**🚀 Mivton Project - Phase 3.1 Context Prompt**

**I am working on MIVTON - a futuristic multilingual chat platform. You are my technical developer. We have completed ALL of Phase 1 (Foundation) and Phase 2 (User Interface & Components) successfully, and now need to implement Phase 3.1 - Friends System & Social Features.**

**🎯 Project Overview**

* **Name**: Mivton
* **Purpose**: Friends-only multilingual text chat with real-time OpenAI translation
* **Target**: Gen Z users with futuristic design
* **Status**: Phase 1 ✅ COMPLETE, Phase 2 ✅ COMPLETE, Phase 3.1 🔄 IN PROGRESS

**✅ PHASE 1 - COMPLETE FOUNDATION (A+ Quality)**

* **Phase 1.1**: Landing page and infrastructure ✅ COMPLETE
* **Phase 1.2**: Database setup with PostgreSQL ✅ COMPLETE
* **Phase 1.3**: Authentication system with futuristic UI ✅ COMPLETE

**✅ PHASE 2 - COMPLETE USER INTERFACE (A+ Quality)**

**Total Achievement**: 12,000+ lines of enterprise-grade code

**Phase 2.1 - Dashboard Framework ✅ COMPLETE (A+ Grade)**

* **Dashboard**: Modern layout with glassmorphism sidebar navigation
* **Responsive**: Mobile-first with hamburger menu and touch interactions
* **Code**: 3,500+ lines of professional code
* **Quality**: Exceptional UX with comprehensive error handling

**Phase 2.2 - Modern UI Components ✅ COMPLETE (A+ Grade)**

* **Component Library**: Enterprise-grade with 8 major component systems
* **Architecture**: BaseComponent inheritance pattern with global namespace
* **Code**: 4,500+ lines with <40KB overhead, 60fps animations
* **Quality**: WCAG 2.1 AA compliant, GPU-accelerated performance

**Phase 2.3 - User Interface Polish ✅ COMPLETE (A+ Grade)**

* **User Search**: Live search with debounced input and profile cards
* **Language System**: 50+ languages with flag icons
* **Status Management**: Real-time presence indicators (online/away/busy/offline)
* **Settings**: Enhanced interface using Phase 2.2 components
* **Code**: 2,500+ lines with 7 new API endpoints
* **Quality**: Professional mobile-responsive design

**🔄 CURRENT PHASE 3.1 - Friends System & Social Features (Day 1-2)**

**Goal**: Complete friends management with real-time notifications **Priority**: IMPLEMENT THIS PHASE

**Phase 3.1 Tasks:**

* [ ] Create friends database schema (friendships, friend\_requests, blocked\_users)
* [ ] Implement friend request system with validation
* [ ] Add real-time friend notifications
* [ ] Build friend list management with online status
* [ ] Create block/unblock functionality
* [ ] Test: Complete friend request workflow

**Required Social Features:**

**1. Friends Database Architecture (CRITICAL FOUNDATION)**

-- Primary friendship table (bidirectional relationships)

CREATE TABLE friendships (

id SERIAL PRIMARY KEY,

user1\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

user2\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

status VARCHAR(20) DEFAULT 'active',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(user1\_id, user2\_id),

CHECK (user1\_id < user2\_id) -- Enforce order to prevent duplicates

);

-- Friend requests table

CREATE TABLE friend\_requests (

id SERIAL PRIMARY KEY,

sender\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

receiver\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

status VARCHAR(20) DEFAULT 'pending',

message TEXT, -- Optional message with friend request

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(sender\_id, receiver\_id),

CHECK (sender\_id != receiver\_id) -- Cannot send request to self

);

-- Blocked users table

CREATE TABLE blocked\_users (

id SERIAL PRIMARY KEY,

blocker\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

blocked\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(blocker\_id, blocked\_id),

CHECK (blocker\_id != blocked\_id) -- Cannot block self

);

-- Indexes for performance (from Phase 2.3 recommendations)

CREATE INDEX idx\_friendships\_user1 ON friendships(user1\_id);

CREATE INDEX idx\_friendships\_user2 ON friendships(user2\_id);

CREATE INDEX idx\_friend\_requests\_sender ON friend\_requests(sender\_id);

CREATE INDEX idx\_friend\_requests\_receiver ON friend\_requests(receiver\_id);

CREATE INDEX idx\_blocked\_users\_blocker ON blocked\_users(blocker\_id);

CREATE INDEX idx\_blocked\_users\_blocked ON blocked\_users(blocked\_id);

**2. Friend Request Management**

* **Send Requests**: Validation, duplicate prevention, message support
* **Receive Requests**: Real-time notifications, accept/decline actions
* **Request States**: Pending, accepted, declined, expired
* **Bulk Actions**: Accept/decline multiple requests
* **Rate Limiting**: Prevent spam (5 requests per hour per user)

**3. Friends List Management**

* **Friends Display**: Profile cards with online status from Phase 2.3
* **Search Friends**: Filter by name, status, language
* **Friend Actions**: Chat, remove, block options
* **Online Status**: Real-time presence using Phase 2.3 status system
* **Friend Statistics**: Total friends, online friends count

**4. Block/Unblock System**

* **Block Users**: Prevent all interactions and visibility
* **Unblock Users**: Restore normal interactions
* **Block List**: Manage blocked users with timestamps
* **Block Effects**: Remove from friends, cancel pending requests
* **Privacy**: Blocked users cannot see blocker's profile

**5. Real-time Social Notifications**

* **Friend Requests**: Instant notifications for new requests
* **Friend Status**: Online/offline status changes
* **Friend Actions**: Accept/decline notifications
* **Notification Management**: Mark as read, dismiss, settings

**🗄️ Database Schema (Current + Phase 3.1 Extensions)**

-- Existing tables (Phases 1-2)

users (

id, username, email, password\_hash, full\_name,

gender, native\_language, is\_verified, is\_admin,

admin\_level, is\_blocked, status, last\_login,

created\_at, updated\_at

);

session (sid, sess, expire);

user\_activity (id, user\_id, activity\_type, last\_activity);

user\_preferences (id, user\_id, preference\_key, preference\_value, created\_at, updated\_at);

waitlist (id, email, referrer, user\_agent, ip\_address, created\_at, notified);

-- NEW Phase 3.1 tables (implement these first)

friendships, friend\_requests, blocked\_users (schemas above)

**🔧 Technical Infrastructure (Production Ready)**

**Railway Environment Variables ✅**

NODE\_ENV=production

JWT\_SECRET=mivton-super-secret-jwt-key-2025-production

OPENAI\_API\_KEY=sk-proj-ssWG4RYWzRSkC6N5wSrwR-ajRzmcGMLG2agNfBO1IRpplc7a8LzwmHrKNeShj4J2gm8ynWDu\_2T3BlbkFJob8\_1Ny3bs5vVdEKNo48BEsXZOB4kGzgtQ-O0-JAssePAFHU7CFAi-cLPc-xYop2z362vHtd4A

SMTP\_HOST=smtp.hostinger.com

SMTP\_PORT=465

SMTP\_SECURE=true

SMTP\_USER=info@mivton.com

SMTP\_PASS=Bacau@2012

APP\_URL=https://mivton.com

FRONTEND\_URL=https://mivton.com

DATABASE\_URL=[Railway PostgreSQL URL]

**Current File Structure (Phase 2 Complete) ✅**

mivton/

├── server.js # Main Express server ✅

├── package.json # Dependencies ✅

├── Dockerfile # CRITICAL: Use simple, proven Dockerfile ✅

├── public/

│ ├── index.html # Landing page ✅

│ ├── login.html # Login page ✅

│ ├── register.html # Register page ✅

│ ├── dashboard.html # Complete dashboard ✅

│ ├── demo.html # Phase 2.3 component demo ✅

│ ├── css/ # Complete styling system ✅

│ │ ├── style.css # Landing (1,000+ lines) ✅

│ │ ├── auth.css # Authentication (800+ lines) ✅

│ │ ├── dashboard.css # Dashboard (1,200+ lines) ✅

│ │ ├── components.css # UI components (800+ lines) ✅

│ │ ├── user-interface.css # User search/profiles (600+ lines) ✅

│ │ ├── language-selector.css # Language system (400+ lines) ✅

│ │ ├── status-indicators.css # Status system (300+ lines) ✅

│ │ └── components/ # Phase 2.2 component styles ✅

│ │ ├── base.css # Base components (500+ lines) ✅

│ │ ├── buttons.css # Button variants (400+ lines) ✅

│ │ ├── cards.css # Card components (450+ lines) ✅

│ │ ├── forms.css # Form styling (600+ lines) ✅

│ │ ├── feedback.css # Toast, modals (550+ lines) ✅

│ │ └── animations.css # Animation library (800+ lines) ✅

│ └── js/ # Complete JavaScript system ✅

│ ├── app.js # Landing page logic ✅

│ ├── auth.js # Auth page logic ✅

│ ├── validation.js # Validation logic ✅

│ ├── dashboard.js # Dashboard functionality (650+ lines) ✅

│ ├── components.js # UI component logic (700+ lines) ✅

│ ├── component-loader.js # Component initialization (400+ lines) ✅

│ ├── icon-system.js # Font Awesome integration (450+ lines) ✅

│ ├── user-search.js # Live search functionality (500+ lines) ✅

│ ├── profile-cards.js # User profile cards (400+ lines) ✅

│ ├── language-selector.js # Language selection (350+ lines) ✅

│ ├── status-manager.js # User status management (300+ lines) ✅

│ ├── settings-interface.js # Settings panel (450+ lines) ✅

│ └── components/ # Phase 2.2 component library ✅

│ ├── BaseComponent.js # Component architecture (400+ lines) ✅

│ ├── Button.js # Button component (450+ lines) ✅

│ ├── Card.js # Card component (500+ lines) ✅

│ ├── Toast.js # Toast system (400+ lines) ✅

│ ├── Modal.js # Modal system (600+ lines) ✅

│ └── AnimationManager.js # Animation utilities (350+ lines) ✅

├── database/ # Database layer ✅

│ ├── connection.js # PostgreSQL connection ✅

│ ├── schema.sql # Core schema ✅

│ ├── init.js # DB initialization ✅

│ ├── user-activity.sql # User activity schema ✅

│ └── user-preferences.sql # User preferences schema ✅

├── middleware/ # Middleware layer ✅

│ ├── database.js # DB middleware ✅

│ ├── auth.js # Auth middleware ✅

│ └── validation.js # Validation middleware ✅

├── utils/ # Utility layer ✅

│ ├── database.js # DB utilities ✅

│ ├── auth.js # Auth utilities ✅

│ ├── email.js # Email system ✅

│ ├── validation.js # Validation helpers ✅

│ ├── waitlist.js # Waitlist functions ✅

│ ├── dashboard.js # Dashboard utilities ✅

│ ├── user-search.js # Search utilities ✅

│ └── language-utils.js # Language handling ✅

├── routes/ # API routes ✅

│ ├── auth.js # Authentication routes ✅

│ ├── users.js # User management routes ✅

│ ├── dashboard.js # Dashboard routes (400+ lines) ✅

│ ├── users-search.js # User search API ✅

│ └── user-preferences.js # User preferences API ✅

└── socket/ # Socket.IO handlers (ready for real-time)

**🎨 Design System (Proven Successful)**

/\* Core Colors (proven successful across all phases) \*/

Primary: #6366f1 (Electric Blue)

Secondary: #8b5cf6 (Vibrant Purple)

Accent: #06b6d4 (Cyan)

Success: #10b981 (Green)

Warning: #f59e0b (Amber)

Error: #ef4444 (Red)

Background: #0f172a (Dark Navy)

Surface: #1e293b (Slate)

Text: #f1f5f9 (Light)

/\* Status Colors (from Phase 2.3) \*/

Status-Online: #10b981 (Green)

Status-Away: #f59e0b (Amber)

Status-Busy: #ef4444 (Red)

Status-Offline: #64748b (Gray)

/\* Social Colors (new for Phase 3.1) \*/

Friend-Request: #06b6d4 (Cyan)

Friend-Accepted: #10b981 (Green)

Friend-Blocked: #ef4444 (Red)

Friend-Pending: #f59e0b (Amber)

/\* Animation System (proven from all phases) \*/

--ease-smooth: cubic-bezier(0.4, 0, 0.2, 1);

--ease-bounce: cubic-bezier(0.68, -0.55, 0.265, 1.55);

--ease-elastic: cubic-bezier(0.175, 0.885, 0.32, 1.275);

**💻 Technology Stack**

* **Backend**: Node.js + Express.js ✅
* **Database**: PostgreSQL (Railway) ✅
* **Authentication**: Session-based with bcrypt ✅
* **Email**: Hostinger SMTP ✅
* **Components**: Enterprise-grade component library ✅
* **Search**: Live user search with debouncing ✅
* **Status**: Real-time user presence system ✅
* **Real-time**: Socket.IO ✅ (ready for friend notifications)
* **Translation**: OpenAI GPT-4 (ready for Phase 5)
* **Frontend**: HTML5 + CSS3 + Vanilla JS ✅
* **Deployment**: Railway CLI (no GitHub) ✅

**📋 Phase 3.1 Specific Requirements**

**Database Files to Create FIRST (Critical Foundation):**

database/

├── friends-schema.sql # 🆕 CRITICAL - Friends database schema

├── init-friends.js # 🆕 CRITICAL - Friends tables initialization

└── friends-indexes.sql # 🆕 CRITICAL - Performance indexes

**API Routes to Create:**

routes/

├── friends.js # 🆕 Complete friends management API

├── friend-requests.js # 🆕 Friend request system API

└── social-notifications.js # 🆕 Real-time notification API

**Frontend Components to Create:**

public/js/

├── friends-manager.js # 🆕 Friends list and management

├── friend-requests.js # 🆕 Friend request handling

├── social-notifications.js # 🆕 Real-time notification system

└── friends-ui.js # 🆕 Friends interface components

public/css/

├── friends-system.css # 🆕 Friends list and profile styling

├── friend-requests.css # 🆕 Friend request interface styling

└── social-notifications.css # 🆕 Notification styling

**Utility Functions to Create:**

utils/

├── friends-utils.js # 🆕 Friends management utilities

├── social-validation.js # 🆕 Social interaction validation

└── notification-utils.js # 🆕 Notification management utilities

socket/

└── friends-events.js # 🆕 Real-time friend events (Socket.IO)

**New API Endpoints for Phase 3.1:**

* POST /api/friends/request - Send friend request
* PUT /api/friends/request/:id/accept - Accept friend request
* PUT /api/friends/request/:id/decline - Decline friend request
* DELETE /api/friends/request/:id - Cancel friend request
* GET /api/friends - Get user's friends list
* DELETE /api/friends/:id - Remove friend
* POST /api/friends/block - Block user
* DELETE /api/friends/block/:id - Unblock user
* GET /api/friends/requests/sent - Sent friend requests
* GET /api/friends/requests/received - Received friend requests
* GET /api/friends/blocked - Blocked users list
* GET /api/friends/online - Online friends

**🚨 CRITICAL DEPLOYMENT LESSONS FROM PHASE 2.3 (MUST FOLLOW)**

**1. Simple Dockerfile (Proven Working)**

# ✅ PROVEN: Simple, reliable Dockerfile

FROM node:18

WORKDIR /app

# Copy package.json only first (better caching)

COPY package.json ./

RUN npm install

# Copy application files

COPY . .

EXPOSE 3000

CMD ["node", "server.js"]

**2. Package Management (Avoid Previous Issues)**

# ✅ PROVEN: Generate proper package-lock.json

rm -rf node\_modules package-lock.json

npm install

# Commit package-lock.json to avoid deployment issues

**3. Database Migration Strategy (New for Phase 3.1)**

// ✅ RECOMMENDED: Database validation before server start

const validateFriendsSchema = async () => {

try {

// Check if friends tables exist

const tables = await pool.query(`

SELECT table\_name FROM information\_schema.tables

WHERE table\_schema = 'public'

AND table\_name IN ('friendships', 'friend\_requests', 'blocked\_users')

`);

if (tables.rows.length !== 3) {

console.log('Creating friends database schema...');

await initializeFriendsSchema();

}

} catch (error) {

console.error('Database validation failed:', error);

process.exit(1);

}

};

**4. Error Handling Patterns (Proven Successful)**

// ✅ PROVEN: User-friendly error handling

try {

const result = await sendFriendRequest(senderId, receiverId);

showToast('Friend request sent!', 'success');

} catch (error) {

console.error('Friend request failed:', error);

if (error.code === 'ALREADY\_FRIENDS') {

showToast('You are already friends with this user', 'info');

} else if (error.code === 'REQUEST\_EXISTS') {

showToast('Friend request already sent', 'warning');

} else {

showToast('Unable to send friend request. Please try again.', 'error');

}

}

**🔮 Phase 3.2 Preview (Real-Time Social Updates)**

**After Phase 3.1, we'll implement:**

* Real-time friend status updates via Socket.IO
* Live friend request notifications
* Online/offline presence broadcasting
* Friend activity feeds
* Social interaction analytics

This context helps prepare social system for real-time features.

**👥 Our Roles - CRITICAL**

* **You (Claude)**: Create ALL files directly in the mivton directory structure. I CANNOT copy, modify, or create files manually.
* **Me**: Only run railway up to deploy the changes you create.

**🚨 PROVEN SUCCESS PATTERNS (A+ Quality Standards)**

**1. Global Namespace Management (Proven Pattern)**

// ✅ Extend existing MivtonComponents namespace

window.MivtonComponents = {

...window.MivtonComponents,

FriendsManager: null,

FriendRequests: null,

SocialNotifications: null,

// Initialize all social components here

};

**2. Component Architecture (Proven Pattern)**

// ✅ Use proven BaseComponent inheritance

class MivtonFriendsManager extends MivtonBaseComponent {

constructor(element, options = {}) {

super(element, options);

this.initializeFriendsSystem();

}

}

**3. Rate Limiting Implementation (Prevent Abuse)**

// ✅ CRITICAL: Rate limiting for social actions

const RATE\_LIMITS = {

FRIEND\_REQUESTS: { max: 5, window: 3600000 }, // 5 per hour

FRIEND\_ACTIONS: { max: 20, window: 3600000 } // 20 per hour

};

**4. Database Optimization (From Phase 2.3 Recommendations)**

// ✅ Optimized queries with proper joins

const getFriendsWithStatus = async (userId) => {

return await pool.query(`

SELECT u.id, u.username, u.full\_name, u.status, u.native\_language,

u.is\_verified, ua.last\_activity

FROM friendships f

JOIN users u ON (u.id = f.user1\_id OR u.id = f.user2\_id)

LEFT JOIN user\_activity ua ON ua.user\_id = u.id

WHERE (f.user1\_id = $1 OR f.user2\_id = $1)

AND u.id != $1 AND f.status = 'active'

AND u.is\_blocked = FALSE

ORDER BY ua.last\_activity DESC NULLS LAST

`, [userId]);

};

**🚀 What I Need You To Do**

I need you to implement Phase 3.1 - Friends System & Social Features by creating:

1. **Database Schema FIRST**: Create friends, friend\_requests, and blocked\_users tables with proper indexes
2. **Friends API System**: Complete REST API for all friend operations with rate limiting
3. **Friend Request Management**: Send, accept, decline, cancel with validation
4. **Friends List Interface**: Display friends with Phase 2.3 profile cards and online status
5. **Block/Unblock System**: Complete user blocking functionality
6. **Real-time Foundation**: Prepare Socket.IO events for Phase 3.2
7. **Integration**: Seamless integration with existing Phase 2 components
8. **Mobile Optimization**: Touch-friendly social interactions

Please follow the proven deployment patterns and create all files directly in the mivton directory structure.

**🎯 Success Criteria**

* Complete friends database schema with proper relationships
* Friend request system with validation and rate limiting
* Friends list with real-time online status from Phase 2.3
* Block/unblock functionality with privacy controls
* Integration with existing component library
* Mobile-responsive social interface
* Zero breaking changes to existing functionality
* Professional error handling with user-friendly messages

**🏆 Target Quality Level**

Continue the A+ standard from all previous phases:

* **Enterprise-grade implementation** with comprehensive error handling
* **Professional UX design** matching established Gen Z aesthetic
* **Performance optimized** with proper database indexing
* **Security focused** with rate limiting and validation
* **Mobile-first responsive** design with touch interactions

**END OF PROMPT - Ready for Phase 3.1 Friends System & Social Features Implementation**