

BudStack Multi-Tenant Architecture

Technical Architecture Documentation

Table of Contents

1. System Overview
 2. Architecture Layers
 3. Database Design
 4. Authentication & Authorization
 5. Multi-Tenant Routing
 6. API Integration
 7. File Storage
 8. Scalability & Performance
-

System Overview

Platform Purpose

BudStack is a multi-tenant SaaS platform that enables medical cannabis dispensaries to launch their own branded online stores with minimal setup time.

Key Features

- **Multi-Tenant Architecture:** Single codebase serves multiple tenant stores
- **Subdomain Routing:** Each tenant gets their own subdomain (e.g., healingbuds.budstack.to)
- **Custom Domains:** Support for tenant custom domains
- **NFT-Based Licensing:** Blockchain-backed tenant licenses
- **Centralized Product Catalog:** Integration with Doctor Green API
- **White-Label Theming:** Each tenant fully customizes their branding
- **Role-Based Access:** Super Admin, Tenant Admin, End User roles

Technology Stack

Frontend - Next.js 14 (App Router) - React 18 - TypeScript - Tailwind CSS - shadcn/ui components

Backend - Next.js API Routes - Prisma ORM - PostgreSQL database - NextAuth.js for authentication

Infrastructure - Cloud storage (AWS S3 or compatible) - Email service (SMTP) - DNS management (Namecheap API)

Third-Party Integrations - Doctor Green API (product catalog) - Payment processors (region-dependent) - Email services

Architecture Layers

3-Tier Architecture

Presentation Layer
(Next.js Pages, React Components)

↓

Application Layer
(API Routes, Business Logic)

↓

Data Layer
(PostgreSQL, Prisma, S3 Storage)

Multi-Tenant Request Flow

User Request → DNS Resolution → Middleware → Tenant Identification

↓

Tenant Context Set → Database Query (filtered by tenant) → Response

Example Flow: 1. User visits `healingbuds.budstack.to` 2. DNS resolves to platform server 3. Middleware extracts subdomain: `healingbuds` 4. Database query: `SELECT * FROM tenants WHERE slug = 'healingbuds'` 5. Tenant context stored in request 6. All subsequent queries filtered by tenant ID 7. Tenant-specific theme applied 8. Response sent to user

Database Design

Schema Overview

Core Entities: - **Tenant:** Stores tenant (dispensary) information - **User:** All user accounts (admins, customers) - **Order:** Customer orders - **OrderItem:** Individual products in orders - **TenantBranding:** Theme and customization settings

Tenant Model

```
model Tenant {
  id          String    @id @default(cuid())
  businessName String
  slug        String    @unique
  subdomain   String?   @unique
  customDomain String?  @unique
  nftId       String?   @unique
  status      TenantStatus @default(ACTIVE)

  // API Configuration
  doctorGreenApiKey String?
  doctorGreenApiUrl  String?

  // Relationships
  users      User[]
  orders     Order[]
  branding   TenantBranding?

  createdAt  DateTime @default(now())
  updatedAt  DateTime @updatedAt
}

enum TenantStatus {
```

```

    ACTIVE
    PENDING
    SUSPENDED
    INACTIVE
}

```

User Model

```

model User {
    id          String    @id @default(cuid())
    email       String    @unique
    name        String?
    password    String
    role        UserRole   @default(USER)

    // Multi-tenant relationship
    tenantId    String?
    tenant      Tenant?    @relation(fields: [tenantId], references: [id])

    // Customer data
    address     String?
    phone       String?
    emailVerified DateTime?

    orders      Order[]

    createdAt   DateTime   @default(now())
    updatedAt   DateTime   @updatedAt
}

enum UserRole {
    SUPER_ADMIN // Platform administrator
    TENANT_ADMIN // Dispensary owner/manager
    USER        // End customer
}

```

Order Model

```

model Order {
    id          String    @id @default(cuid())
    orderNumber String    @unique

    // Multi-tenant
    tenantId    String
    tenant      Tenant    @relation(fields: [tenantId], references: [id])

    // Customer
    userId      String
    user        User      @relation(fields: [userId], references: [id])

    // Order details
    status      OrderStatus @default(PENDING)
    total       Float
    items       OrderItem[]
}

```

```

// Shipping
shippingAddress String
shippingMethod String?
trackingNumber String?

createdAt      DateTime      @default(now())
updatedAt      DateTime      @updatedAt
}

enum OrderStatus {
    PENDING
    CONFIRMED
    PROCESSING
    SHIPPED
    DELIVERED
    CANCELLED
}

model OrderItem {
    id          String  @id @default(cuid())
    orderId     String
    order       Order   @relation(fields: [orderId], references: [id])

    productId   String
    productName String
    quantity    Int
    price       Float

    createdAt   DateTime @default(now())
}

```

TenantBranding Model

```

model TenantBranding {
    id          String  @id @default(cuid())
    tenantId    String  @unique
    tenant      Tenant  @relation(fields: [tenantId], references: [id])

    // Design
    logoUrl     String?
    faviconUrl   String?
    heroImage    String?
    heroHeadline String?
    heroSubheadline String?
    ctaButtonText String?

    // Colors
    primaryColor String  @default("#16a34a")
    secondaryColor String  @default("#15803d")
    backgroundColor String  @default("#ffffff")
    textColor     String  @default("#1f2937")
    headingColor  String  @default("#111827")
    linkColor     String  @default("#2563eb")
    buttonColor   String  @default("#16a34a")
}

```

```

    buttonTextColor    String    @default("#ffffff")
    successColor       String    @default("#10b981")
    warningColor       String    @default("#f59e0b")
    errorColor         String    @default("#ef4444")

    // Typography
    headingFont        String    @default("Inter")
    bodyFont           String    @default("Inter")
    fontSize           Int       @default(16)

    // Layout
    headerStyle        String    @default("solid")
    navigationPosition String    @default("right")
    containerWidth     String    @default("contained")
    sectionSpacing     String    @default("normal")
    borderRadius       String    @default("rounded")
    footerStyle        String    @default("standard")

    // Content
    aboutText          String?
    servicesText       String?
    contactEmail       String?
    contactPhone       String?
    contactAddress     String?

    // SEO
    metaTitle          String?
    metaDescription    String?
    metaKeywords       String?

    // Advanced
    customCss          String?
    customJs           String?

    createdAt          DateTime @default(now())
    updatedAt          DateTime @updatedAt
}

```

Data Isolation Strategy

Row-Level Tenancy - All tenant-specific data includes `tenantId` foreign key - Every query filters by tenant ID - Prevents data leakage between tenants

Example Query:

```

// Get products for specific tenant
const products = await prisma.product.findMany({
  where: {
    tenantId: currentTenant.id
  }
});

```

Middleware Protection:

```

// Prisma middleware ensures tenant isolation
prisma.$use(async (params, next) => {

```

```

// Check if model is tenant-specific
if (isTenantModel(params.model)) {
  // Inject tenant filter
  params.args.where = {
    ...params.args.where,
    tenantId: getCurrentTenantId()
  };
}
return next(params);
});

```

Authentication & Authorization

NextAuth.js Configuration

Authentication Flow: 1. User submits credentials 2. NextAuth validates against database 3. JWT token generated with user info and tenant ID 4. Token stored in secure HTTP-only cookie 5. Subsequent requests include token 6. Middleware validates token and sets user context

Session Structure:

```

interface Session {
  user: {
    id: string;
    email: string;
    name: string;
    role: 'SUPER_ADMIN' | 'TENANT_ADMIN' | 'USER';
    tenantId?: string;
    tenantSlug?: string;
  };
  expires: string;
}

```

Role-Based Access Control (RBAC)

Permission Hierarchy:

SUPER_ADMIN

- Manage all tenants
- Platform configuration
- View all analytics
- System administration

TENANT_ADMIN

- Manage own tenant settings
- Update branding
- View tenant orders
- Manage tenant products

USER

- Browse products
- Place orders
- View own order history
- Update own profile

Authorization Middleware:

```
export function requireRole(allowedRoles: UserRole[]) {
  return async (req: NextApiRequest, res: NextApiResponse) => {
    const session = await getSession({ req });

    if (!session) {
      return res.status(401).json({ error: 'Unauthorized' });
    }

    if (!allowedRoles.includes(session.user.role)) {
      return res.status(403).json({ error: 'Forbidden' });
    }

    // Continue to handler
  };
}
```

Usage Example:

```
// API route for tenant admins only
export default async function handler(req, res) {
  await requireRole(['SUPER_ADMIN', 'TENANT_ADMIN'])(req, res);

  // Protected handler code
}
```

Password Security

- Passwords hashed with bcrypt (10 rounds)
 - Password reset tokens expire in 1 hour
 - Email verification required for new accounts
 - Strong password requirements enforced
-

Multi-Tenant Routing

Subdomain-Based Routing

DNS Configuration:

Main Platform: budstack.to → Server IP

Wildcard DNS: *.budstack.to → Server IP

Tenant Subdomain Structure: - healingbuds.budstack.to → HealingBuds Portugal store -
dispensary2.budstack.to → Second tenant store - www.budstack.to → Main platform marketing site -
app.budstack.to → Super admin/onboarding

Middleware Implementation

File: /middleware.ts

```
export async function middleware(request: NextRequest) {
  const hostname = request.headers.get('host') || '';
  const subdomain = getSubdomain(hostname);

  // Super admin routes
  if (request.nextUrl.pathname.startsWith('/super-admin')) {
```

```

    return verifyAdmin(request);
  }

  // Tenant store routes
  if (subdomain && subdomain !== 'www' && subdomain !== 'app') {
    const tenant = await getTenantBySubdomain(subdomain);

    if (!tenant) {
      return NextResponse.redirect('/404');
    }

    // Set tenant context
    request.headers.set('x-tenant-id', tenant.id);
    request.headers.set('x-tenant-slug', tenant.slug);

    return NextResponse.next();
  }

  return NextResponse.next();
}

function getSubdomain(hostname: string): string | null {
  const parts = hostname.split('.');
  if (parts.length >= 3) {
    return parts[0]; // e.g., 'healingbuds' from 'healingbuds.budstack.to'
  }
  return null;
}

```

Tenant Context Hook

```

// lib/tenant.ts
export function useTenant() {
  const { data: session } = useSession();
  const hostname = typeof window !== 'undefined' ? window.location.hostname : '';
  const subdomain = getSubdomain(hostname);

  // Fetch tenant based on subdomain
  const { data: tenant } = useSWR(
    subdomain ? `/api/tenant/${subdomain}` : null,
    fetcher
  );

  return {
    tenant,
    isLoading: !tenant && subdomain,
    currentTenantId: session?.user?.tenantId || tenant?.id
  };
}

```

Custom Domain Support

Process: 1. Tenant provides custom domain (e.g., healingbuds.pt) 2. Super admin adds domain to tenant record 3. System provides DNS instructions to tenant 4. Tenant adds A record pointing to platform IP 5. SSL certificate provisioned (Let's Encrypt) 6. Domain verified and activated

DNS Requirements:

A Record:

Host: @
Value: [Platform IP]

CNAME Record:

Host: www
Value: [tenant-slug].budstack.to

API Integration

Doctor Green API

Purpose: Central product catalog for medical cannabis products

Authentication: Two-layer system 1. Platform-level credentials 2. Per-tenant credentials (optional override)

Endpoints Used: - GET /products - List all products - GET /products/:id - Get product details - POST /orders - Create order - GET /inventory - Check stock levels

API Client Implementation:

```
// lib/doctor-green-api.ts
export class DoctorGreenAPI {
  private baseUrl: string;
  private apiKey: string;

  constructor(tenantId?: string) {
    // Use tenant-specific credentials if available
    const tenant = await getTenant(tenantId);
    this.baseUrl = tenant?.doctorGreenApiUrl || process.env.DOCTOR_GREEN_API_URL;
    this.apiKey = tenant?.doctorGreenApiKey || process.env.DOCTOR_GREEN_API_KEY;
  }

  async getProducts() {
    try {
      const response = await fetch(`${this.baseUrl}/products`, {
        headers: {
          'Authorization': `Bearer ${this.apiKey}`,
          'Content-Type': 'application/json'
        }
      });
      if (!response.ok) {
        throw new Error(`API error: ${response.status}`);
      }

      return await response.json();
    } catch (error) {
      console.error('Doctor Green API error:', error);
      // Fallback to mock data or cached data
      return getMockProducts();
    }
  }
}
```

```
}  
}
```

Error Handling & Fallback: - Retry logic with exponential backoff - Fallback to cached data if API unavailable - Mock data for development/testing - Error logging and monitoring

API Usage Tracking:

```
interface ApiLog {  
  tenantId: string;  
  endpoint: string;  
  method: string;  
  statusCode: number;  
  responseTime: number;  
  timestamp: Date;  
}
```

File Storage

Cloud Storage Architecture

Strategy: AWS S3 or S3-compatible storage (MinIO, DigitalOcean Spaces)

Folder Structure:

```
bucket-name/  
  tenants/  
    [tenant-id]/  
      logo.png  
      favicon.ico  
      hero-image.jpg  
      documents/  
        license.pdf  
        certification.pdf  
  products/  
    [product-images]  
  orders/  
    [order-documents]
```

Upload Implementation:

```
// lib/s3.ts  
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';  
  
export async function uploadFile(  
  file: Buffer,  
  key: string,  
  contentType: string  
): Promise<string> {  
  const s3Client = new S3Client({});  
  const bucketName = process.env.AWS_BUCKET_NAME;  
  
  await s3Client.send(new PutObjectCommand({  
    Bucket: bucketName,  
    Key: key,  
    Body: file,  
  }));  
}
```

```

        ContentType: contentType,
        ACL: 'public-read'
    }));

    return `https://${bucketName}.s3.amazonaws.com/${key}`;
}

```

File Upload Flow: 1. User selects file in UI 2. File sent to API route as FormData 3. Server validates file (type, size) 4. Generate unique filename with tenant ID 5. Upload to S3 6. Store S3 URL in database 7. Return URL to client

Security: - Signed URLs for private files - File type validation - Size limits enforced - Virus scanning (optional)

Scalability & Performance

Performance Optimizations

Database: - Indexed tenant ID columns - Connection pooling - Query optimization - Read replicas for analytics

Caching: - Redis for session storage - Static asset caching (CDN) - API response caching - Tenant theme caching

Frontend: - Next.js static generation where possible - Image optimization - Code splitting - Lazy loading

API: - Rate limiting per tenant - Request throttling - Background job processing - Async operations

Scalability Considerations

Horizontal Scaling: - Stateless application servers - Load balancer distribution - Shared session store (Redis) - Database replication

Vertical Scaling: - Database scaling (read replicas, sharding) - Resource monitoring - Auto-scaling policies

Monitoring: - Application performance monitoring (APM) - Error tracking - Usage analytics - Resource utilization

Security Measures

Data Security

- **Encryption at Rest:** Database and file storage encrypted
- **Encryption in Transit:** TLS/SSL for all connections
- **Secrets Management:** Environment variables, never in code
- **API Key Rotation:** Regular credential updates

Application Security

- **SQL Injection Protection:** Prisma ORM parameterized queries
- **XSS Protection:** React escapes output by default
- **CSRF Protection:** NextAuth CSRF tokens
- **Rate Limiting:** Prevent brute force attacks
- **Input Validation:** Server-side validation for all inputs

Compliance

- **GDPR:** Data privacy, right to deletion
 - **HIPAA:** Medical information protection
 - **PCI-DSS:** Payment data security (if handling cards)
 - **Cannabis Regulations:** Industry-specific compliance
-

Deployment Architecture

Production Environment

CloudFlare (CDN, DDoS protection)

Load Balancer

App 1 App 2 App 3 App N (Auto-scaled)

Postgres Redis
Primary Cache

Postgres
Replica

Environment Variables

Required:

```
# Database
DATABASE_URL="postgresql://..."

# NextAuth
NEXTAUTH_SECRET="..."
NEXTAUTH_URL="https://budstack.to"

# Email
SMTP_HOST="..."
SMTP_PORT="587"
SMTP_USER="..."
```

```
SMTP_PASSWORD="..."
SMTP_FROM="noreply@budstack.to"
```

Storage

```
AWS_ACCESS_KEY_ID="..."
AWS_SECRET_ACCESS_KEY="..."
AWS_BUCKET_NAME="..."
AWS_REGION="..."
```

Doctor Green API

```
DOCTOR_GREEN_API_URL="..."
DOCTOR_GREEN_API_KEY="..."
```

Namecheap API

```
NAMECHEAP_API_USER="..."
NAMECHEAP_API_KEY="..."
NAMECHEAP_IP_WHITELIST="..."
```

Conclusion

BudStack's multi-tenant architecture provides a scalable, secure, and flexible platform for medical cannabis dispensaries. The subdomain-based routing, combined with row-level tenancy, ensures complete data isolation while maintaining a unified codebase.

Key architectural benefits: - Single codebase for all tenants - Complete data isolation - Flexible theming system - Scalable infrastructure - Secure by design - Easy tenant onboarding

Document Version: 1.0

Last Updated: November 2025