LAMA - Lerne alles mit Anna

Entstanden im Rahmen des Softwaretechnik-Projekts im Sommersemester 2021 unter Leitung von Professor Kammer

Weiterentwicklung

Team:

Dario Pläschke
Franz Leonhardt
Kevin Binder
Lars Kammerer
Tobias Rentsch
Fabian Brecher
Vinzenz Branzk
Florian Silber

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Inhaltsverzeichnis

Einleitung

Für die angestrebte Weiterentwicklung der LAMA-Lernapp soll dieses Dokument als Grundlage dienen. Darin werden wesentliche Schnittstellen für die Weiterentwicklung erklärt und beschrieben, die eine schnelle Einarbeitung für neue ContributorInnen ermöglichen soll. Das zeitaufwändige studieren der Codestruktur und Architektur soll damit minimiert werden.
Neben den Schnittstellen werden noch nicht vollendete, geplante und neue Feature-Ideen vorgestellt, die als Ausgangspunkt für die weitere Entwicklung dienen sollen. Diese kamen durch die Entwickler während der Umsetzung sowie auch von externen Testern.
Gerade bei der Entwicklung einer so umfangreichen App kam und kommt mehr und mehr Feedback von unterschiedlichen Seiten, die zeigen, dass es noch viel Potential auszuschöpfen gilt.

Zukünftige Features

Avatare

In der aktuellen Version von LAMA gibt es lediglich ein Profilbild für den/die AdministratorIn und eins für den/die BenutzerIn. Das Einfügen weiterer Avatare könnte hierbei angestrebt werden, damit NutzerInnen sich somit ihren Account personalisieren könnten. Ebenso wären Avatare, welche nicht von Anfang an freigeschaltet wären, zusätzliche Anreize für das Lernen.
Denkbar für Avatare wären drei verschiedene Systeme:

1. Anpassbarer Avatar

Hierbei wäre es möglich verschiedene Grundformen auszuwählen. So könnten dies einerseits z.B. verschiedene Tiere (Katze, Affe, …) oder andererseits verschiedene Lamaformen sein. Wurde sich für eine Form entschieden, so wäre das Hinzufügen verschiedener Accessoires möglich. Eine Option hierbei wäre es z.B. dem Lama verschiedene Brillen oder Hüte aufzuziehen, die verschieden kombinierbar wären. Auch größere Objekte, wie ein Astronauten- oder Motorradhelm wären denkbar. Des Weiteren könnte es möglich sein, den Gesichtsausdruck des Avatars zu verändern. Ein essentielles Feature hierbei wäre die Auswahl der eigenen Farbe. Das Ergebnis wäre z.B. eine Bestimmung der Fellfarbe oder die der Accessoires.

Hintergründe könnten ebenfalls wechselbar sein. Hierbei sollte die Hintergrundfarbe anpassbar sein. Eine Steigerung davon wäre es, dem Hintergrund ein Muster zu geben wie z.B. gestreift oder kariert.
Neben den variablen Hintergründen ist auch denkbar feste Hintergründe zu erstellen. Diese könnten z.B. im Weltraum oder am Strand sein, jedoch benötigt dies einen erheblichen Design-Aufwand.

2. Unveränderbarer Avatar

Eine weitere Option wäre es mehrere Avatare einzufügen, welche allerdings nicht veränderbar sind. So könnte z.B. ein Roboter, Lamas mit einzelnen Accessoires oder auch andere Designs eingefügt werden. Einer der Vorteile dieser Version wäre, dass neue Avatare nicht immer dynamisch designed werden müssen. Besondere Avatare könnten in zukünftigen Updates eingefügt werden ohne diese mit allen anderen anpassbaren Optionen abstimmen zu müssen. Die veränderbaren Hintergründe könnten genauso wie bei der ersten Option umgesetzt werden.

3. Eigener Avatar

Die letzte angedachte Möglichkeit wäre es, den NutzerInnen zu erlauben eigene Avatare in die App hochzuladen. Dies kann allerdings zu großen Problemen führen, da so sehr schnell unangemessene Inhalte in die App gelangen könnten und dies nicht mit der Altersfreigabe vereinbar wäre. Sollte es in der Zukunft eine Web-Anbindung geben, in welcher Avatare anderer BenutzerInnen sichtbar sind, könnte dies zu gravierenden Schwierigkeiten führen. Zudem bräuchte die App zusätzliche Berechtigungen, um auf den Speicher zugreifen zu können.
Alles in allem ist dieser Ansatz nicht zu empfehlen.

Avatar Menü

Das Avatar-Menü könnte hierbei im Hauptmenü verortet sein. Wichtig ist, dass die SchülerInnen eigenständig ihren Avatar ändern können und dies nicht nur im Adminmenü möglich ist. Wenn im Hauptmenü auf sein Avatar geklickt wird, so könnte dies das Auswahlmenü öffnen. Im Folgenden könnten die zur Verfügung stehenden Avatare in einem scrollbaren Fenster begutachtet werden und mit einem klick auf einen der Avatare, wird dieser ausgewählt und das Fenster geschlossen. Bei einem klick auf den Hintergrund, links oder rechts, wird in das Hauptmenü zurück navigiert.
Bei den unveränderbaren Avataren könnten z.B. zuerst der Avatar ausgewählt werden und im Anschluss der Hintergrund. Wird hierbei der Avatar ausgewählt und im Auswahlmenü auf

eine der Seiten geklickt, so wird mit dem neuen Avatar und einem standard blauen Hintergrund zurück ins Hauptmenü navigiert.

Einbindung

Sollte das Avatar Menü, wie oben erklärt im Main-Menü als Popup umgesetzt werden, so müsste es dort auch innerhalb des Codes eingefügt werden (.../lib/app/screens/home_screen.dart). In der Datenbank wurden schon Einträge für Informationen bezüglich des Avatars hinterlegt. Dabei existieren bereits Methoden wie “userRepository.getAvatar()” um den entsprechenden Pfad zum Avatar aus der Datenbank zu erhalten. Die Avatar-Auswahl müsste hier lediglich an die gegebenen Werte angepasst werden. Die SVG-Dateien der Avatare würden hier innerhalb der flutter-assets gespeichert werden, um sie lokal zur Verfügung stellen zu können. Schlussendlich ist dann nur noch der Avatar zu laden, welcher zu dem hinterlegten Wert in der Datenbank passt.

Skins

Ähnlich zu den vorher erwähnten Avataren, könnten sogenannte Skins für die Spiele oder dem Maskottchen Anna eingefügt werden. So könnte Anna im Hauptmenü verschiedene Kostüme anziehen, wie z.B. eine Weihnachtsmütze. In den Spielen könnten Skins wie folgt realisiert werden:

Snake:

- Äpfel ersetzen durch z.B. Orangen
- Farbe/Muster der Schlange verändern
- Zug statt einer Schlange

FlappyLama:

- Anna in einem kleinen Flugzeug
- Pegasus-Lama
- keine Kakteen sondern Hochhäuser

Affenklettern:

- Koalabär statt Affe
- Hochhaus statt Baum
- Weltraum statt Wolken

Implementierung

Für die Skins müssten in der Datenbank bei einem BenutzerIn noch zusätzliche Werte eingefügt werden. So müsste z.B. der ausgewählte Skin für Flappy Lama hinterlegt sein. Sollten die Skins für das begleitende Maskottchen verändert werden, so müssten diese ebenfalls in der Datenbank hinterlegt sein. Zudem wird Anna standardmäßig im Hauptmenü geladen. Dies müsste folglich geändert werden, zum Beispiel mit einer Methode wie “userRepository.getCostume()”, um den Zugriff auf das richtige Kostüm zu ermöglichen. Der jeweilige Skin sollte dann, wie bei den Avataren, aus den flutter-assets geladen werden.

Freischaltbares

In der aktuellen Version von Lama sind sämtliche Features von Anfang an enthalten. Werden allerdings Avatare oder Skins eingefügt, die erst freigeschaltet werden müssen, würden weitere Felder in der Datenbank benötigt werden, welche anzeigen ob ein

Gegenstand verwendbar ist oder nicht. Diese Freischaltung würde einmalig beim Erwerb oder bei Freispielen erfolgen und den entsprechende Boolean Wert auf true setzen.

Achievement-System

Ein Achievement-System (deutsch “Erfolgs-System”) ist ein in der Videospielwelt weit verbreitetes Feature. Hierbei werden dem/der NutzerIn verschiedene “Achievements” für erbrachte Leistungen verliehen, welche er/sie sich im Achievement-Menü ansehen kann. Achievements können entweder einfach nur einen Fleiß-Fortschritt belohnen (Löse x Mathe Aufgaben), eine außergewöhnliche Leistung (Erreiche x Punkte in Affenklettern), Sammel-Fortschritte (Erhalte x Achievements) oder einfach nur spaßige Easter-Eggs (bekomme jeden Lama Fun-Fact von Anna einmal präsentiert) darstellen. “Hidden”-Achievements sind ebenfalls eine Option. Sie würden mit “???” im System angezeigt werden und erst aufgedeckt, wenn sie erfüllt wurden. So könnte z.B. ein Achievement einfügt werden, in welchem 10 mal auf Anna im Hauptmenü gedrückt werden muss. Dies könnte dazu führen, dass NutzerInnen angeregt werden verschiedener Sachen auszuprobieren. Zusätzlich könnte dadurch potentiell mehr über das Spiel gesprochen werden. (Videos über “lustige” oder versteckte Erfolge sind keine Seltenheit!)
Wird ein Achievement erreicht, so sollte der/die NutzerIn eine kleine Meldung am Rande des Bildschirms erhalten, um zu sehen, welche Errungenschaft gerade freigeschaltet wurde.

Was bringen Achievements?
Achievements bringen einen Anreiz für den/die NutzerIn weiter zu spielen. Sei es nun die Jagd nach Highscores, dem komplettieren einer Sammlung oder nur der Spaß dabei herauszufinden, welche Achievements noch versteckt sein könnten.
Ebenso ist es dadurch möglich, einzelne Belohnungen unabhängig von den Aufgaben zu verteilen. So könnten Achievements einerseits den/die NutzerIn mit Lama-Coins belohnen, andererseits könnte das ganze mit den vorherig erwähnten Skins und Avataren kombiniert werden. Es entsteht so die Möglichkeit für NutzerInnen anderen zeigen zu können, dass sie besonders viele Achievements gesammelt haben indem sie einen Avatar auswählen, welcher als Belohnung für “50 gesammelte Achievements” vergeben wird. Die Freischaltung eines Gegenstands könnte dann beim nächsten Öffnen der Sammlung geschehen in der eine Übersicht der besessenen Errungenschaften aufgeführt ist oder durch einen kleinen “Belohnungen abholen” Button im Achievement-Menü.

Beispiel Achievements

Aufgaben:

z.B. Fach Mathe:

- Löse X Mathe Aufgaben
- Löse ein Mathe-Taskset X mal fehlerfrei
- Löse X mal einen EquationTask

Spiele:

z.B. FlappyLama:

- Erreiche X Punkte in FlappyLama
- Erreiche in X Spielen hintereinander X Punkte in FlappyLama
- Spiele X mal FlappyLama hintereinander

Weiteres:

- Spare X Münzen
- Sammle insgesamt X Münzen
- Sammle X Avatare

“???”:

- Drücke 5 mal hintereinander im Hauptmenü auf Anna
- Erhalte alle Lama Fun Facts im Hauptmenü
- Starte eine Runde Snake, FlappyLama und Affenleiter nacheinander

Implementierung

Die Implementierung von Achievements benötigt ein Achievement-Menü, in welchem der/die NutzerIn eine Übersicht über seine/ihre bereits gesammelten Achievements erhält. Solch ein Menü könnte z.B. in der unteren Leiste oder in der AppBar des Hauptmenüs eingefügt werden. Wird darauf geklickt, so öffnet sich ein eigenes Menü, in welchem alle Achievements kategorisch eingefügt sind. Die Achievements sollten dann im Source-Code hinterlegt sein. Ebenso sollte dem/der NutzerIn angezeigt werden wieviele Achievements er/sie bereits gesammelt hat. Diese Informationen müssten in der Datenbank für jeden NutzerIn hinterlegt werden. In der aktuellen Version von Lama ist dieser Teil bereits in der Datenbank hinterlegt. Eventuell wäre auch eine Implementierung in Richtung “data-driven” möglich, bei dem es dem/der LehrerIn erlaubt sein könnte, eigene Errungenschaften einzubauen.

In-App Shop

Eine weiteres Features, welches großes Potential hat, wäre ein In-App Shop. Dieser könnte vorher erwähnte Avatare und Skins enthalten. In diesem würde mit den bereits implementieren Lama-Coins bezahlt werden. Es muss für BenutzerInnen deutlich gemacht werden, dass nicht mit echtem Geld bezahlt wird. Dabei wäre vielleicht ein alternativer Name nützlich wie “Flohmarkt” oder “Basar” statt “Einkaufsladen”. Ein interessanter Faktor wäre es zu beobachten, ob Kinder bereit wären ihre Münzen, mit denen sie eigentlich Spielzeit erhalten, gegen einen Avatar oder Skin einezutauschen.

Implementierung

Der Shop könnte in der unteren Leiste im Main-Menü angezeigt werden. Dabei sollte dieser als eigene Seite eine .dart Datei haben. Beim Öffnen muss zunächst überprüft werden, was bereits freigeschaltet wurde und was nicht. Wichtig ist es auch den NutzerInnen anzuzeigen, wie viele Münzen sie aktuell besitzen. Wird dann auf ein Item geklickt, sollte ein Kauf-Screen erscheinen, auf welchem sichtbar ist wie viele Münzen nach dem Kauf übrig sind. Sind nicht genug Münzen vorhanden, so sollte angezeigt werden wieviele noch nötig sind zum Kauf. Wichtig ist die Transaktion der Münzen und den Erhalt des Kaufobjekts abzusichern, sodass nicht durch Tricks die App geschlossen werden kann, um nur den Gegenstand zu erhalten ohne Münzen zu verlieren. Andersrum sollte es nicht möglich sein Münzen zu verlieren ohne den Gegenstand zu erhalten.

In-App Anleitung

Da nun schon häufiger festgestellt wurde, dass die Einrichtung von LAMA beim ersten Starten der App für viele Personen erstmal unverständlich sein kann, wäre ein interaktives Tutorial wünschenswert.
Eine Umsetzungsidee dafür wäre es, wenn nach dem Bestätigen der Datenschutzerklärung Anna durch Pop-ups und Highlighting den/die BenutzerIn interaktiv durch die Erstellung eines Admin Accounts, die Funktionen eines Admins und die Erstellung eines ersten Schülers führt.
Dieses Tutorial könnte so gestaltet sein, dass zu Beginn entschieden wird, ob die Erklärung nötig ist oder diese übersprungen werden kann.
Des Weiteren wäre im Adminmenü eine detaillierte Anleitung und oder Verlinkung auf eine solche zum Verfassen einer Taskset Datei oder Nutzerliste vorteilhaft. Diese könnte durch das Anklicken eines Fragezeichens geöffnet werden.

Spiele Weiterentwicklung

Flame Engine

Um die Spiele zu entwickeln, wurde die Flame Engine (https://flame-engine.org/) genutzt. Flame ist eine Game Engine für Flutter. Es wurde die - zu diesem Zeitpunkt - aktuelle Version 0.29.4 verwendet. Einbindung des packages und Informationen zu Versionsänderungen von Flame selbst können unter https://pub.dev/packages/flame nachgelesen werden.
Informationen zu Features und erste Starthilfe zur Spieleentwicklung mit Flame gibt es unter https://flame-engine.org/docs/#/. Während die Dokumentation aller Klassen und Funktionen auf https://pub.dev/documentation/flame/ zu finden ist.
Herzstück der Flame Engine ist das wiederholte automatische Ausführen der zwei Funktionen *update()* und *render()* sowie die vielen unterschiedlichen Komponenten. In *update()* wird die Spiellogik abgehandelt und in *render()* werden die Komponenten auf den Bildschirm gezeichnet. So entsteht ein sogenannter Game Loop, um den sich die weitere Entwicklung aufbaut.

Weiterentwicklung der vorhandenen Spiele

Neben der Erweiterung des Spielekatalogs wäre es auch denkbar den bereits bestehenden Spielen weitere Features hinzuzufügen. Hier folgt eine kleine Aufzählung solcher Features, die in der aktuellen Version noch nicht implementiert werden konnten. Für die Umsetzung ist es jedoch erforderlich eine Komplexitätsanalyse durchzuführen, so dass dem Ziel eines einfachen und reduzierten Spielkonzepts entsprochen wird.

Snake

Bei Snake wäre es denkbar besondere “Äpfel” ins Spiel einzubauen. Diese könnten dann zu festen Zeiten oder zu einem fixen Punktestand auf dem Spielfeld erscheinen und den Punktestand um fünf statt nur einen Punkt erhöhen. Die Visualisierung könnte durch eine andere Farbe oder gar einen anderen Gegenstand erfolgen.
Möglich wäre auch ein sogenanntes “Power-Up”-System. Hier könnten in den vier Ecken Power-Ups liegen, welche dem/der SpielerIn z.B. die Möglichkeit geben durch eine Wand zu gehen und auf der anderen Seite wieder zu erscheinen, die Geschwindigkeit zu erhöhen/verringern oder die Punkte für jeden gegessenen Apfel zu multiplizieren.

Weiterhin ist eine andere Steuermöglichkeit umsetzbar, die neben der bis jetzt implementierten Button-Steuerung auch eine Swipe-Steuerung ermöglicht.

Flappy-Lama

Erweiterungen bei Flappy-Lama könnte ebenfalls durch einsammelbare Gegenstände erfolgen. Ein Beispiel wären kleine Lama-Münzen, die der/die SpielerIn einsammeln kann (aber nicht muss). Diese gäben beispielsweise zusätzliche Punkte, würden aber an schwer erreichbaren Orten auftauchen. Ein anderer Effekt könnte das Verstecken des nächsten Hindernisses oder das Vergrößern/Verkleinern des Lamas sein. Die Ideen sind dabei sehr vielfältig und müssen jedoch einer Komplexitätsabschätzung unterzogen werden.
Ein anderer Aspekt könnte eine variable Schwierigkeit der Spielrunde sein, die je nach Klassenstufe des/der eingeloggten NutzerIn verschieden sein könnte. Die wesentlichsten Veränderungen wären dabei die Größe der Löcher sowie die Geschwindigkeit der Hindernisse.

Affenleiter

Auch bei dem Spiel Affenleiter ist es möglich Gegenstände einzuführen, die eingesammelt werden können. Dabei können diese erscheinen und nach einem kurzen Augenblick wieder verschwinden. Der Grund dafür liegt in den nicht vorhandenen alternativen Wegen.
Die möglichen positiven Effekte dieser Gegenstände könnte das Ausblenden der nächsten x Äste, das zeitweise Verdoppeln der Punkte und dem Zuweisen eines Helms sein, der die nächsten x Kollisionen abhält. Auch negative Effekte sind denkbar wie zum Beispiel das reduzieren des Timers, das beschleunigen des Timers oder das reduzieren der Punkte.
Neben den Gegenständen könnten auch zusätzliche alternative Wege erstellt werden, so dass es beispielsweise mehr als nur eine richtige Wahl geben kann. Als Beispiel könnten hier Inseln implementiert werden, die zur linken oder rechten Seite des Baumes platziert werden könnten. Die Bewegung des Affen müsste dabei allerdings angepasst werden und die Option bieten auf diese sich zu bewegen.

Neues Spiel hinzufügen

Für das Einbinden neuer Spiele in die App gilt es folgende Schritte durchzuführen:

- ***GameWidgetScreen*** erstellen
- ***GameListScreen*** erweitern
- ***GameListScreenBloc*** erweitern

GameWidgetScreen erstellen

Für das neue Game Widget wird eigens dafür eine neue Klasse in folgendem Ordner erstellt **lib → app → screens**. Diese Klasse erbt von ***StatelessWidget*** und erstellt und implementiert das Widget des Spiels. Als Beispiel soll hier im Folgenden "Flappy Lama" herangezogen (**lib → app → screens → flappy_game.screen.dart)** werden, um die einzelnen Bestandteile näher zu erläutern.

```
class FlappyGameScreen extends StatelessWidget {
 final UserRepository userRepository;

 const FlappyGameScreen(this.userRepository);
```

Im ***FlappyGameScreen*** wird im Konstruktor ebenfalls das ***UserRepository*** mitgegeben, welches im weiteren Verlauf notwendig für den Zugriff auf die Datenbank ist. Dabei umfasst dieser das Laden und Speichern der Highscores. Um diese Funktionalität zu gewährleisten, muss dieses Repository im neuen Spiel mit übergeben werden.

```
return Scaffold(
 appBar: AppBar(
   title: Text("Flappy Lama"),
 ),
 body:
```

Der nächste Schritt besteht dabei den ***Scaffold*** aufzubauen mit der passenden ***AppBar***. Diese wurde sehr schlank gestaltet, um unnötige sichtbare Elemente zu minimieren.

```
 Container(
   color: Colors.green,
   child: FlappyLamaGame(context, userRepository).widget,
 )
```

Anschließend wird das ***Widget*** vom "FlappyLama"-Spiel initialisiert und als Child vom ***Container*** gesetzt. Jede Instanz von ***BaseGame*** oder ***Game*** aus der FlameEngine hat einen ***Getter*** Widget, der das Game-Widget zurückgibt. Sollten all diese Schritte umgesetzt worden sein, so ist der Screen bereit eingebunden zu werden.

GameListScreen erweitern

Die Klasse ***GameListScreen*** befindet sich unter **lib → app → screens**. In dieser befindet sich an erster Stelle eine final List von ***GameListItem***, die in der Liste der App als Repräsentation angezeigt werden sollen.

```
final List<GameListItem> games = [
 GameListItem(
      "Snake",
      16,
      "Steuer die Schlange mit den Pfeiltasten ...!"),
 GameListItem(
      "Flappy-Lama",
      15,
      "Tippe auf den Bildschirm, um das Lama ...!"),
 GameListItem(
      "Affen-Leiter",
      18,
      "Tippe die entsprechende Richtung an, um ...!"),
];
```

Jedes ***GameListItem*** verfügt über drei Felder.

- *name* (Name)
- *cost* (Lama Münzen, die für dieses Spiel eingesetzt werden müssen)
- *desc* (eine kurze Beschreibung des Spiels)

Um ein eigenes Spiel einzubinden sind im Vorfeld somit diese drei Eigenschaften des Spiels zu bedenken. Anschließend wird dieser Liste das neue Spiel hinzugefügt.
Bezugnehmend auf die Kosten des Spiels ist zu sagen, dass deren Bepreisung in Abhängigkeit von der angestrebten Spiellänge zu setzen ist. Da grundsätzlich ein Verhältnis von Lernzeit zu Spielzeit von 2:1 angestrebt wird und die Aufgaben für 30 Sekunden Bearbeitungszeit in etwa einen Lama-Coin generieren, sollte der Preis dementsprechend kalkuliert werden.

GameListScreenBloc erweitern

Der letzte Schritt für die Einbindung eines neuen Spielwidgets besteht dabei den ***GameListScreenBloc*** zu erweitern. Dieser befindet sich unter ***lib*** → ***app*** → ***bloc*** → ***game_list_screen_bloc.dart***.

```
void navigateToGame(String gameName, BuildContext context,
UserRepository userRepository) {
 Widget gameToLaunch;
 switch (gameName) {
   case "Snake":
     gameToLaunch = SnakeScreen(userRepository);
     break;
   case "Flappy-Lama":
     gameToLaunch = FlappyGameScreen(userRepository);
     break;
   case "Affen-Leiter":
     gameToLaunch = ClimberGameScreen(userRepository);
     break;
```

```
    default:
      throw Exception("Trying to launch game that does not
exist");
  }
  Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => gameToLaunch));
}
```

In der oben genannten Methode wird in Abhängigkeit vom übergebenen *gameName*, das passende ***GameWidgetScreen*** geladen, welches im ersten Schritt implementiert wurde. Der übergebene *gameName* kann dabei einer aus der Liste von ***GameListItem*** des vorangegangenen Schritts sein und wird in einem *switch* ausgewertet.
Sofern es eine Übereinstimmung gibt wird im ***Navigator*** (Widget das in einem Stack eine Anzahl von KindElement managed) das ***gameToLaunch*** gepusht und somit das ***GameWidgetScreen*** aus dem ersten Schritt geladen.

```
Stream<GameListScreenState> mapEventToState(
    GameListScreenEvent event) async* {
  if (event is TryStartGameEvent) {
    if (userRepository.getLamaCoins() >= event.gameCost) {
      userRepository.removeLamaCoins(event.gameCost);
      navigateToGame(event.gameToStart, event.context,
userRepository);
    } else {
      yield NotEnoughCoinsState();
    }
  }
}
```

Um den Abzug der Lama-Münzen kümmert sich die Methode ***mapEventToState***, die ebenfalls den ***NotEnoughCoinsState*** zurückgibt, sofern der/die NutzerIn nicht genug Lama-Münzen aufweist.
Nach diesen Schritten ist das neu entwickelte Spiel in der Oberfläche der App eingebunden und kann verwendet werden.

Highscores

Speichern und Laden

Für die Nutzung der Datenbank zur Serialisierung der Highscores, wie oben beschrieben, ist die Übergabe des ***UserRepository*** unumgänglich. Dieses gewährleistet den Zugriff zur Datenbank und dem/der eingeloggten BenutzerIn. Ziel ist es dabei, einen neuen Highscore speichern zu können sowie den Highscore laden zu können.

```
void saveHighscore() {
 if (!_savedHighscore) {
   _savedHighscore = true;
   _userRepo.addHighscore(Highscore(
       gameID: _gameId,
       score: score,
       userID: _userRepo.authenticatedUser.id));
 }
}
```

Im hier gezeigten Beispiel wird ein neuer Highscore in die Datenbank gespeichert. Die Variable ***_savedHighscore*** ist eine flag, die kennzeichnet, ob der Highscore schon gespeichert wurde. Dies ist gerade dann notwendig, wenn die Methode innerhalb des Game-Loopers aufgerufen wird und die mehrfache Serialisierung verhindert werden soll.
Sollte der Highscore noch nicht gespeichert sein, kann über das ***UserRepository*** die passende asynchrone Methode aufgerufen werden. Diese benötigt eine Instanz der Klasse ***Highscore***, welche drei Werte bei der Initialisierung benötigt: die eindeutige Spiel-Id (***gameID***), den zu speichernden Punktestand (***score***) sowie die ID des eingeloggten Nutzers (***userID***). Letzteres kann über das ***UserRepository*** über das Feld ***authenticatedUser*** und dessen ***id*** ermittelt werden.

```
this._userHighScore = await _userRepo.getMyHighscore(_gameId);
this._alltimeHighScore = await _userRepo.getHighscore(_gameId);
```

Für das Laden der Highscores dient ebenfalls das ***UserRepository***. In diesem gibt es zwei Methoden die einen ***Future<int>*** zurückgeben, der den Wert des geladenen Highscores widerspiegelt.

```
Future<int> getMyHighscore(int gameId) async {
```

Mit dieser Methode kann der persönliche Highscore des/der eingeloggten NutzerIn ermittelt werden, indem die jeweilige Spiel-ID (***gameId***) übergeben wird. Sollte kein Eintrag in der Datenbank hinterlegt sein, so wird der zurückgegebene Integer standardmäßig auf 0 gesetzt.

```
Future<int> getHighscore(int gameId) async {
```

Diese Methode ermittelt den besten Highscore eines Spieles über alle NutzerInnen des Endgeräts hinweg. Auch diese Methode benötigt nur die Spiel-Id (***gameId***) und gibt standardmäßig als Integer 0 zurück, sofern kein Eintrag gefunden wurde.
Bei beiden Methoden handelt es sich um einen ***Future*** Rückgabewert, so dass diese innerhalb einer ***async*** Methode aufgerufen werden sollten. Als Beispiel hier nochmal das Laden der beiden Highscores:

```
void initialize() async {
 // load userHighScore
 userHighScore = await _userRepo.getMyHighscore(_gameId);
 // load allTimeHighScore
 allTimeHighScore = await _userRepo.getHighscore(_gameId);
```

Zukünftige Features

Die Highscores sind aktuell nur lokal umgesetzt und werden über den aktuell persönlichen und den gesamten Rekord auf dem Gerät in den vorhandenen Spielen unterschieden.
In diesem Zusammenhang würde es sich anbieten diesen Highscores einen zusätzlich gesonderten Ort in der App zu geben, um so einen detaillierten Vergleich der NutzerInnen untereinander zu ermöglichen. Dabei wäre denkbar, dass sich gezielt mit einem anderen NutzerIn in den unterschiedlichen Spielen verglichen werden kann. Weiterhin ist die Unterscheidung der Highscores nach Klassenstufe feingranularer möglich.
Neben dieser Ausgestaltung kann ebenfalls die Möglichkeit implementiert werden, diese auf einen Webserver über einen ***request*** zu pushen, um so einen geräteunabhängigen Vergleich zu ermöglichen. Dabei ist wesentlich darauf zu achten so wenige Daten wie möglich zu übertragen, um so keinerlei Rückschlüsse auf eine Person ziehen zu können.

Multiplayer

Ein weiteres interessantes Feature im Bereich der Spiele würde eine Multiplayeroption darstellen. Die Umsetzung einer solchen erfordert jedoch entweder ein Backend oder einen anderen direkten Verbindungstyp über Bluetooth, WLAN, etc.
Generell würde dies zusätzlich die Motivation steigern Lama-Münzen zu erarbeiten, jedoch ist der Entwicklungsaufwand als sehr hoch anzusiedeln. Nicht nur die Verbindung der Spieler miteinander, sondern auch die Spiele selbst müssten auf den Ansatz “Multiplayer” umstrukturiert werden.

Zukünftige Administrative Features

Passwort vergessen Funktion

Nach aktuellem Stand muss bei fehlendem Passwort für einen Administratorzugriff die App neu installiert werden, da keine Möglichkeit besteht ein AdministratorInnen Konto zu nutzen ohne das entsprechende Passwort zu kennen.
Aus diesem und Gründen der User-Experience muss eine "Passwort vergessen” - Funktion implementiert werden. Für die normale Nutzung hat diese Funktion keine Auswirkung. Für einen/eine AdministratorIn muss jedoch eine Sicherheitsfrage implementiert werden, für die er/sie eine Sicherheitsfrage mit einer Antwort auf diese hinterlegen muss. Dabei ist die Sicherheitsfrage frei wählbar und wird nicht durch standardisierte Fragen vorgegeben.

Implementierung

Im Login Bildschirm soll für alle NutzerInnen ein “***Passwort vergessen?***” - Button zur Verfügung stehen. Betätigt eine normale NutzerIn diesen Button so wird er/sie darauf hingewiesen, sich für das Wiederherstellen des Passworts, an einen/eine AdministratorIn zu wenden. Ein/Eine AdministratorIn wird beim betätigen des Buttons entsprechend auf einen Bildschirm weitergeleitet, welcher die Sicherheitsfrage abfragt.

Dem/Der AdministratorIn muss auch die Möglichkeit gegeben werden die Sicherheitsfrage und Antwort zu ändern. Dazu muss bei der Editierung von Konten zwischen administrativen NutzerInnen und normalen NutzerInnen unterschieden werden. Administrative Konten benötigen keine Manipulation ihrer Münzen oder der Klasse, sondern nur die Manipulation ihres Namens, Passworts und Sicherheitsfrage sowie Antwort.

Zentrale Konfiguration der App

Anmerkung: Das Umsetzen dieses Konzepts sollte nur durchgeführt werden, wenn die App ordentlich refactored wurde und ein solides Grundkonzept aufweist!

Nach aktuellem Stand können nur bestimmte Bereiche der App (Tasksets und NutzerInnen) via URL in die App eingefügt werden. Für den generischen Massengebrauch ist jedoch das Konfigurieren jedes einzelnen Endgeräts trotz dieser Features eine Last. Um diese Last abzuschwächen, besteht die Möglichkeit beim ersten Start einen “Synchronisationsmodus” zu aktivieren. Dazu muss ein Link der App hinzugefügt werden, welcher alle nötigen Informationen bereitstellt, um die App zu konfigurieren. Das beinhaltet zum Beispiel App

Konfigurationen, Nutzerlisten und Tasksets (als vollständiges .json oder als Verweis auf Tasksets als Liste mit Links). Die Inhalte des Links werden dann in die App fest übernommen und bei jedem Start die vorhanden Daten mit den über den Link verfügbaren Daten verglichen und synchronisiert. Wenn eine Schule nun beispielsweise die Standardaufgaben wieder aktivieren möchte, muss sie lediglich die jeweilige Konfiguration in der über den Link erreichbaren json-Datei hinterlegen und alle Endgeräte, die über diesen Link synchronisiert werden, übernehmen automatisch die Einstellung nach einem Neustart.
Der/Die AdministratorIn verfügt in diesem Modus jedoch nicht über die üblichen Möglichkeiten. Er/Sie sollte lediglich den “Synchronisationsmodus” beenden oder aktivieren können.

Aufkommende Probleme und eventuelle Lösungen:

Der mögliche Verlust von Daten wie Lamamünzen oder Highscores

Variable Daten wie Highscore, Lamamünzen oder Achievements werden bei der Synchronisation übergangen und wie gehabt lokal zu einem/einer NutzerIn gespeichert. Wird ein/eine NutzerIn aus dem Synchronisations-Link entfernt, werden auch die damit verbundenen Daten entfernt.

Nichterreichbarkeit des Links

Ist der Synchronisation-Link nicht erreichbar so ist dennoch die letzte Konfiguration der App verfügbar. Es findet also schlicht keine Synchronisation statt.

Langer Appstart und schlechtes Internet

Die App könnte vorerst mit dem letzten Stand starten und im Hintergrund entsprechende Daten nachladen. (Dieser Vorgang ist **SEHR** aufwendig zu implementieren)

Lama als WebApp

(Mit dem Begriff WebApp ist hier eine Progressive Web App gemeint. Eine Webseite die auf einem Endgerät installiert werden kann und dadurch offline verfügbar ist.)

Anmerkung: Das Umsetzen dieses Konzepts sollte nur durchgeführt werden, wenn die App ordentlich refactored wurde und ein solides Grundkonzept aufweist!

Die aktuelle Version der App ist nicht als WebApp nutzbar. Grund dafür ist die genutzte lokale Datenbankvariante, die darauf ausgelegt ist auf mobilen Endgeräten zu arbeiten und dadurch nicht die Fähigkeit besitzt lokal auf einem Rechnersystem Daten abzulegen oder

generell zu verwalten. Es steht für eine WebApp also kein Backend in Form einer Datenbank zur Verfügung.

Um dem fehlenden Backend Abhilfe zu schaffen, könnte eine ähnliche Idee wie in "***Zentrale Konfiguration der App***" genutzt werden. Beim Starten der WebApp wird nach einem Link gefragt über den die Applikation synchronisieren soll. Es werden also alle nötigen Informationen wie BenutzerInnen, Tasks und Konfigurationen über diesen Link bezogen. Der Link wird in den Cookies abgelegt und beim Starten der WebApp genutzt um Konfigurationen zu laden.

Aufkommende Probleme und eventuelle Lösungen:

Link nicht mehr erreichbar oder ein neuer Link verfügbar

Ist der Link nicht mehr erreichbar oder ein neuer Link verfügbar muss es eine Möglichkeit geben einen neuen Link zu hinterlegen. Dazu könnte bei Nichterreichbarkeit die Eingabe eines neuen Links gefordert werden.
Soll der Link geändert werden, obwohl er noch erreichbar ist, sollte über den Link ein/eine AdministratorIn hinterlegt werden und entsprechend für diese Tätigkeit genutzt werden.

Generelles Speichern von NutzerInnen oder Highscores

Generell ist es keine gute Praxis "viele" Daten in den Cookies abzulegen daher sollte darauf verzichtet werden NutzerInnen über die Cookies zu speichern. Es sollte einen/eine NutzerIn geben mit seinen/ihren in dieser Session erarbeiteten Werten.

Eventuelle Ladezeiten

Aufgrund der Tatsache, dass zum Beispiel Tasks nur als json über Session Variablen gespeichert werden, erfordert es für jeden Task ein "*just in time*" parsing. Dieses Vorgehen kann zu eventuell langen Ladezeiten führen. Was jedoch in anbetracht dessen, dass es sich hier um ein, im Vergleich zu einem Mobilgerät, voraussichtlich leistungsfähigeren Gerät handelt, wenig wahrscheinlich ist.

Erstellung neuer Aufgabentypen

Die Erstellung eines neuen Aufgabentyps (TaskType) ist recht aufwändig. Diese kann in Zukunft noch erheblich verbessert werden, da sowohl viel redundanter Code geschrieben werden muss und auch ein Großteil der wichtigen Codeabschnitte nicht vom Compiler forciert wird.

Bei der Erstellung eines neuen Aufgabentyp gilt es folgenden Ablauf zu befolgen.

Schritt 0

Bevor gestartet wird muss klar sein, welcher Aufgabentyp implementiert werden soll und durch welchen String-Identifier dieser identifizierbar sein soll.
(Beim 4-Karten-Quiz ist dieser Identifier zum Beispiel: “4Cards”)

Schritt 1

Zuerst wird eine Klasse für den neuen TaskType erstellt. Dies geschieht in der Datei ***“task.dart”***. Diese befindet sich im Pfad ***“../lib/app/task-system/task.dart”***. In dieser Datei wird nun eine neue Klasse angelegt, die von der Klasse “Task” erbt. Der Klasse können anschließend alle Variablen hinzugefügt werden, die benötigt werden. Hierbei handelt es sich meist um die Dinge, die bei der Aufgabe angezeigt werden sollen. (Zum Vergleich: Bei dem Aufgabentyp “4Cards” handelt es sich hierbei um die Frage, die richtige Antwort und die drei falschen Antworten.)
Nun wird der Konstruktor hinzugefügt. Dieser nimmt als erste vier Parameter - den TaskType Identifier (also das gewählte String Keyword), die Lama-Münzen, die es für das richtige Lösen der Aufgabe gibt, den Text den Anna bei der Aufgabe sagen soll und wie oft der Task richtig gelöst werden kann bevor es keine Münzen mehr gibt - entgegen. Bitte hierbei beachten: Hier werden KEINE konkreten Werte verlangt. Diese werden durch das spätere JSON-Parsing automatisch gesetzt. Zum Abschluss der Klasse muss nun noch die ***toString()*** Methode überschrieben werden, die ***super.toString()*** aufruft und all ihre eigenen Parameter anfügt.
Diese toString() Methode wird für das System genutzt, das schaut wie oft der Task noch gelöst werden kann.

Das Ergebnis könnte nun ungefähr folgendermaßen aussehen:

```
class TaskNEUERTASKTYP extends Task {
  String PARAM1;
  List<int> PARAM2;

  TaskNEUERTASKTYP(String taskType, int reward, String lamaText,
      int leftToSolve, this.PARAM1, this.PARAM2)
      : super(taskType, reward, lamaText, leftToSolve);

  @override
  String toString() {
    String s = super.toString();
    s += PARAM1;
    for (int i = 0; i < PARAM2.length; i++)
      s += PARAM2[i].toString();
    return s;
  }
}
```

Schritt 2

Als nächstes wird in der Task Klasse, die sich ganz oben in der Datei “task.dart” befindet, ein Eintrag im factory Konstruktor ergänzt.
Im vorhandenen switch muss ein case für den neuen TaskType hinzugefügt werden. Hierfür wird der in Schritt 0 überlegte String Identifier genommen. Alles was im case nun getan werden muss, ist eine neue Instanz der gerade erstellten Klasse zurückzugeben und dabei die richtigen Felder der json-Map zu übergeben.
Hierbei gilt zu beachten, dass die ersten vier Parameter bei allen TaskTypen gleich sind. Die JSON Key-Names der eigenen Parameter können jedoch frei gewählt werden.

Diese könnte dann wie folgt aussehen:

```
      case "NEUERTASKIDENTIFIER":
        return TaskNEUERTASKTYP(
            taskType,
            json["task_reward"],
            json["lama_text"],
            json["left_to_solve"],
            json[PARAM1],
            List<int>.from(json[PARAM2]));
```

Schritt 3

Nachdem nun klar ist, welche task-spezifischen JSON-Keys benötigt werden, muss der Task Validator angepasst werden. Dieser überprüft, ob ein Taskset korrekt ist und geparst werden kann, um App Abstürze zu verhindern.

Hierfür wird in ***"../lib/app/task-system/taskset_validator.dart"*** navigiert. Dort findet sich die Methode ***"_isValidTaskset()"***. Am Ende dieser wird in den switch ein neuer case für den gewählten Task String Identifier eingefügt.

Anschließend wird die json Map überprüft ob alle benötigten Schlüssel enthalten sind mit json.containsKey(BENÖTIGTERSCHLÜSSEL1) [...]. Außerdem muss sichergestellt werden, dass wenn alle JSON-Keys vorhanden sind die entsprechenden Values auch den richtigen Typ haben. Dies lässt sich in dart mit VALUE is TYP sicherstellen. Für Listen bietet die TaskValidator Klasse eine Hilfsmethode namens "_checkListType<T>()" an, die überprüft, ob alle Elemente der Liste denselben Typ T haben. Hier lohnt es sich, sich an den anderen Task-Validierungen zu orientieren.

Schritt 4

Nun wird der eigentliche Screen erstellt. Hierfür wird eine neue Datei im Pfad ***"../lib/app/screens/task_type_screens/"*** erstellt. Die Klasse kann entweder ***"StatelessWidget"*** erweitern oder ***"StatefulWidget"***, wenn UI-seitige Logik vorhanden ist. Für komplexere Logik sollte ein Bloc im Ordner ***"../lib/app/bloc/taskBloc/"*** erstellt werden.
Der Konstruktor sollte sowohl den Task als auch einen Parameter vom Typ ***BoxConstraints*** beinhalten, da die BoxConstraints den Platz vorgeben in dem sich der Task "aufhalten" darf ohne über den Bildschirm oder die AppBar zu rutschen. Hier sollte demnach mit relationalen und prozentualen Größen (width und height) gearbeitet werden, unter Zuhilfenahme dieser BoxConstraints.

Der Rest dieser Klasse ist fast komplett frei in der Implementation inklusive der "build()" Methode. Die einzige Ausnahme bietet hierbei das Beantworten einer Aufgabe. Wird eine Antwort gegeben, zum Beispiel durch das Drücken eines "Fertig" Buttons, so muss diese an den TaskBloc gesendet werden, der für die Evaluation zuständig ist. Da dieser TaskBloc bereits im context des TaskScreen vorhanden ist und dadurch jedem TaskScreen zur Verfügung steht, lässt sich eine Antwort durch folgende Codezeile senden:

```
BlocProvider.of<TaskBloc>(context).add(AnswerTaskEvent(ANTWORT))
```

Es gilt zu beachten, dass dies nur für eine Antwort vom Typ String gilt. Soll eine Antwort von einem anderen Type (z.B. eine Liste) als Antwort gesendet werden, so muss dies mit Hilfe eines “named constructors” gemacht werden, der in der Klasse ***AnswerTaskEvent*** erstellt werden muss. Dieses Event befindet sich in der Datei ***“../lib/app/event/task_events.dart”***. Zum Beispiel nutzt der MoneyTask:

```
AnswerTaskEvent.initMoneyTask(double providedAnswerDouble) {
    this.providedAnswerDouble = providedAnswerDouble;
  }
```

Der Aufruf ändert sich dadruch zu:

```
BlocProvider.of<TaskBloc>(context)
      .add(AnswerTaskEvent.initMoneyTask(ANTWORT))
```

Schritt 5

Nachdem der Screen erstellt wurde, muss dem TaskSystem mitgeteilt werden, dass es diesen Screen anzeigen soll, wenn der entsprechende TaskTyp erkannt wird. Hierfür wird in die Datei ***“../lib/app/screens/task_screen.dart”*** navigiert. Dort befindet sich die Methode ***“getScreenForTaskWithConstraints()”***. In dieser Methode findet sich ein switch in dem erneut ein case für den gewünschten Aufgabentyp JSON-Identifier erstellt wird und eine Instanz des neu erstellten Screens zurückgegeben wird.

Schritt 6

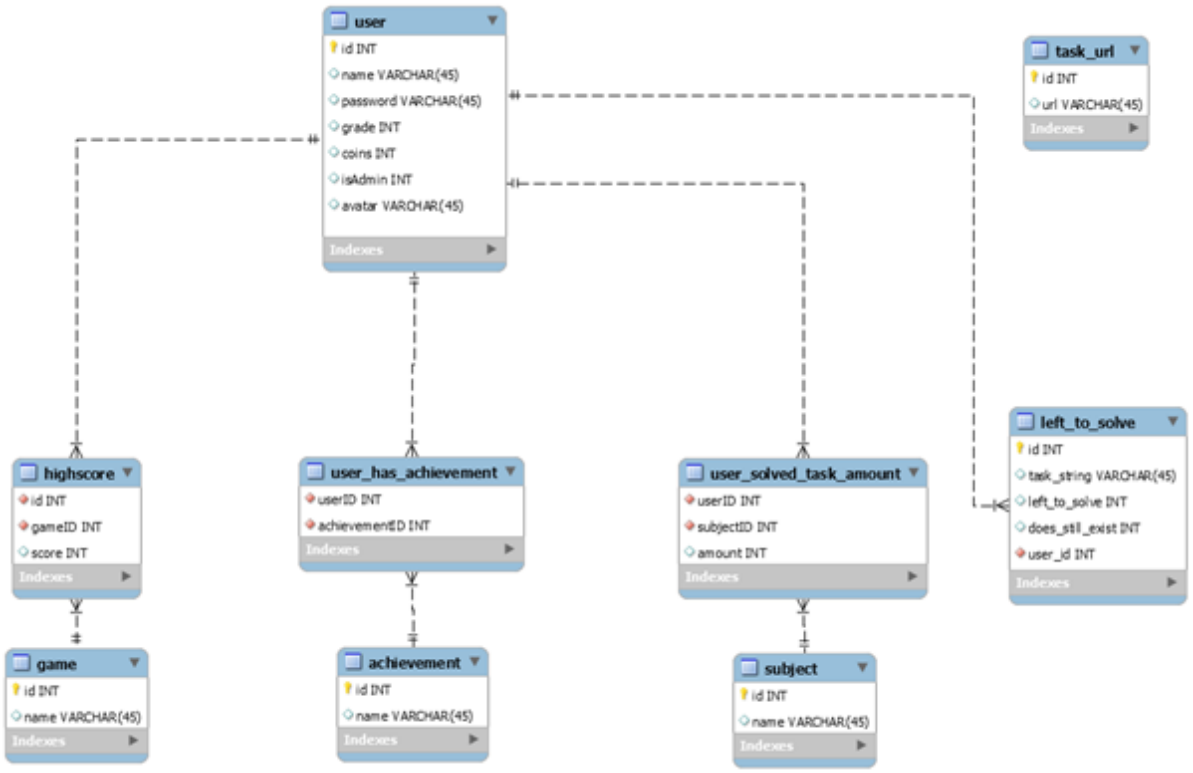
In der Datei ***“../lib/app/bloc/task_bloc.dart”*** muss nun noch in der Methode ***“mapEventToState()”*** ein else if Block für den neuen Task anlegt werden, indem die Antwort evaluiert wird.

```
else if (t is TaskNEUERTASKTYP) {
        if (ANTWORT RICHTIG) {
          rightAnswerCallback(t);
          yield TaskAnswerResultState(true);
        } else {
          wrongAnswerCallback(t);
          yield TaskAnswerResultState(false);
        }
      }
```

Die gegebene Antwort kann dabei aus dem AnswerTaskEvent mit “event.ANSWER_VARIABLE_NAME” erhalten werden. Auf den Task, in dem die richtige Antwort vorhanden ist, kann leicht mit t.TASK_VARIABLE_NAME zugegriffen werden.
ANTWORT RICHTIG muss natürlich durch eine individuelle Evaluation der Antwort ersetzt werden. Auch hier lohnt es sich, die bereits vorhandenen Aufgabentypen zu betrachten.
Die callback Methoden kümmern sich dann um das Hinzufügen von Lama Münzen und um das tracken der richtigen und falschen Antworten für die Zusammenfassung am Ende eines Tasksets.
Hiermit ist die Erstellung eines neuen Aufgabentyps abgeschlossen!

Verwendung der Datenbank

In dem Schaubild sind alle Tabellen und die dazugehörigen Spalten, die in der Datenbank zu Verfügung stehen dargestellt.



EER Diagramm

Die Daten aus der Datenbank sollen ausschließlich über die zu Verfügung gestellten Methoden verwendet werden. Dabei gibt es Methoden zum Einfügen, zum Abfragen, zum Updaten und zum Löschen von Einträgen. Diese sind unter lama_app → lib → db in der

Datei database_provider.dart zu finden. Die Methoden müssen mit dem Schlüsselwort await aufgerufen werden. Der Aufbau des Aufrufes sieht wie folgt aus: DatabaseProvider.db.Methodennamen(Übergabeparameter). Die Anzahl der Übergabeparameter unterscheidet sich, je nach aufgerufener Methode.

Ausführen der Datenbank Tests

Zum Testen der Datenbank steht in dem Ordner database_provider_test.dart eine main Klasse zur Verfügung, in der die Tests ausgeführt werden. Der Aufbau darin ist dem Vorgehen von Unit Test angelegt. Da Unit Tests bei Datenbanken nicht funktionieren, muss zum Ausführen ein Emulator gestartet werden. Zum Ausführen wird im Terminal der Befehl flutter run test/db/database_provider_test.dart eingegeben.
Die einzelnen Tests sind in Gruppen aufgeteilt und überprüfen, ob die erwarteten Rückgabewerte zurückgegeben werden.
In dem Terminalfenster ist sichtbar, ob alle Test erfolgreich waren oder welche Test fehlgeschlagen sind.
Beim Ausführen der Tests ist zu beachten, dass sämtliche Einträge der Datenbank gelöscht werden!

Erweiterung der Datenbank

In der App verwenden wir, zum speichern der Daten, SQFlite in der Version ^1.3.2+4.

Erstellen oder Updaten von Tabellen

Es können nach Bedarf weitere Tabellen hinzugefügt werden oder die vorhandenen Tabellen können angepasst werden.

Schritt 1

Im ersten Schritt wird bei der Erstellung einer neuen Tabelle ein Modell für diese angelegt. Die Modells sind unter lama_app → lib → app → model zu finden. Beim Updaten muss das vorhandene Modell angepasst werden. Als erstes wird eine Variable angelegt in der der Name der Tabelle gesetzt wird. Die Variablen welche die Namen der Spalten enthalten, werden in einer eigenen Klasse deklariert und initialisiert. Diese Variablen sind mit den Schlüsselwörtern static und final gekennzeichnet.

Des Weiteren ist eine Klasse mit den Methoden fromMap und toMap anzulegen. Diese Methoden werden für die einfachere Handhabung der Einträge in die Tabelle oder aus der Tabelle benötigt. Die Klasse besitzt einen Konstruktor über den sie die Parameter, welche in die Tabelle eingetragen werden sollen, entgegennimmt.
Im folgenden ist der Aufbau der Klasse des Modells im Beispiel der achievement Tabelle zu sehen.

```
class Achievement {
int id;
String name;

 Achievement({this.name});

 Map<String, dynamic> toMap() {
   var map = <String, dynamic>{
     AchievementsFields.columnAchievementsName: name,
   };
   return map;
 }

 Achievement.fromMap(Map<String, dynamic> map) {
   id = map[AchievementsFields.columnAchievementsId];
   name = map[AchievementsFields.columnAchievementsName];
 }
}
```

Schritt 2

Anschließend wird in der Klasse DBMigrator, welche sich in der Datei database_migrator.dart befindet, eine neue Map mit dem Namen migrationsVx angelegt. Das x steht hierbei für die aktuelle Versionsnummer in aufsteigender Reihenfolge. Zu finden ist die Datei database_migrator.dart unter diesem Pfad lama_app → lib → db.
In dieser Map werden die Codezeilen für die Änderungen, welche an einer Tabelle durchgeführt werden sollen oder für die Erstellung neuer Tabellen, eingefügt.

Schritt 3

Die in Schritt 2 erstellte Map muss nun in die Map „migrations“ eingetragen werden. Diese ist ebenfalls in der Klasse DBMigrator zu finden. Wenn die Map nicht dort eingetragen wird, kann diese beim Erstellen oder Updaten der Datenbank nicht ausgeführt werden. Dabei

kann es im späteren Verlauf zu Problemen führen, da nach dem einmaligen Ausführen die Methode zum Updaten nur noch den Code mit höheren Versionsnummern aufruft.

Schritt 4

Unter dem Pfad lama_app → lib → db befindet sich die Datei database_provider.dart. Dort muss die neue Version der Datenbank in die Variable currentVersion eingetragen werden. Dabei wird die alte Version um eins inkrementiert.

Hinzufügen von Methode

Methoden können in der Klasse DatabaseProvider hinzugefügt werden. Die Klasse ist unter dem Pfad lama_app → lib → db in der Datei database_provider.dart zu finden. Für das Arbeiten mit der Datenbank können die vorgefertigten Methoden aus dem sqflite package mit in den eigenen Code eingebunden werden. Dabei hilfreich sind die Methoden query zum Abfragen von Daten, insert zum Einfügen von Daten, update zum Ändern von Daten und delete zum Löschen von Daten.

Erweitern der Datenbank Tests

Neue Tests müssen in eine Gruppe eingefügt werden. Gibt es keine passende Gruppe muss diese neu angelegt werden. Des Weiteren müssen Instanzen der Modelle mit Testdaten angelegt werden. Da bei jedem Aufruf eines Test die Datenbank komplett geleert wird, müssen die Daten in die Datenbank eingefügt werden. Danach können die verschiedenen Abläufe, wie das Updaten oder Auslesen von Daten, ausgeführt werden. Mit der Methode expectLater kann überprüft werden, ob der aktuelle Wert dem erwarteten Wert entspricht.
Es ist zu beachten, dass vor jedem Test zwar die Datenbank geleert wird, Änderungen an den Instanzen der Models bleiben jedoch bei aufeinanderfolgenden Tests bestehen. Dies ist kein Fehler, kann aber zu Abweichungen gegenüber den zu erwarteten Werten führen.

Sicherheit der Datenbank

Um die funktionale Sicherheit zu erhöhen, können Transaktionsprotokolle zum Einsatz kommen. Durch das Loggen von Transaktionen kann die Datenbank bei einem Systemfehler wieder in einen konsistenten Zustand gebracht werden.