.list}>
          <NotenListe noten={noten}/>
        </View>
        <View style={styles.durchschnitt}>
          <TrennLinie/>
          <Durchschnitt noten={noten}/>
        </View>
        <Wunschnote fachId={id}/>
      </View>
    )
  ...
}
```

Du hast jetzt schon viele Male das Prinzip der Komponenten benutzt, jetzt ist es an der Zeit das du dieses Wissen zeigst, bereite dich vor und rufe dann nach einem der Lehrlinge. Du versuchst nach deinen Möglichkeiten zu erklären, was in der Datei «[app/components/Wunschnote.js](#)» passiert und wieso es eine Komponente ist.

Du kannst Googlen oder die Theorie im Ordner «[anleitungen](#)» nutzen.

Schritt 19 – PDF-Export

Jetzt kommen wir zum letzten Feature, das wir für dich vorbereitet haben, es geht darum, dass du die eingetragenen Noten aus der App exportieren kannst, um sie bzw. deinen Eltern zu schicken. Das machen wir über einen PDF-Export. Die technische Umsetzung so einer Funktion ist kompliziert, deshalb befindet sich im **lib-Verzeichnis** deiner App die Datei «**PdfGenerator**». Sieh dir die Datei an und achte dabei auf die **Parameter** der Funktion.

Wir wollen vom «**app/index**» aus auf die **Exportseite navigieren**. Diese existiert bereits, erstelle eine View, deren Style setzt du auf icon. In dieser View erstellst du einen «**IconKnopf**», setze Icon auf «**download-outline**». Der Pfad zum Export lautet «**/export**». Gib in selbst an.

Datei: app/fach/[id]/index.js

```
...
function zumExportieren() {
    router.push("")
}

return (
    <View style={styles.container}>
        ...
        <View style={styles.list}>
            <View style={styles.titel}>
                <Text style={styles.text}>Fächer</Text>
                <View style={styles.icon}>
                    <IconKnopf groesse={40} beimKlicken={} icon="" />
                </View>
            </View>
            <FachList faecher={faecher}/>
        </View>
        ...
    </View>
);
...
}
```

Wenn du auf «**PDF Erstellen**» klickst, **passiert im Moment noch nichts**. Finde heraus wieso und Löse das Problem, indem du die Funktion «**generatePDF(optionen)**» aufrufst.



Tipp: Das Problem befindet sich in der Datei «**app/export.js**»

Wenn du jetzt den Export benutzen kannst, haben wir einen letzten Test. Füge ein **Fach** hinzu das deinen **Vor- und Nachnamen** als Name hat. Füge ausserdem ein paar **Dummy Noten** ein und ein weiteres Fach. Exportiere dann alles und schicke es dann an diese E-Mail.

E-Mail: lukas.buehlmann@bit.admin.ch.

Herzlichen Glückwunsch!



Du hast das gut gemacht 🎉! Ganz egal ob du jetzt fertig bist oder nicht. Du hast eine Menge über Reactnative und JavaScript gelernt. Das wird dir bei deiner zukünftigen Lehre als Informatiker Applikationsentwicklung/EFZ helfen 👍.

Theorie



⚠ Den Code hier musst du nirgendwo einfügen, diese Codeausschnitte sind nur Beispiele.

Components

Eine Komponente (eng. Component) ist ein Teil Code, welcher wiederverwendet werden kann. Z. B. Der Knopf, dieser kann an mehreren Orten eingesetzt werden. Damit mehrfacher Code vermieden und Struktur in den Code gebracht.

TouchableOpacity

Eine «TouchableOpacity» ist einfach ein Bereich auf dem Bildschirm, welcher klickbar gemacht wird. Darin wird der Text definiert, welcher in dieser klickbaren-Region angezeigt werden soll. Das Ganze wird noch orange gefärbt und etwas schöner gestaltet.

useState

```
const [wert, wertSetzen] = useState(startWert)
```

useState ist Funktion, welche uns von React(-Native) zur Verfügung gestellt wird, um Werte zu speichern und zu aktualisieren. Warum verwenden wir nicht einfach eine normale Variable? Das spezielle dran ist, dass wenn wir den Wert eines useState verändern, alle **UI-Komponenten** (Texte, Knöpfe, etc.) **automatisch neu geladen werden**, um die Veränderung anzuzeigen. Probiere doch die Beispiele aus.

Beispiel ohne useState

```
export default function MeineKomponente() {
    let text = «Hallo Welt!»

    function ändereText() {
        text = «Tschüss Welt!»
    }

    return (
        <View style={styles.container}>
            <Knopf beimKlicken={ändereText} text={text}/>
        </View>
    );
}
```

Wie du siehst, **passiert nichts**, wenn du auf den Knopf drückst. Der Text wird zwar neu gesetzt, aber der Text im Knopf wird nicht verändert, da React-Native nicht weiß, dass der Text geändert hat.

Beispiel mit useState

```
export default function MeineKomponente() {
  const [text, textSetzen] = useState("Hallo Welt!")

  function ändereText() {
    textSetzen("Tschüss Welt!")
  }

  return (
    <View style={styles.container}>
      <Knopf beimKlicken={ändereText} text={text}/>
    </View>
  );
}
```

Hier kannst du sehen, dass sich der **Text verändert**, sobald der Knopf angeklickt wird. Das liegt daran, dass React-Native automatisch merkt, dass sich der Text verändert hat und den Text neu auf den Bildschirm schreibt.

useFocusEffect & useCallback

```
useFocusEffect(
  useCallback(() => {
    // Code hier
  }, []))
);
```

useFocusEffect und **useCallback** sind Funktionen, welche uns von React(-Native) zur Verfügung gestellt werden, um automatisch beim Laden einer Seite/Komponente Code auszuführen, wie z. B Daten aus einer Datenbank zu laden.

String Interpolation

```
const name = "Max"
const begrüssung = `Hallo ${name}`
// Ergebnis -> «Hallo Max»
```

Dieser Vorgang wird String Interpolation genannt. Dabei werden zwei Texte verbunden. Achte aber die **«`» (Backtick)**. Du musst den Text in Backticks anstelle von normalen Anführungszeichen schreiben.

Async & Await

```
async function meineFunktion() {
  await IchBinEineFunktionWelcheLangeDauert()
  console.log("meineFunktion ist fertig")
}

meineFunktion()
console.log("Fertig")
```

Async und **await** sind zwei Keywords, welche gebraucht werden, um **Funktionen asynchron zu machen und asynchrone Funktionen abzuwarten**. Normalerweise braucht man asynchrone Funktionen, um Code auszuführen, bei welchem **nicht klar ist, wie lange er dauern wird** (z. B Daten aus einer Datenbank zu laden). Das kann je nach Datengröße variieren, wenige Millisekunden bei 10 Einträgen bis zu mehreren Sekunden bei einigen Millionen Einträgen).

Das Wort «**async**» sorgt dafür, dass der Code im Hintergrund weiterläuft, ohne den Rest des Codes zu blockieren.

Das Wort «**await**» zwingt den Code, auf die Fertigstellung solcher Funktionen zu warten, also z. B setzen wir die Fächer erst, wenn alle aus der Datenbank geladen wurden.

Man kann «**await**» nur in asynchronen Funktionen einsetzen, da wenn man es im Hauptcode machen würde, das ganze Smartphone einfrieren würde.

useEffect

```
useEffect(() => {
    // Code hier
}, [useState hier])
```

useEffect ist eine Funktion, welche ähnlich wie **useFocusEffect** funktioniert. Der Unterschied dazu ist, dass **useFocusEffect** jedes Mal ausgeführt wird, wenn die Komponente neu auf dem Bildschirm angezeigt wird (z. B wenn man herunterscrollt verschwindet ein ListItem der Liste, wenn man wieder hochscrollt, erscheint es wieder und die Funktion wird erneut ausgeführt).

useEffect hingegen funktioniert so, dass es nur ausgeführt wird, wenn die Komponente das erste Mal geladen wird (Bis auf eine Seite, da wird es immer ausgeführt, wenn sie angezeigt wird).

useEffect wird auch ausgeführt, wenn ein **useState** verändert wird, dass im Array angegeben wird.

Beispiel

```
export default function MeineKomponente() {
    const [textBearbeitet, textBearbeitetSetzen] = useState("")
    const [text, textSetzen] = useState("Hallo Welt!")

    useEffect(() => {
        textBearbeitetSetzen ("Text wurde verändert")
    }, [text])

    function ändereText() {
        textSetzen("Tschüss Welt!")
    }

    return (
        <View style={styles.container}>
            <Text>{textBearbeitet}</Text>
            <Knopf beimKlicken={ändereText} text={text}/>
        </View>
    );
}
```

Wenn du diesen Codeausführst, wirst du sehen, dass der Text sich zu «Text wurde verändert» ändert, da das **useEffect** merkt, dass sich das **useState** verändert hat.