LegalDiscover Backend Architecture Documentation

Table of Contents

Overview

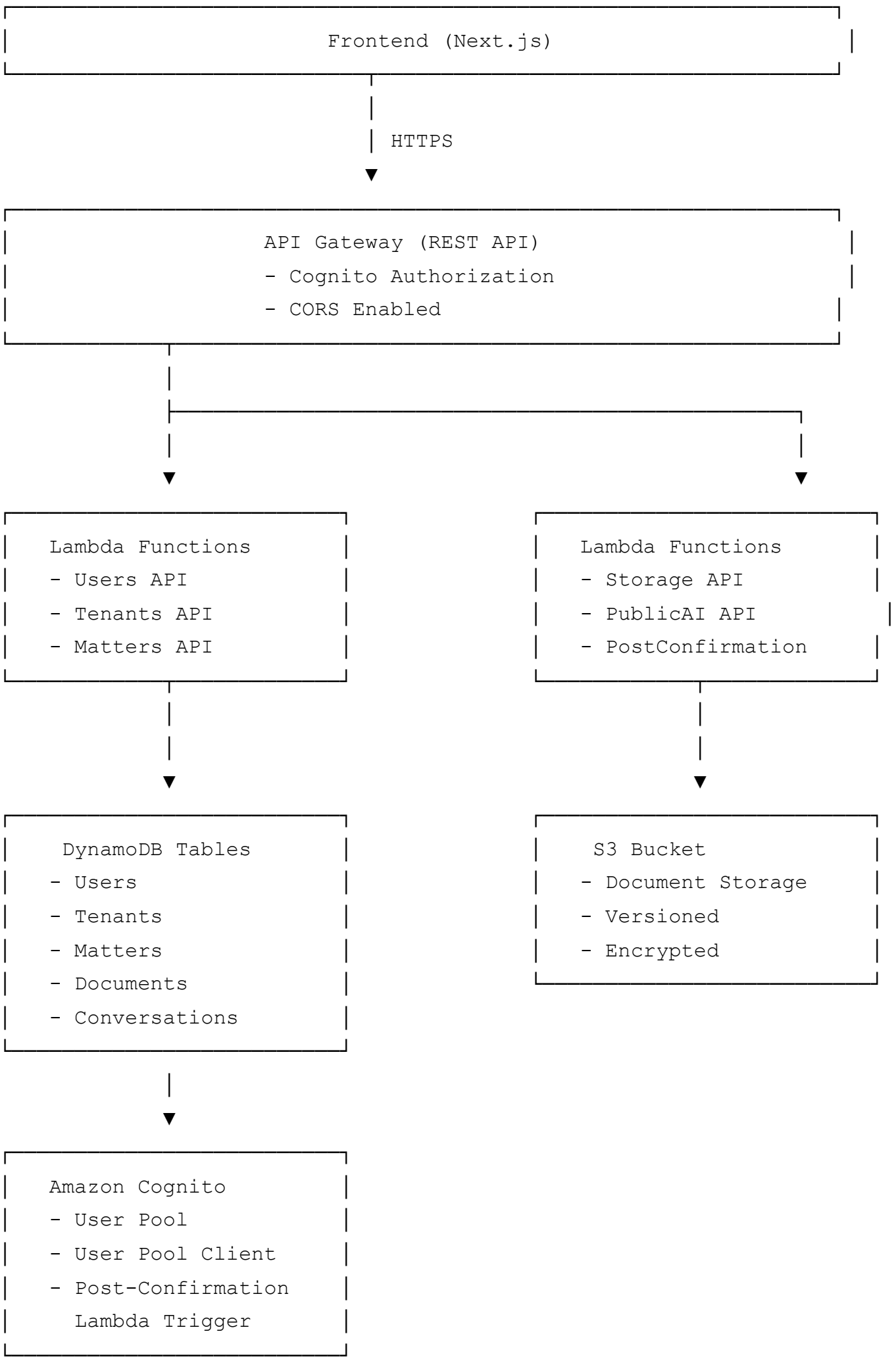
The LegalDiscover backend is a serverless application built on AWS using AWS CDK (Cloud Development Kit). It provides a multi-tenant legal case management system with document storage, user management, matter tracking, and AI-powered legal assistance.

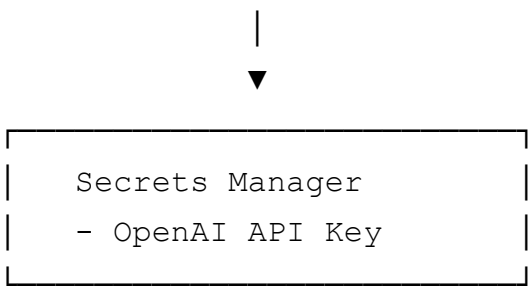
Key Technologies

- **Infrastructure as Code**: AWS CDK (TypeScript)
- **Compute**: AWS Lambda (Node.js 20/22)
- **Database**: Amazon DynamoDB
- **Storage**: Amazon S3
- **Authentication**: Amazon Cognito
- **API Gateway**: REST API
- **AI Integration**: OpenAI GPT-3.5 Turbo
- **Secrets Management**: AWS Secrets Manager

System Architecture

High-Level Architecture


Frontend (Next.js)
HTTPS
API Gateway (REST API)
- Cognito Authorization
- CORS Enabled
Lambda Functions
- Users API
- Tenants API
- Matters API
Lambda Functions
- Storage API
- PublicAI API
- PostConfirmation
DynamoDB Tables
- Users
- Tenants
- Matters
- Documents
- Conversations
S3 Bucket
- Document Storage
- Versioned
- Encrypted
Amazon Cognito
- User Pool
- User Pool Client
- Post-Confirmation
Lambda Trigger



Stack Components

1. Database Stack (`database-stack.ts`)

Purpose: Manages all DynamoDB tables for the application.

Tables:

- **Users Table**: `legaldiscover-users-{stage}`
  - Partition Key: `tenantId` (String)
  - Sort Key: `userId` (String)
  - Purpose: Stores user information per tenant
- **Tenants Table**: `legaldiscover-tenants-{stage}`
  - Partition Key: `tenantId` (String)
  - Purpose: Stores tenant/organization information
- **Matters Table**: `legaldiscover-matters-{stage}`
  - Partition Key: `tenantId` (String)
  - Sort Key: `matterId` (String)
  - Purpose: Stores legal matters/cases per tenant
- **Documents Table**: `legaldiscover-documents-{stage}`
  - Partition Key: `tenantId` (String)
  - Sort Key: `documentId` (String)
  - Purpose: Stores document metadata and S3 references
- **Conversations Table**: `legaldiscover-conversations-v2-{stage}`
  - Partition Key: `tenantId` (String)
  - Sort Key: `userId` (String) - Composite: `userId#messageId`
  - GSI: `conversationId-index`

        - Partition Key: `tenantId`
        - Sort Key: `conversationId`
    - Purpose: Stores AI conversation history

Configuration:

- Billing Mode: Pay-per-request (on-demand)
- Removal Policy: DESTROY (for dev environments)

2. Auth Stack (`auth-stack.ts`)

Purpose: Manages user authentication and authorization using Amazon Cognito.

Components:

- **User Pool**: `legal-discover-users-{stage}`
    - Self-signup enabled
    - Email-based sign-in
    - Auto-verify email
    - Standard attributes: email, fullname, address, phoneNumber
    - Custom attributes: company, tenantId
- **User Pool Client**: OAuth client for frontend
    - Implicit code grant flow
    - No client secret (public client)
    - Scopes: EMAIL, OPENID, PROFILE, COGNITO_ADMIN
- **User Groups**: admin, client, secretary, paralegal, associate, partner
- **Post-Confirmation Lambda**: Automatically creates user record in DynamoDB after signup
    - Triggered on user confirmation
    - Creates user entry in Users table

Hosted UI Domain: `legaldiscover-{stage}.auth.{region}.amazoncognito.com`

3. Storage Stack (`storage-stack.ts`)

Purpose: Manages document storage in S3 and document metadata tracking.

Components:

- **S3 Bucket**: `legaldiscover-storage-{stage}`

    - Block all public access
    - SSL enforced
    - Versioned
    - S3-managed encryption
    - CORS enabled for GET/PUT

- **Storage Lambda**: Handles document operations

    - **S3 Event Trigger**: Processes new document uploads
    - **REST API Integration**: Generates presigned URLs and queries documents
    - **Functions**:
        - Generate presigned upload URLs
        - Generate presigned download URLs
        - Query documents by tenant/client/matter
        - Auto-index documents on S3 upload

Document Path Structure:

```
tenants/{tenantId}/clients/{clientId}/matters/{matterId}/{fileName}
```

4. PublicAI Stack (`publicai-stack.ts`)

Purpose: Manages AI-powered legal assistance using OpenAI.

Components:

- **PublicAI Lambda**: Handles AI conversations

    - Retrieves OpenAI API key from Secrets Manager
    - Manages conversation history
    - Integrates with OpenAI GPT-3.5 Turbo
    - Stores conversations in DynamoDB

- **Secrets Manager**: Stores OpenAI API key securely

    - Secret name: `openai-key-{stage}`
    - Lambda has read permissions

- **WebSocket API**: (Configured but not fully utilized)

    - WebSocket API Gateway for real-time chat
    - Currently uses REST API primarily

Features:

- Conversation context management

- Legal-focused system prompt
- Message rollback on errors
- Conversation history retrieval

5. API Stack (`api-stack.ts`)

Purpose: Provides REST API endpoints and routes requests to appropriate Lambda functions.

Components:

- **REST API**: `LegalDiscoverApi-{stage}`
  - Cognito User Pool authorizer
  - CORS enabled for all origins
  - Default authorization on all methods
- **Lambda Integrations**:
  - **Users API**: `/tenants/{tenantId}/users`
  - **Tenants API**: `/tenants` and `/tenants/{tenantId}`
  - **Matters API**: `/tenants/{tenantId}/matters`
  - **Storage API**: `/tenants/{tenantId}/storage`
  - **PublicAI API**: `/tenants/{tenantId}/publicai`

Authorization: All endpoints require valid Cognito JWT token

Data Models

User Model

```
{
  tenantId: string;       // Partition key
  userId: string;         // Sort key (Cognito sub)
  name: string;
  email: string;
  role: string;           // admin, client, secretary, etc.
  address?: string;
  phone?: string;
  company?: string;
  createdAt: string;      // ISO timestamp
}
```

Tenant Model

```
{
  tenantId: string;      // Partition key
  name: string;
  domain: string;
  plan: string;
  status: string;
  settings: object;      // JSON stringified
  billing: object;       // JSON stringified
  compliance: object;    // JSON stringified
  createdAt: string;
  updatedAt: string;
}
```

Matter Model

```
{
  tenantId: string;           // Partition key
  matterId: string;           // Sort key
  title: string;
  status: string;
  clientId: string;
  clientName: string;
  caseType: string;
  priority: string;
  description: string;
  assignedAttorney: string;
  createdAt: string;
  updatedAt: string;
}
```

Document Model

```
{
  tenantId: string;      // Partition key
  documentId: string;    // Sort key (S3 key)
  clientId?: string;
  matterId?: string;
  fileName: string;
```

```
  s3Bucket: string;
  s3Key: string;
  createdAt: string;
}
```

Conversation Model

```
{
  tenantId: string;          // Partition key
  userId: string;            // Sort key (composite: userId#messageId)
  messageId: string;
  conversationId: string;
  role: "user" | "assistant";
  content: string;
  timestamp: string;
}
```

API Endpoints

Base URL

`https://{api-id}.execute-api.{region}.amazonaws.com/{stage}`

Authentication

All endpoints require `Authorization` header with Cognito JWT token:

`Authorization: Bearer {jwt-token}`

Endpoints

Users

- `POST /tenants/{tenantId}/users` - Create user
- `GET /tenants/{tenantId}/users/{id}` - Get user
- `PUT /tenants/{tenantId}/users/{id}` - Update user
- `DELETE /tenants/{tenantId}/users/{id}` - Delete user

Tenants

- `POST /tenants` - Create tenant
- `GET /tenants/{tenantId}` - Get tenant
- `PUT /tenants/{tenantId}` - Update tenant
- `DELETE /tenants/{tenantId}` - Delete tenant

Matters

- `GET /tenants/{tenantId}/matters` - List all matters
- `POST /tenants/{tenantId}/matters` - Create matter
- `GET /tenants/{tenantId}/matters/{id}` - Get matter
- `PUT /tenants/{tenantId}/matters/{id}` - Update matter
- `DELETE /tenants/{tenantId}/matters/{id}` - Delete matter

Storage

- `POST /tenants/{tenantId}/storage` - Generate presigned URL
  - Body: `{ fileName, clientId, matterId, type: "upload" | "download" }`
- `GET /tenants/{tenantId}/storage?clientId={id}&matterId={id}` - Query documents

PublicAI

- `POST /tenants/{tenantId}/publicai/send` - Send message to AI
  - Body: `{ message, userId, conversationId? }`
- `GET /tenants/{tenantId}/publicai/conversations?userId={id}` - Get conversation history

Data Flow

User Registration Flow

1. User signs up via Cognito Hosted UI
2. Cognito sends confirmation email
3. User confirms email
4. Post-Confirmation Lambda triggered
5. Lambda creates user record in DynamoDB Users table
6. User can now authenticate and access API

Document Upload Flow

1. Client requests presigned upload URL from Storage API
2. Storage Lambda generates presigned PUT URL (1 hour expiry)
3. Client uploads file directly to S3 using presigned URL
4. S3 triggers Storage Lambda on object creation
5. Storage Lambda extracts metadata and creates document record in DynamoDB
6. Document is now queryable via Storage API

Document Download Flow

1. Client queries documents via Storage API (GET)
2. Storage Lambda queries DynamoDB Documents table
3. Returns document metadata list
4. Client requests presigned download URL for specific document
5. Storage Lambda generates presigned GET URL (1 hour expiry)
6. Client downloads file directly from S3

AI Conversation Flow

1. Client sends message to PublicAI API
2. Lambda retrieves OpenAI API key from Secrets Manager
3. Lambda saves user message to Conversations table
4. Lambda retrieves conversation history from DynamoDB
5. Lambda calls OpenAI API with system prompt + history
6. Lambda saves assistant response to Conversations table
7. Lambda returns response to client
8. On error, Lambda rolls back user message

Matter Management Flow

1. Client creates matter via Matters API
2. Matter Lambda generates UUID for matterId
3. Lambda saves matter to DynamoDB Matters table
4. Client can query, update, or delete matters
5. All operations are tenant-scoped

Security

Authentication & Authorization

- **Cognito User Pools**: All API endpoints require valid JWT token
- **Multi-tenant Isolation**: All operations are tenant-scoped
- **User Groups**: Role-based access control (admin, client, etc.)

Data Security

- **S3**: Private bucket with encryption at rest
- **DynamoDB**: Encrypted at rest (AWS managed)
- **Secrets Manager**: OpenAI API key stored securely
- **HTTPS Only**: All API communication over TLS

Network Security

- **CORS**: Configured for frontend domain
- **API Gateway**: Rate limiting (commented out, can be enabled)
- **VPC**: Not used (serverless architecture)

Deployment

Prerequisites

- Node.js 20+
- AWS CLI configured
- AWS CDK CLI installed
- AWS account with appropriate permissions

Build & Deploy

```
cd backend
npm install
npm run build
cdk deploy --all --context stage=dev
```

Environment Variables

- `stage` : Deployment stage (dev, staging, prod)
- `region` : AWS region (default: us-east-1)

Stack Dependencies

1. Database Stack (no dependencies)
2. Auth Stack (depends on Database Stack)
3. Storage Stack (depends on Database Stack)
4. PublicAI Stack (depends on Database Stack)
5. API Stack (depends on all other stacks)

Post-Deployment

1. Configure OpenAI API key in Secrets Manager:

```
aws secretsmanager put-secret-value \
  --secret-id openai-key-dev \
  --secret-string "your-openai-api-key"
```

2. Update Cognito callback URLs for production
3. Configure CORS origins in API Gateway for production domain

Monitoring & Logging

CloudWatch Logs

- All Lambda functions log to CloudWatch
- Log groups: `/aws/lambda/{function-name}`

Metrics

- Lambda invocations, errors, duration
- DynamoDB read/write capacity
- API Gateway request counts, latency
- S3 request metrics

Error Handling

- All Lambda functions return standardized error responses
- Errors logged to CloudWatch
- User-friendly error messages returned to clients

Future Enhancements

1. **WebSocket Support**: Full WebSocket API for real-time AI chat
2. **Rate Limiting**: Enable API Gateway rate limiting
3. **Caching**: Add CloudFront for API responses
4. **Monitoring**: Enhanced CloudWatch dashboards
5. **Testing**: Comprehensive unit and integration tests
6. **Document Processing**: Add document parsing and indexing
7. **Search**: Full-text search for documents and matters
8. **Notifications**: SNS/SES for email notifications