VINEFOX Database & API Architecture Guide

🗄️ Database Architecture Overview

Current Version: vinefox (5f3b53c)
Database: PostgreSQL (production), SQLite (development)
ORM: Prisma 6.8.2
API Framework: Next.js 15.3.1 API Routes

🏗️ Database Schema Design

Core Data Models

Wine Model (Primary Entity)

```
model Wine {
  name       String    // Wine name (e.g., "Château Margaux 2015")
  type       String    // Wine type (e.g., "Red", "White", "Rosé", "Sparkling")
  elaborate  String?   // Detailed description and tasting notes
  grapes     String    // Grape varieties used (e.g., "Cabernet Sauvignon,
Merlot")
  harmonize  String    // Food pairing suggestions
  abv        Float     // Alcohol by volume percentage
  body       String    // Wine body (Light, Medium, Full)
  acidity    String    // Acidity level (Low, Medium, High)
  code       String    // Unique product code for inventory management
  price      Int       // Price in cents (e.g., 7500 = $75.00)
  regionId   Int       // Geographic region reference
  clerkId    String?   // Optional Clerk user ID for user-specific data
  createdAt  DateTime  @default(now()) // Creation timestamp
  featured   Boolean   @default(false) // Featured wine flag for homepage
  image      String?   // Primary image URL
  updatedAt  DateTime  @updatedAt      // Last update timestamp
  id         Int       @id @default(autoincrement()) // Primary key

  // Relationships
  region     Region    @relation("RegionToWine", fields: [regionId],
references: [id])
  cartItems  CartItem[] // Shopping cart items
  favorites  Favorite[] // User favorites
  images     Image[]    // Multiple product images
  orderItems OrderItem[] // Order line items
  reviews    Review[]   // User reviews and ratings
}
```

Key Design Decisions:

- **Price in Cents:** Prevents floating-point precision issues in financial calculations
- **Optional clerkId:** Allows for user-specific wine data without requiring authentication
- **Featured Flag:** Enables dynamic homepage content management
- **Multiple Images:** Supports rich product photography and marketing

Region Model (Geographic Classification)

```
model Region {
  id      Int     @id @default(autoincrement())
  name    String  // Region name (e.g., "Bordeaux", "Tuscany", "Napa Valley")
  country String  // Country name (e.g., "France", "Italy", "United States")
  wines   Wine[]  @relation("RegionToWine") // Wines from this region
}
```

Geographic Organization Benefits:

- **Wine Education:** Helps customers understand wine origins and characteristics
- **Filtering:** Enables region-based product discovery
- **Cultural Context:** Provides educational value about wine regions
- **Marketing:** Supports region-specific promotions and campaigns

Image Management System

```
model Image {
  id     Int    @id @default(autoincrement())
  url    String // Image URL (supports multiple formats: WebP, JPEG, PNG)
  wineId Int    // Associated wine
  wine   Wine   @relation(fields: [wineId], references: [id], onDelete:
Cascade)
}
```

Image System Features:

- **Multiple Images per Wine:** Supports product galleries and marketing
- **Cascade Deletion:** Automatically removes images when wine is deleted
- **Format Flexibility:** Supports modern image formats for performance
- **CDN Ready:** URLs can point to CDN for global performance

User Interaction Models

Favorite System

```
model Favorite {
  id        String   @id @default(uuid()) // UUID for security
  clerkId   String   // Clerk user identifier
  createdAt DateTime @default(now())      // When favorited
  updatedAt DateTime @updatedAt           // Last update
  wineId    Int      // Associated wine
  wine      Wine     @relation(fields: [wineId], references: [id], onDelete:
Cascade)
}
```

Favorites Implementation:

- **User-Specific:** Each user has their own favorites list
- **Real-time Updates:** Immediate UI feedback with optimistic updates
- **Cascade Deletion:** Automatically removed if wine is deleted
- **Performance Optimized:** Efficient queries with proper indexing

Review System

```
model Review {
  id             String   @id @default(uuid())
  clerkId        String   // Authenticated user identifier
  rating         Int      // 1-5 star rating
  comment        String   // Review text content
  authorName     String   // Display name for the review
  authorImageUrl String   // User profile image
  vintage        String?  // Optional vintage year specification
  createdAt      DateTime @default(now())
  updatedAt      DateTime @updatedAt
  wineId         Int      // Associated wine
  wine           Wine     @relation(fields: [wineId], references: [id],
onDelete: Cascade)
}
```

Review System Features:

- **Authenticated Users Only:** Prevents spam and ensures quality
- **Rich Content:** Supports text, ratings, and vintage information
- **User Attribution:** Links reviews to specific users
- **Moderation Ready:** Structure supports admin approval workflow

E-commerce Models

Shopping Cart System

```
model Cart {
  id             String     @id @default(uuid())
  clerkId        String     // User-specific cart
  numItemsInCart Int        @default(0) // Total item count
  cartTotal      Int        @default(0) // Subtotal in cents
  shipping       Int        @default(5) // Shipping cost in cents ($5.00)
  tax            Int        @default(0) // Calculated tax in cents
  taxRate        Float      @default(0.1) // 10% tax rate
  orderTotal     Int        @default(0) // Final total in cents
  createdAt      DateTime   @default(now())
  updatedAt      DateTime   @updatedAt

  cartItems      CartItem[] // Individual cart items
}

model CartItem {
  id        String   @id @default(uuid())
  cartId    String   // Associated cart
  amount    Int      // Quantity of wine
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  wineId    Int      // Associated wine

  // Relationships
  cart      Cart     @relation(fields: [cartId], references: [id], onDelete:
Cascade)
  wine      Wine     @relation(fields: [wineId], references: [id], onDelete:
Cascade)
}
```

Cart System Design:

- **User-Specific:** Each authenticated user has their own cart
- **Real-time Calculations:** Automatic tax, shipping, and total updates
- **Persistent Storage:** Cart survives browser sessions and device changes
- **Cascade Deletion:** Cart items automatically removed when cart is deleted

Order Management System

```
model Order {
  id          String      @id @default(uuid())
  clerkId    String      // Customer identifier
  products   Int         @default(0) // Total product count
  orderTotal Int         @default(0) // Order total in cents
  tax        Int         @default(0) // Tax amount in cents
  shipping   Int         @default(0) // Shipping cost in cents
  email      String      // Customer email for notifications
  isPaid     Boolean     @default(false) // Payment status
  createdAt  DateTime    @default(now())
  updatedAt  DateTime    @updatedAt

  orderItems OrderItem[] // Order line items
}

model OrderItem {
  id        String   @id @default(uuid())
  orderId   String   // Associated order
  wineId    Int      // Wine purchased
  amount    Int      // Quantity purchased
  price     Int      // Price at time of purchase (in cents)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  // Relationships
  order     Order    @relation(fields: [orderId], references: [id], onDelete:
Cascade)
  wine      Wine     @relation(fields: [wineId], references: [id])
}
```

Order System Features:

- **Price Snapshot:** Records price at time of purchase for historical accuracy
- **Payment Tracking:** Tracks payment status for order fulfillment
- **Email Integration:** Customer email for order notifications
- **Audit Trail:** Complete order history with timestamps

🔌 API Architecture & Implementation

API Route Structure

Product Management API (`/app/api/products/`)

Get All Products (`route.ts`)

```
// GET /api/products
export async function GET(request: NextRequest) {
  try {
    const { searchParams } = new URL(request.url);

    // Extract query parameters
    const page = parseInt(searchParams.get("page") || "1");
    const limit = parseInt(searchParams.get("limit") || "12");
    const search = searchParams.get("search") || "";
    const type = searchParams.get("type") || "";
    const region = searchParams.get("region") || "";
    const minPrice = parseInt(searchParams.get("minPrice") || "0");
    const maxPrice = parseInt(searchParams.get("maxPrice") || "1000000");
    const sortBy = searchParams.get("sortBy") || "name";
    const sortOrder = searchParams.get("sortOrder") || "asc";

    // Build Prisma query with filters
    const where: Prisma.WineWhereInput = {
      AND: [
        search
          ? {
              OR: [
                { name: { contains: search, mode: "insensitive" } },
                { elaborate: { contains: search, mode: "insensitive" } },
                { grapes: { contains: search, mode: "insensitive" } },
              ],
            }
          : {},
        type ? { type: { equals: type, mode: "insensitive" } } : {},
        region
          ? { region: { name: { equals: region, mode: "insensitive" } } }
          : {},
        { price: { gte: minPrice, lte: maxPrice } },
      ],
    };

    // Execute query with pagination
    const [wines, total] = await Promise.all([
      prisma.wine.findMany({
        where,
        include: {
          region: true,
          images: true,
          reviews: true,
        },
        orderBy: { [sortBy]: sortOrder },
        skip: (page - 1) * limit,
        take: limit,
      }),
      prisma.wine.count({ where }),
    ]);

    // Calculate pagination metadata
    const totalPages = Math.ceil(total / limit);
    const hasNextPage = page < totalPages;
    const hasPrevPage = page > 1;

    return NextResponse.json({
      wines,
      pagination: {
        page,
        limit,
        total,
        totalPages,
        hasNextPage,
        hasPrevPage,
      },
    });
  } catch (error) {
    console.error("Error fetching products:", error);
    return NextResponse.json(
      { error: "Failed to fetch products" },
      { status: 500 }
    );
  }
}
```

Key Implementation Features:

- **Comprehensive Filtering:** Search, type, region, and price range filtering
- **Pagination Support:** Efficient data loading with page-based navigation
- **Sorting Options:** Multiple sorting criteria for user preference
- **Error Handling:** Graceful error handling with appropriate HTTP status codes
- **Performance Optimization:** Efficient database queries with proper indexing

Get Single Product (`[handle]/route.ts`)

```
// GET /api/products/[handle]
export async function GET(
  request: NextRequest,
  { params }: { params: { handle: string } }
) {
  try {
    const wine = await prisma.wine.findUnique({
      where: { id: parseInt(params.handle) },
      include: {
        region: true,
        images: true,
        reviews: {
          include: {
            user: true,
          },
          orderBy: { createdAt: "desc" },
        },
        _count: {
          select: {
            reviews: true,
            favorites: true,
          },
        },
      },
    });

    if (!wine) {
      return NextResponse.json({ error: "Wine not found" }, { status: 404 });
    }

    return NextResponse.json(wine);
  } catch (error) {
    console.error("Error fetching wine:", error);
    return NextResponse.json(
      { error: "Failed to fetch wine" },
      { status: 500 }
    );
  }
}
```

Single Product Features:

- **Complete Data:** Includes all related data (region, images, reviews)
- **Review Integration:** Fetches reviews with user information
- **Count Aggregation:** Provides review and favorite counts
- **Error Handling:** Proper 404 handling for missing products

Cart Management API (`/app/api/cart/`)

Get User Cart

```
// GET /api/cart
export async function GET(request: NextRequest) {
  try {
    const { userId } = await auth();

    if (!userId) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }
    // Get or create user cart
    let cart = await prisma.cart.findFirst({
      where: { clerkId: userId },
      include: {
        cartItems: {
          include: {
            wine: {
              include: {
                images: true,
                region: true,
              },
            },
          },
        },
      },
    });

    if (!cart) {
      // Create new cart for user
      cart = await prisma.cart.create({
        data: {
          clerkId: userId,
          numItemsInCart: 0,
          cartTotal: 0,
          shipping: 500, // $5.00 in cents
          tax: 0,
          taxRate: 0.1, // 10% tax rate
          orderTotal: 0,
        },
        include: {
          cartItems: {
            include: {
              wine: {
                include: {
                  images: true,
                  region: true,
                },
              },
            },
          },
        },
      });
    }

    // Calculate cart totals
    const cartTotal = cart.cartItems.reduce(
      (sum, item) => sum + item.wine.price * item.amount,
      0
    );

    const tax = Math.round(cartTotal * cart.taxRate);
    const orderTotal = cartTotal + tax + cart.shipping;
    const numItemsInCart = cart.cartItems.reduce(
      (sum, item) => sum + item.amount,
      0
    );

    // Update cart with calculated totals
    const updatedCart = await prisma.cart.update({
      where: { id: cart.id },
      data: {
        cartTotal,
        tax,
        orderTotal,
        numItemsInCart,
      },
      include: {
        cartItems: {
          include: {
            wine: {
              include: {
                images: true,
                region: true,
              },
            },
          },
        },
      },
    });

    return NextResponse.json(updatedCart);
  } catch (error) {
    console.error("Error fetching cart:", error);
    return NextResponse.json(
      { error: "Failed to fetch cart" },
      { status: 500 }
    );
  }
}
```

Cart Management Features:

- **Automatic Creation:** Creates cart for new users automatically
- **Real-time Calculations:** Calculates totals, tax, and shipping automatically
- **Data Relationships:** Includes complete wine and image data
- **Error Handling:** Proper authentication and error handling

Add Item to Cart

```
// POST /api/cart
export async function POST(request: NextRequest) {
  try {
    const { userId } = await auth();

    if (!userId) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const { wineId, amount } = await request.json();

    // Validate input
    if (!wineId || !amount || amount < 1) {
      return NextResponse.json({ error: "Invalid input" }, { status: 400 });
    }

    // Get or create user cart
    let cart = await prisma.cart.findFirst({
      where: { clerkId: userId },
    });

    if (!cart) {
      cart = await prisma.cart.create({
        data: {
          clerkId: userId,
          numItemsInCart: 0,
          cartTotal: 0,
          shipping: 500,
          tax: 0,
          taxRate: 0.1,
          orderTotal: 0,
        },
      });
    }

    // Check if item already exists in cart
    const existingItem = await prisma.cartItem.findFirst({
      where: {
        cartId: cart.id,
        wineId: parseInt(wineId),
      },
    });

    if (existingItem) {
      // Update existing item quantity
      await prisma.cartItem.update({
        where: { id: existingItem.id },
        data: { amount: existingItem.amount + amount },
      });
    } else {
      // Add new item to cart
      await prisma.cartItem.create({
        data: {
          cartId: cart.id,
          wineId: parseInt(wineId),
          amount,
        },
      });
    }

    // Return updated cart
    const updatedCart = await prisma.cart.findFirst({
      where: { id: cart.id },
      include: {
        cartItems: {
          include: {
            wine: {
              include: {
                images: true,
                region: true,
              },
            },
          },
        },
      },
    });

    return NextResponse.json(updatedCart);
  } catch (error) {
    console.error("Error adding item to cart:", error);
    return NextResponse.json(
      { error: "Failed to add item to cart" },
      { status: 500 }
    );
  }
}
```

Add to Cart Features:

- **Quantity Management:** Handles existing items by updating quantity
- **Input Validation:** Validates wine ID and amount
- **Cart Creation:** Automatically creates cart for new users
- **Complete Response:** Returns updated cart with all items

Order Management API (`/app/api/orders/`)

Create Order

```
// POST /api/orders
export async function POST(request: NextRequest) {
  try {
    const { userId } = await auth();

    if (!userId) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const { email, items } = await request.json();

    // Validate input
    if (!email || !items || items.length === 0) {
      return NextResponse.json({ error: "Invalid input" }, { status: 400 });
    }

    // Calculate order totals
    const orderTotal = items.reduce(
      (sum: number, item: any) => sum + item.price * item.amount,
      0
    );

    const tax = Math.round(orderTotal * 0.1); // 10% tax
    const shipping = 500; // $5.00 shipping
    const totalWithTaxAndShipping = orderTotal + tax + shipping;

    // Create order
    const order = await prisma.order.create({
      data: {
        clerkId: userId,
        products: items.reduce(
          (sum: number, item: any) => sum + item.amount,
          0
        ),
        orderTotal: totalWithTaxAndShipping,
        tax,
        shipping,
        email,
      },
    });

    // Create order items
    const orderItems = await Promise.all(
      items.map((item: any) =>
        prisma.orderItem.create({
          data: {
            orderId: order.id,
            wineId: item.wineId,
            amount: item.amount,
            price: item.price,
          },
        })
      )
    );

    // Clear user cart after order creation
    await prisma.cartItem.deleteMany({
      where: {
        cart: {
          clerkId: userId,
        },
      },
    });
```

```
      // Reset cart totals
      await prisma.cart.updateMany({
        where: { clerkId: userId },
        data: {
          numItemsInCart: 0,
          cartTotal: 0,
          tax: 0,
          orderTotal: 0,
        },
      });

      return NextResponse.json({
        order,
        orderItems,
        total: totalWithTaxAndShipping,
      });
    } catch (error) {
      console.error("Error creating order:", error);
      return NextResponse.json(
        { error: "Failed to create order" },
        { status: 500 }
      );
    }
  }
```

Order Creation Features:

- **Complete Calculation:** Handles tax, shipping, and totals
- **Cart Cleanup:** Automatically clears cart after order creation
- **Data Integrity:** Creates order and order items atomically
- **Email Integration:** Stores customer email for notifications

Payment Processing API (`/app/api/payment/`)

Create Stripe Checkout Session

```
// POST /api/payment
export async function POST(request: NextRequest) {
  try {
    const { userId } = await auth();

    if (!userId) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const { items, email } = await request.json();

    // Validate input
    if (!items || items.length === 0 || !email) {
      return NextResponse.json({ error: "Invalid input" }, { status: 400 });
    }

    // Calculate totals
    const subtotal = items.reduce(
      (sum: number, item: any) => sum + item.price * item.amount,
      0
    );

    const tax = Math.round(subtotal * 0.1);
    const shipping = 500;
    const total = subtotal + tax + shipping;
    // Create Stripe checkout session
    const session = await stripe.checkout.sessions.create({
      payment_method_types: ["card"],
      line_items: items.map((item: any) => ({
        price_data: {
          currency: "usd",
          product_data: {
            name: item.name,
            images: item.images,
          },
          unit_amount: item.price,
        },
        quantity: item.amount,
      })),
      mode: "payment",
      success_url: `${process.env.NEXT_PUBLIC_APP_URL}/orders/success?
session_id={CHECKOUT_SESSION_ID}`,
      cancel_url: `${process.env.NEXT_PUBLIC_APP_URL}/cart`,
      customer_email: email,
      metadata: {
        userId,
        orderData: JSON.stringify({
          items,
          subtotal,
          tax,
          shipping,
          total,
        }),
      },
    });

    return NextResponse.json({ sessionId: session.id });
  } catch (error) {
    console.error("Error creating checkout session:", error);
    return NextResponse.json(
      { error: "Failed to create checkout session" },
      { status: 500 }
    );
  }
}
```

Payment Features:

- **Stripe Integration:** Secure payment processing
- **Metadata Storage:** Stores order data in Stripe metadata
- **Success/Cancel URLs:** Proper redirect handling
- **Email Integration:** Pre-fills customer email in Stripe

🚀 Performance Optimization Strategies

Database Query Optimization

Indexing Strategy

```
-- Primary indexes (automatically created by Prisma)
CREATE INDEX "Wine_regionId_idx" ON "Wine"("regionId");
CREATE INDEX "CartItem_cartId_idx" ON "CartItem"("cartId");
CREATE INDEX "CartItem_wineId_idx" ON "CartItem"("wineId");
CREATE INDEX "OrderItem_orderId_idx" ON "OrderItem"("orderId");
CREATE INDEX "OrderItem_wineId_idx" ON "OrderItem"("wineId");

-- Composite indexes for common query patterns
```

```
CREATE INDEX "Wine_search_idx" ON "Wine"("name", "type", "price");
CREATE INDEX "Wine_featured_idx" ON "Wine"("featured", "createdAt");
CREATE INDEX "Review_wineId_createdAt_idx" ON "Review"("wineId", "createdAt");
```

Indexing Benefits:

- **Search Performance:** Fast text search across wine attributes
- **Filter Performance:** Efficient filtering by type, region, and price
- **Sort Performance:** Quick sorting by creation date and featured status
- **Join Performance:** Fast relationship queries between models

Query Optimization Techniques

```
// Efficient product query with proper includes
const wines = await prisma.wine.findMany({
  where: {
    // Use indexed fields for filtering
    price: { gte: minPrice, lte: maxPrice },
    type: type || undefined,
    region: region
      ? { name: { equals: region, mode: "insensitive" } }
      : undefined,
  },
  include: {
    // Only include necessary relations
    region: { select: { name: true, country: true } },
    images: { select: { url: true }, take: 1 }, // Limit to first image
    _count: { select: { reviews: true, favorites: true } },
  },
  orderBy: { [sortBy]: sortOrder },
  skip: (page - 1) * limit,
  take: limit,
});
```

Query Optimization Features:

- **Selective Includes:** Only fetch necessary relation data
- **Image Limiting:** Limit images to prevent data bloat
- **Count Aggregation:** Use Prisma's count feature for performance
- **Pagination:** Efficient skip/take for large datasets

API Response Optimization

Response Caching

```
// Cache frequently accessed data
export async function GET(request: NextRequest) {
  // Check cache first
  const cacheKey = `products:${JSON.stringify(searchParams)}`;
  const cached = await redis.get(cacheKey);

  if (cached) {
    return NextResponse.json(JSON.parse(cached));
  }

  // Fetch from database
  const result = await fetchProductsFromDatabase(searchParams);

  // Cache result for 5 minutes
  await redis.setex(cacheKey, 300, JSON.stringify(result));
  return NextResponse.json(result);
}
```

Caching Benefits:

- **Reduced Database Load:** Minimizes repeated queries
- **Faster Response Times:** Cached responses are nearly instant
- **Scalability:** Handles increased traffic without performance degradation
- **Cost Reduction:** Fewer database operations

Response Compression

```
// Enable gzip compression for API responses
export const config = {
  api: {
    responseLimit: false,
    bodyParser: {
      sizeLimit: "10mb",
    },
  },
};

// Compress large responses
import { gzip } from "zlib";
import { promisify } from "util";

const gzipAsync = promisify(gzip);

export async function GET(request: NextRequest) {
  const data = await fetchLargeDataset();

  // Compress response for large datasets
  if (JSON.stringify(data).length > 10000) {
    const compressed = await gzipAsync(JSON.stringify(data));
    return new Response(compressed, {
      headers: {
        "Content-Encoding": "gzip",
        "Content-Type": "application/json",
      },
    });
  }

  return NextResponse.json(data);
}
```

🔒 Security Implementation

Authentication & Authorization

Clerk Integration

```
// Middleware-based route protection
import { clerkMiddleware } from "@clerk/nextjs/server";

export default clerkMiddleware();

export const config = {
  matcher: [
    // Protect all routes except static assets
    "/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?
|ico|csv|docx?|xlsx?|zip|webmanifest)).*)",
    // Always run for API routes
    "/(api|trpc)(.*)",
  ],
};
```

Security Features:

- **Route Protection:** All routes protected by default
- **Static Asset Access:** Static files remain accessible
- **API Security:** All API endpoints protected
- **Session Management:** Automatic session handling

Input Validation

```
// Comprehensive input validation
import { z } from "zod";

const addToCartSchema = z.object({
  wineId: z.string().regex(/^\d+$/).transform(Number),
  amount: z.number().int().positive().max(100),
});

export async function POST(request: NextRequest) {
  try {
    const body = await request.json();
    const { wineId, amount } = addToCartSchema.parse(body);

    // Process validated data
    // ...
  } catch (error) {
    if (error instanceof z.ZodError) {
      return NextResponse.json(
        { error: "Invalid input data", details: error.errors },
        { status: 400 }
      );
    }

    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 }
    );
  }
}
```

Validation Benefits:

- **Type Safety:** Ensures data types are correct
- **Input Sanitization:** Prevents malicious input
- **Error Handling:** Clear error messages for invalid data
- **Security:** Prevents injection attacks

Data Protection

SQL Injection Prevention

```
// Prisma automatically prevents SQL injection
const wines = await prisma.wine.findMany({
  where: {
    name: { contains: searchTerm, mode: "insensitive" },
    price: { gte: minPrice, lte: maxPrice },
  },
});

// No manual SQL construction - Prisma handles parameterization
```

Prisma Security Features:

- **Parameterized Queries:** Automatic SQL injection prevention
- **Type Safety:** Compile-time type checking
- **Query Validation:** Built-in query validation
- **Connection Pooling:** Secure database connections

XSS Protection

```
// Next.js built-in XSS protection
// All user input is automatically sanitized

// For custom content, use proper escaping
import DOMPurify from "dompurify";

const sanitizedContent = DOMPurify.sanitize(userContent);
```

📊 Database Migration & Maintenance

Migration Strategy

Safe Schema Updates

```
// Prisma migration workflow
// 1. Update schema.prisma
// 2. Generate migration
// 3. Review migration SQL
// 4. Apply migration
// 5. Verify data integrity

// Example migration for adding new field
model Wine {
  // ... existing fields
  vintage String? // New optional field
}
```

Migration Best Practices:

- **Incremental Changes:** Small, focused schema updates
- **Backward Compatibility:** Maintain existing functionality
- **Data Validation:** Verify data integrity after migration
- **Rollback Plan:** Prepare rollback procedures

Data Seeding

```
// prisma/seed.js
import { PrismaClient } from "@prisma/client";
import { parse } from "csv-parse/sync";
import fs from "fs";
```

```
const prisma = new PrismaClient();

async function main() {
  // Read CSV data
  const wineData = parse(fs.readFileSync("./wine100.csv"), {
    columns: true,
    skip_empty_lines: true,
  });

  // Seed database
  for (const wine of wineData) {
    await prisma.wine.create({
      data: {
        name: wine.name,
        type: wine.type,
        elaborate: wine.elaborate,
        grapes: wine.grapes,
        harmonize: wine.harmonize,
        abv: parseFloat(wine.abv),
        body: wine.body,
        acidity: wine.acidity,
        code: wine.code,
        price: parseInt(wine.price),
        regionId: parseInt(wine.regionId),
        featured: wine.featured === "true",
      },
    });
  }
}

main()
  .catch((e) => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
```

Seeding Features:

- **CSV Import:** Bulk data import from CSV files
- **Data Validation:** Ensures data quality during import
- **Error Handling:** Graceful error handling for invalid data
- **Performance:** Efficient bulk insert operations

Database Maintenance

Regular Maintenance Tasks

```
-- Database optimization queries
-- 1. Update table statistics
ANALYZE "Wine";
ANALYZE "Cart";
ANALYZE "Order";

-- 2. Clean up orphaned records
DELETE FROM "CartItem"
WHERE "cartId" NOT IN (SELECT id FROM "Cart");

DELETE FROM "OrderItem"
WHERE "orderId" NOT IN (SELECT id FROM "Order");
```

```sql
-- 3. Vacuum database (PostgreSQL)
VACUUM ANALYZE;
```

Maintenance Benefits:

- **Performance:** Optimized query execution plans
- **Data Integrity:** Clean, consistent data
- **Storage Efficiency:** Reclaimed storage space
- **Query Optimization:** Better database performance

🔍 Monitoring & Debugging

Performance Monitoring

Query Performance Tracking

```typescript
// Prisma query logging
const prisma = new PrismaClient({
  log: [
    {
      emit: "event",
      level: "query",
    },
    {
      emit: "event",
      level: "error",
    },
  ],
});

prisma.$on("query", (e) => {
  console.log("Query: " + e.query);
  console.log("Params: " + e.params);
  console.log("Duration: " + e.duration + "ms");
});

prisma.$on("error", (e) => {
  console.error("Prisma Error:", e);
});
```

Monitoring Features:

- **Query Logging:** Track all database queries
- **Performance Metrics:** Monitor query execution times
- **Error Tracking:** Capture and log database errors
- **Performance Analysis:** Identify slow queries

API Performance Monitoring

```typescript
// API response time monitoring
export async function GET(request: NextRequest) {
  const startTime = Date.now();

  try {
    const result = await fetchData();

    const responseTime = Date.now() - startTime;
    console.log(`API Response Time: ${responseTime}ms`);
    // Log slow responses
    if (responseTime > 1000) {
      console.warn(`Slow API Response: ${responseTime}ms`);
    }

    return NextResponse.json(result);
  } catch (error) {
    const responseTime = Date.now() - startTime;
    console.error(`API Error (${responseTime}ms):`, error);

    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 }
    );
  }
}
```

Error Handling & Logging

Comprehensive Error Handling

```
// Centralized error handling
class DatabaseError extends Error {
  constructor(
    message: string,
    public code: string,
    public statusCode: number = 500
  ) {
    super(message);
    this.name = "DatabaseError";
  }
}

async function handleDatabaseOperation<T>(
  operation: () => Promise<T>,
  errorMessage: string
): Promise<T> {
  try {
    return await operation();
  } catch (error) {
    console.error(`${errorMessage}:`, error);

    if (error.code === "P2002") {
      throw new DatabaseError("Duplicate entry", "DUPLICATE", 409);
    }

    if (error.code === "P2025") {
      throw new DatabaseError("Record not found", "NOT_FOUND", 404);
    }

    throw new DatabaseError("Database operation failed", "DATABASE_ERROR",
500);
  }
}

// Usage in API routes
export async function GET(request: NextRequest) {
  try {
    const wines = await handleDatabaseOperation(
      () => prisma.wine.findMany(),
      "Failed to fetch wines"
    );

    return NextResponse.json(wines);
  } catch (error) {
    if (error instanceof DatabaseError) {
      return NextResponse.json(
        { error: error.message, code: error.code },
        { status: error.statusCode }
      );
    }

    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 }
    );
  }
}
```

Error Handling Features:

- **Custom Error Types:** Specific error classes for different scenarios
- **HTTP Status Codes:** Appropriate HTTP status codes for errors
- **Error Logging:** Comprehensive error logging for debugging
- **User-Friendly Messages:** Clear error messages for users

This database and API architecture document reflects the current implementation state and should be updated whenever significant changes are made to the database schema or API structure. For the most current information, refer to the source code and recent commit history.