Customer Contract Management Portal — System Architecture

Diagram

```
flowchart LR
  subgraph Frontend
    A[React SPA]
  end

  subgraph Backend
    B[API Gateway / Reverse Proxy]
    C[Backend App (FastAPI / Express)]
    CA[Action Service]
    CE[Event Ingest Service]
    AU[Auth / SSO (SAML/OIDC)]
    NOTIF[Notification Service]
    ANALYTICS[Optional AI / Analytics]
  end

  subgraph DataLayer
    DB[SQLite (Primary) \n customers, contracts, events, notes]
    AUDIT[Immutable Audit Store / Append-only Log]
    SEARCH[Optional Full-text Search / Index]
    DW[Optional Data Warehouse]
  end

  subgraph Observability
    LOGS[Centralized Logs]
    TRACES[Distributed Tracing]
    METRICS[Metrics & Alerts]
  end

  A -->|HTTPS REST| B
  B --> C
  C -->|SQL| DB
  C -->|write| AUDIT
  CE -->|ingest events (HTTP/gRPC)| C
  CE -->|append| DB
  C --> SEARCH
  NOTIF -->|email/IM| ExternalSystems[(Email / Internal Chat)]
  C --> AU
```

```
ANALYTICS -.-> DB
ANALYTICS -.-> C

C --> LOGS
C --> TRACES
C --> METRICS
CE --> LOGS
B -->|TLS termination| AU

%% Endpoints
classDef endpoints fill:#f9f,stroke:#333;
subgraph Endpoints[Main REST Endpoints]
  E1[/customers]
  E2[/contracts]
  E3[/contracts/{id}]
  E4[/events]
  E5[/actions]
end
B --> E1
B --> E2
B --> E3
B --> E4
B --> E5

%% Future integrations
ExternalAI[(External AI / Risk Scoring)]
ExternalIDP[(Enterprise IdP)]
ExternalEventSources[(Event Producers / Kafka)]
ExternalSIEM[(SIEM / Security)]

CE --> ExternalEventSources
ANALYTICS --> ExternalAI
LOGS --> ExternalSIEM
AU --> ExternalIDP
```

Textual explanation

High-level overview

This system is a classic three-tier web app:

- **Frontend**: React single-page application providing pages (Dashboard, Customers, Contracts, Contract Details, Event Log). Communicates with backend via HTTPS REST API; uses token-based session from SSO (OIDC/SAML).

- **Backend**: Monolithic or modular service implemented in **FastAPI (Python)** or **Express (Node.js)** exposing the main REST endpoints. Implements business logic (state machine for contract lifecycle), RBAC checks, input validation, audit recording, and event ingestion.
- **Database**: SQLite as requested for Phase 1. Stores `customers`, `contracts`, `events`, and `notes`. An append-only **Audit Store** is used to keep tamper-evident records for approvals, flags, and comments.

Components and responsibilities

1. **React SPA (Frontend)**
2. Pages: Dashboard, Customers, Contracts, Contract Details, Event Log.
3. Uses OIDC/SAML SSO for authentication and receives a scoped access token and user role claims.
4. Calls backend REST endpoints, displays data tables and charts, enforces client-side input validation and masking rules (partial national ID, emails).
5. Handles optimistic UI updates for comments/flags and displays action results with proper error handling.
6. **API Gateway / Reverse Proxy**
7. TLS termination, basic rate limiting, routing to backend instances, and static asset hosting.
8. Enforces authentication early, validates tokens with IdP, and injects user context (user id, roles) into requests.
9. **Backend App (FastAPI / Express)**
10. Exposes REST endpoints: `/customers`, `/contracts`, `/contracts/{id}`, `/events`, `/actions`.
11. Responsibilities: input validation, authorization (RBAC + row-level checks), business workflows (contract state machine), comments/flags/actions handling, CSV export generation, and pagination.
12. Persists all mutating operations to the **Audit Store** in append-only fashion and writes normalized data to SQLite.
13. Event ingestion endpoint(s) accept external events and persist into `events` table after schema validation.
14. **Action Service (module / queue consumer)**
15. Processes long-running or side-effectful actions (e.g., notifications, heavy exports, or integration calls). Keeps API responsive by returning accepted responses and doing work asynchronously.
16. **Event Ingest Service**
17. Dedicated endpoint for event producers. Validates schema, deduplicates, and writes into `events` table. If event volume grows, this can buffer into Kafka or a queue.
18. **Auth / SSO (Enterprise IdP)**

19. SAML/OIDC provider for single sign-on and MFA. Backend validates tokens and applies role-based claims.

20. **Notification Service**

21. Sends email or internal notifications when an approver is needed or when @mentions occur.

22. **Analytics / AI (Optional)**

23. Offline or microservice that consumes event/contract data to produce risk scores, trends, and suggested actions. Integrates with backend via an authenticated API or database exports.

24. **Data Layer**

25. **SQLite** as primary Phase-1 datastore. Tables: `customers`, `contracts`, `events`, `notes`. Use WAL mode and file-level backups for durability.
26. **Audit Store**: could be a separate append-only file, secure blob store, or a write-once S3 bucket with integrity hashes to meet tamper-evidence and retention.

27. **Optional Search Index** (e.g., Tantivy / Elasticsearch in future) for fast full-text search over contracts and comments.

28. **Observability**

    - Centralized logs, metrics, and tracing (e.g., Prometheus + Grafana, Jaeger). Export logs to SIEM for compliance.

Data flow (typical scenarios)

Contract approval flow 1. Approver loads Contract Details page in React. Frontend calls `GET /contracts/{id}`. 2. Backend authorizes the call, reads from SQLite, returns contract data and comments. 3. Approver clicks Approve → frontend `POST /actions` with `{action: 'approve', contract_id, comment}`. 4. Backend validates role, records `ContractAction` in Audit Store and `contracts` table status change, then returns success. Backend enqueues notifications for interested parties. 5. Notification Service sends internal message or email.

Event ingestion flow 1. External system posts event to `/events` (or to Event Ingest Service). 2. Event service validates schema and writes event to `events` table and `audit` if required. 3. Dashboard or Event Log queries `GET /events` with filters to display recent events.

REST endpoints (concise mapping)

- `GET /customers` — list, search, filters, pagination
- `GET /customers/{id}` — detail with associated contracts
- `GET /contracts` — list and filters
- `GET /contracts/{id}` — contract details, comments, flags, history
- `GET /events` — query events by customer, type, time range

- `POST /actions` — `{action: 'approve'|'reject'|'flag'|'note', payload: {...}}` (authz required)
- `POST /events` — event ingestion (secure, authenticated producer)
- `GET /export` — CSV export endpoint (admin/analyst permissions)

Best practices — Security

- Use enterprise SSO (SAML/OIDC) + MFA. Validate tokens at the gateway.
- Enforce RBAC and row-level authorization in the backend for all mutating/read operations.
- Encrypt transport with TLS 1.2+ and enable HSTS.
- Encrypt sensitive fields at rest (AES-256) — especially National ID and any PII; use field-level encryption if possible.
- Store secrets in a vault and rotate regularly.
- Implement CSRF protections and strong Content Security Policy in frontend.
- Input validation and strict schema for events; sanitize and scan attachments for malware.
- Immutable audit logs with write-once semantics; sign exported CSVs using a hash for integrity.

Best practices — Scalability & Availability

- SQLite is fine for Phase 1 and low concurrency proofs of concept. For production scale targets (1M customers, 500M events) plan migration to a client-server RDBMS (Postgres, cloud managed RDS) and use partitioning for events.
- Introduce read replicas and a caching layer (Redis) for hot reads (dashboard counts, frequent queries).
- Move event ingestion to an append-only distributed log (Kafka or cloud streaming) to decouple producers from the portal.
- Use horizontal scaling for backend (stateless containers) behind the API gateway.
- Use async workers (Celery / RQ / Bull) for heavy tasks (exports, notifications).

Best practices — Modularity & Maintainability

- Keep bounded contexts: split services (ingest, actions, analytics) as separate modules/services.
- Define clear API contracts and use OpenAPI / AsyncAPI for documentation and client generation.
- Write comprehensive tests: unit, integration, and contract tests for the event schema.
- Feature flags for gating dashboard widgets and edit windows.
- CI/CD with automated security scans (SAST/DAST) and deployment pipelines per environment.

Deployment considerations

- Environments: dev, staging, prod. Use IaC (Terraform) for infra.
- Back up SQLite files regularly; ensure consistent snapshots when writing (use WAL + file system snapshot) or migrate to managed DB for safer backups.
- Retention policy: keep audit and events per compliance (7 years); consider cold storage for older events.

Notes & trade-offs

- **SQLite advantages**: simple to deploy, low ops overhead for Phase 1. **Limitations**: concurrency, scalability, backups and multi-tenant safety — plan to migrate to Postgres or cloud RDS for scale and reliability.
- **Audit requirements**: implement append-only audit store separate from writable DB to meet tamper-evident requirements.
- **Performance**: heavy event volumes require sharding/partitioning — implement time-based partitions and/or move events to a purpose-built event store.

If you'd like, I can: - Produce a sequence diagram for the contract approval flow. - Produce an infrastructure diagram (cloud provider specific — AWS/Azure/GCP) with recommended managed services. - Generate Kubernetes manifests / Dockerfile + sample FastAPI or Express skeleton implementing the endpoints.

Tell me which of those you'd like next and I'll add it directly into this canvas.