Nightlife OS - Architecture Addendum: Interactive Status & Advanced Features

Erweiterungen für Phase 9+: Party-Modus, Dating-Status, Trust-System, Club-Pläne & Events

Version: 1.0
Erstellt am: 3. Dezember 2025
Basis: ARCHITECTURE.md (Phase 1-8)
Zweck: Fundament für interaktive Features (Party Alarm, Hunt the Fox, Ticketing, Loyalty)

Inhaltsverzeichnis

---

1. ÜBERBLICK & MOTIVATION

Dieses Addendum erweitert die bestehende Nightlife OS-Architektur um **fünf zentrale Konzepte**, die als Fundament für alle zukünftigen interaktiven Features dienen:

Warum diese Erweiterungen?

Problem 1: Push-Notification-Fatigue

Herausforderung: Zu viele Push-Benachrichtigungen nerven User und werden ignoriert/deaktiviert.

Lösung: Party-Modus als bewusste Opt-In-Mechanik ohne Push-Spam. Der User entscheidet aktiv, wann er an Aktionen teilnimmt.

Problem 2: Missbrauch von Dating-Features

Herausforderung: User ändern ihren Dating-Status ständig, um wiederholt Gratis-Drinks zu bekommen.

Lösung: 6-Stunden-Regel mit serverseitiger Validierung.

Problem 3: Zechpreller & Sicherheit

Herausforderung: Anonyme User können Bestellungen aufgeben und nicht bezahlen.

Lösung: Trusted User-System mit Phone-Verifizierung und Türsteher-Validation.

Problem 4: One-Size-Fits-All

Herausforderung: Nicht jeder Club braucht alle Features, aber alle bezahlen dafür.

Lösung: Club-Pläne & Feature-Flags für Free vs. Premium.

Problem 5: Event-Context fehlt

Herausforderung: Features wie Ticketing, Party Alarm, Gutscheine brauchen Event-Context.

Lösung: Event-Lifecycle-Modell mit Vor/Während/Nach-Phasen.

Architektur-Prinzipien dieses Addendums

1. **Minimal Invasive:** Keine Breaking Changes an bestehenden Collections
2. **Firestore-First:** Alle neuen Felder sind direkt abfragbar
3. **Security-First:** Alle neuen Felder haben explizite Security Rules
4. **TypeScript-First:** Alle Interfaces sind typsicher

5. **Future-Proof:** Erweiterbar für kommende Features

---

2. INTERAKTIVER ON-SITE-STATUS: PARTY-MODUS & CHILL-MODUS

Konzept

Der **Party-Modus** ist ein dynamischer Status, der die **physische Anwesenheit** (`isInClub`) mit **aktiver Aufmerksamkeit** verknüpft. Ziel ist es, User ohne aufdringliche Push-Benachrichtigungen in DJ-Aktionen einzubinden.

Kern-Idee

- User checkt sich im Club ein → `isInClub = true`
- User aktiviert Party-Modus durch Auswahl einer der **6 täglichen Zahlen** → `partyModeActive = true`
- DJ ruft über Mikrofon eine der 6 Zahlen auf
- User, die die Ansage hören, öffnen die App und wählen die Zahl
- Bei Match: Teilnahme an DJ-Aktion (Gewinnspiel, Lichtshow, etc.)
- Nach 30 Minuten Inaktivität: Automatisches Verfallen des Party-Modus

Warum keine Push-Notifications?

- ✅ **Kein Spam:** User werden nicht ständig gestört
- ✅ **Bewusste Teilnahme:** Nur wer aktiv hinhört und die App öffnet, nimmt teil
- ✅ **Natürliche Club-Experience:** DJ-Ansagen sind Teil der Clubatmosphäre
- ✅ **Battery-Friendly:** Keine permanenten Background-Listener

---

2.1 Datenmodell-Erweiterungen

User-Dokument: `clubs/{clubId}/users/{uid}`

Neue Felder für Club-Status:

Feldname	Datentyp	Beschreibung	Beispielwert
`isInClub`	`boolean`	Physisch im Club eingecheckt?	`true`
`currentClubId`	`string | null`	ID des Clubs, in dem User eingecheckt ist	`"club_abc123"`
`checkedInAt`	`number | null`	Timestamp des Check-ins (Unix ms)	`1701388800000`

Neue Felder für Party-Modus:

Feldname	Datentyp	Beschreibung	Beispielwert
`partyModeActive`	`boolean`	Aktiv an DJ-Aktionen teilnehmend?	`true`
`partyModeLastActivatedAt`	`number | null`	Timestamp der letzten Aktivierung	`1701388800000`
`selectedNumber`	`number | null`	Aktuell gewählte Zahl (1-99)	`42`

2.2 Die 6 Interaktions-Zahlen

Club-Day-Config: `clubs/{clubId}/state/global`

Neue Felder für tägliche Zahlen:

Feldname	Datentyp	Beschreibung	Beispielwert
`dailyInteractionNumbers`	`number[]`	6 zufällige Zahlen (1-99)	`[7, 23, 42, 55, 68, 91]`
`dailyNumbersGeneratedAt`	`number | null`	Timestamp der Generierung	`1701388800000`

Generierungs-Logik

Wann werden die Zahlen generiert?
- Einmal pro Tag zur **Öffnungszeit** des Clubs (definiert in `clubs/{clubId}/config/settings.openingHours`)
- Automatisch per **Cloud Function** (Cron-Job)
- Oder manuell durch Club-Admin/DJ

Wie werden die Zahlen generiert?
- 6 **eindeutige**, zufällige Zahlen zwischen 1 und 99
- Algorithmus: Fisher-Yates-Shuffle auf Pool [1…99]
- Speicherung in `clubs/{clubId}/state/global`

Gültigkeit:
- Bis zur **Sperrstunde** des Clubs (z.B. 05:00 Uhr)
- Danach: Neue Zahlen werden beim nächsten Öffnen generiert

2.3 Kernregeln & Business Logic

Regel 1: Check-In als Voraussetzung

```
isInClub = true → User kann Party-Modus aktivieren
isInClub = false → Kein Party-Modus möglich
```

Wer setzt `isInClub`?
- **Türsteher** (via QR-Scanner in `staff-door` App)
- **User selbst** (via QR-Code-Scan, nur wenn Geo-Location stimmt)
- **Kassensystem** (via API-Integration)

Regel 2: Party-Modus-Aktivierung

```
User wählt eine der 6 Zahlen → partyModeActive = true
```

Ablauf:
1. User öffnet App
2. Sieht die 6 Zahlen des Tages
3. Wählt eine Zahl aus (z.B. `42`)
4. System setzt:
- `partyModeActive = true`
- `selectedNumber = 42`
- `partyModeLastActivatedAt = Date.now()`

Regel 3: 30-Minuten-Timer

```
partyModeActive = true → Timer startet (30 Minuten)
Keine erneute Zahlen-Eingabe innerhalb 30 Minuten → partyModeActive = false
```

Implementierungs-Strategien:

Option A: Client-seitiger Timer (Empfohlen für Phase 9)

- **Vorteil:** Keine zusätzlichen Cloud Functions
- **Nachteil:** User kann Timer manipulieren (aber ohne Vorteil, da DJ-Aktionen serverseitig validiert werden)
- **Implementierung:**
  ```typescript
  // In club-app
  useEffect(() => {
  if (userData.partyModeActive) {
  const elapsed = Date.now() - userData.partyModeLastActivatedAt;
  const remaining = PARTY_MODE_DURATION - elapsed; // 30 min

  if (remaining <= 0) {
  // Abgelaufen → Deaktivieren
  updateDoc(userDocRef, { partyModeActive: false });
  } else {
  // Timer setzen
  const timeout = setTimeout(() => {
  ```

```
updateDoc(userDocRef, { partyModeActive: false });
}, remaining);
return () => clearTimeout(timeout);
}
}
}, [userData.partyModeActive]);
```

Option B: Lazy Server-Side Check (Empfohlen für Production)

- **Vorteil:** Manipulation-sicher
- **Implementierung:** Bei jeder DJ-Aktion prüft die Cloud Function:
  ```typescript
  function isPartyModeValid(user: UserDocument): boolean {
    const elapsed = Date.now() - user.partyModeLastActivatedAt;
    return user.partyModeActive && elapsed < 30 * 60 * 1000;
  }
  ```

Option C: Cloud Scheduler (Overkill für Phase 9)

- **Vorteil:** Exakte Zeitsteuerung
- **Nachteil:** Hohe Kosten bei vielen Usern
- **Nicht empfohlen** für dieses Feature

Regel 4: Erneute Aktivierung

```
User wählt erneut eine Zahl → Timer wird zurückgesetzt
```

Ablauf:
1. User hört neue DJ-Ansage
2. Öffnet App
3. Wählt neue Zahl (kann dieselbe oder eine andere sein)
4. `partyModeLastActivatedAt` wird aktualisiert → Timer startet neu

Regel 5: DJ-Aktion & Match-Logik

```
DJ wählt Zahl X → Alle User mit selectedNumber === X && partyModeActive === true
nehmen teil
```

Serverseitige Validierung (Cloud Function):

```
async function triggerDJAction(clubId: string, djSelectedNumber: number, actionType:
string) {
  // 1. Hole alle aktiven Party-Mode-User mit der richtigen Zahl
  const participants = await db
    .collection(`clubs/${clubId}/users`)
    .where('isInClub', '==', true)
    .where('partyModeActive', '==', true)
    .where('selectedNumber', '==', djSelectedNumber)
    .get();

  // 2. Validiere 30-Minuten-Regel
  const validParticipants = participants.docs.filter(doc => {
    const user = doc.data();
    const elapsed = Date.now() - user.partyModeLastActivatedAt;
    return elapsed < 30 * 60 * 1000;
  });

  // 3. Führe Aktion aus (z.B. Gewinnspiel, Lichtshow)
  if (actionType === 'lottery') {
    const winners = selectRandomWinners(validParticipants, 3);
    await updateDoc(doc(`clubs/${clubId}/state/global`), {
      mode: 'lottery_result',
      winnerIds: winners.map(w => w.id)
    });
  }

  // 4. (Optional) Setze Party-Modus nach Aktion zurück
  // → Nein, Timer läuft weiter! User kann an mehreren Aktionen teilnehmen.
}
```

2.4 Funktionen & Interfaces

TypeScript Interfaces

```typescript
// packages/shared-types/src/user.ts

export interface UserPartyModeFields {
  /** Physisch im Club eingecheckt? */
  isInClub: boolean;

  /** ID des Clubs, in dem User eingecheckt ist */
  currentClubId: string | null;

  /** Check-In-Timestamp (Unix ms) */
  checkedInAt: number | null;

  /** Aktiv an DJ-Aktionen teilnehmend? */
  partyModeActive: boolean;

  /** Timestamp der letzten Party-Modus-Aktivierung */
  partyModeLastActivatedAt: number | null;

  /** Aktuell gewählte Zahl (1-99), eine der 6 täglichen Zahlen */
  selectedNumber: number | null;
}

// Erweitere bestehende UserDocument-Interface
export interface UserDocument extends UserPartyModeFields {
  uid: string;
  email: string;
  displayName: string | null;
  photoURL: string | null;
  roles: string[];
  checkedIn: boolean; // ← DEPRECATED, wird durch isInClub ersetzt
  // ... weitere bestehende Felder
}
```

```typescript
// packages/shared-types/src/club.ts

export interface ClubDailyNumbers {
  /** 6 zufällige, eindeutige Zahlen für den heutigen Abend (1-99) */
  dailyInteractionNumbers: number[];

  /** Timestamp der Generierung (Unix ms) */
  dailyNumbersGeneratedAt: number | null;
}

// Erweitere bestehende ClubStateDocument-Interface
export interface ClubStateDocument extends ClubDailyNumbers {
  mode: 'normal' | 'lightshow' | 'message' | 'countdown' | 'lottery_result';
  lightColor: string | null;
  // ... weitere bestehende Felder
}
```

Core Functions (Pseudocode)

```typescript
// packages/core/src/utils/party-mode.ts

/**
 * Aktiviert den Party-Modus für einen User
 * @param userId - Firebase UID des Users
 * @param clubId - Club-ID
 * @param selectedNumber - Gewählte Zahl (muss eine der 6 täglichen Zahlen sein)
 * @returns Promise<void>
 * @throws Error wenn User nicht eingecheckt oder Zahl ungültig
 */
export async function activatePartyMode(
  userId: string,
  clubId: string,
  selectedNumber: number
): Promise<void> {
  // 1. Validierung: User muss eingecheckt sein
  const userDoc = await getDoc(doc(db, `clubs/${clubId}/users/${userId}`));
  const user = userDoc.data();

  if (!user.isInClub) {
    throw new Error('User ist nicht im Club eingecheckt');
  }

  // 2. Validierung: Zahl muss eine der 6 täglichen Zahlen sein
  const clubStateDoc = await getDoc(doc(db, `clubs/${clubId}/state/global`));
  const clubState = clubStateDoc.data();

  if (!clubState.dailyInteractionNumbers.includes(selectedNumber)) {
    throw new Error('Ungültige Zahl gewählt');
  }

  // 3. Aktivierung
  await updateDoc(doc(db, `clubs/${clubId}/users/${userId}`), {
    partyModeActive: true,
    selectedNumber: selectedNumber,
    partyModeLastActivatedAt: Date.now()
  });
}

/**
 * Deaktiviert den Party-Modus (manuell oder nach Ablauf)
 */
export async function deactivatePartyMode(
  userId: string,
  clubId: string
): Promise<void> {
  await updateDoc(doc(db, `clubs/${clubId}/users/${userId}`), {
    partyModeActive: false,
    selectedNumber: null
  });
}

/**
 * Prüft, ob der Party-Modus noch gültig ist (30-Minuten-Regel)
 */
export function isPartyModeExpired(user: UserDocument): boolean {
  if (!user.partyModeActive || !user.partyModeLastActivatedAt) {
    return true;
  }

  const elapsed = Date.now() - user.partyModeLastActivatedAt;
  const THIRTY_MINUTES = 30 * 60 * 1000;
```

```
  return elapsed >= THIRTY_MINUTES;
}

/**
 * Generiert 6 zufällige, eindeutige Zahlen für den Tag
 */
export function generateDailyNumbers(): number[] {
  const pool = Array.from({ length: 99 }, (_, i) => i + 1); // [1...99]

  // Fisher-Yates-Shuffle
  for (let i = pool.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));
    [pool[i], pool[j]] = [pool[j], pool[i]];
  }

  // Nimm die ersten 6 Zahlen
  return pool.slice(0, 6).sort((a, b) => a - b);
}

/**
 * Speichert die täglichen Zahlen in Firestore
 * (Wird von Cloud Function oder Admin-Panel aufgerufen)
 */
export async function saveDailyNumbers(clubId: string): Promise<void> {
  const numbers = generateDailyNumbers();

  await updateDoc(doc(db, `clubs/${clubId}/state/global`), {
    dailyInteractionNumbers: numbers,
    dailyNumbersGeneratedAt: Date.now()
  });
}

/**
 * Findet alle User, die an einer DJ-Aktion teilnehmen
 * (Serverseitige Funktion für Cloud Functions)
 */
export async function findParticipants(
  clubId: string,
  djSelectedNumber: number
): Promise<UserDocument[]> {
  // 1. Query: Alle User mit der richtigen Zahl
  const snapshot = await getDocs(
    query(
      collection(db, `clubs/${clubId}/users`),
      where('isInClub', '==', true),
      where('partyModeActive', '==', true),
      where('selectedNumber', '==', djSelectedNumber)
    )
  );

  // 2. Filter: 30-Minuten-Regel
  const validUsers = snapshot.docs
    .map(doc => doc.data() as UserDocument)
    .filter(user => !isPartyModeExpired(user));

  return validUsers;
}
```

2.5 Firestore-Schema

Collection: `clubs/{clubId}/users/{uid}`

Erweiterte Felder:

```
{
  // ... bestehende Felder (uid, email, displayName, roles, etc.)

  // NEU: Club-Status
  "isInClub": true,                          // boolean
  "currentClubId": "club_abc123",            // string | null
  "checkedInAt": 1701388800000,              // number | null (Unix ms)

  // NEU: Party-Modus
  "partyModeActive": true,                   // boolean
  "partyModeLastActivatedAt": 1701388800000, // number | null (Unix ms)
  "selectedNumber": 42                       // number | null (1-99)
}
```

Collection: `clubs/{clubId}/state/global`

Erweiterte Felder:

```
{
  // ... bestehende Felder (mode, lightColor, countdownActive, etc.)

  // NEU: Tägliche Interaktions-Zahlen
  "dailyInteractionNumbers": [7, 23, 42, 55, 68, 91],  // number[] (6 Zahlen)
  "dailyNumbersGeneratedAt": 1701388800000             // number | null (Unix ms)
}
```

Firestore Queries

Query 1: Alle aktiven Party-Mode-User mit Zahl X

```
const participants = await getDocs(
  query(
    collection(db, `clubs/${clubId}/users`),
    where('isInClub', '==', true),
    where('partyModeActive', '==', true),
    where('selectedNumber', '==', djSelectedNumber)
  )
);
```

Query 2: Alle eingecheckten User

```
const checkedInUsers = await getDocs(
  query(
    collection(db, `clubs/${clubId}/users`),
    where('isInClub', '==', true)
  )
);
```

Benötigte Firestore Composite Indexes:

```
clubs/{clubId}/users
  - isInClub (ASC) + partyModeActive (ASC) + selectedNumber (ASC)
```

---

3. DATING-STATUS & 6-STUNDEN-REGEL

Konzept

Der **Dating-Status** ermöglicht es Usern, ihren Beziehungsstatus anzuzeigen (ähnlich wie Tinder/Bumble). Clubs können diesen Status für Promotions nutzen (z.B. “Singles trinken heute gratis”).

Problem: Missbrauch

User könnten ihren Status ständig ändern, um wiederholt Benefits zu bekommen.

Lösung: 6-Stunden-Regel

Änderungen sind nur alle **6 Stunden** möglich. Dies wird **serverseitig** validiert (nicht umgehbar).

3.1 Datenmodell-Erweiterungen

User-Dokument: `clubs/{clubId}/users/{uid}`

Neue Felder für Dating-Status:

Feldname	Datentyp	Beschreibung	Beispielwert
`relationshipStatus`	`'single' | 'taken' | 'complicated' | 'dont_touch' | null`	Beziehungsstatus	`"single"`
`relationshipStatusLastChangedAt`	`number | null`	Timestamp der letzten Änderung (Unix ms)	`1701388800000`

Status-Definitionen

Status	Bedeutung	Anzeige-Farbe (Beispiel)
`single`	Single, offen für Kontakte	🟢 Grün
`taken`	Vergeben, nicht interessiert	🔴 Rot
`complicated`	Es ist kompliziert	🟡 Gelb
`dont_touch`	Bitte nicht ansprechen	🔵 Blau
`null`	Kein Status gesetzt	⚪ Grau

3.2 Kernregeln & Validierung

Regel 1: 6-Stunden-Sperre

```
Letzter Wechsel vor < 6 Stunden → Änderung NICHT erlaubt
Letzter Wechsel vor ≥ 6 Stunden → Änderung erlaubt
```

Regel 2: Erste Festlegung ist sofort möglich

```
relationshipStatusLastChangedAt === null → Änderung erlaubt
```

Regel 3: Serverseitige Validierung (KRITISCH!)

Die Validierung MUSS serverseitig erfolgen, da Client-seitige Checks umgehbar sind.

Option A: Firestore Security Rules (Empfohlen)

```
// In firestore.rules
match /clubs/{clubId}/users/{uid} {
  allow update: if request.auth.uid == uid && (
    // Erlaubt: Keine relationshipStatus-Änderung
    !request.resource.data.diff(resource.data).affectedKeys().hasAny(['relationship-
Status']) ||

    // Erlaubt: Erste Festlegung
    !resource.data.relationshipStatusLastChangedAt ||

    // Erlaubt: 6 Stunden sind vergangen
    (request.time.toMillis() - resource.data.relationshipStatusLastChangedAt) >= (6 *
60 * 60 * 1000)
  );
}
```

Option B: Cloud Function (Alternative)

```
// Cloud Function: onUserUpdateRequest
export const validateRelationshipStatusChange = functions.firestore
  .document('clubs/{clubId}/users/{uid}')
  .onUpdate(async (change, context) => {
    const before = change.before.data();
    const after = change.after.data();

    // Prüfe ob relationshipStatus geändert wurde
    if (before.relationshipStatus !== after.relationshipStatus) {
      const lastChanged = before.relationshipStatusLastChangedAt;
      const now = Date.now();

      // Erste Festlegung?
      if (!lastChanged) {
        return; // OK
      }

      // 6 Stunden vergangen?
      const elapsed = now - lastChanged;
      const SIX_HOURS = 6 * 60 * 60 * 1000;

      if (elapsed < SIX_HOURS) {
        // REVERT CHANGE!
        await change.after.ref.update({
          relationshipStatus: before.relationshipStatus
        });

        throw new functions.https.HttpsError(
          'permission-denied',
          'Statusänderung erst in ' + Math.ceil((SIX_HOURS - elapsed) / 1000 / 60) +
' Minuten möglich'
        );
      }
    }
  });
```

Empfehlung: Nutze **Security Rules** (Option A), da diese Firestore-nativ sind und keine zusätzlichen Cloud Function-Kosten verursachen.

3.3 Funktionen & Interfaces

TypeScript Interfaces

```typescript
// packages/shared-types/src/user.ts

export type RelationshipStatus = 'single' | 'taken' | 'complicated' | 'dont_touch';

export interface UserRelationshipFields {
  /** Beziehungsstatus (Dating-Feature) */
  relationshipStatus: RelationshipStatus | null;

  /** Timestamp der letzten Statusänderung (Unix ms) */
  relationshipStatusLastChangedAt: number | null;
}

// Erweitere UserDocument
export interface UserDocument extends UserRelationshipFields {
  // ... bestehende Felder
}
```

Core Functions (Pseudocode)

```
// packages/core/src/utils/relationship-status.ts

const SIX_HOURS_MS = 6 * 60 * 60 * 1000;

/**
 * Prüft, ob der User seinen Relationship-Status ändern kann
 * @returns true wenn Änderung erlaubt, false wenn noch gesperrt
 */
export function canChangeRelationshipStatus(user: UserDocument): boolean {
  // Erste Festlegung?
  if (!user.relationshipStatusLastChangedAt) {
    return true;
  }

  // 6 Stunden vergangen?
  const elapsed = Date.now() - user.relationshipStatusLastChangedAt;
  return elapsed >= SIX_HOURS_MS;
}

/**
 * Gibt die verbleibende Sperrzeit in Minuten zurück
 * @returns Minuten bis zur nächsten Änderung (0 wenn Änderung möglich)
 */
export function getRemainingCooldown(user: UserDocument): number {
  if (canChangeRelationshipStatus(user)) {
    return 0;
  }

  const elapsed = Date.now() - user.relationshipStatusLastChangedAt!;
  const remaining = SIX_HOURS_MS - elapsed;
  return Math.ceil(remaining / 1000 / 60); // Minuten
}

/**
 * Ändert den Relationship-Status (mit Client-seitiger Validierung)
 * WICHTIG: Serverseitige Validierung via Security Rules ist zusätzlich aktiv!
 */
export async function updateRelationshipStatus(
  userId: string,
  clubId: string,
  newStatus: RelationshipStatus | null
): Promise<void> {
  // Client-seitige Pre-Check (UI-Feedback)
  const userDoc = await getDoc(doc(db, `clubs/${clubId}/users/${userId}`));
  const user = userDoc.data() as UserDocument;

  if (!canChangeRelationshipStatus(user)) {
    const remaining = getRemainingCooldown(user);
    throw new Error(`Statusänderung erst in ${remaining} Minuten möglich`);
  }

  // Update (Security Rules prüfen serverseitig nochmal!)
  await updateDoc(doc(db, `clubs/${clubId}/users/${userId}`), {
    relationshipStatus: newStatus,
    relationshipStatusLastChangedAt: Date.now()
  });
}
```

3.4 Firestore-Schema

Collection: `clubs/{clubId}/users/{uid}`

Erweiterte Felder:

```
{
  // ... bestehende Felder

  // NEU: Dating-Status
  "relationshipStatus": "single",                    // "single" | "taken" | "complic-
ated" | "dont_touch" | null
  "relationshipStatusLastChangedAt": 1701388800000   // number | null (Unix ms)
}
```

Firestore Security Rules

```
// In firestore.rules
match /clubs/{clubId}/users/{uid} {
  allow update: if request.auth.uid == uid && (
    // Fall 1: relationshipStatus wird NICHT geändert → OK
    !request.resource.data.diff(resource.data).affectedKeys().hasAny(['relationship-
Status']) ||

    // Fall 2: Erste Festlegung (noch nie gesetzt) → OK
    resource.data.relationshipStatusLastChangedAt == null ||

    // Fall 3: 6 Stunden sind vergangen → OK
    (request.time.toMillis() - resource.data.relationshipStatusLastChangedAt) >= 21600
000 // 6h in ms
  );
}
```

---

4. TRUSTED USER & ZECHPRELLER-SCHUTZ

Konzept

Das **Trusted User-System** ist ein zweistufiger Sicherheitsmechanismus gegen Zechpreller und Missbrauch:

Stufe 1: Phone-Verifizierung (Pflicht für Bestellungen)

User muss Telefonnummer per SMS verifizieren, bevor er Bestellungen aufgeben kann.

Stufe 2: Trusted User (Türsteher-Verifizierung)

Nach einmaliger persönlicher Verifizierung durch Türsteher (z.B. Ausweis-Check) erhält der User den Status **“Trusted”**, der **global in allen Clubs** des Netzwerks gültig ist.

Warum global?

- ✅ **User-Friendly:** Einmal verifiziert, nie wieder Probleme
- ✅ **Club-Network-Effect:** Clubs vertrauen einander

- ✅ **Blacklist-Sharing:** Zechpreller können nicht einfach in anderen Clubs weiter betrügen

---

4.1 Datenmodell-Erweiterungen

User-Dokument: `clubs/{clubId}/users/{uid}` (Club-spezifisch)

Neue Felder für Trust-System:

Feldname	Datentyp	Beschreibung	Beispielwert
`phoneVerified`	`boolean`	Telefonnummer per SMS verifiziert?	`true`
`trustLevel`	`'normal' | 'trusted'`	Trust-Level des Users	`"trusted"`
`trustedVerifiedAt`	`number | null`	Timestamp der Trusted-Verifizierung	`1701388800000`
`trustedVerifiedByClubId`	`string | null`	Club-ID, in dem Verifizierung stattfand	`"club_abc123"`

WICHTIG: `trustLevel` wird in **allen** Club-Dokumenten des Users synchronisiert!

User-Dokument: `platform/users/{uid}` (Global)

Neue Felder für globales Trust-System:

Feldname	Datentyp	Beschreibung	Beispielwert
`phoneNumber`	`string | null`	Verifizierte Telefonnummer	`"+491234567890"`
`phoneVerified`	`boolean`	Telefonnummer verifiziert?	`true`
`globalTrustLevel`	`'normal' | 'trusted'`	Globaler Trust-Level	`"trusted"`
`trustedVerifiedAt`	`number | null`	Timestamp der Verifizierung	`1701388800000`
`trustedVerifiedByClubId`	`string | null`	Club, in dem Verifizierung stattfand	`"club_abc123"`
`trustedVerifiedByStaffId`	`string | null`	Staff-UID, der verifiziert hat	`"staff_door1"`

4.2 Kernregeln & Trust-Mechanismus

Regel 1: Phone-Verifizierung als Basis

```
phoneVerified = false → Keine Bestellungen möglich
phoneVerified = true → Bestellungen erlaubt (mit Limits)
```

Verifizierungs-Flow:

1. User gibt Telefonnummer in App ein
2. Firebase Authentication sendet SMS mit Code
3. User gibt Code ein
4. System setzt `phoneVerified = true` (in `platform/users/{uid}` UND `clubs/{clubId}/users/{uid}`)

Regel 2: Trusted-Status durch Türsteher

```
trustLevel = 'normal' → Bestellungen mit Limits (z.B. max. 50€)
trustLevel = 'trusted' → Unbegrenzte Bestellungen, Rechnungskauf
```

Verifizierungs-Flow:

1. Türsteher scannt User-QR-Code (oder sucht User in Liste)
2. Türsteher prüft Ausweis (Alter, Identität)
3. Türsteher klickt “Als Trusted markieren”
4. System setzt:

- `platform/users/{uid}.globalTrustLevel = 'trusted'`
- `platform/users/{uid}.trustedVerifiedAt = Date.now()`
- `platform/users/{uid}.trustedVerifiedByClubId = clubId`
- `platform/users/{uid}.trustedVerifiedByStaffId = staffUid`

5. **Cloud Function synchronisiert** `trustLevel = 'trusted'` in **alle** Club-Dokumente des Users

Regel 3: Trust-Level ist global gültig

```
User wird in Club A als Trusted markiert
→ In Club B, C, D, ... ist er automatisch auch Trusted
```

Implementierung:

```
// Cloud Function: onTrustLevelChange
export const syncTrustLevel = functions.firestore
  .document('platform/users/{uid}')
  .onUpdate(async (change, context) => {
    const before = change.before.data();
    const after = change.after.data();

    // Prüfe ob globalTrustLevel geändert wurde
    if (before.globalTrustLevel !== after.globalTrustLevel) {
      const uid = context.params.uid;
      const newTrustLevel = after.globalTrustLevel;

      // Hole alle Clubs, in denen User Mitglied ist
      const memberClubs = after.memberClubs || [];

      // Update in allen Club-Dokumenten
      const batch = db.batch();
      for (const clubId of memberClubs) {
        const clubUserRef = db.doc(`clubs/${clubId}/users/${uid}`);
        batch.update(clubUserRef, { trustLevel: newTrustLevel });
      }
      await batch.commit();
    }
  });
```

Regel 4: Bestellungs-Validierung

```
Bestellung wird aufgegeben → Prüfe phoneVerified UND trustLevel
```

Implementierung (Pseudocode):

```
function canPlaceOrder(user: UserDocument): { allowed: boolean; reason?: string } {
  // 1. Phone-Verifizierung
  if (!user.phoneVerified) {
    return { allowed: false, reason: 'Telefonnummer muss verifiziert werden' };
  }

  // 2. Trust-Level-Check (optional, für Limits)
  if (user.trustLevel === 'normal') {
    // Prüfe Bestellhistorie → Max. 50€ pro Abend
    // (Wird später in Order-System implementiert)
  }

  return { allowed: true };
}
```

4.3 Funktionen & Interfaces

TypeScript Interfaces

```typescript
// packages/shared-types/src/user.ts

export type TrustLevel = 'normal' | 'trusted';

export interface UserTrustFields {
  /** Telefonnummer per SMS verifiziert? */
  phoneVerified: boolean;

  /** Trust-Level des Users */
  trustLevel: TrustLevel;

  /** Timestamp der Trusted-Verifizierung (Unix ms) */
  trustedVerifiedAt: number | null;

  /** Club-ID, in dem Verifizierung stattfand */
  trustedVerifiedByClubId: string | null;
}

// Erweitere UserDocument (club-spezifisch)
export interface UserDocument extends UserTrustFields {
  // ... bestehende Felder
}

// Globale User-Felder (platform/users/{uid})
export interface PlatformUser extends UserTrustFields {
  uid: string;
  email: string;
  phoneNumber: string | null;
  phoneVerified: boolean;
  globalTrustLevel: TrustLevel;
  trustedVerifiedByStaffId: string | null;
  memberClubs: string[];
  // ... weitere Felder
}
```

Core Functions (Pseudocode)

```typescript
// packages/core/src/utils/trust-system.ts

/**
 * Verifiziert die Telefonnummer des Users via Firebase Auth
 * @param phoneNumber - Telefonnummer im E.164-Format (+491234567890)
 */
export async function verifyPhone(userId: string, phoneNumber: string): Promise<void>
{
  // 1. Firebase Auth: Send SMS
  // (Implementierung mit Firebase Phone Authentication)

  // 2. Nach erfolgreicher Verifizierung: Update in Firestore
  await updateDoc(doc(db, `platform/users/${userId}`), {
    phoneNumber: phoneNumber,
    phoneVerified: true
  });

  // 3. Sync in alle Club-Dokumente
  const platformUser = await getDoc(doc(db, `platform/users/${userId}`));
  const memberClubs = platformUser.data()?.memberClubs || [];

  const batch = writeBatch(db);
  for (const clubId of memberClubs) {
    batch.update(doc(db, `clubs/${clubId}/users/${userId}`), {
      phoneVerified: true
    });
  }
  await batch.commit();
}

/**
 * Setzt den Trust-Level eines Users auf "Trusted"
 * Wird von Türsteher-App aufgerufen
 * @param userId - User-UID
 * @param staffId - Staff-UID (Türsteher)
 * @param clubId - Club-ID, in dem Verifizierung stattfindet
 */
export async function setTrustedLevel(
  userId: string,
  staffId: string,
  clubId: string
): Promise<void> {
  // 1. Update in Platform-Dokument (global)
  await updateDoc(doc(db, `platform/users/${userId}`), {
    globalTrustLevel: 'trusted',
    trustedVerifiedAt: Date.now(),
    trustedVerifiedByClubId: clubId,
    trustedVerifiedByStaffId: staffId
  });

  // 2. Cloud Function synchronisiert automatisch in alle Club-Dokumente
  // (siehe Regel 3)
}

/**
 * Prüft, ob User Bestellungen aufgeben kann
 */
export function canPlaceOrder(user: UserDocument): { allowed: boolean; reason?:
string } {
  if (!user.phoneVerified) {
    return {
      allowed: false,
```

```
      reason: 'Bitte verifiziere zuerst deine Telefonnummer in den Einstellungen'
    };
  }

  return { allowed: true };
}

/**
 * Prüft, ob User Rechnungskauf nutzen kann (nur Trusted)
 */
export function canPayOnAccount(user: UserDocument): boolean {
  return user.phoneVerified && user.trustLevel === 'trusted';
}
```

4.4 Firestore-Schema

Collection: `platform/users/{uid}` (Global)

Erweiterte Felder:

```
{
  // ... bestehende Felder (uid, email, displayName, etc.)

  // NEU: Phone-Verifizierung
  "phoneNumber": "+491234567890",            // string | null
  "phoneVerified": true,                     // boolean

  // NEU: Globales Trust-System
  "globalTrustLevel": "trusted",             // "normal" | "trusted"
  "trustedVerifiedAt": 1701388800000,        // number | null (Unix ms)
  "trustedVerifiedByClubId": "club_abc123",  // string | null
  "trustedVerifiedByStaffId": "staff_door1", // string | null

  // Bestehend (für Sync)
  "memberClubs": ["club_abc123", "club_def456"]  // string[]
}
```

Collection: `clubs/{clubId}/users/{uid}` (Club-spezifisch)

Erweiterte Felder:

```
{
  // ... bestehende Felder

  // NEU: Trust-Felder (synchronisiert aus Platform)
  "phoneVerified": true,                     // boolean
  "trustLevel": "trusted",                   // "normal" | "trusted"
  "trustedVerifiedAt": 1701388800000,        // number | null
  "trustedVerifiedByClubId": "club_abc123"   // string | null
}
```

---

5. CLUB-PLÄNE & FEATURE-FLAGS

Konzept

Das **Club-Plan-System** ermöglicht verschiedene Abo-Modelle mit unterschiedlichen Features. Dies ist die Monetarisierungs-Strategie von Nightlife OS.

Plan-Modelle

Plan A: Free (mit Werbung)

- ✅ Basis-Features: Chat, Check-In, Friends
- ✅ Lichtshow (basic)
- ❌ Kein Party Alarm
- ❌ Kein Hunt the Fox
- ❌ Keine erweiterten Analytics
- ❌ Kein White-Label-Branding
- 📢 **Mit Werbeanzeigen** (In-App-Ads)

Plan B: Premium (werbefrei)

- ✅ Alle Free-Features
- ✅ Party Alarm
- ✅ Hunt the Fox
- ✅ Erweiterte Analytics
- ✅ White-Label-Branding (eigenes Logo, Farben)
- ✅ **Keine Werbeanzeigen**
- ✅ Priority-Support

5.1 Datenmodell-Erweiterungen

Club-Dokument: `platform/clubs/{clubId}`

Erweiterte Felder für Pläne:

Feldname	Datentyp	Beschreibung	Beispielwert
`plan`	`'free' | 'premium'`	Aktueller Plan	`"premium"`
`featureFlags`	`{ [key: string]: boolean }`	Feature-Schalter	`{ adsEnabled: false, partyAlarmEnabled: true }`

Feature-Flag-Definitionen

Feature-Key	Beschreibung	Free	Premium
`adsEnabled`	In-App-Werbung anzeigen	✅	❌
`partyAlarmEnabled`	Party Alarm-Feature	❌	✅
`huntTheFoxEnabled`	Hunt the Fox-Game	❌	✅
`advancedAnalyticsEnabled`	Erweiterte Analytics	❌	✅
`whiteLabelBrandingEnabled`	Eigenes Branding	❌	✅
`prioritySupportEnabled`	Priority-Support	❌	✅
`customNotificationsEnabled`	Custom Push-Notifications	❌	✅
`guestLimitUnlimited`	Unbegrenzte Gästeanzahl	❌	✅

5.2 Plan-Definitionen

TypeScript Interfaces

```typescript
// packages/shared-types/src/club.ts

export type ClubPlan = 'free' | 'premium';

export interface FeatureFlags {
  /** In-App-Werbung anzeigen? */
  adsEnabled?: boolean;

  /** Party Alarm-Feature verfügbar? */
  partyAlarmEnabled?: boolean;

  /** Hunt the Fox-Game verfügbar? */
  huntTheFoxEnabled?: boolean;

  /** Erweiterte Analytics verfügbar? */
  advancedAnalyticsEnabled?: boolean;

  /** White-Label-Branding verfügbar? */
  whiteLabelBrandingEnabled?: boolean;

  /** Priority-Support verfügbar? */
  prioritySupportEnabled?: boolean;

  /** Custom Push-Notifications verfügbar? */
  customNotificationsEnabled?: boolean;

  /** Unbegrenzte Gästeanzahl? */
  guestLimitUnlimited?: boolean;
}

// Erweitere Club-Interface
export interface Club {
  clubId: string;
  name: string;
  slug: string;
  ownerId: string;

  // NEU: Plan & Features
  plan: ClubPlan;
  featureFlags?: FeatureFlags;

  // Bestehende Felder
  subscriptionTier: string;
  subscriptionStatus: string;
  // ... weitere Felder
}
```

Default Feature-Flags

```
// packages/core/src/constants/plans.ts

export const DEFAULT_FEATURE_FLAGS: Record<ClubPlan, FeatureFlags> = {
  free: {
    adsEnabled: true,
    partyAlarmEnabled: false,
    huntTheFoxEnabled: false,
    advancedAnalyticsEnabled: false,
    whiteLabelBrandingEnabled: false,
    prioritySupportEnabled: false,
    customNotificationsEnabled: false,
    guestLimitUnlimited: false
  },
  premium: {
    adsEnabled: false,
    partyAlarmEnabled: true,
    huntTheFoxEnabled: true,
    advancedAnalyticsEnabled: true,
    whiteLabelBrandingEnabled: true,
    prioritySupportEnabled: true,
    customNotificationsEnabled: true,
    guestLimitUnlimited: true
  }
};
```

5.3 Funktionen & Interfaces

Core Functions (Pseudocode)

```typescript
// packages/core/src/utils/feature-flags.ts

/**
 * Prüft, ob ein Feature für einen Club verfügbar ist
 * @param clubId - Club-ID
 * @param featureKey - Feature-Schlüssel (z.B. "partyAlarmEnabled")
 * @returns true wenn Feature aktiviert, false wenn deaktiviert
 */
export async function isFeatureEnabled(
  clubId: string,
  featureKey: keyof FeatureFlags
): Promise<boolean> {
  // 1. Hole Club-Dokument
  const clubDoc = await getDoc(doc(db, `platform/clubs/${clubId}`));
  const club = clubDoc.data() as Club;

  // 2. Prüfe ob Feature-Flag explizit gesetzt ist
  if (club.featureFlags && featureKey in club.featureFlags) {
    return club.featureFlags[featureKey] ?? false;
  }

  // 3. Fallback: Default für Plan
  const defaults = DEFAULT_FEATURE_FLAGS[club.plan];
  return defaults[featureKey] ?? false;
}

/**
 * Gibt den aktuellen Plan eines Clubs zurück
 */
export async function getClubPlan(clubId: string): Promise<ClubPlan> {
  const clubDoc = await getDoc(doc(db, `platform/clubs/${clubId}`));
  const club = clubDoc.data() as Club;
  return club.plan || 'free';
}

/**
 * Ändert den Plan eines Clubs (nur für Super-Admins/Billing-System)
 */
export async function updateClubPlan(
  clubId: string,
  newPlan: ClubPlan
): Promise<void> {
  const defaultFlags = DEFAULT_FEATURE_FLAGS[newPlan];

  await updateDoc(doc(db, `platform/clubs/${clubId}`), {
    plan: newPlan,
    featureFlags: defaultFlags
  });
}

/**
 * Setzt einen einzelnen Feature-Flag (für Custom-Konfigurationen)
 */
export async function setFeatureFlag(
  clubId: string,
  featureKey: keyof FeatureFlags,
  value: boolean
): Promise<void> {
  await updateDoc(doc(db, `platform/clubs/${clubId}`), {
    [`featureFlags.${featureKey}`]: value
  });
}
```

Verwendung in Components

```
// Example: club-app/src/components/party-alarm-button.tsx

import { isFeatureEnabled } from '@nightlife-os/core';

export function PartyAlarmButton({ clubId }: { clubId: string }) {
  const [featureEnabled, setFeatureEnabled] = useState(false);

  useEffect(() => {
    isFeatureEnabled(clubId, 'partyAlarmEnabled').then(setFeatureEnabled);
  }, [clubId]);

  if (!featureEnabled) {
    return (
      <UpgradePrompt
        feature="Party Alarm"
        message="Upgrade auf Premium, um Party Alarm zu nutzen!"
      />
    );
  }

  return <button onClick={triggerPartyAlarm}>🚨 Party Alarm!</button>;
}
```

5.4 Firestore-Schema

Collection: `platform/clubs/{clubId}`

Erweiterte Felder:

```
{
  // ... bestehende Felder (clubId, name, ownerId, etc.)

  // NEU: Plan & Features
  "plan": "premium",                          // "free" | "premium"
  "featureFlags": {
    "adsEnabled": false,
    "partyAlarmEnabled": true,
    "huntTheFoxEnabled": true,
    "advancedAnalyticsEnabled": true,
    "whiteLabelBrandingEnabled": true,
    "prioritySupportEnabled": true,
    "customNotificationsEnabled": true,
    "guestLimitUnlimited": true
  }
}
```

---

6. EVENT-LIFECYCLE: VOR / WÄHREND / NACH DEM EVENT

Konzept

Das **Event-Lifecycle-Modell** strukturiert alle zeitbasierten Features rund um Club-Events:

- **Vor dem Event:** Ticketkauf, Fast-Lane-Reservierung, Reminder-Notifications
- **Während des Events:** Party Alarm, Hunt the Fox, Lightshow, Live-Interaktionen
- **Nach dem Event:** Feedback-Push, Gutscheine, CRM/VIP-Flagging

Warum wichtig?

- ✅ **Kontext für Notifications:** User erhalten relevante Benachrichtigungen zur richtigen Zeit
- ✅ **Event-basierte Features:** Ticketing, Loyalty-Punkte, After-Party-Gutscheine
- ✅ **Analytics:** “Welche Events hatten die meisten Teilnehmer?”

6.1 Event-Grundmodell

Collection: `clubs/{clubId}/events/{eventId}`

Event-Dokument:

Feldname	Datentyp	Beschreibung	Beispielwert
`eventId`	`string`	Eindeutige Event-ID	Auto-generiert
`clubId`	`string`	Club-ID	`"club_abc123"`
`title`	`string`	Event-Titel	`"90s Night"`
`description`	`string | null`	Event-Beschreibung	`"Die besten Hits der 90er!"`
`startAt`	`number`	Start-Timestamp (Unix ms)	`1701392400000` (22:00 Uhr)
`endAt`	`number`	End-Timestamp (Unix ms)	`1701410400000` (03:00 Uhr)
`imageUrl`	`string | null`	Event-Poster-URL	`”https://place-hold.co/1200x600/e2e8f0/1e293b?text=Event_poster_for_a_90s_themed_night_party_featurin
`ticketUrl`	`string | null`	Ticketkauf-URL (extern/intern)	`"https://tickets.club.com/90s"`
`tags`	`string[]`	Event-Tags (für Filtering)	`["90s", "retro", "dance"]`
`createdAt`	`number`	Erstellungs-Timestamp	`1701388800000`
`updatedAt`	`number`	Update-Timestamp	`1701388800000`

6.2 Notification-Kategorien

Notification-Typen nach Event-Phase

Phase	Notification-Typ	Beschreibung	Timing	Push?
VOR	`event_reminder_24h`	“Morgen: 90s Night!”	24h vor Start	Ja
VOR	`event_reminder_2h`	“Gleich geht’s los!”	2h vor Start	Ja
VOR	`ticket_available`	“Tickets jetzt verfügbar”	Bei Ticket-Release	Ja
VOR	`fast_lane_offer`	“Jetzt Fast-Lane buchen”	12h vor Start	Ja (opt-in)
WÄHREND	`party_alarm`	“DJ ruft Zahl 42!”	Live während Event	**NEIN** (siehe Party-Modus)
WÄHREND	`hunt_the_fox`	“Hunt the Fox startet!”	Live während Event	**NEIN**
WÄHREND	`lightshow_sync`	“Aktiviere dein Display!”	Live während Event	**NEIN**
NACH	`feedback_request`	“Wie war dein Abend?”	1h nach End	Ja
NACH	`voucher_thankyou`	“10€ Gutschein fürs nächste Mal”	24h nach End	Ja
NACH	`vip_upgrade_offer`	“Werde VIP-Member!”	48h nach End	Ja (CRM)

Push vs. Keine Push

WICHTIG: Während des Events gibt es **KEINE** Push-Notifications für DJ-Aktionen!
- ✅ **Vor/Nach dem Event:** Push-Notifications erlaubt
- ❌ **Während des Events:** Nur In-App-Notifications oder Party-Modus-basiert

---

6.3 Integration mit Party-Modus

Wie Event-Lifecycle und Party-Modus zusammenarbeiten

Szenario: Party Alarm während Event

1. **Event ist aktiv** (`startAt ≤ now ≤ endAt`)
2. User ist **im Club eingecheckt** (`isInClub = true`)
3. User hat **Party-Modus aktiviert** (`partyModeActive = true`)
4. DJ startet **Party Alarm** (wählt Zahl)
5. System prüft:
   `typescript`
   `const participants = await findParticipants(clubId, djSelectedNumber);`
6. Alle User mit **richtigem** `selectedNumber` nehmen teil
7. **Keine Push-Notification!** User sieht Overlay nur wenn App geöffnet ist

Szenario: Ticketkauf vor Event

1. **Event ist geplant** (`startAt` ist in der Zukunft)
2. System sendet **Push-Notification** an alle User, die den Club favorisiert haben
3. User klickt auf Notification → Landet auf Event-Detail-Seite
4. User kann Ticket kaufen (extern oder intern)

6.4 Funktionen & Interfaces

TypeScript Interfaces

```typescript
// packages/shared-types/src/event.ts

export interface Event {
  /** Eindeutige Event-ID */
  eventId: string;

  /** Club-ID, zu dem das Event gehört */
  clubId: string;

  /** Event-Titel */
  title: string;

  /** Event-Beschreibung (optional) */
  description?: string | null;

  /** Start-Timestamp (Unix ms) */
  startAt: number;

  /** End-Timestamp (Unix ms) */
  endAt: number;

  /** Event-Poster-URL (optional) */
  imageUrl?: string | null;

  /** Ticketkauf-URL (optional) */
  ticketUrl?: string | null;

  /** Event-Tags für Filtering */
  tags?: string[];

  /** Erstellungs-Timestamp */
  createdAt: number;

  /** Update-Timestamp */
  updatedAt: number;
}

export type EventPhase = 'upcoming' | 'ongoing' | 'past';

export interface EventNotification {
  /** Notification-ID */
  notificationId: string;

  /** Event-ID */
  eventId: string;

  /** Notification-Typ */
  type: 'event_reminder_24h' | 'event_reminder_2h' | 'ticket_available' | 'fast_lane_o
ffer' |
        'party_alarm' | 'hunt_the_fox' | 'lightshow_sync' |
        'feedback_request' | 'voucher_thankyou' | 'vip_upgrade_offer';

  /** Notification-Text */
  message: string;

  /** Push-Notification senden? */
  sendPush: boolean;

  /** Geplanter Sende-Zeitpunkt (Unix ms) */
  scheduledAt: number;

  /** Wurde gesendet? */
```

```
  sent: boolean;

  /** Tatsächlicher Sende-Zeitpunkt (Unix ms) */
  sentAt?: number | null;
}
```

Core Functions (Pseudocode)

```typescript
// packages/core/src/utils/events.ts

/**
 * Gibt die aktuelle Phase eines Events zurück
 */
export function getEventPhase(event: Event): EventPhase {
  const now = Date.now();

  if (now < event.startAt) {
    return 'upcoming';
  } else if (now >= event.startAt && now <= event.endAt) {
    return 'ongoing';
  } else {
    return 'past';
  }
}

/**
 * Prüft, ob ein Event gerade läuft
 */
export function isEventOngoing(event: Event): boolean {
  return getEventPhase(event) === 'ongoing';
}

/**
 * Holt alle aktiven Events eines Clubs
 */
export async function getActiveEvents(clubId: string): Promise<Event[]> {
  const now = Date.now();

  const snapshot = await getDocs(
    query(
      collection(db, `clubs/${clubId}/events`),
      where('startAt', '<=', now),
      where('endAt', '>=', now)
    )
  );

  return snapshot.docs.map(doc => doc.data() as Event);
}

/**
 * Holt alle zukünftigen Events eines Clubs
 */
export async function getUpcomingEvents(clubId: string): Promise<Event[]> {
  const now = Date.now();

  const snapshot = await getDocs(
    query(
      collection(db, `clubs/${clubId}/events`),
      where('startAt', '>', now),
      orderBy('startAt', 'asc')
    )
  );

  return snapshot.docs.map(doc => doc.data() as Event);
}

/**
 * Erstellt automatisch Notifications für ein neues Event
 * (Wird von Cloud Function aufgerufen)
 */
```

```
export async function scheduleEventNotifications(event: Event): Promise<void> {
  const notifications: Partial<EventNotification>[] = [
    // 24h vor Event
    {
      eventId: event.eventId,
      type: 'event_reminder_24h',
      message: `Morgen: ${event.title}! 🎉`,
      sendPush: true,
      scheduledAt: event.startAt - 24 * 60 * 60 * 1000,
      sent: false
    },
    // 2h vor Event
    {
      eventId: event.eventId,
      type: 'event_reminder_2h',
      message: `Gleich geht's los: ${event.title}! 🚀`,
      sendPush: true,
      scheduledAt: event.startAt - 2 * 60 * 60 * 1000,
      sent: false
    },
    // 1h nach Event
    {
      eventId: event.eventId,
      type: 'feedback_request',
      message: 'Wie war dein Abend? Gib uns Feedback! 💬',
      sendPush: true,
      scheduledAt: event.endAt + 1 * 60 * 60 * 1000,
      sent: false
    }
  ];

  // Speichere Notifications
  const batch = writeBatch(db);
  for (const notif of notifications) {
    const ref = doc(collection(db, `clubs/${event.clubId}/notifications`));
    batch.set(ref, { ...notif, notificationId: ref.id, createdAt: Date.now() });
  }
  await batch.commit();
}
```

6.5 Firestore-Schema

Collection: `clubs/{clubId}/events/{eventId}`

Event-Dokument:

```
{
  "eventId": "event_abc123",
  "clubId": "club_abc123",
  "title": "90s Night",
  "description": "Die besten Hits der 90er!",
  "startAt": 1701392400000,              // Unix ms (22:00 Uhr)
  "endAt": 1701410400000,                // Unix ms (03:00 Uhr)
  "imageUrl": "https://placehold.co/1200x600/e2e8f0/1e293b?
text=Promotional_image_for_a_90s_themed_night_event_fea",
  "ticketUrl": "https://tickets.club.com/90s",
  "tags": ["90s", "retro", "dance"],
  "createdAt": 1701388800000,
  "updatedAt": 1701388800000
}
```

Collection: `clubs/{clubId}/notifications/{notificationId}`

Notification-Dokument:

```
{
  "notificationId": "notif_abc123",
  "eventId": "event_abc123",
  "type": "event_reminder_24h",
  "message": "Morgen: 90s Night! 🎉",
  "sendPush": true,
  "scheduledAt": 1701306000000,          // 24h vor Event
  "sent": false,
  "sentAt": null,
  "createdAt": 1701388800000
}
```

Firestore Queries

Query 1: Alle aktiven Events

```
const activeEvents = await getDocs(
  query(
    collection(db, `clubs/${clubId}/events`),
    where('startAt', '<=', Date.now()),
    where('endAt', '>=', Date.now())
  )
);
```

Query 2: Alle zukünftigen Events (sortiert)

```
const upcomingEvents = await getDocs(
  query(
    collection(db, `clubs/${clubId}/events`),
    where('startAt', '>', Date.now()),
    orderBy('startAt', 'asc')
  )
);
```

Benötigte Firestore Composite Indexes:

```
clubs/{clubId}/events
  - startAt (ASC) + endAt (ASC)
  - startAt (ASC) + endAt (DESC)
```

---

7. INTEGRATION MIT SPÄTEREN FEATURES

Dieses Addendum schafft das Fundament für alle zukünftigen interaktiven Features. Hier eine Übersicht, wie die neuen Konzepte genutzt werden:

7.1 Smart Lightshow

Was ist Smart Lightshow?
Erweiterte Lichtshow mit Musik-Sync, Benutzer-Interaktion und personalisierten Effekten.

Verwendete Konzepte:

Party-Modus

- User mit `partyModeActive = true` sehen intensivere Effekte
- User mit `selectedNumber` können spezielle Farben sehen

Feature-Flags

- `advancedAnalyticsEnabled`: Tracking von User-Engagement mit Lightshow
- `whiteLabelBrandingEnabled`: Custom-Farben für Club-Branding

Event-Lifecycle

- Lightshow-Notifications nur **während** Events (`isEventOngoing = true`)
- Keine Push-Notifications für Lightshow-Start

Beispiel-Flow:
1. DJ startet Lightshow während Event
2. System prüft: Event ist aktiv + Club hat Premium-Plan
3. Alle User mit `isInClub = true` sehen Overlay
4. User mit `partyModeActive = true` sehen zusätzlich Pulse-Effekte

7.2 Party Alarm

Was ist Party Alarm?
DJ kann einen “Alarm” auslösen, der alle aktiven Party-Mode-User gleichzeitig benachrichtigt (ohne Push).

Verwendete Konzepte:

Party-Modus (KERN-FEATURE!)

- Nur User mit `partyModeActive = true` nehmen teil

- DJ wählt eine der 6 Zahlen → Match mit `selectedNumber`

Feature-Flags

- `partyAlarmEnabled`: Nur für Premium-Clubs verfügbar

Event-Lifecycle

- Party Alarm nur während Events möglich
- Analytics: “Wie viele User haben reagiert?”

Beispiel-Flow:
1. DJ klickt “Party Alarm” in DJ-Console
2. DJ wählt Zahl (z.B. `42`)
3. System findet alle User mit `partyModeActive = true` && `selectedNumber = 42`
4. Diese User sehen sofortiges Overlay (nur wenn App geöffnet)
5. Nach 30 Sekunden: Overlay verschwindet

7.3 Hunt the Fox

Was ist Hunt the Fox?
Interaktives Spiel: DJ versteckt virtuellen “Fuchs” in einem der Bereiche des Clubs. User müssen ihn finden.

Verwendete Konzepte:

Party-Modus

- Nur User mit `partyModeActive = true` können am Spiel teilnehmen
- User müssen physisch im Club sein (`isInClub = true`)

Feature-Flags

- `huntTheFoxEnabled`: Nur für Premium-Clubs

Event-Lifecycle

- Hunt the Fox nur während Events spielbar
- Nach Event: Gewinner-Liste in Post-Event-Notification

Beispiel-Flow:
1. DJ startet Hunt the Fox während Event
2. System zeigt allen Party-Mode-Usern einen Hinweis
3. User bewegen sich physisch im Club zu verschiedenen Bereichen
4. App nutzt GPS/iBeacons, um Position zu tracken
5. Wer den “Fuchs” findet, gewinnt Preis

7.4 Active Sync-Gewinnspiele

Was ist Active Sync?
Gewinnspiele, die durch DJ-Aktionen live ausgelöst werden (z.B. “Wer zuerst auf den Button drückt”).

Verwendete Konzepte:

Party-Modus (ABSOLUT ZENTRAL!)

- Nur User mit `partyModeActive = true` können teilnehmen
- DJ wählt Zahl → Nur User mit passendem `selectedNumber` sind im Pool

Dating-Status

- Spezielle Gewinnspiele nur für `relationshipStatus = 'single'`
- Z.B. “Singles-Speed-Dating-Gewinnspiel”

Trust-System

- Nur User mit `phoneVerified = true` können an Gewinnauslosungen teilnehmen
- Verhindert Fake-Accounts

Beispiel-Flow:
1. DJ kündigt Gewinnspiel an: “Zahl 23 gewinnt!”
2. User mit `selectedNumber = 23` sehen Button in App
3. Wer zuerst klickt, gewinnt
4. System prüft: `phoneVerified = true` → Gewinn gültig

7.5 Ticketing & Loyalty

Was ist Ticketing & Loyalty?
User können Tickets in der App kaufen, sammeln Loyalty-Punkte, erhalten VIP-Status.

Verwendete Konzepte:

Trust-System

- `phoneVerified = true`: Voraussetzung für Ticketkauf
- `trustLevel = 'trusted'`: Schneller Check-In (Fast-Lane)

Event-Lifecycle

- Tickets sind Event-spezifisch (`eventId`)
- Vor Event: Ticketkauf-Reminder
- Nach Event: Loyalty-Punkte gutschreiben

Club-Pläne

- `guestLimitUnlimited`: Premium-Clubs können unlimitiert Tickets verkaufen
- Free-Clubs: Max. 100 Tickets pro Event

Beispiel-Flow:
1. User sieht zukünftiges Event in App
2. User kauft Ticket (nur wenn `phoneVerified = true`)
3. Am Event-Tag: Push-Notification “Dein Ticket für heute”
4. Check-In: Türsteher scannt QR-Code → `isInClub = true`
5. Nach Event: +50 Loyalty-Punkte

---

8. SECURITY RULES-ERWEITERUNGEN

Die folgenden Security Rules müssen zur bestehenden `firestore.rules` hinzugefügt werden:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // ===== EXISTING HELPER FUNCTIONS (from ARCHITECTURE.md) =====
    // ... (alle bestehenden Functions bleiben unverändert)

    // ===== NEW HELPER FUNCTIONS =====

    /**
     * Prüft, ob ein Club ein bestimmtes Feature aktiviert hat
     */
    function hasFeature(clubId, featureKey) {
      let club = get(/databases/$(database)/documents/platform/clubs/$(clubId)).data;
      return club.featureFlags[featureKey] == true;
    }

    /**
     * Prüft, ob ein Event gerade läuft
     */
    function isEventOngoing(eventId, clubId) {
      let event = get(/databases/$(database)/documents/clubs/$(clubId)/events/$(event-
Id)).data;
      return event.startAt <= request.time.toMillis() && event.endAt >= re-
quest.time.toMillis();
    }

    // ===== UPDATED RULES: clubs/{clubId}/users/{uid} =====

    match /clubs/{clubId}/users/{uid} {
      // Lesen: (unverändert)
      allow read: if isSuperAdmin() ||
        hasRole(clubId, 'admin') ||
        hasAnyRole(clubId, ['staff', 'door', 'waiter', 'bar', 'cloakroom', 'dj']) ||
        (request.auth.uid == uid) ||
        (request.auth.uid in resource.data.friendIds);

      // Erstellen: (unverändert)
      allow create: if request.auth.uid == uid;

      // Updaten: ERWEITERT mit neuen Regeln
      allow update: if hasRole(clubId, 'admin') ||

        // Door-Staff: Trust-Felder + Check-In
        (hasRole(clubId, 'door') &&
         request.resource.data.diff(resource.data).affectedKeys()
           .hasOnly(['trustedLevel', 'verifiedBy', 'verifiedAt', 'checkedIn', 'checked
InAt',
                     'isInClub', 'currentClubId', 'phoneVerified', 'blacklisted', 'bla
cklistReason'])) ||

        // User selbst: Basis-Felder + Party-Modus + Dating-Status (mit 6h-Check)
        (request.auth.uid == uid && (
          // Fall 1: Party-Modus-Felder (ohne Validierung, da harmlos)
          request.resource.data.diff(resource.data).affectedKeys()
            .hasOnly(['partyModeActive', 'partyModeLastActivatedAt',
'selectedNumber',
                      'isInClub', 'currentClubId', 'checkedInAt']) ||

          // Fall 2: Basis-Felder (unverändert)
          request.resource.data.diff(resource.data).affectedKeys()
            .hasOnly(['displayName', 'photoURL', 'checkedIn', 'language',
```

```
'lastSeen']) ||

          // Fall 3: Dating-Status (mit 6h-Regel!)
          (request.resource.data.diff(resource.data).affectedKeys()
             .hasOnly(['relationshipStatus', 'relationshipStatusLastChangedAt']) &&
           (
             // Erste Festlegung
             resource.data.relationshipStatusLastChangedAt == null ||
             // 6 Stunden vergangen
             (request.time.toMillis() -
resource.data.relationshipStatusLastChangedAt) >= 21600000
           ))
        ));
    }

    // ===== NEW RULES: clubs/{clubId}/events/{eventId} =====

    match /clubs/{clubId}/events/{eventId} {
      // Alle Club-Mitglieder können Events lesen
      allow read: if isClubMember(clubId);

      // Nur Admins können Events erstellen/updaten/löschen
      allow create, update, delete: if hasRole(clubId, 'admin');
    }

    // ===== NEW RULES: clubs/{clubId}/notifications/{notificationId} =====

    match /clubs/{clubId}/notifications/{notificationId} {
      // Nur Admins/System können Notifications lesen/schreiben
      allow read, write: if hasRole(clubId, 'admin');
    }

    // ===== UPDATED RULES: platform/clubs/{clubId} =====

    match /platform/clubs/{clubId} {
      // Lesen: (unverändert)
      allow read: if isSuperAdmin() || isClubOwner(clubId);

      // Erstellen/Löschen: (unverändert)
      allow create, delete: if isSuperAdmin();

      // Updaten: ERWEITERT (plan + featureFlags nur für Super-Admins)
      allow update: if isSuperAdmin() ||
        (isClubOwner(clubId) &&
         !request.resource.data.diff(resource.data).affectedKeys()
           .hasAny(['subscriptionTier', 'subscriptionStatus', 'plan', 'featureFlags'])
);
    }

    // ===== UPDATED RULES: clubs/{clubId}/state/global =====

    match /clubs/{clubId}/state/global {
      // Lesen: (unverändert)
      allow read: if isClubMember(clubId);

      // Schreiben: ERWEITERT (dailyInteractionNumbers dürfen nur von Admins/DJ
geändert werden)
      allow write: if hasAnyRole(clubId, ['admin', 'dj']);
    }
  }
}
```

9. ZUSAMMENFASSUNG & NÄCHSTE SCHRITTE

Was wurde dokumentiert?

Dieses Architecture Addendum erweitert die bestehende Nightlife OS-Architektur um **fünf zentrale Konzepte**:

✅ 1. Interaktiver On-Site-Status: Party-Modus & Chill-Modus

- **6 tägliche Interaktions-Zahlen** pro Club
- **Party-Modus** als bewusste Opt-In-Mechanik (30-Minuten-Timer)
- **Keine Push-Notifications** für DJ-Aktionen (User muss Ansage hören)
- Firestore-Felder: `isInClub`, `partyModeActive`, `selectedNumber`, `dailyInteractionNumbers`

✅ 2. Dating-Status & 6-Stunden-Regel

- **4 Relationship-Status-Typen** (single, taken, complicated, dont_touch)
- **6-Stunden-Sperre** für Status-Änderungen (serverseitig validiert via Security Rules)
- Firestore-Felder: `relationshipStatus`, `relationshipStatusLastChangedAt`

✅ 3. Trusted User & Zechpreller-Schutz

- **2-Stufen-System:** Phone-Verifizierung + Türsteher-Verifizierung
- **Globaler Trust-Level:** Einmal verifiziert → Gilt in allen Clubs
- Firestore-Felder: `phoneVerified`, `trustLevel`, `trustedVerifiedAt`, `globalTrustLevel`

✅ 4. Club-Pläne & Feature-Flags

- **2 Pläne:** Free (mit Ads) vs. Premium (werbefrei, mehr Features)
- **8 Feature-Flags:** adsEnabled, partyAlarmEnabled, huntTheFoxEnabled, etc.
- Firestore-Felder: `plan`, `featureFlags`

✅ 5. Event-Lifecycle: Vor / Während / Nach dem Event

- **Event-Modell** mit `startAt`, `endAt`, `title`, `description`
- **Notification-Kategorien** je nach Phase (Vor: Reminder, Während: Keine Push, Nach: Feedback)
- Firestore-Collections: `clubs/{clubId}/events`, `clubs/{clubId}/notifications`

Firestore-Schema-Übersicht (Neue Felder)

`platform/users/{uid}` (erweitert)

```
{
  // ... bestehende Felder
  "phoneNumber": "+491234567890",
  "phoneVerified": true,
  "globalTrustLevel": "trusted",
  "trustedVerifiedAt": 1701388800000,
  "trustedVerifiedByClubId": "club_abc123",
  "trustedVerifiedByStaffId": "staff_door1"
}
```

`platform/clubs/{clubId}` (erweitert)

```
{
  // ... bestehende Felder
  "plan": "premium",
  "featureFlags": {
    "adsEnabled": false,
    "partyAlarmEnabled": true,
    "huntTheFoxEnabled": true,
    "advancedAnalyticsEnabled": true,
    "whiteLabelBrandingEnabled": true
  }
}
```

`clubs/{clubId}/users/{uid}` (erweitert)

```
{
  // ... bestehende Felder

  // Party-Modus
  "isInClub": true,
  "currentClubId": "club_abc123",
  "checkedInAt": 1701388800000,
  "partyModeActive": true,
  "partyModeLastActivatedAt": 1701388800000,
  "selectedNumber": 42,

  // Dating-Status
  "relationshipStatus": "single",
  "relationshipStatusLastChangedAt": 1701388800000,

  // Trust-System
  "phoneVerified": true,
  "trustLevel": "trusted",
  "trustedVerifiedAt": 1701388800000,
  "trustedVerifiedByClubId": "club_abc123"
}
```

`clubs/{clubId}/state/global` (erweitert)

```
{
  // ... bestehende Felder (mode, lightColor, etc.)
  "dailyInteractionNumbers": [7, 23, 42, 55, 68, 91],
  "dailyNumbersGeneratedAt": 1701388800000
}
```

`clubs/{clubId}/events/{eventId}` (neu)

```
{
  "eventId": "event_abc123",
  "clubId": "club_abc123",
  "title": "90s Night",
  "description": "Die besten Hits der 90er!",
  "startAt": 1701392400000,
  "endAt": 1701410400000,
  "imageUrl": "https://placehold.co/1200x600/e2e8f0/1e293b?
text=Promotional_image_for_a_90s_themed_dance_party_or_",
  "ticketUrl": "https://tickets.club.com/90s",
  "tags": ["90s", "retro", "dance"],
  "createdAt": 1701388800000,
  "updatedAt": 1701388800000
}
```

Benötigte TypeScript-Interfaces (Zusammenfassung)

```typescript
// packages/shared-types/src/user.ts

export interface UserDocument {
  // ... bestehende Felder

  // Party-Modus
  isInClub: boolean;
  currentClubId: string | null;
  checkedInAt: number | null;
  partyModeActive: boolean;
  partyModeLastActivatedAt: number | null;
  selectedNumber: number | null;

  // Dating-Status
  relationshipStatus: 'single' | 'taken' | 'complicated' | 'dont_touch' | null;
  relationshipStatusLastChangedAt: number | null;

  // Trust-System
  phoneVerified: boolean;
  trustLevel: 'normal' | 'trusted';
  trustedVerifiedAt: number | null;
  trustedVerifiedByClubId: string | null;
}

// packages/shared-types/src/club.ts

export interface Club {
  // ... bestehende Felder

  plan: 'free' | 'premium';
  featureFlags?: {
    adsEnabled?: boolean;
    partyAlarmEnabled?: boolean;
    huntTheFoxEnabled?: boolean;
    advancedAnalyticsEnabled?: boolean;
    whiteLabelBrandingEnabled?: boolean;
  };
}

export interface ClubStateDocument {
  // ... bestehende Felder

  dailyInteractionNumbers: number[];
  dailyNumbersGeneratedAt: number | null;
}

// packages/shared-types/src/event.ts

export interface Event {
  eventId: string;
  clubId: string;
  title: string;
  description?: string | null;
  startAt: number;
  endAt: number;
  imageUrl?: string | null;
  ticketUrl?: string | null;
  tags?: string[];
  createdAt: number;
  updatedAt: number;
}
```

Nächste Schritte (Empfehlung)

Phase 9: Foundations (dieses Addendum implementieren)

1. **Datenmodell erweitern**
   - [ ] Neue Felder in `packages/shared-types` hinzufügen
   - [ ] Firestore Security Rules erweitern
   - [ ] Composite Indexes erstellen

2. **Core-Funktionen implementieren**
   - [ ] `packages/core/src/utils/party-mode.ts`
   - [ ] `packages/core/src/utils/relationship-status.ts`
   - [ ] `packages/core/src/utils/trust-system.ts`
   - [ ] `packages/core/src/utils/feature-flags.ts`
   - [ ] `packages/core/src/utils/events.ts`

3. **UI-Komponenten (Minimal)**
   - [ ] Party-Modus-Button in `club-app`
   - [ ] Dating-Status-Picker in `club-app`
   - [ ] Trust-Badge-Display in `staff-door`
   - [ ] Feature-Flag-Checks in allen Apps

4. **Cloud Functions (Optional für Phase 9)**
   - [ ] `generateDailyNumbers` (Cron-Job)
   - [ ] `syncTrustLevel` (Trigger)
   - [ ] `scheduleEventNotifications` (Trigger)

Phase 10: Party Alarm & Hunt the Fox

(Baut auf Phase 9 auf)

Phase 11: Ticketing & Loyalty

(Baut auf Phase 9 + Phase 10 auf)

Offene Fragen / Decisions Needed

1. **Party-Modus-Timer:** Client-seitig oder Server-seitig?
   - **Empfehlung:** Client-seitig für Phase 9, serverseitig für Production

2. **Phone-Verifizierung:** Welcher Service?
   - **Empfehlung:** Firebase Phone Authentication (bereits integriert)

3. **Event-Notifications:** Push-Service?
   - **Empfehlung:** Firebase Cloud Messaging (FCM)

4. **Feature-Flags:** Statisch oder dynamisch?
   - **Empfehlung:** Dynamisch (Admin kann in Club-Settings ändern)

5. **Dating-Status:** Opt-In oder Opt-Out?
   - **Empfehlung:** Opt-In (User muss explizit Status setzen)

Kontakt & Feedback

Dieses Addendum ist ein **Living Document** und wird während der Implementierung weiter verfeinert. Bei Fragen oder Änderungswünschen bitte in der bestehenden Dokumentation (`ARCHITECTURE.md`) nachschlagen oder das Entwicklungsteam kontaktieren.

Version: 1.0
Stand: 3. Dezember 2025
Nächstes Review: Nach Phase 9-Implementierung