16. APRIL 2021

Thema der Projektarbeit

Erweiterung des Kids Modes um eine Skin Freischaltung für die Plattform MRCS.

Prüfungsbewerber

Liana Unseld
Oberer Stadtberg 10, 73492 Dalkingen

Identnummer: 492421

E-Mail: Liana.Unseld@Plan-B-GmbH.com
Telefon: +49 151 26056490

Ausbildungsbetrieb:

PlanB. GmbH
Kocherstraße 15, 73460 Hüttlingen

Projektbetreuer:
Felix Rohmeier
E-Mail: Felix.Rohmeier@Plan-B-GmbH.com
Tel.: +49 151 52602815

Dokumentation Projektarbeit

Inhaltsverzeichnis

Projektumfeld

Die PlanB. GmbH ist ein auf IT-Dienstleistungen spezialisiertes Unternehmen. Hierbei bietet sie ihren Kunden hochspezialisierte Softwareanwendungen und IT-Infrastrukturlösungen basierend auf Microsoft Technologien. Das Unternehmen agiert hauptsächlich innerhalb des deutschen Binnenmarkts. Kunden der PlanB. GmbH sind in den meisten Fällen Konzerne und Unternehmen des gehobenen Mittelstands. Der Hauptsitz des Unternehmens befindet sich in Hüttlingen in der Kocherstraße 15. Bei den drei Standorten in Hüttlingen, Frankfurt sind 130 Mitarbeiter angestellt.

Abbildung 1: PlanB

Eine von der PlanB. entwickelte Plattform ist MRCS (Mixed Reality Cleaning Solution, folgend nur noch MRCS genannt), welche unter anderem auch eine App beinhaltet und mehrere Kunden bedient. Es werden Reinigungsprozesse im Business und Consumer Umfeld digitalisiert, in denen unterschiedliche Szenarien abgebildet werden u.a. die Häusliche-Reinigung. Dies funktioniert durch das Scannen eines Raumes über die Smartphone Kamera. In diesem Raum kann der Kunde dann Informationspunkte setzen und speichern. Die Punkte werden in der Cloud gespeichert und können so nach einem erneuten Scan wiedergefunden, erledigt, gelöscht oder bearbeitet werden.

Abbildung 2: Erstelle kinderleicht Aufgabenpläne und markiere Aufgaben.

Projektziel

Bei der Erweiterung geht es darum, dass der bereits enthaltene Kids Mode vergrößert wird, indem der Benutzer dort neue Skins freischalten und diese dann verwenden kann. Dafür muss der Benutzer jedoch erst Punkte sammeln, welche er durch das Erledigen von Aufgaben bekommt.

Projektbegründung

Aktuell gibt es einen Normalen Skin und einen Kids Mode Skin. Um die Auswahl zu vergrößern soll es die Möglichkeit geben im Kids Mode neue Skins durch Benutzerpunkte freizuschalten. Ziel ist es das Putzerlebnis für den Benutzer spannender zu gestalten. Durch eine größere Auswahl steigt der Reiz neue Skins freizuschalten.

Projektschnittstellen

Die Erweiterung hat über einen Datenservice eine Verbindung zu einer Azure CosmosDB, und kann so die Daten abrufen. Des Weiteren ist die Erweiterung ein Teil der MRCS Plattform. Die Entwicklungsumgebung ist Visual Studio 2019 und Unity. Unity ist eine 3D-Engine für Games, Mobile Applications und vieles mehr.

Die größte fachliche Schnittstelle bildet Felix Rohmeier, der Product Owner bei MRCS ist. Weitere Schnittstellen sind Hermann Fröhlich, Mahmoud Alhayek und Melanie Achmetow, die als Entwickler bei MRCS tätig sind.

Projektabgrenzung

Die Erweiterung des Kids Modes kann wie im Pflichtenheft beschrieben umgesetzt werden.

Projektplanung

Projektphasen und Ablaufplan

Die Erweiterung des Kids Modes um eine Skin-Freischaltung soll innerhalb von 9 Tagen bei einem Aufwand von 8 Stunden pro Arbeitstag umgesetzt werden.
Grobe Zeitschätzung:

Tätigkeit	Unter aufgaben	Aufwand in Stunden
Vorbereitungen	Ist-/ Soll-Analyse	3
	Technische Analyse und Lösungsfindung	1
	Aussuchen und kaufen eines neuen Skins	1
	Projektplan	3

Implementierung	Frontend	14
	Backend	16
Testen	Testen der Erweiterung	5
	Testen des neuen Skins	5
Abnahme + Dokumentation		22
Summe		70

Tabelle 1: Projektphasen

Ressourcenplanung

Für das Umsetzen der Erweiterung sind folgende Ressourcen nötig:

Personalkosten:

Name, Vorname	Position	Stundenlohn	Zeitaufwand in Stunden	Kosten
Unseld, Liana	Entwickler	47.50€	70	3.325€
Rohmeier, Felix	Projektleiter	95€	3	285€
		Summe		3.610€

Tabelle 2: Personalplanung

Arbeitsmittel:

Arbeitsmittel	Anzahl
Laptop	1
Azure-Account	1
Unity	1
Visual Studio 2019	1
Arbeitsplatz	1
Internetanschluss	1

Tabelle 3: Arbeitsmittelplanung

Ist-/ Soll-Analyse

Ist:

MRCS besitzt für das Finden der Reinigungspunkte in einem Reinigungsbereich einen Kids Mode, der den Reinigungspunkt als ein Drachenei in Mixed Reality darstellt. Diesen kann der Benutzer über die Handykamera sehen. Vor der Umsetzung der Skin Erweiterung hatte der Benutzer nur den Standard Skin, unseren MRCS Drachen. Es konnten Dracheneier gesetzt werden und nach abschließen der Aufgabe konnte der Benutzer den Drachen schlüpfen lassen.

Soll:

Es soll möglich sein unter „Gewinne“ im MRCS-App Menü ein Icon erscheinen zu lassen und dieses anklicken zu können (wichtig ist, dass der Benutzer im Kids Mode ist). Klickt der Nutzer auf dieses Icon kommt man zur Erweiterung. Folgend kann nun mit den gesammelten Punkten ein neuer Skin freigeschaltet werden. Für das Freischalten muss der Nutzer Punkte besitzen, welche er über das Erledigen von Aufgaben sammeln kann. Möchte der Benutzer einen Skin freischalten und bestätigt die Freischaltung, werden die Informationen des Skins sowie die UserID und die CleaningSpaceID in der Datenbank gespeichert. Ein freigeschalteter

Skin wird in der Erweiterung mit einem anderen Icon (Geöffnetes Schloss) dargestellt. Wird Der neue Skin aktiviert oder deaktiviert, wird der neue Aktivitäts-Status in der Datenbank geupdatet.

Pflichtenheft

Musskriterien:

Für die Plattform MRCS soll der vorhandene Kids Mode erweitert werden, so dass es die Möglichkeit gibt neue Skins freizuschalten und diese zu verwenden. Die freigeschalteten Skins des Benutzers werden in einem neuen Datenbank-Container gespeichert. Die Daten der freischaltbaren Skins befinden sich ebenfalls in einem neuen Datenbank-Container der MRCS CosmosDB. Das Freischalten eines Skins wird über die Erweiterung ausgeführt. Der Nutzer muss für das Freischalten seine vorher gesammelten Aufgabenpunkte verwenden. Der Preis des Skins wird ihm dabei von seinem Score abgezogen. Der freigeschaltene Skin muss verwendbar sein und der Benutzer soll ihn selbst in der Erweiterung aktivieren oder deaktivieren können.

Wunschkriterien:

Die Erweiterung soll einfach gestaltet werden, so dass ein Kind diesen bedienen kann. Die Implementierung soll jeder Zeit erweiterbar sein, so dass später dort auch mehrere Skins zu Auswahl stehen können.

Prozessschritte

Tätigkeit	Arbeitsschritt	Aufwand in Stunden
Vorbereitung	Analyse der Anforderungen	1
	Erstellen Pflichtenheft und Projektplan	4
	Abstimmung und Abnahme durch Product Owner	1
Implementierung	Implementieren des Backend	
	Erstellen eines Containers für die Skins, die der Benutzer freischalten kann.	1
	Erstellen eines Containers für die Speicherung der freigeschalteten Skins	1
	Erstellen dreier Data Access Layer für den MRCS-Datenservice	4
	Hinzufügen zweier DTO's zur MRCS Shared Library	2
	Testen der Speicherung und der Data Access Layer; Get and Post Abfragen über Postman	4
	Implementieren des Frontends	
	Erstellen der Erweiterungs Scene in Unity	2
	Implementieren eines Controllers für die Scene	2
	Implementieren der Anzeige logik für Cleani	1
	Erstellen einer Info-Scene in Unity	1

	Implementieren eines Controllers für die Info Scene mit einer Übersetzung der Description des Skins	2
	Kauf und Import des Skins aus dem Unity Asset Store	1
	Erstellen des Prefabs mit Animation für den Ghosty Skin	2
	Implementierung eines Animations Controllers für den Skin	1
	Implementierung der Verwendungslogik für den Skin anstatt der Drachen	2
Qualitätssicherung	Testen der Erweiterung	5
	Testen des neuen Skins	5
Dokumentation	Verfassen der Dokumentation	11
	Erstellung des Benutzerhandbuchs	7
	Finale Abnahme durch Product Owner	3

Tabelle 4: Prozessschritte

Bereits bestehende Systeme oder Produkte:

Es soll der schon vorhandene Kids Mode erweitert werden, um MRCS selbst zu erweitern. Implementiert wird mit Unity und Visual Studio 2019 in der Programmiersprache C#. Die MRCS Datenbank muss ebenfalls um zwei neue Container für das Speichern der Informationen des Skins und der freigeschalteten Skins erweitert werden.

Rahmenbedingungen:

Das Projekt soll in 70 Stunden durchgeführt werden bei einem 8 Stunden Tag.

Projektentwicklung:

Das Projekt wird allein von Liana Unseld programmiert und umgesetzt. Dazu siehe die Prozessschritte auf Seite 6, wo der zeitliche Ablauf der einzelnen Schritte aufgelistet.

Implementierungsphase

Vorbereitung

Für die Umsetzung der Skin Erweiterung muss ein neuer Skin ausgesucht und im Assetstore von Unity von einem Asset Ersteller (Third Party) gekauft werden (siehe Auswahl des Skins). Es muss daraus ein Prefab (ein Zusammenschluss von Gameobjekten im Unity) erstellt und ein für die App passenden Animator und Skin Controller implementiert werden, so dass dieser Skin später verwendet werden kann. Ebenfalls müssen in der MRCS-CosmosDB zwei neue Container erstellt werden. Davon sollen in dem einen die für das Freischalten zur Verfügung Stehenden Informationen gespeichert werden, und in dem anderen die freigeschalteten Skins.

Auswahl des Skins

Um einen altersgerechten Skin auszuwählen habe ich eine Umfrage innerhalb der PlanB. GmbH durchgeführt. Dort wurden die Kinder von Mitarbeitern gefragt welchen Skin sie am liebsten hätten.

Es wird für dieses IHK-Projekt Vorerst nur ein Skin implementiert. Allerdings soll das Systemjederzeit erweiterbar sein.

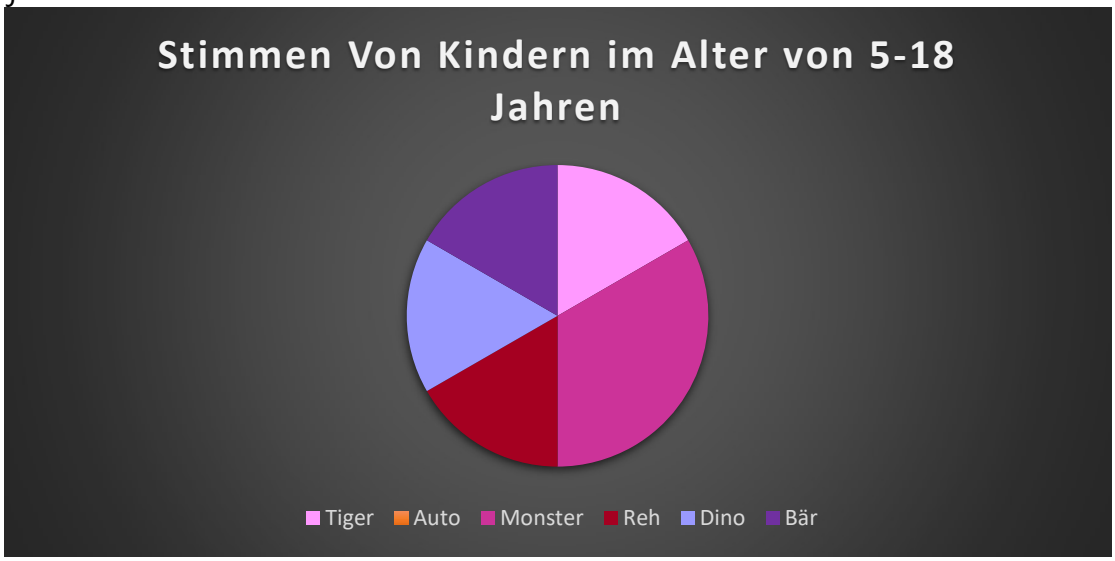


Abbildung 3: Umfrage Auswertung

Das Ergebnis der Umfrage war eindeutig und es wurde mit 3 Stimmen (von 5-18 Jahren) für die auf der Abbildung 4 zusehenden Monster gestimmt.

Abbildung 4: Monster

Projekterstellung

Die Erweiterung wird in Unity als Scene erstellt und die Funktionen in Visual Studio 2019 in der Sprache C# implementiert. Unity ist eine 3D-Engine welche es ermöglicht, Scenen mit Game-Objekten zu erstellen. Die Scenen werden in der App angezeigt. Dadurch ist es möglich ein schönes Layout zu schaffen und eine gute Funktionalität der App-Seiten umzusetzen. Unity bietet die Möglichkeit, die App aufs Handy zu installieren und über die im Editor verfügbare Konsole Debugging-Informationen zu erhalten. Für die Speicherung der Skin-Informationen und der freigeschalteten Skins werden zwei Container in der MRCS-CosmosDB erstellt. Für den neuen Skin wird ein Animator erstellt, der die 3D Figur animiert. Ebenfalls wird ein Controller implementiert, der für das Timing der richtigen Animation sorgt. Es muss daraus ein Prefab erstellt werden.

Erstellen der neuen Datenbank Container

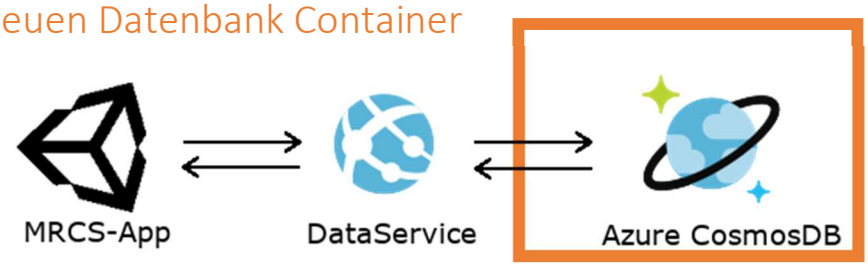


Abbildung 5: Verbindung Datenbank und App

Erstellt werden zwei neue Container, einer genannt MRCS-BoughtSkins welcher als Partitionkey die UserID enthält. Dieser wurde verwendet da ein möglichst eindeutiger Partitionkey Ressourcen schonend und am schnellsten ist. Die Eindeutigkeit in Bezug auf Queries stell kein Problem da, da die Daten nicht benutzerübergreifend sind. Der andere Container, genannt MRCS-Skins, enthält alle Daten der Skins die der Benutzer freischalten kann. Der Container hat den Partitionkey Country dieser wird bei länderspezifischen Informationen standardmäßig bei MRCS verwendet. Die Blaue Linie stellt den Partitionkey da.

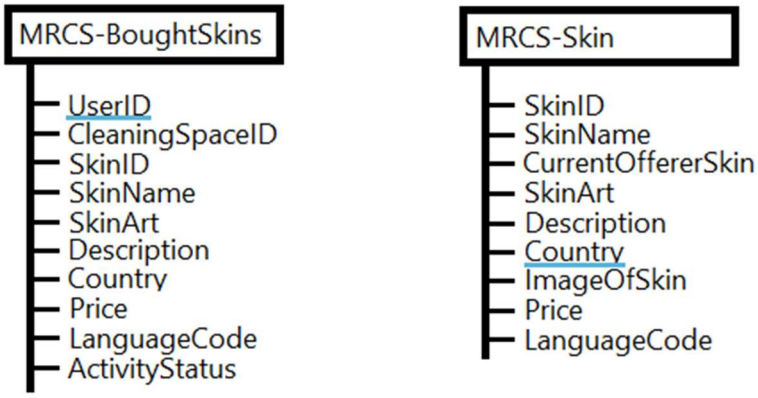


Abbildung 6: Datenbank Container

Zugriff auf die neuen Container im MRCS Dataservice

Um den Zugriff auf die neuen Container herzustellen muss eine Verbindung über den Webservice der App, genannt MRCS-DataService, erstellt werden. Dazu wurden drei neue Data Access Layer implementiert. Die Plattform MRCS kommuniziert über den Webservice mit der Datenbank. Die DAL (Data Access Layer) werden im MRCS-DataService implementiert.

Abbildung 7: Verbindung Datenbank und App

Skins.cs

Dieser DAL sendet einen http-Request mit einem Get und einer Route zum richtigen Datenbank Container. Dort werden alle mit dem passenden „CurrentOffererSkinStatus“ Skins abgefragt. Zurückgegeben wird ein JSON-Dokument mit allen passenden Skins.

```
public static class GetBoughtSkins
{
    // gives the FunctionName CPGetBoughtSkinsByUserId
    [FunctionName("CPGetBoughtSkinsByUserId")]
    // runs the Http Request with a get to get all unlocked skins from the database
    public static async Task<IActionResult> Run(
        [HttpTrigger(                                , "get", Route = "BoughtSkins/CPGetBoughtSkinsByUserId/{UserId}")] HttpRequest req,
        ILogger log,
        [CosmosDB(Const.CDB_Databasename, Const.CDB_CollcectionNameBoughtSkins, ConnectionStringSetting = Const.CDB_ConnectionStringSetting,
        SqlQuery = "select * from r where r.UserID = {UserID}",
        PreferredLocations = "%PreferredLocations%")] IEnumerable<BoughtSkinsDTO> documentsToRead)
    {
        log.LogInformation($"Detected {documentsToRead.ToString()} incoming documents");
        // returns the result of the Http Request
        return new JsonResult(documentsToRead);
    }
}
```

Abbildung 8: Skins.cs

GetBoughtSkins.cs

Dieser Data Access Layer sendet ebenfalls einen http-Request mit einem Get und einer Route zum Datenbank Container und fragt alle Skins mit der passenden „UserID“ ab. Zurückgegeben wird ein JSON-Dokument mit allen passenden Skins zur UserID.

```
public static class Skins
{
    // gives the FunctionName CPGetSkins
    [FunctionName("CPGetSkins")]
    // runs the Http Request with a get to get all skins from the database where the user can unlock
    public static async Task<IActionResult> Run(
        [HttpTrigger(                                , "get", Route = "Skins/CPGetSkins/{CurrentOffererSkinStatus}")] HttpRequest req,
        ILogger log,
        [CosmosDB(Const.CDB_Databasename, Const.CDB_CollcectionNameSkins, ConnectionStringSetting = Const.CDB_ConnectionStringSetting,
        SqlQuery = "select * from r where r.CurrentOffererSkin = {CurrentOffererSkinStatus}",
        PreferredLocations = "%PreferredLocations%")] IEnumerable<SkinsActivationDTO> documentsToRead)
    {
        log.LogInformation($"Detected {documentsToRead.ToString()} incoming documents");
        // returns the result from the Http Request
        return new JsonResult(documentsToRead);
    }

}
```

Abbildung 9: GetBoughtSkins

BoughtASkin.cs

Dieser Data Access Layer sendet einen http-Request mit einem Post und einer Route zum Datenbank Container und speichert dort den vom Benutzer freigeschalteten Skin.

```
public static class BoughtASkin
{
    // gives the Function name  BoughtASkin
    [FunctionName("BoughtASkin")]
    // runs the Http Request with a Post to save the unlocked skin
    public static async Task<IActionResult> Run(
       [HttpTrigger([redacted], "post", Route = "BoughtSkins/BoughtASkin")] HttpRequestMessage req,
        [CosmosDB(Const.CDB_Databasename, Const.CDB_CollcectionNameUser, ConnectionStringSetting = Const.CDB_ConnectionStringSetting,
        PreferredLocations = "%PreferredLocations%")] DocumentClient userClient,
        ILogger log,
       [CosmosDB(Const.CDB_Databasename, Const.CDB_CollcectionNameBoughtSkins, ConnectionStringSetting = Const.CDB_ConnectionStringSetting,
        PreferredLocations = "%PreferredLocations%")] IAsyncCollector<BoughtSkinsDTO> documentsToStore)
    {
        // a try catch if the Request failed or not
        try
        {
            var Skin = await req.Content.ReadAsAsync<BoughtSkinsDTO[]>();
            log.LogInformation($"Detected {Skin.ToString()} incoming documents");
            foreach (var oneCp in Skin)
            {
                await documentsToStore.AddAsync(oneCp);
            }

            return new OkResult();
        }
        catch (System.Exception e)
        {
            return new ObjectResult(e);
        }

    }
}
```

Abbildung 11: BoughtASkin.cs

Hinzufügen der passenden Data Transfer Objects

In MRCS nutzen wir Data Transfer Objects. Für die Erweiterung wurden zwei DTOs (Data Transfer Objects) implementiert, eines für das Speichern der Daten des gekauften Skins, sowie ein anderes für das Abrufen der Daten der freischaltbaren Skins. In MRCS wird eine Shared Library verwendet welche die DTOs enthält. Die DTOs enthalten dieselben Daten wie die aus den zwei neuen Datenbank Containern (siehe Abbildung 6).

```
[Serializable]
public class SkinsActivationDTO
{
    [JsonProperty(PropertyName = "id")]
    /// <summary>
    /// is for the Skin id
    /// </summary>
    public string SkinId;

    /// <summary>
    /// is for the name of the Skin
    /// </summary>
    public string SkinName;

    /// <summary>
    /// is the art of the SKin
    /// </summary>
    public string SkinArt;

    /// <summary>
    /// is for the description of the cleaning point
    /// </summary>
    public string Description;

    /// <summary>
    /// is for the country
    /// </summary>
    public string Country;

    /// <summary>
    /// is for the list to have a Image
    /// </summary>
    public string ImageOfSkin;

    /// <summary>
    /// is for the Price
    /// </summary>
    public int Price;

    /// <summary>
    /// is the language code
    /// </summary>
    public string LanguageCode;

}
```

Abbildung 13: SkinActivationDTO CodeSniped

SkinActivationDTO

Dieser DTO ist für das Abrufen der Daten, die ein Skin besitzt, den der Benutzer freischalten kann; wie die SkinID, SkinName, SkinArt oder die Description für die Beschreibung des Skins. Ebenfalls gibt es noch Country, ImageOfSkin (für den Pfad des Skin Bildes) sowie den Preis des Skins, um ihn freizuschalten. Diese Daten sind auch wichtig um den Skin abzuspeichern, wenn er freigeschaltet wird.

BoughtSkinsDTO

Für das Abspeichern wird dieser DTO benötigt, der dieselben Daten enthält wie die SkinActivation; bis auf das Image. Dafür besitzt er noch weitere Daten, die mit abgespeichert werden sollen, wie die UserID des aktiven Benutzers, CleaninSpaceID des aktuellen Reinigungsbereiches und den ActivityStatus; also ob der Skin aktiviert oder deaktiviert ist.

```
public class BoughtSkinsDTO
{
    [JsonProperty(PropertyName = "id")]
    /// <summary>
    /// is for the ID from the object in the database
    /// </summary>
    public string Id;
    /// <summary>
    /// is for the active UserID
    /// </summary>
    public string UserID;
    /// <summary>
    /// is for the CleaningspaceID from the Active User
    /// </summary>
    public string CleaningSpaceID;
    /// <summary>
    /// is for the Skin id
    /// </summary>
    public string SkinId;
    /// <summary>
    /// is for the name of the Skin
    /// </summary>
    public string SkinName;
    /// <summary>
    /// is the art of the SKin
    /// </summary>
    public string SkinArt;
    /// <summary>
    /// is for the description of the cleaning point
    /// </summary>
    public string Description;
    /// <summary>
    /// is for the country
    /// </summary>
    public string Country;
    /// <summary>
    /// is for the Price
    /// </summary>
    public int Price;
    /// <summary>
    /// is the language code
    /// </summary>
    public string LanguageCode;
    /// <summary>
    /// is the ActivityStus of the Skin
    /// </summary>
    public bool ActivityStatus;

}
```

Abbildung 12: BoughtSkinsDTO

Data Access Layer im MRCS Projekt

Im Unity Projektteil von MRCS wurden ebenfalls zwei DAL`s (Data Access Layer) implementiert.

Abbildung 15: Verbindung Datenbank und App

SkinsDAL

Fragt über den Webservice alle Skins ab, die den passenden CurrentOffererSkinStatus haben und gibt ein Array zurück.

```
public SkinsActivationDTO[] GetByCurrentOffererStatus(string CurrentOffererSkinStatus)
{
    // makes the Route for the DAL(DataService) with the "Skins/CPGetSkins/" and the CurrentOffererSkinStatus
    string endPoint = "Skins/CPGetSkins/" + CurrentOffererSkinStatus;
    // Sends the route to the database with the help of the DAL of the web service and saves the result in returnData
    var returnData = this.SendHttpRequestAsnyc(HttpMethod.Get, endPoint).Result;
    // if the request does not work or it gives an error, null is returned
    if (returnData == null)
    {
        return null;
    }
    // Deserializes the Jason object returned by the request to a SkinsActivationDTO array
    var returnObj = JsonConvert.DeserializeObject<SkinsActivationDTO[]>(returnData);
    // returns the result of the request as a SkinsActivationDTO array
    return returnObj;
}
```

Abbildung 16: SkinsDAL

BoughtSkinsDAL

GetByUserID: Fragt alle freigeschalteten Skins ab, die dem aktuellen Benutzer gehören. Dafür wird die UserID mit der passenden Abfrageroute gesendet und zurückgegeben wird ein BoughtSkinsDTO Array mit allen Skins des Benutzers.

```
public BoughtSkinsDTO[] GetByUserID(string UserID)
{
    // creates the route with the help of the passed string for the Http request
    string endPoint = "BoughtSkins/CPGetBoughtSkinsByUserId/" + UserID;
    // Sends the route to the database with the help of the DAL of the web service and saves the result in returnData
    var returnData = this.SendHttpRequestAsnyc(HttpMethod.Get, endPoint).Result;
    // if the request does not work or it gives an error, null is returned
    if (returnData == null)
    {
        return null;
    }
    // Deserializes the Jason object returned by the request to a BoughtSkinsDTO array
    var returnObj = JsonConvert.DeserializeObject<BoughtSkinsDTO[]>(returnData);
    return returnObj;
}
```

Abbildung 17: BoughtSkinsDAL

UpdateBoughtSkins:

Wird der Skin aktiviert oder deaktiviert, so muss der freigeschaltete Skin upgedatet werden. Bei dieser Methode wird der freigeschaltete Skin übergeben, sowie der neue Aktivitätsstatus. Die Werte des freigeschalteten Skins und der neue Status werden am Ende in der Datenbank angepasst.

```
/// <summary>
/// is to update a BoughtSkin from a User
/// </summary>
/// <param name="bought"></param>
/// <param name="Status"></param>
public void UpdateBoughtSkins(BoughtSkinsDTO bought,bool State)
{
    // creates a new object of data type BoughtSkinsDTO
    BoughtSkinsDTO isBought = new BoughtSkinsDTO();
    // initializes the data of the transferred object of the data type BoughtSkinsDTO into the previously created object
    isBought.Price = bought.Price;
    isBought.SkinArt = bought.SkinArt;
    isBought.SkinId = bought.SkinId;
    isBought.SkinName = bought.SkinName;
    isBought.UserID = BusinessObjects.User.ActiveUser.Id;
    isBought.CleaningSpaceID = BusinessObjects.CleaningSpace.ActiveCleaningSpace.Id;
    // initializes the activity status with the transferred value
    isBought.ActivityStatus = State;
    isBought.Description = bought.Description;
    isBought.Id = bought.Id;
    // creates the route for the Http request
    string endPoint = "BoughtSkins/BoughtASkin";
    // saves the BoughtSkinsDTO object isBought to an new BoughtSkinsDTO array
    var cpArray = new BoughtSkinsDTO[] { isBought };
    // Convert the Array to a JSON object
    string body = JsonConvert.SerializeObject(cpArray);
    // is a debug log to test if the body was empty or not in the Unity Editor Debug Console
    UnityEngine.Debug.Log("CP Upload body: " + body);
    // Sends the route to the database with the help of the DAL of the web service and saves the result in returnData
    var returnData = this.SendHttpRequestAsnyc(HttpMethod.Post, endPoint, body).Result;
    // is a debug log to test if the Result was empty or not in the Unity Editor Debug Console
    UnityEngine.Debug.Log("CP Upload Data: " + returnData);
    // if the result is null sends a Exeption
    if (returnData == null)
    {
        throw new Exception("Can't create BoughtSkinsDTO: " + returnData);
    }
}
```

Abbildung 18: UpdateBoughtSkins

BoughtSkinsSaveCreate:

Schaltet der Benutzer einen Skin frei, so muss dieser in der Datenbank gespeichert werden. Die Methode sorgt dafür, dass die UserID und die CleaningSpaceID (Aktuelle PutzbereichsID) mit den Skin-Daten gespeichert wird.

```
public void BoughtSkinsSaveCreate(SkinsActivationDTO boughtSkins,bool Status)
{
    // creates a new object of data type BoughtSkinsDTO
    BoughtSkinsDTO bought = new BoughtSkinsDTO();
    bought.Price = boughtSkins.Price;
    bought.SkinArt = boughtSkins.SkinArt;
    bought.SkinId = boughtSkins.SkinId;
    bought.SkinName = boughtSkins.SkinName;
    bought.UserID = BusinessObjects.User.ActiveUser.Id;
    bought.CleaningSpaceID = BusinessObjects.CleaningSpace.ActiveCleaningSpace.Id;
    bought.ActivityStatus = Status;
    bought.Description = boughtSkins.Description;
    // creates a ID for the object ID
    bought.Id = Guid.NewGuid().ToString();
    // if the description of the object is zero, the description is initialized with the SkinName
    if (boughtSkins.Description == null)
    {
        bought.Description = boughtSkins.SkinName;
    }
    // if Country is zero or empty, the Country value of the current user is initialized in country
    if (string.IsNullOrEmpty(bought.Country))
    {
        bought.Country = BusinessObjects.User.ActiveUser.Country;
    }
    // creates the route for the Http request
    string endPoint = "BoughtSkins/BoughtASkin";
    // saves the BoughtSkinsDTO object isBought to an new BoughtSkinsDTO array
    var cpArray = new BoughtSkinsDTO[] { bought };
    // Convert the Array to a JSON object
    string body = JsonConvert.SerializeObject(cpArray);
    // is a debug log to test if the body was empty or not in the Unity Editor Debug Console
    UnityEngine.Debug.Log("CP Upload body: " + body);
    // Sends the route to the database with the help of the DAL of the web service and saves the result in returnData
    var returnData = this.SendHttpRequestAsnyc(HttpMethod.Post, endPoint, body).Result;
    // is a debug log to test if the Result was empty or not in the Unity Editor Debug Console
    UnityEngine.Debug.Log("CP Upload Data: " + returnData);
    // if the result is null sends a Exeption
    if (returnData == null)
    {
        throw new Exception("Can't create BoughtSkinsDTO: " + returnData);
    }
}
```

Abbildung 20: BoughtSkinsSaveCreate

Implementation Frontend

Beim Frontend wurden Unity-Scenen mit passenden Controllern implementiert. Das Erstellen der Scenen passiert im Unity-Editor, das Implementieren der Skripte in Visual Studio 2019.

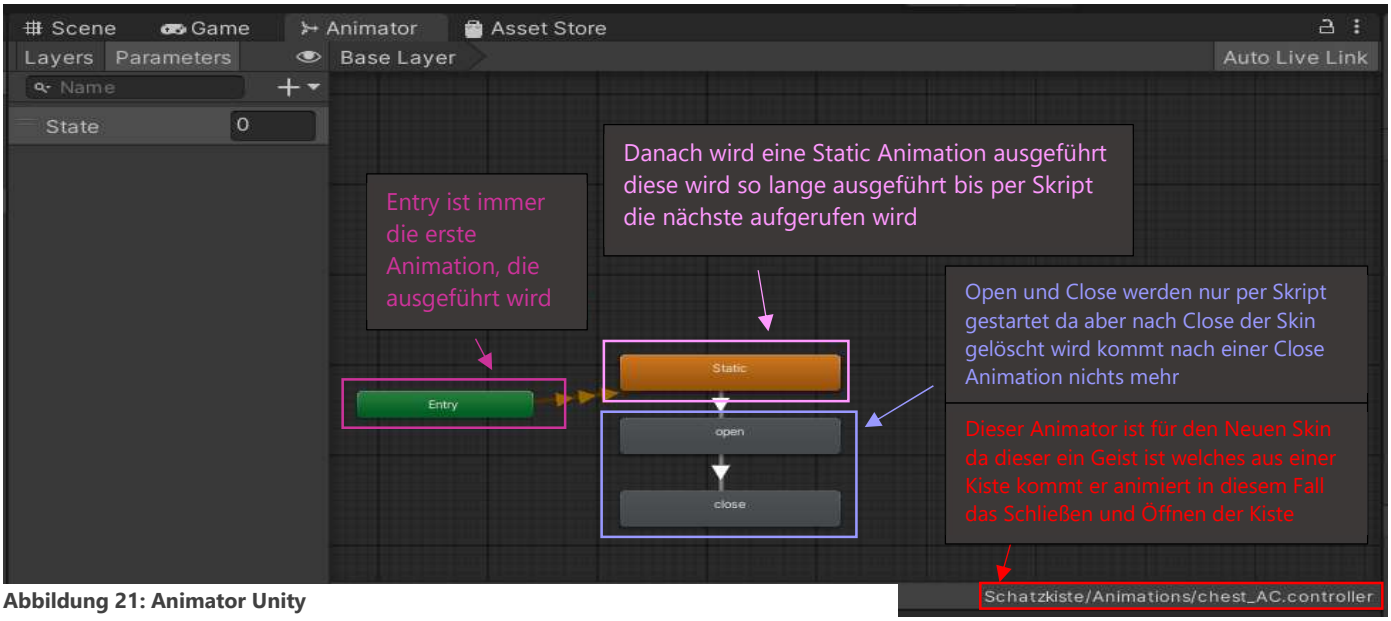


Abbildung 21: Animator Unity

Als Beispiel wie eine Unity Scene im Editor implementiert wird, hier beispielhaft zwei Screens. Das ist der Unity-Animator zum Steuern der Animationen.
Unity.Editor mit der Erweiterungs-Scene:

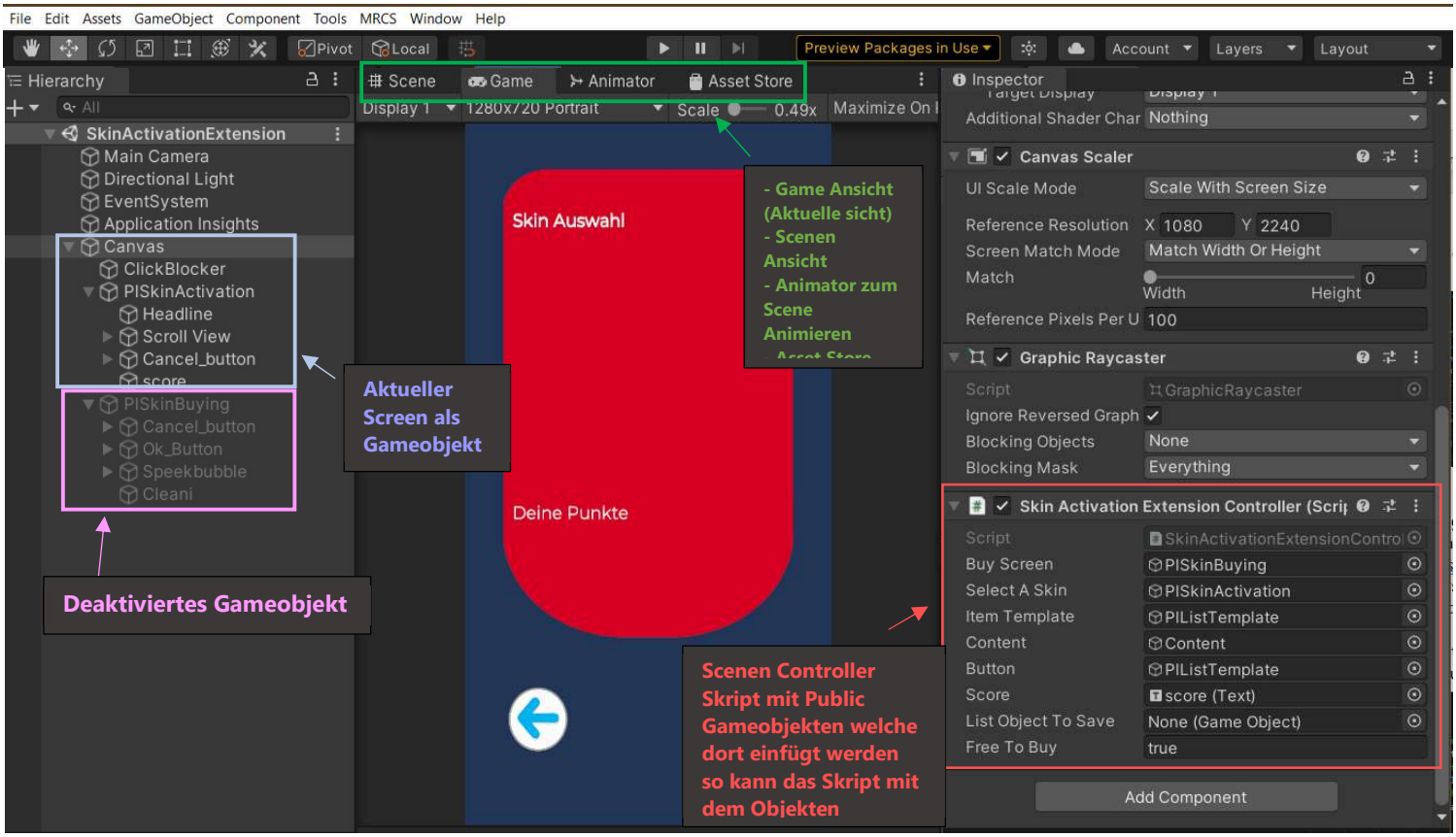


Abbildung 22: Unity Editor

Controller der Erweiterungs-Scene

Ganz oben im Controller werden die Gameobjekte der Scene initialisiert, so dass diese in den Methoden später angepasst; also aktiv oder inaktiv gesetzt werden können. Ebenfalls werden zwei Listen erstellt, eine für die Skins, die der Benutzer freischalten kann, sowie eine weitere für die vom Benutzer freigeschalteten Skins. Der Controller wird der Scene im Unity-Editor hinzugefügt und dort werden die Public Gameobjekte gesetzt. So kann das Skript mit den Gameobjekten arbeiten.

```
public class SkinActivationExtensionController : MonoBehaviour
{
    // initializes unity game objects
    public GameObject BuyScreen;
    public GameObject SelectASkin;
    public GameObject ItemTemplate;
    public GameObject Content;
    public GameObject Button;
    public Text Score;
    public GameObject ListObjectToSave;
    public string FreeToBuy = "true";
    // List with Skins who the User can unlock
    private List<SkinsActivationDTO> activeSkins = new List<SkinsActivationDTO>();
    // List with unlocked Skins
    private List<BoughtSkinsDTO> BoughtSkinsList = new List<BoughtSkinsDTO>();
    // starts when the scene is called
```

Abbildung 23: Controller der Erweiterungs Scene

Beim Starten der Erweiterung holt die „Start"-methode als Erstes die Skins, die der Benutzer freischalten kann, aus der Datenbank. Danach wird der aktive Cleaningspace über die AktiveSpaceID und die Skins, die von dem Benutzer bereits freigeschaltet wurden, über die UserID des aktiven Benutzers geholt. Folgend sieht man eine If-Else-Abfrage, welche die Punkte des Benutzers im aktuellen Cleaningspace mit passender Übersetzung zur Nutzersprache des Handys anzeigt. Dann wird der Score.Text gesetzt, den man in der Scene sieht. Die Foreach-Schleife schreibt alle Skin-Items aus dem Array und fügt diese zur Liste der verfügbaren Skins hinzu.

```
// starts when the scene is called
void Start()
{
    // fetches all skins from the database
    var SkinsToBuy = SkinsDAL.Instance.GetByCurrentOffererStatus("true");
    // fetches the Active CleaningSpace of the user from the database
    var cleaningSpace = CleaningSpaceDAL.Instance.GetById(CleaningSpace.ActiveCleaningSpace.Id);
    // fetches the Skins who Unlocked with the UserID from the database
    var BoughtSkins = BoughtSkinsDAL.Instance.GetByUserID(User.ActiveUser.Id);
    //is for the Points Score of the User
    int highscore = 0;
    // if the Score not null
    if (cleaningSpace.HighScore.Count != 0 && cleaningSpace.HighScore.ContainsKey(User.ActiveUser.Id))
    {
        // translates the text into the mobile phone user language and initializes the result of the translation into the text of the unity scene
        Score.text = LocalizationManager.GetTranslation("Deine Punkte") + ": " + cleaningSpace.HighScore[User.ActiveUser.Id].ToString();
        highscore = cleaningSpace.HighScore[User.ActiveUser.Id];
    }
    // if the Score is null
    else
    {
        //translates the text into the mobile phone user language and initializes the result of the translation into the text of the unity scene
        Score.text = LocalizationManager.GetTranslation("Deine Punkte") + ": " + "0";
    }
    // goes through the skins from the database and saves them in the list
    foreach (var item in SkinsToBuy)
    {
        activeSkins.Add(item);
    }
```

Abbildung 25: Controller der Erweiterungs Scene

In einer weiteren Foreach-Schleife werden die Skins gelesen, welche zum Freischalten zur Verfügung stehen. Der obere Teil, der Copy genannt wurde, ist um den Scrollview in der Scene zu füllen. Für jeden Skin der Liste wird ein neues Gameobjekt in dem Scrollview erstellt. Der Name des Gameobjektes ist die ID des Skins, so dass später beim Setzen des Skins nur das Game-Objekt übergeben werden muss und man in der Liste schauen kann, welches nun gesetzt werden soll. Es werden die einzelnen Buttons gefunden und initialisiert. Setzt man ein Game-Objekt auf false wird es nicht mehr angezeigt. Sollte die Liste für die freigeschalteten Skins leer sein wird der Inhalt der if-Abfrage ausgeführt.

```
foreach (var element in SkinsToBuy)
{
    // if the element of the SkinsToBuy is not null
    if (element != null)
    {
        // creates a unity Scrollview object
        var copy = Instantiate(ItemTemplate);
        copy.SetActive(true);
        // gives the Object the Name of the SkinID
        copy.name = element.SkinId;
        copy.transform.SetParent(Content.transform, false);
        // initializes the title of the scroll view element with the SkinName
        copy.GetComponentInChildren<Text>().text = element.SkinName;
        // gets all Buttons, Icons, Text, and more of the new Object
        var allComponents = copy.GetComponentsInChildren(typeof(Text));
        // goes through all objects of the new Scrollview object and outputs the name in the Unity Console
        foreach (var oneCompChild in allComponents)
        {
            Debug.Log(oneCompChild.name);
        }
        // Finds the Text Object with the name Points and initializes it in oneText
        var oneText = (Text)allComponents.FirstOrDefault(c => c.name == "Points");
        oneText.text = element.Price.ToString();
        Debug.Log("Text ok");
        // Finds all buttons of the new Scrollview object an initializes it in allButtons
        var allButtons = copy.GetComponentsInChildren(typeof(Button));
        // Finds all Images of the new Scrollview object an initializes it in allButtons
        var Images = copy.GetComponentsInChildren(typeof(Image));
        // goes through all Buttons of the allButtons and outputs the name in the Unity Console
        foreach (var oneCompChild in allButtons)
        {
            Debug.Log(oneCompChild.name);
        }
        // If the User have no unlocked Skins
        if (BoughtSkins.Length == 0)
        {
            // sets the buttons inaktiv or aktiv
            var oneLockedButton = (Button)allButtons.FirstOrDefault(c => c.name == "LockedIcon");
            var oneUnlockedButton = (Button)allButtons.FirstOrDefault(c => c.name == "Unlocked");
            var oneOkButton = (Button)allButtons.FirstOrDefault(c => c.name == "OkButton");
            oneLockedButton.gameObject.SetActive(true);
            oneUnlockedButton.gameObject.SetActive(false);
            oneOkButton.gameObject.SetActive(false);
            // sets the Image of the Skin
            var ImageofSkin = (Image)Images.FirstOrDefault(c => c.name == "Image");
            ImageofSkin.sprite = Resources.Load<Sprite>(element.ImageOfSkin);
        }
```

Abbildung 26: Controller der Erweiterungs Scene

Unterhalb der If-Abfrage geht es dann mit einer Foreach-Schleife für die Skins, und mit einer weiteren Foreach-Schleife für die freigeschalteten Skins weiter. Es wird als Erstes das aktuelle Element der BoughtSkinsDTO zur Liste hinzugefügt. Danach werden die Buttons gesetzt und überprüft, ob der Skin gesetzt ist oder nicht, und demnach die Buttons aktiviert oder deaktiviert. Ist der aktuelle Skin nicht in der freigeschalteten Skin-Liste oder die Liste ist leer wird die untere If-Abfrage ausgeführt dafür ist auch der Boolean-Wert wichtig.

```
foreach (var Bought in BoughtSkins)
{
    // Add the Skin To the BoughtSKins List
    BoughtSkinsList.Add(Bought);
    // is for the test when the User have Unlocked an locked skins
    bool isFound = false;
    // initializes all Buttons
    var oneLockedButton = (Button)allButtons.FirstOrDefault(c => c.name == "LockedIcon");
    var oneUnlockedButton = (Button)allButtons.FirstOrDefault(c => c.name == "Unlocked");
    var oneOkButton = (Button)allButtons.FirstOrDefault(c => c.name == "OkButton");
    // initializes the Image of the Skin
    var ImageofSkin = (Image)Images.FirstOrDefault(c => c.name == "Image");
    ImageofSkin.sprite = Resources.Load<Sprite>(element.ImageOfSkin);
    //goes through the Skins from the database
    foreach (var Skins in SkinsToBuy)
    {
        // if the skinID is not null an the skinID of the activated skin matches that of the current skin element
        if (Bought.SkinId != null && Bought.SkinId == Skins.SkinId)
        {
            // sets the right buttons active and the false inactive
            oneLockedButton.gameObject.SetActive(false);
            oneUnlockedButton.gameObject.SetActive(true);
            // sets the OK button if the ActivityStatus false on inactive
            if (Bought.ActivityStatus == false)
            {
                oneOkButton.gameObject.SetActive(false);
            }
            // sets the OK button active
            else
            {
                oneUnlockedButton.gameObject.SetActive(false);
            }
            // Skin is Bought so set isFound in true
            isFound = true;
        }

    }
    // if the skin is not found is isFound false so
    if (isFound==false)
    {
        // sets the right buttons active and the false inactive
        oneLockedButton.gameObject.SetActive(true);
        oneUnlockedButton.gameObject.SetActive(false);
        oneOkButton.gameObject.SetActive(false);
    }

}
```

Abbildung 27: Controller der Erweiterungs Scene

Skin Freischalten

Für das Freischalten des Skins wird ein Panel geöffnet, das den Benutzer fragt, ob er seine Punkte für das Freischalten des Skins verwenden will. Bestätigt er dies, wird das Listobjekt übergeben und die Skins werden mit einer Foreach-Schleife nach dem Namen (also der SkinID) des Game-Objektes durchsucht. Hat er eine Übereinstimmung, werden die Punkte durch die Methode „BuyASkinWithPoints()“ vom Benutzer-Score abgezogen und der Skin wird über die „BoughtSkinsDAL.Instance.BoughtSkinsSaveCreate(item, false);“ in der Datenbank gespeichert.

```
/// <summary>
/// is the method for the save of buying a Skin
/// </summary>
public void BuyASkinWithPoints()
{
    // fetches all skins from the database
    var SkinsToBuy = SkinsDAL.Instance.GetByCurrentOffererStatus(FreeToBuy);
    // goes through the Skins from the database
    foreach (var item in SkinsToBuy)
    {
        // if the name of the Object the SkinID
        if (item.SkinId == ListObjectToSave.name)
        {
            // Create a new CleaningSpaceService object and interrogate the score of the User
            CleaningSpaceService css = new CleaningSpaceService();
            var UserScore = css.GetMyPoints();
            // if the user score is greater than or equal to the price of the skin
            if (UserScore == item.Price || UserScore >= item.Price)
            {
                // Saves the skin data in the database
                BoughtSkinsDAL.Instance.BoughtSkinsSaveCreate(item, false);
                // deducts the price from the user score
                css.BuyASkinWithPoints(item.Price);
                SceneManager.UnloadScene(Const.SCENE_SkinActivationExtension_Pfad);
                SceneManager.LoadScene(Const.SCENE_SkinActivationExtension_Pfad, LoadSceneMode.Additive);
            }
            else
            {

            }
        }
    }

}
```

Abbildung 29 : Skin Freischalten

Skin aktivieren

Wird das offene Schloss gedrückt, so aktiviert der Benutzer den Skin und die Methode „SettheSkinActive“ wird aufgerufen, um das Listobjekt zu übergeben. Wieder wird in der Liste der freigeschalteten Skins nach dem Namen des Objektes gesucht und die „UpdateBoughtSkins“ Methode des „BoughtSkinsDAL“ aufgerufen und mit Status und Objekt übergeben. Zusätzlich erfolgt ein Update in der Datenbank.

```
public void SettheSkinActive(GameObject gameObject)
{
    // create a new BoughtSkinsDAL object to update the ActivityStatus of a skin
    BoughtSkinsDAL bought = new BoughtSkinsDAL();
    // goes through the BoughtSkins from the database and update the right one
    bought.UpdateBoughtSkins(BoughtSkinsList.FirstOrDefault(f => f.SkinId == gameObject.name),true);
    // close and open the expansion scene
    SceneManager.UnloadScene(Const.SCENE_SkinActivationExtension_Pfad);
    SceneManager.LoadScene(Const.SCENE_SkinActivationExtension_Pfad, LoadSceneMode.Additive);
}
```

Abbildung 30: SetTheSkinActiv

Skin deaktivieren

Nach dem Klicken des grünen Buttons wird die Methode „SettheSkinInActive“ aufgerufen und es geschieht dasselbe, wie beim Aktivieren des Skins nur mit dem Status „false“.

```
public void SettheSkinInActive(GameObject gameObject)
{
    // create a new BoughtSkinsDAL object to update the ActivityStatus of a skin
    BoughtSkinsDAL bought = new BoughtSkinsDAL();
    // goes through the BoughtSkins from the database and update the right one
    bought.UpdateBoughtSkins(BoughtSkinsList.FirstOrDefault(f => f.SkinId == gameObject.name),false);
    // close and open the expansion scene
    SceneManager.UnloadScene(Const.SCENE_SkinActivationExtension_Pfad);
    SceneManager.LoadScene(Const.SCENE_SkinActivationExtension_Pfad, LoadSceneMode.Additive);
}
```

Abbildung 31: Deaktivieren

Skin Detail Controller

Es wurde auch ein Controller für die Skin-Detail-Scene implementiert. Eine Methode, um den Text des Popups für den passenden Skin zu setzen und eine um den Text davor in der Handy Nutzersprache zu übersetzen. In der „Start“-Methode wird die Übersetzung gestartet und diese endet mit dem Aufruf der Setzen Methode. Es wird das Listobjekt übergeben und in ein „SkinsActivationDTO“ Objekt gespeichert, sollte der Skin existieren. Danach wird die Scene geöffnet und die „Start“-Methode ausgeführt. Übersetzung des Textes, bevor es in der Scene angezeigt wird:

```
IEnumerator Translate()
{
    // gives the User Language of the smathPhone
    string languageCode = LocalizationManager.CurrentLanguageCode;
    // check if the item to translate has the right user Language
    if (string.IsNullOrEmpty(ToChangeItem.LanguageCode) || ToChangeItem.LanguageCode == LocalizationManager.CurrentLanguageCode)
    {
        this.SetTheInfos(ToChangeItem);
    }
    // is this false translate the Description
    else
    {
        // string for the Transleted text
        string translatetText = null;
        Task.Run(() =>
        {
            // translate the description of the skin in the language with the LanguageCode
            translatetText = TranslationService.TranslateText(ToChangeItem.Description, languageCode).Result;
        });
        yield return null;
        // return null if the translation failed
        while (translatetText == null)
        {
            yield return null;
        }
        ToChangeItem.Description = translatetText;
        // return the translated Description of the Skin
        this.SetTheInfos(ToChangeItem);
    }
}
```

Abbildung 32: Übersetzung Skin Detail

Qualitätssicherung

Für das Testen der MRCS-Plattform nachdem Änderungen vorgenommen wurden, wird alles nach dem aktuellen Testplan getestet. Dieser Testplan enthält neben Units-Test, Integration-Test auch einen händischen Testplan mit folgendem Beispielhaften Test-Szenario:

Bei diesem wird die MRCS-App neu auf das Test-Smartphone installiert und nach dem Beenden der Installation wird die App gestartet. Nach dem Start der App wird ein neuer Benutzer erstellt und mit diesem alle Funktionen der App getestet und geprüft, ob nach den Änderungen auch alles noch funktioniert. Es wird die Genauigkeit der Punkte in der realen Welt getestet. In die Vierecke werden die Punkte beim Erstellen gesetzt, bei dem erneuten Suchen wird geprüft, ob sich die Punkte immer noch in den Vierecken befinden. Nachdem alles einwandfrei mit einem neuen Benutzer getestet wurde, macht man dies noch einmal mit einem schon vorhandenen Benutzer, um sicher zu stellen, dass die Änderung weder Neu-User noch Bestands-Benutzer beeinträchtig.
Nach all dem wurden noch kleinere Probleme, wie eine falsche Button-Funktion, behoben und nochmals getestet.

Projektergebnis

Ist-/ Soll-Vergleich

Die folgende Tabelle stellt die *Muss-Kriterien* und deren Umsetzung dar. Die Umsetzung der Muss-Kriterien verlief problemlos und die Skin-Erweiterung konnte wie vorgesehen umgesetzt werden.

Muss Kriterien	Umsetzung
Skin Freischalten	
Darstellen aller Skins die der Benutzer freischalten kann	Die Skins werden aus der Datenbank abgerufen und in einem ScrollView in der Scene dargestellt
Freischalten Scene mit Abbruch und Bestätigen vor Freischaltung	Will der Benutzer einen Skin freischalten, kommt erst eine Frage, ob man sich sicher ist. Bestätigt er diese, wird der Skin als freigeschaltet in der Datenbank gespeichert
Punkt Abzug vom Score nach Freischaltung	Nach Bestätigung wird der Preis der Skins vom Score abgezogen
Skin Setzen und Verwenden	
Skin einsetzbar anstatt der Drachen	Skin wurde animiert, als Prefab in Unity erstellt und ist anwendbar
Aktivieren und Deaktivieren des Skins durch klicken in der Erweiterung	Über die Erweiterung kann der Benutzer den Skin aktivieren oder deaktivieren was durch ein grünes Icon sichtbar ist
Beim Erledigen von Aufgaben Animation sehen	Nach Erledigen der Aufgabe wird eine Animation gestartet

Tabelle 5: Muss Kriterien

Abnahme

Die Erweiterung wurde nach dem Testen und dem einwandfreien Funktionieren dem Product Owner vorgestellt. Dieser hatte nichts auszusetzen und keine Änderungswünsche, so dass die Implementierung abgeschlossen wurde.

Fazit

Die Skin Freischaltung ist eine gute und kreative Erweiterung des Kids Modes und macht diesen noch spannender. Die Skin Freischaltungserweiterung soll in Zukunft weiter implementiert und vergrößert werden. Die Umsetzung konnte wie geplant durchgeführt werden.

Benutzerhandbuch

Öffnen der Skin Freischaltung

Um auf die Skin Freischaltung zu kommen, muss der Benutzer in der MRCS-App auf das Menü gehen und auf „Gewinne“. Danach kommt der Benutzer auf die Preisliste, wo Cleani (Staubsauger) geklickt werden muss.

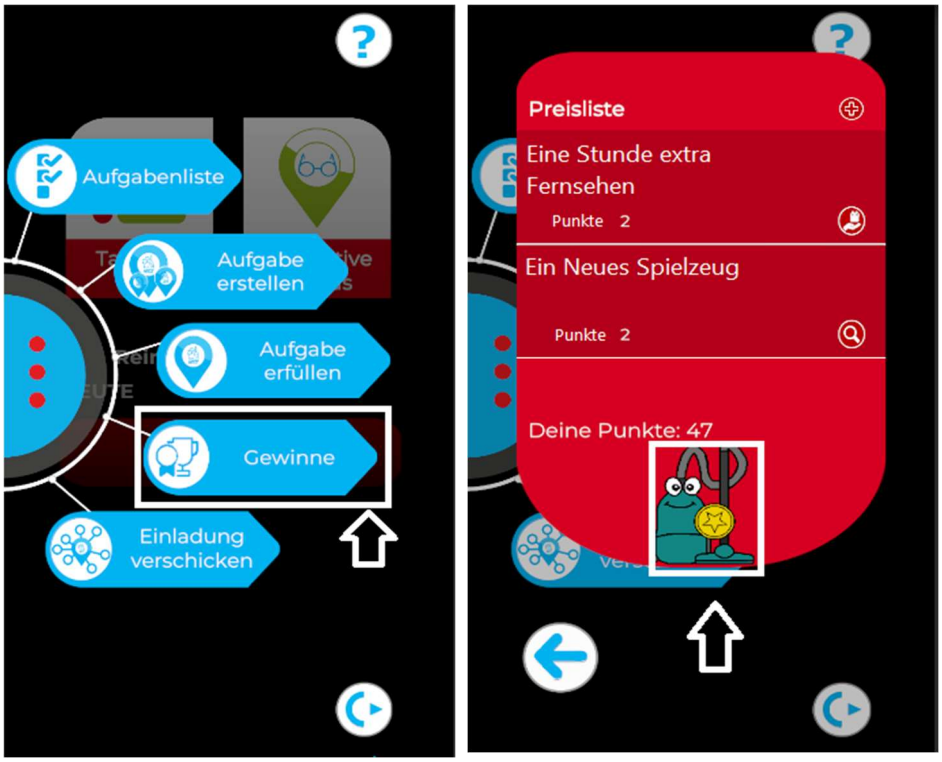


Abbildung 33: Skin Freischalten

Skin Informationen

Wenn der Benutzer genauere Informationen zu dem Skin bekommen möchte, kann er auf die Lupe gehen und bekommt einen Info-Screen, welchen er durch den „Zurück“-Button wieder schließen kann.

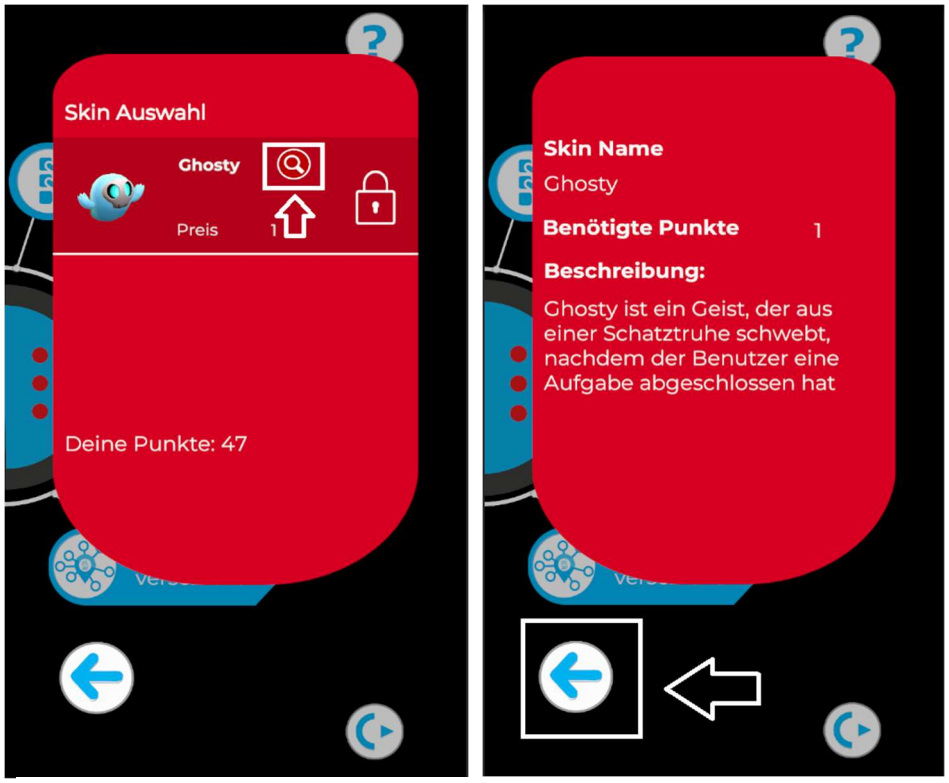


Abbildung 34: SkinInfo

Skin Freischalten

Nach dem Klicken auf Cleani kommt der Benutzer auf die Erweiterung. Um einen Skin freizuschalten, muss er auf das Schloss gehen. Dann öffnet sich ein Screen, in dem der Benutzer entscheidet, ob er seine Punkte dafür verwenden möchte oder nicht. Klickt er auf „Bestätigen" wird der Preis vom Score abgezogen und der Skin freigeschaltet.

Abbildung 35: Skin Freischalten

Skin aktivieren/deaktivieren

Ist der Skin freigeschaltet, kann der Benutzer das geöffnete-Schloss-Icon anklicken und somit den Skin aktivieren, dann kommt ein grünes Icon. Um den Skin zu deaktivieren muss der Benutzer den grünen Icon nochmals klicken Dann erscheint erneut das geöffnete Schloss.

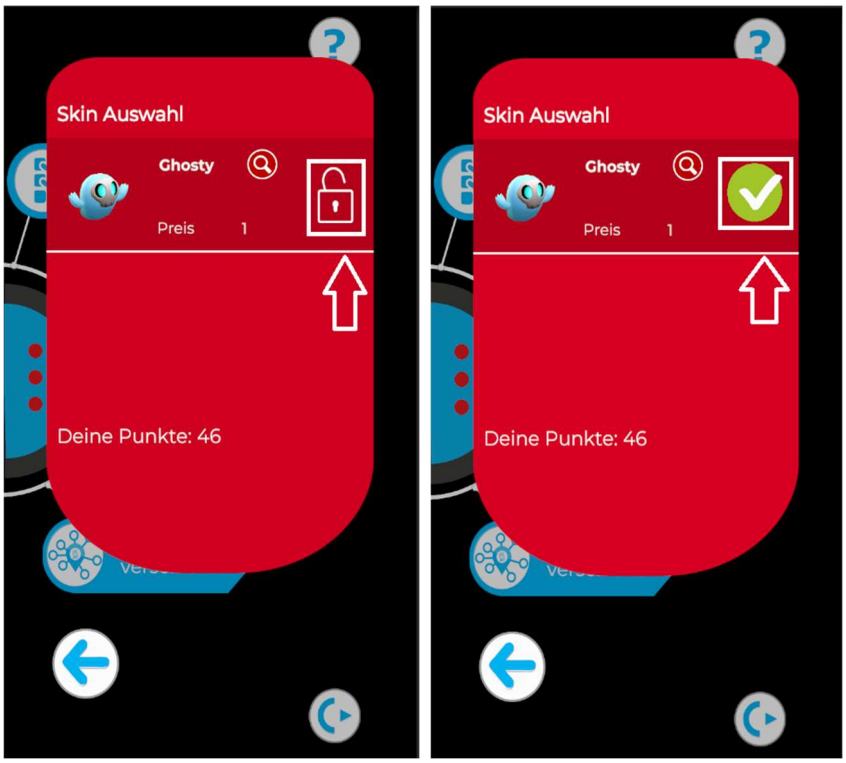


Abbildung 36: Skin Aktivieren/ Deaktivieren

Glossar

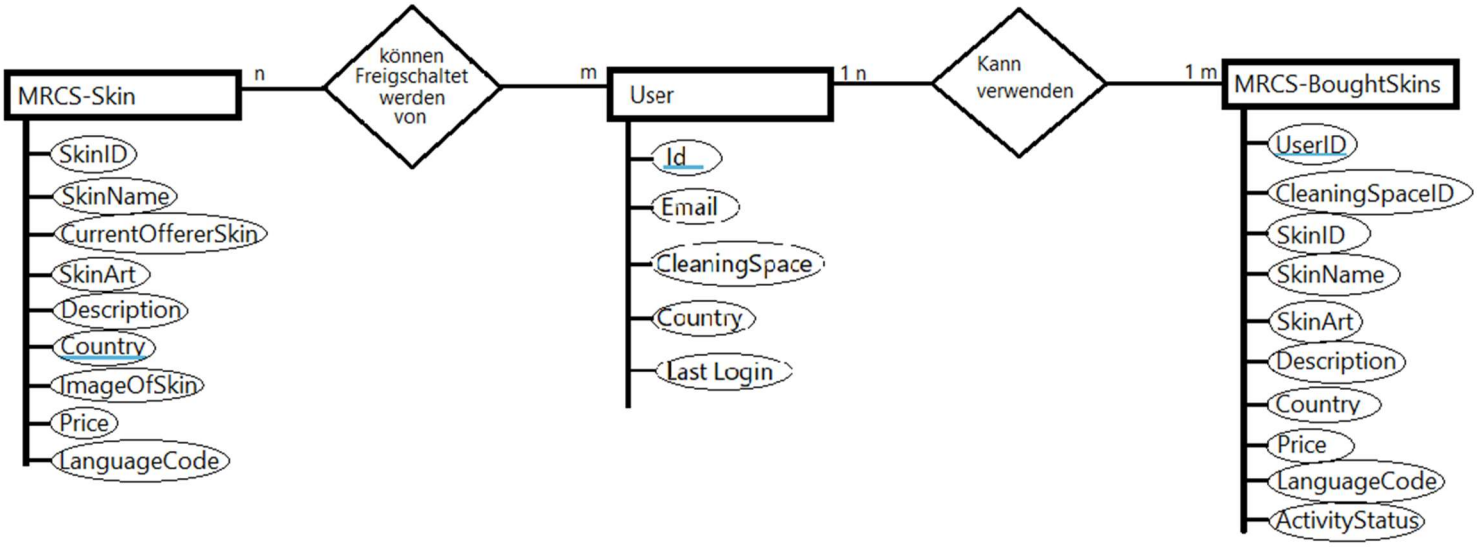
Product Owner	=	Projektleiter
Gameobjekte	=	Objekte in einer Scene, z.B. Button, Monster
Assetstore	=	Ist ein Store Für Unity-Assets im Unity Editor

Scenen	=	Seiten der App, die der User sieht
Skin	=	Animierte Mixed Reality Figur für den Kids Mode
Prefab	=	Ein Zusammenschluss von Gameobjekten im Unity
Cleani	=	Staubsauger Icon in der Erweiterung
CosmosDB	=	Datenbank zum Abspeichern von Daten
MRCS	=	Mixed Reality Cleaning Solution
Mixed Reality	=	Vermischung der realen und virtuellen Welt
UserID	=	Die ID des Aktiven Benutzers
CleaningSpaceID	=	ID des Aktuellen Reinigungs Bereiches

Abbildungen

Tabellen

Anlagen



Anlage 1: Datenbankschema

IHK

Diese Erklärung ist beim Einreichen der Dokumentation jedem Exemplar anzuhängen!

IV. 3 Persönliche Erklärung

zur Projektarbeit und Dokumentation im Rahmen der Abschlussprüfung in den IT-Berufen

Ich versichere durch meine Unterschrift, dass ich die Durchführung der betrieblichen Projektarbeit als auch die dazugehörige Dokumentation selbständig in der vorgegebenen Zeit erarbeitet habe. Alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, wurden von mir als solche kenntlich gemacht.

Ebenso bestätige ich, dass ich bei der Erstellung der Dokumentation meiner Projektarbeit weder teilweise noch vollständig Passagen aus Projektarbeiten übernommen habe, die bei der prüfenden oder einer anderen Kammer eingereicht wurden.

Ich bestätige, dass die Dokumentation keine Betriebsgeheimnisse bzw. schutzwürdige Betriebs- oder Kundendaten enthält, das Urheberrecht beachtet wurde und es keine datenschutzrechtlichen Bedenken gibt.

Ort, Datum

Rainau 05.05.2021

Unterschrift des Prüfungsteilnehmers

Liana Unseld

Ich habe die obige Erklärung zur Kenntnis genommen und bestätige, dass die betriebliche Projektarbeit einschließlich der Dokumentation in der vorgegebenen Zeit in unserem Betrieb durch den Prüfungsteilnehmer angefertigt wurde.

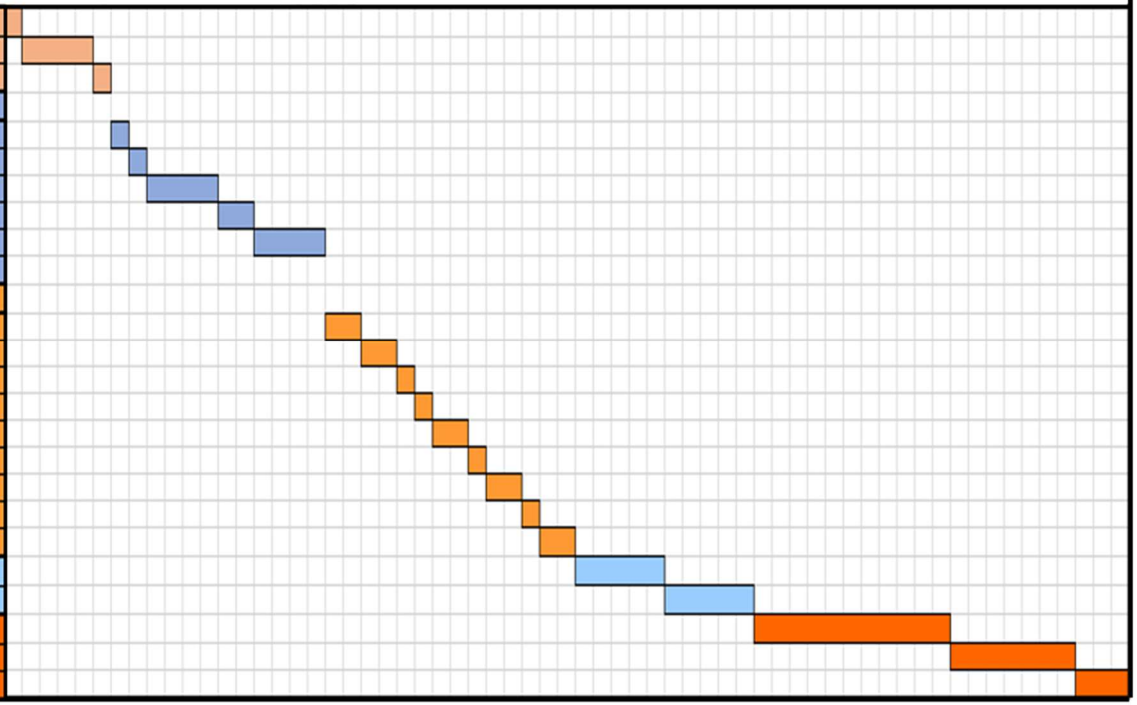
Ort, Datum

Hüttlingen, 05.05.2021

Unterschrift des Ausbilders

Anlage 2: Eigenständigkeitserklärung

Tätigkeit	Arbeitsschritt
Vorbereitung	Analyse der Anforderungen
	Erstellen Pflichtenheft und Projektplan
	Abstimmung und Abnahme durch Product Owner
Implementierung	Implementieren des Backend
	Erstellen eines Containers für die Skins, die der Benutzer freischalten kann.
	Erstellen eines Containers für die Speicherung der freigeschalteten Skins
	Erstellen dreier Data Access Layer für den MRCS-Datenservice
	Hinzufügen zweier DTO's zur MRCS Shared Library
	Testen der Speicherung und der Data Access Layer;
	Get and Post Abfragen über Postman
	Implementieren des Frontends
	Erstellen der Erweiterungs Scene in Unity
	Implementieren eines Controllers für die Scene
	Implementieren der Anzeige logik für Cleani
	Erstellen einer Info-Scene in Unity
	Implementieren eines Controllers für die Info Scene mit einer Übersetzung der Description des Skins
	Kauf und Import des Skins aus dem Unity Asset Store
	Erstellen des Prefabs mit Animation für den Ghosty Skin
	Implementierung eines Animations Controllers für den Skin
	Implementierung der Verwendungslogik für den Skin anstatt der Drachen
Qualitätssicherun	Testen der Erweiterung
	Testen des neuen Skins
Dokumentation	Verfassen der Dokumentation
	Erstellung des Benutzerhandbuchs
	Finale Abnahme durch Product Owner



Anlage 3: Gantt-Diagramm