Prototype Entry Points

Note: I am using typescript for the types. If you prefer another language, please do tell.

All entry points requires the accessToken in the auth header. Except for anonymous entry points:

- `/login`
- `/register`

All entry points uses `application/x-www-form-urlencoded` except for `profiles/avatar` as it uses `multipart/form-data` for uploading the profile picture.

All `cursor` and `offset` can be either 'start' or 'end' of the set that is being fetched to. Just ensure that for fetching with an `offset`, the returned `cursor` should be used.

Some request body requires a partial `User`, these entry points are used to update the user, this does not include the unchangeable fields (`id`).

Error responses are not defined in here. I have created a 'temporary' handler for that inside the prototype for displaying thrown error status codes along with their messages.

Security implementations like input validation are done through the frontend but you can implement those in the backend. Cors, Rate limiting etc are not impelemented yet.

Table of Summary:

Auth

*I am using JWT Token for auth. If you are using another way, please do tell.

** `expiresIn` uses the `<number><unit>` format, where `unit` can be `ms` / `s` / `m` / `h` / `d` / `w` / `y` (See Time Duration String Format)

/login

POST

Request Body:

```
{
    email: string;
    password: string;
}
```

Returns:

```
{
    accessToken: string; //*
    refreshToken: string;
    expiresIn: string; //**
    refreshExpiresIn: string;
    userId: number;
}
```

/register

POST

Request Body:

```
{
    email: string;
    password: string;
    type: UserRole;
    metadata: UserMetadata;
}
```

Returns:

```
{
    accessToken: string;
    refreshToken: string;
    expiresIn: string;
    refreshExpiresIn: string;
    userId: number;
}
```

Note: Register skips logging back in. Ideally, after register, it should proceed to profile/preference setup.

`/refresh-token`

POST

Request Body:

```
{
    refresh_token: string;
}
```

Returns:

```
{
    accessToken: string;
    refresh_token: string; // new refresh token
    expiresIn: string;
    refreshExpiresIn: string;
}
```

`/logout`

POST

Just invalidates tokens and disconnects sockets

Preferences / Profiles

`/profile`

GET

Returns: User (Currently Logged in)

POST

Updates the current user details

Request Body: Partial type of User (all fields are optional)

Does not return anything.

`/profile/avatar`

PUT

Used for updating profile picture.

Executed after successful registration, if you want the `/register` entry point to handle the profile picture instead, please do tell.

Uses `multipart/form-data`.

```
{
    image: File;
}
```

`/profile/[id]`

GET

> NOT BEING USED - CAN BE SKIPPED

Used for general user details but is not currently used.

URL Params:

```
id = number; // user id
```

Returns: User

Matching/Favoriting

`/matches?offset`

GET

Query Params:

```
offset? = number; // For pagination
```

Returns:

```
{
    matches: Match[];

    /** For "pagination",
    * for when the amount of matches exceeds your pre-defined limit.
    * This signifies the 'offset' needed if we want to get the next set of matches.
    */
    cursor?: number;
}
```

The limit is defined in the backend instead in the app. If you require the limit passed in the app, please do tell.

`/set-favorite`

POST

```
{
    id: number; // User ID (Provider User Type)
}
```

Returns: Nothing

Toggles the provider to be favorited/unfavorited.

Ratings

`/ratings/[userId]?offset`

GET

Query Params:

```
offset? = number; // Pagination
```

URL Params:

```
userId = number; // User Id (Provider)
```

Returns:

```
{
    average: number;
    count: number; // Total number of all ratings of a provider
    ratings: Rating[],
    cursor?: number;
}
```

POST

URL Params:

```
userId = number; // User Id (Provider)
```

Request Body:

```
{
    rating: number;
    details: string;
}
```

Returns: `Rating`

`/ratings/dispute/[ratingsId]`

POST

Request Body:

```
{
    reason: string;
}
```

Returns: `Nothing`

Messaging

`/conversations?offset`

GET

Query Params:

```
offset? = number; // For 'load on scroll'
```

Returns:

```
{
    messages: Conversation[];
    cursor?: number;
}
```

`/conversations/[userId]?offset`

GET

Query Params:

```
offset? = number; // For 'load on scroll'
```

Returns:

```
{
    messages: Message[],
    cursor?: number;
}
```

POST

Request Body:

```
{
    message: string;
    // Will be adding a 'type' field for special message types (like sharing GPS)
}
```

Returns: `Message`

Notifications

`/notifications?offset`

GET

Query Params:

```
offset? = number; // For 'load on scroll'
```

Returns:

```
{
    notifications: Notification[],
    cursor?: number;
}
```

`/read-notification/[notificationId]`

POST

URL Params:

```
notificationid = number;
```

Returns: `Nothing`

Subscriptions

`/subscriptions?offset`

GET

Query Params:

```
offset? = number;
```

Returns:

```
{
    subscriptions: Subscription[];
    cursor?: number;
}
```

Roster Management

`/roster?offset`

GET

Query Params:

```
offset? = number; // For pagination
```

Returns:

```
{
    provider: User[];
    cursor?: number;
}
```

POST

Request Body:

```
{
    email: string;
    password: string;
    metadata: UserMetadata;
}
```

Returns: `User`

`/roster/[userId]`

GET

URL Params:

```
userId = number;
```

Returns: `User`

POST

Used for editing roster provider data, including setting their profile (used for matching)

Request Body: Partial `User` (all fields are optional)

Returns `Nothing`

GPS

`/gps-initiate`

POST

This will start sending location data (stored preferably, to accomodate latency issues, that gets updated by a WS emit event) to participating users.

The one who initiates this is the user that accepts the offer (usually the Recipient). The offer is created as a message a message type (See `Message`) where the user who receives the invitation can accept, and executes this request to start the exchange.

This is what I designed in the prototype but if you see any security issues or any necessary changes, please do tell.

Request Body:

```
{
    // For security reason, the user who sent the invitation is not provided
    // The message ID that was sent as location share invite
    // Is used to know who the other participant is

    // If you have a better idea for this, please do change
    messageId: number;
}
```

Returns: `Nothing`

Websockets

If you prefer long polling that is also fine, but I will be requiring entry points for those or re-purpose my WS interface to be used for long polling.

`/_ws`

Authentication is done through the `init` message only.

Message Events

Message events are structured with the interface:

```
{
    event: string; // The event name (init, message:reply etc)
    data: any[]; // Dynamic array, based on the event name
}
```

Emit Events (client -> server)

Init

Initialize the peer connection. Used to apply user information in the peer object in the server for sending specific broadcasts.

```
{
    event: "init",
    data: [
        accessToken: string;
    ]
}
```

Read Message

Updates the message data 'seen' field to true

```
{
    event: "message:seen",
    data: [
        messageId: number; //
    ]
}
```

The accessToken should then be decoded inside the server for identification.

GPS Location Send

```
{
    event: "gps:send",
    data: [
        latitude: number;
        longitude: number;
    ]
}
```

GPS Location End

This is not required for stopping a Location share. Disconnection or timeout can cause the broadcast event to be broadcasted (listener version) to clients.

```
{
    event: "gps:stop",
    data: [
        otherUserId: number; // The other user that is included in the exchange
    ]
}
```

This event is used to update the location of a user.

Listen Events (server -> client)

Message Reply

Used inside the messenger for realtime messaging

```
{
    event: "message:reply",
    data: [
        senderId: number,
        message: string,
        sentDate: string // ISO8601
    ]
}
```

Notification Event

This is Work In Progress. If you have a better way to handle notification websockets then please do tell.

```
{
    event: "notification",
    data: [
        notificationObject: Notification;
    ]
}
```

GPS Location Receive

This is sent on both the users that are the partipants from `/gps-initiate`

In my backend, I am running tasks on interval. Of course this is instantiated and will not run when not needed (See NitroJS Tasks)

```
{
    event: "gps:receive",
    data: [
        fromId: number; // User ID
        latitude: number;
        longitude: number;
    ]
}
```

GPS Location End

```
{
    event: "gps:stop",
    data: [
        otherUserId: number;
    ]
}
```

Types

I have seen the php models and most of these types does not align with what is in there. Please give feedbacks regarding to changes that are necessary.

UserRole

```
type UserRole =
    | "ROLE_RECIPIENT"
    | "ROLE_PROVIDER"
    | "ROLE_ROSTER_PROVIDER"
    | "ROLE_COMPANY";
```

UserMetadata

```typescript
// Dynamic, based on user type
type UserMetadata = { profilePictureURL: string } & (
    | // Recipients/Providers
    {
          firstname: string;
          lastname: string;
      }
    // Companies
    | {
          name: string;
      }
);
```

User

```typescript
type User = {
    id: number;
    email: string;
    roles: UserRole[];
    createdAt: string; // ISO8601
    updatedAt: string; // ISO8601
    lastLogin: string; // ISO8601
    isActive: boolean;

    metadata: UserMetadata;

    /**
     * Contains all preference fields that are used for matching
     * Things like but not limited to:
     * prefTitle, prefVeteran, prefGender etc...
     * I can't list all since I haven't completely implemented all the preference fields
     */
    preferences: Map<string, any>;
};
```

Match

```typescript
type Match = {
    provider: User;
    isFavorite: boolean;
    compatibility: number;
    ratings: {
        average: number;
        count: number;
    };
};
```

Rating

```typescript
type Rating = {
    id: number;
    author: Omit<User, "preferences">; // Does not need the 'preferences' field
    provider: Omit<User, "preferences">; // Does not need the 'preferences' field
    rating: number;
    details: string;
    createdAt: string; // ISO8601
};
```

Conversation

```
type Conversation = {
    user: Omit<User, "preferences">; // No need for preferences field
    lastMessage: Message;
};
```

Message

```
type Message = {
    id: number;
    senderId: number;
    receiverId: number;
    content: string;
    type: MessageType;
    createdAt: string | null;
    seen: boolean;
};

type MessageType = "text" | "gps_invite";
```

Notification

```
type Notification = {
    id: number;
    receiverId: number; // User Id
    type: NotificationType;
    subject: string;
    description: string;
    metadata: NotificationMetadata; //
    isRead: boolean;
};
```

NotificationType

```
type NotificationType = "message" | "ratings" | "subscription";
```

NotificationMetadata

This is used for interaction when clicking the notification. Ex: Clicking a message notification will jump to the conversation.

```
type NotificationMetadata =
    // Message Type
    | {
          senderId: number; // User Id
      }
    // Rating Type
    | {
          ratingId: number; // Used to immediately jump to the specific rating
      };
// Subscription type no metadata yet
```

Subscription

Subscriptions are not implemented yet. Please suggest if you have better way.

```
type Subscription = {
    id: number;
    status: boolean;
    datePurchased: string; // ISO8601
    expiryDate: string; // ISO8601
    searchCredits: number;
    // Supposed to be subscript plans, can be number for enum instead
    plan: string;
};
```