

DHBW Mosbach Onlinemedien 2021

Frontend-Entwicklung

Dokumentation & Reflexion

Zum

Einzelprojekt

Breakoutgame

Entwicklerin: Isabel Köcher

Dozent: Josef Slezak

Abgabe: 13.05.2022

**1 Gliederung**

2 Spielidee & Einstieg

3 Anleitung

4 Dokumentation

4.1 Fremdcode als Grundlage & Einstieg

4.2 Ideen für Erweiterung/Ausbau des Spiels

4.3 Umsetzung der Ideen

4.4 Struktur des Codes

5 Reflexion

Lernfortschritt

Kenntniserweiterung

Schwierigkeiten?

Ausblick/Fazit

Spielkonzept

* Startseite
  + HelpButton für Anleitung
  + Enter name um Namen / Spieler anzulegen und in LocalStorage zu speichern
  + Start The Game um zum Spiel zu gelangen
* Voraussetzung um Spiel zu spielen ist, dass der Spieler einen Namen hat (selber einen eingeben oder bei Default als Mysterie Player spielen)
* Spielseite
  + Canvas für das Spiel selber
  + Play Button, um Spiel zu starten
  + Current Score – zählt wie oft, Bricks getroffen wurden (ist immer gleich für jeden Spieler)
  + Timer – zählt wie lange ein Spieler für das jeweilige Level braucht (wird dann im Local Storage gespeichert) 🡪 Problem mit richtiger Zeitanzeige und Speicherung
  + Highscores – hier werden die Scores für alle im LocalStorage gespeicherten Spieler angezeigt (maximal einmal pro Level 🡪 Es wird nicht die kürzeste Zeit angezeigt, sondern immer die letzte überschrieben) 🡪 Problem: jeder Spieler braucht für jedes Level gleich lang (37 Sekunden für das erste) 🡪 daher random Startposition für den Ball
* Level
  + 5 Stück
  + Unterscheiden sich in Bricks, Ball/ Paddle Speed und Paddle Breite
* Weitere Ideen
  + Lösung des Ball / Paddle Speed Problems
  + Highscore verbessern, dass immer nur die beste Zeit angezeigt wird
  + Leben einbauen 🡪 drei Leben, dann muss man wieder bei Level 1 anfangen
  + Zurück Button
  + Beginner, Intermediate und Expert Levels

# **Inhaltsverzeichnis**

[**1** **Inhaltsverzeichnis** 4](#_Toc102951346)

[**2** **Spielidee & Einstieg** 5](#_Toc102951347)

[**3** **Anleitung** 5](#_Toc102951348)

[**4** **Dokumentation des Spiels** 6](#_Toc102951349)

[4.1 Fremdcode als Grundlage 6](#_Toc102951350)

[4.2 Ideen für Erweiterungen / Ausbau des Spiels 6](#_Toc102951351)

[4.3 Umsetzung der Ideen 6](#_Toc102951352)

[4.4 Struktur des Codes 6](#_Toc102951353)

[**5** **Reflexion** 6](#_Toc102951354)

# **Spielidee & Einstieg**

Aller Anfang ist schwer – das trifft auch auf die Suche nach einer geeigneten Projektidee für dieses Einzelprojekt zu. Ich habe viel recherchiert, die Beispielprojekte bei GitHub durchstöbert und zusätzlich einige Tutorial Videos bei Youtube angeschaut, bevor ich mich schließlich für die Idee „BreakoutGame“ entschieden habe.

Die Idee und Teile des Codes, sowie Bilder sind Fremdeigentum und basieren auf dem Youtube Video von freeCodeCamp, in dem schrittweise ein Arkanoid Game erstellt und zum Laufen gebracht wird. Aufbauend auf dieser Logik wurde das nun vorliegende Spiel erweitert und die Codelogik eigenständig ausgebaut. Die übernommenen Codeteile sind im Code entsprechend markiert.

# **Anleitung**

Um das Spiel zu starten, bitte „npm run watch“ in das Terminal eingeben und das Spiel mit dem Live-Server von Visual Studio Code im Browser öffnen.

Es erscheint die Startseite mit drei Buttons. Der Spieler muss über den „Enter Name“-Button zunächst seinen Spielernamen eingeben, um auf den „Start the Game“-Button klicken zu können. Der Name des Spielers wird nach erfolgreicher Eingabe auf der Startseite angezeigt (Hinweis: jeder Spielername kann nur ein einziges Mal existieren). Über den mittleren Button gelangt der Spieler zum eigentlichen Spiel. Er/Sie kann das Spiel nun über den „Play“-Button starten.

Ziel des Breakoutgame ist es, in jedem Level so schnell wie möglich mit dem Ball alle Bricks im Canvas zu treffen. Der Spieler muss dabei mit dem Paddle (Spielbalken am Boden des Canvas) den Ball in der Luft halten. Die Steuerung des Paddles erfolgt mit der linken und rechten Pfeiltaste der Tastatur. Der Ball wird zu Beginn des Spiels automatisch gestartet und ändert bei jedem Aufprall (Wand, Brick oder Paddle) die Richtung. Für jedes Mal, bei dem der Spieler einen Brick trifft, wird der Score um eins nach oben gezählt. Sind alle Bricks getroffen, so hat der Spieler gewonnen und kann über den Button das nächste Level starten. Verliert der Spieler den Ball und fällt dieser auf den Boden des Canvas, so muss das Level erneut gestartet werden.

Das Spiel besteht aus fünf unterschiedlichen Levels, je höher das Level, desto schwieriger wird es für den Spieler. Die Schwierigkeit entsteht dadurch, dass die Bricks unterschiedlich oft getroffen werden müssen, der Ball schneller und das Paddle schmaler wird.

Anmerkung: Das Spiel verwendet den LocalStorage des Browsers, um Highscore Informationen zu einzelnen Spielern zu speichern.

# **Dokumentation des Spiels**

## 4.1 Fremdcode als Grundlage

Zum Einstieg wurde das Tutorial von freeCodeCamp durchgearbeitet. Der Code wurde zunächst verstanden, dann schrittweise im eigenen Projekt implementiert, mit eigenen Ideen und Logiken erweitert und in logische Module aufgeteilt. Folgende Ideen und Logiken wurden übernommen:

* Bilder der Spielelemente Ball, Paddle und Bricks
* Brick Klasse – Variablen und Funktionen wurden unter anderem Namen implementiert.
* Ball Klasse – Variablen und Funktionen wurden unter anderem Namen implementiert und die Klasse wurde um zwei interne SetterFunctions und einer zu exportierenden Funktion erweitert.
* Paddle Klasse – Variablen und Funktionen wurden unter anderem Namen implementiert.
* Canvas View Klasse – Variablen und Funktionen wurden unter anderem Namen implementiert. Andere Imports wurden aufgrund der eigenen Logik benötigt.
* Collision Klasse - Variablen und Funktionen wurden unter anderem Namen implementiert.
* Helpers.ts – wurde zu brick-array.ts und mit eigener Spiellogik verändert.
* Setup.ts – die Setups für die einzelnen Game-Elemente wurden in einzelnen Dateien ausgelagert und im Ordner setup-helpers gespeichert. Das Setup für den Ball wurde eigenständig modifiziert.
* Index.ts – die eigentliche Spiel Logik wurde in game.ts ausgelagert und basierend auf den eigenen Änderungen und Spielerweiterungen angepasst.

Aus Platz Gründen werden die übernommenen Logiken im Folgenden nicht weiter erläutert.

## 4.2 Ideen für Erweiterungen / Ausbau des Spiels

Um zu zeigen, dass der Code nicht nur verstanden wurde, sondern um die geforderte Eigenleistung für dieses Projekt zu erbringen und die eigenen Fähigkeiten auszubauen, wurden Ausbaumöglichkeiten des Spiels festgehalten.

Idee 1 – Design verschönern & Startseite hinzufügen

Während der Bearbeitung des Tutorials stand bereits fest, dass Design und Aufbau des Spiels angepasst und verschönert werden müssen. Unter anderem entstand die Idee, dass der Spieler zuerst auf einer Startseite beginnt, bevor er zum eigentlichen Spiel gelangt.

Idee 2 – Spieler(namen) anlegen

In Zusammenhang mit Idee 3 entstand der Wunsch, den Spieler einen Namen eingeben zu lassen, um das Spiel zu personalisieren.

Idee 3 – Highscores zeigen

Um das Spiel einen weiteren Touch von Gamification zu geben, wurde entschieden eine Art Highscore für den Spieler anzeigen zu lassen. Aufgrund der Tatsache, dass der Score für ein Level unveränderlich und für jeden Spieler der gleiche ist, entstanden im Laufe der Entwicklung weitere Erweiterungen: Timer Funktion und Nutzung des LocalStorages zur Speicherung.

Idee 4 – Hilfe einblenden

Bereits zu Beginn des Projekts war klar, dass es für den Spieler die Möglichkeit geben sollte, sich auf Wunsch vor dem Spiel eine Art Anleitung durchlesen zu können.

Idee 5 – Verschiedene Levels

Es entstand die Idee, verschiedene Levels und Schwierigkeiten einzubauen, um das Spiel interessanter und spannender zu gestalten. Zuerst war der Gedanke, nur mit einer unterschiedlichen Anordnung und Anzahl von Bricks zu arbeiten, später kam dann noch die Idee hinzu, in den höheren Leveln den Ball schneller und das Paddle schmaler werden zu lassen.

## 4.3 Umsetzung der Ideen

Idee 1 – Design verschönern & Startseite hinzufügen

Das Design bzw. der Aufbau der Spielseite wurden in dem Tutorial sehr einfach gehalten. Somit wurde zunächst das Styling der Seite über das CSS-Sheet vorgenommen. Ziel dieses Projekts ist es unter anderem die eigenen Fähigkeiten in Typescript zu üben und auszubauen. Daher wurde beschlossen den Wechsel von einer Startseite zu der bestehenden Spielseite nur durch DOM-Manipulation zu bewirken.

Als erster Arbeitsschritt wurde ein neues Modul domutils.ts erstellt. Es bestand die Absicht, in diesem alle Variablen anzulegen, die zur Speicherung der HTML-Elemente dienen, und diese so im Verlauf des Spiels manipulieren zu können. Anschließend wurde das Modul gamepage-setup.ts erstellt, um die Logik zu implementieren, die durch DOM-Manipulation beeinflusst, dass die Seite von der Startseite zur Gameseite wechselt („changeToGamePage“). Unter anderem werden folgende Änderungen mit letzterer Methode vorgenommen: das im HTML eingebundene CSS-Stylesheet wird mit dem Stylesheet für die Gameseite ausgetauscht, der Header Bereich wird angepasst und im Main Bereich werden die Elemente ebenfalls ausgetauscht (vgl. index.html, changeHeader und changeMain Methoden). Im gleichen Modul befindet sich eine weitere Methode, „displayHighscoreList“. Diese erhält als Parameter ein Array aller gespeicherten Player (s. LocalStorage) und prüft, ob es bereits Spieler mit einem Highscore gibt. Ist dies der Fall, so werden die entsprechenden Scores auf der Gameseite als „Spielername: Level Time“ unter Highscores dargestellt. Gibt es noch keine Highscores wird diese Nachricht dem Spieler angezeigt.

*//set up the highscore list on gamepage in html*

export function **displayHighscoreList**(playerList: **Player**[]) {

*//loop over all players in playerlist*

  let atLeastOneHighscore = false;

  for (let player of playerList) {

*//if players, have highscores, then display PlayerName: Level Time*

    if (player.highscore !== undefined) {

      atLeastOneHighscore = true;

*//show all highscores in highscore array in html*

**displayPlayerHighscores**(player);

    }}

*// if no player has an highscore yet, show one time in html*

  if (!atLeastOneHighscore) {

    const liItem = **createNewListItem**();

    highscoreList.**appendChild**(liItem);

    liItem.innerHTML = "<li>No highscores yet</li>";

  }}

Als nächstes wurde die Implementierung der Logik des „Start the Game“-Buttons auf der Startseite vorgenommen. Wie bereits zuvor beschrieben bewirkt ein Klick-Event auf den zweiten Button, dass – sofern der Spieler einen Namen hat (s. Umsetzung Idee 2) – sich die Startseite zur Gameseite verwandelt und die „displayHighscoreList“ Funktion ausgeführt wird.

Idee 2 – Spieler(namen) anlegen

(1 – „Enter Name“-Button) Zunächst sollte die Voraussetzung sein, dass ein Spielername existieren muss, um auf die Gameseite zu gelangen. Um dies zu prüfen, wurde die Variable vom Typ Boolean „nameEntered“ erstellt. Der nächste Schritt bestand darin, den Spieler bei Klick auf den ersten Button nach seinem Namen zu fragen und diesen in einer Variablen zu speichern. Hierzu wurde die Methode „askForName“ im Modul player.ts erstellt. Gibt der Spieler keine Antwort, so wird er automatisch als Default-„Mystery Player“ registriert und spielt unter diesem Namen.

export function **askForName**(playerList: **Player**[]): string {

*//ask player for name and save answer in userInput*

  const userInput = **prompt**("Please enter your name for the game.");

*// if player did not enter any character, then set current player to mysterPlayer from local Storage*

  if (userInput === null || userInput === "") {

    return playerList[0].name;

  }

*// otherwise return userInput*

  return userInput;

}

Zu einem späteren Zeitpunkt im Entwicklungsprozess musste der EventListener für den ersten Button modifiziert werden. Aufgrund der Speicherung der Spieler und ihrer Highscores im LocalStorage (s. Umsetzung Idee 3), fehlte die Überprüfung, ob der eingegebene Spielername bereits im gespeicherten SpielerArray existiert oder diesem neu hinzugefügt werden muss. Also wurde diese dem Klick-Event hinzugefügt.

Idee 3 – Highscores zeigen

Um der Highscore-Idee eine sinnvolle Bedeutung zu geben, musste zunächst ein Timer erstellt werden, der die benötigte Zeit für das jeweilige Level hochzählen sollte. Bei der Recherche nach möglichen Implementierungen konnten unterschiedliche Optionen gefunden werden, meist jedoch nur für Countdown-Timer. Durch Ausprobieren und die Kombination von verschiedenen Lösungsansätzen, konnte so eine zufriedenstellende Logik im Modul timer.ts. implementiert werden.

function **countTime**() {

  if (active) {

    ++totalSeconds;

    minutes = Math.**floor**(totalSeconds / 60);

    seconds = totalSeconds - minutes \* 60;

    if (minutes < 10) minutes = "0" + minutes;

    if (seconds < 10) seconds = "0" + seconds;

    time = `${minutes}:${seconds}`;

    timer.innerHTML = `<p class="timer">Time: ${minutes}:${seconds}</p>`;

**setTimeout**(() => {

**countTime**();

    }, 1000);

  }

}

Eine mögliche Lösung, die Idee mit den Highscores umzusetzen, wurde mit der Demonstration des LocalStorage in einer der Q&A Sessions geliefert. Nach weiterem Einlesen wurde die Idee für gut befunden und versucht umzusetzen. Die Idee war, jeden Spieler mit seinen Scores als Objekt in einem Array unter dem Key „Players“ im LocalStorage zu speichern. Zuerst wurde die Speicherung im LocalStorge in der index.ts ausprobiert, doch es wurde schnell klar, dass es am sinnvollsten ist auch hier ein neues Modul localStorage.ts anzulegen. Es wurden Funktionen implementiert, um unter anderem an den Inhalt de LocalStorages zukommen, um den Default-Player zu speichern, den LocalStorage nach einer Änderung zu aktualisieren oder nach bereits existierenden Spielern oder Scores zu suchen und diese ggf. ersetzen zu können.

Die Idee war, dass jeder Spieler (bzw. jeder Name) einmal im LocalStorage gespeichert wird und über den LocalStorage unter den Highscores die Scores aller gespeicherten Spieler angezeigt werden können. Bei der Implementierung kam es zu mehreren Schwierigkeiten. Zunächst wurden die Spieler-Objekte zwar gespeichert, jedoch wurden auch gleiche Namen als neue Objekte im Array angelegt. Um dies zu verhindern, wurde als erstes eine Variable playerList in der index.ts erstellt, in der zu Beginn des Spiels das Array des LocalStorage beim key „players“ gespeichert werden (sollte es noch kein Array geben, so wird der Mystery Player als Default-Player hinzugefügt). In einem zweiten Schritt wurde die Funktion „checkStorageforPlayer“ erstellt, um zu prüfen, ob der eingegeben Spielername bereits in der gespeicherten Liste existiert. Im weiteren Verlauf wurden dann jedoch auch die Scores nicht korrekt im HighscoreList-Array eines Spielers ersetzt, sondern für jedes Mal ein neuer Score dem Array des aktiven Spielers hinzugefügt (für den gleichen Spieler wurden somit mehrere Scores für Level 1 angezeigt). Um dieses Problem zu beheben, wurden vier weitere Funktionen im LocalStorage Modul implementiert (playerHasHighscoreStorage, getExistingHighscoreList, hasScoreForLevel und getExistingScoreFromStorage). Mithilfe dieser Funktionen konnte das Problem innerhalb der setHighscore Funtkion (player.ts) gelöst werden.

export function **setHighscore**(currentPlayer: **Player**,playerList: **Player**[],currentLevel: number,time: string): **Player**[] {

*//create variable for new Highscore List*

  let newHighscoreList: **score**[] = [{ level: 0, time: "" }];

*// get the current playerlist form local Storage*

  const storagePlayerList = **getLocalStorage**();

*// check if player has already an highscorelist in storage*

  const hasHighscoreList = **playerHasHighscoreStorage**(currentPlayer);

*// If player has no HighscoreList*

  if (hasHighscoreList === false) {

    console.**log**("No highscore at all for player", currentPlayer);

    newHighscoreList = [{ level: currentLevel, time: time }];

*// Player has already a Highscore list in local Storage*

  } else {

*//get the existing highscore list for that player*

    const existingHighscoreList = **getExistingHighscoreList**(currentPlayer,

      storagePlayerList);

*//Check if player has already a score for the current level*

    const scoreForLevel = **hasScoreForLevel**(currentPlayer, currentLevel);

*//if player has already a score for that level*

    if (scoreForLevel && existingHighscoreList) {

*// get the old score*

      const oldScore = **getExistingScoreFromStorage**(existingHighscoreList,

        currentLevel);

      newHighscoreList = existingHighscoreList.**map**((score) => {

        if (score.level === oldScore?.level) {

*// update the old score with the new time for that level and save it to newHighscoreList*

          return { ...score, time: time };

        }

        return score;

      });

*// player has no highscore for this level yet*

    } else if (!scoreForLevel && existingHighscoreList) {

*// expand existing highscore list with new score for this level*

      const newScore = {level: currentLevel, time: time };

      existingHighscoreList?.**push**(newScore);

      newHighscoreList = existingHighscoreList;

    }

  }

  const updatedPlayerList = **updatePlayerHighscores**(currentPlayer,

playerList,newHighscoreList);

  return updatedPlayerList;

}

Während der Behebung ist jedoch schnell aufgefallen, dass jeder Spieler die gleiche Zeit für ein Level benötigt (immer 37 Sekunden für Level eins). Um die Highscores etwas realer abbilden zu können, entstand die Idee, die Startposition des Balls zufällig zu gestalten, um dem Ball jedes Mal einen anderen Verlauf zu geben. Hierfür wurde das ball-setup Modul um die Funktion „randomStartPos“ ergänzt. Mithilfe von Math.floor und Math.random wird so eine zufällige Zahl zwischen 850 und 100 generiert, die bestimmt, wo der Ball im Canvas beginnt, sobald ein neues Spiel/Level gestartet wird. Durch diese Lösung entsteht jedes Mal eine unterschiedliche Zeitangabe für den Spieler (dass dies nur Zufall ist und der Spieler, die Zeit eigentlich nicht durch sein Können beeinflussen kann, muss der Spieler ja nicht wissen).

Idee 4 – Hilfe einblenden

(3 – „Help“-Button) – Muss noch implementiert werden.

Idee 5 – Verschiedene Levels

Um die Idee der verschiedenen Levels umzusetzen, wurde zunächst in der game.ts Datei eine neue Variable „let level“ erstellt und mit 1 initialisiert, die angibt, in welchem Level der Spieler sich befindet. Diese wird dynamisch hochgezählt, sobald ein Level mit „Game Won“ abgeschlossen wurde. Anschließend wurde das aus dem Tutorial übernommene Level-Array aus der setup.ts Datei genommen und in ein neues Modul level-list.ts integriert. So bleiben die Setup Dateien übersichtlich und Entwickler finden schneller die richtige Datei, falls ein Level angepasst oder weitere Levels ergänzt werden müssen.

In der neuen Datei wurde für jedes der gewünschten fünf Levels ein eigenes Array mit unterschiedlichen Steinen erstellt. Die Steine werden im Array durch die Zahlen 1 bis 5 dargestellt, die wiederrum für die „BrickDifficulty“ stehen (wie oft ein Brick mit dem Ball getroffen werden muss bis, dass er verschwindet). Die fünf Level werden exportiert und in das Modul brick-array.ts importiert, in dem sich die Methode für die Erstellung des BrickArrays befindet, mit dem später die einzelnen Bricks des jeweiligen Levels in das Canvas gezeichnet werden können. Des Weiteren musste die „createBrickArray“ Methode modifiziert werden. Um der „createBrickArray“ Methode dynamisch das Array des jeweiligen Levels zum Iterieren geben zu können, wird ein weiteres Array mit key-value-Paaren aller Levels erstellt (const level). Der Methode muss nun das „currentLevel“ vom Typ number als Parameter übergeben werden, damit sie sich über die key-value-Paare das entsprechende LevelArray ziehen kann.

*//create Array of Levels*

const levels: {[key: number]: number[]} = {

    1: LevelOne,

    2: LevelTwo,

    3: LevelThree,

    4: LevelFour,

    5: LevelFive

}

*// method to set the bricks according to current level*

export function **createBrickArray**(currentLevel: number): **Brick**[] {

    return levels[currentLevel].**reduce**((accumulator, element, i) => {//code continues

Anschließend wurde die Funktion “setGameWin” modifiziert, indem die level-Variable nach einem Sieg um eins erhöht wird. Zudem war die Idee, dass sich der Button-Text ebenfalls dynamisch von „Play“ zu „Next Level“ ändern sollte.

Mit der später entstandenen Idee, die fünf Level zusätzlich durch weitere Anpassungen zu unterscheiden, musste die „setGameWin“ Methode erneut angepasst werden. Die ursprüngliche Idee war, im gleichen Zug mit der Erhöhung des Levels auch den BallSpeed für jedes Level um 1 oder 2 zu erhöhen. Bei der Umsetzung kam es dann allerdings zu Problemen. Es wurde durch Testen festgestellt, wenn der BallSpeed erhöht wurde, die Collision für Paddle und Ball nicht mehr gegriffen hat und der Ball somit nicht vom Paddle wieder nach oben befördert werden konnte. Leider konnte nach einiger Recherche keine Lösung für dieses Problem gefunden werden. Es wurde mit verschiedenen Werten für BallSpeed und PaddleSpeed getestet und festgestellt, dass nur bestimmte Kombinationen von BallSpeed und PaddleSpeed zum gewünschten Ergebnis führten (5&10 / 6&12 / 10&20). Aufgrund von Zeitmangel wurde sich dazu entschieden mit diesen drei Varianten zu arbeiten und Speed von Ball und Paddle nur in ausgewählten Leveln zu steigern (ab Level 2 auf 6&12 und ab Level 4 auf 10&20). Resultierend von diesen Schwierigkeiten und dem Wunsch, die Spieldynamik so hoch wie möglich zu gestalten, wurde analog die Logik implementiert, die Breite des Paddles in bestimmten Leveln zu ändern.

Um die Änderungen spezifisch für Level 2 bis 5 im Code auszuüben, wurde also die „setGameWin“ Methode ein zweites Mal modifiziert und die zuvor beschriebenen Ideen und Logiken in einem Switch-Case-Statement verpackt.

## 4.4 Struktur des Codes

Bei der Strukturierung des Codes wurde versucht Logiken zum gleichen „Thema“ (z.B. Spieler, LocalStorage oder Canvas betreffend) in gemeinsamen Modulen zu implementieren. Außerdem wurde versucht keine Dateien mit mehreren hunderten Zeilen code zu erzeugen, sondern lieber kleinere Funktionen und Dateien zu erstellen. Der Ordner „src“ ist wie folgt aufgebaut:

* Images 🡪 Hier befinden sich alle Bilder (Quelle freeCodeCamp Tutorial)
* Modules 🡪 Hier befinden sich alle Module
  + Canvas-elements Ordner 🡪 hier befinden sich alle Module, die das Canvas betreffen (Bricks, Paddle, Ball, CanvasView, setups etc.)
  + Helpers Ordner 🡪 Hier befinden sich die Module zur DOM-Manipulation, dem Setup der Gamepage und helpers für den Player.
  + Game, Level-List, LocalStorage, Player und Timer
* Styles 🡪 Hier befinden sich die beiden CSS-Stylesheets
* Index.ts

# **Reflexion**