ADR-2: Database and Schema Decision

1 CONTEXT

The DECODER API requires persistent storage for sensor readings, buildings, and users. A decision must be made on:

1. Database type (SQL vs NoSQL)
2. Specific database technology
3. Schema design
4. Data access strategy (ORM vs raw SQL)

2 DECISION

I have chosen to use SQLite as the database with JPA/Hibernate for object-relational mapping, using a relational schema with three main tables: readings, buildings, and users.

3 RATIONALE

3.1 WHY SQLITE?

3.1.1 Simplicity for Research Slice

- No separate database server required
- Single file database (decoder.db)
- Perfect for local development and demonstration
- Zero configuration needed

3.1.2 Easy Deployment

- Database file can be included or generated
- No external dependencies for basic setup
- Can easily switch to PostgreSQL for production

3.1.3 SQL Compliance

- Uses standard SQL syntax
- Easy to migrate to PostgreSQL later
- Supports all required features (foreign keys, indexes, etc.)

3.2 WHY RELATIONAL SCHEMA?

3.2.1 Structured Data

- Sensor readings have clear relationships (building → readings)
- Users have clear relationships (user → buildings)
- Relational model fits the domain naturally

3.2.2 Query Requirements

- Need to query readings by building and time range
- Need to filter buildings by owner
- Relational SQL handles these queries efficiently

3.3 DATA INTEGRITY:

- Foreign key constraints ensure data consistency
- Referential integrity prevents orphaned records
- Transaction support for atomic operations

3.4 WHY JPA/HIBERNATE?

3.4.1 Spring Boot Integration

- Seamless integration with Spring Boot
- Automatic transaction management
- Repository pattern support

3.4.2 Productivity

- Reduces boilerplate code
- Automatic schema generation (ddl-auto: update)
- Type-safe queries with method names

3.4.3 Flexibility

- Can write custom queries when needed
- Easy to switch between databases
- Good for research/prototyping

3.5 SCHEMA DESIGN

3.5.1 Tables

3.5.1.1 users

sql

```
CREATE TABLE users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username VARCHAR(255) UNIQUE NOT NULL,
```

```
  role VARCHAR(50) NOT NULL  -- ADMIN or OWNER
);
```

3.5.1.2 buildings

sql

```
CREATE TABLE buildings (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name VARCHAR(255) NOT NULL,
  owner_id INTEGER NOT NULL,
  address VARCHAR(255),
  FOREIGN KEY (owner_id) REFERENCES users(id)
);
```

3.5.1.3 readings

sql

```
CREATE TABLE readings (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  building_id INTEGER NOT NULL,
  sensor_id VARCHAR(255) NOT NULL,
  timestamp TIMESTAMP NOT NULL,
  value REAL NOT NULL,
  FOREIGN KEY (building_id) REFERENCES buildings(id)
);


CREATE INDEX idx_building_timestamp ON readings(building_id, timestamp);
CREATE INDEX idx_sensor_timestamp ON readings(sensor_id, timestamp);
```

3.6 DESIGN DECISIONS

3.6.1 Normalization: Third normal form

- Buildings separate from users (many-to-one)
- Readings separate from buildings (many-to-one)
- Avoids data duplication

3.6.2 Indexes
- Index on `(building_id, timestamp)` for efficient time-range queries
- Index on `(sensor_id, timestamp)` for sensor-specific queries

3.6.3 Data Types
- `REAL` for sensor values (allows decimals)
- `TIMESTAMP` for timestamps (ISO 8601 format)
- `INTEGER` for IDs (auto-increment)

3.6.4 Constraints
- Foreign keys ensure referential integrity
- NOT NULL constraints prevent missing data
- UNIQUE on username prevents duplicate users

3.7 Consequences

3.7.1 Positive
- Simple Setup: No database server required
- Fast Development: Database auto-creates schema
- Easy Migration: SQLite schema can be migrated to PostgreSQL
- Type Safety: JPA provides compile-time type checking
- Query Efficiency: Indexes support fast time-range queries

3.7.2 Negative
- Concurrency: SQLite has limited concurrent write support
- Scalability: Not suitable for high-volume production use
- No Advanced Features: Missing some PostgreSQL features (full-text search, etc.)

3.8 Alternatives Considered

3.8.1 PostgreSQL
Considered but not chosen for initial implementation:

- Requires separate database server
- More complex setup for research slice
- Future: Can be switched by changing datasource URL

3.8.2 MongoDB (NoSQL)
Rejected because:

- Sensor readings have structured schema (better fit for SQL)
- Need complex queries (time ranges, joins)
- JPA/Hibernate don't work well with NoSQL
- Relational model fits the domain better

3.8.3 In-Memory Database (H2)

Considered for tests only:

- Used in integration tests (H2)
- Not suitable for production (data lost on restart)
- SQLite provides persistence while maintaining simplicity

3.8.4 Migration Path

If moving to PostgreSQL:

3.8.4.1 Schema Migration

yaml

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/decoder
    driver-class-name: org.postgresql.Driver
```

3.8.4.2 Dialect Change

yaml

```
hibernate:
  dialect: org.hibernate.dialect.PostgreSQLDialect
```

3.8.4.3 SQL Adjustments

- Change `INTEGER PRIMARY KEY AUTOINCREMENT` → `SERIAL PRIMARY KEY`
- Adjust timestamp handling if needed
- SQLite-specific syntax may need adjustment

3.9 DATA MIGRATION

- Export SQLite data
- Import into PostgreSQL
- Verify data integrity

3.10 TESTING STRATEGY

Unit Tests: Use in-memory H2 database

- Integration Tests: Use H2 for faster test execution
- Production: SQLite (or PostgreSQL for scale)