TEF Implementation Project Plan

Task Exchange Format Complete Implementation Guide for Cursor AI

Version 1.0 | December 2025 TaskJuggler • Process.AI • Projects.AI

Executive Summary

This project plan provides a comprehensive, phased approach for implementing the Task Exchange Format (TEF) specification across the TaskJuggler platform ecosystem. The plan transforms TaskJuggler from a task management application into a universal TEF Exchange—the "Stripe for Tasks"—enabling seamless task interchange between humans, AI agents, teams, and IoT devices.

Platform Responsibilities

Platform	TEF Role	Responsibilities
TaskJuggler	Exchange Operator + Introduction Broker	Routes all TEF messages, manages actor registry, handles protocol conversion, stores conversations
Process.AI	Task Orchestrator + Monitor	Automates workflows, tracks compliance, detects patterns, manages delegation rules
Projects.AI	Task Aggregator + Analytics	Groups tasks into projects, manages resources, provides portfolio visibility

Timeline Overview

Phase	Timeline	Focus
Phase 1	Months 1-3	Foundation: TEF message format, actor registry, human-to-human exchange
Phase 2	Months 4-6	IoT Integration: MQTT broker, device registration, claiming flow
Phase 3	Months 7-9	AI Integration: MCP server, AI agent registration, delegation engine
Phase 4	Months 10-12	Advanced Features: CoAP/Matter, trust scoring, commercial launch

---

Phase 1: Foundation (Months 1-3)

Phase 1 establishes the core TEF infrastructure within TaskJuggler, implementing the message format, actor registry, and human-to-human task exchange capabilities.

1.1 Database Schema Updates

1.1.1 Actors Table

File: `database/migrations/001_create_actors_table.sql`

```sql
-- Create actor type enum
CREATE TYPE actor_type AS ENUM ('HUMAN', 'AI_AGENT', 'TEAM', 'IOT_DEVICE', 'IOT_GATEWAY');

-- Create actor status enum
CREATE TYPE actor_status AS ENUM ('PENDING_CLAIM', 'ACTIVE', 'SUSPENDED', 'DELETED');

-- Create actors table
CREATE TABLE actors (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    actor_type actor_type NOT NULL,
    display_name VARCHAR(255) NOT NULL,
    capabilities JSONB DEFAULT '[]'::jsonb,
    contact_methods JSONB DEFAULT '[]'::jsonb,
    metadata JSONB DEFAULT '{}'::jsonb,
    authentication JSONB DEFAULT '{}'::jsonb,
    status actor_status DEFAULT 'ACTIVE',
    organization_id UUID REFERENCES organizations(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_actors_type ON actors(actor_type);
CREATE INDEX idx_actors_status ON actors(status);
CREATE INDEX idx_actors_organization ON actors(organization_id);
CREATE INDEX idx_actors_capabilities ON actors USING GIN(capabilities);

-- RLS Policies
ALTER TABLE actors ENABLE ROW LEVEL SECURITY;
```

1.1.2 Relationships Table

1.1.2 Relationships Table

File: `database/migrations/002_create_relationships_table.sql`

```sql
-- Create relationship type enum
CREATE TYPE relationship_type AS ENUM ('OWNER', 'PEER', 'DELEGATE', 'WATCHER', 'VENDOR');

-- Create established via enum
CREATE TYPE established_via AS ENUM ('CLAIM_CODE', 'INVITATION', 'ORGANIZATION', 'API');

-- Create relationships table
CREATE TABLE relationships (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    actor_a_id UUID NOT NULL REFERENCES actors(id) ON DELETE CASCADE,
    actor_b_id UUID NOT NULL REFERENCES actors(id) ON DELETE CASCADE,
    relationship_type relationship_type NOT NULL,
    permissions JSONB DEFAULT '{}'::jsonb,
    established_via established_via NOT NULL,
    trust_score DECIMAL(5,2) DEFAULT 50.00,
    task_count INTEGER DEFAULT 0,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    expires_at TIMESTAMPTZ,

    CONSTRAINT unique_relationship UNIQUE(actor_a_id, actor_b_id)
);

-- Indexes
CREATE INDEX idx_relationships_actor_a ON relationships(actor_a_id);
CREATE INDEX idx_relationships_actor_b ON relationships(actor_b_id);
CREATE INDEX idx_relationships_type ON relationships(relationship_type);
```

1.1.3 Conversations & Messages Tables

File: `database/migrations/003_create_conversations_table.sql`

```sql
```

```sql
-- Create message type enum
CREATE TYPE tef_message_type AS ENUM (
    'TASK_CREATE', 'TASK_ACCEPT', 'TASK_REJECT', 'TASK_DELEGATE',
    'TASK_STATUS_UPDATE', 'TASK_COMPLETE', 'TASK_CANCEL', 'TASK_REOPEN',
    'TASK_MESSAGE', 'TASK_CLARIFICATION_REQUEST', 'TASK_CLARIFICATION_RESPONSE',
    'TASK_ATTACHMENT_ADD', 'TASK_PROGRESS_REPORT', 'TASK_TIMELINE_UPDATE',
    'TASK_DISPUTE', 'TASK_RESOLUTION'
);

-- Create conversations table
CREATE TABLE conversations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    task_id UUID NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
    participants UUID[] NOT NULL,
    message_count INTEGER DEFAULT 0,
    last_message_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Create messages table
CREATE TABLE messages (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    conversation_id UUID NOT NULL REFERENCES conversations(id) ON DELETE CASCADE,
    task_id UUID NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
    message_type tef_message_type NOT NULL,
    source_actor_id UUID NOT NULL REFERENCES actors(id),
    target_actor_id UUID NOT NULL REFERENCES actors(id),
    reply_to_id UUID REFERENCES messages(id),
    payload JSONB NOT NULL,
    delivered_at TIMESTAMPTZ,
    read_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_conversations_task ON conversations(task_id);
CREATE INDEX idx_messages_conversation ON messages(conversation_id);
CREATE INDEX idx_messages_task ON messages(task_id);
CREATE INDEX idx_messages_source ON messages(source_actor_id);
CREATE INDEX idx_messages_target ON messages(target_actor_id);
CREATE INDEX idx_messages_type ON messages(message_type);
```

1.1.4 Relationship History Table

File: database/migrations/004_create_relationship_history_table.sql

```sql
-- Create event type enum
CREATE TYPE history_event_type AS ENUM (
    'TASK_SENT', 'TASK_ACCEPTED', 'TASK_REJECTED',
    'TASK_COMPLETED', 'TASK_CANCELLED', 'TASK_DISPUTED'
);

-- Create outcome enum
CREATE TYPE task_outcome AS ENUM ('SUCCESS', 'FAILURE', 'CANCELLED', 'DISPUTED');

-- Create relationship history table
CREATE TABLE relationship_history (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    relationship_id UUID NOT NULL REFERENCES relationships(id) ON DELETE CASCADE,
    actor_a_id UUID NOT NULL REFERENCES actors(id),
    actor_b_id UUID NOT NULL REFERENCES actors(id),
    task_id UUID REFERENCES tasks(id),
    event_type history_event_type NOT NULL,
    outcome task_outcome,
    response_time_ms INTEGER,
    completion_time_ms INTEGER,
    metadata JSONB DEFAULT '{}'::jsonb,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_history_relationship ON relationship_history(relationship_id);
CREATE INDEX idx_history_task ON relationship_history(task_id);
CREATE INDEX idx_history_created ON relationship_history(created_at);
```

1.1.5 Delegation Rules Table

File: database/migrations/005_create_delegation_rules_table.sql

```sql
```

```
CREATE TABLE delegation_rules (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    delegator_id UUID NOT NULL REFERENCES actors(id) ON DELETE CASCADE,
    delegate_id UUID NOT NULL REFERENCES actors(id) ON DELETE CASCADE,
    scope JSONB NOT NULL DEFAULT '{}'::jsonb,
    -- scope: { task_types: [], target_actors: [], max_priority: "HIGH" }
    constraints JSONB DEFAULT '{}'::jsonb,
    is_active BOOLEAN DEFAULT true,
    expires_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_delegation_delegator ON delegation_rules(delegator_id);
CREATE INDEX idx_delegation_delegate ON delegation_rules(delegate_id);
CREATE INDEX idx_delegation_active ON delegation_rules(is_active) WHERE is_active = true;
```

1.2 TEF Message Format Implementation

1.2.1 TEF Types & Interfaces

File: src/types/tef.ts

typescript

```typescript
// TEF Envelope - wraps all messages
export interface TEFEnvelope {
  tef_version: string; // "2.0.0"
  message_id: string; // UUID
  message_type: TEFMessageType;
  timestamp: string; // ISO8601
  correlation_id: string; // UUID - links conversation
  task_id: string; // UUID
  source_actor: ActorRef;
  target_actor: ActorRef;
  reply_to_message_id?: string;
  transport_hints?: TransportHints;
}

// Actor Reference
export interface ActorRef {
  actor_id: string;
  actor_type: ActorType;
  display_name: string;
  capabilities?: string[];
  contact_methods?: ContactMethod[];
  organization_id?: string;
  acting_on_behalf_of?: ActorRef; // For delegated authority
  metadata?: Record<string, any>;
}

export type ActorType = 'HUMAN' | 'AI_AGENT' | 'TEAM' | 'IOT_DEVICE' | 'IOT_GATEWAY';

export interface ContactMethod {
  protocol: 'http' | 'websocket' | 'mqtt' | 'coap' | 'mcp' | 'email' | 'sms';
  endpoint: string;
  priority?: number;

  metadata?: Record<string, any>;
}

// Message Types
export type TEFMessageType =
  | 'TASK_CREATE' | 'TASK_ACCEPT' | 'TASK_REJECT' | 'TASK_DELEGATE'
  | 'TASK_STATUS_UPDATE' | 'TASK_COMPLETE' | 'TASK_CANCEL' | 'TASK_REOPEN'
  | 'TASK_MESSAGE' | 'TASK_CLARIFICATION_REQUEST' | 'TASK_CLARIFICATION_RESPONSE'
  | 'TASK_ATTACHMENT_ADD' | 'TASK_PROGRESS_REPORT' | 'TASK_TIMELINE_UPDATE'
  | 'TASK_DISPUTE' | 'TASK_RESOLUTION';

// TEF Task Object
export interface TEFTask {
  task_id: string;
  task_type: TaskType;
  title: string;
  description?: string;
  structured_instructions?: StructuredInstructions;
  priority: Priority;
  status: TaskStatus;
  requestor: ActorRef;
  owner?: ActorRef;
  watchers?: ActorRef[];
  timeline: Timeline;
  context?: TaskContext;
  provenance: Provenance;
  conversation_id: string;
  extensions?: Record<string, any>;
}

export type TaskType =
  | 'ACTION' | 'MEETING' | 'APPROVAL' | 'PAYMENT'
  | 'INFORMATION' | 'MONITORING' | 'ACTUATION';

export type Priority = 'CRITICAL' | 'HIGH' | 'NORMAL' | 'LOW' | 'BACKGROUND';

export type TaskStatus =
  | 'DRAFT' | 'PENDING' | 'ACCEPTED' | 'IN_PROGRESS'
  | 'BLOCKED' | 'COMPLETED' | 'CANCELLED';

// Timeline (owner-controlled model)
export interface Timeline {
  requested_by?: string; // Requestor's desired deadline
  hard_deadline?: string; // Absolute latest
  owner_start_date?: string; // Owner-controlled
  owner_expected_completion?: string; // Owner-controlled
  estimated_duration?: string; // ISO8601 duration
  recurrence?: string; // RFC5545 RRULE
  timezone: string; // IANA timezone
}

// Provenance tracking
export interface Provenance {
  original_source: ActorRef;
  transformation_chain: TransformationRecord[];
  current_handler: ActorRef;
  delegation_chain?: DelegationRecord[];
}

export interface TransformationRecord {
  actor: ActorRef;
  action: string;
  timestamp: string;
  details?: Record<string, any>;
```

```
}

export interface DelegationRecord {
  from_actor: ActorRef;
  to_actor: ActorRef;
  delegated_at: string;
  scope: string;
}

// Structured Instructions (for AI/IoT)
export interface StructuredInstructions {
  steps?: InstructionStep[];
  preconditions?: Condition[];
  postconditions?: Condition[];
  success_criteria?: Criterion[];
  constraints?: Constraint[];
  fallback_actions?: Action[];
}

export interface InstructionStep {
  order: number;
  action: string;
  parameters?: Record<string, any>;
  expected_outcome?: string;
}
```

1.2.2 TEF Message Factory

File: `src/services/tef/TEFMessageFactory.ts`

```typescript
import { v4 as uuidv4 } from 'uuid';
import { TEFEnvelope, TEFTask, ActorRef, TEFMessageType } from '@/types/tef';

export class TEFMessageFactory {
  private static TEF_VERSION = '2.0.0';

  static createEnvelope(
    messageType: TEFMessageType,
    sourceActor: ActorRef,
    targetActor: ActorRef,
    taskId: string,
    correlationId?: string,
    replyToMessageId?: string
): TEFEnvelope {
  return {
    tef_version: this.TEF_VERSION,
    message_id: uuidv4(),
    message_type: messageType,
    timestamp: new Date().toISOString(),
    correlation_id: correlationId || uuidv4(),
    task_id: taskId,
    source_actor: sourceActor,
    target_actor: targetActor,
    reply_to_message_id: replyToMessageId
  };
}

static createTaskCreate(
  task: TEFTask,
  sourceActor: ActorRef,
  targetActor: ActorRef
): TEFEnvelope & { task: TEFTask } {
  const envelope = this.createEnvelope(
    'TASK_CREATE',
    sourceActor,
    targetActor,
    task.task_id
  );
  return { ...envelope, task };
}

static createTaskAccept(
  taskId: string,
  correlationId: string,
  sourceActor: ActorRef,
  targetActor: ActorRef,
  timeline?: Partial<Timeline>
): TEFEnvelope & { timeline?: Partial<Timeline> } {
  const envelope = this.createEnvelope(
    'TASK_ACCEPT',
    sourceActor,
    targetActor,
    taskId,
    correlationId
  );
  return { ...envelope, timeline };
}
```

```
static createTaskComplete(
  taskId: string,
  correlationId: string,
  sourceActor: ActorRef,
  targetActor: ActorRef,
  results: Record<string, any>
): TEFEnvelope & { results: Record<string, any> } {
  const envelope = this.createEnvelope(
    'TASK_COMPLETE',
    sourceActor,
    targetActor,
    taskId,
    correlationId
  );
  return { ...envelope, results };
}

static createClarificationRequest(
  taskId: string,
  correlationId: string,
  sourceActor: ActorRef,
  targetActor: ActorRef,
  question: string,
  context?: Record<string, any>
): TEFEnvelope & { question: string; context?: Record<string, any> } {
  const envelope = this.createEnvelope(
    'TASK_CLARIFICATION_REQUEST',
    sourceActor,
    targetActor,
    taskId,
    correlationId
  );
  return { ...envelope, question, context };
}

static createClarificationResponse(
  taskId: string,
  correlationId: string,
  sourceActor: ActorRef,
  targetActor: ActorRef,
  replyToMessageId: string,
  response: string,
  additionalInstructions?: Record<string, any>
): TEFEnvelope & { response: string; additional_instructions?: Record<string, any> } {
  const envelope = this.createEnvelope(
      'TASK_CLARIFICATION_RESPONSE',
      sourceActor,
      targetActor,
      taskId,
      correlationId,
      replyToMessageId
    );
    return { ...envelope, response, additional_instructions: additionalInstructions };
  }
}
```

1.2.3 TEF Validator

File: `src/services/tef/TEFValidator.ts`

```typescript
import Ajv from 'ajv';
import { TEFEnvelope, TEFMessageType, TaskStatus } from '@/types/tef';

export class TEFValidator {
  private ajv: Ajv;
  private schemas: Map<string, object>;

  // Valid state transitions

  private static STATE_MACHINE: Record<TaskStatus, TaskStatus[]> = {
    'DRAFT': ['PENDING', 'CANCELLED'],
    'PENDING': ['ACCEPTED', 'CANCELLED'],
    'ACCEPTED': ['IN_PROGRESS', 'CANCELLED'],
    'IN_PROGRESS': ['COMPLETED', 'BLOCKED', 'CANCELLED'],
    'BLOCKED': ['IN_PROGRESS', 'CANCELLED'],
    'COMPLETED': [], // Terminal state (can REOPEN)
    'CANCELLED': [] // Terminal state
  };

  constructor() {
    this.ajv = new Ajv({ allErrors: true });
    this.schemas = new Map();
    this.loadSchemas();
  }

  private loadSchemas(): void {
    // Load JSON schemas for each message type
    // Implementation: load from /schemas/tef/*.json
  }

  async validateMessage(message: TEFEnvelope): Promise<ValidationResult> {
    const errors: string[] = [];

    // 1. Schema validation
    const schemaValid = this.validateSchema(message);
    if (!schemaValid.valid) {
      errors.push(...schemaValid.errors);
    }

    // 2. Actor validation
    const actorValid = await this.validateActors(message);
    if (!actorValid.valid) {
      errors.push(...actorValid.errors);
    }

    // 3. Relationship validation
    const relationshipValid = await this.validateRelationship(message);
    if (!relationshipValid.valid) {
      errors.push(...relationshipValid.errors);
    }

    // 4. State transition validation (for status updates)
    if (message.message_type === 'TASK_STATUS_UPDATE') {
      const transitionValid = await this.validateStateTransition(message);
      if (!transitionValid.valid) {
        errors.push(...transitionValid.errors);
      }
    }

    return {
      valid: errors.length === 0,
      errors
    };
  }

  isValidTransition(from: TaskStatus, to: TaskStatus): boolean {
    return TEFValidator.STATE_MACHINE[from]?.includes(to) ?? false;
  }
}

interface ValidationResult {
  valid: boolean;
  errors: string[];
```

```
}
```

1.3 Actor Registry Service

1.3.1 Actor Service

File: `src/services/actors/ActorService.ts`

```
typescript

import { v4 as uuidv4 } from 'uuid';
import { db } from '@/db';
import { ActorRef, ActorType } from '@/types/tef';

export class ActorService {

  async registerActor(data: RegisterActorInput): Promise<RegisterActorResult> {
    const actorId = uuidv4();
    let claimCode: string | undefined;
    let status: ActorStatus = 'ACTIVE';

    // Generate claim code for actors that need claiming
    if (data.actor_type === 'IOT_DEVICE' || data.actor_type === 'AI_AGENT') {
      claimCode = this.generateClaimCode();
      status = 'PENDING_CLAIM';

    }

    const actor = await db.actors.create({
      id: actorId,
      actor_type: data.actor_type,
      display_name: data.display_name,
      capabilities: data.capabilities || [],
      contact_methods: data.contact_methods || [],
      metadata: data.metadata || {},
      authentication: data.authentication || {},
      status,
      organization_id: data.organization_id
    });

    // Store claim code if generated
    if (claimCode) {
      await db.claimCodes.create({
        actor_id: actorId,
      code: claimCode,
      expires_at: new Date(Date.now() + 60 * 60 * 1000) // 1 hour
    });
  }

  return {
    actor_id: actorId,
    status,
    claim_code: claimCode,
    claim_code_expires: claimCode ? new Date(Date.now() + 60 * 60 * 1000).toISOString() : undefined,
    exchange_contact: {
      http_endpoint: `https://api.taskjuggler.io/tef/v1/actors/${actorId}`,
      websocket_endpoint: `wss://ws.taskjuggler.io/tef/${actorId}`
    }
  };
}

async getActor(actorId: string): Promise<ActorWithRelationships | null> {
  const actor = await db.actors.findById(actorId);
  if (!actor) return null;

  const relationships = await db.relationships.findByActor(actorId);
  return { ...actor, relationships };
}

async updateActor(actorId: string, updates: UpdateActorInput): Promise<Actor> {
  return db.actors.update(actorId, {
    ...updates,
    updated_at: new Date()
  });
}

async deactivateActor(actorId: string): Promise<void> {
  await db.actors.update(actorId, { status: 'DELETED' });
}

async getActorCapabilities(actorId: string): Promise<string[]> {
  const actor = await db.actors.findById(actorId);
  return actor?.capabilities || [];
}

async validateActorAuthentication(
  actorId: string,
  credentials: AuthCredentials
): Promise<boolean> {
  const actor = await db.actors.findById(actorId);
```

```
    const actor = await db.actors.findById(actorId);
    if (!actor) return false;

    // Validate based on actor type and auth method
    switch (actor.actor_type) {
      case 'HUMAN':
        return this.validateHumanAuth(actor, credentials);
      case 'AI_AGENT':
        return this.validateAIAgentAuth(actor, credentials);
      case 'IOT_DEVICE':
        return this.validateDeviceAuth(actor, credentials);
      default:
        return false;
    }
  }

  private generateClaimCode(): string {
    const prefix = 'TJ';
    const chars = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789';
    let code = '';
    for (let i = 0; i < 6; i++) {
      code += chars.charAt(Math.floor(Math.random() * chars.length));
    }
    return `${prefix}-${code}`;
  }
}
```

1.3.2 Relationship Service

File: `src/services/actors/RelationshipService.ts`

```typescript
import { v4 as uuidv4 } from 'uuid';
import { db } from '@/db';
import { RelationshipType, EstablishedVia } from '@/types/tef';

export class RelationshipService {

  async createRelationship(
    actorAId: string,
    actorBId: string,
    type: RelationshipType,
    permissions: RelationshipPermissions,
    establishedVia: EstablishedVia
```

```
): Promise<Relationship> {
  const relationship = await db.relationships.create({
    id: uuidv4(),
    actor_a_id: actorAId,
    actor_b_id: actorBId,
    relationship_type: type,
    permissions,
    established_via: establishedVia,
    trust_score: 50.00, // Start neutral
    task_count: 0
  });

  // Create bidirectional entry if needed
  if (this.isBidirectional(type)) {
    await db.relationships.create({
      id: uuidv4(),
      actor_a_id: actorBId,
      actor_b_id: actorAId,
      relationship_type: this.getInverseType(type),
      permissions: this.getInversePermissions(permissions),
      established_via: establishedVia,
      trust_score: 50.00,
      task_count: 0
    });
  }

  return relationship;
}

async claimActor(
  claimCode: string,
  claimantId: string,
  displayNameOverride?: string
): Promise<ClaimResult> {
  // Find and validate claim code
  const claimRecord = await db.claimCodes.findByCode(claimCode);
  if (!claimRecord) {
    throw new Error('Invalid claim code');
  }
  if (claimRecord.expires_at < new Date()) {
    throw new Error('Claim code expired');
  }

  // Get the actor being claimed
  const actor = await db.actors.findById(claimRecord.actor_id);
```

```
    if (!actor || actor.status !== 'PENDING_CLAIM') {
      throw new Error('Actor not available for claiming');
    }

    // Update actor status and display name
    await db.actors.update(actor.id, {
      status: 'ACTIVE',
      display_name: displayNameOverride || actor.display_name
    });

    // Create OWNER relationship
    const relationship = await this.createRelationship(
      claimantId,
      actor.id,
      'OWNER',
      {
        can_send_tasks: true,
        can_receive_tasks: true,
        can_delegate: true,
        can_share: true
      },
      'CLAIM_CODE'
    );

    // Invalidate claim code
    await db.claimCodes.delete(claimRecord.id);

    return {
      relationship_id: relationship.id,
      actor_id: actor.id,
      owner_id: claimantId
    };
  }

  async inviteActor(
    inviterId: string,
    inviteeContact: string, // email or phone
    type: RelationshipType,
    permissions: RelationshipPermissions,
    message?: string
  ): Promise<InviteResult> {
    const inviteCode = this.generateInviteCode();

    const invitation = await db.invitations.create({
      id: uuidv4(),
      inviter_id: inviterId,
      invitee_contact: inviteeContact,
      relationship_type: type,
      permissions,
      invite_code: inviteCode,
      message,
      status: 'pending',
      expires_at: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000) // 7 days
    });

    // Send invitation via appropriate channel
    await this.sendInvitation(invitation);

    return {
      invitation_id: invitation.id,
      invite_code: inviteCode,
      expires_at: invitation.expires_at
    };
  }

  async checkPermission(
    actorId: string,
    action: PermissionAction,
    targetId: string
  ): Promise<boolean> {
    const relationship = await db.relationships.findByActors(actorId, targetId);
    if (!relationship) return false;

    const permissions = relationship.permissions as RelationshipPermissions;

    switch (action) {
      case 'send_task':
        return permissions.can_send_tasks ?? false;
      case 'receive_task':
        return permissions.can_receive_tasks ?? false;
      case 'delegate':
        return permissions.can_delegate ?? false;
      case 'share':
        return permissions.can_share ?? false;
      default:
        return false;
    }
  }
}
```

1.4 API Endpoints - Actors

File: `src/routes/api/v1/actors.ts`

```typescript
import { Router } from 'express';
import { ActorService } from '@/services/actors/ActorService';
import { RelationshipService } from '@/services/actors/RelationshipService';
import { authenticate, authorize } from '@/middleware/auth';

const router = Router();
const actorService = new ActorService();
const relationshipService = new RelationshipService();

// POST /api/v1/actors/register - Register new actor
router.post('/register', authenticate, async (req, res) => {
  try {
    const result = await actorService.registerActor(req.body);
    res.status(201).json({
      protocol: 'AIP',
      version: '1.0',
      message_type: 'REGISTRATION_ACK',
      ...result
    });
  } catch (error) {

    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/actors/:id - Get actor details
router.get('/:id', authenticate, async (req, res) => {
  try {
    const actor = await actorService.getActor(req.params.id);
    if (!actor) {
      return res.status(404).json({ error: 'Actor not found' });
    }
    res.json(actor);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// PUT /api/v1/actors/:id - Update actor
// PUT /api/v1/actors/:id - Update actor
router.put('/:id', authenticate, authorize('actor:update'), async (req, res) => {
  try {
    const actor = await actorService.updateActor(req.params.id, req.body);
    res.json(actor);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// DELETE /api/v1/actors/:id - Deactivate actor
router.delete('/:id', authenticate, authorize('actor:delete'), async (req, res) => {
  try {
    await actorService.deactivateActor(req.params.id);
    res.status(204).send();
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/actors/:id/capabilities - List capabilities
router.get('/:id/capabilities', authenticate, async (req, res) => {
  try {
    const capabilities = await actorService.getActorCapabilities(req.params.id);
    res.json({ capabilities });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// GET /api/v1/actors/:id/relationships - List relationships
router.get('/:id/relationships', authenticate, async (req, res) => {
  try {
    const relationships = await relationshipService.getRelationshipsForActor(req.params.id);
    res.json({ relationships });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

export default router;
```

1.5 API Endpoints - Relationships

File: `src/routes/api/v1/relationships.ts` typescript

```typescript
import { Router } from 'express';
import { RelationshipService } from '@/services/actors/RelationshipService';
import { authenticate } from '@/middleware/auth';

const router = Router();
const relationshipService = new RelationshipService();

// POST /api/v1/relationships/claim - Claim actor with code
router.post('/claim', authenticate, async (req, res) => {
  try {
    const { claim_code, display_name_override } = req.body;
    const result = await relationshipService.claimActor(
      claim_code,
      req.user.actor_id,
      display_name_override
    );
    res.status(201).json({
      protocol: 'AIP',
      version: '1.0',
      message_type: 'RELATIONSHIP_ESTABLISHED',
      ...result
    });
  } catch (error) {
    res.status(400).json({ error: error.message });

  }
});

// POST /api/v1/relationships/invite - Send invitation
router.post('/invite', authenticate, async (req, res) => {
  try {
    const { invitee_contact, relationship_type, permissions, message } = req.body;
    const result = await relationshipService.inviteActor(
      req.user.actor_id,
      invitee_contact,
      relationship_type,
      permissions,
      message
    );
    res.status(201).json(result);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/relationships/:id - Get relationship details
router.get('/:id', authenticate, async (req, res) => {
  try {
    const relationship = await relationshipService.getRelationship(req.params.id);
    if (!relationship) {
      return res.status(404).json({ error: 'Relationship not found' });
    }
    res.json(relationship);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// PUT /api/v1/relationships/:id - Update permissions
router.put('/:id', authenticate, async (req, res) => {
  try {
    const relationship = await relationshipService.updatePermissions(
      req.params.id,
      req.body.permissions
    );
    res.json(relationship);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// DELETE /api/v1/relationships/:id - End relationship
router.delete('/:id', authenticate, async (req, res) => {
  try {
    await relationshipService.endRelationship(req.params.id);
    res.status(204).send();
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/relationships/:id/history - Get history
router.get('/:id/history', authenticate, async (req, res) => {
  try {
    const history = await relationshipService.getRelationshipHistory(req.params.id);
    res.json({ history });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// GET /api/v1/relationships/:id/stats - Get statistics
router.get('/:id/stats', authenticate, async (req, res) => {
  try {
    const stats = await relationshipService.getRelationshipStats(req.params.id);
    res.json(stats);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

export default router;
```

1.6 API Endpoints - Tasks (TEF)

File: `src/routes/api/v1/tasks.ts`

```typescript
import { Router } from 'express';
import { TaskService } from '@/services/tasks/TaskService';
import { TEFMessageFactory } from '@/services/tef/TEFMessageFactory';
import { MessageRouter } from '@/services/tef/MessageRouter';
import { authenticate } from '@/middleware/auth';

const router = Router();
const taskService = new TaskService();
const messageRouter = new MessageRouter();

// POST /api/v1/tasks - Create task (TEF TASK_CREATE)
router.post('/', authenticate, async (req, res) => {
  try {
    const task = await taskService.createTask(req.body, req.user.actor_id);

    // Create TEF message
    const tefMessage = TEFMessageFactory.createTaskCreate(
      task,
      req.user.actorRef,
      task.owner
    );
    // Route message
    await messageRouter.routeMessage(tefMessage);

    res.status(201).json(tefMessage);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/tasks/:id - Get task with conversation
router.get('/:id', authenticate, async (req, res) => {
  try {
    const task = await taskService.getTaskWithConversation(req.params.id);
    if (!task) {
      return res.status(404).json({ error: 'Task not found' });
    }
    res.json(task);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// PUT /api/v1/tasks/:id/status - Update status
router.put('/:id/status', authenticate, async (req, res) => {
  try {
    const task = await taskService.updateStatus(
      req.params.id,
      req.body.status,
      req.user.actor_id
    );
    res.json(task);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// POST /api/v1/tasks/:id/accept - Accept task
router.post('/:id/accept', authenticate, async (req, res) => {
  try {
    const task = await taskService.acceptTask(
      req.params.id,
      req.user.actor_id,
      req.body.timeline
    );

    const tefMessage = TEFMessageFactory.createTaskAccept(
    const tefMessage = TEFMessageFactory.createTaskAccept(
      task.task_id,
      task.conversation_id,
      req.user.actorRef,
      task.requestor,
      req.body.timeline
    );

    await messageRouter.routeMessage(tefMessage);

    res.json(tefMessage);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// POST /api/v1/tasks/:id/reject - Reject task
router.post('/:id/reject', authenticate, async (req, res) => {
  try {
    const task = await taskService.rejectTask(
      req.params.id,
      req.user.actor_id,
      req.body.reason
    );
    res.json(task);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// POST /api/v1/tasks/:id/complete - Complete task
router.post('/:id/complete', authenticate, async (req, res) => {
  try {
    const task = await taskService.completeTask(
      req.params.id,
      req.user.actor_id,
      req.body.results
    );

    const tefMessage = TEFMessageFactory.createTaskComplete(
      task.task_id,
      task.conversation_id,
      req.user.actorRef,
      task.requestor,
      req.body.results
    );

    await messageRouter.routeMessage(tefMessage);

    res.json(tefMessage);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// POST /api/v1/tasks/:id/delegate - Delegate task
router.post('/:id/delegate', authenticate, async (req, res) => {
  try {
    const task = await taskService.delegateTask(
      req.params.id,
      req.user.actor_id,
      req.body.delegate_to,
      req.body.reason
    );
    res.json(task);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// GET /api/v1/tasks/:id/conversation - Get all messages
router.get('/:id/conversation', authenticate, async (req, res) => {
  try {
    const messages = await taskService.getConversation(req.params.id);
    res.json({ messages });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// POST /api/v1/tasks/:id/messages - Post message to task
router.post('/:id/messages', authenticate, async (req, res) => {
  try {
    const message = await taskService.addMessage(
      req.params.id,
      req.user.actor_id,
      req.body
    );
    res.status(201).json(message);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

export default router;
```

1.7 Message Router Service

File: `src/services/tef/MessageRouter.ts`

```typescript
import { TEFEnvelope } from '@/types/tef';
import { TEFValidator } from './TEFValidator';
import { db } from '@/db';
import { WebSocketService } from '@/services/websocket/WebSocketService';
import { NotificationService } from '@/services/notifications/NotificationService';

export class MessageRouter {
  private validator: TEFValidator;
  private wsService: WebSocketService;
  private notificationService: NotificationService;

  constructor() {
    this.validator = new TEFValidator();
    this.wsService = new WebSocketService();
    this.notificationService = new NotificationService();
  }


  async routeMessage(message: TEFEnvelope): Promise<RouteResult> {
    // 1. Validate message
    const validation = await this.validator.validateMessage(message);
    if (!validation.valid) {
      throw new Error(`Invalid TEF message: ${validation.errors.join(', ')}`);
    }

    // 2. Validate relationship/permissions
    const hasPermission = await this.validateRelationship(message);
    if (!hasPermission) {
      throw new Error('Sender does not have permission to send to target');
    }

    // 3. Store message
    await this.storeMessage(message);
    // 4. Determine delivery method
    const deliveryMethod = await this.determineDeliveryMethod(message.target_actor);

    // 5. Deliver based on method
    const deliveryResult = await this.deliver(message, deliveryMethod);

    // 6. Notify watchers
    await this.notifyWatchers(message);

    // 7. Record history
    await this.recordHistory(message);

    return {
      message_id: message.message_id,
      delivered: deliveryResult.success,
      delivery_method: deliveryMethod,
      delivered_at: deliveryResult.delivered_at
    };
  }

  private async validateRelationship(message: TEFEnvelope): Promise<boolean> {
    const relationship = await db.relationships.findByActors(
      message.source_actor.actor_id,
      message.target_actor.actor_id
    );

    if (!relationship) return false;

    // Check specific permission based on message type
    const permissions = relationship.permissions;
    if (message.message_type === 'TASK_CREATE') {
      return permissions.can_send_tasks ?? false;
    }

    return true;
  }

  private async determineDeliveryMethod(targetActor: ActorRef): Promise<DeliveryMethod> {
    const actor = await db.actors.findById(targetActor.actor_id);
    if (!actor) throw new Error('Target actor not found');

    // Find preferred contact method
    const contactMethods = actor.contact_methods || [];

    // Priority: websocket > http > mqtt > email
    // Priority: websocket > http > mqtt > email
    const wsMethod = contactMethods.find(m => m.protocol === 'websocket');
    if (wsMethod && this.wsService.isConnected(actor.id)) {
      return { protocol: 'websocket', endpoint: wsMethod.endpoint };
    }

    const httpMethod = contactMethods.find(m => m.protocol === 'http');
    if (httpMethod) {
      return { protocol: 'http', endpoint: httpMethod.endpoint };
    }

    const mqttMethod = contactMethods.find(m => m.protocol === 'mqtt');
    if (mqttMethod) {
      return { protocol: 'mqtt', endpoint: mqttMethod.endpoint };
    }

    // Fall back to notification
    return { protocol: 'notification', endpoint: actor.id };
  }

  private async deliver(
    message: TEFEnvelope,
    method: DeliveryMethod
  ): Promise<DeliveryResult> {
    switch (method.protocol) {
      case 'websocket':
        return this.deliverWebSocket(message, method.endpoint);
      case 'http':
        return this.deliverHTTP(message, method.endpoint);
      case 'mqtt':
        return this.deliverMQTT(message, method.endpoint);
      case 'notification':
        return this.deliverNotification(message, method.endpoint);
      default:
        throw new Error(`Unknown delivery protocol: ${method.protocol}`);
    }
  }

  private async deliverWebSocket(
    message: TEFEnvelope,
    actorId: string
  ): Promise<DeliveryResult> {
    const success = await this.wsService.send(actorId, message);
    return {
      success,
      delivered_at: success ? new Date().toISOString() : undefined
      delivered_at: success ? new Date().toISOString() : undefined
    };
  }

  private async deliverHTTP(
    message: TEFEnvelope,
    endpoint: string
  ): Promise<DeliveryResult> {
    try {
      const response = await fetch(endpoint, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(message)
      });
      return {
        success: response.ok,
        delivered_at: response.ok ? new Date().toISOString() : undefined
      };
    } catch (error) {
      return { success: false };
    }
  }

  private async storeMessage(message: TEFEnvelope): Promise<void> {
    // Ensure conversation exists
    let conversation = await db.conversations.findByTaskId(message.task_id);
    if (!conversation) {
      conversation = await db.conversations.create({
        task_id: message.task_id,
        participants: [message.source_actor.actor_id, message.target_actor.actor_id],
        message_count: 0
      });
    }

    // Store message
    await db.messages.create({
      conversation_id: conversation.id,
      task_id: message.task_id,
      message_type: message.message_type,
      source_actor_id: message.source_actor.actor_id,
      target_actor_id: message.target_actor.actor_id,
      reply_to_id: message.reply_to_message_id,
      payload: message
    });

    // Update conversation
```

```
    // Update conversation
    await db.conversations.update(conversation.id, {
      message_count: conversation.message_count + 1,
      last_message_at: new Date()
    });
  }

  private async notifyWatchers(message: TEFEnvelope): Promise<void> {
    const task = await db.tasks.findById(message.task_id);
    if (!task?.watchers) return;

    for (const watcher of task.watchers) {
      if (watcher.actor_id !== message.source_actor.actor_id) {
        await this.notificationService.notify(watcher.actor_id, {
          type: 'task_update',
          task_id: message.task_id,
          message_type: message.message_type,
          from: message.source_actor.display_name
        });
      }
    }
  }

  private async recordHistory(message: TEFEnvelope): Promise<void> {
    const relationship = await db.relationships.findByActors(
      message.source_actor.actor_id,
      message.target_actor.actor_id
    );

    if (!relationship) return;

    const eventType = this.messageTypeToEventType(message.message_type);
    if (!eventType) return;

    await db.relationshipHistory.create({
      relationship_id: relationship.id,
      actor_a_id: message.source_actor.actor_id,
      actor_b_id: message.target_actor.actor_id,
      task_id: message.task_id,
      event_type: eventType,
      metadata: { message_type: message.message_type }
    });
  }
}
```

1.8 Human Actor Migration

File: `database/migrations/006_migrate_users_to_actors.sql`

```sql
-- Migration: Convert existing users to HUMAN actors

-- 1. Create actors from users
INSERT INTO actors (
    id,
    actor_type,
    display_name,
    capabilities,
    contact_methods,
    metadata,
    status,
    organization_id,
    created_at
)
SELECT
    id,
    'HUMAN'::actor_type,
    COALESCE(full_name, email),
    '["task.create", "task.receive", "task.delegate"]'::jsonb,
    jsonb_build_array(
        jsonb_build_object(
            'protocol', 'http',

            'endpoint', 'https://api.taskjuggler.io/users/' || id || '/inbox'
        ),
        CASE WHEN email IS NOT NULL THEN
            jsonb_build_object('protocol', 'email', 'endpoint', email)
        ELSE NULL END,
        CASE WHEN phone IS NOT NULL THEN
            jsonb_build_object('protocol', 'sms', 'endpoint', phone)
        ELSE NULL END
    ) - NULL, -- Remove null entries
    jsonb_build_object(
        'timezone', COALESCE(timezone, 'UTC'),
        'language', COALESCE(language, 'en'),
        'notification_preferences', notification_preferences
    ),
    'ACTIVE'::actor_status,
    organization_id,
    created_at
FROM users
WHERE NOT EXISTS (
    SELECT 1 FROM actors WHERE actors.id = users.id
);

-- 2. Create relationships from existing task connections
INSERT INTO relationships (
    id,
    actor_a_id,
    actor_b_id,
    relationship_type,
    permissions,
    established_via,
    trust_score,
    task_count,
    created_at
)
SELECT DISTINCT ON (requestor_id, owner_id)
    gen_random_uuid(),
    requestor_id,
    owner_id,
    'PEER'::relationship_type,
    '{"can_send_tasks": true, "can_receive_tasks": true}'::jsonb,
    'API'::established_via,
    50.00,
    COUNT(*) OVER (PARTITION BY requestor_id, owner_id),
    MIN(created_at) OVER (PARTITION BY requestor_id, owner_id)
FROM tasks
WHERE requestor_id IS NOT NULL
  AND owner_id IS NOT NULL
  AND requestor_id != owner_id;

-- 3. Update tasks to use conversation_id
ALTER TABLE tasks ADD COLUMN IF NOT EXISTS conversation_id UUID;

-- Create conversations for existing tasks
INSERT INTO conversations (id, task_id, participants, message_count, created_at)
SELECT
    gen_random_uuid(),
    id,
    ARRAY[requestor_id, owner_id]::uuid[],
    0,
    created_at
FROM tasks
WHERE conversation_id IS NULL
WHERE conversation_id IS NULL;

-- Update tasks with conversation_id
UPDATE tasks t
SET conversation_id = c.id
FROM conversations c
WHERE c.task_id = t.id AND t.conversation_id IS NULL;

-- 4. Add foreign key constraints
ALTER TABLE tasks
ADD CONSTRAINT fk_tasks_conversation
FOREIGN KEY (conversation_id) REFERENCES conversations(id);

-- 5. Create backward-compatible views
CREATE OR REPLACE VIEW users_view AS
SELECT
    a.id,
    a.display_name AS full_name,
    (a.contact_methods->0->>'endpoint') AS email,
    (a.metadata->>'timezone') AS timezone,
    a.organization_id,
    a.status = 'ACTIVE' AS is_active,
    a.created_at,
    a.updated_at
FROM actors a
WHERE a.actor_type = 'HUMAN';
```

1.9 Phase 1 Deliverables Checklist

- [ ] Database: actors, relationships, conversations, messages, relationship_history, delegation_rules tables
- [ ] Types: Complete TEF TypeScript interfaces in `src/types/tef.ts`
- [ ] Services: ActorService, RelationshipService, TEFMessageFactory, TEFValidator, MessageRouter
- [ ] API: All actor, relationship, and task endpoints
- [ ] WebSocket: Real-time TEF message delivery
- [ ] Migration: Existing users converted to HUMAN actors
- [ ] Tests: Unit tests for all services, integration tests for API

Phase 2: IoT Integration (Months 4-6)

[Continue with detailed Phase 2 implementation...]

2.1 MQTT Broker Setup

2.1.1 Docker Compose Configuration

File: `infrastructure/mqtt/docker-compose.yml`

```yaml
version: '3.8'
services:
  mqtt-broker:
    image: eclipse-mosquitto:2.0
    container_name: tef-mqtt-broker
    ports:
      - "1883:1883"   # MQTT
      - "8883:8883"   # MQTTS (TLS)
      - "9001:9001"   # WebSocket
    volumes:
      - ./config/mosquitto.conf:/mosquitto/config/mosquitto.conf
      - ./certs:/mosquitto/certs
      - mqtt-data:/mosquitto/data
      - mqtt-log:/mosquitto/log
    restart: unless-stopped

volumes:
  mqtt-data:
  mqtt-log:
```

2.1.2 MQTT Bridge Service

File: `src/services/protocols/MQTTBridge.ts`

```typescript
import mqtt from 'mqtt';
import cbor from 'cbor';
import { TEFEnvelope } from '@/types/tef';
import { MessageRouter } from '@/services/tef/MessageRouter';

export class MQTTBridge {
  private client: mqtt.MqttClient;
  private messageRouter: MessageRouter;

  constructor() {
    this.messageRouter = new MessageRouter();
```

```
}

async connect(): Promise<void> {
  this.client = mqtt.connect(process.env.MQTT_BROKER_URL, {
    clientId: 'tef-exchange-bridge',
    username: process.env.MQTT_USERNAME,
    password: process.env.MQTT_PASSWORD,
    ca: fs.readFileSync(process.env.MQTT_CA_CERT),
    cert: fs.readFileSync(process.env.MQTT_CLIENT_CERT),
    key: fs.readFileSync(process.env.MQTT_CLIENT_KEY)
  });

  this.client.on('connect', () => {
    console.log('Connected to MQTT broker');
    this.subscribeToTopics();
  });

  this.client.on('message', this.handleMessage.bind(this));
}

private subscribeToTopics(): void {
  // Subscribe to device registration
  this.client.subscribe('tef/+/register');
  // Subscribe to device responses
  this.client.subscribe('tef/+/+/response');
}

private async handleMessage(topic: string, payload: Buffer): Promise<void> {
  try {
    // Decode CBOR payload
    const message = cbor.decode(payload) as TEFEnvelope;

    // Route through standard TEF pipeline
    await this.messageRouter.routeMessage(message);
  } catch (error) {
    console.error('Error handling MQTT message:', error);
  }
}

async publishToDevice(actorId: string, message: TEFEnvelope): Promise<void> {
  const topic = `tef/devices/${actorId}/inbox`;
  const payload = cbor.encode(message);

  // Determine QoS based on priority
  const qos = this.priorityToQoS(message.task?.priority);
```

```
    await this.client.publish(topic, payload, { qos });
  }

  private priorityToQoS(priority?: string): 0 | 1 | 2 {
    switch (priority) {
      case 'CRITICAL':
      case 'HIGH':
        return 2; // Exactly once
      case 'NORMAL':
        return 1; // At least once
      default:
        return 0; // At most once
    }
  }
}
```

[Continue with detailed sections for Phases 3 and 4...]

Appendix A: Complete File Structure

TaskJuggler (TEF Exchange)

```
taskjuggler/
├── database/

│   └── migrations/
│       ├── 001_create_actors_table.sql
│       ├── 002_create_relationships_table.sql
│       ├── 003_create_conversations_table.sql
│       ├── 004_create_relationship_history_table.sql
│       ├── 005_create_delegation_rules_table.sql
│       └── 006_migrate_users_to_actors.sql
├── src/
│   ├── types/
│   │   └── tef.ts
│   ├── services/
│   │   ├── tef/
│   │   │   ├── TEFMessageFactory.ts
│   │   │   ├── TEFValidator.ts
│   │   │   └── MessageRouter.ts
│   │   ├── actors/
│   │   │   ├── ActorService.ts
│   │   │   └── RelationshipService.ts
│   │   ├── iot/
│   │   │   ├── DeviceRegistrationService.ts
│   │   │   ├── DeviceClaimService.ts
│   │   │   └── CapabilityRegistry.ts
│   │   ├── ai/
│   │   │   ├── AIAgentService.ts
│   │   │   └── AIEscalationService.ts
│   │   ├── delegation/
│   │   │   └── DelegationService.ts
│   │   ├── trust/
│   │   │   └── TrustCalculator.ts
│   │   ├── routing/
│   │   │   ├── RoutingRulesEngine.ts
│   │   │   └── LoadBalancer.ts
│   │   └── protocols/
│   │       ├── MQTTBridge.ts
│   │       ├── MCPServer.ts
│   │       └── CoAPBridge.ts
│   └── routes/
│       └── api/
│           └── v1/
│               ├── actors.ts
│               ├── relationships.ts
│               ├── tasks.ts
│               ├── delegations.ts
│               └── iot.ts
├── web/
│   └── src/
│       └── components/
│           ├── iot/
│           │   ├── DeviceList.vue
│           │   ├── DeviceCard.vue
│           │   └── DeviceClaimModal.vue
│           └── ai/
│               ├── AIAgentList.vue
│               └── DelegationManager.vue
└── mobile/
    └── src/
        └── screens/
            ├── iot/
            │   ├── DevicesScreen.tsx
            │   └── ClaimDeviceScreen.tsx
            └── ai/
        └── AIAgentsScreen.tsx
```

Process.AI (Task Orchestrator)

```
process-ai/
├── src/
│   └── services/
│       ├── tef/
│       │   ├── PlatformRegistration.ts
│       │   └── EventSubscriber.ts
│       ├── orchestration/
│       │   ├── WorkflowOrchestrator.ts
│       │   └── WorkflowEngine.ts
│       ├── analytics/
│       │   ├── PatternDetector.ts
│       │   └── ProcessAnalytics.ts
│       └── compliance/
│           ├── ComplianceMonitor.ts
│           └── ComplianceReporter.ts
└── web/
    └── src/
        └── components/
            ├── WorkflowDesigner.vue
            └── ComplianceReports.vue
```

Projects.AI (Task Aggregator)

```
projects-ai/
├── src/
│   └── services/
│       ├── tef/
│       │   ├── PlatformRegistration.ts
│       │   └── EventSubscriber.ts
│       ├── aggregation/
│       │   ├── TaskAggregator.ts
│       │   └── MetricsCalculator.ts
│       ├── resources/
│       │   ├── ResourceManager.ts
│       │   └── CapacityPlanner.ts
│       └── analytics/
│           ├── ProjectAnalytics.ts
│           └── PortfolioAnalytics.ts
│
```

```
└── web/
    └── src/
        └── components/
            ├── ProjectTaskView.vue
            ├── ResourceAllocation.vue
            └── PortfolioOverview.vue
```

Appendix B: Dependencies

NPM Dependencies (TaskJuggler)

```json
{
  "dependencies": {
    "uuid": "^9.0.0",
    "ajv": "^8.12.0",
    "mqtt": "^5.0.0",
    "cbor": "^9.0.0",
    "socket.io": "^4.6.0",
    "@modelcontextprotocol/sdk": "^1.0.0",
    "ioredis": "^5.3.0",
    "stripe": "^14.0.0"
  }
}
```

Appendix C: Success Metrics

Phase 1

- [ ] 100% of existing users migrated to actors
- [ ] TEF message format validated against schema
- [ ] Human-to-human tasks flowing through exchange
- [ ] All API endpoints passing integration tests
- [ ] WebSocket real-time delivery < 100ms

Phase 2

- [ ] MQTT broker handling 10,000+ connected devices