Ein Bild, das Screenshot, Grafiken, Schrift, Grafikdesign enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Hausarbeit**im Bildungsgang

„Staatlich geprüfte/r Wirtschaftsinformatiker/in“

gemäß §5 der Ausbildungs- und Prüfungsordnung

Entwicklung einer LegoSet-Verwaltungssoftware

vorgelegt von: Florian Emilio Fritz

Klasse: WI23A1

Adresse: Steinbachstraße 97

Ort: 66424 Homburg

E-Mail: florian.fritz.1@gmx.de

Abgabetermin: 15.05.2025

Betreuer/in: Herr Schuler

Inhaltsverzeichnis

[Benutzerhandbuch 3](#_Toc197900420)

[API-Schlüssel eingeben 3](#_Toc197900421)

[Login 4](#_Toc197900422)

[Die Startseite 6](#_Toc197900423)

[LegoSet-Suche 7](#_Toc197900424)

[Sammlung 10](#_Toc197900425)

[Statistken 11](#_Toc197900426)

[Implementierung 12](#_Toc197900427)

[Klassendiagramm 12](#_Toc197900428)

[Beschreibung wichtiger Quellcode Ausschnitte. 13](#_Toc197900429)

[Beschreibung der Schnittstellen (API) 19](#_Toc197900430)

[Abweichungen vom Pflichtenheft/Lastenheft 20](#_Toc197900431)

## Benutzerhandbuch

### API-Schlüssel eingeben

Ein Bild, das Text, Screenshot, Reihe, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Beim ersten Start fragt das Programm nach deinem persönlichen Rebrickable-Schlüssel. Diesen bekommst du zusammen mit dem Programm (z. B. als Textdatei oder per E-Mail).

Rebrickable ist die API, die im Hintergrund die LEGO-Datenbank abfragt.

Damit das funktioniert, muss dein Schlüssel bei jeder Abfrage mitgeschickt werden.

Was passiert nach dem Klick auf Speichern?

* Der Schlüssel wird geprüft

Das Programm schaut, ob der eingegebene Schlüssel das richtige Format hat – also z. B. nicht zu kurz oder leer ist.

* Der Schlüssel wird dauerhaft gespeichert

Damit man ihn nicht jedes Mal neu eingeben muss, wird er automatisch in einer Textdatei auf deinem Rechner abgelegt.

* Die Verbindung zu Rebrickable wird aktiviert

Von jetzt an wird der Schlüssel bei jeder Online-Abfrage automatisch mitgesendet.

Dadurch kannst du Sets suchen, Themen vorladen oder Preise abrufen – ohne dich weiter darum kümmern zu müssen.

### Login

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

KI-generierte Inhalte können fehlerhaft sein.

Oben rechts kannst du dich jetzt anmelden.  
Nach dem Klick öffnet sich ein neues Fenster, in dem du dich entweder einloggen, neu registrieren oder dein Passwort mit einem Sicherheitscode zurücksetzen kannst.

#### Loginfenster

Ein Bild, das Text, Screenshot, Display, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

* **Benutzername**  
  hier gibst du deinen registrierten Namen ein.
* **Passwort**  
  Das zugehörige Passwort wird verdeckt eingegeben.
* **Login-Button**  
  Meldet dich an, nur möglich wenn ihre Einlogdaten stimmen
* **„Registrieren“**  
  öffnet ein weiteres Fenster zur Neuanmeldung.
* **„Passwort vergessen“**  
  Öffnet das Zurücksetzungs-Fenster, in dem du über deinen persönlichen Sicherheitscode ein neues Passwort setzen kannst.

#### Registrierungsfenster

Ein Bild, das Text, Screenshot, Display, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

Im Registrierungsfenster kannst du dich jetzt mit einem neuen Benutzerkonto Registrieren.

Trag dafür deinen gewünschten Benutzernamen, deine E-Mail-Adresse und ein sicheres Passwort ein.

Nach dem Klick auf „Registrieren“ bekommst du eine wichtige Info: dein persönlicher Sicherheitscode.

Diesen Code brauchst du später, falls du dein Passwort mal vergessen solltest – also am besten gleich notieren.

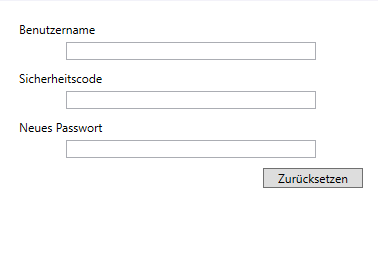
Ein Bild, das Text, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Dein Benutzer wird direkt in der lokalen Datenbank gespeichert, und du kannst dich danach sofort anmelden.

Mit dem Button „Zurück“ kommst du wieder zum Loginfenster.

#### Passwort vergessen



Nachdem drücken auf Passwort vergessen öffnet sich ein neues Fenster in dem du dein Benutzername, Sicherheitscode und dein neues Passwort eingibst

Wenn der Sicherheitscode zu deinem Benutzernamen passt, wird dein neues Passwort übernommen.

### Die Startseite

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

KI-generierte Inhalte können fehlerhaft sein.

Die Startseite begrüßt dich nach dem Login mit deinem Namen und zeigt dir direkt, wie viele Sets aktuell in deiner Sammlung gespeichert sind.

Außerdem siehst du auf einen Blick, ob dein API-Key aktiv ist – also ob die Verbindung zur Rebrickable-Datenbank funktioniert.

Über die farbigen Schaltflächen kannst du direkt in die wichtigsten Bereiche springen:

* Sammlung anzeigen: deine gespeicherten Sets verwalten
* Set suchen: neue Sets per Setnummer, Name oder Thema finden
* Statistiken: Auswertungen zu deinem Inventar (Wert, Gewinn)

Außerdem siehst du deine Zuletzt hinzugefügten Sets.

Oben rechts kannst du dich außerdem jederzeit wieder abmelden.

### LegoSet-Suche

Ein Bild, das Text, Screenshot, Software, Display enthält.

KI-generierte Inhalte können fehlerhaft sein.

Hier kannst du gezielt nach LEGO-Sets per Api-Abfrage suchen und neue Sets in deiner Sammlung speichern.

Dafür stehen dir zwei Möglichkeiten zur Verfügung:

* Manuelle Suche über die Eingabezeile (Setnummer, Name oder Thema)

Optional mit Jahres Filter.

* Vorladen ganzer Themen (z. B. „Star Wars“) direkt aus der Rebrickable-Datenbank

Die Ergebnisse werden dir in einer ListView angezeigt.

Mit einem Klick auf „Hinzufügen“ wird das gewünschte Set zu deiner lokalen Sammlung hinzugefügt.

Im Hintergrund wird geprüft, ob das Set bereits vorhanden ist.

Falls es noch nicht in deiner Datenbank gespeichert ist, wird es dort neu angelegt (inkl. Name, Setnummer, Jahr, Thema usw.).

Der Datensatz wird dauerhaft in deiner lokalen SQLite-Datenbank gespeichert.

Ein Bild, das Text, Screenshot, Software, Display enthält.

KI-generierte Inhalte können fehlerhaft sein.

Mit dem Suchfeld am oberen Rand kannst du gezielt nach bestimmten LEGO-Sets suchen.

Dabei wählst du im Dropdown-Menü aus, wonach du suchen möchtest:

* nach einer Setnummer (z. B. „10188“)
* nach einem Namen (z. B. „Death Star“)
* nach einem Thema (z. B. „Ultimate Collector Series“)
* Optional Jahres-Filter (z.B „2024“)

Nach dem Klick auf „Suchen“ wird die Rebrickable-Datenbank abgefragt.

Alle passenden Sets werden in einer Liste darunter angezeigt – inklusive Setnummer, Name, Thema und Jahr.

Außerdem kannst du direkt die gewünschte Anzahl an Sets angeben, die du zu deiner Sammlung hinzufügen möchtest.

Mit „Hinzufügen“ wird das ausgewählte Set in deiner lokalen Datenbank gespeichert.

Falls es noch nicht vorhanden ist, wird es automatisch neu angelegt.

Die Angabe zur Anzahl wird in der Tabelle (BenutzerSet) hinterlegt und steht dann direkt in deiner Sammlung zur Verfügung.

Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

KI-generierte Inhalte können fehlerhaft sein.

Mit der Funktion „Thema vorladen“ kannst du ganze LEGO-Themen wie z. B. „City“, „Star Wars“ oder „Technic“ vollständig in deine lokale Datenbank speichern.

Um die wichtigsten Sets auch dann griffbereit zu haben, wenn du das Programm mal offline genutzt werden soll.

Wähle dazu einfach ein Thema aus der Liste aus und klicke auf „Thema vorladen“.

Das Programm lädt alle zugehörigen Sets über die Rebrickable-API herunter und speichert sie dauerhaft auf deinem Rechner. Bereits vorhandene Sets werden dabei automatisch übersprungen

### Sammlung

Ein Bild, das Text, Screenshot, Zahl, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

In der Sammlungsansicht siehst du alle LEGO-Sets, die du deiner persönlichen Sammlung hinzugefügt hast.

Für jedes Set werden dir wichtige Informationen angezeigt – wie Setnummer, Name, Thema, Jahr, Anzahl, gezahlter Preis und UVP.

Man kann die Anzahl, den gezahlten Preis , denn UVP (€) und die Notizen jederzeit ändern.

Mit einem Klick auf „Speichern“ werden deine Änderungen direkt in der lokalen Datenbank gespeichert.

Falls du ein Set nicht mehr hast, kannst du es über den Button „Löschen“ aus deiner Sammlung löschen.

Mit dem Button „Set manuell hinzufügen“ oben rechts kannst du eigene Sets eintragen, die du nicht über die Rebrickable-Suche gefunden hast.

So lassen sich z. B. seltene oder eigene Sets ebenfalls erfassen.

#### Set manuell hinzufügen

Ein Bild, das Text, Screenshot, Zahl, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Im Fenster „Set manuell hinzufügen“ kannst du LEGO-Sets eintragen, die nicht über die Rebrickable-Suche verfügbar sind – zum Beispiel seltene Modelle, Sondereditionen oder eigene Sets.

Du gibst dafür einfach die wichtigsten Daten per Hand ein:  
Setnummer, Name, Thema, Jahr, Preis, Anzahl und optional eine Notiz.

Mit einem Klick auf „Speichern“ wird das neue Set in deiner lokalen Datenbank gespeichert und automatisch zu deiner Sammlung hinzugefügt.  
Es verhält sich anschließend genauso wie ein regulär importiertes Set und kann jederzeit bearbeitet oder gelöscht werden.

### Statistken

Ein Bild, das Text, Screenshot, Software, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

In den Statistiken bekommst du eine Übersicht über deine LEGO-Sammlung.

Alle Werte beziehen sich auf den aktuell angemeldeten Benutzer.

* Gesamtanzahl Sets

Wie viele Sets du insgesamt in deiner Sammlung hast (unabhängig vom Thema).

* Gesamtwert bezahlt (€)

Der Gesamtwert basierend auf dem Preis, den du tatsächlich gezahlt hast.

* Gesamtwert UVP (€)

Der Gesamtwert basierend auf dem offiziellen Listenpreis (UVP) der Sets.

* Differenz UVP vs. Bezahlt (€)

Zeigt dir, ob du günstiger oder teurer eingekauft hast.

Diese Werte werden direkt aus der lokalen Datenbank berechnet und live aktualisiert, sobald du in der Sammlung etwas änderst.

Sie helfen dir, einen Überblick über den Wert deiner Sammlung und deine Investitionen zu behalten.

## Implementierung

### Klassendiagramm

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

In diesem Projekt geht es um eine Anwendung zur Verwaltung von LEGO-Sets. Das Klassendiagramm zeigt, wie die verschiedenen Teile der Anwendung zusammenhängen und welche Daten dabei eine Rolle spielen.

Im Mittelpunkt steht die AppDbContext-Klasse. Sie ist sozusagen das Herzstück der Datenbankanbindung und sorgt dafür, dass alle Informationen über Benutzer, Sets und deren Besitz sauber gespeichert und abgerufen werden können. Dabei greift sie auf drei wichtige Klassen zu: Benutzer, LegoSet und BenutzerSet.

**Benutzer**:  
Diese Klasse repräsentiert eine Person, die sich bei der Anwendung anmeldet. Gespeichert werden z. B. Benutzername, E-Mail und Passwort (als Hash). Ein Benutzer kann natürlich mehrere Sets besitzen.

**LegoSet**:  
Jedes LEGO-Set hat hier seine eigene Klasse mit Infos wie Name, Set-Nummer, Thema, Jahr und Preis. Auch ein Set kann von mehreren Benutzern gekauft worden sein.

**BenutzerSet**:  
Diese Klasse verbindet Benutzer und Sets. Sie speichert z. B., wann ein Set gekauft wurde, wie viel es gekostet hat, wie oft es gekauft wurde und ob es irgendwelche Notizen dazu gibt. So lässt sich genau nachvollziehen, wer was besitzt.

**Externer Service: RebrickableService**

Ein weiterer Teil des Projekts ist der RebrickableService. Der verbindet die App mit der offiziellen Rebrickable-API im Internet. Darüber kann man z. B. nach neuen Sets suchen oder sich alle Sets eines bestimmten Themas anzeigen lassen. Auch Preise lassen sich damit abrufen. Die Klasse ist komplett statisch und hilft einfach dabei, Live-Daten von außen in die App zu bringen.

### Beschreibung wichtiger Quellcode Ausschnitte.

#### Login

private void BtnLogin\_Click(object sender, RoutedEventArgs e)

{

string benutzername = txtBenutzername.Text.Trim();

string passwort = txtPasswort.Password.Trim();

string hash = HashPasswort(passwort);

using (var db = new AppDbContext())

{

var user = db.Benutzer.FirstOrDefault(u => u.Benutzername == benutzername && u.PasswortHash == hash);

if (user != null)

{

App.Current.Properties["BenutzerId"] = user.BenutzerId;

App.Current.Properties["BenutzerName"] = user.Benutzername;

this.DialogResult = true;

this.Close();

}

else

{

MessageBox.Show("Falscher Benutzername oder Passwort.");

}

}

Hier handelt es sich um die Methode, die aufgerufen wird, wenn man im Login-Fenster auf den Button klickt. Der Ablauf ist recht simpel, aber effektiv.

Zuerst holt sich die Methode den Benutzernamen und das Passwort, die der User eingegeben hat. Beide Werte werden von überflüssigen Leerzeichen befreit. Das Passwort wird direkt in einen Hash umgewandelt, damit es nicht im Klartext weiterverarbeitet wird – wie es sich gehört.

Dann wird eine Verbindung zur Datenbank geöffnet (über den AppDbContext). In dieser wird geprüft, ob es einen Benutzer gibt, dessen Benutzername und Passwort-Hash zu dem eingegebenen passen. Also ein klassischer Login-Vergleich.

Wenn ein passender Benutzer gefunden wird, werden dessen ID und Name in den globalen App-Eigenschaften gespeichert, damit man später im Programm noch darauf zugreifen kann. Dann wird das Login-Fenster geschlossen und der Benutzer ist eingeloggt.

Falls der Benutzername oder das Passwort falsch war, wird einfach eine Fehlermeldung ausgegeben.

#### Set per API suchen

private async void BtnSuchen\_Click(object sender, RoutedEventArgs e)

{

string suchbegriff = txtSuche.Text.Trim();

if (string.IsNullOrEmpty(suchbegriff))

{

MessageBox.Show("Bitte einen Suchbegriff eingeben.");

return;

}

string suchtyp = (cmbSuchtyp.SelectedItem as ComboBoxItem)?.Content?.ToString();

// Sets suchen (immer über API)

suchErgebnisse = suchtyp switch

{

"Setnummer" => await RebrickableService.SucheSetsAsync(suchbegriff, "set\_num"),

"Name" => await RebrickableService.SucheSetsAsync(suchbegriff, "name"),

"Thema" => await RebrickableService.SucheSetsAsync(suchbegriff, "theme"),

\_ => new List<LegoSet>()

};

// Jahr auslesen

string jahrText = cmbJahr.SelectedItem?.ToString();

int? filterJahr = (jahrText != "Alle" && int.TryParse(jahrText, out int jahr)) ? jahr : null;

// 🧹 Jahrfilter anwenden – unabhängig von Datenquelle

if (filterJahr != null)

{

suchErgebnisse = suchErgebnisse

.Where(set => set.Jahr == filterJahr.Value)

.ToList();

}

// Anzeige aktualisieren

lvErgebnisse.ItemsSource = suchErgebnisse;

}

Diese Methode wird ausgeführt, wenn man in der App auf den "Suchen"-Button klickt. Sie sorgt dafür, dass man nach bestimmten LEGO-Sets suchen kann – entweder nach Name, Thema oder Setnummer.

Zuerst wird überprüft, ob überhaupt ein Suchbegriff eingegeben wurde. Wenn das Feld leer ist, bekommt der Nutzer direkt eine Meldung und der ganze Vorgang wird abgebrochen.

Wenn aber ein Begriff drinsteht, wird geschaut, welcher Suchtyp im Dropdown ausgewählt wurde (also ob man z. B. nach dem Namen oder der Setnummer suchen will). Danach wird eine passende Abfrage an die Rebrickable-API geschickt also direkt online gesucht. Das Ganze läuft asynchron, damit die App dabei nicht hängen bleibt.

Dann geht’s weiter mit einem optionalen Filter: Wenn im Jahr-Dropdown ein konkretes Jahr ausgewählt wurde (z. B. 2021), werden die Suchergebnisse nochmal darauf gefiltert. So sieht man nur die Sets aus diesem Jahr.

Zum Schluss werden die Ergebnisse in der ListView angezeigt, also schön im UI dargestellt.

#### Thema Vorabladen

private async void BtnThemaVorladen\_Click(object sender, RoutedEventArgs e)

{

if (cmbVorladeThema.SelectedItem == null)

{

MessageBox.Show("Bitte ein Thema auswählen.");

return;

}

string thema = cmbVorladeThema.SelectedItem as string;

var sets = await RebrickableService.SucheAlleSetsNachThemaAsync(thema);

if (sets == null || sets.Count == 0)

{

MessageBox.Show("Keine Sets gefunden.");

return;

}

// Fortschrittsbalken konfigurieren

progressBar.Visibility = Visibility.Visible;

progressBar.Minimum = 0;

progressBar.Maximum = sets.Count;

progressBar.Value = 0;

int neueSets = 0;

// Sets lokal in der Datenbank speichern, wenn sie noch nicht existieren

using (var db = new AppDbContext())

{

foreach (var set in sets)

{

bool existiert = db.Sets.Any(s => s.Nummer == set.Nummer);

if (!existiert)

{

db.Sets.Add(set);

neueSets++;

}

progressBar.Value += 1;

await Task.Delay(5); // kleine Pause für optisches Feedback

}

db.SaveChanges();

}

progressBar.Visibility = Visibility.Collapsed;

MessageBox.Show($"{sets.Count} Sets gefunden.\nDavon {neueSets} neue Sets gespeichert.");

}

Diese Methode wird ausgeführt, wenn man auf den Button klickt, um LEGO-Sets zu einem bestimmten Thema vorzuladen. Ziel ist es, Sets aus der Rebrickable-API zu holen und sie lokal in der eigenen Datenbank zu speichern aber natürlich nur, wenn sie dort noch nicht existieren.

Zuerst wird geprüft, ob überhaupt ein Thema ausgewählt wurde. Falls nicht, bekommt man einen Hinweis und es passiert nichts weiter. Wenn ein Thema gewählt wurde, wird über den RebrickableService nach passenden Sets gesucht das läuft wieder asynchron, also im Hintergrund.

Wenn keine Sets gefunden wurden, gibt’s ebenfalls eine Meldung. Wenn doch, geht’s los:

Ein Fortschrittsbalken wird eingeblendet, damit man sieht, dass gerade etwas passiert. Danach wird durch alle gefundenen Sets durchiteriert und jeweils geprüft, ob das Set anhand seiner Nummer schon in der lokalen Datenbank vorhanden ist. Falls nicht, wird es hinzugefügt und gezählt.

Die kleine Pause (await Task.Delay(5)) sorgt dafür, dass der Fortschrittsbalken flüssiger aussieht, rein fürs Auge.

Wenn alles durchgelaufen ist, wird gespeichert und der Fortschrittsbalken wieder ausgeblendet. Zum Schluss gibt’s nochmal eine kurze Zusammenfassung: Wie viele Sets wurden insgesamt gefunden und wie viele davon waren wirklich neu.

#### Set zur Sammlung hinzufügen

private async void BtnHinzufuegen\_Click(object sender, RoutedEventArgs e)

{

if (sender is Button btn && btn.Tag is LegoSet selectedSet)

{

using (var db = new AppDbContext())

{

var setInDb = db.Sets.FirstOrDefault(s => s.Nummer == selectedSet.Nummer);

// Set noch nicht in DB → Preis laden + speichern

if (setInDb == null)

{

double preis = await RebrickableService.LadePreisVonSetAsync(selectedSet.Nummer);

selectedSet.PreisUVP = preis;

db.Sets.Add(selectedSet);

db.SaveChanges();

setInDb = selectedSet;

}

// Benutzer-ID aus globalem Speicher holen

if (App.Current.Properties["BenutzerId"] is int benutzerId)

{

// Anzahl aus der TextBox auslesen (pro Zeile)

int anzahl = 1;

foreach (var item in lvErgebnisse.Items)

{

if (item == selectedSet)

{

var container = lvErgebnisse.ItemContainerGenerator.ContainerFromItem(item) as ListViewItem;

if (container != null)

{

var textBoxes = FindVisualChildren<TextBox>(container);

foreach (var tb in textBoxes)

{

if (int.TryParse(tb.Text, out int parsedAnzahl) && parsedAnzahl > 0)

{

anzahl = parsedAnzahl;

}

}

}

}

}

// BenutzerSet erstellen oder erhöhen

var vorhandenesBenutzerSet = db.BenutzerSets.FirstOrDefault(bs => bs.BenutzerId == benutzerId && bs.SetId == setInDb.SetId);

if (vorhandenesBenutzerSet == null)

{

var neuesBenutzerSet = new BenutzerSet

{

BenutzerId = benutzerId,

SetId = setInDb.SetId,

Kaufdatum = DateTime.Now,

Notizen = "",

GezahlterPreis = setInDb.PreisUVP,

Anzahl = anzahl

};

db.BenutzerSets.Add(neuesBenutzerSet);

}

else

{

vorhandenesBenutzerSet.Anzahl += anzahl;

}

db.SaveChanges();

MessageBox.Show($"Set {setInDb.Name} wurde deiner Sammlung hinzugefügt!");

}

else

{

MessageBox.Show("Bitte zuerst einloggen, um Sets zu speichern.");

}

}

}

}

Diese Methode wird aufgerufen, wenn man in der Trefferliste auf den „Hinzufügen“-Button klickt also wenn ein bestimmtes LEGO-Set zur eigenen Sammlung gespeichert werden soll.

Zuerst wird geprüft, welches Set überhaupt gemeint ist (über das Tag-Attribut des Buttons). Dann wird eine Verbindung zur Datenbank aufgebaut.

Falls das ausgewählte Set noch nicht in der lokalen Datenbank existiert, wird zuerst der Preis über die Rebrickable-API geladen und das Set wird neu in die Datenbank eingetragen. Das passiert also nur einmal pro Set.

Anschließend wird geschaut, ob der Nutzer eingeloggt ist, die Benutzer-ID wird dabei aus dem globalen Speicher geholt (App.Current.Properties). Wenn kein Benutzer angemeldet ist, gibt’s eine Meldung, dass das so nicht funktioniert.

Jetzt kommt der etwas aufwendigere Teil: Die Methode versucht herauszufinden, wie oft der Nutzer das Set hinzufügen will. Dafür wird in der Trefferliste (ListView) nach der passenden TextBox gesucht, die die Anzahl enthält. Falls ein gültiger Wert drinsteht, wird dieser verwendet, ansonsten bleibt es bei 1.

Dann wird geprüft, ob der Nutzer dieses Set schon in seiner Sammlung hat. Falls nicht, wird ein neuer Eintrag erstellt mit aktuellem Datum, Preis und Anzahl. Falls das Set schon da ist, wird einfach nur die Anzahl erhöht.

Zum Schluss wird alles gespeichert und es gibt eine kleine Erfolgsmeldung, dass das Set jetzt in der Sammlung ist.

#### Datenmodell-Konfiguration

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<LegoSet>()

.HasKey(x => x.SetId); // <<<< Hier explizit sagen: SetId ist der Primärschlüssel

modelBuilder.Entity<BenutzerSet>()

.HasKey(bs => new { bs.BenutzerId, bs.SetId });

modelBuilder.Entity<BenutzerSet>()

.HasOne(bs => bs.Benutzer)

.WithMany(b => b.BenutzerSets)

.HasForeignKey(bs => bs.BenutzerId);

modelBuilder.Entity<BenutzerSet>()

.HasOne(bs => bs.Set)

.WithMany(s => s.BenutzerSets)

.HasForeignKey(bs => bs.SetId);

}

public void EnsureDatabase()

{

this.Database.EnsureCreated();

}

Wie im Pflichten-/Lastenheft vorgesehen, kommt SQLite als eingebettete Datenbank zum Einsatz. Diese ermöglicht eine einfache Verteilung der Anwendung ohne separate Datenbankinstallation und speichert alle Benutzer- und Setdaten dauerhaft lokal in einer .db-Datei.

* OnModelCreating wird genutzt, um die Struktur der Datenbank beim Start zu definieren.
* SetId wird als Primärschlüssel für die Tabelle LegoSet festgelegt.
* Für BenutzerSet wird ein zusammengesetzter Primärschlüssel aus BenutzerId und SetId definiert.
* Ein BenutzerSet gehört zu genau einem Benutzer, ein Benutzer kann mehrere BenutzerSet-Einträge haben (1:n-Beziehung).
* Ein LegoSet kann ebenfalls in mehreren BenutzerSet-Einträgen vorkommen (1:n-Beziehung).
* Die Beziehungen werden über Foreign Keys abgebildet.
* EnsureDatabase prüft beim Start, ob die Datenbank existiert, und erstellt sie bei Bedarf automatisch.
* Erleichtert den Entwicklungsprozess, da keine manuelle DB-Erstellung notwendig ist.

### Beschreibung der Schnittstellen (API)

Für die Datenerweiterung nutzt das Programm die **Rebrickable API** – eine externe Schnittstelle, über die Informationen zu offiziellen LEGO-Sets abgefragt werden können.  
Die API liefert alle relevanten Set-Daten wie Name, Nummer, Thema, Erscheinungsjahr und – sofern verfügbar – den UVP-Preis.

Die Schnittstelle wird im Programm über die statische Klasse RebrickableService verwendet.  
Der Zugriff erfolgt über HTTPS-Anfragen mit einem persönlichen API-Key, den der Nutzer beim ersten Programmstart eingibt.  
Die erhaltenen JSON-Daten werden mithilfe von System.Text.Json deserialisiert und in eigene C#-Modelle (RebrickableSet, RebrickableTheme) überführt.

**Verwendete Endpunkte:**

* GET /sets/?search={suchbegriff}  
  → Sucht LEGO-Sets anhand von Name, Nummer oder Thema.
* GET /sets/?theme\_id={id}&page={page}  
  → Lädt alle Sets eines bestimmten Themas – wichtig für die Vorab-Ladefunktion.
* GET /sets/{set\_num}/  
  → Liefert Detailinformationen zu einem bestimmten Set (z. B. UVP in USD).

**Verarbeitung im Code:**  
Der Zugriff erfolgt über die Klasse HttpClient, wobei der API-Key im Header übergeben wird:  
Authorization: key <API\_KEY>

Die empfangenen Daten werden in Listen von LegoSet-Objekten umgewandelt und bei Bedarf dauerhaft in der lokalen SQLite-Datenbank gespeichert.

### Abweichungen vom Pflichtenheft/Lastenheft

#### ER-Diagramm

Ein Bild, das Entwurf, Zeichnung, Diagramm, Lineart enthält.

KI-generierte Inhalte können fehlerhaft sein.

Im Laufe der praktischen Umsetzung wurde das ursprünglich im Pflichten-/Lastenheft entworfene ER-Diagramm überarbeitet und an die tatsächlichen Anforderungen und technischen Rahmenbedingungen angepasst. Ziel war es, ein realistisches, sauberes Datenmodell zu entwickeln, das sowohl manuelle Eingaben als auch API-basierte Datensätze zuverlässig abbilden kann.

Im ursprünglichen Modell wurde eine eher abstrakte Struktur verwendet, mit Entitäten wie „API Daten“ und „Inventar“. Diese wurden im finalen Datenmodell ersetzt durch die konkreteren Entitäten LegoSet und BenutzerSet. Dabei ergaben sich folgende wichtige Änderungen:

Die Entität „API Daten“ wurde zu LegoSet, da dort nicht nur API-Daten gespeichert werden, sondern auch manuell erfasste Sets.

Die Entität „Inventar“ wurde durch BenutzerSet ersetzt. Diese neue Entität verbindet Benutzer und LegoSet miteinander, enthält jedoch zusätzlich wichtige Informationen wie Anzahl, GezahlterPreis, Kaufdatum und Notizen.

Die Beziehungen wurden präzisiert: Ein LegoSet „hat“ viele BenutzerSet-Einträge, ein Benutzer „besitzt“ viele BenutzerSet-Einträge. Damit wurde klar abgebildet, dass mehrere Benutzer dasselbe Set besitzen können – mit jeweils eigenen Informationen.

Attribute wie Teil-Anzahl, Genre oder Preis pro Stück wurden verworfen, da sie im praktischen Einsatz keine Relevanz hatten bzw. nicht über die API bereitgestellt wurden.

Zusätzlich wurde ein Sicherheitscode beim Benutzer eingeführt, um ein Zurücksetzen des Passworts zu ermöglichen. Auch dies war im ursprünglichen Modell noch nicht vorgesehen.

Durch diese Anpassungen ist ein praxisnahes, technisch sauberes Datenmodell entstanden, das sowohl funktionale als auch sicherheitsrelevante Anforderungen zuverlässig erfüllt.

#### Weitere Änderungen gegenüber dem Pflichten-/Lastenheft

**1. Benutzerverwaltung erweitert**

* **Sicherheitscode-Funktion** wurde ergänzt, damit Benutzer ihr Passwort zurücksetzen können.
* Dies war im ursprünglichen Lastenheft nicht vorgesehen, erhöht aber die Sicherheit und Benutzerfreundlichkeit.

**2. Keine separate Inventar-Tabelle**

* Im ursprünglichen Modell war Inventar eine eigene Entität.
* Diese wurde vollständig durch die Verknüpfung BenutzerSet ersetzt, wodurch eine klarere und relationale Struktur entstanden ist.

**3. Offline-Funktion durch „Thema vorladen“**

* Ursprünglich war nur die API-Suche geplant.
* Durch das **Thema-Vorladen** können Nutzer nun auch Sets lokal speichern und später offline durchsuchen.

**4. Verzicht auf überflüssige Felder**

* Felder wie „Teil-Anzahl“, „Genre“ oder „Preis pro Stück“ aus der ursprünglichen Planung wurden entfernt, da sie:
  + über die API nicht konsistent vorhanden waren
  + für die Nutzung keinen praktischen Mehrwert geboten hätten

**5. API-Key-System eingeführt**

* Um die Nutzung transparent und steuerbar zu machen, wird beim ersten Start ein **API-Key** abgefragt und gespeichert.
* Dies war nicht im Pflichten-/Lastenheft enthalten, ist aber in der Praxis notwendig, da Rebrickable eine Authentifizierung verlangt.

**6. Testdaten beim ersten Start**

* In der Umsetzung wird beim ersten Start ein Benutzer „Test“ mit mehreren Beispielsets erstellt.
* Diese Daten dienen zur Demonstration und zum Testen, waren aber ursprünglich nicht Teil des Pflichtenhefts.

## Tests