Property management software architecture deep dive

Four leading platforms reveal distinct but converging patterns for RBAC, multi-tenancy, and data modeling. AppFolio uses Ruby/Rails with microservices; Buildium and Yardi are .NET shops; TenantCloud uniquely offers GraphQL. All share similar entity hierarchies and permission structures you can adapt to your NextJS/Drizzle/Postgres/Better Auth stack.

The most actionable finding: **TenantCloud's GraphQL API documentation** provides the clearest public schema, while **Buildium's OpenAPI spec** offers the most complete REST entity model. For RBAC patterns, AppFolio's portal-based isolation and Yardi's 5,000+ granular permissions show opposite ends of the complexity spectrum.

AppFolio: Ruby on Rails with microservices orchestration

AppFolio is a **Ruby on Rails monolith** with a **React/Redux frontend**, deployed on AWS with MySQL. Their integration layer (Stack API) operates as a microservices orchestration layer with contract testing via Pact.

Technology stack confirmed

Layer	Technology	Confidence
Backend	Ruby on Rails	HIGH (engineering blog, job posts)
Frontend	React, Redux	HIGH (job postings)
Database	MySQL	MEDIUM (StackShare)
Cloud	AWS	HIGH (engineering blog)
API	REST with OpenAPI, some GraphQL	HIGH
Auth	OAuth 2.0, JWS webhooks	HIGH (official docs)

RBAC architecture pattern

AppFolio implements **portal-based isolation** rather than granular permissions. Each user type gets a completely separate authentication context:

```
Property Manager Portal → Full administrative access
Owner Portal            → Read-only financial access to owned properties
Tenant Portal           → Payments, maintenance requests, lease viewing
Vendor Portal           → Work order management only
```

The Ledger Labs

Tenant invitation flow: Landlords send email invitations through the system. Tenants self-register, creating accounts linked to their lease/occupancy records. AppFolio All portals require **two-factor authentication** (SMS, phone, or authenticator app) with 30-day device trust. AppFolio

For Better Auth adaptation: Model this as separate "portal" scopes in your JWT claims. Rather than complex permission matrices, consider whether distinct authentication contexts for each user type simplifies your architecture.

Data model from Stack API

The entity hierarchy revealed through their API endpoints:

```
Properties
  ├── Units
  │     ├── Occupancies (tenant-to-unit join table with lease data)
  │     │     └── Recurring Charges
  │     └── Listings
  └── Property Groups (for portfolio organization)

Property Manager Users (staff accounts)
Owners → Owner Groups
Tenants → Tenant Ledgers (financial records)
Vendors → Work Orders

Financial entities:
Bank Accounts → GL Accounts → GL Details (transactions)
Bills (with Attachments) → Charges → Delinquent Charges
```

Key insight for Drizzle schema: The Occupancy entity is their core join table connecting tenants, units, and financial data. This pattern—a first-class "occupancy" or "residency" entity rather than a simple tenant-unit foreign key—enables tracking move-in/move-out dates, multiple tenants per unit, and historical lease data.

Webhook signature verification

AppFolio uses **JWS (JSON Web Signature) with detached payload**, algorithm RSASSA_PSS_SHA_256. Public keys are available at https://api.appfolio.com/.well-known/jwks.json: GitHub

```json
{
  "keys": [{
    "alg": "PS256",
    "kty": "RSA",
    "use": "sig",
    "kid": "...",
    "n": "...",
    "e": "AQAB"
  }]
}
```

Webhook payload structure:

```json
{
  "client_id": "example_id",
  "topic": "work_order_updates",
  "entity_id": "uuid",
  "update_timestamp": "2023-03-27T16:55:12Z"
}
```

Rate limiting is structured per customer-partner pair—important multi-tenancy detail showing they isolate API quotas at the organization level.

Buildium: .NET with comprehensive OpenAPI specification

Buildium (owned by RealPage) runs on **.NET/C#** with what's likely SQL Server. Their **OpenAPI specification** is the most complete public documentation found, with auto-generated SDKs available.

Technology stack confirmed

Layer	Technology	Confidence
Backend	.NET (C#), ASP.NET	HIGH (Glassdoor reviews)
Database	SQL Server (inferred)	MEDIUM
API	REST v1 with OpenAPI/Swagger	HIGH (official docs)

Layer	Technology	Confidence
Auth	API Key pairs (client_id + secret)	HIGH
Sandbox	Dedicated sandbox environment	HIGH

RBAC and permission granularity

Buildium's permission model operates at **three distinct levels**:

1. Account-level isolation (multi-tenancy)

```
Each Property Management Company = Separate Account (AccountId)
  └── Complete data isolation
  └── Separate API keys
  └── Separate sandbox
```

2. User roles within accounts

```
Administrator   → Full access, API key management, user management
Staff Members   → Configurable via "User Roles" feature
Rental Owners   → Portal access to their properties only
Vendors         → Work order access for assigned jobs only
Tenants         → Resident Center portal
```

3. API key permission scoping

```
Per API key, you can restrict:
  - Specific Buildium entities (properties, tenants, etc.)
  - Read-only mode (GET resources only)
  - Granular data access checkboxes
```

For Better Auth: This three-tier model translates well. Use Better Auth's organization feature for account isolation, roles for the five user types, and fine-grained permissions on sessions/tokens for API access scoping.

Complete entity relationship model

From their 200+ API endpoints, the full hierarchy:

```
Organization/Account
├── Rental Properties
│   ├── Rental Units
│   │   ├── Leases
│   │   │   ├── Lease Tenants (many-to-many)
│   │   │   ├── Lease Transactions
│   │   │   │   └── Types: Charge, Payment, Credit, Refund
│   │   │   ├── Rent Schedules
│   │   │   └── Lease Renewals
│   │   └── Listings
│   ├── Rental Owners (can own multiple properties)
│   └── Appliances (tracked per property)
│
├── Associations (HOA/Condo management)
│   ├── Association Units
│   │   └── Ownership Accounts
│   │       ├── Association Owners
│   │       └── Ownership Account Transactions
│   ├── Association Tenants
│   └── Board Members
│
├── Maintenance
│   ├── Tasks (Contact Requests, Resident Requests, Owner Requests, To-Do)
│   ├── Work Orders
│   └── Vendors
│
└── Accounting
    ├── Bank Accounts
    │   ├── Checks, Deposits, Withdrawals, Transfers
    │   └── Reconciliations
    ├── Bills & Bill Payments
    ├── Budgets
    └── General Ledger
        ├── GL Accounts (with ParentGLAccountId for hierarchy)
        ├── Transactions
        └── Journal Entries
```

Key entity fields for Drizzle schema design

Lease entity (central to the system):

typescript

```
// Drizzle schema equivalent
const leases = pgTable('leases', {
  id: serial('id').primaryKey(),
  unitId: integer('unit_id').references(() => units.id),
  leaseType: text('lease_type'), // enum
  leaseStatus: text('lease_status'),
  rentAmount: numeric('rent_amount'),
  startDate: date('start_date'),
  endDate: date('end_date'),
  lastUpdatedAt: timestamp('last_updated_at'),
});

const leaseTenants = pgTable('lease_tenants', {
  id: serial('id').primaryKey(),
  leaseId: integer('lease_id').references(() => leases.id),
  tenantId: integer('tenant_id').references(() => tenants.id),
  status: text('status'),
  moveInDate: date('move_in_date'),
});
```

GL Account with hierarchical structure:

```
typescript

const glAccounts = pgTable('gl_accounts', {
  id: serial('id').primaryKey(),
  parentGlAccountId: integer('parent_gl_account_id').references(() => glAccounts.id),
  defaultAccountName: text('default_account_name'),
  isBankAccount: boolean('is_bank_account'),
});
```

Webhook implementation pattern

Buildium uses **HMAC-SHA256** for webhook signatures:

```
Headers:
  buildium-webhook-timestamp: UNIX timestamp
  buildium-webhook-signature: Base64(HMAC-SHA256(timestamp + "." + minified_json_body))
```

Retry strategy with exponential backoff:

- 1st retry: 1 minute

- 2nd retry: 10 minutes
- 3rd retry: 1 hour
- Suspension after 20 consecutive failures

Rate limit: 10 concurrent requests/second, returns 429 with ~200ms recommended retry.

Yardi Voyager: Enterprise .NET with SOAP and granular permissions

Yardi is the enterprise incumbent, running **.NET Core/Angular** with **SQL Server**. Their architecture is notably different: a monolithic ERP with **5,000+ granular permissions** and primarily **SOAP/XML APIs** (though newer RentCafe modules use REST).

Technology stack confirmed

Layer	Technology	Confidence
Backend	C#, .NET Core, ASP.NET	HIGH (Glassdoor interviews)
Frontend	Angular, HTML5/jQuery	HIGH (job postings)
Database	Microsoft SQL Server	HIGH (job postings)
API	SOAP/WSDL (primary), REST (RentCafe)	HIGH
Cloud	Yardi private cloud, SaaS	HIGH

RBAC: The 5,000-permission model

Yardi represents the **maximum complexity** approach to RBAC:

```
User
├── Security Group(s)
│   ├── Menu Permissions (5,000+ options)
│   ├── Button Control Access
│   ├── Charge Code/GL Security
│   └── Workflow Approvals
├── Property List Access (row-level security)
└── Role-Based Dashboard Assignment
```

Permission granularity includes:

- Browse, Read, Edit, Add, Delete per module
- Posting rights (accounting)
- Bank access levels
- Check/ACH processing permissions
- Delete access (separate from edit) Saxony Partners

Role archetypes (not fixed roles, but permission templates):

Role	Description
Property Manager	Full operational access to assigned properties
Leasing Agent	RentCafe CRM access, lead/application management
Maintenance User	Service requests only, no financial access
Accountant	GL, AR, AP posting and reporting
Admin	System configuration, user management
Investor/Owner	Portal-only access for reports and statements

Property-based isolation: "If you restrict a user to one [property list], the user can only see data linked to that property." Yardi Breeze This is implemented via Dynamic Property Lists for grouping assets by region, ownership, or function.

Multi-tenancy architecture

Yardi uses **single-database-per-client** rather than shared multi-tenant schemas:

> Options:
> 1. Single Global Database - Centralized data, standardized processes
> 2. Multiple Regional Databases - Localized compliance, regional workflows

33Floors

Each client gets dedicated database instance with property-level row isolation within that database. This is the opposite of true SaaS multi-tenancy but offers stronger data isolation for enterprise clients.

For Neon Postgres: You likely want the multi-tenant shared schema approach with organization_id columns and row-level security policies, not Yardi's pattern. But their property-list concept for scoping user access is worth adopting.

API architecture (SOAP + REST hybrid)

SOAP endpoints (legacy, but primary):

```
URL: https://www.yardiasp[N].com/{CLIENT}/webservices/{Interface}.asmx?WSDL

Key interfaces:
- ItfResidentTransactions20 (billing, payments)
- ItfCommercialAPI (property, lease, tenant data)
- ItfILSGuestCard (unit availability, leads)
- ItfServiceRequest (maintenance work orders)
- ItfVendorInvoicing (AP workflows)
```

RentCafe REST API (newer):

```
Base: https://api.rentcafe.com/rentcafeapi.aspx
Auth: Vendor API Token + Company Code

Endpoints:
- getapartmentavailability
- getfloorplans
- createlead (Premium)
- getpropertydetails
- getunitpricingdetails
```

Data model (inferred from integrations)

```
Portfolio/Database
└── Property (PropertyCode)
    ├── Unit (UnitCode, ApartmentId)
    │     ├── Unit Type / Floorplan
    │     ├── Amenities
    │     └── Listings
    ├── Lease
    │     ├── Tenant/Resident
    │     ├── Rent Schedule
    │     ├── Escalations (rent increases)
    │     ├── Options/Clauses
    │     └── Transactions
    ├── Service Requests (Work Orders)
    └── Common Areas
```

Key identifiers: `VoyagerPropertyCode`, `ApartmentId` (RentCafe), `IDType/IDValue` pairs for flexible identification. `Unit Map`

TenantCloud: GraphQL API with Laravel/PHP backend

TenantCloud stands out with a **publicly documented GraphQL API**—the only platform in this research offering GraphQL. Their backend is **PHP (Laravel)** with some .NET components and SQL Server.

Technology stack confirmed

Layer	Technology	Confidence
Backend	PHP (Laravel), some .NET	HIGH (GitHub, job posts)
Database	SQL Server	HIGH (job posting)
API	**GraphQL**	HIGH (official docs)
Endpoint	`https://api.tenantcloud.com/graphql`	HIGH
Auth	Personal Access Tokens (Bearer)	HIGH
Architecture	Microservices + SPA	HIGH (job posting)

GraphQL API documentation

TenantCloud provides full API documentation at `docs.api.tenantcloud.com` with:

- Interactive GraphQL explorer at `/graphql/explore`
- Scoped Personal Access Tokens (PAT) with configurable permissions
- Webhook support for real-time events
- Built-in pagination and field metadata

Sample authentication:

```bash
curl -X POST https://api.tenantcloud.com/graphql \
  -H "Authorization: Bearer {PAT_TOKEN}" \
  -H "Content-Type: application/json" \
  -d '{"query": "{ me { id } }"}'
```

For NextJS: This GraphQL pattern maps directly to your stack. Consider using similar PAT-based auth with Better Auth's token features, and expose a GraphQL endpoint via Apollo Server or GraphQL Yoga in your NextJS API routes.

RBAC with account modes

TenantCloud uniquely offers **two distinct account modes**:

Landlord Mode (DIY landlords):

- Simplified interface
- Owner automatically assigned to all properties
- No management fee tracking

Property Manager Mode (PM companies):

- Requires owner assignment per property
- Management fee tracking and owner distributions
- Team member management with property-level permissions

Role hierarchy:

```
Main Admin
  └── Full account access, subscription management, team management

Team Members (Sub-Admins)
  └── Configurable permissions per property
  └── Granular view/manage per section:
    - Accounting, Maintenance, Listings, Applications, Leases

Property Owners
  └── Separate owner portal
  └── Access limited to their properties

Tenants
  └── Separate tenant portal
  └── Rent payment, maintenance requests, documents
```

For Better Auth: The account modes pattern is interesting—you could implement this as a boolean flag or enum on the organization that changes available features and required fields.

Data model from unofficial API client

An unofficial .NET client (github.com/yllibed/TenantCloudClient) reveals the entity structure:

```typescript
```

```
// Drizzle schema translation
const properties = pgTable('properties', {
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id),
  ownerId: integer('owner_id').references(() => users.id), // in PM mode
  // ... property details
});

const units = pgTable('units', {
  id: serial('id').primaryKey(),
  propertyId: integer('property_id').references(() => properties.id),
  status: text('status'), // 'Occupied' | 'Vacant'
});

const tenants = pgTable('tenants', {
  id: serial('id').primaryKey(),
  name: text('name'),
  status: text('status'), // 'MovedIn' | 'NoLease' | 'Archived'
  validEmails: jsonb('valid_emails'), // array
  validPhones: jsonb('valid_phones'), // array
});

const leases = pgTable('leases', {
  id: serial('id').primaryKey(),
  tenantId: integer('tenant_id').references(() => tenants.id),
  unitId: integer('unit_id').references(() => units.id),
  isActive: boolean('is_active'),
  expirationDate: date('expiration_date'),
});

const transactions = pgTable('transactions', {
  id: serial('id').primaryKey(),
  tenantId: integer('tenant_id').references(() => tenants.id),
  propertyId: integer('property_id').references(() => properties.id),
  unitId: integer('unit_id').references(() => units.id), // optional
  amount: numeric('amount'),
  balance: numeric('balance'),
  category: text('category'), // 'Income' | 'Expense' | 'Refund' | 'Credits' | 'Liability'
  status: text('status'), // 'Due' | 'Paid' | 'Partial' | 'Pending' | 'Void' | 'WithBalance' | 'Overdue' | 'Waive'
});
```

Open source Laravel packages

TenantCloud maintains **62 public repositories** on GitHub with reusable Laravel packages:

Package	Purpose
`laravel-graphql-platform`	Platform for GraphQL APIs with Laravel
`laravel-boolean-softdeletes`	Optimized soft deletes using boolean indexing (performance pattern)
`laravel-better-cache`	Redis cache tags implementation
`php-data-transfer-objects`	DTO pattern implementation

The **boolean soft-deletes pattern** is notable: instead of `deleted_at` timestamp columns that are hard to index, they use a boolean `is_deleted` column for high-load query optimization.

Cross-platform patterns for your stack

Recommended RBAC model for NextJS/Better Auth

Synthesizing all four platforms, here's a recommended approach:

```typescript
```

```
// Better Auth organization + role structure
const organizations = pgTable('organizations', {
  id: serial('id').primaryKey(),
  name: text('name').notNull(),
  accountMode: text('account_mode').default('landlord'), // 'landlord' | 'property_manager'
  createdAt: timestamp('created_at').defaultNow(),
});

const organizationMembers = pgTable('organization_members', {
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id),
  userId: integer('user_id').references(() => users.id),
  role: text('role').notNull(), // 'admin' | 'staff' | 'owner' | 'tenant' | 'vendor'
  permissions: jsonb('permissions'), // granular permissions object
  propertyScope: jsonb('property_scope'), // array of property IDs or 'all'
});

// Portal-based access (AppFolio pattern)
const portalSessions = pgTable('portal_sessions', {
  id: serial('id').primaryKey(),
  userId: integer('user_id').references(() => users.id),
  portalType: text('portal_type'), // 'manager' | 'owner' | 'tenant' | 'vendor'
  organizationId: integer('organization_id').references(() => organizations.id),
  // Each portal type gets different JWT claims/scopes
});
```

Recommended core entity schema

typescript

```
// Core property management entities for Drizzle
const properties = pgTable('properties', {
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id).notNull(),
  ownerId: integer('owner_id').references(() => users.id), // for PM mode
  name: text('name'),
  propertyType: text('property_type'), // 'single_family' | 'multi_family' | 'commercial' | 'hoa'
  address: jsonb('address'), // structured address object
  tenantPortalEnabled: boolean('tenant_portal_enabled').default(true),
  createdAt: timestamp('created_at').defaultNow(),
  updatedAt: timestamp('updated_at'),
  deletedAt: timestamp('deleted_at'), // soft delete
});

const units = pgTable('units', {
  id: serial('id').primaryKey(),
  propertyId: integer('property_id').references(() => properties.id).notNull(),
  unitNumber: text('unit_number'),
  bedrooms: integer('bedrooms'),
  bathrooms: numeric('bathrooms'),
  squareFeet: integer('square_feet'),
  marketRent: numeric('market_rent'),
  status: text('status').default('vacant'), // 'occupied' | 'vacant' | 'maintenance'
  amenities: jsonb('amenities'),
  createdAt: timestamp('created_at').defaultNow(),
});

// The critical join entity (AppFolio's "Occupancy" pattern)
const occupancies = pgTable('occupancies', {
  id: serial('id').primaryKey(),
  unitId: integer('unit_id').references(() => units.id).notNull(),
  leaseId: integer('lease_id').references(() => leases.id),
  tenantId: integer('tenant_id').references(() => tenants.id).notNull(),
  moveInDate: date('move_in_date'),
  moveOutDate: date('move_out_date'),
  status: text('status'), // 'current' | 'past' | 'future'
  isPrimaryTenant: boolean('is_primary_tenant').default(false),
});

const leases = pgTable('leases', {
  id: serial('id').primaryKey(),
  unitId: integer('unit_id').references(() => units.id).notNull(),
  leaseType: text('lease_type'), // 'fixed' | 'month_to_month'
```

```
  status: text('status'), // 'draft' | 'active' | 'expired' | 'terminated'
  startDate: date('start_date').notNull(),
  endDate: date('end_date'),
  rentAmount: numeric('rent_amount').notNull(),
  securityDeposit: numeric('security_deposit'),
  paymentDueDay: integer('payment_due_day').default(1),
  lateFeeAmount: numeric('late_fee_amount'),
  lateFeeGraceDays: integer('late_fee_grace_days'),
  createdAt: timestamp('created_at').defaultNow(),
});

const tenants = pgTable('tenants', {
  id: serial('id').primaryKey(),
  userId: integer('user_id').references(() => users.id), // linked Better Auth user
  organizationId: integer('organization_id').references(() => organizations.id),
  firstName: text('first_name'),
  lastName: text('last_name'),
  email: text('email'),
  phone: text('phone'),
  status: text('status'), // 'active' | 'past' | 'applicant'
  emergencyContact: jsonb('emergency_contact'),
  createdAt: timestamp('created_at').defaultNow(),
});

// Financial transactions (Buildium pattern)
const transactions = pgTable('transactions', {
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id),
  leaseId: integer('lease_id').references(() => leases.id),
  tenantId: integer('tenant_id').references(() => tenants.id),
  unitId: integer('unit_id').references(() => units.id),
  transactionType: text('transaction_type'), // 'charge' | 'payment' | 'credit' | 'refund'
  category: text('category'), // 'rent' | 'deposit' | 'late_fee' | 'utility' | etc.
  amount: numeric('amount').notNull(),
  dueDate: date('due_date'),
  paidDate: date('paid_date'),
  status: text('status'), // 'pending' | 'paid' | 'partial' | 'overdue' | 'void'
  glAccountId: integer('gl_account_id').references(() => glAccounts.id),
  notes: text('notes'),
  createdAt: timestamp('created_at').defaultNow(),
});

// GL accounts with hierarchy (Buildium pattern)
const glAccounts = pgTable('gl_accounts', {
```

```
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id),
  parentId: integer('parent_id').references(() => glAccounts.id),
  accountNumber: text('account_number'),
  name: text('name').notNull(),
  accountType: text('account_type'), // 'asset' | 'liability' | 'equity' | 'income' | 'expense'
  isBankAccount: boolean('is_bank_account').default(false),
  isSystemAccount: boolean('is_system_account').default(false),
});

// Maintenance (all platforms)
const maintenanceRequests = pgTable('maintenance_requests', {
  id: serial('id').primaryKey(),
  organizationId: integer('organization_id').references(() => organizations.id),
  propertyId: integer('property_id').references(() => properties.id),
  unitId: integer('unit_id').references(() => units.id),
  tenantId: integer('tenant_id').references(() => tenants.id),
  vendorId: integer('vendor_id').references(() => vendors.id),
  requestType: text('request_type'), // 'tenant_request' | 'owner_request' | 'internal'
  priority: text('priority'), // 'emergency' | 'high' | 'normal' | 'low'
  status: text('status'), // 'received' | 'scheduled' | 'in_progress' | 'completed' | 'cancelled'
  title: text('title'),
  description: text('description'),
  scheduledDate: timestamp('scheduled_date'),
  completedDate: timestamp('completed_date'),
  cost: numeric('cost'),
  createdAt: timestamp('created_at').defaultNow(),
});
```

Webhook implementation recommendation

Use **HMAC-SHA256** (Buildium pattern) for simplicity, or **JWS with JWKS** (AppFolio pattern) for stronger security:

typescript

```
// HMAC approach (simpler)
import crypto from 'crypto';

function verifyWebhookSignature(
  payload: string,
  timestamp: string,
  signature: string,
  secret: string
): boolean {
  const signedPayload = `${timestamp}.${payload}`;
  const expectedSig = crypto
    .createHmac('sha256', secret)
    .update(signedPayload)
    .digest('base64');
  return crypto.timingSafeEqual(
    Buffer.from(signature),
    Buffer.from(expectedSig)
  );
}
```

Open source references discovered

Two production-ready open source alternatives for additional patterns:

Condo (github.com/open-condo-software/condo) - 15,549 commits, Node.js/PostgreSQL stack with ticket management, resident contacts, payments, and mini-app extension system.

MicroRealEstate (github.com/microrealestate/microrealestate) - MIT license, JavaScript/MongoDB, Docker deployment with separate landlord/tenant portals.

Conclusion

The four platforms converge on similar entity hierarchies despite different tech stacks. **AppFolio's occupancy-based tenant-unit modeling** and **Buildium's hierarchical GL accounts** are the most transferable patterns for your Drizzle schema. For RBAC, **TenantCloud's account modes** (landlord vs. property manager) combined with **Buildium's three-tier permission model** (organization → role → API scope) offers a practical middle ground between simplicity and enterprise granularity.

The key architectural decision for your stack: implement **organization-based multi-tenancy with row-level security** in Neon Postgres (all platforms do this), then layer **portal-based authentication contexts** (AppFolio pattern) using Better Auth's session management to simplify permission checking at runtime.