Requirements and Architecture for Marketplace API Development

✅ 1. Requirements Gathering

📌 Key Functionalities

- **User registration, login, and profile sync**
- **Marketplace browsing** (listings, agrotourism, services)
- **Transaction initiation** (with PING loop)
- **Node verification via Registry API**
- **Event broadcast and geofencing**
- **Upload and status reporting**

🧩 User Stories

- "As a farmer, I want to list my produce and manage sales through the frontend."
- "As a buyer, I want to browse listings and receive notifications."
- "As a co-op node admin, I want to validate federation node health."

User stories will inform **form layouts**, **API endpoints**, and **feedback handling** (loading, errors, confirmations).

✅ 2. Architecture Design

Pattern: Service-Oriented Architecture

- **Wix Frontend**: Static pages + Velo JS + optional proxy endpoints
- **Backend**: RESTful APIs hosted on EC2 (`api.ntari.org`, `registry.ntari.org`)

Data Flow

- **REST-based interactions** over HTTPS
- Optional Velo backend proxy for secure token injection or CORS mitigation

✅ 3. API Design

Essential Endpoints

- `POST /login` – user authentication
- `GET /marketplace` – fetch public listings
- `POST /transaction` – initiate or record transaction
- `GET /registry/health` – node status check
- `GET /ping` – get broadcast or alerts

RESTful Best Practices

- Use nouns in URLs
- Use `GET`, `POST`, `PUT`, `DELETE` verbs properly
- Consistent response format (`{ status, data, error }`)

✅ 4. Data Handling

Validation

- **Frontend**: Client-side checks (empty fields, formatting)
- **Backend**: Schema validation (e.g., Joi or custom middleware)

Synchronization

- API returns timestamps or version IDs
- Transactions use `PING_ID` or `TX_ID` for audit trail

✅ 5. Security Considerations

In Transit

- HTTPS enforced across all connections
- CORS headers restricted to `*.ntari.org`

Auth

- JWT-based authentication (`Authorization: Bearer <token>`)
- User sessions managed client-side; backend validates token

✅ 6. Error Handling

Strategy

- **Backend**: Return structured error `{ status: 'error', message: '...' }`
- **Frontend**: Handle `4xx`, `5xx` with user-friendly messages

Logging

- Backend logs errors with context (e.g., failed DB write)
- Frontend logs major failures to console or Sentry (optional)

✅ 7. Testing Strategies

Types

- **Unit tests** for backend controllers and frontend modules
- **Integration tests** for API endpoints
- **E2E tests** with Cypress or Playwright (frontend flows)

Structure

- Separate tests per route/module
- Include failure cases (e.g., auth denied, missing field)

✅ 8. Deployment and CI/CD

Flow

- GitHub Actions or Bitbucket Pipelines
- Deploy backend to EC2 via `ssh` or Docker
- Frontend changes published via Wix deployment system

Coordination

- Use versioned APIs (`/v1/`) to prevent frontend breaking
- Maintain `.env.staging` and `.env.production` configs

✅ 9. Performance Monitoring

Metrics

- **API latency** and throughput
- **Error rate** (5xx or auth issues)
- **User bounce rates** (frontend interaction)

Tools

- Backend: `PM2 + logrotate`, `uptime monitoring (e.g., UptimeRobot)`
- Frontend: Wix Performance Dashboard

✅ 10. User Experience

Impact

- Fast responses improve trust
- Clear messaging on submission, errors, and success

Feedback Mechanisms

- Use `#statusText` or modals to show API responses
- Optional: integrate Airtable/Typeform for feedback

🚀 Next Steps:

1. Define actual API endpoints and expected payloads
2. Create Velo backend proxy template (if needed)
3. Build API interaction logic for the Wix frontend pages