# Project: Insurance Management Web Application Specification

**Goal:** Deliver a working, end-to-end insurance management web application implementing core features (policies, claims, payments, users/agents), with enterprise-minded design (security, scalability, maintainability), and modern development practices (testing, CI/CD, documentation).

## 1. Project Overview

### 1.1 Scope

A lightweight insurance management web application with: - User authentication & authorization (customer, agent, admin) - Policy browsing, purchase (basic flow), and policy management - Claim submission and basic claim processing - Payment recording (no payment gateway integration — dummy) - Agent management and simple assignment - Audit logging and minimal reporting endpoints

The app will include a simple Angular frontend, an Express/Node backend using Mongoose (MongoDB), and basic deployment instructions.

### 1.2 Assumptions & Constraints

* Time box: 4 days/32 hours
* No production-grade payment gateway integration (simulate success/failure).
* One developer building and testing.

## 2. Functional Requirements

Each item lists endpoints, brief description, and important request/response shapes.

Common notes: All backend endpoints under base path /api/v1.

### 2.1 Authentication & User Management

**Endpoints** –

POST /api/v1/auth/register — register a new user (role: customer or agent).

Request: { name, email, password, role }. -

POST /api/v1/auth/login — returns JWT. Request: { email, password }. Response: { token, user: {...} }. - GET /api/v1/auth/me — get current user (protected).

**Roles**: customer, agent, admin.

**Notes**: JWT used for stateless auth.

### 

### 2.2 Policies

**Entity**: Policy template (products) and UserPolicy (purchased policies).

**Endpoints** –

GET /api/v1/policies — list available policy products (public). –

GET /api/v1/policies/:id — get product details. –

POST /api/v1/policies/:id/purchase — purchase policy (protected customer). Request: { startDate, termMonths, nominee }. Response: created UserPolicy resource. –

GET /api/v1/user/policies — list policies owned by current user (protected). –

PUT /api/v1/user/policies/:id/cancel — cancel policy (protected, within business rules).

### 

### 2.3 Claims

**Entity**: Claim (linked to UserPolicy).

**Endpoints** –

POST /api/v1/claims — submit a claim (protected customer). Request: { policyId, incidentDate, description, amount }. –

GET /api/v1/claims — list claims (admin/agent sees all; customer sees own). –

GET /api/v1/claims/:id — claim detail (role-based access). –

PUT /api/v1/claims/:id/status — update status (PENDING, APPROVED, REJECTED) (protected agent or admin). Request: { status, notes }.

### 

### 2.4 Payments (Simulated)

**Entity**: Payment record

**Endpoints** –

POST /api/v1/payments — record payment for a policy (protected). Request: { policyId, amount, method, reference }. –

GET /api/v1/payments/user — list payments for current user.

### 

### 2.5 Agents & Admin

**Endpoints** –

GET /api/v1/agents — list agents (admin only). –

POST /api/v1/agents — create agent (admin). –

PUT /api/v1/agents/:id/assign — assign agent to a policy or claim (admin).

### 

### 2.6 Audit & Reporting (Minimal)

**Endpoints** –

GET /api/v1/admin/audit — last N audit logs (admin only). –

GET /api/v1/admin/summary — minimal KPIs: counts of users, policies sold, claims pending, total payments.

## 3. Data Model (Mongoose Schemas)

Below are the primary collections and example fields.

### 3.1 User

{  
 \_id: ObjectId,  
 name: String,  
 email: String (unique),  
 passwordHash: String,  
 role: String (customer|agent|admin),  
 createdAt: Date,  
 updatedAt: Date  
}

### 

### 3.2 PolicyProduct

{  
 \_id: ObjectId,  
 code: String,  
 title: String,  
 description: String,  
 premium: Number, // monthly or yearly depending on term  
 termMonths: Number,  
 minSumInsured: Number,  
 createdAt: Date  
}

### 

### 3.3 UserPolicy (purchased policy)

{  
 \_id: ObjectId,  
 userId: ObjectId,  
 policyProductId: ObjectId,  
 startDate: Date,  
 endDate: Date,  
 premiumPaid: Number,  
 status: String (ACTIVE|CANCELLED|EXPIRED),  
 assignedAgentId: ObjectId?,  
 nominee: { name, relation },  
 createdAt: Date  
}

### 3.4 Claim

{  
 \_id: ObjectId,  
 userId: ObjectId,  
 userPolicyId: ObjectId,  
 incidentDate: Date,  
 description: String,  
 amountClaimed: Number,  
 status: String (PENDING|APPROVED|REJECTED),  
 decisionNotes: String,  
 decidedByAgentId: ObjectId?,  
 createdAt: Date  
}

### 3.5 Payment

{  
 \_id: ObjectId,  
 userId: ObjectId,  
 userPolicyId: ObjectId,  
 amount: Number,  
 method: String (CARD|NETBANKING|OFFLINE|SIMULATED),  
 reference: String,  
 createdAt: Date  
}

### 3.6 AuditLog

{  
 \_id: ObjectId,  
 action: String,  
 actorId: ObjectId,  
 details: Object,  
 ip: String,  
 timestamp: Date  
}

## 4. Non-Functional Requirements

### 4.1 Security

* Authenticate with JWT, short expiry (e.g., 1h) + refresh token strategy optional.
* Input validation with express-validator or Joi.
* Role-based access control (middleware to check role/ownership).

### 4.2 Scalability & Performance

* Keep backend stateless (JWT) to allow horizontal scaling.
* Use connection pooling for MongoDB and add indexes on frequently queried fields (email, userId, createdAt).

### 4.3 Maintainability

* Layered structure: routes → controllers → services → models.
* Use ESLint + Prettier; commit hooks (husky) for linting/tests on commit.
* Modularize features in folders (auth, policy, claim, payment).
* Write small unit-tested functions where feasible.

### 4.4 Availability & Reliability

* For this MVP: document backup and restore (MongoDB mongodump instructions) and logs.

## 5. Architecture & Tech Stack

**Frontend**: Angular (v17+), Angular CLI, RxJS, simple component structure.

**Backend**: Node.js (e.g., v18+), Express, Mongoose, JWT auth,

**Database**: MongoDB (Atlas or local). Use a development DB and a production DB URI.

**Dev Tools**: Git, GitHub, VS Code, Postman

**Testing**: Jest (backend), Jasmine/Karma (Angular unit), Playwright for simple e2e (optional if time).

**API Docs**: Swagger (OpenAPI)

## 6. API Examples (Detailed)

### 6.1 Register

POST /api/v1/auth/register Request body:

{ "name": "Ravi Kumar", "email": "ravi@example.com", "password": "pass@123", "role": "customer" }

Response (201):

{ "message": "registered", "user": { "\_id":"...", "name":"Ravi Kumar", "email":"ravi@example.com", "role":"customer" } }

### 6.2 Login

POST /api/v1/auth/login Request:

{ "email": "ravi@example.com", "password": "pass@123" }

Response:

{ "token": "<jwt>", "user": { "\_id": "...", "name": "Ravi" } }

### 6.3 Purchase Policy

POST /api/v1/policies/:id/purchase (Auth: Bearer) Request:

{ "startDate": "2025-09-01", "termMonths": 12, "nominee": { "name":"Asha", "relation":"spouse" } }

Response: created UserPolicy JSON.

(Other endpoints follow similar patterns.)

## 7. Minimal UI Pages (Angular)

1. Public Home / Policies listing
2. Policy detail page
3. Register / Login
4. User Dashboard — My Policies, Submit Claim, My Payments
5. Agent Dashboard — Assigned Claims
6. Admin Dashboard — Summary KPIs, Manage Agents & Policies

## 8. Testing Strategy

* **Unit tests**: Small controller/service tests (Jest) — aim for a few critical endpoints (auth, purchase, claims).
* **E2E**: Playwright test to run a login→purchase→submit-claim scenario (Optional)
* **Manual tests**: Use Postman collection for all endpoints.

## 9. CI/CD (GitHub Actions) — Minimal pipeline

**Workflow**: 1. On push to main or PR: run npm ci, npm run lint, npm test (backend); build Angular app. 2. If all pass, optionally deploy to a staging host

Use environment variables for DB URI and JWT secret.

Provide a sample .github/workflows/ci.yml template in the repo (not included inline — you can scaffold from templates).

## 10. Implementation Tips & Minimal Folder Structure

backend/  
 src/  
 models/ # Mongoose schemas  
 controllers/  
 services/  
 routes/  
 middlewares/  
 utils/  
 index.js  
frontend/  
 src/  
 app/  
 components/  
 pages/  
 services/http.service.ts  
 environments/

Code style: ESLint config + Prettier. Keep controllers thin and move business logic into services.

## 11. Acceptance Criteria (MVP)

* User can register/login and view policies.
* Customer can purchase a policy and see it in their dashboard.
* Customer can submit a claim for a purchased policy.
* Agent/Admin can view and change claim status.
* Payments are recorded.
* Basic security middleware in place and basic tests running in CI.