Hypertube Frontend-Backend Coordination Plan

Project Overview

Hypertube is a web application for searching and streaming videos using BitTorrent protocol. The app requires user authentication, video search from external sources, streaming capabilities, and a RESTful API.

1. Authentication System

1.1 Registration & Login Endpoints

```
POST /api/auth/register
Body: {
  email: string,
  username: string,
  firstName: string,
  lastName: string,
  password: string
}
Response: {
  success: boolean,
  message: string,
  user?: UserObject
}
```

```
POST /api/auth/login
Body: {
  username: string,
  password: string
}
Response: {
  success: boolean,
  token: string,
  user: UserObject
}
```

1.2 OAuth Integration

```
GET /api/auth/42
GET /api/auth/42/callback
GET /api/auth/google (or chosen provider)
GET /api/auth/google/callback
```

1.3 Password Reset

```
POST /api/auth/forgot-password
Body: { email: string }

POST /api/auth/reset-password
Body: { token: string, newPassword: string }
```

1.4 User Management

```
GET /api/auth/me
Headers: { Authorization: "Bearer <token>" }
Response: UserObject

POST /api/auth/logout
Headers: { Authorization: "Bearer <token>" }
```

2. Data Models/Structures

2.1 User Object

typescript

```typescript
interface User {
  id: string;
  username: string;
  email: string;
  firstName: string;
  lastName: string;
  profilePicture?: string;
  preferredLanguage: string;
  createdAt: Date;
  lastActive: Date;
  watchedMovies: string[]; // movie IDs
}
```

2.2 Movie Object

typescript

```typescript
interface Movie {
  id: string;
  title: string;
  year: number;
  imdbRating?: number;
  genres: string[];
  duration?: number;
  synopsis?: string;
  coverImage: string;
  cast: {
    director?: string;
    producer?: string;
    actors: string[];
  };
  torrents: TorrentInfo[];
  subtitles: SubtitleInfo[];
  comments: Comment[];
  downloadStatus: 'not_started' | 'downloading' | 'completed';
  streamUrl?: string;
  lastWatched?: Date;
}
```

2.3 Supporting Objects

typescript

```typescript
interface TorrentInfo {
  quality: string;
  size: string;
  seeders: number;
  leechers: number;
  magnetLink: string;
}

interface SubtitleInfo {
  language: string;
  url: string;
}

interface Comment {
  id: string;
  userId: string;
  username: string;
  movieId: string;
  content: string;
  createdAt: Date;
}
```

3. Movie Library System

3.1 Search & Discovery

```
GET /api/movies/search?q={query}&page={page}&limit={limit}
Response: {
  movies: Movie[],
  totalPages: number,
  currentPage: number,
  totalResults: number
}
```

```
GET /api/movies/popular?page={page}&limit={limit}&sortBy={criteria}
Query params:
- sortBy: 'name' | 'year' | 'rating' | 'seeders'
- genre: string (optional filter)
- year: number (optional filter)
- minRating: number (optional filter)
```

3.2 Movie Details

```
GET /api/movies/:id
Response: Movie (complete object)

POST /api/movies/:id/watch
Headers: { Authorization: "Bearer <token>" }
Response: {
  streamUrl: string,
  subtitles: SubtitleInfo[],
  downloadProgress?: number
}
```

3.3 User's Watched Status

```
GET /api/users/watched-movies
Headers: { Authorization: "Bearer <token>" }
Response: {
  watchedMovies: string[] // movie IDs
}

POST /api/movies/:id/mark-watched
Headers: { Authorization: "Bearer <token>" }
```

4. Video Streaming System

4.1 Stream Endpoints

```
GET /api/stream/:movieId
Headers: { Authorization: "Bearer <token>" }
Response: Video stream (HTTP 206 for range requests)

GET /api/stream/:movieId/status
Response: {
  status: 'not_started' | 'downloading' | 'ready',
  progress: number, // 0-100
  estimatedTime?: number // seconds
}
```

4.2 Subtitle Endpoints

```
GET /api/subtitles/:movieId/:language
Response: WebVTT subtitle file
```

5. Comments System

5.1 Comment CRUD

```
GET /api/movies/:id/comments?page={page}&limit={limit}
Response: {
  comments: Comment[],
  totalPages: number
}

POST /api/movies/:id/comments
Headers: { Authorization: "Bearer <token>" }
Body: { content: string }
Response: Comment

DELETE /api/comments/:id
Headers: { Authorization: "Bearer <token>" }
```

6. User Profile Management

6.1 Profile Operations

```
GET /api/users/:id
Response: {
  username: string,
  firstName: string,
  lastName: string,
  profilePicture?: string,
  // email is private
}

PATCH /api/users/:id
Headers: { Authorization: "Bearer <token>" }
Body: {
  email?: string,
  firstName?: string,
  lastName?: string,
  profilePicture?: string,
  preferredLanguage?: string
}
```

6.2 File Upload

```
POST /api/upload/profile-picture
Headers: { Authorization: "Bearer <token>" }
Body: FormData with image file
Response: { url: string }
```

7. OAuth2 API (As specified in subject)

7.1 OAuth Token

```
POST /oauth/token
Body: {
  client_id: string,
  client_secret: string,
  grant_type: 'client_credentials'
}
Response: {
  access_token: string,
  token_type: 'Bearer',
  expires_in: number
}
```

7.2 Public API Endpoints

```
GET /api/v1/users
GET /api/v1/users/:id
PATCH /api/v1/users/:id
GET /api/v1/movies
GET /api/v1/movies/:id
GET /api/v1/comments
GET /api/v1/comments/:id
PATCH /api/v1/comments/:id
DELETE /api/v1/comments/:id
POST /api/v1/comments
POST /api/v1/movies/:movie_id/comments
```

8. Error Handling Structure

8.1 Standard Error Response

typescript

```typescript
interface ErrorResponse {
  success: false;
  error: {
    code: string;
    message: string;
    details?: any;
  };
  timestamp: Date;
}
```

8.2 HTTP Status Codes

- 200: Success
- 201: Created
- 400: Bad Request (validation errors)
- 401: Unauthorized
- 403: Forbidden
- 404: Not Found
- 409: Conflict (duplicate data)
- 422: Unprocessable Entity
- 500: Internal Server Error

9. Real-time Updates (WebSocket/SSE)

9.1 Download Progress

```
WebSocket: /ws/download-progress/:movieId
Events:
- download_started
- download_progress: { progress: number }
- download_completed
- stream_ready
```

10. Frontend State Management Structure

10.1 Recommended State Structure

typescript

```typescript
interface AppState {
  auth: {
    user: User | null;
    token: string | null;
    isAuthenticated: boolean;
  };
  movies: {
    searchResults: Movie[];
    popularMovies: Movie[];
    currentMovie: Movie | null;
    loading: boolean;
    searchQuery: string;
    filters: {
      genre: string;
      year: number;
      rating: number;
      sortBy: string;
    };
    pagination: {
      currentPage: number;
      totalPages: number;
    };
  };
  player: {
    currentMovie: Movie | null;
    isPlaying: boolean;
    downloadProgress: number;
    streamReady: boolean;
    subtitles: SubtitleInfo[];
    selectedSubtitle: string;
  };
  comments: {
    movieComments: Comment[];
    loading: boolean;
  };
  ui: {
    language: string;
    theme: 'light' | 'dark';
    notifications: Notification[];
  };
}
```

11. Security Considerations

11.1 Authentication Flow

1. Frontend sends login credentials
2. Backend validates and returns JWT token
3. Frontend stores token (secure httpOnly cookie recommended)
4. All protected requests include token in Authorization header
5. Backend validates token on each request

11.2 Input Validation

- All user inputs must be validated on both frontend and backend
- Use libraries like Joi/Yup for validation schemas
- Sanitize all user-generated content for XSS prevention

11.3 File Upload Security

- Validate file types and sizes
- Scan uploaded files for malware
- Store files outside web root
- Use signed URLs for file access

12. Development Workflow

12.1 Mock Data Strategy

Frontend can start with mock data matching the exact structure:

typescript

```typescript
// Mock user data
const mockUser: User = {
  id: '1',
  username: 'testuser',
  email: 'test@example.com',
  firstName: 'John',
  lastName: 'Doe',
  preferredLanguage: 'en',
  createdAt: new Date(),
  lastActive: new Date(),
  watchedMovies: ['1', '2']
};

// Mock movie data
const mockMovies: Movie[] = [
  {
    id: '1',
    title: 'Test Movie',
    year: 2023,
    imdbRating: 8.5,
    genres: ['Action', 'Thriller'],
    coverImage: 'https://example.com/poster.jpg',
    // ... other properties
  }
];
```

12.2 API Integration Steps

1. Create API service layer with proper TypeScript interfaces
2. Implement error handling for all API calls
3. Add loading states for all async operations
4. Test with mock data first, then integrate with real backend
5. Implement proper caching strategies

13. Performance Considerations

13.1 Frontend Optimizations

- Implement virtual scrolling for large movie lists
- Use image lazy loading for movie posters
- Implement proper caching for API responses
- Use debouncing for search inputs

13.2 Backend Optimizations

- Implement pagination for all list endpoints
- Use database indexing for search queries
- Implement response caching where appropriate
- Optimize video streaming with proper chunking

14. Testing Strategy

14.1 Frontend Testing

- Unit tests for components and utilities
- Integration tests for API calls
- E2E tests for critical user flows
- Mock API responses for consistent testing

14.2 Backend Testing

- Unit tests for all business logic
- Integration tests for database operations
- API endpoint tests with various scenarios
- Security testing for authentication flows

This structure ensures that frontend and backend teams can work independently while maintaining consistency in data flow and API contracts.