**Gambling-Webseite**

**Lars, Tilo, Atal, Alexander, Carl, Mina, Silas, Florian**

# Die Idee

Wir wollen als Gruppe eine funktionsfähige Glücksspielanwendung programmieren. Dabei soll es möglich sein, dass ein Benutzer einen Account erstellen und verifizieren kann. So soll er in der Lage sein, persönliche Informationen zu seinem Konto hinzuzufügen, darunter Name, Vorname, Geburtsdatum und Adresse. Zusätzlich soll ein Status gespeichert werden, ob dieser Benutzer verifiziert wurde. Das Geburtsdatum wird überprüft, um festzustellen, ob der Benutzer an der Glücksspielanwendung teilnehmen darf. Nur in Deutschland wohnende Menschen ab einem Alter von 18 Jahren dürfen teilnehmen.

Die Anwendung soll außerdem in der Lage sein, die Gewinne bzw. die Verluste eines Benutzers zu verwalten. Dazu hat der Benutzer in der Datenbank einen gespeicherten Kontostand, der speichert, wieviel Geld ihm zur Verfügung steht. Die Teilnahme an den Spielen bzw. die dafür notwendigen Transaktionen werden ebenfalls in der Datenbank in einer anderen Tabelle gespeichert, sodass wir unsere Gewinne und Verluste genau überprüfen können. Über eine grafische Oberfläche sollen zwei verschiedene Spiele, Blackjack und eine Slotmaschine, spielbar sein.

# Backend/Frontend

Über ein sicheres Backend sollen die Daten ohne Einfluss des Clients gespeichert werden. Das bedeutet, dass der Client sich mit dem Server verbindet und sich über seinen Account-Token verifiziert. Daraufhin setzt das Backend über den http-Header einen Cookie mit dem Token des Benutzerkontos auf dem Gerät. Dadurch wird gewährleistet, dass der Benutzer auch nach dem Neuladen oder Wechseln der Seite immernoch angemeldet ist.

# Slots/Blackjack - Spiele

## Idee

Das Slots-Spiel (auch bekannt als Spielautomat oder „Einarmiger Bandit“) soll auf einem festen Einsatz basieren, der vom Konto des Benutzers beim Spielstart in der Datenbank abgezogen wird. Dazu soll in der Tabelle Transaktionen eine neue Transaktion angelegt werden. Vorher soll überprüft werden, ob der Benutzer genügend Geld auf dem Konto hat. Der aktuelle Kontostand des Benutzers kann durch die Summe aller Transaktionen ermittelt werden. Daher ist es nicht notwendig, den Kontostand redundant in der **Kunden**-Tabelle zu speichern. Stattdessen kann der Kontostand jedes Mal aus der Summe der Transaktionen des jeweiligen Benutzers in der **Transaktionen**-Tabelle ausgelesen werden. Ansonsten könnte es zu Anomalien kommen, wenn der Kunden-Kontostand nicht mit der Transaktion rechtzeitig aktualisiert werden kann.

Das Gleiche gilt für das Blackjackspiel. Hier ist allerdings der Einsatz variabel und muss, wie eben beschrieben, mit der Datenbank abgeglichen werden. Danach wird im Backend separat vom Client die gesamte Spiel-Logik berechnet. Dazu gehören unter anderem normale Spielfunktionen wie Karten ziehen, Split oder Double-Down. Dabei muss jeder Schritt im Spiel zwischen Backend und Frontend synchronisiert werden. Dazu kommt noch eine graphische Oberfläche, die es dem Benutzer einfach macht, die oben genannten Aktionen auszuführen.

Zu Beginn des Spiels öffnet der Benutzer die entsprechende Frontend-Seite. Über einen Cookie, der auf dem Client-Browser gesetzt wird, lässt sich erkennen, welchen Benutzer-Account der Benutzer verwendet. Es wird jedoch nicht direkt die Benutzer-ID im Cookie gespeichert, da diese manipuliert werden könnte. Stattdessen wird ein Token gespeichert, der als Referenz zum Nutzer-Account dient. Nach dem Spiel wird dieses Token durch die Benutzer-ID aufgelöst, sofern der Benutzer angemeldet ist. Anschließend wird eine neue Speicherungstransaktion (?) eingefügt. Die Kommunikation zwischen dem Spiel-Frontend und dem Skript im Backend erfolgt über einzelne HTTP-Anfragen. Wir haben uns jedoch gegen die Verwendung von WebSockets entschieden, da sie relativ viel Bandbreite und Serverleistung benötigen. Hier muss die TCP-Verbindung durchgehend aufrechterhalten werden. Bei den einzelnen HTTP-Anfragen wird die Authentifizierung durch den Benutzer-Token sichergestellt. Die Blackjack-Anwendung verwendet den Port 8080, während die Slots-Anwendung stattdessen auf den Port 8081 zurückgreift. Wir benötigen zwei verschiedene Endpunkt-URLs, über die die wesentliche Kommunikation mit dem Benutzer läuft:

/balance: Dieser Endpunkt wird vom Frontend genutzt, um den aktuellen Kontostand des Benutzers abzufragen. Hierbei erfolgt die Authentifizierung über das Benutzer-Token. Diese Abfrage erstellt einen zufälligen Wert für jedes benötigte Rad der Slot-Maschine.

/play: Dieser Endpunkt dient dazu, eine neue Transaktion zu erstellen, z. B. wenn der Benutzer seinen Einsatz für das Slot-Spiel tätigt.

Auf das Anfordern der TCP-Anfragen wird das Backend dann mit dem Ergebnis gespeichert. So wird gleichzeitig die entsprechende Transaktion in den Datenbank-Tabellen-Transaktionen. In den http Headern wird ein Cookie gesetzt, sodass die Benutzeranmeldung erhalten bleibt.

# AdminPanel

### Idee:

Die Klasse AdminPanel stellt eine grafische Benutzeroberfläche (GUI) zur Verfügung, mit der sich Administratoren mit einer Datenbank verbinden und SQL-Abfragen ausführen können. Die abgerufenen Daten werden in einer Tabelle angezeigt, die bearbeitet werden kann. Änderungen an der Tabelle werden nach Bestätigung in der Datenbank aktualisiert. Abfragen und Änderungen sollen durch Dialoge erfragt werden. Die GUI benutzt Buttons, Textfelder und Eingabefelder für die Dialoge.

### Attribute

#### GUI-Elemente

* frame – Ein JFrame-Fenster, das die gesamte Benutzeroberfläche enthält.
* panel – Das Haupt-Panel, auf dem alle Komponenten platziert werden.
* rightButtonPanel – Ein Panel, das Schaltflächen am unteren Rand der Oberfläche enthält.
* scrollPane – Ein JScrollPane, das die Tabelle enthält, um das Scrollen bei vielen Daten zu ermöglichen.
* table – Eine JTable, die die abgerufenen Daten der SQL-Abfrage anzeigt und bearbeitet werden kann.
* queryField – Ein JTextField, in dem der Benutzer SQL-Abfragen eingibt.
* pwField – Ein JPasswordField, in dem das Datenbankpasswort eingegeben wird.
* label – Eine JLabel, die den aktuellen Status oder Anweisungen anzeigt.
* newQuery – Ein JButton, mit dem eine neue SQL-Abfrage gestartet werden kann.
* exit – Ein JButton, um das Fenster zu schließen (aktuell nicht implementiert).
* submitChange – Ein JButton, um Änderungen an der Datenbank zu bestätigen.
* cancelChange – Ein JButton, um Änderungen abzulehnen.

#### Datenbank-Attribute

* query – Ein String, der die aktuelle SQL-Abfrage speichert.
* height – Die Anzahl der Zeilen in der abgerufenen Tabelle.
* width – Die Anzahl der Spalten in der abgerufenen Tabelle.
* data – Ein zweidimensionales String-Array zur Speicherung der abgerufenen Daten.
* columnNames – Ein String-Array zur Speicherung der Spaltennamen.
* clientDB – Ein Connection-Objekt, das die Verbindung zur Datenbank speichert.
* id – Die ID der aktuell ausgewählten Zeile für Updates.
* stmt – Ein Statement-Objekt zur Ausführung von SQL-Befehlen.

#### Bildschirmgrößen

* screenWidth – Die Bildschirmbreite zur dynamischen Anpassung der GUI-Elemente.
* screenHeight – Die Bildschirmhöhe zur dynamischen Anpassung der GUI-Elemente.

### Konstruktor

public AdminPanel()

Beim Erstellen eines AdminPanel-Objekts wird das GUI-Fenster initialisiert und angezeigt. Der Aufbau erfolgt wie folgt:

1. Das JFrame-Fenster wird maximiert und in den Vollbildmodus gesetzt.
2. Farben und Schriftarten für die GUI-Elemente werden definiert.
3. Das panel wird mit einem GridBagLayout eingerichtet.
4. Ein Passwortfeld (pwField) wird platziert, um die Authentifizierung mit der Datenbank zu ermöglichen.
5. Ein ActionListener für das pwField wird hinzugefügt, um nach Eingabe eines Passworts die Datenbankverbindung herzustellen.
6. Nach erfolgreicher Anmeldung wird das queryField angezeigt, in das der Benutzer SQL-Befehle eingeben kann.
7. Die GUI aktualisiert sich dynamisch je nach Status der Datenbankabfrage und Benutzerinteraktionen.

### Methoden

#### getConnection(String pPW)

public Connection getConnection(String pPW)

* Stellt eine Verbindung zur MariaDB-Datenbank her.
* Verbindet sich mit dem Server db.ontubs.de und nutzt den Benutzer carl.
* Gibt ein Connection-Objekt zurück, wenn die Verbindung erfolgreich ist, ansonsten null.

#### updateDatabase(DefaultTableModel tableModel, String[] columnNames, ResultSetMetaData md)

public void updateDatabase(DefaultTableModel tableModel, String[] columnNames, ResultSetMetaData md)

* Fügt einen TableModelListener hinzu, der Änderungen in der Tabelle überwacht.
* Sobald eine Zelle bearbeitet wird, ermittelt die Methode die zugehörige id der Zeile.
* Ein Bestätigungsdialog (JLabel + Buttons Yes / No) wird angezeigt.
* Falls der Nutzer bestätigt, wird ein SQL-Update-Befehl generiert und ausgeführt.
* Danach wird die Tabelle aktualisiert (reloadTable()).

#### reloadTable()

private void reloadTable()

* Entfernt alle aktuellen GUI-Komponenten.
* Erstellt die JTable neu und lädt die aktualisierten Daten.
* Stellt sicher, dass Farben und Formatierungen der Tabelle erhalten bleiben.
* Fügt den newQuery-Button erneut hinzu.
* Ruft panel.revalidate() und panel.repaint() auf, um die Änderungen anzuzeigen.

#### update()

public void update()

* Ruft frame.repaint() auf, um das Fenster neu zu zeichnen und Änderungen sofort anzuzeigen.
* Wird von einer while-Schleife in der main-Methode (nicht enthalten) regelmäßig aufgerufen.

## ActionListener

* **pwField.addActionListener**
  + Wird ausgelöst, wenn der Benutzer die Eingabetaste drückt.
  + Stellt eine Verbindung zur Datenbank her.
  + Falls die Verbindung erfolgreich ist, wird das queryField anstelle des pwField angezeigt.
  + Falls das Passwort falsch ist, wird eine Fehlermeldung angezeigt.
* **queryField.addActionListener**
  + Wird ausgelöst, wenn der Benutzer eine SQL-Abfrage eingibt und Enter drückt.
  + Führt die Abfrage aus und speichert die Ergebnisse in data[][].
  + Erstellt eine JTable mit den abgerufenen Daten.
* **newQuery.addActionListener**
  + Entfernt die aktuelle Tabelle und ermöglicht die Eingabe einer neuen SQL-Abfrage.
* **submitChange.addActionListener**
  + Bestätigt Änderungen in der Tabelle und aktualisiert die Datenbank.
* **cancelChange.addActionListener**
  + Verwirft Änderungen und lädt die ursprünglichen Daten erneut.