# Dokumentation

Für SharpTodo - Projekt von Benjamin Brandtner und Lukas Dürer  
Am Beruflichen Schulzentrum für Wirtschaft und Datenverarbeitung für das Fach C-Sharp  
Lehrkraft: Katrin Stremme  
Abgabe: 27.06.2019

## Zweck des Programmes

[Habitica.com](https://habitica.com/) ist eine Online-Zeitmanagement-Anwendung, auf der man unter anderem eine Todoliste anlegen kann, tägliche Verpflichtungen (*Dailies*) und gute und schlechte Gewohnheiten (*Habits*) tracken kann. Habitica nutzt verschiedene Mechaniken aus Rollenspielen, wie z.B. einem Charakter der aufleveln oder Schaden nehmen kann, um den Nutzer spielerisch zur Erledigung seiner Todos anzutreiben und für gute Gewohnheiten zu belohnen.  
Habitica stellt eine ausführliche API[[1]](#footnote-1) zur Verfügung, mit der man 3rd Party Anwendungen entwickeln kann, die beinahe alles können was man auch auf der Webseite machen kann.  
Wir haben uns nur einen kleinen Ausschnitt der Funktionalität von Habitica hergenommen: die Todoliste.  
Mit *SharpTodo* kann man alle auf Habitica angelegten Todos abfragen, Änderungen machen, abhaken und neue anlegen. Das Ziel war es, eine Oberfläche zu erschaffen, die schneller und kompakter ist als die auf der Webseite.

## Frontend

### View

#### MainWindow.xaml

#### In diesem .xaml-Dokument befindet sich die Hauptbenutzeröberfläche von *SharpTodo*, in der man Todos vom Server laden, erstellen, ändern und löschen kann. Desweiteren kann über den „Options“-Knopf in der Menüleiste das Optionsmenü aufgerufen werden.

#### Erwähnenswert ist die Verwendung eines *DataTemplates* für die in der ListBox aufgeführten Todos. Anstatt das standardmäßige Verhalten der ListBox zu nutzen, die den String anzeigt, der von den *ToString()-*Methoden der enthaltenen Elemente ausgegeben wird, kann man mittels DataTemplates xaml-Code zur Formatierung der Elemente in der ListBox einsetzen. In diesem Programm wird jedes Todo in der ListBox durch eine CheckBox und einen TextbBlock dargestellt. Der TextBlock zeigt den Titel des Todos und die Checkbox spiegelt die Completed-Property wieder und kann auch benutzt werden, um ein Todo als abgeschlossen oder noch offen zu markieren.

#### Dieses .xaml-Dokument sowie Options.xaml sind nach dem MVVM-Pattern entworfen, d.h. die xaml-Dokumente enthalten nur die Elemente der Benutzeroberflächen. Die Funktionalität wird in der jeweiligen ViewModel-Klasse realisiert, mit der das .xaml-Dokument über DataBindings und CommandBindings verbunden ist.

#### Die einzige Ausnahme von der Einhaltung des MVVM-Patterns stellt die Funktionalität des Buttons dar, der das Optionsmenü öffnet. Die Funktionalität dieses Buttons ist in der Code-Behind-Datei von MainWindow.xaml kodiert, da die MVVM-konforme Implementierung dieser Funktionalität unter Verwendung von Frameworks wie MVVMlight[[2]](#footnote-2) oder PRISM[[3]](#footnote-3) angesichts des geringen Umfangs dieses Projektes unnötig aufwändig wäre.

#### Options.xaml

#### Dieses Fenster übernimmt die Aufgabe, die für die Benutzung des Programms nötigen Informationen, die User-ID und den API-Schlüssel, vom Benutzer abzufragen und abzuspeichern. Des Weiteren besteht hier die Möglichkeit, API-Schlüssel und User-ID vom Habitica-Server auf Richtigkeit überprüfen zu lassen.

### Viewmodel

#### UserCommand

Diese Klasse implementiert das ICommand-Interface und wird benötigt, um den in der View angelegten Steuerelementen im ViewModel Funktionalitäten zuweisen zu können.

#### VMHabiticaTodo

Diese Klasse kapselt die vom Backend übergebenen HabiticaTodos und gibt über ihre Getter und Setter Zugriff auf die Attribute der enthaltenen Todos.

#### VMMainWindow

#### Diese Klasse enthält die Funktionalität des Hauptfensters der Benutzeroberfläche, sendet Anfragen an das Backend und stellt Daten für die Bindings der Benutzeroberfläche bereit.

VMMainWindow implementiert das INotifyPropertyChanged-Interface, um die Benutzeroberfläche über Änderungen an den gebundenen Daten benachrichtigen zu können, damit die Benutzeroberfläche sich aktualisieren kann. Benötigt wird dies für die Ausgabe der Fehlermeldungen, da die Liste der Todos als ObservableCollection implementiert ist und ObservableCollections diese Funktionalität von sich aus mitbringen.

#### VMOptions

Ähnlich wie VMMainWindow enthält VMOptions die Funktionalität des Optionsmenüs und stellt die Daten zur Anzeige auf der Benutzeroberfläche bereit. Auch VMOptions implementiert die INotifyPropertyChanged-Schnittstelle, um die Ausgabe von Fehlermeldungen zu ermöglichen.

## Kommunikation Frontend → Backend

## Die Klasse VMMainWindow nutzt aus dem Backend den HabiticaClient, um HabiticaTodos zu laden, erstellen, usw. In VMOptions wird der ConfigManager wird benutzt, um die Konfigurationsdatei zu lesen und zu speichern. Beide Klassen reagieren auf Exceptions die im Backend geworfen werden können mit der Ausgabe einer Fehlermeldung.

## Backend

### HabiticaClient

Im Backend ist der HabiticaClient, der von HttpClient erbt, verantwortlich, verschiedene HttpRequests an die Habitica API Server zu senden.  
Da ein Request eine Weile dauert, aber in dieser Zeit die Gui und andere Programmteile nicht stillstehen sollen, werden die Requests asynchron abgeschickt.

Für jeden Request an die API müssen bestimmte Header mitgeschickt werden, die einen auf Habitica registrierten Benutzer identifizieren.  
Die userId und das apiToken stehen in einem Config Objekt, dass entweder beim Instanziieren mitgegeben oder durch den ConfigManager geladen wird. Sind diese Einträge leer, zum Beispiel beim ersten Starten des Programmes, wird eine NoCredentialsException geworfen.

Für die Verwendung der Habitica-API halten wir uns natürlich an die vorgegebenen Regeln[[4]](#footnote-4), wie keine Endlosschleifen, einen Header der das Tool und den Author indentifiziert, offener Sourcecoude, etc.

#### Kurzer Einblick in Asynchrone Programmierung mit Tasks [[5]](#footnote-5)

Durch Task Objekte und die Schlüsselwörter async und await wird die asynchrone Programmierung vereinfacht. Sie ersetzen das komplizierte manuelle Arbeiten mit Threads.

Ermöglicht wird das durch die Task Parallel Library (TPL). Sie ist der bevorzugte Weg, in .NET asynchron zu programmieren.

Innerhalb einer Methode, die selbst asynchron ist, kann auf eine weitere asynchrone Methode gewartet werden, ohne dass das Programm blockiert. Mit dem Schlüsselwort await wird die Kontrolle an den Aufrufer der Methode zurückgegeben.

async void DownloadButton\_Click()

{

//Durch das await wird nach dem Aufruf von GetStringAsync die Kontrolle an z.B. die GUI zurückgegeben.

//Ist GetStringAsync fertig ausgeführt läuft diese Methode weiter.

String s = await myClient.GetStringAsync(url);

doSomething(s);

}

### ConfigManager

Ist dafür verantwortlich, Konfigurationseinstellungen im JSON Format in einer config.ini Datei zu speichern und sie auszulesen.

Das Config-Objekt ist ein dynamisches Objekt, und kann damit Properties mit beliebigem Namen und beliebigen Datentypen annehmen. Dies wurde gewählt, damit bei der weiteren Entwicklung schnell und simpel weitere Einstellungen (z.B. Ansichtseinstellungen, Keybindings) implementiert und gespeichert werden können.

### HabiticaSerializer

Da die Habitica API Daten im JSON Format erwartet und zurückschickt, nutzt der Client den HabiticaSerializer um HabitcaTodos zu serialisieren und die Server-Antworten zu deserialisieren.  
Der HabiticaSerializer nutzt das Framework *Json.NET*.  
Die meisten Methoden kriegen einen JSON-String übergeben, den parsen sie, zum Teil mit der Methode *ParseResponseData()*, und geben ein oder mehrere *HabiticaTodos* zurück. Die Methode *ParseResponseData()* reagiert folgendermaßen auf Fehlermeldungen vom Server:

Sind der eingegebene userId oder apiToken falsch, antwortet der Server, dass ein solcher Nutzer nicht existiert. Daraufhin wird eine *WrongCredentialsException* geworfen.  
Wenn etwas mit dem Request nicht stimmt, weil zum Beispiel die JSON-Daten die falsche Form haben, steht das in der Serverantwort. Daraufhin wird eine *UnsuccessfulException* geworfen, deren Message genauere Informationen enthält. Dies sollte theoretisch nur während der Entwicklung, nicht im fertigen Programm passieren.

### HabiticaTodo

Die Klasse HabiticaTodo definiert selbst nur ein Attribut, die Checklist. Es wird noch nicht benutzt, da ChecklistItems in der API gesondert behandelt werden müssen, und dies im HabiticaClient noch nicht implementiert ist. Sie erbt den Rest von der abstrakten Klasse Todo.  
Sie hat auch eine interne Klasse, mit der Prioritäten eines Todos dargestellt werden können. Diese wird noch nicht genutzt.

### Todo

Um vom JSON der API in ein Objekt der Klasse HabiticaTodo zu deserialisieren, müssen die Properties dieser Klasse um Attribute ergänzt werden, die dem JsonSerializer sagen, welche Namen die Properties im JSON haben sollen.

[JsonProperty("text")]

public string Title { get => title; set => title = value; }

Ansonsten ist diese abstrakte Klasse eine reine Datenklasse mit Feldern, Properties und einer *ToString()* Methode.

## Erweiterbarkeit des Backends

Zurzeit braucht man, um SharpTodo zu nutzen einen Account auf Habitica. Alle Todos die man anlegt werden auf diesem Account gespeichert.  
Es wäre recht simpel, stattdessen lokale Speicherung zu implementieren. Ich würde dafür eine Klasse LocalClient anlegen, die dieselben Methoden hat wie der HabiticaClient (hier bietet sich auch ein Interface TodoClient an). Der LocalClient speichert dann Todos z.B. in einer lokalen Datenbank oder JSON-Datei ab.

Auf der Oberfläche würde ich eine Einstellung hinzufügen, mit der man zwischen dem HabitcaClient und dem LocalClient wechseln kann.

1. <https://habitica.com/apidoc/> [↑](#footnote-ref-1)
2. Erhältlich unter <http://www.mvvmlight.net/> [↑](#footnote-ref-2)
3. Erhältlich unter <http://prismlibrary.github.io/> [↑](#footnote-ref-3)
4. <https://habitica.fandom.com/wiki/Guidance_for_Comrades#API_Server_Calls> [↑](#footnote-ref-4)
5. Siehe: <https://docs.microsoft.com/en-us/dotnet/csharp/async> ,

   <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming> [↑](#footnote-ref-5)