**Backend für Benutzer Verwaltung**

Die Klasse „user“ verwaltet alle Benutzer um ein an- und abmelden mit einem Benutzernamen und Passwort zu ermöglichen. Die Anmeldedaten werden in einer eigenen Tabelle „Users“ in der Datenbank (=DB) gespeichert. Diese hat folgende Attribute:

User\_ID (integer):

Diese Interne ID dient als Primärschlüssel um jeden Datensatz eindeutig zu identifizieren und wird automatisch inkrementell ab 1 um 1 hochgezählt. Das Limit an Datensätzen und damit an Benutzern beträgt hier bei einer Speicherkapazität von 8 Byte pro integer bei (28)8 -1 = 264 -1 Benutzern oder 1,8E10 Milliarden. Das Limit liegt also weit über der Anzahl der Bevölkerung von Deutschland (ca. 84 Millionen in 2024). Es würde sehr lange dauern bis alle IDs ausgeschöpft sind.

Login\_Name (text):

Speichert den Benutzernamen (=BN) der zum Anmelden benötigt wird und darf max. 32 Zeichen lang sein. Es könnte auch als Primärschlüssel dienen, da bei unserer Implementierung der BN eindeutig sein muss (keine 2 gleiche BNs). Das wäre aber unsauber da der BN vom Nutzer in Zukunft geändert werden könnte.

PW\_Hashed (text):

Speichert das Passwort als Hash für die Anmeldung

Salt (text):

Wird intern benötigt beim Anmelden und zum Hashen des Passwortes.

Rechte (integer)

Die Rechte auf die Patientenakten werden als Zahl gespeichert. Dabei entspricht die:

0 einen Fehler bzw. ungültig

1 Lese- und Schreibrechte (z.B. Arzt)

2 nur Lese- Rechte (z.B. Patient)

**Hashen des Passworts**

Um das Passwort sicher zu speichern, also gegen Hacker Angriffe zu schützen, wird es nicht als Klartext in der Datenbank hinterlegt. Es wurde ein SHA-256 Hash Algorithmus benutzt um das Passwort mit einem Salt in einen Hash umzuwandeln. Dieser wird in Hexadezimalzahlen in die Datenbank abspeichert. Anders als bei der Verschlüsselung kann dies nicht rückgängig gemacht werden.

Der Salt schützt gegen Rainbow Table Attacken, also eine Art pregenerierte Tabelle mit Hashen und ihren zugehörigen Passwörtern. Dies erschwert deutlich das Knacken des Passwortes, da für jedes Passwort eine Brut-Force Attacke von vorne starten müsste, da ein anderer Salt verwendet wird. Damit muss der Salt nicht zwingend geschützt sein. Zur Vereinfachung entspricht der Salt dem BN da dieser bereits eindeutig ist. Besser wäre aber ein zufälliger Wert der eindeutig ist bzw. nur einmal im Datensatz auftaucht.

Der Hash ist als hex mit 64 chars darstellbar aber speicher inneffizient (1 char ≙ mind. 1 Byte und 1 HEX ≙ 1 Nibble). Die 256 bezieht sich somit auf die Länge des Hashes (64 Hex Zahlen ≙ 32 Byte ≙ 256 Bit). Somit sollte das Passwort+Salt eine maximale Länge im Speicher von 32 Bytes haben, also 32 Zeichen. Es wurde einen einfache Salt funktion implementiert. Um ein möglichst langes Passwort zu haben wurde dafür 20 Zeichen für das Passwort und 12 Zeichen für den Salt reserviert (20+12=32).

**Fehlerbehandlung**

Fehler werden entweder intern als 0 (! nicht Null !) oder String zurückgegeben. Damit kann die Fehlermeldung direkt weitergeben und dem Benutzer angezeigt werden.

**Funktionen**

checkPW:

verifiziert den Benutzer. Dabei wird das eingegebene Passwort mit dem Salt aus der DB gehasht und mit dem alten Hash aus der DB verglichen. Gleichzeitig wird die Berechtigung als Zahl zurückgegeben. Die Rückgabe dient gleichzeitig als Autorisierung: bei einem falschen Passwort wird eine 0 (Fehler) zurückgeben.

salt\_generator sollte in Zukunft einen zufälligen Wert erzeugen. Bis jetzt wird nur der Benutzername wieder zurückgegeben.

encrypt\_pw hasht das Passwort.

inserUserDB erstellt einen neuen Datensatz in der Tabelle Users bzw. fügt einen neuen Benutzer hinzu.

changePW ändert das Passwort.