Dokumentation des Codes der Peer2Peer-Data Gruppe

* Die folgenden Erläuterungen beziehen sich auf den Code im github: [peer2peer-wwi19dsa](https://github.com/michael-spengler/peer2peer-wwi19dsa)
* Die von uns erstellten Dateien sind server.ts, login.html und Nutzerdaten.json

**Rahmenbedingungen:**

Json Dateien:

* Aufgrund fehlender Erfahrung mit dem Programmieren mit und Anbinden von Datenbanken wurde entschieden Daten im JSON-Format zu speichern.
* Damit wird der Zusatzaufwand für dieses Projekt zusätzlich z.B. SQL-Anbindung zu lernen umgangen.
* Die Json-Speicherung erlaubt es zusätzlich, dass während der Entwicklungsphase keine Kosten für einen Server aufkommen, sondern die lokale Datenspeicherung verwendet werden kann.
* Die Json Dateien können zu einem späteren Zeitpunkt mit vergleichsweise wenig Aufwand zu einer Online-Umgebung transferiert werden.
* Diese Art der Speicherung ist im No-SQL-Format. Dies erlaubt beispielsweise eine variable Wahl des Keys - je nach Anwendung. Beispielsweise wird beim registrieren die Email geprüft und agiert somit als Key. In einem anderen Anwendungsfall könnte aber ein anderer Teil des JSON-Objekts zum Key werden.

# Was wir erreicht haben:

## Server.ts-Datei:

* Import der deno-libraries “opine” und “jsonfile”
* durch “opine” wird die Kommunikation zwischen Server und Client ermöglicht.
* durch “jsonfile” wird das Speichern gegebener Werte in JSON-Dateien ermöglicht
* Mithilfe der app.get Methode wird die übergebene HTML-Seite im localhost aufgerufen. Mittels Deno.cwd() wird der aktuelle Basispfad als root übergeben
* Die Funktion arrayAuslesen() speichert in einer globalen Variable den aktuellen Inhalt der Nutzerdaten.json Datei
* Anschließend werden zwei Methoden zur Verarbeitung der übergebenen Client-Daten geschrieben. Sie geben einen console.log als Bestätigung des Ergebnisses aus
* Die Methode “userAnlegen(pDaten, pNutzerMail, pDict)” prüft, ob die Email der übergebenen Registrierungsdaten schon im Datenspeicher vorhanden ist.
  + Wenn dies nicht der Fall ist: Die Nutzerdaten.json Datei mit einem aktualisierten Dict, das die neuen Registrierungsdaten enthält, überschrieben
* Die Methode “LoginPrüfen(pDaten, pNutezrmail, pNutzerPW)” prüft, ob die vom Nutzer übergebene Login-Mail in der Nutzerdaten.json Datei vorhanden ist und, ob das Passwort korrekt ist
* Anschließend wird zweimal ein app.post-Befehl ausgeführt, welcher jeweils einen Endpoint definiert, an den die Daten aus der login.html gesendet werden
* innerhalb dieser Funktionen wird es dadurch ermöglicht, dass die zuvor erläuterten Methoden mit den vom Client eingegebenen Daten ausgeführt werden können
* Die erste version (app.post("/saveDataToServer” [..]) ) wird als grundlegende Blaupause für ähnliche Interaktionen verwendet. Es müsste nur jeweils der Methodenaufruf geändert werden. Gegebene console.log Befehle werden temporär zum Zwecke des Debuggens verwendet und sind nicht für den finalen Zustand relevant.
* Abschließend sorgt app.listen(3000) dafür, dass das Programm auf dem Port 3000 des LocalHosts läuft

## <Script>-Teil der login.html-Datei:

* Es werden alle Werte der Input-Felder in eindeutig genannten Variablen gespeichert. Es wird zwischen Sign\_Up (Registrieren) und Sign\_In (Einloggen) unterschieden
* Es gibt zwei Funktionen, die jeweils durch einen Click auf einen Button ausgelöst werden: Die Funktion “registrieren()”, wenn auf der Oberfläche auf Sign\_Up geklickt wird. Die Funktion “login()”, wenn auf der Oberfläche Sign\_In geklickt wird
* Die Funktion “registrieren()” prüft zuerst, ob die eingegeben Passwörter übereinstimmen. Dann wird ein fetch ausgeführt, bei dem der erste Endpoint der server.ts-Datei angesprochen wird (“saveDataToServer”). Es werden die Registrierungsdaten des Nutzers im JSON-Format an diesen Endpoint und damit an die server.ts-Datei geschickt.
* Die Funktion “login()” ruft ebenfalls einen fetch-Befehl auf. Dieser spricht aber einen anderen Endpoint (“saveloginDataToServer”) an. Hier werden im JSON-Format die Logindaten des Nutzers an den Server geschickt

## Nutzerdaten.json-Datei:

* In dieser Datei liegt ein Array aus JSON-Objekten. Jedes JSON-Objekt spiegelt hierbei die Daten eines Nutzers wider.

# Was gab es für Probleme? (Troubleshooting)

* Import in html-Datei nicht möglich
  + Lösung: Auslagern der Imports in separate server.ts-Datei
* Kommunikation der html-Datei mit der server.ts-Datei
  + Lösung: Fetch-Befehl in html-Datei sendet die Daten an einen Endpoint in der server.ts-Datei, der dort ausgelesen werden kann
* Bei Auslesen der an die server.ts-Datei gesendeten Daten gab “req.body” nur “AsyncFunction: read” aus.
  + Lösung: Import von “json” als Teil der Opine-Library. app.use(json()).
    - Hierdurch wird die gesendete Nachricht als json “geparsed” und somit korrekt in der server.ts-Datei ausgelesen
* Eingangs nicht ersichtliche Inkompatibilitäten zwischen Opine und Deno-Versionen
* Lösung: neueste Version von beidem verwenden. (neuere Versionen können jedoch neue Probleme aufwerfen, die wieder gelöst werden müssen)
* Änderungen von Syntax oder verlangten Parametern zwischen Opine Versionen (neuere Version braucht anderen Pfad)
  + Lösung: “Root” des Pfads mit “Deno.cwd()” angegeben
* Pfadprobleme auf verschiedenen Betriebssystemen
  + Lösung: Mac nutzt “/” im Pfad und Windows nutzt “\\” im Pfad
* Fehler bei Angabe des Dateipfades
* Komplexer Dateipfad mit Leerzeichen oder Sonderzeichen verhindert das Auslesen von Dateien
* Fehlende Dokumentation / Problemlösungs Ressourcen zu Deno und Opine
  + Deno bietet als “moderne Software” nicht viel Dokumentation, was die Zeit, die man zur Problemlösung braucht, deutlich verlängert
* Oft ausführliche Diskussionen über Lösungen mit node.js und oder Express, welche häufig nicht mit Deno kompatibel waren
* Arbeiten mit jQuery verursachte einige Errors und Fehlermeldungen
  + Lösung: Alternativen mit anderen/moderneren Tools finden, um jQuery zu umgehen

# Was wären zukünftige To-Dos?

* Datenspeicherung auf einem Server
  + Domain beantragen und statt des LocalHosts einsetzen
* Antworten der server.ts-Datei an die HTML-Datei zurückschicken können
  + Dies erlaubt, dass auf der Oberfläche für den Nutzer sinnvolle Antworten angezeigt werden können, bzw. ermöglicht erst die Funktion, dass ein Nutzer Daten aus dem Datenspeicher abfragen kann
* Datenspeicherung auf alle Teile der Website ausweiten (nicht nur Nutzerregistrierung, sondern auch Speicherung der Angebote, Bewertungen etc.)
* Registrierte Kunden auf Wunsch per Klick löschen (neben Registrierfunktion im Datenspeicher auch Löschfunktion umsetzen)
* Problembehebung bei Vergessen des Passworts: Option Passwort vergessen mit der Möglichkeit einer neuen Passwortvergabe
* Kunden sollen die Möglichkeit besitzen ihre E-Mail zu aktualisieren ohne einen neuen Kunden anlegen zu müssen.
  + Kunden behalten die Anzahl ihrer erledigten Aufträge, die für andere Kunden sichtbar ist
  + Die Vergabe von Aufträgen an aktive Nutzer ist durchaus wahrscheinlicher