|  |
| --- |
|  |
| FlightSimBackend |
| Node.JS Webservice und WPF-Administrationstool für FlightSim-Daten |

|  |
| --- |
| Patrick Müller  11.7.2016 |

Inhaltsverzeichnis

[Zielsetzung 2](#_Toc457499628)

[Webservice – SOAP oder REST 2](#_Toc457499629)

[REST 2](#_Toc457499630)

[SOAP 2](#_Toc457499631)

[Fazit 3](#_Toc457499632)

[Webservice – Technologien 4](#_Toc457499633)

[Webservice – Aufbau des Interface 4](#_Toc457499634)

[/User 4](#_Toc457499635)

[/State 5](#_Toc457499636)

[Webservice – Codestruktur 5](#_Toc457499637)

[Verwaltungsprogramm 6](#_Toc457499638)

[Allgemeiner UI-Aufbau 6](#_Toc457499639)

[Userverwaltung 6](#_Toc457499640)

[Create new User-Window 6](#_Toc457499641)

[GameState-Verwaltung 7](#_Toc457499642)

[Create new GameState-Window 7](#_Toc457499643)

# Zielsetzung

Ziel dieses Projekts ist es, einen Webservice zu entwerfen und zu entwickeln, der in der Lage ist, die Spielstände der FlightSimulator-App online zu speichern und von dort jederzeit abrufbar zu machen.

Des Weiteren muss die FlightSimulator-App um die nötigen Funktionen erweitert werden, sodass eine Verbindung mit dem Webservice hergestellt werden kann, der Nutzer bzw. das Gerät sich identifiziert, und so individuelle Spielstände speichern und laden kann.

Um die gespeicherten Daten verwalten zu können, soll ausserdem noch eine WPF-Applikation entwickelt werden, mit deren Hilfe Daten hinzugefügt, gelöscht und bearbeitet werden können.

# Webservice – SOAP oder REST

## REST

Die wichtigsten Merkmale von **REST** (**Re***presentational*-**S***tate*-**T***ransfer*) sind

* Performance
* Skalierbarkeit
* Einfachheit der Schnittstellen
* Einfache Modifizierbarkeit
* Verlässlichkeit

In Anbetracht des relativ kleinen Umfangs der geplanten Applikation sind Performance und Skalierbarkeit nicht die oberste Priorität. Einfache Modifizierbarkeit hingegen ist ein sehr starker Pluspunkt, da sich die Features der FlightSimulator-App ständig ändern können oder gar neue hinzukommen. Somit ist es auch sehr wichtig, dass eine simple Schnittstelle vorhanden ist, die bei Änderung der Anforderungen keine große Anpassung erfordert. Dass der Service verlässlich sein muss steht außer Frage, da ohne die gespeicherten Spielstände ein Nutzer immer von vorne beginnen müsste, was nicht zumutbar ist.

**REST** ist über eine Vielzahl von Protokollen nutzbar, z.B. HTTP/HTTPS, FTP, OData. Das Format der Daten ist den Anforderungen anpassbar, so lassen sich die Daten in XML, JSON oder sogar Plaintext-Format übermitteln. Meist werden bei einer **REST-**API die vier CRUD-Methoden Create, Refresh, Update und Delete implementiert, z.B. beim HTTP-Protokoll in Form von POST, GET, PUT, DELETE.

## SOAP

**SOAP** (**S***imple*-**O***bject*-**A***ccess*-**P***rotocol*) zeichnet sich durch folgende Merkmale aus:

* Sehr hohe Sicherheit (WS-Security, WS-ReliableMessaging)
* Sprach-, Plattform- und Transportprotokollunabhängig
* Standardisiert
* Schnittstelle liefert Beschreibung der Daten

Das hohe Maß an Sicherheit ist bei sensiblen Daten wie z.B. Bankdaten ein sehr wichtiger Faktor. In der FlightSimulator-App werden jedoch keine sensiblen Daten von Nutzern verarbeitet oder gespeichert. Die Unabhängigkeit des Protokolls von der Programmiersprache, Plattform und dem Transportprotokoll ist sehr nützlich, wenn eine Vielzahl verschiedener Clients auf die Schnittstellen des WebService zugreift. Da das **SOAP-**Protokoll standardisiert ist und die Schnittstellen auch Informationen über die von ihnen gelieferten Daten zur Verfügung stellen, wie z.B. Datentypen, Namen der Objekte und Eigenschaften, ist es sehr einfach den Großteil der Arbeit nur einmal im Webservice zu erledigen, statt es in jedem einzelnen Client implementieren zu müssen.

Auch **SOAP** ist über eine Vielzahl von Protokollen nutzbar, das Format der Daten ist allerdings auf XML festgelegt. Zu jeder Schnittstelle muss auch eine standardisierte Beschreibung vorliegen, in der die Daten und verfügbare Operationen aufgeführt sind

## Fazit

Nach abwägen der Vor- und Nachteile der zwei Möglichkeiten, habe ich mich dazu entschieden, den Webservice als eine **REST-**API zu implementieren.

Durch ständige Veränderung in der FlightSimulator-App, ist es erheblich einfacher mit den Schnittstellen einer **REST-**API zu arbeiten, da diese leicht zu modifizieren und erweitern sind. Bei **SOAP** hingegen müsste bei jeder Änderung der Schnittstelle auch die dazugehörigen Beschreibungen geändert werden.

Bezüglich der Sicherheit des Webservice ist die SSL-Verschlüsselung einer HTTPS-Verbindung ausreichend, die erweiterten Features von SOAP sind überflüssig.

Für den Zweck ein Speichersystem für Spielstände zu implementieren sind die HTTP-Schlüsselmethoden

* POST (Speichern eines neuen Spielstandes)
* PUT (Vorhandenen Spielstand überschreiben)
* GET (Laden eines vorhandenen Spielstandes)
* DELETE (Löschen eines vorhandenen Spielstandes)

hervorragend geeignet, und diese lassen sich mit einer **REST-**API sehr leicht abbilden.

# Webservice – Technologien

Als Technologie wird für diesen Webservice Node.js, express.js und SQLite genutzt.

Node.js ist die zugrundeliegende JavaScript-Engine auf welcher der Code ausgeführt werden kann.

Express.js ist ein Web-Application Framework mit dessen Hilfe die Routen der Webservice-REST-API erstellt werden.

Zum Speichern der Nutzerinformationen und Spielstände wird SQLite als Datenbank genutzt. Zur einfachen Ansteuerung der Datenbank kommt Sequelize zum einsatz.

Die Entwicklung findet vorerst lokal statt, zu einem späteren Zeitpunkt soll der Service allerdings auf einen im Internet zugänglichen Server betrieben werden.

# Webservice – Aufbau des Interface

Zum Ansprechen des Service sind zunächst zwei Basisrouten nötig

* /User
  + Für alle Methoden die sich auf User beziehen
* /State
  + Für alle Methoden die sich auf Spielstände (game**state**s) beziehen

## /User

Auf der Userroute werden folgende REST-Methoden zur Verfügung gestellt

* /User
  + GET
    - Gibt ein JSON-Objekt zurück, das eine Liste aller User enthält
  + POST
    - Erstellt einen neuen User mit den im request-body mitgegebenen Informationen Username und Passwort
* /User/:UserId
  + GET
    - Gibt ein JSON Objekt zurück, das Name und Passwort des Users mit der UserId enthält, die in der Route angegeben wurde
  + PUT
    - Updatet die Daten des Users mit der in der Route angegebenen UserID, die neuen Daten werden im request-body mitgegeben
  + DELETE
    - Löscht den User mit der angegebenen UserId unwiderruflich
* /User/:UserId/state
  + GET
    - Gibt ein JSON Objekt zurück, das eine Liste aller Gamestates enthält, die mit der mitgegebenen UserId verknüpft sind

## /State

Auf der Stateroute werden folgende REST-Methoden zur Verfügung gestellt

* /State
  + GET
    - Gibt ein JSON-Objekt zurück, das eine Liste aller Gamestates enthält
  + POST
    - Erstellt einen neuen Gamestate mit den im request-body mitgegebenen Informationen
* /State/:StateId
  + GET
    - Gibt ein JSON-Objekt zurück, das alle Informationen des Gamestates mit der StateId enthält, die in der Route angegeben wurde
  + PUT
    - Updatet die Daten des Gamestates mit der in der Route angegebenen StateId, die neuen Daten werden im request-body mitgegeben
  + DELETE
    - Löscht den Gamestate mit der angegebenen StateId unwiderruflich

# Webservice – Codestruktur

Die Codestruktur orientiert sich am Standardaufbau einer node.js/express Applikation

* bin/
  + *www*
* database/
  + *db.js*
  + *db.sqlite*
* routes/
  + *user.js*
  + *state.js*
* *app.js*
* *package.json*

Die *www*-Datei im bin Ordner ist der Einstiegspunkt des Programms, hier wird der Webserver erstellt der http-Anfragen empfängt. In der *app.js*-Datei wird express.js konfiguriert und die Routen aus den Dateien *user.js* und *state.js* importiert. In den *user.js* und *state.js* werden die einzelnen Routen und ihre Funktionalitäten definiert und implementiert. In der *db.js*-Datei wird die Datenbankverbindung konfiguriert und Modelle für User und Gamestates definiert. Die *db.sqlite*-Datei ist die Datenbank selbst. In der *package.json*-Datei werden die genutzten Packages für **npm** (node-package-manger) definiert und einige grundlegenden Informationen zur Applikation abgelegt.

# Verwaltungsprogramm

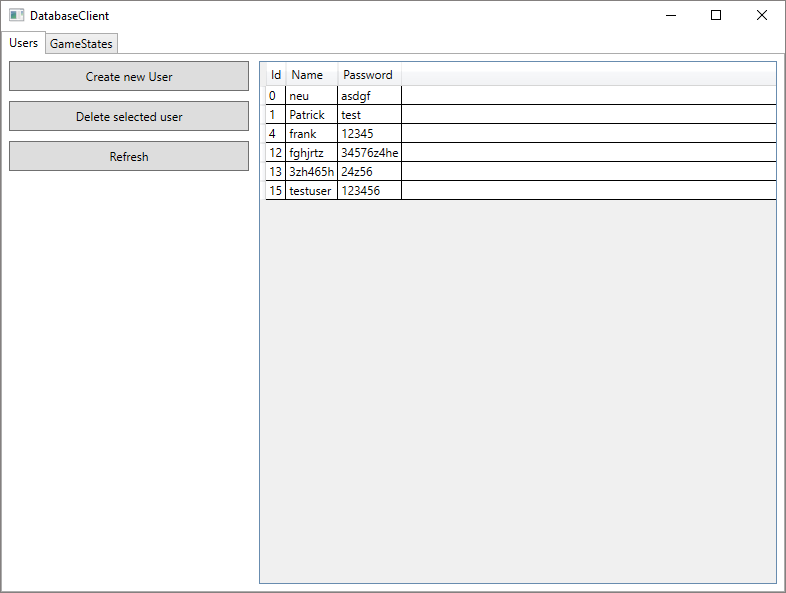
Zur Verwaltung der auf dem Server gespeicherten Daten soll eine WPF-Applikation entwickelt werden, mit der ein Administrator in der Lage ist neue Daten zu erstellen und vorhandene Daten zu bearbeiten oder zu löschen.

## Allgemeiner UI-Aufbau

Im MainWindow gibt es ein TabControl mit aktuell zwei TabItems, Users und Gamestates.

In den einzelnen TabItems sind jeweils eine UsersPage und eine GameStatesPage mittels eines Frames eingebunden.

## Userverwaltung



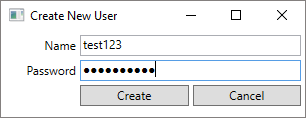
Die UsersPage besteht aus einem Grid das in zwei Columns eingeteilt ist. Die erste Column ist von statischer Weite, die zweite Column passt sich automatisch an die restliche Größe des Fensters an.

In der ersten Column sind 3 Buttons in einem StackPanel untergebracht.

* Create new User
  + Öffnet ein neues Fenster zum Erstellen eines neuen Users
* Delete selected User
  + Löscht den im rechten DataGrid ausgewählten User
* Refresh
  + Aktualisiert die Ansicht im DataGrid mit aktuellen Informationen vom Server

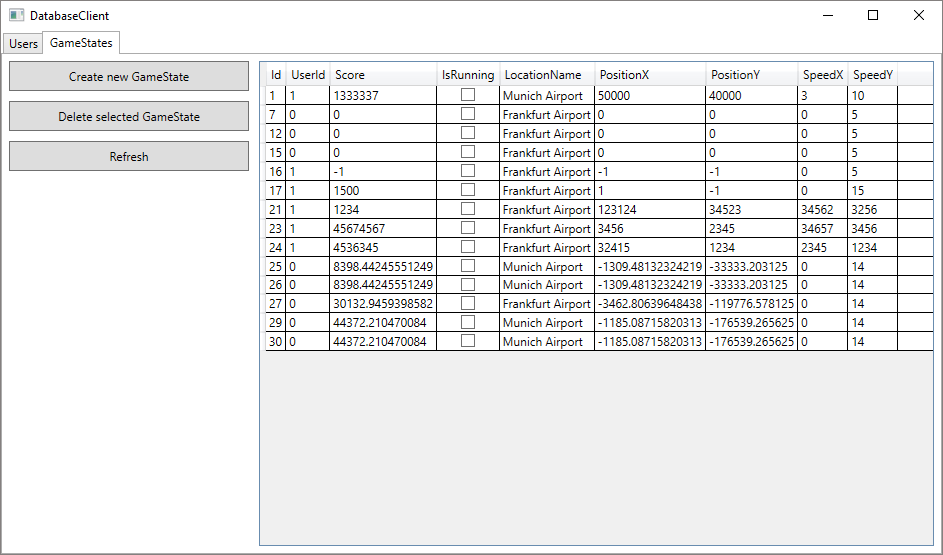
In der zweiten Column befindet sich ein DataGrid, das beim ersten Laden der UsersPage automatisch mit User-Daten vom Server befüllt wird. Änderungen im DataGrid werden beim Bestätigen übernommen und an den Server als Update geschickt. Die Id ist nicht editierbar

### Create new User-Window

Hier lässt sich der Name und das Passwort eines neuen Nutzers eingeben.

Beim klicken des Create Buttons wird direkt ein Request an den Server gesendet und anschließend das DataGrid automatisch aktualisiert um die neuen Daten anzuzeigen.

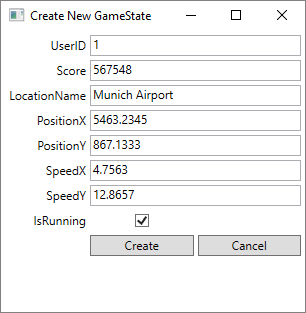
## GameState-Verwaltung



Die GameStatePage ist aufgebaut wie die UsersPage, abgesehen vom DataGrid und des Fensters zum Erstellen eines neuen Eintrags.

Die Columns des DataGrid werden aus der GameState-Klasse automatisch generiert. Auch hier ist die Id nicht editierbar und Änderungen werden automatisch per Update-request an den Server übermittelt.

### Create new GameState-Window

Hier lassen sich alle Daten eines GameStates eintragen. Der bool’sche Wert IsRunning ist über eine CheckBox einstellbar.

Beim klicken des Create Buttons wird direkt ein Request an den Server gesendet und anschließend das DataGrid auf der GameStatesPage automatisch aktualisiert um die neuen Daten anzuzeigen.