

Softwaretechnikprojekt Sommersemester 2022: Erweiterung der LAMA Lern App

Dokumentation und Weiterentwicklung

Im Studiengang
Informatik (Bachelor of Science)

am Fachbereich MNI der Technischen Hochschule Mittelhessen

Inhaltsverzeichnis

Abbildungsverzeichnis	ii
1 Anforderungen	1
1.1 Anforderung 1: Tap the Lama.....	1
1.2 Anforderung 2: Taskset Erstellung	2
1.3 Anforderung 3: Aufgabenverwaltung.....	3
1.4 Anforderung 4: Text To Speech.....	4
2 Praktische Umsetzung	5
2.1 Tap the Lama	5
2.1.1 Auswahl des Spiels und Startmenü	5
2.1.2 Codeorganisation und Flame Implementierung.....	7
2.1.3 Komponenten der Spiel-Oberfläche und Animationen im Spiel	9
2.1.4 Spielelogik und Animationen.....	12
2.1.5 Game Over Menü	16
2.2 Taskset Erstellung	18
2.2.1 Neue Ordner und Dateien	18
2.2.2 Tasksets und Tasks	21
2.2.3 Custom – Widgets	23
2.2.4 Aufbau der Screens zum Erstellen eines Tasksets.....	28
2.2.5 Aufbau der Screens zum Erstellen eines Tasks.....	32
2.2.6 Einen neuen Screen hinzufügen	38
2.3 Aufgabenverwaltung	42
2.3.1 Die UI	42
2.3.2 Die Implementierung des Taskset managements	44
2.3.3 Der Server Screen	47
2.3.4 Die Datenbank	48
2.3.5 Das Server Repository(neu).....	49
2.3.6 Das Taskset Repository(erweitert)	49
2.4 Text To Speech	52
2.4.1 Text To Speech Framework mit bloc Architektur.....	52
2.4.2 Task Screens mit TTS Bloc.....	55
2.4.3 readText Methode.....	58
2.4.4 Ein und Ausschalten von TTS.....	59
2.4.5 Sprachen	61
3 Deployment und Installation der App	63

4	Weiterentwicklungsmöglichkeiten.....	64
4.1	Tap the Lama	64
4.2	Taskset Erstellung.....	66
4.2.1	Weitere Screens hinzufügen	66
4.2.2	Screens automatisch generieren lassen.....	66
4.2.3	Preview – Funktion	66
4.3	Aufgabenverwaltung	67
4.3.1	Aufgabenverwaltung	67
4.4	Text to Speech	67

Abbildungsverzeichnis

Abbildung 1.1-1:	Tap the Lama Mockup	1
Abbildung 1.2-1	Screen zum Erstellen eines Tasksets.....	2
Abbildung 1.2-2	Screen zum Hinzufügen von Tasks.....	3
Abbildung 1.3-1	Modell Serverstruktur.....	4
Abbildung 2.1-1:	Neuen Eintrag in Spieleliste einfügen.....	5
Abbildung 2.1-2:	Auswahl des Spiels mit Übergabeparametern	6
Abbildung 2.1-3:	Tap the Lama Screen.....	6
Abbildung 2.1-4:	Starten des Tap the Lama Spiels.....	7
Abbildung 2.1-5:	Übersicht der TapTheLamaGame Klasse	8
Abbildung 2.1-6:	Kommentare innerhalb einer Coderegion.....	8
Abbildung 2.1-7:	Komponenten in Tap the Lama	9
Abbildung 2.1-8:	LamaButton als klickbare Sprite Komponente	10
Abbildung 2.1-9:	Initialisierung der LamaButtons.....	11
Abbildung 2.1-10:	Visualisierte Attribute der LamaHead Komponenten	11
Abbildung 2.1-11:	Initialisierung der LamaHead Spalten.....	12
Abbildung 2.1-12:	onLoad() Methode	12
Abbildung 2.1-13:	update() Methode	13
Abbildung 2.1-14:	checkSingleHit() Methode	14
Abbildung 2.1-15:	Treffer Überschneidung bei guten Treffern	14
Abbildung 2.1-16:	Aktionen bei einem guten Treffer	15
Abbildung 2.1-17:	moveLamaHeads() Methode	15
Abbildung 2.1-18:	render() Methode	16
Abbildung 2.1-19:	Game Over Menü	17
Abbildung 2.2-1	Überblick der neuen Ordner für die Screens zur Taskseterstellung.....	18
Abbildung 2.2-2	CreateTasksetBloc.....	19
Abbildung 2.2-3	Einblick in CreateTasksetBloc	19
Abbildung 2.2-4	Datei CreateTasksetListBloc.....	19
Abbildung 2.2-5	Überblick der Datei.....	20
Abbildung 2.2-6	State – Datei.....	20
Abbildung 2.2-7	Attribute der Klasse Taskset	21
Abbildung 2.2-8	Die toJSON – Funktion	21
Abbildung 2.2-9	Die Attribute der Klasse Task.....	21

Abbildung 2.2-10 Beispiel einer Task - JSON	22
Abbildung 2.2-11 DynamicTextFormField	23
Abbildung 2.2-12 Konstruktor des Screens	23
Abbildung 2.2-13 Funktionalität des Widgets	24
Abbildung 2.2-14 Aufruf des Widgets	24
Abbildung 2.2-15 Aussehen LamacoinInput.....	24
Abbildung 2.2-16 Konstruktor	25
Abbildung 2.2-17 Es sind nur bestimmte Zeichen erlaubt	25
Abbildung 2.2-18 Validator von LamacoinInput.....	25
Abbildung 2.2-19 Aufruf.....	25
Abbildung 2.2-20 Aussehen NumberInput.....	26
Abbildung 2.2-21 Konstruktor	26
Abbildung 2.2-22 Validator	26
Abbildung 2.2-23 Aufruf.....	27
Abbildung 2.2-24 Screen zum Erstellen eines neuen Tasksets	28
Abbildung 2.2-25 Controller im Screen	28
Abbildung 2.2-26 (Weiter) Button.....	29
Abbildung 2.2-27 Die Funktion buildWholeTaskset	29
Abbildung 2.2-28 Screen zum Verwalten der Task – Liste	30
Abbildung 2.2-29 Screen zum Hinzufügen eines neuen Tasks	31
Abbildung 2.2-30 Ordnerstruktur.....	32
Abbildung 2.2-31 Konstruktor	33
Abbildung 2.2-32 Jeder Screen braucht Controller	33
Abbildung 2.2-33 Überprüfung, ob neuer Task oder editierter	34
Abbildung 2.2-34 Angaben Custom – Appbar	34
Abbildung 2.2-35 Angaben für den Body	35
Abbildung 2.2-36 Eingabe für Lamacoins.....	35
Abbildung 2.2-37 Button (Task hinzufügen).....	36
Abbildung 2.2-38 Was ist anpassbar?	38
Abbildung 2.2-39 fromJSON in task.dart	38
Abbildung 2.2-40 toJSON in task.dart	39
Abbildung 2.2-41 Aufruf der toJSON Funktion.....	39
Abbildung 2.2-42 Task muss vom Validator erkennbar sein.....	39
Abbildung 2.2-43 Enum in task.dart.....	40
Abbildung 2.2-44 Freigabe für bestimmte Fächer	40
Abbildung 2.2-45 Screen hinzufügen	40
Abbildung 2.2-46 Screen hinzufügen	41
Abbildung 2.3-1: All Tasksets Screen.....	42
Abbildung 2.3-2: Alle Tasksets (standart in Pool).....	43
Abbildung 2.3-3: Taskset Pool	44
Abbildung 2.3-4: Taskset Manage Bloc	44
Abbildung 2.3-5: Taskset Pool bearbeiten	45
Abbildung 2.3-6: Taskset Pool bearbeiten mit einer Liste	45
Abbildung 2.3-7: Delete Taskset Event.....	46
Abbildung 2.3-8: Upload Taskset Event.....	46
Abbildung 2.3-9: Menu Screen.....	47
Abbildung 2.3-10: Server Einstellung Screen	48
Abbildung 2.3-11: Insert Server Settings Methode	49

Abbildung 2.3-12: Server Settings Repository.....	49
Abbildung 2.3-13: SSH Client.....	50
Abbildung 2.3-14: Server Settings Check	50
Abbildung 2.3-15: Delete Funktion	50
Abbildung 2.3-16: Upload Funktion	50
Abbildung 2.3-17: Pfad der Files	51
Abbildung 2.3-18: Download Files.....	51
Abbildung 2.4-1 : prinzipielle Schema für Interaktion	52
Abbildung 2.4-2: Das Schema für TTS Bloc Interaktion.....	53
Abbildung 2.4-3: TTS Bloc Datei	53
Abbildung 2.4-4: Event Stream in der Bloc Datei	54
Abbildung 2.4-5: erlaubte States für TTS Bloc Datei	55
Abbildung 2.4-6: automatisch vorlesen beim Aufmachen.....	55
Abbildung 2.4-7: Bloc Konzept Umsetzung. Wichtige Abschnitte	56
Abbildung 2.4-8: Bloc Konzept Umsetzung Four Card Task Screen	57
Abbildung 2.4-9: Bloc Konzept Umsetzung Match Category Task Screen	58
Abbildung 2.4-10 readText Methode	58
Abbildung 4.4-12	59
Abbildung 4.4-13 TTS ausgeschaltet	59
Abbildung 4.4-15 TTS ausgeschaltet	60
Abbildung 2.4-14 TTS eingeschaltet	60
Abbildung 4.4-17	60
Abbildung 4.4-18 TTS Zustandsklasse	60
Abbildung 2.4-17 Button auf dem Screen.....	61
Abbildung 4.4-20	61
Abbildung 4.4-21 Four Cards in Englisch.....	61
Abbildung 2.4-20 JSON mit Sprachen.....	62
Abbildung 2.4-21 Taskklassen mit Sprachen	62
Abbildung 2.4-22 Check, ob die Sprache null ist	62
Abbildung 2.4-1: Deployment der Lama App auf ein Android Gerät.....	63
Abbildung 2.4-2: Hot Reload in Android Studio	63
Abbildung 4.1-1: mögliche Verbesserungen im Start Menü	64
Abbildung 4.1-2: mögliche Verbesserungen im Spiel.....	65
Abbildung 4.2-1Screen mit Preview – Funktion.....	66
Abbildung 4.4-1	68
Abbildung 4.4-2	68
Abbildung 4.4-3	68
Abbildung 4.4-4	69
Abbildung 4.4-5	69
Abbildung 4.4-6	69

1 Anforderungen

In diesem Abschnitt werden die neuen Funktionalitäten in Form von in Form- und Lastenheft definierten Anforderungen erläutert.

1.1 Anforderung 1: Tap the Lama

In dem zu entwickelnden Spiel sollen herunterfallende Lama-Köpfe im richtigen Moment getroffen werden. Dies soll geschehen, indem der Benutzer in dem Moment die unteren Lamakopf-Buttons drückt, in dem der herunterfallende Lamakopf über dem Lamakopf-Button liegt (siehe Abbildung 1.1-1)

In dem Spiel soll es eine Lebensanzeige sowie einen Punktestand geben (siehe Abbildung 1.1-1). Treffer sollen den Punktestand erhöhen, wobei besonders lange Kombinationen ohne fehlerhaftes Drücken die erhaltenen Punkte pro Treffer noch weiter erhöhen sollen. Fehlerhaftes Drücken der Lama-Buttons oder vorbei fallende Lama-Köpfe sollen Lebenspunkte von der Lebensanzeige abziehen. Das Spiel soll so lange gehen, bis die Lebensanzeige komplett rot ist, also keine Lebenspunkte mehr vorhanden sind. Mit erhöhter Punktzahl soll die Geschwindigkeit steigen, mit der die Lama-Köpfe herunter fallen, was schrittweise den Schwierigkeitsgrad erhöhen soll. Um den Schwierigkeitsgrad zusätzlich zu steigern, soll es Lamaköpfe geben, die nicht getroffen werden sollen (siehe Abbildung 1.1-1). Werden diese dennoch vom Benutzer getroffen, sollen Lebenspunkte abgezogen werden. Weiterhin soll es in dem Spiel die Möglichkeit geben, die Lamaköpfe mit unterschiedlicher Geschwindigkeit herunter schnellen zu lassen, sodass der Schwierigkeitsgrad weiter gesteigert werden kann. Das Spiel soll mit der Spiele Engine „Flame“ in der Version 1.1.1 implementiert werden.

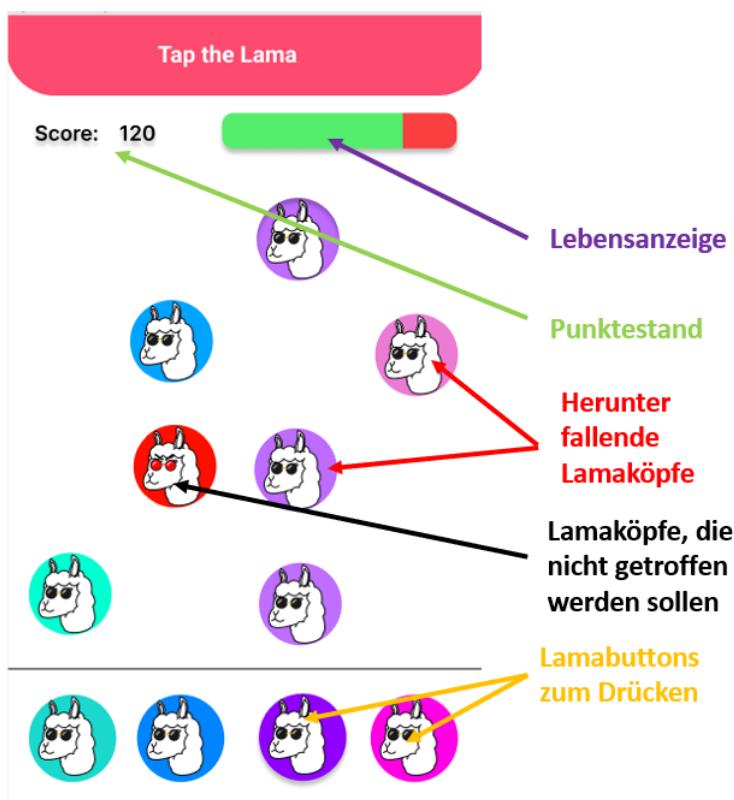


Abbildung 1.1-1: Tap the Lama Mockup

Anforderungen

Die bisherigen in der Lama-App programmierten Spiele sind jeweils nach einem "Hit" beendet, also dann, wenn der User einen Fehler im Spiel macht. Das neu zu implementierende Spiel soll dem Nutzer die Möglichkeit geben, so lange zu spielen, bis seine Lebensanzeige aufgebraucht ist. Dies soll die Motivation steigern, Lama-Coins für dieses Spiel auszugeben, da man das Spiel voraussichtlich länger spielen kann. Weiterhin soll dieses Spiel die Reaktion des Spielers schulen, da die Geschwindigkeit und somit der Schwierigkeitsgrad des Spiels sukzessiv steigt.

1.2 Anforderung 2: Taskset Erstellung

Das Erstellen von Tasksets soll auch innerhalb der App möglich sein. Der Nutzer ist somit nicht mehr darauf angewiesen, die JSON Dateien in einem textuellen Editor erstellen zu müssen. Er kann somit seine Eingaben bequem in vordefinierten Feldern machen und die App generiert daraus die notwendige JSON Datei.

Die Erstellung sollte für den Nutzer möglichst intuitiv sein und Fehleingaben verhindern. Die erstellten Tasksets sollen sich fehlerfrei nutzen lassen.

Der Admin hat einen Screen, in dem er seine Eingaben für das Taskset machen kann.

The screenshot shows a mobile application interface for creating a task set. At the top, there is a blue header bar with the title 'Taskset erstellen' and a back arrow icon. Below the header, there are two input fields: 'Tasksetname' and 'Kurzbeschreibung', each with a horizontal line for text entry. Under these fields, there are two dropdown menus labeled 'Klassenstufe' and 'Fach', both with the placeholder text 'Klasse auswählen' and a downward arrow icon indicating they are dropdowns. At the bottom of the screen is a blue button labeled 'Weiter' (Continue).

Abbildung 1.2-1 Screen zum Erstellen eines Tasksets

Anforderungen

Mit einem Klick auf (Weiter) gelangt er zur nächsten Seite und kann von dort aus die Liste seiner Tasks für das Taskset hinzufügen.

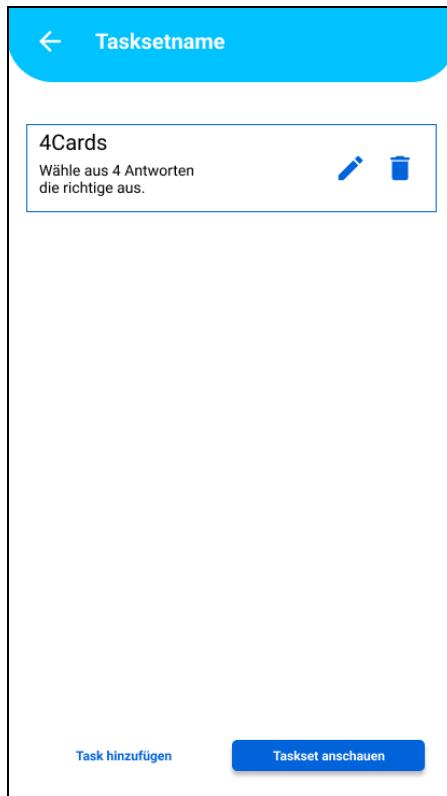


Abbildung 1.2-2 Screen zum Hinzufügen von Tasks

Ist er mit seinen Eingaben zufrieden, schickt er das Taskset über den Button (Taskset anschauen) ab und generiert sich so seine JSON Datei.

1.3 Anforderung 3: Aufgabenverwaltung

Die Tasksets die in dem vorherigen Abschnitt erstellt wurden sollten auch extern gespeichert und verwaltet werden. Dies ist bereits möglich, aber jeweils nicht in der App. Das bedeutet, dass der Admin seine Tasksets auf einem, von ihm betriebenen, Server ablegen können, sollten. Über diesen können die Schüler der Lehrenden Person auf diese Tasksets zugreifen.

Dabei soll es die Möglichkeit geben, einzelne Tasksets auf dem Server pausieren zu können, sodass diese den Schülern nicht mehr zur Verfügung stehen. Natürlich können die vom Admin pausierten Tasksets jederzeit wieder durch diesen freigeben werden. Diese Möglichkeiten soll dem Admin ebenfalls durch die GUI zur Verfügung gestellt werden.

Ein Server könnte als Schulserver von allen Lehrern gemeinsam genutzt werden. Aufgaben werden in Klassen Pools strukturiert, die wiederum die Tasksets beinhalten sollen. Der Server soll ebenfalls von allen Lehrern verwaltet werden können.

Anforderungen

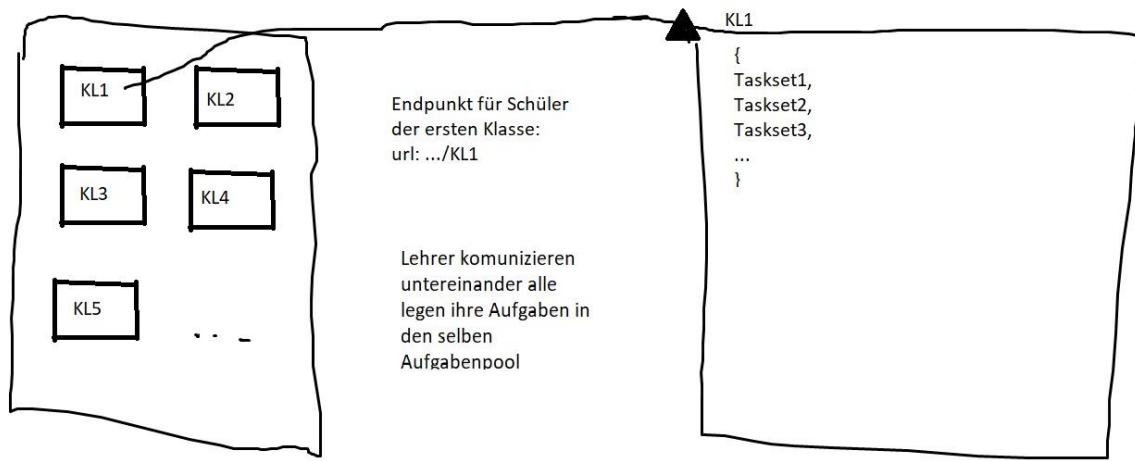


Abbildung 1.3-1 Modell Serverstruktur

In Abbildung 1.3-1 wird im Groben dargestellt, wie sich die Struktur des Servers darstellt. Dabei soll die Tasksets für die Schüller der ersten Klasse unter KL1 abgelegt werden, die der zweiten Klasse unter KL2 und dementsprechend für die anderen Klassen auch.

Unter diesen Klassen sollen dann die Tasksets abgelegt werden.

1.4 Anforderung 4: Text To Speech

Durch ein Text to Speech Feature, soll es möglich sein: Aufgabenstellungen, Texte und Fragen vorzulesen, wenn Schüler z. B. noch nicht lesen können. Der Schüler soll Fragen oder Aufgabenstellungen vorgelesen bekommen, wenn der Schüler die Aufgabe öffnet. Außerdem soll der Schüler die Möglichkeit haben sich die Fragen oder Aufgabenstellungen erneut anzuhören, falls sie diese nicht verstanden haben, oder sie erneut hören wollen. Auch sollte es möglich sein sich mögliche Antworten vorlesen zu lassen. Es sollte auch möglich sein das Feature auszuschalten, falls man es nicht benutzen will. Außerdem soll ein Cloud-basiertes Plugin vermieden werden, da dies datenschutzrechtliche Probleme darstellen könnte.

2 Praktische Umsetzung

In diesem Abschnitt wird die Anforderungsumsetzung beschrieben. Das bedeutet, dass von allen neuen Features sowohl die Umsetzung der grafischen Benutzeroberfläche, als auch die Umsetzung der Programmlogik beschrieben wird. Dabei wird nicht im Detail jede Codezeile erklärt. Stattdessen werden Programmabschnitte erklärt, welche wegen höherer Komplexität einer Erklärung bedürfen. Bei der Beschreibung der praktischen Umsetzung der Features wird sowohl auf die Implementierung als auch auf das Testen eingegangen.

2.1 Tap the Lama

2.1.1 Auswahl des Spiels und Startmenü

Tap the Lama lässt sich wie alle anderen Spiele auch über die Spiele-Liste in der App auswählen. Dazu wurde im Verzeichnis `lama_app/lib/app/screens` in der `game_list_screen.dart` Datei ein weiterer Eintrag in der statischen Spiele Liste angelegt. Hierzu wurde der Spieldatitl, eine Beschreibung und die Kosten des Spiels in Lamacoins angegeben, wie sich in Abbildung 2.1-1 erkennen lässt.

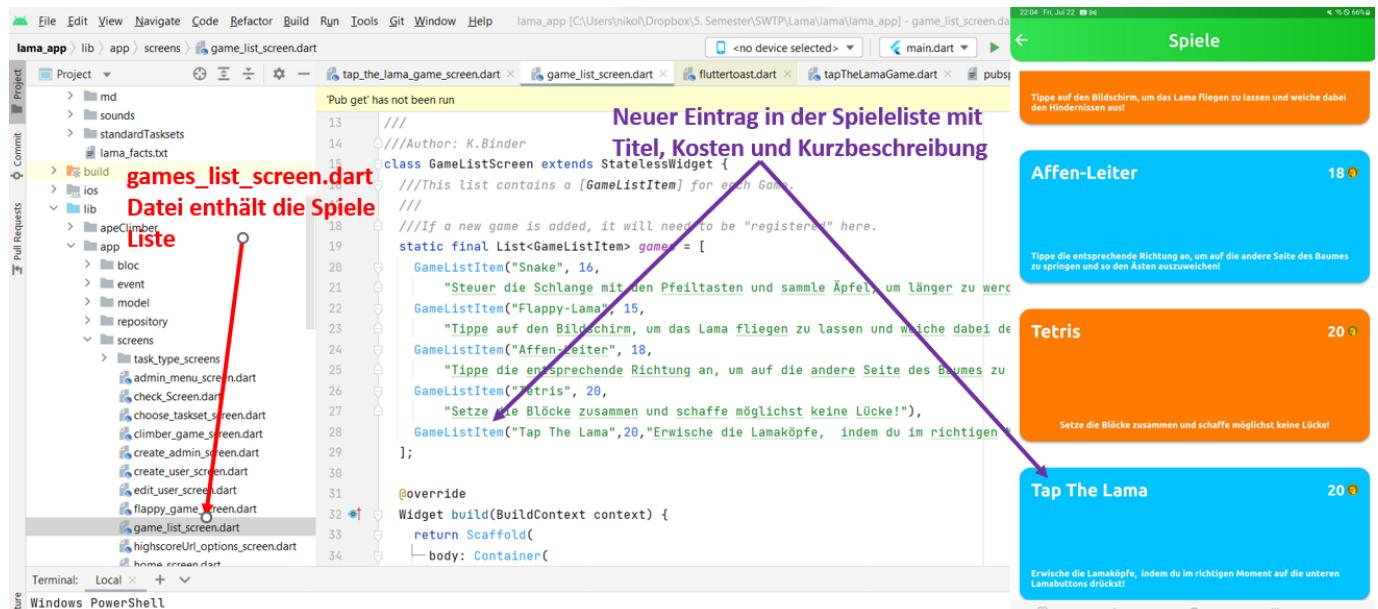


Abbildung 2.1-1: Neuen Eintrag in Spieliste einfügen

In der selben Datei (`game_list_screen.dart`) befindet sich eine Methode namens `navigateToGame()`, mit welcher zu dem in der Liste ausgewählten Spiel navigiert werden kann. In dieser Methode wird über eine switch-case Operation das jeweilige Spiel-Hauptmenü ausgewählt. Bei Tap the Lama werden 3 Parameter übergeben, die wichtig für Highscoreinformationen und Benutzerinformationen in dem Spiel sind (siehe Abbildung 2.1-2). Dabei handelt es sich um folgende:

- `userRepository` → enthält Daten des aktuellen Benutzers
- `userHighScore` → enthält den Highscore für Tap the Lama des aktuellen Benutzers
- `allTimeHighScore` → enthält den Benutzer übergreifenden höchsten Score, der jemals erzielt wurde

```

games_list_screen.dart Datei enthält die Spiele Liste
admin_menu_screen.dart
check_Screen.dart
choose_taskset_screen.dart
climber_game_screen.dart
create_admin_screen.dart
create_user_screen.dart
edit_user_screen.dart
flappy_game_screen.dart
game_list_screen.dart
highscoreUrl_options_screen.dart
home_screen.dart

Übergebene drei Parameter für die Score und User Verwaltung im Spiel
break;
case "Tap The Lama":
    int? userHighScore = await userRepository!.getMyHighscore(5);
    int? allTimeHighScore = await userRepository.getHighscore(5);
    gameLaunch =
        TapTheLamaScreen(userRepository, userHighScore, allTimeHighScore);
    break;
default:
    throw Exception("Trying to launch game that does not exist");
}

```

Abbildung 2.1-2: Auswahl des Spiels mit Übergabeparametern

Wenn das Spiel ausgewählt wurde, wird ein Screen für das Hauptmenü erstellt. Dies wird dadurch realisiert, dass ein Objekt der Klasse TapTheLamaScreen mit den drei vorher genannten Parametern instanziert wird (siehe Abbildung 2.1-2).

Diese Klasse ist auffindbar in `lama_app/lib/app/screens/tap_the_lama_game_screen.dart`.

In dem Objekt der Klasse TapTheLamaScreen wird das Startmenü für Tap the Lama erstellt, indem bei der Erstellung eines TapTheLamaMenu Objektes wieder die Parameter userRepository, userHighScore und allTimeHighScore weitergegeben werden, wie es in Abbildung 2.1-3 erkennbar ist.

```

In dieser Datei ist der Screen für Tap the Lama
Check_Screen.dart
choose_taskset_screen.dart
climber_game_screen.dart
create_admin_screen.dart
edit_user_screen.dart
flappy_game_screen.dart
game_list_screen.dart
highscoreUrl_options_screen.dart
home_screen.dart
safty_quastion_screen.dart
snake_screen.dart
tap_the_lama_game_screen.dart
task_screen.dart
taskset_option_screen.dart
tetris_game_screen.dart
user_login_screen.dart
user_management_screen.dart
user_selection_screen.dart
userlist_url_screen.dart

Pub get' has not been run
//Main Menu
const TapTheLamaScreen(
    this.userRepository, this.userHighScore, this.allTimeHighScore);

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            automaticallyImlyLeading: true,
            title: const Text('Tap the Lama'),
        ), // AppBar
        backgroundColor: Colors.white,
        body: Menu(context, userHighScore, allTimeHighScore, userRepository),
    ); // Scaffold
}

```

Übergabe der drei Parameter für die Score Verwaltung bei Erstellung des Startmenüs

Abbildung 2.1-3: Tap the Lama Screen

In der `tapTheLamaMenu.dart` Datei, welche unter dem Pfad `lama_app/lib/tapTheLama/screens` auffindbar ist, befindet sich die Programmlogik und der GUI Aufbau des Startmenüs von Tap the Lama in Codeform. In der GUI wird eine kurze Spielanleitung in Textform angezeigt. Weiterhin gibt es eine Anzeige über den persönlichen High Score sowie den größten Benutzer übergreifenden Highscore. Zudem gibt es einen Start-Button, welcher das Spiel startet, wenn der Spieler genügend Lama Coins hat. Der Start-Button löst die Methode `onPlayClicked()` aus, welche zunächst über die Variable `userRepo` kontrolliert, ob der Spieler über genügend Lama Coins verfügt. Ist dies nicht der Fall, wird

Praktische Umsetzung

der User über einen Aufruf von Navigator.pop() zum vorherigen Screen, also der Liste der Spieleauswahl, zurückgeführt. Verfügt der Spieler über genügend Lama Coins, wird das Spiel gestartet, indem über Navigator.push() der Screen für das Tap the Lama Spiel mit einer Instanziierung eines Projektes der Klasse TapTheLamaGameScreen erstellt wird. Auch bei dieser Instanziierung werden wieder die Highscores und die benutzerrelevanten Parameter weitergegeben, wie es in Abbildung 2.1-4 erkennbar ist.



Abbildung 2.1-4: Starten des Tap the Lama Spiels

Über das instanzierte Objekt der Klasse TapTheLamaGameScreen wird das Spiel mit allen nötigen GUI Elementen gemäß der Klasse TapTheLamaGame erstellt und angezeigt. Dies wird in den folgenden Abschnitten genauer erklärt.

2.1.2 Codeorganisation und Flame Implementierung

Das Spiel wurde im Wesentlichen innerhalb der lama/app/lib/tapTheLama/tapTheLamaGame.dart Datei entwickelt. Für eine bessere Codeübersicht wurden der Code in verschiedene Regionen aufgeteilt, welche sich mit gängigen IDEs „ausklappen“ lassen. In Abbildung 2.1-5 lässt sich erkennen, in welche Regionen der Code unterteilt ist und wie sich diese Coderegionen „ausklappen“ lassen, wobei das in der Abbildung gezeigte Beispiel sich auf die IDE Android Studio bezieht. Weiterhin wurde für das Spiel das Framework Flame gewählt. Um dieses Framework adäquat für das Spiel zu nutzen, erbt die Klasse TapTheLamaGame von der Klasse FlameGame. Weiterhin implementiert die Klasse TapTheLamaGame das Interface HasTappables, um die Button Komponenten im Spiel klickbar zu machen. Dies wird näher in Abschnitt 2.1.3 erläutert. Das Erben und Implementieren sind in Abbildung 2.1-5 visualisiert. Die Dokumentation zu der Klasse FlameGame lässt sich [Stand 27.07.2022] unter folgendem Link finden:

<https://docs.flame-engine.org/1.0.0/game.html>

Weiterhin ist bei der Implementierung von Flame darauf zu achten, dass dafür ein Eintrag in der pubspec.yaml Datei erstellt wurde. Zum Stand 27.07.2022 wurde folgende Flame-Version in der pubspec.yaml Datei hinterlegt:

flame: ^1.2.0

Praktische Umsetzung

```

class TapTheLamaGame extends FlameGame with HasTappables {
    Constructor And Background
    Global Variables Screen
    Global Variables Lama Buttons
    Global Variables Lama Heads
    Global Variables Game Logic
    Global Game Over Variables
    Override Methods
    Check Hits
    Visual Effects
}

```

Spiel ist im Wesentlichen in dieser Datei programmiert worden

TapTheLamaGame erbt von FlameGame und implementiert HasTappables

Verschiedene Coderegionen für bessere Codeübericht

Coderegion lässt sich über kleines Plus-Symbol ausklappen

Abbildung 2.1-5: Übersicht der TapTheLamaGame Klasse

Jede Coderegion verfügt nach „Ausklappen“ über Beschreibungen der ihr innewohnenden Methoden und Variablen in Kommentarform. Dies ist in Abbildung 2.1-6 beispielhaft für die Coderegion „Override Methods“ erkennbar.

```

## region Override Methods
//method that gets loaded one time initially when the game starts
@Override
Future<void> onLoad() async {
    super.onLoad();
    loadHighScores();
    initScoreDisplayAndLifeBar();
    initLamaButtonsWithEffects();
    initLamaHeadColumns();
}

//method that updates the game regularly differently depending on its state (game over or still ongoing)
@Override
Future<void> update(double dt) async {
    updateLifeBar();
    super.update(dt);
}

```

Ausgeklappte Coderegion

Beschreibungen der Methoden in Kommentarform

Abbildung 2.1-6: Kommentare innerhalb einer Coderegion

2.1.3 Komponenten der Spiel-Oberfläche und Animationen im Spiel

Die visualisierten Objekte in Tap the Lama wurden über Komponenten realisiert. Zum einen wurden sie über Sprite Komponenten und zum anderen über Shape Komponenten realisiert. Bei Sprite Komponenten handelt es sich um Komponenten in der grafischen Benutzeroberfläche, welche sich gut animieren lassen z.B. durch Einblenden von visuellen Effekten, das Rotieren der Sprite-Komponente und vieles mehr. Sprite Komponenten haben den Vorteil, dass sie als Inhalt eine Grafik in png Form oder in svg Form enthalten können und man somit Bilder im Spiel animieren kann (wie in diesem Beispiel die Lama-Buttons). Shape Komponenten sind grafische Komponenten, mit denen man eine einfache geometrische Form auf dem Bildschirm erstellen und animieren kann. Ein Beispiel dafür ist die Rectangle-Komponente, welche ein Rechteck visualisiert.¹

Eine ausführliche Dokumentation zu den Komponenten in Flame gibt es [Stand 27.07.2022] unter folgendem Link:

<https://docs.flame-engine.org/1.0.0/components.html>

Abbildung 2.1-7 visualisiert eine Übersicht der wichtigsten in diesem Spiel implementierten Komponenten.

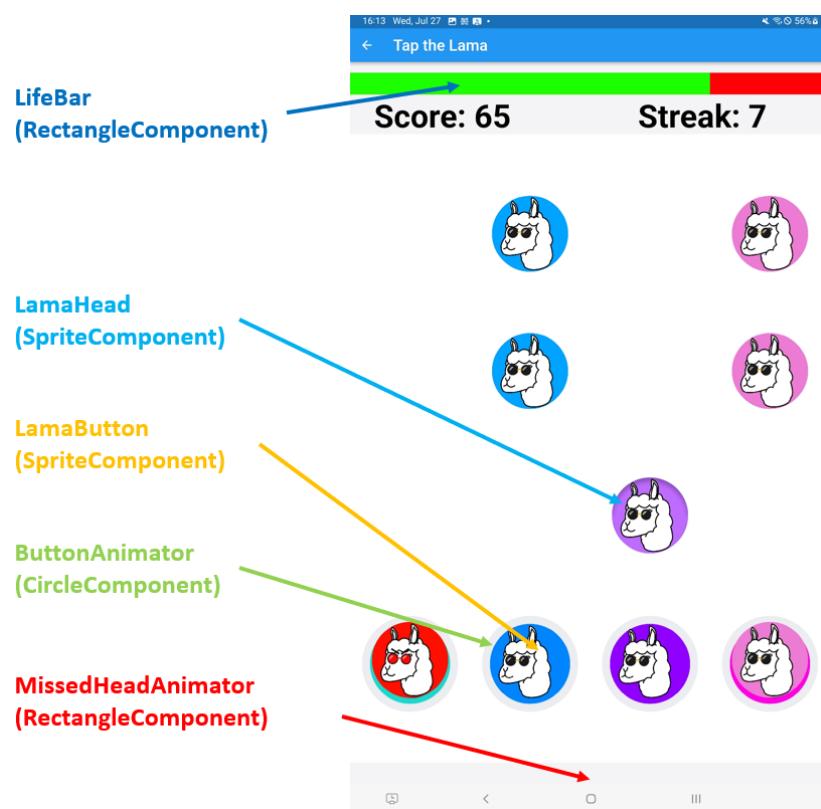


Abbildung 2.1-7: Komponenten in Tap the Lama

Bis auf die Button-Animator Komponente wurden alle in Abbildung 2.1-7 aufgezeigten Komponenten in separaten Klassen und separate Dart Dateien ausgelagert (LamaButton.dart, lamaHead.dart,

¹ Flame, o.J.

Praktische Umsetzung

LifeBar.dart, missedHeadAnimator.dart). Diese Komponenten Dateien sind in dem Verzeichnis lama/lib/tapTheLama/components zu finden.

Die Klasse LamaButton erbt von der Klasse SpriteComponent und erbt somit alle Methoden und Attribute mit der sich der LamaButton animieren lässt. Weiterhin wurde für LamaButton das mixin Tappable hinzugefügt, um den LamaButton klickbar zu machen. Dies ist wichtig, um beim Klicken des Lamabuttons bestimmte Aktionen auszuführen. Der Klick wird über eine Methode namens onTapDown() als boolean erfasst und kann somit im Spiel verwendet werden. Des Weiteren wird im Konstruktor der Klasse LamaButton ein Attribut namens priority auf 1 gesetzt. Dieses Attribut bestimmt, ob die jeweilige Komponente im Vordergrund oder im Hintergrund auf der grafischen Benutzeroberfläche ist. Ist priority beispielsweise auf 1 gesetzt, liegt diese Komponente im Hintergrund. Das bedeutet, dass alle anderen Komponenten mit höherer Priorität im Vordergrund sind (siehe Abbildung 2.1-8).

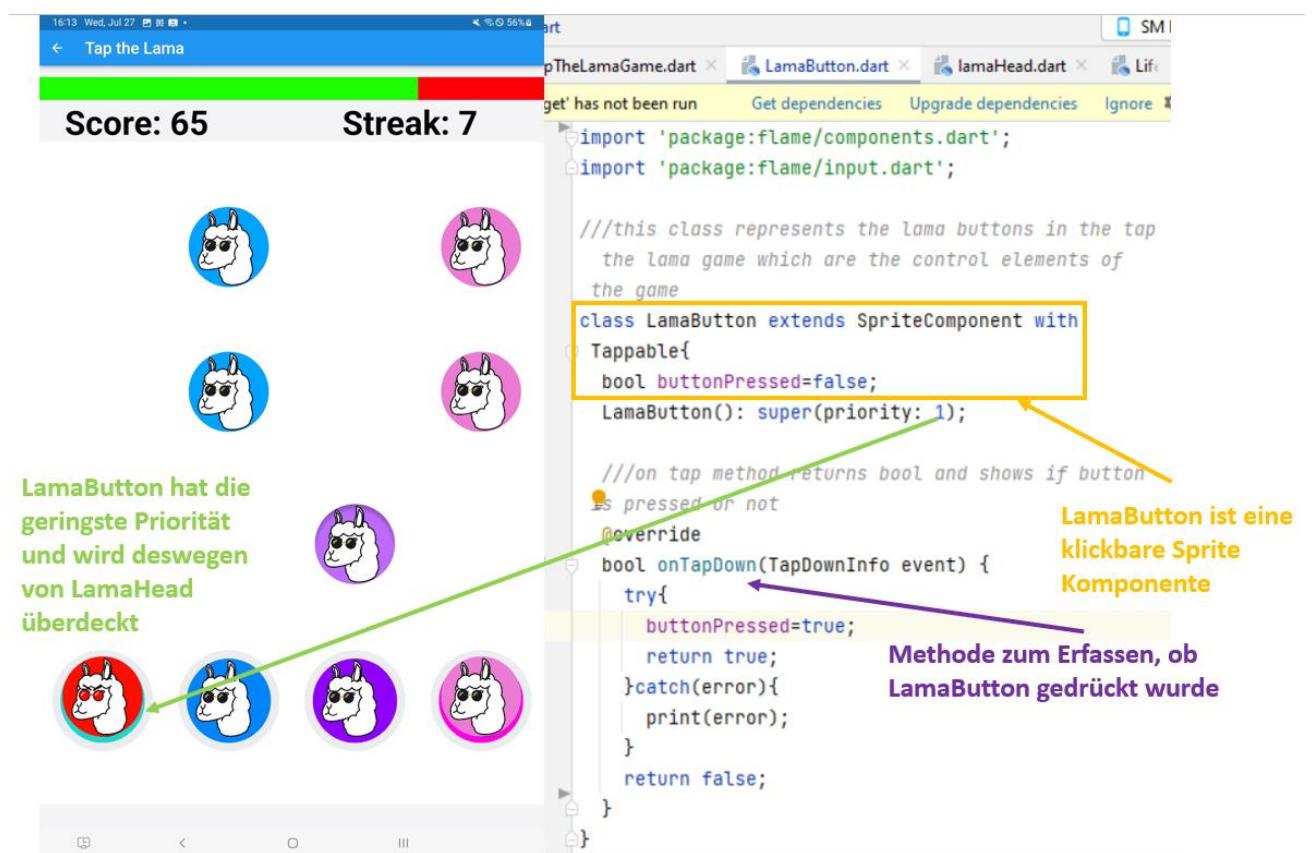


Abbildung 2.1-8: LamaButton als klickbare Sprite Komponente

Die Lamabuttons werden in der Klasse TapTheLamaGame als globale Variablen in der Coderegion „Global Variables Lama Buttons“ deklariert und schließlich in der Coderegion „Initialise Score Counter, Life Bar, Lama Buttons and Button Animators“ initialisiert. Die Initialisierung der LamaButtons erfolgt über die Methode initLamaButton(). In dieser Methode wird der LamaButton mit den vererbten Attributen der SpriteComponent Klasse aufgebaut wie in Abbildung 2.1-9 erkennbar ist.

Praktische Umsetzung

```
// initialise all LamaButtons
initLamaButton(lamaButtonTurkis, xPosTurkis, lamaButtonImageTurkis);
initLamaButton(lamaButtonBlue, xPosBlue, lamaButtonImageBlue);
initLamaButton(lamaButtonPurple, xPosPurple, lamaButtonImagePurple);
initLamaButton(lamaButtonPink, xPosPink, lamaButtonImagePink);
```

Aufruf der initLamaButton()
Methode für alle vier
LamaButtons mit
Übergabeparametern

```
///method to initialise a single lama button
void initLamaButton(
    LamaButton button, double xPos, String imageSource) async {
    button.sprite = await loadSprite(imageSource);
    button.size = Vector2(lamaButtonSize, lamaButtonSize);
    button.y = yPosButtons;
    button.x = xPos;
    button.anchor = Anchor.center;
    add(button);
}
```

Größe des LamaButtons
x und y Position des jeweiligen Buttons
auf dem Bildschirm
Schwerpunkt des Lamabuttons auf den
Mittelpunkt des Buttons legen

Button zum Bildschirm hinzufügen

Abbildung 2.1-9: Initialisierung der LamaButtons

Die LamaHead Komponente in der lamaHead.dart Datei hat verschiedene Attribute, welche wichtig für das spätere Implementieren der Spielelogik sind. Zu diesen Attributen zählt beispielsweise der bool-Wert isExisting, welcher aussagt, ob dieser LamaHead später auf dem Spielfeld mit einer png angezeigt wird oder leer ist. Weiterhin gibt es die bool-Werte isAngry und isHittable. Das Attribut isAngry sagt aus, ob der Lamakopf rot oder nicht rot ist. Das Attribut isHittable sagt aus, ob der LamaHead sich in der Spielfeld Region befindet, in der er von einem LamaButton erfasst, also weggedrückt werden kann (siehe Abbildung 2.1-10).

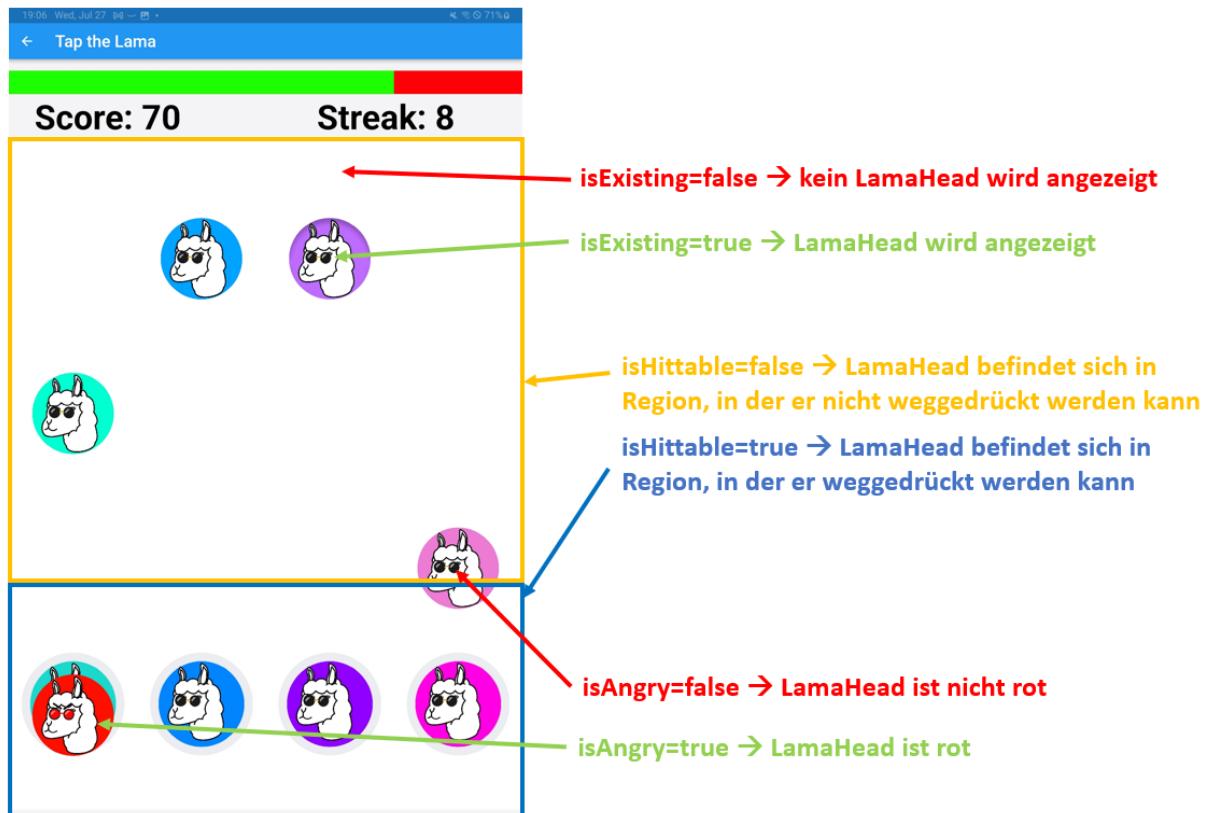


Abbildung 2.1-10: Visualisierte Attribute der LamaHead Komponenten

Praktische Umsetzung

Die LamaHead Komponenten sind in vier Spalten angeordnet, die jeweils zu den 4 dazugehörigen LamaButtons ausgerichtet sind. Demensprechend werden die LamaHead Komponenten auch über Schleifenkonstrukte spaltenweise erstellt, wie sich in Abbildung 2.1-11 erkennen lässt.

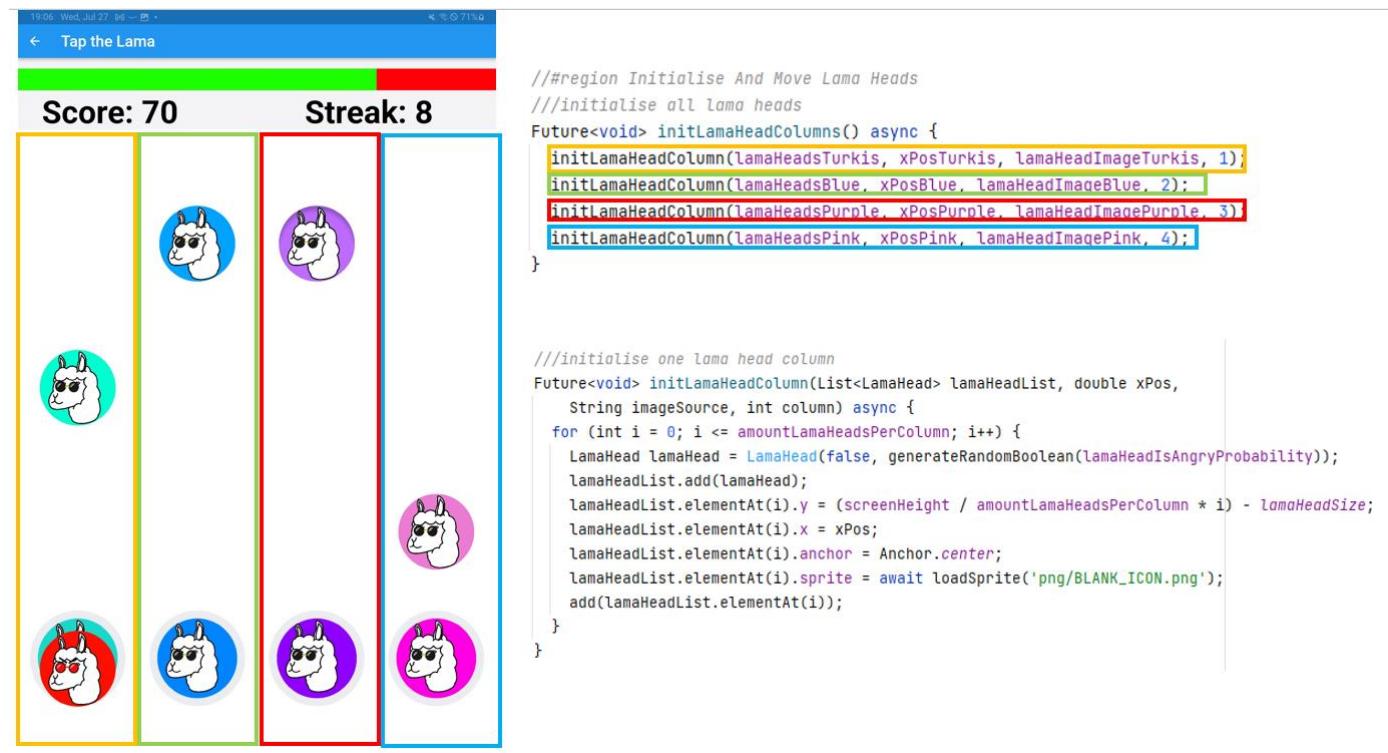


Abbildung 2.1-11: Initialisierung der LamaHead Spalten

2.1.4 Spielelogik und Animationen

Die Klasse FlameGame enthält bestimmte Methoden, die im Tap the Lama Spiel überschrieben werden mussten. Zu diesen Methoden zählen onLoad(), update() und render(). Diese Methoden sind in der Klasse TapTheLamaGame in der Coderegion „Override Methods“ aufzufinden. Die onLoad() Methode ist der Einstiegspunkt des Programmes. Sie wird aufgerufen, sobald das Spiel gestartet wird. In ihr werden zunächst die vorher übergebenen Highscores in das Spiel geladen. Im Anschluss werden alle grafischen Elemente im Spiel initialisiert wie beispielsweise die in Abschnitt 2.1.3 aufgezeigten Komponenten (siehe Abbildung 2.1-12).

```

//# region Override Methods
//method that gets loaded one time initially when the game starts
@Override
Future<void> onLoad() async {
    super.onLoad();
    loadHighScores(); Laden der vorher übergebenen HighScores
    initScoreDisplayAndLifeBar();
    initLamaButtonsWithEffects();
    initLamaHeadColumns(); Initialisieren der verschiedenen Komponenten der Spieleoberfläche
}

```

Abbildung 2.1-12: onLoad() Methode

Praktische Umsetzung

Die update() Methode ist die wichtigste Methode von FlameGame. Dies ist dadurch bedingt, dass sie permanent neu aufgerufen wird. Durch das immer wieder erneute Aufrufen können in dieser Methode also alle Funktionalitäten zusammengefasst werden, welche die Spielelogik und Animationen betreffen. In der in Tap the Lama überschriebenen update() Methode, wird innerhalb der Methode mit dem boolean gameOver immer unterschieden, ob das Spiel schon vorbei ist oder nicht, da so unterschiedliche Aktionen ausgeführt werden müssen. Wenn das Spiel nicht vorbei ist, werden zunächst mit checkHits() die Treffer geprüft. Danach werden alle LamaHeads spaltenweise durch Veränderung ihrer y-Position mit der Methode moveLamaHeads() bewegt. Im Anschluss werden Spieleparameter mit der Methode updateSpeedAndProbabilityParameters() aktualisiert und zum Schluss mit der Methode checkIfGameOver() eine Prüfung vollzogen, ob man keine Lebenspunkte mehr hat und somit das Spiel vorbei ist und der boolean gameOver auf true gesetzt werden kann (siehe Abbildung 2.1-13).

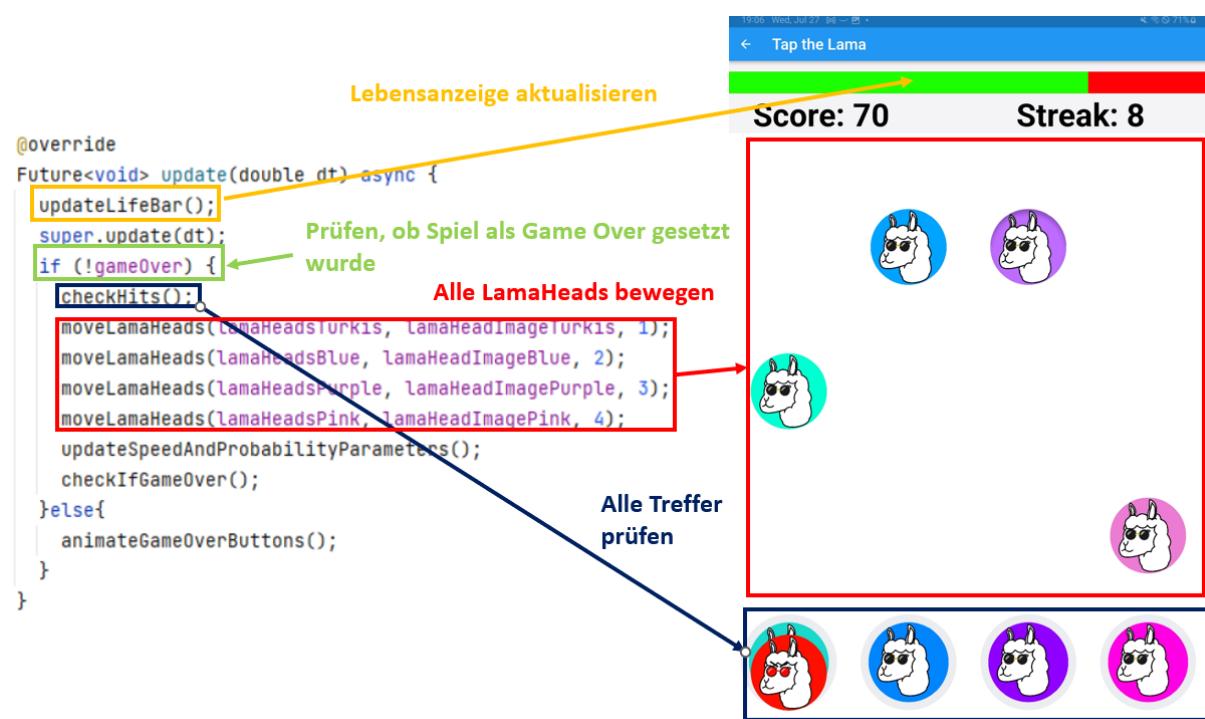


Abbildung 2.1-13: update() Methode

Innerhalb der update() Methode sind die wichtigsten Methoden checkHits() und moveLamaHeads(), da diese die Trefferlogik und Animation des Spieles gewährleisten.

In der checkHits() Methode wird für alle Spalten jeweils die checkSingleHit() Methode aufgerufen. In dieser checkSingleHit() Methode wird geprüft, ob bei Betätigung des jeweiligen LamaButtons eine Überschneidung mit einem LamaHead existiert. Dazu wird geprüft, welcher LamaHead sich in dem „wegdrückbaren“ Bereich befindet, also welcher LamaHead den boolean isHittable auf true gesetzt hat. Im Anschluss wird geprüft, ob der „wegdrückbare“ LamaHead auf dem Bildschirm angezeigt wird, also den boolean isExisting auf true gesetzt hat. Danach wird eine Unterscheidung getroffen, ob der LamaHead rot ist oder nicht, was durch das boolean isAngry definiert wird. Ist der LamaHead rot, wird Leben abgezogen und der streakCounter auf den Wert 0 gesetzt, was bedeutet, dass die fehlerfreie Serie beim Spielen abgerissen ist (siehe Abbildung 2.1-14).

Praktische Umsetzung

```

Future<void> checkSingleHit(List<LamaHead> lamaHeads, LamaButton lamaButton,
    CircleComponent animatorButton) async {
    //initialise a copy of the lama head which currently is the nearest to the lama button
    //specified by isHittable attribute)
    LamaHead lamaHeadCopy =
        lamaHeads.firstWhere((element) => element.isHittable);
    //checks if the lama head is existing (has an lama head image on the screen)
    if (lamaHeadCopy.isExisting) {
        //decrease streak and life when an angry (/red) lama head is pressed
        if (lamaHeadCopy.isAngry) {
            lifePercent -= lifeDecreaserRedLamaHit;
            streakCounter = 0;
            //showToast(LamaColors.redAccent, "Aua :(");
        }
        //if lama head isn't a red lama
        else {
    
```

Prüfen, ob der LamaHead im „wegdrückbaren“ Bereich liegt

Prüfen, ob der LamaHead auf dem Bildschirm angezeigt wird

Wenn der LamaHead rot ist, Leben abziehen und den Streak nullen

Abbildung 2.1-14: checkSingleHit() Methode

Wenn es sich bei dem Treffer allerdings um keinen roten LamaHead handelt, wird der Treffer je nach Qualität, also nach Überschneidungsgrad von LamaHead und LamaButton bewertet. Dies geschieht, indem man die y-Position des Mittelpunktes des LamaHeads mit der des LamaButtons beim Betätigen des LamaButtons vergleicht. Dazu wird mit einem relativen Positionswert kalkuliert. Für einen guten Treffer (, ein Treffer mit hoher Überschneidung,) wurde dies in Abbildung 2.1-15 visualisiert.

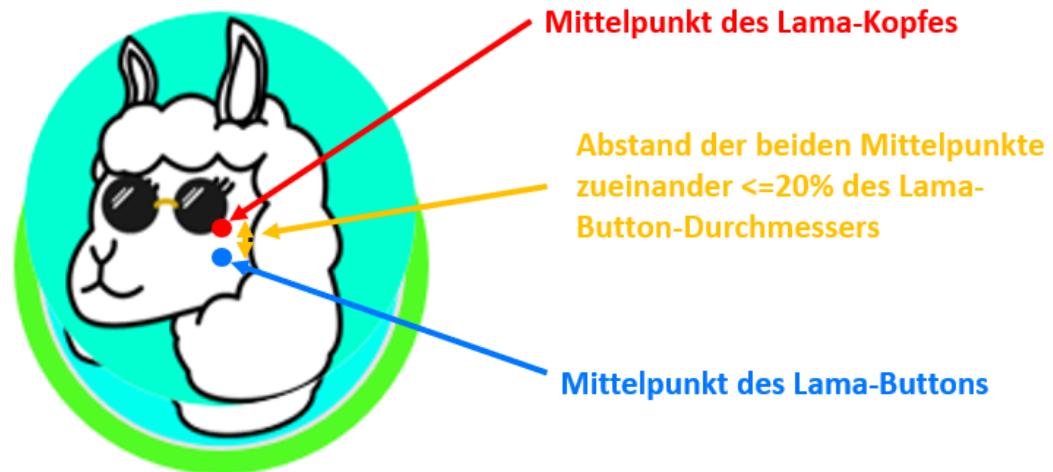


Abbildung 2.1-15: Treffer Überschneidung bei guten Treffern

Ein guter Treffer wird dann erzielt, wenn es eine maximale Abweichung von 20% des LamaButton Durchmessers von LamaButton Mittelpunkt zu LamaHead Mittelpunkt gibt. Wird ein guter Treffer erzielt, werden bestimmte Aktionen ausgeführt, welche in Abbildung 2.1-16 visualisiert sind.

Praktische Umsetzung

```

if (lamaHeadCopy.y >= yPosButtons - lamaButtonSize * 0.2 &&
    lamaHeadCopy.y <= yPosButtons + lamaButtonSize * 0.2) {
    //animate the lama button with flash effect
    flashButton(lamaButton, buttonAnimatorColorGreen, animatorButton);
    //increment score
    score += scoreIncrementerGoodHit;
    //check if there is a streak and increment streakScore if so
    if(streak){streakScore+=scoreIncrementerGoodHit;}
    //increment the streakCounter and check if lifeBar gets recovered
    streakCounter++;
    recoverLifeBarCheck();
}

Abfrage ob beide Mittelpunkte zueinander <20% Abweichung haben

Button wird grün umrandet animiert
Score wird inkrementiert
Streak und Streak-Score werden inkrementiert, Lebensanzeige wird ggf. aktualisiert

```

Abbildung 2.1-16: Aktionen bei einem guten Treffer



Für mittelgute und schlechte Treffer wurde die Logik ähnlich implementiert. Dies lässt sich aus dem restlichen Code der checkSingleHit() Methode entnehmen.

In der moveLamaHeads() Methode werden alle LamaHeads bewegt. Dies geschieht, indem ihre y Position inkrementiert wird. Gelangt ein angezeigter/existierender und nicht roter LamaHead am Ende des Bildschirms an, ohne also vorher weggedrückt worden zu sein, zieht dies Lebenspunkte ab und setzt den Streak zurück. Alle am Ende des Bildschirms anlangenden LamaHeads werden wieder an die oberste Position des Bildschirms zurückgesetzt. Hierbei wird durch Zufallswerte bestimmt, ob die LamaHeads angezeigt werden und ob sie rot sind (siehe Abbildung 2.1-17).

```

Future<void> moveLamaHeads(
    List<LamaHead> lamaHeadList, String imageSource, int column) async {
    for (int i = 0; i <= amountLamaHeadsPerColumn; i++) {
        //check if lama head is at lower display end
        if (lamaHeadList.elementAt(i).y >= screenHeight + lamaHeadSize) {
            //generate new lama head with offset at upper display end
            lamaHeadList.elementAt(i).y = 0 - lamaHeadSize / 4*3;
            //decrease life and end streak when lama reaches lower display end
            untapped
        }
        if (lamaHeadList.elementAt(i).isExisting &&
            !lamaHeadList.elementAt(i).isAngry) {
            lifePercent -= lifeDecreaserStandard;
            streakCounter = 0;
            flashMissedHeadAnimator();
        }
        //set newly generated lama head with set probability to an angry
    (red) lama
    lamaHeadList.elementAt(i).isAngry =
        generateRandomBoolean(lamaHeadIsAngryProbability);
        //logic to set a new lama just once for two columns
        //Example: if column one contains a lama at the given position
        then column 2 can't contain a lama head because game must be playable with
        two thumbs
        switch (column) {
            case 1:
                lamaHeadList.elementAt(i).isExisting =
                    generateRandomBoolean(lamaHeadAppearingProbability);
                if (lamaHeadList.elementAt(i).isExisting) {

```

LamaHead-Position wird von ganz unten nach ganz oben gesetzt

nicht weggedrückte LamaHeads ziehen Leben ab und setzen den Streak zurück

oberster Lama Head wird durch Zufallswert auf rot oder nicht rot gesetzt

oberster LamaHead wird durch Zufallswert angezeigt oder nicht

Abbildung 2.1-17: moveLamaHeads() Methode

Praktische Umsetzung

Eine weitere überschriebene Methode stellt die render() Methode dar. In dieser werden in FlameGame angezeigte Texte aktualisiert. In dem Fall von Tap the Lama sind dies der Score Text und der Streak Text. Diese werden in regelmäßigen Abständen aktualisiert, solange das Spiel noch nicht beendet ist, wie sich in Abbildung 2.1-18 erkennen lässt.

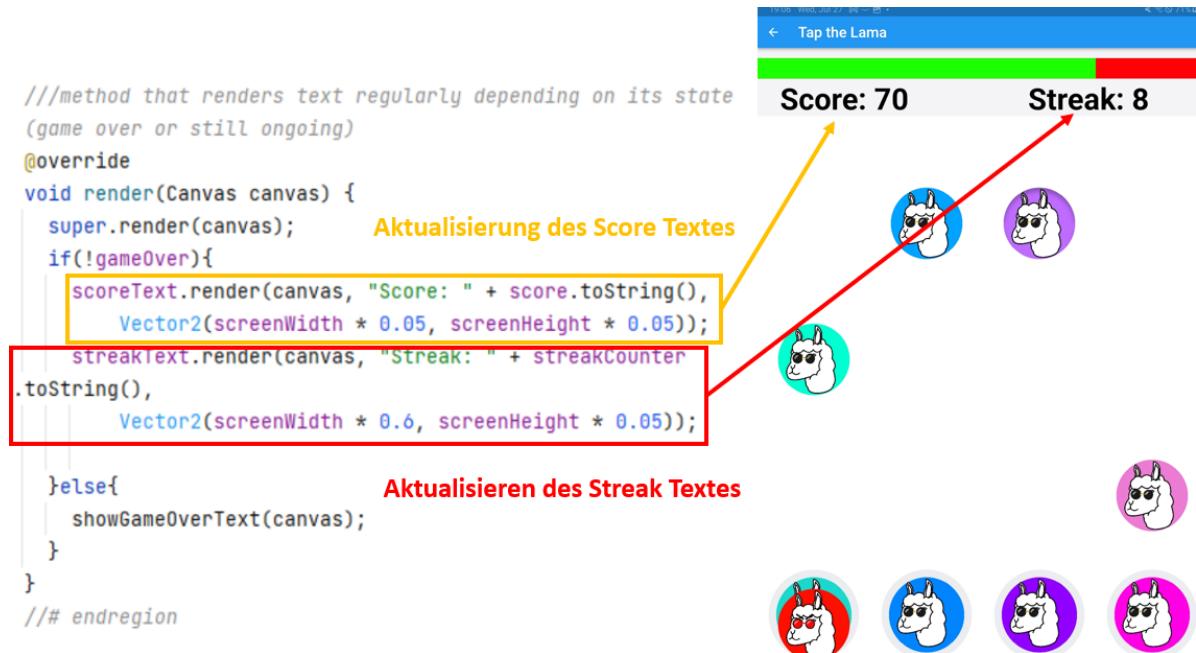


Abbildung 2.1-18: render() Methode

2.1.5 Game Over Menü

Wenn das Spiel vorbei ist, wird ein Game Over Menü angezeigt. Dieses überdeckt den Spielbildschirm und ist innerhalb der Klasse TapTheLamaGame implementiert. Das Game Over Menü ist mit Animationen und Texten versehen, weshalb es mit den überschriebenen Methoden update() und render() angesprochen bzw. aktualisiert wird. Wenn der boolean gameOver im Verlauf des Spiels auf true gesetzt wird, wird in der update() Methode eine Methode namens animateGameOverButtons() aufgerufen. Diese sorgt dafür, dass die LamaButtons vergrößert werden und sich drehen. In der render() Methode hingegen wird bei einem auf true gesetzten boolean gameOver die Methode showGameOverText() aufgerufen. In dieser wird je nach High Score und Spielausgang ein individueller Text für den Spieler angezeigt. Weiterhin werden aller relevanten Punktestände, wie beispielsweise eigener High Score, Benutzer übergreifender High Score und der eigene Punktestand aufgezeigt (siehe Abbildung 2.1-19).

Praktische Umsetzung

```

@Override
Future<void> update(double dt) async {
    updateLifeBar();
    super.update(dt);
    if (!gameOver) {
        checkHits();
        moveLamaHeads(lamaHeadsTurkis, lamaHeadImageTurkis, 1);
        moveLamaHeads(lamaHeadsBlue, lamaHeadImageBlue, 2);
        moveLamaHeads(lamaHeadsPurple, lamaHeadImagePurple, 3);
        moveLamaHeads(lamaHeadsPink, lamaHeadImagePink, 4);
        updateSpeedAndProbabilityParameters();
        checkIfGameOver();
    }  

    else{
        animateGameOverButtons();
    }
}

///method that renders text regularly depending on its state
'game over or still ongoing'
@Override
void render(Canvas canvas) {
    super.render(canvas);
    if(!gameOver){
        scoreText.render(canvas, "Score: " + score.toString(),
            Vector2(screenWidth * 0.05, screenHeight * 0.05));
        streakText.render(canvas, "Streak: " + streakCounter
toString(),
            Vector2(screenWidth * 0.6, screenHeight * 0.05));
    }  

    else{
        showGameOverText(canvas);
    }
}

```

Game Over!

Super, dein bestes Spiel bisher!

Dein Score: 610
(Score: 400 + Streak-Score: 210)

Dein Rekord: 610

High-Score: 4240

Generiert einen individuellen Game Over Text und führt alle relevanten Scores auf

Abbildung 2.1-19: Game Over Menü

2.2 Taskset Erstellung

2.2.1 Neue Ordner und Dateien

2.2.1.1 Screens

Zur besseren Übersicht, welche Screens sich auf das Adminmenü beziehen, haben wir die Ordnerstruktur im Ordner (...\\lama\\lama_app\\lib\\app\\screens) soweit angepasst, dass alle für die Taskseterstellung notwendigen Screens in dem Ordner (...\\lama\\lama_app\\lib\\app\\screens\\admin_menu_folder) und seinen Unterordnern zu finden sind.

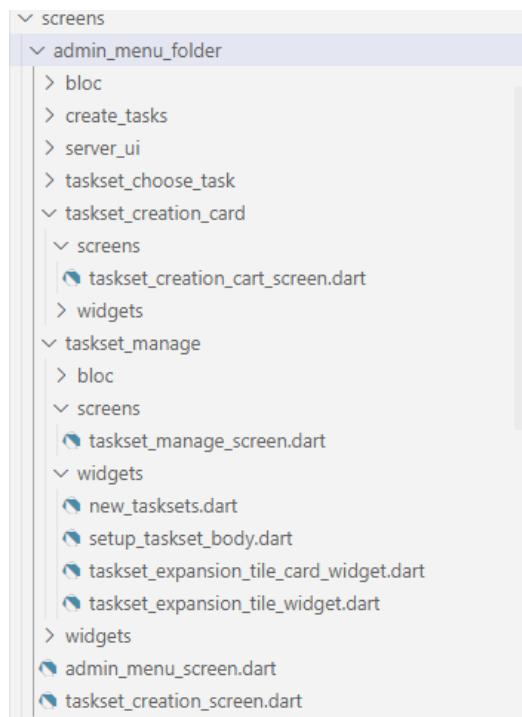


Abbildung 2.2-1 Überblick der neuen Ordner für die Screens zur Taskseterstellung

Praktische Umsetzung

2.2.1.2 CreateTasksetBloc

Hier sind auch die benötigten Bloc – Dateien zu finden. Die Ausnahmen bildet die Datei (create_taskset_bloc.dart). Diese befindet sich im Ordner (...\\lama\\lama_app\\lib\\app\\bloc\\).

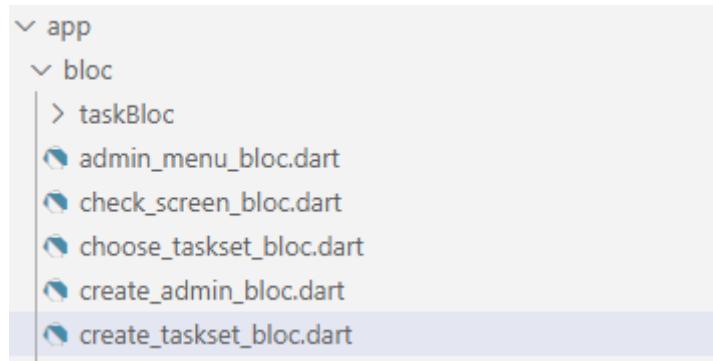


Abbildung 2.2-2 CreateTasksetBloc

In diesem Bloc wird hauptsächlich das zur Erstellung eines neuen Tasksets benötigte Taskset – Objekt verwaltet.

```
class CreateTasksetBloc extends Bloc<CreateTasksetEvent, CreateTasksetState> {
    Taskset? taskset; ← Zu erstellende Taskset
    CreateTasksetBloc({this.taskset}) : super(InitialState()) {
        on<CreateTasksetAbort>((event, emit) => _abort(event.context));
        on<EditTaskset>((event, emit) => taskset = event.taskset);
        on<AddTaskListToTaskset>((event, emit) {
            //taskset!.tasks!.addAll(event.taskList);
            taskset!.tasks = event.taskList;
        });
        on<CreateTasksetGenerate>((event, emit) => _generate());
        on<AddUrlToTaskset>((event, emit) {
            taskset!.taskurl = event.taskUrl;
            print("bloc: " + taskset!.taskurl.toString());
        });
    }
}
```

Abgefangene Events

Abbildung 2.2-3 Einblick in CreateTasksetBloc

2.2.1.3 CreateTasksetlistBloc

Die zweite wichtige Bloc – Datei verwaltet die im Taskset beinhaltete Liste an Tasks. Sie befindet sich im Ordner (lib\\app\\screens\\admin_menu_folder\\bloc\\taskset_create_tasklist_bloc.dart).

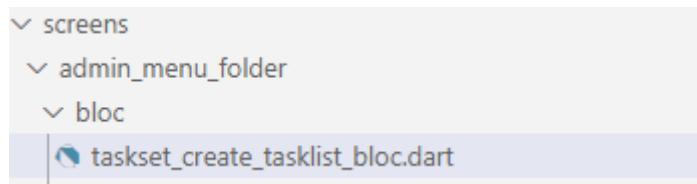


Abbildung 2.2-4 Datei CreateTasksetListBloc

Wir haben die Verwaltung der Liste von der Verwaltung des Tasksets deshalb voneinander getrennt, da es beim Editieren von einzelnen Tasks in der Liste zu Bugs kam. Diese Bugs ließen sich dadurch beheben.

Praktische Umsetzung

```
class TasksetCreateTasklistBloc extends Bloc<TasksetCreateTasklistEvent, TasksetCreateTasklistState> {
  List<Task> taskList;
  TasksetCreateTasklistBloc(this.taskList) : super(TasksetCreateTasklistInitial()) {
    on<AddToTaskList>((event, emit) {
      taskList.add(event.task);
      emit(UpdateTaskList());
    });
    on<RemoveFromTaskList>((event, emit) {
      taskList.removeWhere((element) => element.id == event.id);
      emit(UpdateTaskList());
    });
    on<EditTaskInTaskList>((event, emit) {
      taskList.removeAt(event.pos!);
      taskList.insert(event.pos!, event.task);
      emit(UpdateTaskList());
    });
  }
}
```

Abbildung 2.2-5 Überblick der Datei

2.2.1.4 State

Die States werden in einer Datei im Ordner (...\\lama\\lama_app\\lib\\app\\state) verwaltet.



Abbildung 2.2-6 State – Datei

2.2.1.5 Event

Um das Bloc – Pattern zu komplementieren, gibt es dann noch eine Event – Datei im Ordner (...\\lama\\lama_app\\lib\\app\\event). Hier werden die ganzen Events, die über Die Benutzeroberfläche ausgelöst werden können, verwaltet.

Praktische Umsetzung

2.2.2 Tasksets und Tasks

2.2.2.1 Taskset – Klasse

In der Datei (lib\app\task-system\taskset_model.dart) befindet sich die Klasse Taskset.

```
class Taskset {  
    String? name;  
    TaskUrl? taskurl;  
    String? subject;  
    String? description;  
    int? grade;  
    bool? randomizeOrder;  
    int? randomTaskAmount;  
    List<Task>? tasks;  
    bool isInPool = false;
```

Abbildung 2.2-7 Attribute der Klasse Taskset

Für die Aufgabenerstellung in der App mussten wir diese nur um diese Funktion erweitern:

```
Map<String, dynamic> toJson() => {  
    "taskset_url": taskurl?.toJson(),  
    "taskset_name": name,  
    "is_in_Pool": isInPool,  
    "taskset_subject": subject,  
    "taskset_description": description,  
    "taskset_grade": grade,  
    "taskset_randomize_order": randomizeOrder,  
    "tasks": tasks!.map((task) => task.toJson()).toList(),  
};
```

Abbildung 2.2-8 Die toJSON – Funktion

So lässt sich am Ende des Erstellungsprozesses eine JSON Datei aus dem verwalteten Taskset – Objekt generieren.

2.2.2.2 Task – Klasse

In der Datei (lib\app\task-system\task.dart) befindet sich die Task – Klasse.

```
class Task {  
    String id;  
    TaskType type;  
    int reward;  
    String lamaText;  
    int originalLeftToSolve;  
    int? leftToSolve;
```

Abbildung 2.2-9 Die Attribute der Klasse Task

Praktische Umsetzung

Hier haben wir die (id) hinzugefügt. Dies erlaubt es uns, beim Editieren von einzelnen Tasks im Erstellungsprozess eine einfachere Identifikation, welche Task editiert werden soll.

2.2.2.3 JSON – Datei

Aufgrund der Änderung in der Task – Klasse mussten auch die JSON – Dateien angepasst werden. Diese beinhalten nun einen Key (id).

```
{  
    "id": "1",  
    "task_type": "TaskType.clock",  
    "task_reward": 2,  
    "lama_text": "Wie viel Uhr ist es?",  
    "left_to_solve": 3,  
    "uhr": "vollStunde",  
    "timer": false  
},
```

Abbildung 2.2-10 Beispiel einer Task - JSON

2.2.3 Custom – Widgets

Um die Screens einheitlich(er) zu gestalten, insbesondere im Hinblick auf die Abstände der einzelnen Widgets zueinander, haben wir ein paar Widgets erstellt. Zu finden sind diese im Ordner (`lib\app\screens\admin_menu_folder\create_tasks\widgets\`). In diesem Kapitel möchten wir beispielhaft einige dieser Widgets vorstellen. Die Funktionsweise der übrigen Widgets ist ähnlich zu den vorgestellten und wie diese zu nutzen sind, kann in den jeweiligen Screens nachgeschaut werden.

2.2.3.1 *DynamicTextField*

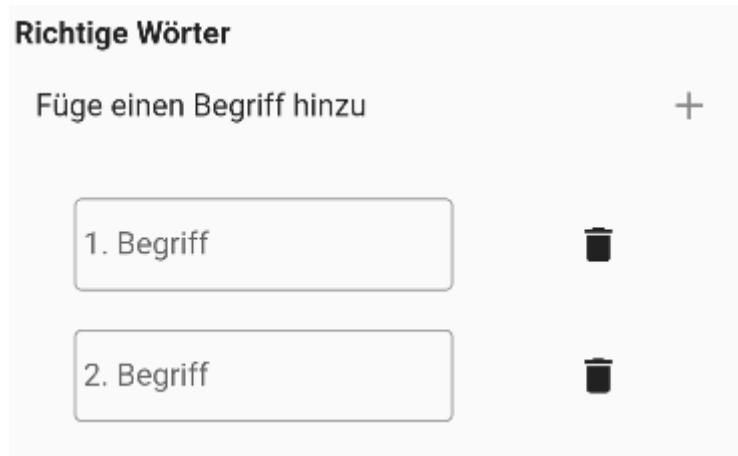


Abbildung 2.2-11 *DynamicTextField*

Für einige Tasks ist es notwendig, dem Nutzer eine dynamische Anzahl an Begriffen (siehe z.B. MarkWords) eingeben zu lassen. Hierfür nutzen wir das Widget *DynamicTextField* aus der Datei (`dynamic_textFormField_widget.dart`).

Dieses Widget nimmt eine Liste von Controllern und Textfeldern als Argumente an und ordnet diesen eine Zeile in der Eingabe zu.

```
class DynamicTextFields extends StatefulWidget {
  final List<TextEditingController> controllers;
  final List<TextField> fields;
  DynamicTextFields(
    {Key? key, required this.controllers, required this.fields})
    : super(key: key);
  @override
  State<StatefulWidget> createState() => DynamicTextFieldsState();
```

Abbildung 2.2-12 Konstruktor des Screens

Praktische Umsetzung

Mit jedem Klick auf das Plus – Symbol im Screen, wird eine neue Zeile und somit ein neuer Controller erstellt, auf den man dann zugreifen kann. So kann man ihn bspw. auslesen.

Der erste Teil des Widgets fügt mit jedem Klick auf das Plus – Symbol einen neuen Controller und Texteingabefeld hinzu.

```
ListTile(  
    title: Text("Füge einen Begriff hinzu"),  
    trailing: Icon(Icons.add),  
    onTap: () {  
        final TextEditingController controller = TextEditingController();  
        final TextFormField textField = TextFormField(  
            controller: controller,  
            validator: (text) {  
                if (text == null || text.isEmpty) {  
                    return "Eingabe fehlt";  
                }  
                return null;  
            },  
        ),  
  
        setState(() {  
            widget.controllers.add(controller);  
            widget.fields.add(textField);  
        });  
    },
```

Abbildung 2.2-13 Funktionalität des Widgets

So wird das Widget aufgerufen:

```
DynamicTextFormFields(  
    controllers: _controllers1,  
    fields: _fields1,  
) // DynamicTextFormFields
```

Abbildung 2.2-14 Aufruf des Widgets

2.2.3.2 LamacoinInput

Erreichbare Lamacoins

Gib die Anzahl der erreichbaren Lamacoins an

Abbildung 2.2-15 Aussehen LamacoinInput

Das LamacoinInput – Feld vereinfacht die Eingabe der in einer Task zu verdienenden Lamacoins, in dem bspw. verhindert wird, dass andere Zeichen als Ziffern zwischen 1 bis 9 angegeben werden können. Auch die Zahl 0 ist nicht erlaubt.

Praktische Umsetzung

```
class LamacoinInputWidget extends StatefulWidget {  
    final TextEditingController numberController;  
    final FormFieldValidator? validator;  
  
    LamacoinInputWidget(  
        {Key? key, required this.numberController, this.validator})  
        : super(key: key);  
  
    @override  
    State<StatefulWidget> createState() => LamacoinInputWidgetState();  
}
```

Abbildung 2.2-16 Konstruktor

Dieses Eingabefeld erlaubt nur die Eingabe von ganzen Zahlen zwischen 0 und 9.

```
keyboardType: TextInputType.number,  
inputFormatters: <TextInputFormatter>[  
    FilteringTextInputFormatter.digitsOnly  
>, // <TextInputFormatter>[]
```

Abbildung 2.2-17 Es sind nur bestimmte Zeichen erlaubt

Da eine Belohnung von 0 Lamacoins wenig sinnvoll ist, haben wir uns dazu entschieden, auch zu prüfen, ob die eingegebene Zahl kleiner gleich 0 ist. Ist dies der Fall, soll die Eingabe ebenfalls eine Warnung ausgeben.

```
validator: (text) {  
    if (text == null || text.isEmpty) {  
        return "Lamacoins fehlen";  
    }  
    if (int.parse(text) <= 0) {  
        return "Lamacoins müssen größer als 0 sein";  
    }  
    return null;  
},
```

Abbildung 2.2-18 Validator von LamacoinInput

So wird das Widget aufgerufen:

```
LamacoinInputWidget(  
    numberController: _rewardController,  
>, // LamacoinInputWidget
```

Abbildung 2.2-19 Aufruf

Praktische Umsetzung

2.2.3.3 NumberInputWidget

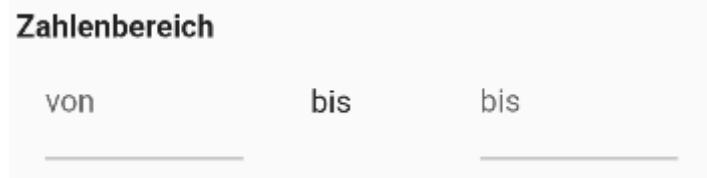


Abbildung 2.2-20 Aussehen NumberInput

Ähnlich wie das Widget LamacoinInput arbeitet das NumberInputWidget. Dies benutzen wir jedes mal, wenn der Nutzer eine Zahl eingeben kann.

Erwähnenswert ist hier, dass man beim Erstellen neuer Screens auch die Kriterien des Validators erweitern kann.

```
class NumberInputWidget extends StatefulWidget {
    final TextEditingController numberController;
    final String? labelText;
    final Function(String)? validator;

    NumberInputWidget(
        {Key? key,
        required this.numberController,
        required this.labelText,
        this.validator})
        : super(key: key);

    @override
    State<StatefulWidget> createState() => NumberInputWidgetState();
}
```

Abbildung 2.2-21 Konstruktor

Hierbei gibt man dem Konstruktor eine Funktion mit, die im Validator ausgeführt wird. Lässt man dieses Argument weg, wird nur auf eine fehlende Eingabe geprüft.

```
validator: (text) {
    if (text == null || text.isEmpty) {
        return "Eingabe fehlt";
    }
    if (widget.validator == null) {
        return null;
    }
    return widget.validator!(text);
},
```

Abbildung 2.2-22 Validator

Praktische Umsetzung

So wird das Widget aufgerufen:

```
NumberInputWidget(  
    numberController: _stepsController,  
    labelText: "Gib die Schritte ein",  
    validator: (text) {  
        if (int.parse(text) <= 0) {  
            return "Zu klein";  
        } else if ((int.parse(_bisController.text) -  
                    int.parse(_vonController.text)) %  
                    int.parse(text) !=  
                    0) {  
            return "Step size muss ein Teiler von (Ende - Start) sein";  
        }  
        return null;  
    },  
) // NumberInputWidget
```

Abbildung 2.2-23 Aufruf

Praktische Umsetzung

2.2.4 Aufbau der Screens zum Erstellen eines Tasksets

2.2.4.1 SetupTasksetBody

Im Menü der Aufgabenverwaltung (siehe 2.3) kommt man durch einen Klick auf das “Plus” – Symbol zum Tab für die Taskseterstellung. Dieser ist in der Datei (lib\app\screens\admin_menu_folder\taskset_manage\widgets\setup_taskset_body.dart) abgelegt.

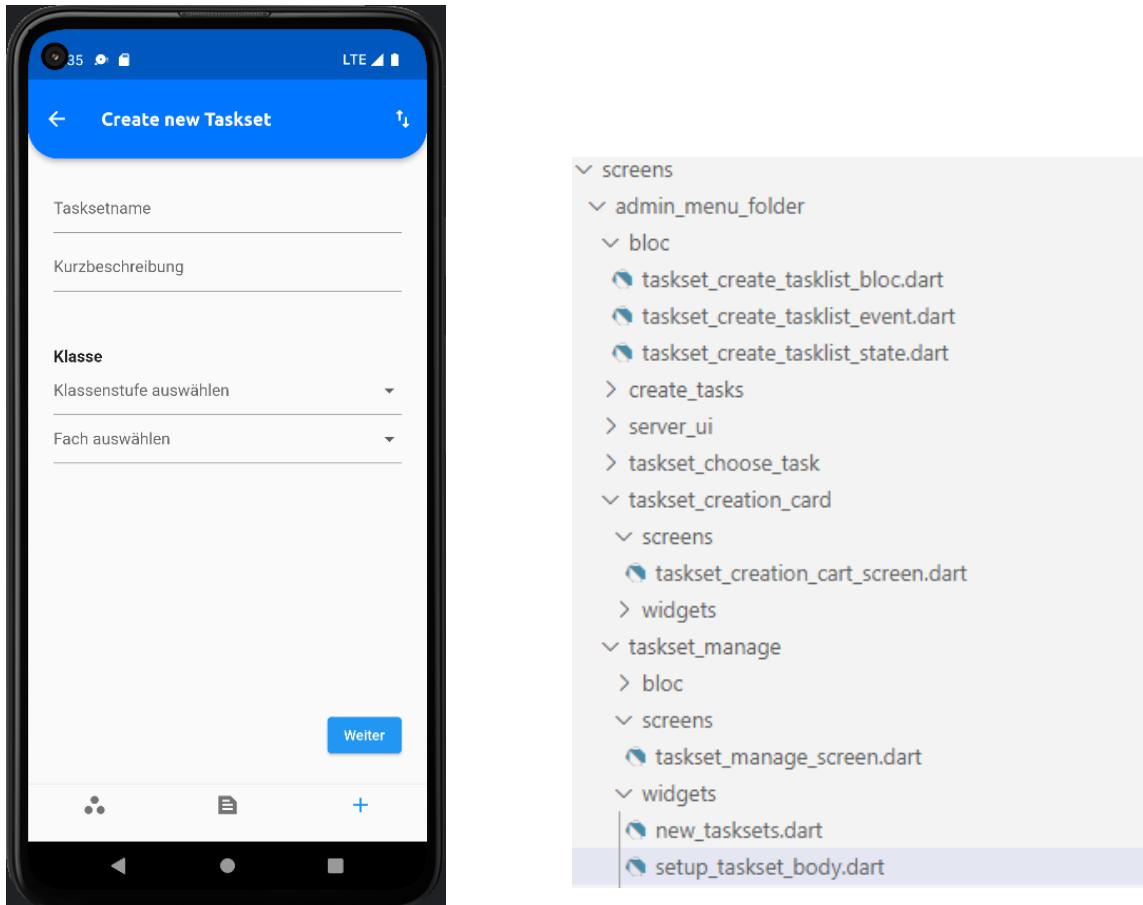


Abbildung 2.2-24 Screen zum Erstellen eines neuen Tasksets

Dieser Screen (bzw. diese Datei) lässt Eingaben zu, welche die Werte der Attribute des in der Bloc – Datei befindlichen Taskset – Objekts ändern. Im Eingabefeld (Tasksetname) gibt man den Namen des Tasksets an, das Feld (Kurzbeschreibung) lässt eine Kurzbeschreibung zu, das Dropdownmenü (Klassenstufe auswählen) erlaubt die Auswahl einer Klassenstufe und (Fach auswählen) lässt den Nutzer ein Fach auswählen.

Jedes Eingabefeld besitzt seine eigene Variable, um die Informationen temporär zu speichern.

```
String? _currentSelectedGrade;
String? _currentSelectedSubject;
 TextEditingController _nameController = TextEditingController();
 TextEditingController _descriptionController = TextEditingController();
```

Abbildung 2.2-25 Controller im Screen

Da der Code der UI jetzt nicht sonderlich spannend ist und in der Datei schnell nachvollzogen werden kann, konzentrieren wir uns darauf, was bei einem Klick auf den Button (Weiter) passiert.

Praktische Umsetzung

```
if (_nameController.text.isNotEmpty &&
    _currentSelectedGrade != null &&
    _currentSelectedSubject != null) {
  // initialize everything else in taskset
  bloc.add(EditTaskset(buildWholeTaskset(blocTaskset)));
  print("Abgeschickte tasklist: " +
    blocTaskset?.tasks! ?? [].toString());
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (_) => MultiBlocProvider(
        providers: [
          BlocProvider.value(value: bloc),
          BlocProvider.value(
            value: BlocProvider.of<TasksetCreateTasklistBloc>(
              context),
            ), // BlocProvider.value
        ],
        child: TasksetCreationCartScreen(isEdit: !first, editedTaskset: blocTaskset),
      ), // MultiBlocProvider
    ), // MaterialPageRoute
  );
}
```

Überprüfen, ob alle Felder ausgefüllt sind

Taskset aus Eingaben bauen

Zum nächsten Screen gehen

Abbildung 2.2-26 (Weiter) Button

Nachdem alle Eingabefelder ausgefüllt worden sind und die Überprüfung erfolgreich war, wird in der Funktion (`buildWholeTaskset()`) eine Instanz der Klasse (Taskset) erzeugt. Das Taskset im Bloc wird dann damit überschrieben.

```
Taskset buildWholeTaskset(Taskset? blocTaskset) {
  if (blocTaskset != null && _currentSelectedSubject != blocTaskset.subject) {
    BlocProvider.of<TasksetCreateTasklistBloc>(context).taskList.clear();
  }
  Taskset taskset = Taskset(
    _nameController.text,
    _currentSelectedSubject,
    _descriptionController.text,
    int.parse(_currentSelectedGrade!),
  );
  taskset.tasks = blocTaskset != null ? blocTaskset.tasks : [];
  return taskset;
}
```

Abbildung 2.2-27 Die Funktion `buildWholeTaskset`

Im Anschluss wird zu dem nächsten Screen gewechselt (TasksetCreationCartScreen).

Praktische Umsetzung

2.2.4.2 TasksetCreationCartScreen

Der nächste wichtige Screen liegt in der Datei (lib\app\screens\admin_menu_folder\taskset_creation_card\screens\taskset_creation_cart_screen.dart). Dieser verwaltet die Liste an hinzugefügten Tasks in unserem Taskset – Objekt und lässt den Nutzer einzelne Tasks hinzufügen, editieren oder löschen und am Ende eine JSON – Datei des Tasksets auf einen Server hochladen.

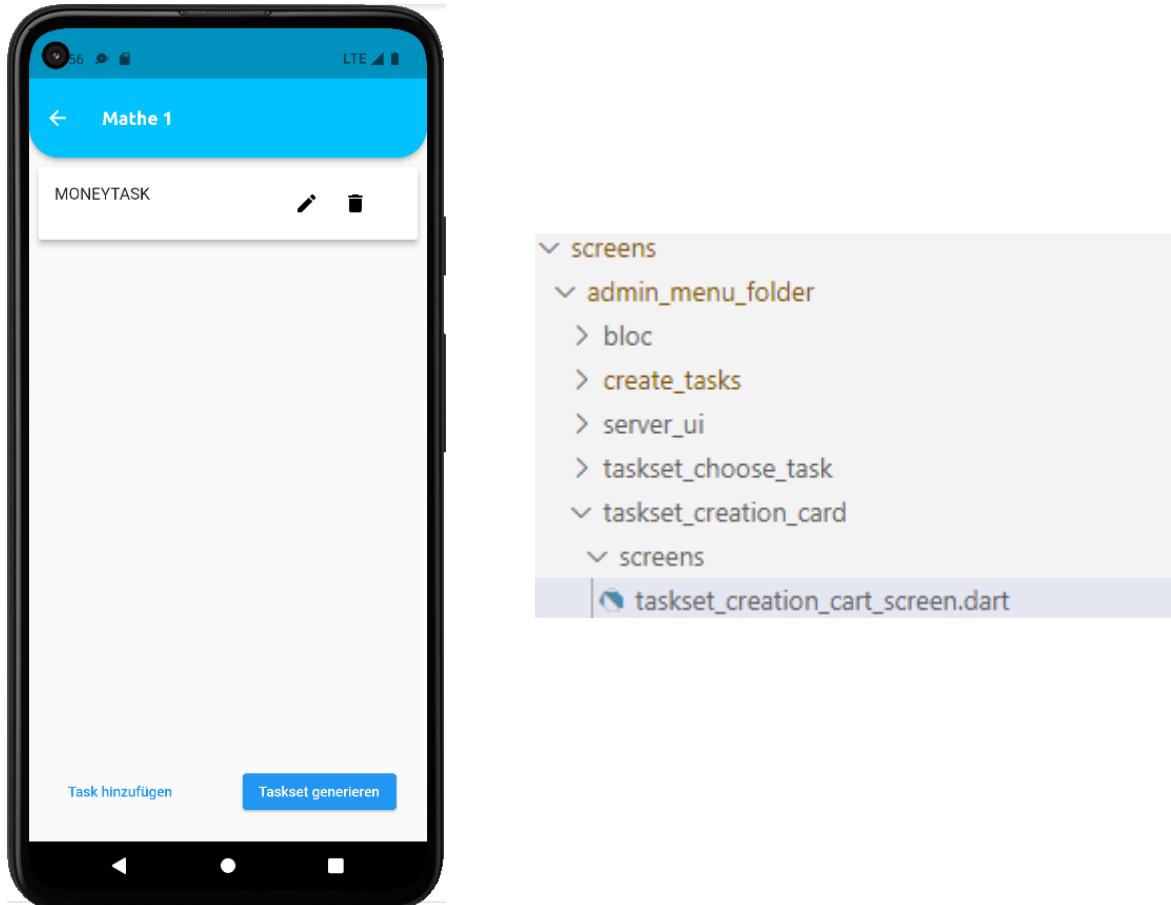


Abbildung 2.2-28 Screen zum Verwalten der Task – Liste

Zur Darstellung der bereits hinzugefügten Tasks benutzen wir das Widget (lib\app\screens\admin_menu_folder\taskset_creation_card\widgets\taskset_creation_cart_widget.dart). Die hinzugefügten Tasks werden als Liste von Cards dargestellt.

Die in der Datei befindliche Funktion (`Widget screenDependingOnTaskType(TaskType taskType)`) wird benutzt, um beim Editieren (Stiftsymbol) die bereits gespeicherten Werte in den jeweiligen Screen zu laden.

Mit einem Klick auf (Taskset generieren) wird eine JSON Datei generiert und auf den Server hochgeladen.

Praktische Umsetzung

2.2.4.3 TasksetChooseTaskScreen

Mit einem Klick auf (Task hinzufügen) öffnet sich die Auswahl an Aufgabentypen.

Zu finden ist die Datei hier (`lib\app\screens\admin_menu_folder\taskset_choose_task\screens\taskset_choose_task_screen.dart`). Das dazugehörige Widget liegt in der Datei (`lib\app\screens\admin_menu_folder\taskset_choose_task\widgets\task_card_widget.dart`).

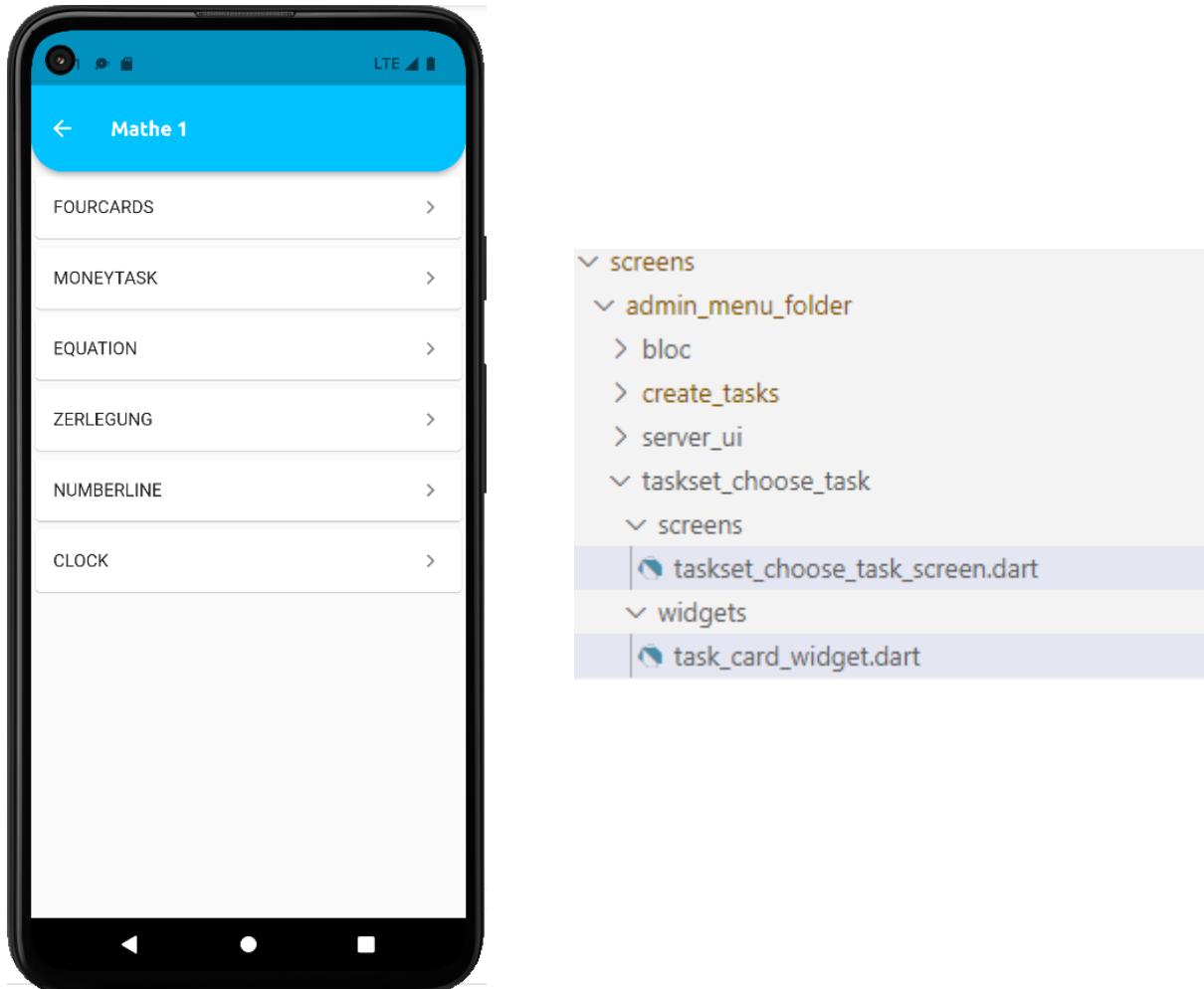


Abbildung 2.2-29 Screen zum Hinzufügen eines neuen Tasks

Hier kann der Nutzer aus einer Auswahl von Aufgabentypen wählen und diese befüllen (siehe 2.2.5).

Die Auswahl ist für jedes Fach beschränkt, da nicht jeder Aufgabentyp für jedes Fach sinnvoll ist (z.B. Vokabeltest für Mathe). So möchten wir die Liste übersichtlicher halten.

Welcher Typ für welches Fach zugelassen ist, kann in der Datei (`lib\app\repository\taskset_repository.dart`) angeschaut und geändert werden.

Praktische Umsetzung

2.2.5 Aufbau der Screens zum Erstellen eines Tasks

Die Screens zum Erstellen eines Tasks sind alle im Ordner (`lib\app\screens\admin_menu_folder\create_tasks`) zu finden.

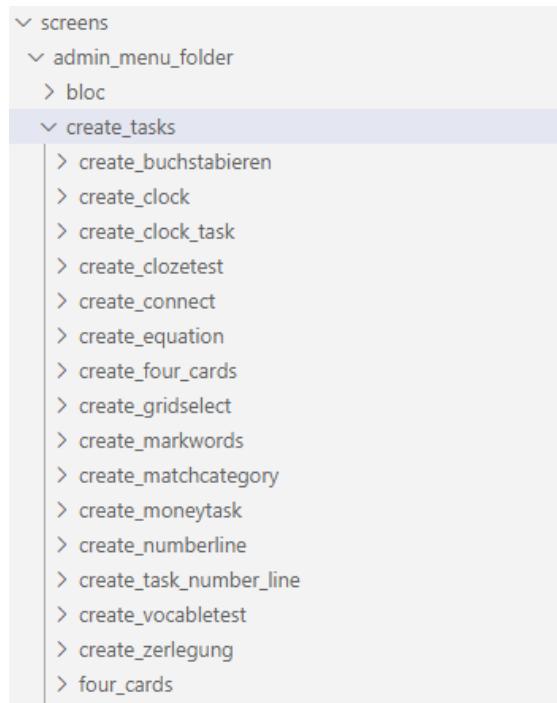


Abbildung 2.2-30 Ordnerstruktur

Wir betrachten beispielhaft den Aufbau des Screens zum Erstellen des Tasks (MatchCategory). Der Code befindet sich in der Datei (`lib\app\screens\admin_menu_folder\create_tasks\create_matchcategory\create_matchcategory_screen.dart`).

Da der Aufbau aller Screens nahezu identisch ist und sie sich häufig nur in der Anzahl und Art der Eingabefelder unterscheiden, lässt sich dieses Beispiel gut auf alle anderen Screens adaptieren.

Praktische Umsetzung

2.2.5.1 Die Klasse CreateXXXScreen

Jeder Screen erweitert StatefulWidget und benötigt einen Konstruktor, der sowohl einen Index als auch den hierfür vorgesehenen Tasktypen übergeben bekommt.

```
class CreateMatchCategoryScreen extends StatefulWidget {  
    final int? index;  
    final TaskMatchCategory? task;  
  
    const CreateMatchCategoryScreen(  
        {Key? key, required this.index, required this.task})  
        : super(key: key);  
    @override  
    CreateMatchCategoryScreenState createState() =>  
        CreateMatchCategoryScreenState();  
}
```

Abbildung 2.2-31 Konstruktor

2.2.5.2 Die Klasse CreateXXXScreenState

Danach muss in der selben Datei eine Klasse für den "ScreenState" erstellt werden. Hier müssen im ersten Schritt die benötigten Variablen deklariert werden.

Jedes Eingabefeld benötigt einen Controller (eine Liste mit Controllern ist ebenfalls möglich).

```
 GlobalKey<FormState> _formKey = GlobalKey<FormState>();  
 TextEditingController _rewardController = TextEditingController();  
 TextEditingController _category1Controller = TextEditingController();  
 TextEditingController _category2Controller = TextEditingController();  
  
 List<TextEditingController> _controllers1 = [];  
 List<TextFormField> _fields1 = [];  
 List<TextEditingController> _controllers2 = [];  
 List<TextFormField> _fields2 = [];  
  
 bool newTask = true;
```

Abbildung 2.2-32 Jeder Screen braucht Controller

Die Variable newTask gibt an, ob es sich um einen neuen Task handelt oder ob ein bereits erstellter Task editiert wird.

Die überschriebene build(BuildContext context) Funktion prüft im ersten Schritt, ob es sich um keine neue Task (d.h. ein Task im Taskset wird editiert) handelt. Ist dies der Fall, werden die Eingabefelder entsprechend des vorhandenen Tasks befüllt. Im anderen Fall wären die Felder zu Beginn leer.

Praktische Umsetzung

```
if (widget.task != null && newTask) {
    _rewardController.text = widget.task!.reward.toString();
    _category1Controller.text = widget.task!.nameCatOne.toString();
    _category2Controller.text = widget.task!.nameCatTwo.toString();

    DynamicTextFormFields.loadListFromTask(
        _controllers1, _fields1, widget.task!.categoryOne);
    DynamicTextFormFields.loadListFromTask(
        _controllers2, _fields2, widget.task!.categoryTwo);

    newTask = false;
}

Taskset blocTaskset = BlocProvider.of<CreateTasksetBloc>(context).taskset!;
Size screenSize = MediaQuery.of(context).size;
```

Abbildung 2.2-33 Überprüfung, ob neuer Task oder editierter

Im nächsten Schritt wird der Aufbau des Screens (dessen Aussehen) programmiert. Hierfür verwenden wir hauptsächlich die in 2.2.3 beschriebenen Custom – Widgets.

Als Standard – Appbar verwenden wir unsere CustomAppbar. Hierbei geben wir den Titel als String (sollte sich auf den Aufgabentyp beziehen) und die Farbe ein. Die Farbe ist abhängig von dem ausgewählten Fach am Anfang, weshalb wir hier mit einer Funktion arbeiten, welche dem Fach entsprechend die Appbar einfärbt.

```
appBar: CustomAppbar(
    size: screenSize.width / 5,
    titel: "Match Category",
    color: LamaColors.findSubjectColor(blocTaskset.subject ?? "normal"),
), // CustomAppbar
```

Abbildung 2.2-34 Angaben Custom – Appbar

Den Body des Screens beschreiben wir nur stichpunktartig. Der komplette Code lässt sich in der entsprechenden Datei nachlesen.

Zu erst wird ein **Standardlayout** benötigt. Dies ist für alle Screens gleich und sorgt u.a. dafür, dass die Screens scrollbar sind. Wie bereits erwähnt, arbeiten wir im **Hauptteil** hauptsächlich mit Custom – Widgets.

Praktische Umsetzung

```
body: Column(  
  children: [  
    Expanded(  
      child: SingleChildScrollView(  
        child: Container(  
          margin: EdgeInsets.all(5),  
          child: Form(  
            key: _formKey,  
            child: Column(  
              children: [  
                HeadLineWidget("1. Kategorie"),  
                TextInputWidget(  
                  textController: _category1Controller,  
                  labelText: "Gib die 1. Kategorie ein",  
                ), // TextInputWidget  
                DynamicTextFormFields(  
                  controllers: _controllers1,  
                  fields: _fields1,  
                ), // DynamicTextFormFields
```

Abbildung 2.2-35 Angaben für den Body

Für die Eingabe der Lamacoins benutzen wir immer das LamacoinInputWidget. Diesem müssen wir auch nur einen Controller übergeben.

```
LamacoinInputWidget(  
  numberController: _rewardController,  
) // LamacoinInputWidget
```

Abbildung 2.2-36 Eingabe für Lamacoins

Den Abschluss eines Screens bildet immer die BottomNavigationbar. Wichtig zu erwähnen ist hier, was passiert, sobald man auf diese klickt. Aussehen und Formatierung sind in der Klasse (CustomBottomNavigationBar) innerhalb der Datei (lib\app\screens\admin_menu_folder\create_tasks\widgets\custom_bottomNavigationBar_widget.dart) festgelegt.

Zu erst wird überprüft, ob alle Felder ausgefüllt sind. So entstehen keine inkonsistenten Objekte. Danach wird ein **Objekt des entsprechenden Task – Objekts** erzeugt, welche als Argumente die Eingaben aus den Eingabefeldern enthält. Dabei musst darauf geachtet werden, welche Argumente seitens des Tasks als Parameter vorgesehen sind (siehe jeweilige Task – Klasse).

Praktische Umsetzung

```
bottomNavigationBar: CustomBottomNavigationBar(
  color: LamaColors.findSubjectColor(blocTaskset.subject ?? "normal"),
  newTask: newTask,
  onPressed: () {
    if (_formKey.currentState!.validate()) {
      TaskMatchCategory taskMatchCategory = TaskMatchCategory(
        widget.task?.id ??
        KeyGenerator.generateRandomUniqueKey(blocTaskset.tasks!),
        TaskType.matchCategory,
        int.parse(_rewardController.text),
        "Schiebe jedes Wort in die Richtige Kategorie",
        _category1Controller.text,
        _category2Controller.text,
        _controllers1.map((e) => e.text).toList(),
        _controllers2.map((e) => e.text).toList(),
        null,
        null,
      ); // TaskMatchCategory
      if (newTask) {
        // add Task
        BlocProvider.of<TasksetCreateTasklistBloc>(context)
          .add(AddToTaskList(taskMatchCategory));
        Navigator.pop(context);
      } else {
        // edit Task
        BlocProvider.of<TasksetCreateTasklistBloc>(context)
          .add(EditTaskInTaskList(widget.index, taskMatchCategory));
      }
      Navigator.pop(context);
    }
  },
); // CustomBottomNavigationBar // Scaffold
```

Erzeugen eines Task – Objekts

Neu zur Liste hinzufügen

Bestehende Task editieren

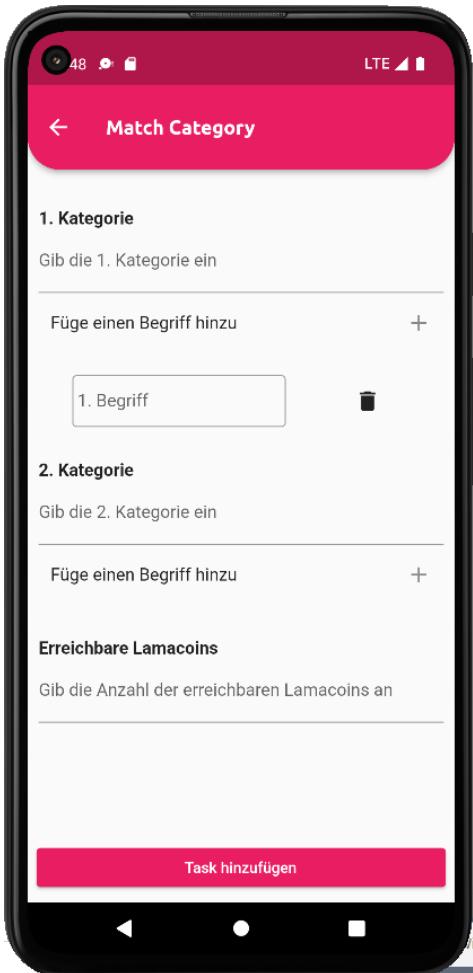
Abbildung 2.2-37 Button (Task hinzufügen)

Im Anschluss wird überprüft, ob es sich um einen **neuen Task** handelt oder um einen **editierten Task**. Abhängig davon, wird eine andere Funktion aufgerufen. Diese Abfrage muss hinter dem Konstruktor erfolgen!

Praktische Umsetzung

2.2.5.3 Aussehen des Screens

So sieht der Screen zum Hinzufügen einer MatchCategory – Aufgabe am Ende aus:



2.2.6 Einen neuen Screen hinzufügen

Wurde zu dem Projekt ein neuer Aufgabentyp hinzugefügt, ist es auch notwendig, einen Screen zum Erstellen dieses Aufgabentyps bereitzustellen. So kann der Nutzer der LAMA – App ganz einfach diese Aufgabe seinem Taskset hinzufügen. Dieses Kapitel zeigt auf, welche Schritte hierzu zu befolgen sind. Wir gehen diese Liste beispielhaft mit dem bereits hinzugefügten Aufgabentyp (MoneyTask) durch. Dies dient der Veranschaulichung.

2.2.6.1 Konzeptionierung

Bei der Konzeptionierung ist es wichtig, festzulegen, welche Parameter der Lehrer angeben kann. Betrachten wir die JSON – Datei für MoneyTask:

```
{
    "id": "6",
    "task_type": "TaskType.moneyTask",
    "lama_text": "",
    "left_to_solve": 3,
    "task_reward": 2,
    "difficulty": 1,
    "optimum": true
},
```

Abbildung 2.2-38 Was ist anpassbar?

Der Lehrer hat hier **Einfluss auf 3 Parameter**, während **einige andere** von der App generiert werden. Was der Lehrer angeben kann, muss für jeden Aufgabentyp individuell festgelegt werden.

2.2.6.2 Implementierung des Aufgabentyps

Der Aufgabentyp muss in die App implementiert werden. In der Datei (lib\app\task-system\task.dart) muss eine Unterklasse erstellt werden, welche unbedingt eine fromJSON – Funktionalität (damit der Schüler darauf zugreifen kann) als auch eine toJSON – Funktion (um eine JSON – Datei zu erstellen) enthält.

```
case "TaskType.moneyTask":
    return TaskMoney(
        json['id'],
        TaskType.moneyTask,
        json['task_reward'],
        json['lama_text'],
        json['left_to_solve'],
        json['difficulty'],
        json['optimum'],
        json['question_language'],
        json['answer_language']
    );
```

Abbildung 2.2-39 fromJSON in task.dart

```
Map<String, dynamic> toJson() => {  
    "id": id,  
    "task_type": type.toString(),  
    "task_reward": reward,  
    "lama_text": lamaText,  
    "left_to_solve": leftToSolve,  
    "difficulty": difficulty,  
    "optimum": optimum,  
};
```

Abbildung 2.2-40 toJSON in task.dart

Die toJSON – Funktion der Unterklasse muss dann noch der Funktion toJSON (eine switch – case Unterscheidung) hinzugefügt werden:

```
case TaskMoney:  
    return (this as TaskMoney).toJson();
```

Abbildung 2.2-41 Aufruf der toJSON Funktion

Damit die JSON – Datei des Aufgabentyps erkannt wird, muss ebenfalls ein Case – Eintrag in der Datei (lib\app\task-system\taskset_validator.dart) innerhalb der Funktion (`static String? _isValidTask(Map<String, dynamic> json)`) gemacht werden:

```
//MoneyTask  
case "TaskType.moneyTask":  
    if (json.containsKey("difficulty") &&  
        json["difficulty"] is int &&  
        json['optimum'] is bool) return null;  
    return "Aufgabentyp: MoneyTask";
```

Abbildung 2.2-42 Task muss vom Validator erkennbar sein

2.2.6.3 Implementierung des Screens zum Erstellen

Als nächstes folgt die Implementierung des Screens, um den Nutzer diesen Aufgabentyp zu seinem Taskset hinzufügen zu lassen. Wir erstellen einen Screen parallel zu der Ausführung in Kapitel 4.2.5 oder können uns auch andere bereits implementierte Screens als Beispiel anschauen. Wichtig ist, dass die Datei in einem Unterordner innerhalb des Ordners (lib\app\screens\admin_menu_folder\create_tasks) angelegt wird.

Für unser Beispiel wäre das: (lib\app\screens\admin_menu_folder\create_tasks\create_moneytask\create_moneytask_screen.dart).

Der Code zur Implementierung von MoneyTask kann in dieser Datei angeschaut werden.

Praktische Umsetzung

Es sind jetzt noch ein paar Einträge zu machen:

In der Datei (lib\app\task-system\task.dart) muss der Aufgabentyp zu dem Enum hinzugefügt werden.

```
enum TaskType {  
    empty,  
    fourCards,  
    clozeTest,  
    zerlegung,  
    clock,  
    clockDifferent,  
    moneyTask,  
    markWords,  
    numberLine,  
    matchCategory,  
    gridSelect,  
    vocableTest,  
    connect,  
    equation,  
    buchstabieren  
}
```

Abbildung 2.2-43 Enum in task.dart

In der Datei (lib\app\repository\taskset_repository.dart) muss der Enum – Bezeichner in der Funktion (`static List<TaskType> giveEnumBySubject(String subject)`) den Fächern zugeordnet werden, für welche der Aufgabentyp angedacht ist.

```
static List<TaskType> giveEnumBySubject(String subject) {  
    switch (subject) {  
        case "Mathe":  
            return [  
                TaskType.fourCards,  
                TaskType.moneyTask,  
                TaskType.equation,  
                TaskType.zerlegung,  
                TaskType.numberLine,  
                TaskType.clock  
            ];  
    };
```

Abbildung 2.2-44 Freigabe für bestimmte Fächer

In der Datei (lib\app\screens\admin_menu_folder\taskset_choose_task\widgets\task_card_widget.dart) folgt ein Eintrag in die Funktion ()

```
Widget screenDependingOnTaskType(TaskType taskType) {  
    switch (taskType) {  
        // TODO bloc task?  
        case TaskType.moneyTask:  
            return MoneyEinstellenScreen(index: null, task: null);  
    };
```

Abbildung 2.2-45 Screen hinzufügen

Hier soll der Screen zum Erstellen des Aufgabentyps zurückgegeben werden.

Wichtig: Hier wird immer null als Argument übergeben!

Praktische Umsetzung

In der Datei (lib\app\screens\admin_menu_folder\taskset_creation_card\widgets\taskset_creation_cart_widget.dart) folgt ein Eintrag in die Funktion ()

```
Widget screenDependingOnTaskType(TaskType taskType) {  
  switch (taskType) {  
    case TaskType.moneyTask:  
      return MoneyEinstellenScreen(index: index, task: task as TaskMoney);  
  }  
}
```

Abbildung 2.2-46 Screen hinzufügen

Wichtig: Hier werden Werte und nicht null übergeben!

Jetzt sind alle Referenzen gemacht und der Screen sollte auswählbar sein.

2.3 Aufgabenverwaltung

2.3.1 Die UI

Die Umsetzung kann in verschiedene Teile unterteilt werden. Den Anfang macht die UI. Hier liegt der Fokus auf der visuellen Darstellung der verschiedenen Listen. Dabei haben wir uns für die Dreiteilung der Komponenten entschieden. Es gibt den, in der Aufgabenerstellung, beschriebenen Teil, welcher vom Hinzufügen eines Tasksets handelt. Den Screen, der alle Tasksets enthält, die in die App importiert wurden oder über den Server erreichbar sind und zu guter Letzt den Screen, auf dem lediglich jene Tasksets sind, die dem User zur Bearbeitung zur Verfügung stehen (siehe Abbildung 2.3-1: All Tasksets Screen).

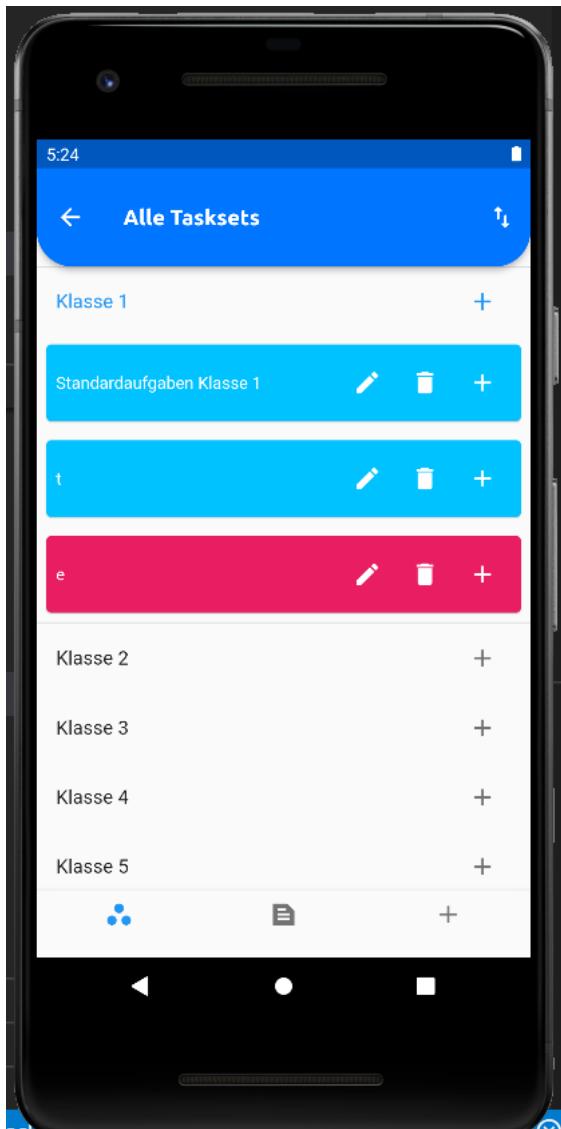


Abbildung 2.3-1: All Tasksets Screen

Nun schauen wir als erstes genauer auf den Screen mit allen Tasksets. Diese Tasksets sind wie folgt in zwei Ebenen strukturiert. Erstens die Schubladen, in denen sich die Klassen befinden und zweitens die sich in den Schubladen befindlichen Tasksets. Diese werden mit Hilfe eines Farbschemas voneinander unterschieden, welche auf den Fächern der Tasksets basieren. Dabei entspricht blau Mathe, rot Deutsch, orange Englisch, usw. (Abbildung 2.3-2: Alle Tasksets (standard in Pool))

Praktische Umsetzung

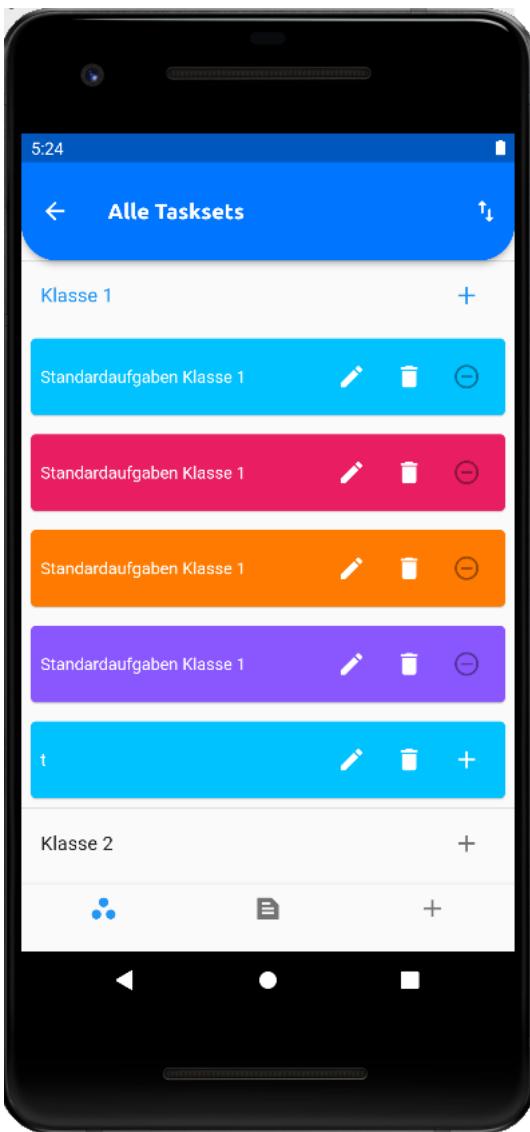


Abbildung 2.3-2: Alle Tasksets (standard in Pool)

Mit dem Plus Button kann ein einzelnes, beim Druck auf das Plus des Tasksets, oder eine Liste, beim Druck auf das Plus der Klasse, in den Pool geschoben werden.

Andersherum, falls ein Taskset bereits im Pool ist, wird das Plus zu einem Minus(in abb xxx zu sehen). Nun kann bei erneutem Druck auf den Button das Taskset wieder aus dem Pool entfernt werden. Dies ist auch direkt im Screen des Pools möglich, genau so wie das Editieren oder Löschen eines Tasksets. Sind alle Tasksets einer Klasse im Pool, so wird wiederum dieses Minus zu einem Plus und die ganze Liste kann potentiell aus dem Pool entfernt werden. Ähnliches Spiel im Screen, welcher nur die im Pool befindlichen Tasksets anzeigt, hier wird jedoch beim Druck auf das Minus das Taskset direkt aus dieser Liste und somit auch dem Screen entfernt(siehe Abbildung 2.3-3: Taskset Pool).

Das Importieren eines Tasksets ist über den Button in der rechten oberen Ecke zu erreichen.

Praktische Umsetzung

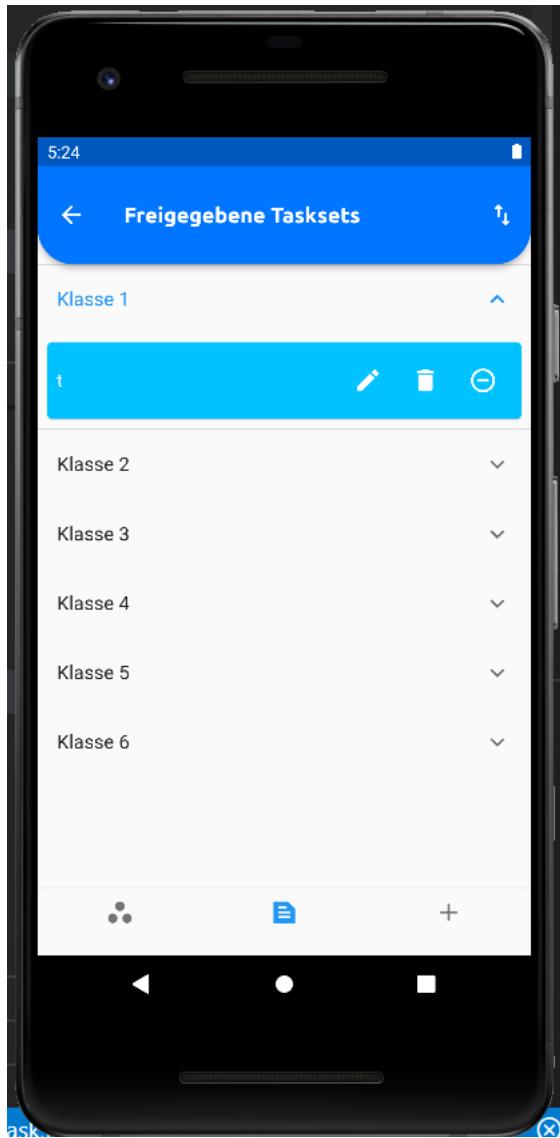


Abbildung 2.3-3: Taskset Pool

2.3.2 Die Implementierung des Taskset managements

Die ursprünglich einzelne Tasksetliste existiert nun zweimal, sowohl in Form der Map im Repository als auch im Bloc der die Tasksets managed. In Abbildung 2.3-4: Taskset Manage Bloc ist der Bloc zu sehen, in dem die Liste für den Pool und die Liste aller Tasksets verwaltet werden.

```
class TasksetManageBloc extends Bloc<TasksetManageEvent, TasksetManageState> {
    List<Taskset> tasksetPool;
    List<Taskset> allTaskset;

    TasksetManageBloc({required this.tasksetPool, required this.allTaskset})
        : super(TasksetManageInitial()) {
```

Abbildung 2.3-4: Taskset Manage Bloc

Diese werden von den Screens verwendet und beziehen ihre Daten zu Beginn aus dem Repository. Die nun folgenden Events und ihre Implementierung korrespondieren zu den oben erläuterten Funktionalitäten.

Praktische Umsetzung

```
on<AddTasksetPool>((event, emit) {
    tasksetPool.add(event.taskset);
    event.taskset.isInPool = true;
    updateTaskset(event.context, event.taskset);
    emit(ChangeTasksetStatus());
});
on<RemoveTasksetPool>((event, emit) async {
    tasksetPool.remove(event.taskset);
    event.taskset.isInPool = false;
    updateTaskset(event.context, event.taskset);
    emit(ChangeTasksetStatus());
});
```

Abbildung 2.3-5: Taskset Pool bearbeiten

Beginnen wir mit dem Hinzufügen eines Tasksets zum Pool und dem Entfernen eines Tasksets aus dem Pool. Es wird zunächst das Taskset dem Pool hinzugefügt, wobei der Status, ob es sich in diesem befindet, auf true gesetzt wird. Auf der anderen Seite entfernt das Remove das Element aus dem Pool und setzt den Status auf false. Die Updatemethode wird bei beiden aufgerufen und passt das Taskset auf dem Server an. Zu guter Letzt wird der State an die UI zurückgegeben, wodurch diese neu geladen wird und die Button aktualisieren (siehe Abbildung 2.3-5: Taskset Pool bearbeiten).

```
on<AddListOfTasksetsPool>((event, emit) {
    event.tasksetList.forEach((element) {
        if (!element.isInPool && element.taskurl!.id == null) {
            addATaskset(event.context, element);
        }
    });
    emit(ChangeTasksetStatus());
});
on<RemoveListOfTasksetsPool>((event, emit) {
    event.tasksetList.forEach((element) {
        if (element.taskurl!.id == null) {
            removeATaskset(event.context, element);
        }
    });
    emit(ChangeTasksetStatus());
});
```

Abbildung 2.3-6: Taskset Pool bearbeiten mit einer Liste

Das Hinzufügen und Entfernen der Listen verhält sich sehr ähnlich zu dem zuvor erläuterten Verwalten eines einzelnen Tasksets. Hierbei wird jedoch nicht mehr nur ein Taskset vom Event übergeben, sondern eine Liste durch diese wird in beiden Fällen durchgegangen. Wenn nun noch nicht alle Tasksets der Liste im Pool sind wird das AddListOfTasksetPool Event aufgerufen. Für jedes Element wird geprüft, ob es in der Liste ist, dann soll es natürlich nicht hinzugefügt werden und ob die id der Url null ist. Dann handelt es sich nämlich um ein Taskset vom Server und kann bearbeitet werden. Die addTaskset methode ist von der Logik gleich dem Adden in Abbildung 2.3-5: Taskset Pool bearbeiten.

Das RemoveListOfTasksetPool Event hat die selben Parameter. Es wird für jedes Element geprüft, ob es sich um ein Taskset vom Server handelt, ist das der Fall so wird wieder die selbe Funktionalität aus

Praktische Umsetzung

Abbildung 2.3-5: Taskset Pool bearbeiten vom remove Event aufgerufen. Bei beiden wird wieder ein State zurückgegeben, um die UI neu zu laden(siehe Abbildung 2.3-6: Taskset Pool bearbeiten mit einer Liste).

```
on<DeleteTaskset>((event, emit) async {
    final repo = RepositoryProvider.of<TasksetRepository>(event.context);
    emit(WaitingTasksetStatus());
    await repo.deleteTasksetFromServer(event.context, event.taskset);
    repo
        .tasksetLoader
        .loadedTasksets[
            | | SubjectGradeRelation(event.taskset.subject, event.taskset.grade)]!
        .remove(event.taskset);

    allTaskset.remove(event.taskset);
    if (event.taskset.isInPool) {
        tasksetPool.remove(event.taskset);
    }
    emit(ChangeTasksetStatus());
});
```

Abbildung 2.3-7: Delete Taskset Event

Mit dem DeleteTaskset Event kommen wir zum Löschen eines Tasksets. Dies kann einen Moment dauern, daher wird erstmal eine Waiting screen erzeugt, wenn es nötig wird. Dabei erscheint ein CircularProgressIndikator in der Mitte des Screens. Darauf folgt das Löschen des Taskset vom Server. Um die Implementierung dessen soll es aber zu einem späteren Zeitpunkt gehen. Der Fokus liegt also auf dem lokalen Löschen des Tasksets, sowohl im Repository als auch im Bloc. Der Repositoryeintrag wird in einem ersten Schritt entfernt. Dies hat nicht so eine große Bedeutung wie das Entfernen des Tasksets aus den Bloc Listen, da nur so die Änderungen in der UI dargestellt werden können (siehe Abbildung 2.3-7: Delete Taskset).

```
on<UploadTaskset>((event, emit) async {
    final repo = RepositoryProvider.of<TasksetRepository>(event.context);
    await repo.fileUpload(event.context, event.taskset).toString();
    if (repo.tasksetLoader.loadedTasksets[SubjectGradeRelation(
        | | event.taskset.subject, event.taskset.grade)] ==
        null) {
        repo.tasksetLoader.loadedTasksets[SubjectGradeRelation(
            | | event.taskset.subject, event.taskset.grade)] = [];
    }
    repo
        .tasksetLoader
        .loadedTasksets[
            | | SubjectGradeRelation(event.taskset.subject, event.taskset.grade)]!
        .add(event.taskset);
    allTaskset.add(event.taskset);
});
```

Abbildung 2.3-8: Upload Taskset Event

Praktische Umsetzung

Das letzte Event im Manage Bloc ist das UploadTaskset Event, welches aufgerufen wird wenn auf Generieren gedrückt wird beim erzeugen eines Tasksets. Dabei wird das Taskset als File geuploaded. Auch hier soll es in einem späteren Teil mit dem Löschen, um die Implementierung gehen. Wenn diese SubjectGradeRelation noch nicht vorhanden ist, wird eine leere Liste angelegt und das neue Element hinzugefügt. Existiert die Relation schon, wird lediglich das Element hinzugefügt. Zu guter letzt wird die Tasksetliste um das Element erweitert. Dabei ist zu beachten, dass im Moment ein neu angelegtes Taskset nicht zugänglich ist für die Schüler, dies muss über den Manage screen manuell hinzugefügt werden(siehe Abbildung 2.3-8: Upload Taskset Event).

2.3.3 Der Server Screen

Im folgenden soll es darum gehen, wie der Server über die App genutzt werden kann und die Implementierung der auf den Server zugreifenden Methoden. Hierbei wird es wieder zu einer Aufteilung von UI und Implementierung kommen.

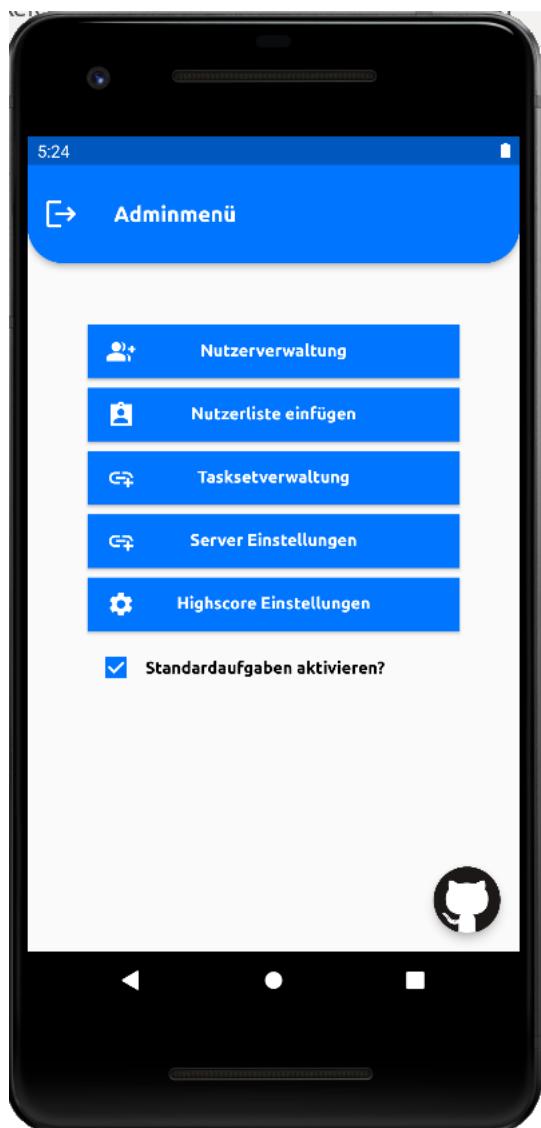


Abbildung 2.3-9: Menu Screen

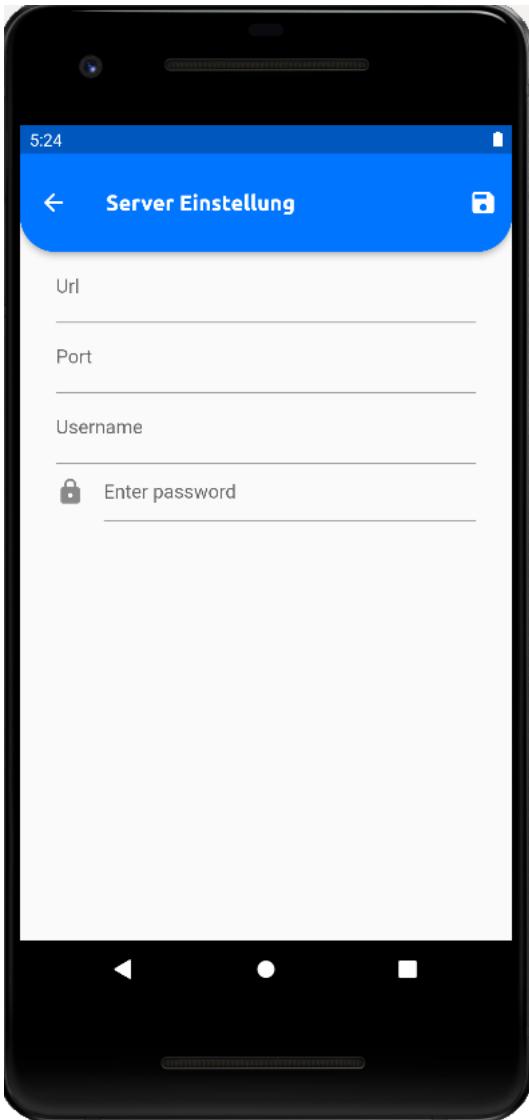


Abbildung 2.3-10: Server Einstellung Screen

In Abbildung 2.3-9: Menu Screen ist der Menuscreen zu sehen unter dem neuen Eintrag Server Einstellung kommt man in Abbildung 2.3-10: Server Einstellung Screen. Hier kann man die Einstellung seines Servers in die App übertragen, diese werden dann verwendet um die files herunterzuladen, Tasksets hochzuladen oder Tasksets zu löschen.

Man hat im Einstellungsscreen vier Felder, die alle auszufüllen sind. Beim Druck auf den Button in der oberen rechten Ecke werden die Daten in einer Lokalen Datenbank gespeichert oder aktualisiert. Beim zurück gehen wird nichts gespeichert und der alte Stand bleibt erhalten.

2.3.4 Die Datenbank

Ein Tabelleneintrag besteht aus den obigen vier Feldern und einer id. Die Methode `getServerSettings` ist trivial und gibt einfach anhand des Tabelleneintrags die Settings zurück. Diese können im `ServerRepository` hinterlegt und verwendet werden.

Praktische Umsetzung

```
Future<ServerSettings> insertServerSettings(
    ServerSettings serverSettings) async {
    final db = await (database);
    final size = await dbSize;
    print("size" + size.toString());
    if (size == null || size == 0) {
        print("insert");
        serverSettings.id =
            (await db?.insert(tableServer, serverSettings.toJson()))!;
    } else {
        print("update");
        db?.update(tableServer, serverSettings.toJson(),
            where: '${ServerFields.columnId} = ?',
            whereArgs: [serverSettings.id]);
    }
    return serverSettings;
}
```

Abbildung 2.3-11: Insert Server Settings Methode

Spannender ist die insertServerSettings Methode in Abbildung 2.3-11 hier gibt es zwei Fälle, die eintreten können. Zum einen gibt es den initialen Stand ohne Tabelleneintrag, also muss ein neuer Eintrag geschrieben werden, basierend auf den übergebenen Serversettings. Dies ist der Fall, wenn die Datenbank noch nicht existiert oder leer ist. Der andere Fall ist, es existiert bereits ein Tabelleneintrag, dann muss dieser geupdated werden. Dabei werden die alten Werte an einer bestimmten id durch die neuen ersetzt (siehe Abbildung 2.3-11: Insert Server Settings Methode).

2.3.5 Das Server Repository(neu)

Im Server Repository werden, sowohl beim Initialisieren als auch in der setOrUpdate Methode, die Serversettings Variable auf den zurückgegebenen Wert gesetzt (siehe Abbildung 2.3-12: Server Settings Repository).

```
ServerSettings? serverSettings;

/// loads db values in url, username, password
void initialize() async {
    serverSettings = await DatabaseProvider.db.getServerSettings();
}

Future<void> setOrUpdate(ServerSettings serverS) async {
    serverSettings = await DatabaseProvider.db.insertServerSettings(serverS);
}
```

Abbildung 2.3-12: Server Settings Repository

2.3.6 Das Taskset Repository(erweitert)

Neben der Verwaltung des Servers soll es nun um die Bearbeitung der Dateien auf dem Server gehen. Dabei wird einmal in die Implementierung des Löschens, des Hochladens eines Tasksets und des Downloadens aller auf dem Server befindlichen Files geschaut.

Zu Beginn jeder dieser Methoden wird ein SSH Client erzeugt. Dieser hat einen SSH Socket, welcher aus der URL und dem Port besteht. Zur Sicherheit erhält der Client noch einen Username und ein

Praktische Umsetzung

Passwort. Die Konkrete Implementierung ist bei allen Methoden gleich und sieht wie folgt aus (siehe Abbildung 2.3-13: SSH Client).

```
final client = SSHClient(  
    await SSHSocket.connect(  
        serverRepo.serverSettings!.url,  
        serverRepo.serverSettings!.port,  
    ),  
    username: serverRepo.serverSettings!.userName,  
    onPasswordRequest: () => serverRepo.serverSettings!.password,  
);
```

Abbildung 2.3-13: SSH Client

Zum Anfang der Löschfunktion wird abgefragt, ob es sich um ein auf dem Server befindliches Taskset handelt, denn nur diese können verändert werden. Des Weiteren kann nichts gelöscht, hochgeladen oder heruntergeladen werden, wenn die Serversettings nicht gesetzt sind. Dies wird durch das if in Abbildung 2.3-14: Server Settings Check abgefangen welches zu Beginn der der Funktionen überprüft wird.

```
if (serverRepo.serverSettings != null && taskset.taskurl!.id == null) {
```

Abbildung 2.3-14: Server Settings Check

Abbildung 2.3-15: Delete Funktion zeigt das Herz der Löschfunktion, das File in der Removefunktion entfernt wurde.

```
final sftp = await client.sftp();  
await sftp.remove('./upload/${taskset.grade}/${taskset.name}');  
client.close();  
client.done;
```

Abbildung 2.3-15: Delete Funktion

In Abbildung 2.3-16: Upload Funktion wenden wir uns nun der etwas umfangreicherem Uploadfunktion zu. Die Aufgabe ist offensichtlich das Laden eines Tasksets auf einen Server. Dies wird erreicht, indem man ein File im schreibenden Zustand öffnet und falls es nicht existiert erzeugt. Daraufhin wird in dieses File, das aus dem Taskset generierte JSON geschrieben und geschlossen.

```
final file = await sftp.open(  
    './upload/${taskset.grade}/${taskset.name!}',  
    mode: SftpFileOpenMode.create | SftpFileOpenMode.write,  
);  
String tmp = json.encode(taskset.toJson());  
var bytes = Utf8Encoder().convert(tmp);  
await file.write(Stream.value(bytes));  
file.close();  
client.close();  
client.done;
```

Abbildung 2.3-16: Upload Funktion

Praktische Umsetzung

In der letzten Funktion soll es um das Herunterladen der Inhalte des Servers gehen. Dabei wird zunächst eine Liste an Strings gebaut, die den Pfaden der Files entspricht (siehe Abbildung 2.3-17: Pfad der Files).

```
final sftp = await client.sftp();
final list = await sftp.listdir('./upload');

for (var folderName in list) {
    if (klassenStufe.contains(folderName.filename)) {
        final listOfFileNames =
            await sftp.listdir('./upload/${folderName.filename}');
        final l = listOfFileNames.map((e) => e.filename).toList();
        l.removeWhere((element) => element == '.' || element == '..');
        allFileNames.addAll({'${folderName.filename}': l});
    }
}
var listTmp = [];
allFileNames.forEach((key, value) {
    value.forEach((element) {
        listTmp.add('./upload/$key/$element');
    });
});
}
```

Abbildung 2.3-17: Pfad der Files

Im folgenden Schritt werden die Files mit Hilfe der Pfade geöffnet und gelesen. Die gelesenen Inhalte werden zu einem Tasksetobjekt geformt und einer Liste hinzugefügt, die von der Methode zurückgegeben wird. Die Methode wird beim Initialisieren der App aufgerufen und der Rückgabewert der existierenden Liste hinzugefügt (siehe Abbildung 2.3-18: Download).

```
for (var fullname in listTmp) {
    final tmp = await sftp.open(fullname);
    final content = await tmp.readBytes();
    print(utf8.decode(content));
    tasksetList
        .add(Taskset.fromJson(json.decode(Utf8Decoder().convert(content))));
    //print(latin1.decode(content));
    tmp.close();
}
```

Abbildung 2.3-18: Download Files

2.4 Text To Speech

Das TTS Feature soll Fragen, Aufgabestellungen und Antwortmöglichkeiten, dem Nutzer vorlesen. Das heißt es gibt keine eigenen Screens dazu, allerdings werden alle Taskscreens in `lama_app/lib/tapTheLama/screens/task_screens` erweitert. Um Text to Speech umzusetzen, haben wir uns für das Text to Speech Framework ver. 3.5 von Tundralabs.com entschieden. Das Framework bietet die Möglichkeit Wörter, Zahlen und ganze Sätze in mehrere Sprachen und Dialekten vorzulesen. In der Lama App sind zurzeit [Stand 03.08.2022] nur deutsche und englische Tasks implementiert, weswegen unsere bisherige Entwicklung nur zwei Sprachen unterstützt, das kann man aber sehr einfach ändern und um mehr Sprachen erweitern. Das Framework ist lauffähig auf iOS, Android, Web und MacOS. Derzeit [Stand 03.08.2022] ist die Umsetzung für iOS und Android Plattformen umgesetzt und getestet. Mindestversion des Android Plattform für TTS Framework ist SDK 21, für iOS – 10.

Um das Framework ins Projekt hinzufügen, müsste man einige Schritte durchführen:

- Den Command `$ flutter pub add flutter_tts` ausführen
- In `pubspec.yaml` Datei eine Abhängigkeit hinzufügen: `flutter_tts: ^3.5.0`
- Entsprechendes Widget mit der Zeile `import 'package: flutter_tts/ flutter_tts.dart';` ergänzen.

Um Das Framework allein aktualisieren:

- `dart pub upgrade flutter_tts`

Die Dokumentation zu dem TTS Framework lässt sich [Stand 03.08.2022] unter folgendem Link finden:
https://pub.dev/packages/flutter_tts

2.4.1 Text To Speech Framework mit bloc Architektur

In der Lama App wird auch das bloc Framework benutzt, damit es möglich ist, das Backend und das Frontend voneinander zu trennen. Das Konzept ist in „Abbildung 2.4-1 : prinzipielle Schema für Interaktion“ zu sehen: auf der UI Seite werden die Events ausgelöst beispielsweise beim Anklicken eines Knopfes, die dann an bloc Datei weitergesendet werden. Die bloc Datei empfängt die Events, führt seine Arbeit aus und liefert dann die States an das Widget zurück.

Genau nach diesem Schema wird das TTS Framework, das ist in TTS Bloc Datei eingebaut, mit den Widgets interagieren.

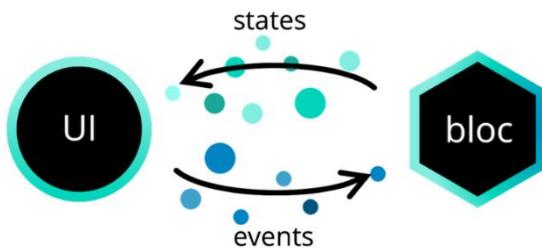


Abbildung 2.4-1 : prinzipielle Schema für Interaktion

Die ganze Logik für TTS Framework ist in die `tts_bloc.dart` Datei ausgelagert. Die Datei befindet sich im Ordner `\lama_app\lib\app\bloc\taskBloc`.

Im Verzeichnis `\lama_app\lib\app\screens\task_type_screens` sind die Task Widgets enthalten, die erforderliche Logik aus `tts_bloc.dart` abrufen können. Jetzt können die Task Widgets die Events an TTS Bloc absenden und von der bloc Datei, die States entsprechend empfangen. In „Abbildung 2.4-2: Das Schema für TTS Bloc Interaktion“ wird schematisch dargestellt, wie dieses Austausch geschieht. Jeder

Praktische Umsetzung

Task Screen, bei dem eine bestimmte Nutzerinteraktion auftritt, beispielsweise ein Anklicken eines spezifischen Knopfes, sendet ein Event an TTS Bloc Datei und empfängt einen veränderteren State. Für jeden Task-Screen muss ein Anfangs-State geben, der später durch andere Events verändert werden kann.

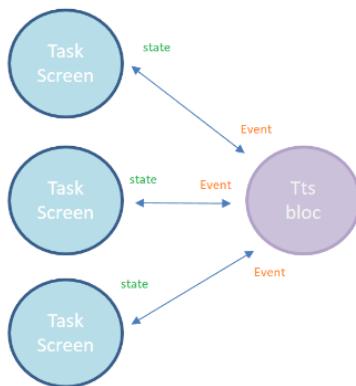


Abbildung 2.4-2: Das Schema für TTS Bloc Interaktion

TTSBloc Klasse erweitert die Bloc Klasse mit dazugehörigen TTS Events und TTS States. Dann werden die Handler nach Anleitung durch on<Event> API registriert. Nun werden Events, die in einem Screen erzeugt wurden, empfangen und danach in States konvertiert. Ein State ändert sich in bloc erst, wenn neue Events hinzugefügt werden, also ein on<Event> getriggert wird.

In der tts_bloc.dart Datei werden zwei Felder deklariert: „text“ und „lang“. In diesen wird der vorzulesende Text hinterlegt und die Sprache, in der dieser vorgelesen werden soll. Außerdem gibt es die Methode „readText“(siehe Abbildung 2.4-3: TTS Bloc DateiAbbildung 2.4-3: TTS Bloc Datei), diese wird aufgerufen, um den Text tatsächlich vorzulesen, auf die genauere Funktion der Methode wird später weiter eingegangen.

Die Datei enthält die Logik

```

import ...
// Authors: J.Wißbach and A.Brisbin
/*
Bloc Klasse mit Methode, um Texte, die übergeben werden, vorzulesen.
Und Events, um diese Texte vorzulesen.
*/
class TTSBloc extends Bloc<TTSEvent,TTSSState> {
    final FlutterTts flutterTts = FlutterTts(); ← TTS instanziieren
    String text = "";
    String lang = ""; ← Eingabefelder für Events

    readText(String text, String lang) async {
        if(!home_screen_state.isTTS()) { ← TTS ein oder aus
            return;
        }
        if (lang == "Englisch") { ← Spracheauswahl
            await flutterTts.setLanguage("en-US");
            await flutterTts.setSpeechRate(0.4);
            await flutterTts.setPitch(1.1);
        } else {
            await flutterTts.setLanguage("de-De");
        }
        await flutterTts.setVolume(1.0); ← Parameters für angenehmes anhören
        await flutterTts.speak(text);
    }
}
  
```

Annotations im Bild:

- Die Datei enthält die Logik
- TTS instanziieren
- Eingabefelder für Events
- TTS ein oder aus
- Spracheauswahl
- Parameters für angenehmes anhören

Abbildung 2.4-3: TTS Bloc Datei

Praktische Umsetzung

Nach dem Schema wie in der Bloc-Anleitung, werden die Events durch einen Event Stream in der Bloc Datei empfangen und zu den entsprechenden Events werden die neuen States zurückgegeben. TTS Bloc kann eine Reihe von Events empfangen und abarbeiten:

- QuestionOnInitEvent: Event wird empfangen, wenn ein Aufgaben Widget neu geladen wird, damit die Aufgabestellung automatisch beim Öffnen des Widgets vorgelesen wird.
- ClickOnQuestionEvent: Event wird beim Anklicken einer Frage oder Aufgabenstellung im Task Widget ausgelöst.
- ClickOnAnswerEvent: Beim Anklicken einer Antwortmöglichkeit.
- SetDefaultEvent: Dient dazu, dass der State auf den default(Start) state zurückgesetzt wird. Damit beim Laden eines neuen Widgets wieder alle Events erzeugt werden können.
- ReadQuestionEvent: Wird beim Anklicken der Frage im Task Widget ausgelöst, allerdings wird kein neuer State zurückgeliefert.

```
TTSBloc() : super(EmptyTTSState()){  
    on<QuestionOnInitEvent>((event, emit) async { ←  
        readText(event.question, event.questionLanguage);  
        emit(IsNotEmptyState());  
    });  
    on<ClickOnQuestionEvent>((event, emit) async { ←  
        readText(event.textToPlay, event.answerLanguage);  
        emit(VoiceTtsState());  
    });  
    on<ClickOnAnswerEvent>((event, emit) async { ←  
        readText(event.answer, event.answerLanguage);  
        emit(VoiceAnswerTtsState(event.answer));  
    });  
    on<SetDefaultEvent>((event, emit) async { ←  
        emit(EmptyTTSState());  
    });  
    on<IsNotEmptyEvent>((event, emit) async { ←  
        emit(IsNotEmptyState());  
    });  
    on<ReadQuestionEvent>((event, emit) async { ←  
        readText(event.question, event.questionLanguage);  
    });  
}
```

Vorlesen beim Aufmachen des Aufgabe Screens

Vorlesen beim Klick auf die Frage oder Aufgabestellung

Vorlesen beim Klick auf die Antwort

Task Screen wird ins default Zustand gesetzt

Task Screen wird ins not Empty Zustand gesetzt

Die Frage wird vorgelesen ohne Zustand zu ändern

Abbildung 2.4-4: Event Stream in der Bloc Datei

In Verzeichnis lama_app\lib\app\event wird in der Datei tts_event.dart alle mögliche Events deklariert.

Die abstrakte Klasse TTS Event ist nach der bloc Anleitung definiert. So wird eine Menge von vererbbaaren Events definiert, die alle auf TTS Event basieren. Dies ist notwendig, damit man später in den Task Widgets über den BlocBuilder Widget ein beliebiges TTS Event an TTS Bloc senden kann.

Alle Events außer SetDefaultEvent, erhalten ein vorzulesender Text und die Sprache, in welcher der Text vorgelesen werden soll.

In Verzeichnis lama_app\lib\app\state ist die Datei tts_state.dart enthalten, wo alle mögliche States für UI definiert sind(siehe Abbildung 2.4-5: erlaubte States für TTS Bloc Datei).

Praktische Umsetzung

```
abstract class TTSState {}

class EmptyTTSState extends TTSState {}

class VoiceTtsState extends TTSState {}

class VoiceAnswerTtsState extends TTSState {
  String selectedAnswer;
  VoiceAnswerTtsState(this.selectedAnswer);
}

class IsNotEmptyState extends TTSState {
```

Abbildung 2.4-5: erlaubte States für TTS Bloc Datei

Genauso wie bei Events wird zuerst die abstrakte Klasse TTSSState deklariert. So werden eine Reihe an States erzeugt, die zu TTS Bloc gehören, die für die BlocProvider Widgets benötigt werden:

- EmptyTTSState: Der dient dazu, ein default State für Screens zu deklarieren, wenn bisher noch kein Event erzeugt und abgeschickt wurde.
- IsNotEmptyState: Dieser wird gesetzt, falls die Aufgabenstellung beim Screen Aufmachen schon vorgelesen wurde.
- VoiceTtsState: Dieser State wird an ein Widget zurückgeliefert, wenn ClickOnQuestionEvent ausgelöst wurde.
- VoiceAnswerTtsState: Dieser wird gesetzt, falls man Antworten im Aufgaben Screen anklickt.

2.4.2 Task Screens mit TTS Bloc

Es sind genau 14 Screens, die mit TTS Bloc versorgt sind. Alle Screens liegen in `lama_app\lib\app\screens\task_type_screens` Verzeichnis. Als Beispiel wird „Four Card“ Task Screen hier genommen. Wenn der Nutzer einen Task öffnet, wird dem Nutzer sofort die Frage oder falls es keine gibt, die Aufgabenstellung vorgelesen (siehe Abbildung 2.4-6: automatisch vorlesen beim Aufmachen).

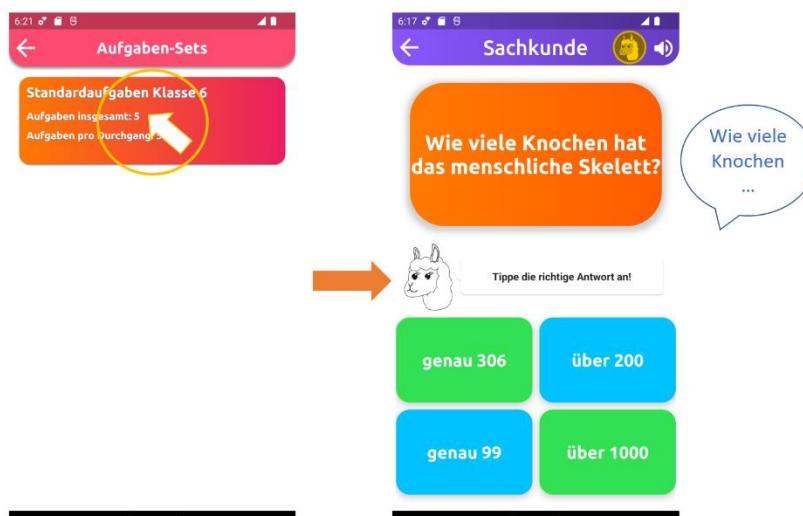


Abbildung 2.4-6: automatisch vorlesen beim Aufmachen

Praktische Umsetzung

Wenn ein Task jedoch aus mehreren Screens besteht, wie beispielsweise die „Buchstabieren“ Task, wird die Aufgabenstellung nur beim Öffnen des ersten Screens vorgelesen, da ansonsten das häufige Vorlesen des immer gleichen Textes sehr überflüssig ist und als sehr nervig empfunden werden kann.

Wie das jetzt genau umgesetzt wird, betrachten wir ebenfalls am „Four Card“ Task Screen. Zuallererst wird TTS Bloc in den Widget Baum mithilfe des BlocProvider Widget übergeben, dies nennt man Dependency Injektion, damit es für BlocBuilder klar ist, welcher Bloc geliefert wird und mit welchem man bei späteren Aufrufen interagieren muss. BlocBuilder ist ein Flutter Widget, das dafür verantwortlich ist, ein Widget nach entsprechend empfangenes State umzubauen und anzupassen. Bevor TTS Bloc eine Instanz des Kontexts erhält, durchsucht der BlocBuilder den Widget-Baum nach Bloc Elementen. In diesem Fall wird beim Aufbauen des Screens ein BlocBuilder „empty State“ geschaffen wird. Weswegen er dann ein Event an TTS Bloc schicken kann. Dieses Event enthält Information über den Text der Frage selbst und der Sprache, in welcher die Frage definiert wurde.

The image shows a split-screen view. On the left is a screenshot of a mobile application titled "Sachkunde". It displays a question: "Wie viele Knochen hat das menschliche Skelett?" (How many bones does the human skeleton have?) and four answer options: "genau 306", "über 200", "genau 99", and "über 1000". On the right is a side-by-side comparison of the Dart code for this screen and a visual representation of how it would look on a device. The code highlights several parts with annotations:

- "TTS Bloc Instanz als DI ins Widget Tree zu liefern" (Deliver TTS Bloc instance as DI into the Widget Tree) points to the line where the BlocProvider is created.
- "BlocBuilder behandelt das Erstellen eines Widgets" (BlocBuilder handles the creation of a widget) points to the BlocBuilder definition.
- "Anhand des States wird Event an bloc geschickt" (Event is sent to the bloc based on the state) points to the logic where the BlocBuilder checks the state and sends an event to the bloc.

Abbildung 2.4-7: Bloc Konzept Umsetzung. Wichtige Abschnitte

Allerdings reicht einmaliges Vorlesen der Aufgabenstellung oder Frage noch nicht aus. Falls der Nutzer den Text nicht genau verstanden hat, da zum Beispiel der Ton zu leise war, oder nur die Frage vorgelesen wurde und der Nutzer auch noch die Aufgabenstellung hören möchte. Muss es eine Möglichkeit geben, diese Texte durch Klicken vorlesen zu lassen. Weiter unter in Widget Baum benutzt man deshalb Inkwell-Widgets, um die Fragen und Aufgabenstellungen darzustellen, da diese die onTap Properties haben. Beim einmaligen Anklicken des Widgets wird ein Event erzeugt, dass den Text, der im jeweiligen Widget enthalten ist, beinhaltet und gibt dieses Event an TTS Bloc weiter. So kann man diese Texte die in den jeweiligen Felder enthalten sind vorlesen.

Auch die Antwortmöglichkeiten, falls es diese gibt, sollen vorgelesen werden können. Auch bei den Antwort Widgets haben wir uns für InkWell Widgets entschieden, denn sie haben nicht nur das OnTap – Properties, sondern auch die OnDoubleTap Properties, was für uns sehr nützlich ist, denn es gibt für jede Antwort zwei mögliche Aktionen, zum Einen das Auswählen als Antwort auf die Frage, zum Anderen das Vorlesen der Frage. Wir haben uns entschieden das Auswählen der Antwort auf „onTap“ zu belassen, da dies einfacher für Nutzer ist die, das TTS Feature nicht benutzen wollen, da ein Doppelklick, um eine Antwort auszuwählen, wenn TTS sowieso ausgeschaltet ist, nicht sehr intuitiv ist. Wir haben uns allerdings auch dagegen entschieden, das mit dem Ein- und Ausschalten von TTS zu verändern. Also, dass wenn TTS aktiviert wäre „onDoubleTap“ die Antwort auswählt und „onTap“ die

Praktische Umsetzung

Frage vorliest, denn das könnte Nutzer verwirren, wenn sie zum Beispiel: TTS aus Versehen ausschalten würden und „onTap“ nun sofort die Antwort auswählen würde, anstatt sie nur vorzulesen. Also haben wir uns entschieden die App einheitlich zu halten, also dass „onDoubleTap“ die Antworten vorliest und „onTap“ die Antwort auswählt.

Genau wie bei den Fragen funktioniert es mit Antworten, in onDoubleTap Methode wird zuerst TTS Bloc in Widget-Tree gefunden, was BlocProvider bereitgestellt hat und dann wird das Event an bloc geschickt, in diesem Fall „ClickOnQuestionEvent“ (siehe Abbildung 2.4-8: Bloc Konzept Umsetzung Four Card Task Screen). Dieses Event enthält genauso wie bei der Frage die Info über den vorzulesenden Text und seine Sprache.



Abbildung 2.4-8: Bloc Konzept Umsetzung Four Card Task Screen

Genauso wie beim Four Card Task Screen sind andere Task Screens mit dem TTS Bloc versorgt.

In Abbildung 2.4-9: Bloc Konzept Umsetzung Match Category Task Screen wird dasselbe nochmals für den Match Category Task Screen mit Codeauszuge dargestellt. Wie man sieht, muss auch hier der BlocProvider, den TTS Bloc an den Widget-Tree weitergeben. Danach achtet BlocBuilder auf Zustandsänderungen und sucht nach dem notwendigem Bloc Element weiter oben im Widget-Tree. Danach ermöglicht der BlocProvider ein neues Event, mit Hilfe der „add“ Methode weiterzugeben.

Praktische Umsetzung

```

return BlocProvider(
  create: (context) => TTSBloc(),
  child: Column(
    children: [
      // Lama Speechbubble
      // Aufgabestellung vorlesen jedes mal wenn screen neu geladen w...
      BlocBuilder<TTSBloc, TTSState>(
        builder: (context, state) {
          ...
        },
      ),
    ],
  ),
),
child: InkWell(
  onTap: () => {
    BlocProvider.of<TTSBloc>(context).add(
      ClickOnQuestionEvent.initVoice(
        task.lamaText!, qlang), // ClickOnQuestionEvent.initVoice
    );
  },
),
child: InkWell(
  onDoubleTap: () => {
    String lang;
    task.answerLanguage == null ? lang = "Deutsch" : lang =
    BlocProvider.of<TTSBloc>(context).add(
      ClickOnAnswerEvent(items[i].item!, lang));
  },
),

```

Annotations on the left side of the code:

- Bloc an Widget liefern**: Points to the first `BlocProvider` in the code.
- TTS Bloc wird gesucht**: Points to the `BlocBuilder` in the code.
- Ein mal Klick**: Points to the `onTap` event in the code.
- Frage vorlesen**: Points to the `ClickOnQuestionEvent.initVoice` method in the code.
- Zwei mal anklicken**: Points to the `onDoubleTap` event in the code.
- Antwort vorlesen**: Points to the `ClickOnAnswerEvent` method in the code.

Abbildung

Abbildung 2.4-9: Bloc Konzept Umsetzung Match Category Task Screen

2.4.3 readText Methode

Wie eben beschrieben wird der Text erst in der „readText“ Methode in der Klasse: „tts_bloc.dart“ vorgelesen(siehe Abbildung 2.4-10 readText Methode).

```

readText(String text, String lang) async {
  if(!home_screen_state.isTTs()) {
    return;
  }
  if (lang == "Englisch") {
    await flutterTts.setLanguage("en-US")
    await flutterTts.setSpeechRate(0.4);
    await flutterTts.setPitch(1.1);
  } else {
    await flutterTts.setLanguage("de-De")
  }
  await flutterTts.setVolume(1.0);
  await flutterTts.speak(text);
}

```

Abbildung 2.4-10 readText Methode

Praktische Umsetzung

Bevor der Text mit der „`flutterTts.speak()`“ vorgelesen wird, werden allerdings noch zwei Überprüfungen durchgeführt. Diese überprüfen, ob TTS überhaupt eingeschaltet ist und, in welcher Sprache der Text verfasst ist.

2.4.4 Ein und Ausschalten von TTS

In der ersten if-Abfrage wird geprüft, ob TTS überhaupt eingeschaltet ist denn es kann natürlich vorkommen, dass der Nutzer die TTS Funktion gar nicht benutzen will, wenn man zum Beispiel in einem öffentlichen Raum ist und andere Menschen nicht nerven will, deshalb sollte es wie im Pflichtenheft gefordert, eine Möglichkeit bestehen das Feature auch auszuschalten.

Die Option sollte es im Task Menü geben, damit der Nutzer, bevor er überhaupt Tasks auswählt, die TTS Funktion ausschalten kann. Die Option sollte sehr einfach ersichtlich sein, weshalb wir uns entschieden haben, einen Flutter Icon Button zu verwenden, welcher mehrere Symbole zur Verfügung stellt, und haben uns hierbei für einen Lautsprecher entschieden, aus dem Tonwellen kommen. Außerdem haben wir uns entschieden den Button oben rechts in der Ecke, sehr einfach auffindbar für den Nutzer unterzubringen. Wenn man TTS über diesen Button dann ausschaltet, sollte dies auch einfach ersichtlich sein, das Symbol wechselt dann einfach auf einen Lautsprecher ohne Tonwellen. Ein erneuter Klick würde TTS wieder anschalten.



Abbildung 2.4-12 TTS eingeschaltet

Abbildung 2.4-12 TTS ausgeschaltet

Auch während eines Task falls er sie vergessen hat auszuschalten oder sie ausgeschaltet war und wieder anschalten will. Um es einheitlich mit dem Task Menü Screen zu halten haben wir uns für dasselbe Symbol in derselben Stelle entschieden, dass sich exakt so verhält. Allerdings befindet sich dort in der Ecke bereits der Lama Coin der indiziert, ob man durch das Lösen der Aufgabe noch Punkte verdienen kann, weshalb der Coin ein Stück nach links verschoben wurde, um Platz für den Ton Button zu schaffen(siehe Abbildung 2.4-14 TTS eingeschaltet und Abbildung 4.4-15 TTS ausgeschaltet).

Praktische Umsetzung



.. Dino ist groß

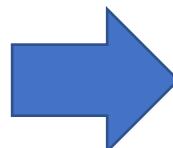


Tippe die richtige Antwort an!

Das

Der

Die



.. Dino ist groß



Tippe die richtige Antwort an!

Das

Der

Die



Abbildung 2.4-14 TTS eingeschaltet



Abbildung 2.4-13 TTS ausgeschaltet

Tatsächliche Umsetzung:

Der Status von TTS wird in einer statischen bool Variable, in einer externen Klasse gespeichert (siehe Abbildung 4.4-18 TTS Zustandsklasse). Der Wert der Variable entspricht dabei dem Zustand von TTS, ist die Variable „true“ ist TTS angeschaltet, ist die Variable „false“, ist TTS ausgeschaltet. Bei einem Klick auf den TTS Button wird die toggle Methode aufgerufen (siehe Abbildung 2.4-17 Button auf dem Screen), die den Wert von isTTSTurnedOn umschaltet. Außerdem wird sofort jedes Vorlesen eines Textes abgebrochen, falls TTS durch den Klick auf den Button ausgeschaltet wird.

```
class home_screen_state {
    static bool isTTSTurnedOn = true;           ← Zustand von TTS
    static FlutterTts flutterTts = FlutterTts();

    static toggle() async {
        if(isTTSTurnedOn) {
            isTTSTurnedOn = false;
            await flutterTts.stop();
        } else {
            isTTSTurnedOn = true;
        }
    }
    static isTTs() {                         ← Gibt Status von TTS zurück
        return isTTSTurnedOn;
    }
}
```

Abbildung 2.4-16 TTS Zustandsklasse

Praktische Umsetzung

```
child: IconButton(  
    onPressed: () {  
        home_screen_state.toggle();  
    },
```

Abbildung 2.4-17 Button auf dem Screen

2.4.5 Sprachen

In der zweiten If-Abfrage wird überprüft, ob die Sprache, in der, der Text vorgelesen werden soll Englisch oder Deutsch ist, denn es gibt Kategorien in denen Englisch verwendet wird, im Moment gibt es nur die Kategorie "Englisch", in der das der Fall ist (siehe Abbildung 4.4-21 Four Cards in Englisch).

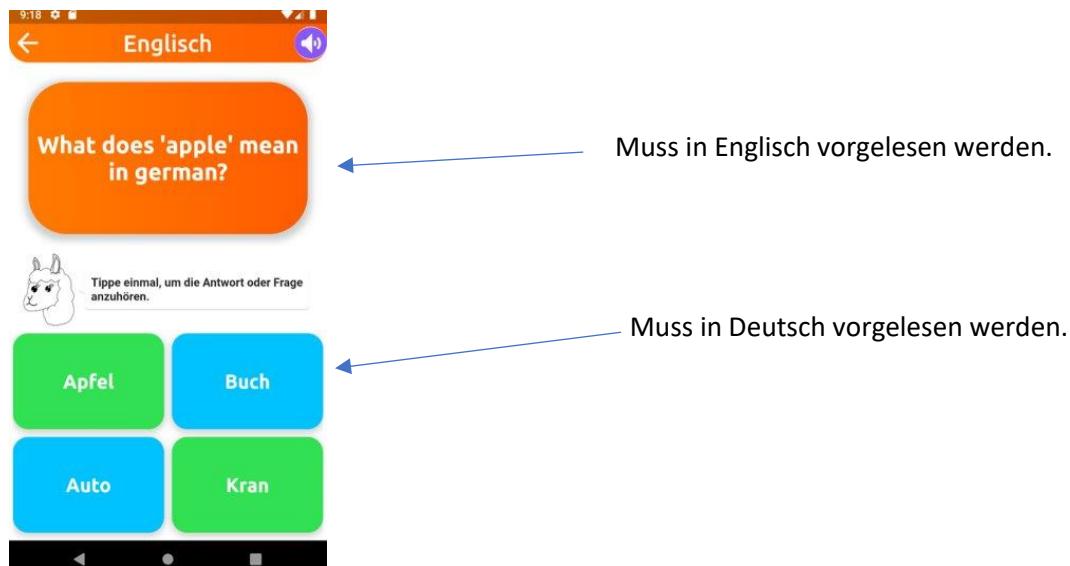


Abbildung 2.4-19 Four Cards in Englisch

Bei keinem Aufgabentypen lässt sich direkt feststellen in welcher Sprache der Text vorgelesen werden sollte (Bild), muss man beim Erstellen einer Frage die Sprache festlegen in der diese vorgelesen werden soll (siehe Abbildung 2.4-20 JSON mit Sprachen).

Praktische Umsetzung

```
"task_type": "4Cards",
"task_reward": 2,
"question": "What does 'apple' mean in german?",
"lama_text": "Tap the right Answer!",
"left_to_solve": 3,
"right_answer": "Apfel",
"wrong_answers": ["Auto", "Buch", "Kran"],
"question_language": "Englisch",
"answer_language": "Deutsch"
}
```

Abbildung 2.4-20 JSON mit Sprachen

Um das Erstellen von Aufgaben, die keine anderen Sprachen benutzen zu vereinfachen, kann man die Attribute weglassen und Deutsch wird als Default Wert genommen. In dem Task werden zwei Attribute angelegt, um die Sprache in dem die Antwort oder die Frage vorgelesen werden soll gespeichert (siehe Abbildung 2.4-21 Taskklassen mit Sprachen)

```
class Task4Cards extends Task {
    String? question;
    String? rightAnswer;
    List<String> wrongAnswers;
    String? questionLanguage;
    String? answerLanguage;
```

Abbildung 2.4-21 Taskklassen mit Sprachen

Wie bereits erwähnt kann man die Attribute auch leer lassen, um Deutsch automatisch auszuwählen. Um sicherzustellen, dass die Null Safety eingehalten wird, überprüft man vor der Übergabe die Sprache, ob die Variablen null sind und übergibt dann automatisch „Deutsch“ ansonsten ist ja bereits eine Sprache gesetzt und man kann diese übergeben (siehe Abbildung 2.4-22 Check, ob die Sprache null ist).

```
String qlang;
task.questionLanguage == null ? qlang = "Deutsch" : qlang = task.questionLanguage!;
```

Abbildung 2.4-22 Check, ob die Sprache null ist

Wenn man nun einen Text vorlesen will, muss man die Sprache mit übergeben, falls man Englisch mit übergibt, wird die Sprache, in der vorgelesen wird, auf Englisch gesetzt, ansonsten wird automatisch Deutsch ausgewählt.

3 Deployment und Installation der App

Die App ist auf Githhub unter folgendem Repository gespeichert:

<https://github.com/thm-mni-ii/lama>

Mit dem Code in diesem Repository lässt sich die App auf einem Gerät deployen. Dies lässt sich über verschiedene IDEs wie beispielsweise Visual Studio oder Android Studio realisieren. Um die App zu deployen muss also zunächst das Repository lokal auf dem Rechner gespeichert werden. Dies lässt sich über den „git clone“ Befehl realisieren. Zum Deployen der App auf das Gerät muss das Projekt in der jeweiligen IDE geladen werden. Wenn ein Projekt auf einem Android Gerät deployt werden soll, muss sicher gestellt werden, dass dieses per USB mit dem PC verbunden ist und die Entwickleroptionen auf dem Android Gerät aktiviert sind. Weiterhin lässt sich die App auf einem Emulator deployen. Dies hat den Vorteil, dass man die App auf verschiedenen Displaygrößen (auch Tablets) testen kann. Das Deployen findet statt, indem man in der IDE das Gerät für das Deployment auswählt und anschließend über ein Run Button das Deployment beginnt (siehe Abbildung 2.4-1).



Abbildung 2.4-1: Deployment der Lama App auf ein Android Gerät

Um neue Codeabschnitte schnell auf einem Gerät zu testen, muss allerdings nicht zwangsweise die App noch mal komplett auf das Gerät durch Klicken des Run-Buttons deployt werden. Stattdessen kann man einen Hot Reload verwenden. Mit einem Hot Reload lassen sich alle Neuerungen in der App auf das gewünschte Gerät übertragen, ohne alle bereits geladenen Daten erneut zu laden. Diese Vorgehensweise geht wesentlich schneller als das erneute Übertragen aller Daten auf das Gerät. Im Falle der Lama App bedeutet ein Hot Reload auch, dass man nicht wieder auf den Startbildschirm der App geführt wird, sondern auf dem aktuellen Bildschirm bleibt. Ein Hot Reload lässt sich in Android Studio durch das Drücken des Blitz Symbols erreichen (siehe Abbildung 1.1-1/Abbildung 2.4-2).

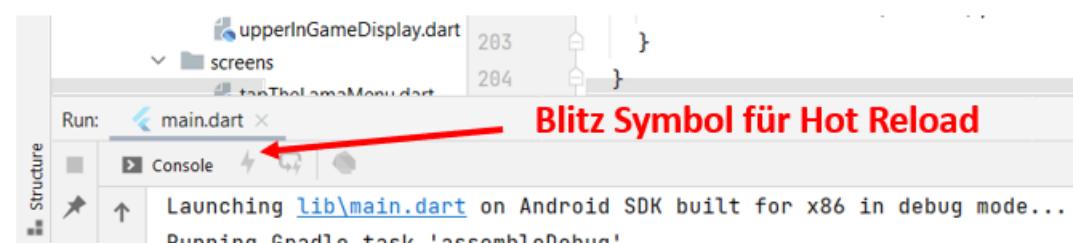


Abbildung 2.4-2: Hot Reload in Android Studio

4 Weiterentwicklungsmöglichkeiten

In diesem Abschnitt wird aufgezeigt, inwieweit die in diesem Projekt erstellten, neuen Funktionalitäten der Lama-App verbessert werden könnten.

4.1 Tap the Lama

Tap the Lama ist ein vollfunktionsfähiges Spiel. Nichtsdestotrotz kann das Spiel in einigen Teilen der grafischen Benutzeroberfläche optimiert werden. Zum einen können Optimierungen im Startmenü und im Spiel selbst erfolgen. So könnte im Startmenü noch der Hinweis gegeben werden, dass man keine roten Lamaköfe wegdrücken soll. Weiterhin kann der Start Button für Tablets etwas größer gemacht werden (siehe Abbildung 4.1-1). Der Text des Start Buttons ist auf einem Tablet zwar gut lesbar, könnte aber trotzdem durch eine Vergrößerung optisch ansprechender wirken.

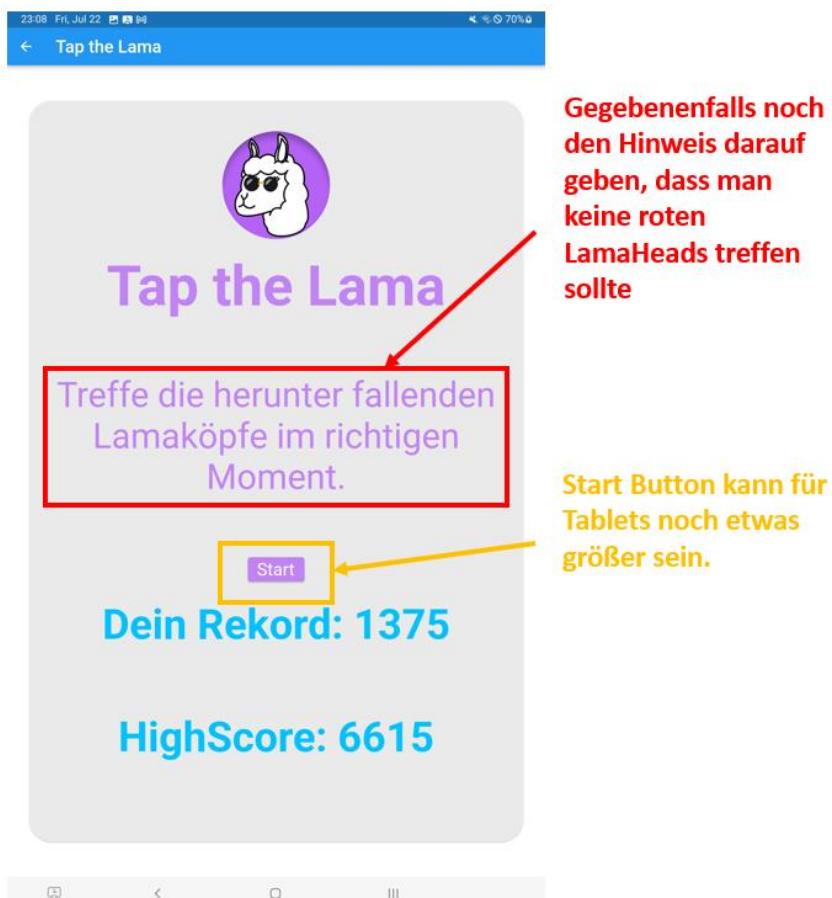


Abbildung 4.1-1: mögliche Verbesserungen im Start Menü

Zur optischen Verbesserung des Spiels könnten beispielsweise die LamaHeads transparent gestaltet werden. Die hätte den Vorteil, dass sie sich beim Schweben über die LamaButtons mehr von diesen abheben würden. Eine Realisierung dessen kann darüber erfolgen, dass die jeweilige LamaHead Komponente auf eine png-Datei oder eine svg-Datei zugreift, welche transparent ist. Weiterhin könnte die LifeBar Komponente im Spiel etwas kürzer und mit abgerundeten Ecken gestaltet werden, da diese Veränderung in der grafischen Benutzeroberfläche vom Benutzer als moderner empfunden werden könnte. Eine weitere Veränderung der Spieloberfläche könnte darin bestehen, dass beim

Weiterentwicklungsmöglichkeiten

Drücken eines roten LamaHeads der Bildschirm kurz rot aufleuchtet oder ein Toast Text in roter Aufschrift erzeugt (siehe Abbildung 4.1-2).

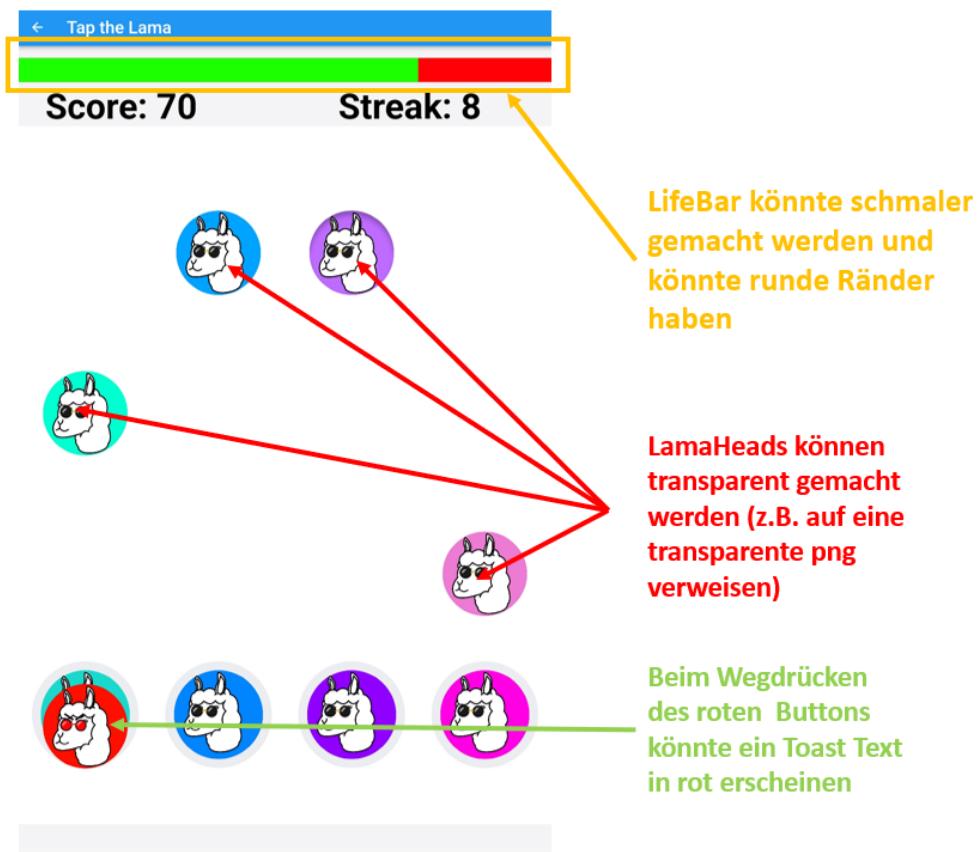


Abbildung 4.1-2: mögliche Verbesserungen im Spiel

Neben den Verbesserungen im Tap the Lama Spiel gilt es zu beachten, dass noch nicht alle über die Spielliste auswählbaren Spiele funktionieren [Stand: 01.08.2022]. Dies liegt daran, dass noch nicht alle Spiele vollständig auf die Flame Version 1.1.1 umgestellt wurden. Folgende Spiele sind davon betroffen:

- Snake
- Flappy-Lama
- Affen-Leiter

4.2 Taskset Erstellung

4.2.1 Weitere Screens hinzufügen

Selbstverständlich lassen sich auch für kommende Aufgabentypen noch Screens zum Erstellen hinzufügen. Dieses kann unter der Anleitung (Kapitel 4.2.6) einfach gemacht werden.

4.2.2 Screens automatisch generieren lassen

Damit Screens nicht immer händisch erstellt werden müssen, wäre es eine Möglichkeit, diese automatisch anhand der für den Task benötigten JSON – Formatierung generieren zu lassen.

Zur Darstellung und für einen einheitlichen Look könnte man die bereits benutzten Custom – Widgets benutzen (2.2.3). In der JSON müsste dann angegeben sein, welcher Datentyp für den spezifischen Key benötigt wird und es müsste dann intern eine Zuordnung von Datentyp und Widget geben (z.B. int -> NumberInputWidget).

Anhand dieser Zuordnung wüsste die App, welche Widgets dem Screen bei Abruf hinzugefügt werden muss und kann diesen dementsprechend darstellen.

Dies würde den Aufwand zur Erstellung neuer Screens auf ein Minimum reduzieren.

4.2.3 Preview – Funktion

Ein Feature, welches es nicht ins Endergebnis geschafft hat, ist die Previewfunktion. Der Lehrer könnte im Screen eines Tasks auf einen Button (Beispiel anzeigen) klicken und kann die Aufgabe mit seinen Angaben testen.

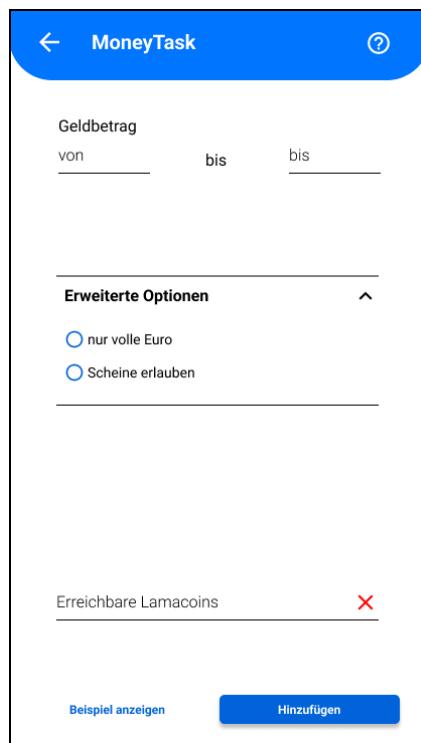


Abbildung 4.2-1 Screen mit Preview – Funktion

4.3 Aufgabenverwaltung

4.3.1 Aufgabenverwaltung

Für die Weiterentwicklung dieses Features ist eine Recht große Strukturelle Veränderung nötig. Momentan, mit der Einführung dieses Features, ist es dem Lehrer nur möglich bestimmte Tasksets für die Bearbeitung der User zuzulassen. Dies ist zu sehen, da die variable `isInPool` an das Taskset gebunden ist und nicht an die User. Somit ist das Taskset für alle User gleich entweder eingeschaltet oder pausiert.

```
class Taskset {  
    String? name;  
    TaskUrl? taskurl;  
    String? subject;  
    String? description;  
    int? grade;  
    bool? randomizeOrder;  
    int? randomTaskAmount;  
    List<Task>? tasks;  
    bool isInPool = false;
```

Jedoch sollten alle User individuell betrachtet werden und somit auch jeweils einen eigenen Pool an aufgaben besitzen. Daraus resultierend, sollte man jedem User eine Tasksetliste mitgeben. Diese enthält die Tasksets, die vom User einzusehen sind. Man könnte beim Taskset eine id hinterlegen und diese in die Liste des Users schreiben. Dann würden beim Laden der Inhalte nur die, für den User, Relevaten Aufgaben angezeigt werden.

```
List<String> tasksetList;
```

Des Weiteren sollte in folgenden Schritten ein Auge auf das Errorhandling gelegt werden, um eine bessere UX zu generieren. Beispielsweise gibt es keine Response, wenn ein Fehler beim Laden der Tasksets geschieht. Dabei sollte Fehler in der Passwort eingabe oder der URL erkannt werden. Momentan würde lediglich eine leere Liste dem Benutzer zur Verfügung stehen, falls es zu Verbindungsfehlern kommt. Dieser jedoch keine Kenntnis über den Grund. Die Grundlage ist geschaffen, da teilweise ein String zurückgegeben wird. Der kann verwendet werden, um ein Fehler Screen anzuzeigen.

4.4 Text to Speech

Text to Speech Feature erfährt alle in Pflichtenheft genannte Anforderungen. Allerdings kann man die Funktionalitäten noch relativ leicht weiter erweitern. Zum einen kann die Auswahl der Sprachen oder Dialekten erweitert werden. Es lässt sich durch das Hinzufügen weniger Zeilen in `tts_bloc.dart` Datei zu implementieren, siehe Abbildung 4.4-1. Es gibt insgesamt 63 Sprachen, die potenziell eingebaut werden können.

Weiterentwicklungsmöglichkeiten

```
await flutterTts.setLanguage("en-AU");
//
await flutterTts.setLanguage("it-IT");
//
await flutterTts.setLanguage("fr-FR");
```

Abbildung 4.4-1

Um die möglichen Sprachen und deren Abkürzungen zu finden, kann man eingebaute Methode getLanguages benutzen:

```
List<dynamic> languages = await flutterTts.getLanguages;
print(languages.toString());

[ko-KR, mr-IN, ru-RU, zh-TW, hu-HU, th-TH, ur-PK, nb-NO, da-DK, tr-TR,
NL, fr-CA, sr, pt-BR, ml-IN, si-LK, de-DE, ku, cs-CZ, pl-PL, sk-SK, fil-
```

Abbildung 4.4-2

Außerdem könnte man es dem Nutzer ermöglichen Tonhöhe und Geschwindigkeit einzustellen, falls dieser Probleme hat, die Stimme zu verstehen. Eine mögliche Realisierung wäre, ein neues Widget in Einstellungsmenü einzubauen (siehe Abbildung 4.4-3).

```
Widget _rate() {
  return Slider(
    value: rate,
    onChanged: (newRate) {
      setState(() => rate = newRate);
    },
    min: 0.0,
    max: 1.0,
    divisions: 10,
    label: "Rate: $rate",
    activeColor: Colors.green,
  );
}
```

Abbildung 4.4-3

Text to Speech Feature erfüllt alle in Pflichtenheft genannte Anforderungen. Allerdings kann man die Funktionalitäten noch relativ leicht weiter erweitern. Zum einen kann die Auswahl der Sprachen oder Dialekten erweitert werden. Es lässt sich durch das Hinzufügen weniger Zeilen in tts_bloc.dart Datei zu implementieren, siehe Abbildung 4.4-1. Es gibt insgesamt 63 Sprachen, die potenziell eingebaut werden können.

Weiterentwicklungsmöglichkeiten

```
await flutterTts.setLanguage("en-AU");
//
await flutterTts.setLanguage("it-IT");
//
await flutterTts.setLanguage("fr-FR");
```

Abbildung 4.4-4

Um die möglichen Sprachen und deren Abkürzungen zu finden, kann man eingebaute Methode getLanguages benutzen:

```
List<dynamic> languages = await flutterTts.getLanguages;
print(languages.toString());

[ko-KR, mr-IN, ru-RU, zh-TW, hu-HU, th-TH, ur-PK, nb-NO, da-DK, tr-TR,
NL, fr-CA, sr, pt-BR, ml-IN, si-LK, de-DE, ku, cs-CZ, pl-PL, sk-SK, fil]
```

Abbildung 4.4-5

Außerdem könnte man es dem Nutzer ermöglichen Tonhöhe und Geschwindigkeit einzustellen, falls dieser Probleme hat, die Stimme zu verstehen. Eine mögliche Realisierung wäre, ein neues Widget in Einstellungsmenü einzubauen (siehe Abbildung 4.4-6 **Fehler! Verweisquelle konnte nicht gefunden werden.**).

```
Widget _rate() {
    return Slider(
        value: rate,
        onChanged: (newRate) {
            setState(() => rate = newRate);
        },
        min: 0.0,
        max: 1.0,
        divisions: 10,
        label: "Rate: $rate",
        activeColor: Colors.green,
    );
}
```

Abbildung 4.4-6

Weiterentwicklungsmöglichkeiten