



Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

11. März 2025

Anleitung Noten-App

Schnuppertag Applikationsentwicklung 2025

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
✍	Ausfüllen
🧐	Erklärung
💡	Wissen
ℹ️	Info / Tipp
🌟	Optional für Schnelle

Endprodukt

The screenshots show a mobile application interface for managing grades and notes.

Home Screen (08:15): Shows a list of subjects with their average grades. A yellow button at the top right allows adding a new subject.

Fächer	Durchschnitt
Mathematik	4.83
Deutsch	5.00
Französisch	4.25
Englisch	6.00
Docker	5.50
NoSQL	5.40
Geschichte	3.50
Durchschnitt	4.89

Details Screen (08:16): Shows the details for the Mathematics subject. It includes a note creation section, a list of topics with their grades, and a summary section for the subject.

Kategorie	Inhalt	Wert
Dreiecke		5
Algebra		5.5
Planimetrie		4
Durchschnitt		4.83

Note Creation Screen (08:16): A modal for creating a note for the Algebra topic. It asks for the title, note value, and weight, and provides fields for desired grade and number of attempts.

Delete Confirmation Screen (08:16): A modal asking if the user wants to delete a note for Algebra. It shows the note details and provides "Bestätigen" and "Abbrechen" buttons.

Vorbereitung

Umgebung

Öffne auf deinem Laptop das «Terminal», dies erreichst du, indem du, unten links in der Ecke auf das Kreis-Symbol klickst. Suche nach «Terminal» und öffne das Programm. (Das Icon von Terminal ist ein «>_» auf schwarzem Hintergrund). Wenn du das Terminal geöffnet hast, kannst du den Befehl unterhalb dieses Textblocks abtippen und dann Enter drücken. Dieser Befehl holt den Code von unserem GitHub Repository und setzt lokal auf dem Laptop eine Umgebung auf mit der wir arbeiten können. Warte, bis sich das Programm **VS-Code** öffnet. VS-Code ist ein Codeeditor den wir für das Bearbeiten und Schreiben von Programmcode benötigen.

```
curl -L https://raw.githubusercontent.com/timoGeiss/grades-bit/main/install.sh | bash
```

In VS-Code auf der linken Seite siehst du die Ordnerstruktur. Im Ordner «**grades-bit**» hat es zwei Unterordner «**anleitung**» und «**application**». Im Ordner «**anleitung**» befinden sich Anleitungen zu den Programmiersprachen. Im Ordner «**application**» hat es den Ordner «**grades-bit**», darin befindet sich der Code für unsere App.

Es hat mehrere Unterordner, wie etwa «**app**», «**components**» und «**lib**», die anderen Ordner kannst du ignorieren (diese beinhalten z. B Code das React-Native überhaupt funktioniert). Du wirst gleich mehr über die Ordnerstruktur erfahren.

Expo-Go

Verbinde dich mit deinem Smartphone in das WLAN-Netzwerk «**Softwareschmiede**», das Passwort lautet «**work@BIT**». Öffne den App-Store/Play-Store/Galaxy-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 vornehmst, 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.

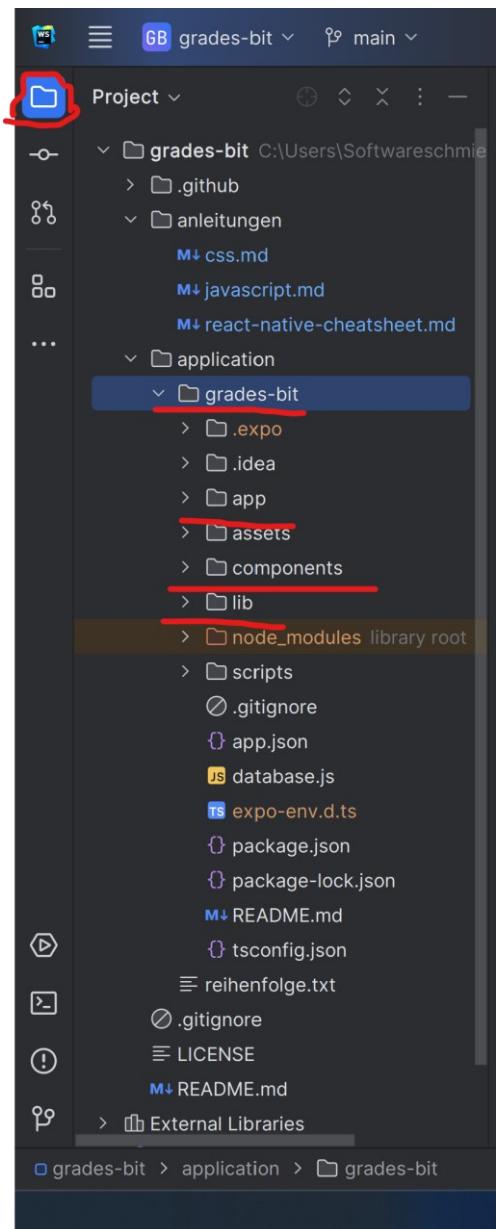
Ordnerstruktur

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).

«**export.js**» ist eine zweite Seite, welche sich auf derselben Ebene wie die Hauptseite «index.js» befindet, dazu später mehr.

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.

Die Bezeichnung «**[id]**» ist etwas speziell, es bedeutet, dass «**[id]**» **dynamisch** verändert werden kann. Also «**fach/[id]/index.js**» wird z. B zu «**fach/1/index.js**» oder «**fach/2/index.js**». So können später Detailseiten für einzelne Fächer angezeigt werden. Wie du vielleicht schon gemerkt hast, hat es in den Ordner «**[id]**» wieder eine «**index.js**» Datei (so wie bei der Hauptseite), diese Dateien sind wieder die Einstiegspunkte der Ordner. Das heisst: Wenn man auf die Detailseite eines Fachs navigiert, wird der Inhalt von «index.js» angezeigt.



Seitenstruktur

Die Ordnerstruktur der Dateien spiegelt in React-Native die Seitenstruktur wider. Ist das nicht cool? Wenn wir das auf eine Website wie die des BITs abbilden, würde das etwa so aussehen:

URL: www.bit.admin.ch/de/kontakt-bit

Datei: de/kontakt-bit/index.js (Also Ordner «de», Ordner «kontakt-bit» und darin die «index.js» Datei)

Was hat es mit den «_layout.js» Dateien auf sich? Diese Dateien definieren das Verhalten und Aussehen der Navigation. Es wird unter anderem definiert, welche Farbe der Titel «Home» im Header auf der Hauptseite hat. **Wir raten dir, die Finger von diesen Dateien zu lassen! (Fehler in «_layout.js» zu beheben kann sehr mühsam sein)**

Happy Coding! 😊

Anleitung

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:

```
<View style={styles.titel}>
  <Text style={styles.text}>Fächer</Text>
</View>
```

Wir verwenden hier das `<Text></Text>` Tag, um einen Text anzuzeigen. Das Tag wird von einem `<View></View>` Tag umgeben. Ein `<View></View>` Tag ist ein Container, welcher nichts bewirkt. Es kann gebraucht werden, um Dinge logisch zu gruppieren und / oder hier in diesem Fall den `style` «**styles.titel**» zu geben. Der `<View>` Tag ist mit einem `<div>` zu vergleichen.

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

Nimm dem `<View></View>` Tag den `style` «**styles.titel**» weg, indem du `styles={styles.titel}` entfernst. Beobachte was passiert.

 **Halte fest:** Was bewirkt `styles={styles.titel}`?

Füge dem `<View></View> styles={styles.titel}` wieder hinzu. Wir wollen ja, dass unsere App schön aussieht 😊.

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:

```
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>  
    );  
}
```

⚠️ **Was passiert hier genau?** Als erstes definieren wir eine Funktion namens «**zumErstellen**», darin wird mit der Zeile «**router.push('/fach/create')**» auf die Erstellseite navigiert. Jedoch nützt uns diese Funktion nichts, wenn sie nicht verwendet wird.

Unten im **<View></View>** Tag wird nun ein neues Tag hinzugefügt: **<Knopf></Knopf>**, dieses Tag werden wir gleich selbst schreiben. Darin sind die beiden Attribute «**beimKlicken**» und «**text**» vorhanden. Im Attribut «**beimKlicken**» wird nun die Funktion «**zumErstellen**» aufgerufen. Das Attribut «**text**» gibt an, welcher Text auf dem Knopf stehen soll.

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

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

⚠️ **Was passiert hier genau?** Hier wird nun eine «**TouchableOpacity**» definiert. 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.

Da wir nun einen Knopf haben, können wir diesen in eine eigene Datei auslagern. **Das Ganze wird dann eine «Komponente» genannt.** Wenn wir diesen Knopf jetzt irgendwo in unserem Code einsetzen möchten, können wir ganz einfach einen Tag schreiben, bei welchem der Name «Knopf» ist. Das ist genau, was du oben (beim Codeblock für die Homepage) bereits gemacht hast!

```
<Knopf beimKlicken={meineFunktion} text={«Mein Text Hier»}/>
```

Das hier ist das **Grundgerüst** für eine eigene Komponente:

```
import {React, StyleSheet, Text} from "react-native";

// Der Name der Funktion definiert den Namen des Tags, also hier: <MeineKomponente/>
// Eigene Komponenten werden direkt wieder geschlossen, es gibt also kein zweiter
Teil wie bei
// <Text></Text>
export default function MeineKomponente({eineVariable}) {
    return (
        <View style={styles.container}>
            // Hier kommen jetzt Tags hin
            <Text>{eineVariable}</Text>
        </View>
    );
}

// Definiert Styling (Farben, Abstände, etc.)
const styles = StyleSheet.create({
    container: {
        // Hier kannst du styling definieren
    }
});
```

✍ **Halte fest:** Was denkst du, warum sollte man Komponenten einsetzen? Notiere mindestens drei Punkte.

```
-  
-  
-
```

Schritt 3 – Fächer hinzufügen

Nun können wir zur Fachhinzufüge Seite mit einem Knopf navigieren. Aber diese ist noch leer. Das Wollen wir nun ändern.



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 useStates 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.

So, da du nun weisst was ein useState ist, wollen wir das Ganze implementieren. Jedoch wollen wir, dass der Name des Faches **mindestens 3 und maximal 20 Zeichen lang** sein darf. Der Code musst du in die Datei «**fach/[id]/create.js**» einfügen.

Schreibe den notwendigen Code noch selbst in die Funktion «formularBestätigt».

💡 **Halte fest:** Wie lauten die beiden If-Bedingungen für den Namen?

💡 **Tipp:** Mit «**name.length**» kannst du auf die Länge des Namens zugreifen. Verwende «**errorSetzen(«text»)**», damit eine Nachricht angezeigt wird, wenn der Namen nicht die Bedingungen erfüllt.

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

    async function formularBestätigt() {
        // Implementiere hier die Logik für die Länge des Namens
        await insertIntoFach(name) // Hiermit wird das Fach gespeichert
        router.back() // Hier wird zurück auf die Hauptseite navigiert
    }

    return (
        <View style={styles.container}>
            <StatusBar/>
            <Text style={styles.bigText}>Welches Fach möchtest du hinzufügen?</Text>
            <Textfeld
                titel={"Name des Fachs"}
                inhalt={name}
                wennInhaltVerändertWird={(neuerInhalt) => nameSetzen(neuerInhalt)}
                platzhalter={"Mathematik"}
            />

            {error ? <Text style={styles.error}>{error}</Text> : null} // Hier werden Fehler angezeigt
            <Knopf beimKlicken={formularBestätigt} text={"Bestätigen"} />
        </View>
    )
}
```

Schritt 4 – Fächer anzeigen

Grossartig, jetzt können wir Fächer hinzufügen! Unsere App soll schliesslich auch Fächer anzeigen können. Was brauchen wir, um Fächer anzeigen zu können. Genau! Eine Liste von Fächern.



Ternary Operator

```
Bedingung ? Wahr : Falsch
```

Der Ternary Operator ist eine verkürzte Schreibweise für eine *If-Verzweigung*. Es wird eine Bedingung formuliert, welche entweder *True* oder *False* sein kann. Wenn die Bedingung zu *True* ausgewertet wird, wird der Code hinter dem Fragezeichen (?) ausgeführt, sonst hinter dem Doppelpunkt (:).

Füge diesen Code in die Datei «**FachList.js**» ein. Im Ordner «**components**». Platziere den Code am richtigen Ort in der Datei.

renderItem Funktion

```
function renderItem({item}) {
    return <FachListItem fach={item}/>
}
```

HTML für die Liste

```
{faecher.length > 0 ?
  <FlatList
    data={faecher}
    renderItem={renderItem}
    keyExtractor={(fach, index) => index.toString()}
  />
  :
  <Text></Text>}
```

Aktuell werden die Elemente der Liste noch ungetrennt untereinander angezeigt. Das wollen wir ändern, es gibt ein weiteres Attribut für die Flatlist, um eine eigene Trenn-Komponente einzufügen. Versuche eigenständig das Attribut zu finden und setze es «Trennlinie».



Tipp: Das Attribut «separiert» die einzelnen «Items»



Halte fest: Wie lautet das Attribut, welches die Trennlinie in der Flatlist angibt.

Jetzt haben wir zwar eine Liste aber noch kein **FachListItem** um das zu ändern kannst du diesen Code in der Datei «**FachListItem.js**», einfügen:

```

export default function FachListItem({fach}) {
    return (
        <View>
            <Text style={[styles.text, styles.titel]} numberOfLines={1} ellip-
sizeMode="tail">{fach.name}</Text>
        </View>
    )
}

```

Super! 🎉 Jetzt haben wir eine funktionierende Fachliste sowie ein FachListItem. Spätestens jetzt sollte klar werden, wofür Komponenten gebraucht werden: FachListItem ist eine Komponente, da wir ja mehrere Fächer haben und die Namen der Fächer dynamisch angezeigt werden sollen.

Die Fächer werden jetzt zwar gespeichert und können auch angezeigt werden, aber das werden sie nicht? **Warum?** Die Fächer werden gar nicht geladen, somit können sie auch nicht angezeigt werden. Füge den Code im «**index.js**» hinzu.

```

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

    useEffect(
        useCallback(() => {
            async function getFaecher() {
                const daten = await getAllFaecher()
                setFaecher(daten)
            }

            getFaecher()
        }, [])
    );
}

return (
    <View style={styles.container}>
        <View style={styles.list}>
            <View style={styles.titel}>
                <Text style={styles.text}>Fächer</Text>
            </View>
            <FachList faecher={faecher}/>
        </View>
    );
}

```

🧐 **Was passiert hier genau?** Zuerst oben definieren wir eine «**useState**» namens «**faecher**», welches zu Beginn eine Leere Liste ist. Darunter wird ein «**useEffect**» eingesetzt, um beim Laden der Seite automatisch alle Fächer zu laden und diese in den «**useState**» «**faecher**» zu speichern. Im HTML wird die «**FachList**» benutzt, um alle Fächer in Form einer Liste anzuzeigen.



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.



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.

Zerbrich dir nicht den Kopf, falls du das noch nicht ganz verstehst. Die Funktionsweise von `async` und `await` zu verstehen ist für unsere Noten-App nicht wichtig (Korrekt eingesetzt muss es trotzdem werden, sonst könnte es sein, dass die Fächer nicht richtig geladen werden) 😊.

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 Platzhalter, wir müssen diesen von Hand ersetzen.

Füge diesen Code in die Datei «FachListItem.js» ein. Im Ordner «components». Platziere den Code am richtigen Ort in der Datei.

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

Da wir jetzt die Funktion korrekt geschrieben haben, müssen wir sie auch noch irgendwo im Code verwenden.

 **Halte fest:** Welches HTML-Tag benötigen wir, um einen Bereich auf dem Bildschirm klickbar zu machen? Notiere auch das Attribut, welches benötigt wird, um Code bei einem Klick auszuführen.

Implementiere nun den benötigten Code. Ersetze «TagX» durch das richtige Tag, sowie «AttributX» durch das Attribut, welches benötigt wird.

```
export default function FachListItem({fach}) {
  // Hier ist noch sonstiger Code

  return (
    <View>
      <TagX AttributX={funktionHier} style={styles.container}>
        <Text style={[styles.text, styles.titel]} numberOfLines={1} ellip-
sizeMode={"tail"}>{fach.name}</Text>
      </TagX>
    </View>
  )
}
```

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. Du kannst den Code in die Datei «/fach/[id]/create.js» einfügen.

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

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

      async function FaecherLaden() {
        // Schreibe hier den Code damit das Fach geladen wird.
      }

      FaecherLaden()
    }, [id])
  );

  return (
    <View style={styles.container}>
      <StatusBar/>
      <View style={styles.titleBar}>
        <Text style={styles.titel} numberOfLines={1}>Name des Fachs</Text>
        // Lies hier den Namen des Faches aus, das hast du schonmal gemacht.
        </View>
      </View>
    )
}
```

💡 **Was passiert hier genau?** Zuerst oben lesen wir die Id aus dem Pfad («/fach/[id]») aus, damit wir wissen, um welches Fach es sich handelt. Das «useState» «fach» wir noch leer gesetzt, damit es später mit den Fachdaten gefüllt werden kann. Im «useEffect» werden die Daten des Faches geladen und in das «useState» «fach» gesetzt. Im HTML wird ein Text definiert, welcher dafür sorgt, dass der Name des Faches angezeigt wird.

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, sieh ihn dir unter «**/components/Eingaben**» an.

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.

```
async function fachLöschen() {
    const noten = await getNotenByFachId(id)
    for (const note of noten) {
        await removeNote(note.id)
    }
    await removeFach(id)
    router.back()
}
```

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 gemerkt, wofür das **icon** Attribut verwendet wird. Mit diesem Attribut bestimmen wir, wie unser Knopf aussehen soll, wir verwenden hier passend «**trash**».

```
return (
    <View style={styles.container}>
        <StatusBar/>
        <View style={styles.titleBar}>
            <Text style={styles.titel} numberOfLines={1}>{fach.name}</Text>
            <View style={styles.icons}>
                <IconKnopf beimKlicken={fachLöschen} icon={"trash"} />
            </View>
        </View>
    </View>
)
```



Achtung: eigentlich würde unser Löschen so funktionieren, es ist jedoch üblich vor einer nicht umkehrbaren Aktion eine Bestätigung vom Benutzer anzufordern.

Um eine Bestätigung einzubauen, hängen wir noch eine zusätzliche Funktion zwischen dem Knopf und der Funktion «**fachLöschen()**». Definiere eine neue Funktion mit dem Namen «**löschenBestätigen**», in dieser Funktion wird unsere Abfrage stattfinden.

 **Überlege:** schreibe auf, welche Änderungen du am Löschevorgang vornehmen musst und wie du die Benutzerabfrage lösen kannst (du darfst auch Dr. Google fragen 😊). Wenn du fertig bist, kannst du dich bei einem Lernenden melden.

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.

Damit wir eine Möglichkeit zum Navigieren haben, brauchen wir wieder einen Knopf. Was würde sich schon besser als ein IconKnopf eignen, richtig ein zweiter IconKnopf.

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



Tipp: Der Pfad zur Bearbeitungsseite lautet: «**fach/[id]/edit**». Vergiss nicht, die Id zu ersetzen 😊.

Schritt 9 – Farbe der Icons

Jetzt haben wir zwei Icons auf unserer Fach-Detail Seite, aktuell sind die beiden schwarz gefärbt. Um dem Style unserer Applikation gerecht zu werden, wollen wir die Farbe auf Orange wechseln. Später kannst du selbst bestimmen, wie deine App aussehen soll. Fürs Erste ändere die IconKnopf Komponente so, dass die Icons orange werden.

Schritt 10 – Fachbearbeitungsseite

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

💡 **Halte fest:** Zuerst überlegen wir uns, was genau passiert, wenn ein Benutzer den Namen eines Faches bearbeitet. Ordne den Ablauf in der richtigen Reihenfolge an.

Daten werden geupdated, Stift anklicken, Existierende Daten werden geladen, zurück zur Detailansicht navigieren, Name wird validiert, Benutzer verändert Namen, Benutzer klickt auf Speichern



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 (in den []) angegeben wird.

Beispiel mit useState

```
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 Code ausführst, wirst du sehen, dass der Text sich zu «Text wurde verändert» ändert, da das **useEffect** merkt, dass sich das **useState** verändert hat.

Da du nun weißt was ein **useEffect** ist, können wir das ganze implementieren.

```

export default function Edit() {
    const {id} = useLocalSearchParams()
    const [fach, fachSetzen] = useState({})
    const [error, errorSetzen] = useState(null)

    useEffect(() => {
        if (!id) {
            return;
        }

        async function FachLaden() {
            // Implementiere Code um Fach zu laden
        }

        FachLaden()
    }, [id]);

    async function formularBestätigt() {
        // Implementiere Validation für die Länge des Namens
        // Speichere das Fach in der Datenbank. Tipp: updateFach(Id, Name)
    }

    function abbrechen() {
        // Implementiere Code beim Abbrechen, navigiere zurück
    }

    return (
        <View style={styles.container}>
            <StatusBar/>
            <Text style={styles.bigText}>Titel 123</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}
            {/* Implementiere noch Knöpfe */}
        </View>
    )
}

```

 **Was passiert hier genau?** Zuerst oben lesen wir die Id aus dem Pfad (`«fach/[id]/edit»`) aus, damit wir wissen, um welches Fach es sich handelt. Das `useState «fach»` wir noch leer gesetzt, damit es später mit den Fachdaten gefüllt werden kann. Im `useEffect` werden die Daten des Faches geladen und in das `useState «fach»` gesetzt. Im HTML wird ein Textfeld definiert, damit man den Text bearbeiten kann.

Das Ganze ist aber noch nicht vollständig, schau den Code an und implementiere noch die fehlenden Dinge.

Schritt 11 – Noten auf der Fach-Ansicht anzeigen

Nun haben wir alle wichtigen Funktionen für Fächer implementiert, jetzt wird es Zeit, das Ganze für Noten ebenfalls durchzuführen. Die Notenübersicht machen wir auf der Detailseite vom Fach, da wir alle Noten nach Fächern geordnet ansehen wollen.

Das Erstellen der Liste wirst du jetzt selbstständig machen. Die Vorgehensweise ist dieselbe wie bei der Fachliste, du kannst nach «**copy and adapt**» vorgehen (kopieren und anpassen).

Für das Item der Liste verwendest du NotenListItem, welches sich im gleichen Verzeichnis befindet wie deine NotenListe. Den HTML-Code fügst du zwischen den Views ein.

```
function navigiereZuDetailAnsicht() {
    // folgt später
}

return (
    <TouchableOpacity onPress={navigiereZuDetailAnsicht} style={styles.container}>
        <Text style={[styles.text, styles.titel]} numberOfLines={1} ellipsisMode={"tail"}>{note.titel}</Text>
        <Text style={[styles.text, styles.wert]}>{note.wert}</Text>
    </TouchableOpacity>
)
```

Wenn du die Komponente geschrieben hast, kannst du sie in der Übersicht einfügen. Gehe also zum index von Fach und füge dort einen Aufruf der Liste ein, die Noten musst du natürlich auch mitgeben. Für die Noten geben wir den useState an, der anfangs die Noten speichert.

```
<View style={styles.list}>
    <NotenListe noten={noten}/>
</View>
```

Im Moment sollte der alternative Text angezeigt werden, z. B. «Keine Noten». In Schritt 13 werden wir erst Noten hinzufügen. Wenn du deine Lösung kontrollieren möchtest, darfst du natürlich jemanden von uns rufen.

Schritt 12- Note hinzufüge Knopf

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

Verwende diese Zeile Code, um zu Erstellseite zu navigieren:

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

Wir müssen dem Pfad noch mit «**?id=id**» die id mitgeben, da wir ja beim Erstellen einer Note wissen müssen, für welches Fach wir die Note erstellen wollen. Warum mussten wir das vorher nicht machen? Da ReactNative bereits selbstständig weiß, dass mit dem Angeben von [id] im Ordnernamen die Id von dort genommen werden muss. Hier jedoch befindet sich die Datei «create.js» nicht in einem Ordner namens [id].

Schritt 13 – Notenhinzufüge Seite

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

- Text: «Was für eine Note möchtest du erstellen?»
- Textfeld für den Titel der Prüfung
- Zahlenfeld für die Note
- Zahlenfeld für die Gewichtung
- Errorfeld
- Bestätigungsbutton

Bedingungen für die Validierung:

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

Verwende diesen Code hier, um die Note in der Datenbank zu speichern. Es werden noch einige Validierungen auf die Note und Gewichtung angewendet, um sicherzustellen, dass nur «.» bei Kommazahlen erlaubt sind.

```
const überprüfteNote = Number(note)
if (isNaN(überprüfteNote)) {
    errorSetzen("Die Note muss eine Zahl sein (Nur . erlaubt kein ,)")
    return
}

const überprüfteGewichtung = Number(gewichtung)
if (isNaN(überprüfteGewichtung)) {
    errorSetzen("Die Gewichtung muss eine Zahl sein (Nur . erlaubt kein ,)")
    return
}

await insertIntoNote(id, titel, überprüfteNote, überprüfteGewichtung);
router.back()
```



Tipp: Du benötigst vier useState:

- titel Anfangswert: «»
- note Anfangswert: 0
- gewichtung Anfangswert: 0
- error Anfangswert: null

Schau bei der Fachdetailseite, wie du die Id aus dem Pfad herausliest.

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 dieselben Bedingungen für die Validierung der Note wie beim Erstellen.

Du kannst diesen HTML-Code als Vorlage für die Detailseite verwenden.

```
export default function Index() {
    // Id auslesen

    useEffect(
        useCallback(() => {
            if (!id) {
                return
            }
            // Funktion Note laden
            // Funktion Note laden aufrufen
        }, [id])
    );

    async function formularBestätigt() {
        // Validierung der Note
        // Note speichern
        // Zurück navigieren
    }

    if (!note) {
        return
    }

    return (
        <View style={styles.container}>
            <StatusBar/>
            <Text style={styles.bigText}>Was für eine Note möchtest du erstellen?</Text>
            <Textfeld
                titel={"Titel der Prüfung"}
                inhalt={note.titel}
                wennInhaltVerändertWird={(neuerInhalt) => noteSetzen({...note, titel: neuerInhalt})}
                platzhalter={"Neuer Name"}
            />

            {error ? <Text style={styles.error}>{error}</Text> : null}
            <Knopf beimKlicken={formularBestätigt} text={"Bestätigen"} />
        </View>
    )
}
```

Nun kommt noch eine Kleinigkeit dazu: Wir wollen auch Noten löschen können. Füge eine **View** mit dem styles «**styles.löschen**» hinzu. Darin soll es einen Text mit style «**styles.bigText**» und Text «**Prüfung Löschen?**» sowie einen Knopf mit dem Text «**Löschen?**» haben.

Weisst du noch, als wir ein Fach gelöscht haben? Verwende dasselbe Prinzip hier, um eine Note zu löschen. Baue ebenfalls eine Bestätigung ein.

Finde selbständig heraus wie du eine Note löschen kannst.

Schritt 15 – Durchschnitt anzeigen

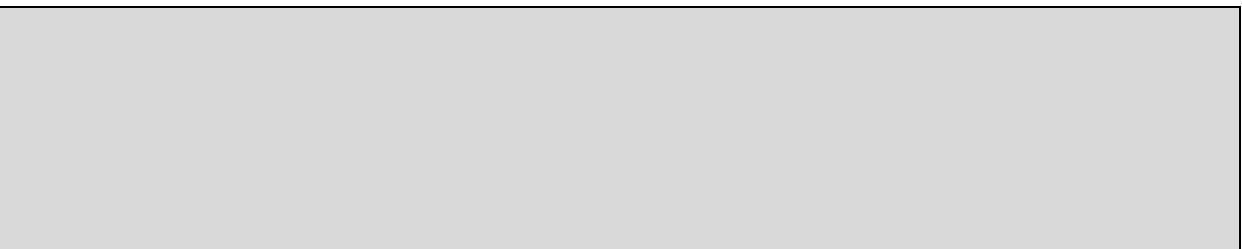
So, jetzt haben wir alles, was es unbedingt braucht. Es wäre aber schön, wenn man sehen könnte, wo man steht. Deshalb werden wir jetzt noch den Durchschnitt berechnen und anzeigen. Einmal auf der Fachdetail Ansicht und einmal auf der Homepage.

Im Components Ordner gibt es die Komponente «**Durchschnitt**», diese wirst du jetzt schreiben. Du bekommst von oben alle Noten, die du zum Rechnen benutzen kannst. Dieses Resultat zeigst du auf gleiche Weise an wie das NoteListitem.

```
<View style={styles.container}>
    <Text style={styles.text}>Durchschnitt</Text>
    <Text style={[styles.text, styles.wert]}>{</Text>
</View>
```

In den geschweiften Klammern wird der Funktionsaufruf für die Rechnung geschrieben. Definiere eine Funktion mit dem Namen «**BerechneNotenDurchschnitt**».

💡 **Formel:** Jetzt musst du überlegen, wie der Durchschnitt ausgerechnet wird, beachte dabei, dass wir die Gewichtungen auch miteinbeziehen müssen



Tipp: <https://avidii.ch/blog/notendurchschnitt-berechnen>

Mit der Formel, die du geschrieben hast. Können wir die Funktion «**BerechneNotenDurchschnitt**» nun schreiben. Solltest du nicht weiterkommen, nimm die JavaScript Doku zur Hand (unter Anleitungen). Wenn es weiterhin Probleme gibt, sag uns Bescheid.

Den Durchschnitt kannst du jetzt auf beiden Seiten (Home, Fach) einfügen. Fangen wir beim Fach an:

```
<View style={styles.list}>
    <NoteListe noten={noten}/>
</View>
<View style={styles.durchschnitt}>
    <TrennLinie/>
    <Durchschnitt noten={noten}/>
</View>
```

Das Gleiche kannst du jetzt im Home machen.

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 – Durchschnitt färben

Jetzt haben wir grossartige Durchschnitte, es wäre aber noch cooler, wenn man auf den ersten Blick erkennen könnte, wie gut man ist. Deswegen möchten wir jetzt noch Farben ins Spiel bringen. Es gibt mehrere Orte, an denen wir die Noten einfärben können:

NotenListItem.js

Füge der ToucheableOpacity im NotenListItem.js diese Zeile hinzu:

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

Diese Zeile erzeugt ein Element, das aussieht wie ein Balken. Dem Balken geben wir die Hintergrundfarbe vom Rückgabewert der Funktion «**erhalteFarbeNachNotenWert()**»

Schreibe die Funktion «**erhalteFarbeNachNotenWert()**» so dass es folgende Farbstufen gibt:

- Genügend: green
- Ungenügend: red

Du darfst gerne auch noch selbst weitere Farbstufen hinzufügen 😊

Durchschnitt.js

Füge dem View im «**Durchschnitt.js**» diese Zeile hinzu:

```
<View style={[styles.balken, {backgroundColor: erhalteFarbeNachNotenSchnitt(berech-  
neNotenDurchschnitt())}]}><Text></Text></View>
```

Diese Zeile erzeugt ein Element, das aussieht wie ein Balken. Dem Balken geben wir die Hintergrundfarbe vom Rückgabewert der Funktion «**erhalteFarbeNachNotenSchnitt (durchschnitt)**»

Schreibe die Funktion «**erhalteFarbeNachNotenSchnitt (durchschnitt)**» so dass es folgende Farbstufen gibt:

- Genügend: green
- Ungenügend: red

Du darfst gerne auch noch selbst weitere Farbstufen hinzufügen 😊

Ist dir da gerade etwas aufgefallen? Du hast zweimal fast genau dasselbe gemacht. Dies nennt man **Redundanzen**. **Redundanzen sind nicht gut.** Redundanzen sind zumindest als Applikationsentwickler nicht gut. Applikationsentwickler sind faul, wir wollen nicht mehrmals denselben Code schreiben.

Füge nun noch die Farbstufe «yellow» von 4 bis 4.5 hinzu.

Siehst du, du musstest jetzt die ganze Arbeit zweimal machen. Das bedeutet, die **Wartbarkeit** deines Codes sinkt. Wenn du jetzt noch eine weitere Farbstufe an einem anderen Ort hinzufügen willst, musst du das gleich dreimal machen. Wenn das so weitergeht, weisst du nicht mehr genau, wo du welche Farbstufen eingebaut hast.

Lagere die beiden Funktionen «**erhalteFarbeNachNotenWert**» und «**erhalteFarbeNachNoten-Schnitt**» in eine Datei namens «**NotenFarbe.js**» im Ordner «**lib**» aus (Schreibe eine Funktion in dieser Datei, damit du von Durchschnitt.js und NotenListItem.js aus auf diese Funktion zugreifen kannst).

Das würde dann etwa so aussehen:

```
export function erhalteFarbe(wert) {  
    // Code hier  
}
```

Wenn du möchtest, kannst du jetzt noch **einen Schritt weitergehen** und auch noch das **HTML auslagern (Du würdest dann eine Komponente «Balken.js» im «components» Ordner schreiben, anstelle einer Datei im «lib» Ordner)**. Du kannst ja nachschauen, wie man seine eigene Komponente schreibt, orientiere dich an diesem Aufbau. Das styling der Komponente musst du auch übernehmen. Melde dich doch bei einem der Lehrlinge, wenn du dies geschafft hast. Styling:

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

Übrigens: «lib» steht für «Library» und wird normalerweise dazu verwendet, um gemeinsame Dateien zu speichern wie eben «**NotenFarbe.js**»

Schritt 17 – Modal anstelle von Alert

Was ist ein Modal? Ein Modal ist eigentlich genau das, was wir bereits mit Alert machen, wenn wir ein Fach oder eine Note löschen. Wir fragen den Benutzer, ob er das wirklich will. Dazu verwenden wir ein kleines Fenster, welches wie ein Pop-up aufpopt. Nun wollen wir aber ein eigenes Fenster erstellen.

Wir haben dir die Hälfte der Arbeit schon abgenommen. Öffne und studiere mal die Datei «**FrageFenster.js**» im Ordner «**Eingaben**» im Ordner «**components**».

Überlege dir, wie du das ganze Einbauen müsstest. Welche Attribute müssen gesetzt werden? Was sind die Attribute für Attribute: Texte, useState, Funktionen? Spiele etwas damit herum, bis du verstehst, wie es funktioniert.

 **Halte fest:** Hier kannst du Notizen festhalten.

Versuche nun das FrageFenster in den Dateien «**fach/[id]/index.js**» und «**note/[id]/index.js**» zu implementieren.



Tipp: Soll dein Code nichts machen, wenn abbrechen geklickt wird? Verwende:

```
(() => {})
```

Schritt 18 – Wunschnote berechnen

Du kannst nun alle deine Noten festhalten und siehst auch die Durchschnitte der Fächer. Wäre es jetzt nicht auch noch toll, wenn du deine Wunschnote berechnen könntest? Nun, zum Glück kann man so etwas einbauen!

 **Formel:** Notiere hier die Formel, welche du brauchst, um deine Wunschnote zu berechnen. Verwende z.B x als unbekannte für die Wunschnote. (Die Formel hier wirst du später nicht mehr brauchen, also keine Sorge).

Implementiere den benötigten Code in der Datei «**Wunschnote.js**» im Ordner «**components**»

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 **IconKonpf**, setze Icon auf «**download-outline**». Der Pfad zum Export lautet «**/export**».

Wenn du jetzt auf die Seite navigieren kannst, fangen wir mit dem Auswahlverfahren an. Wir wollen wählen können, welches Fach exportiert wird oder gleich alle exportieren. Dazu brauchen wir ein paar Dinge.

- Ein **UseEffect** zum holen aller Fächer (**getAllFaecher()**)
- Ein **useState** zum Speichern dieser Fächer
- Ein **useState** der das ausgewählte Fach speichert
- Einen **Picker** zum Auswählen eines Fachs
- Ein **Knopf** zum Bestätigen (Export auslösen).

 **Tipp:** Lies die Doku des Pickers durch: <https://www.npmjs.com/package/@react-native-picker/picker>

 **Tipp:** Du kannst die Fächer im Picker wie folgt anzeigen:

```
{fächer.map((item) => {
  return (
    <Picker.Item label={item.name} value={item.id.toString()} key={item.name}/>
  )
})}
```

 **Tipp:** verwende Lambda-Annotation, um Hooks und Funktionen aufzurufen, denen du einen Wert mitgibst

```
beimKlicken={() => generatePDF(option)}
```

```
onValueChange={(itemValue) => {
  optionSetzen(itemValue)
}}
```

Versuch selbst mit diesen Angaben eine Auswahl zu bauen. Wenn du nicht weiter weisst, dann Google am besten mal zuerst oder rufe uns.

Um alle Fächer zu exportieren, musst du lediglich der Funktion generatePDF eine 0 als Wert mitgeben. Wir möchten das man im Picker «alle Noten» wählen kann. Derselbe Picker der die Fächer anzeigt.

Viel Glück 

Herzlichen Glückwunsch!



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