Nuxt + Cloudflare Workers + Hyperdrive + Prisma

Architecture, Setup, and Integration Guide

1. Overview & Goals

This document describes how to build and integrate a backend API for a marketplace-style Nuxt 3 application using Cloudflare Workers, Cloudflare Hyperdrive, Amazon RDS (PostgreSQL), and Prisma. The API is split into two logical surfaces, both implemented in a single Worker script:

- **Private API**: Consumed only by your own Nuxt frontend. Authenticated via session JWT stored in an HttpOnly cookie, and exposed on the hostname api.example.com.
- **Public API**: Exposed to third-party integrators. Authenticated via per-account API keys passed in an Authorization: ApiKey ... header, and exposed on public-api.example.com.

Backend data is stored in a PostgreSQL database hosted on Amazon RDS. Cloudflare Hyperdrive provides a pooled, serverless-friendly connection between Cloudflare Workers and the RDS instance. Prisma is used as the ORM/data access layer, with its driver adapters feature to run inside Workers.

High-Level Component Diagram (conceptual)

- Nuxt app (app.example.com) → calls private API on api.example.com using cookies (session JWT).
- Third-party clients → call public API on public-api.example.com using API keys.
- Both hostnames map to the same Cloudflare Worker script.
- Worker → connects to Amazon RDS Postgres through Cloudflare Hyperdrive using Prisma.

2. Infrastructure Setup

2.1 PostgreSQL on Amazon RDS

1) In AWS, create a PostgreSQL RDS instance (choose appropriate size and storage for development or production).
2) Configure security groups / networking so that only Cloudflare / Hyperdrive can reach the database. For early development you may temporarily allow your IP, but plan to tighten this for production.
3) Create a dedicated database user for your application (do not use the master user in production).
4) Note the connection string in the form: postgresql://USER:PASSWORD@HOST:PORT/DB_NAME. This is used by Prisma migrations and by Hyperdrive.

2.2 Cloudflare Worker Project (Wrangler)

```
# Create a new Workers project in TypeScript
npm create cloudflare@latest marketplace-api-worker -- --type=hello-world --ts=true --git=true --deploy=false
cd marketplace-api-worker
```

This creates a minimal Worker project. You will extend it with Prisma, routing, and authentication logic.

2.3 Enable Node.js Compatibility and Hyperdrive

Edit wrangler.toml to enable Node.js compatibility (needed for Prisma and the pg driver) and to bind your Hyperdrive configuration.

```
# wrangler.toml (core parts only)
name = "marketplace-api-worker"
main = "src/index.ts"
compatibility_date = "2024-09-23"
compatibility_flags = ["nodejs_compat"]

[[hyperdrive]]
binding = "HYPERDRIVE"
id = "<your-hyperdrive-id-here>
```

The binding value (HYPERDRIVE) becomes env.HYPERDRIVE inside your Worker code. The id is obtained when you create a Hyperdrive config in the Cloudflare dashboard.

2.4 Hostnames and Routes

Configure two hostnames in Cloudflare DNS and route them to the same Worker:

```
[env.production]
routes = [
  { pattern = "api.example.com/*", zone_name = "example.com" },
  { pattern = "public-api.example.com/*", zone_name = "example.com" }
]
```

Both hostnames execute the same Worker script. The script will inspect url.hostname to decide whether to handle the request as private (Nuxt) or public (third-party API).

3. Prisma Setup (Workers + Hyperdrive)

3.1 Install Prisma and Dependencies

```
npm install -D prisma
npx prisma init

npm install pg@>8.13.0 @prisma/adapter-pg
npm install -D @types/pg
```

This initializes a Prisma project and installs the PostgreSQL driver plus the Prisma driver adapter for pg. Prisma will be used both locally (for migrations) and in the Worker (via the adapter).

3.2 Prisma Schema (Marketplace Models)

Edit prisma/schema.prisma to define the generator, datasource, and basic marketplace models. The schema below is intentionally simplified and can be extended as needed.

```
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["driverAdapters"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String   @id @default(cuid())
  email     String   @unique
  password  String
  role      String   // "buyer" | "seller" | "admin"
  createdAt DateTime @default(now())
  orders    Order[]
  apiKeys   ApiKey[]
}

model Product {
  id          String   @id @default(cuid())
  title       String
  description String
  priceCents  Int
  stock       Int
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  orderItems  OrderItem[]
}

model Order {
  id        String      @id @default(cuid())
  userId    String
  user      User        @relation(fields: [userId], references: [id])
  status    String      // "PENDING" | "PAID" | "CANCELLED"
  createdAt DateTime    @default(now())
  items     OrderItem[]
}

model OrderItem {
  id         String   @id @default(cuid())
  orderId    String
  productId  String
  quantity   Int
  priceCents Int

  order   Order   @relation(fields: [orderId], references: [id])
  product Product @relation(fields: [productId], references: [id])
}
model ApiKey {
```

```
  id           String   @id @default(cuid())
  userId       String
  user         User     @relation(fields: [userId], references: [id])
  name         String
  keyHash      String   @unique
  scopes       String[]
  rateLimit    Int      @default(10000)
  createdAt    DateTime @default(now())
  lastUsedAt   DateTime?
  disabledAt   DateTime?
}
```

3.3 Local .env and Migrations

Prisma uses a local .env file to find DATABASE_URL for migrations and generation. This value should be a direct PostgreSQL connection string to your RDS instance (or to a local Postgres during early development).

```
# .env (local dev / CI for migrations)
DATABASE_URL="postgresql://USER:PASSWORD@HOST:5432/DB_NAME"
```

Add scripts to package.json:

```
"scripts": {
  "prisma:migrate": "prisma migrate dev",
  "prisma:generate": "prisma generate --no-engine"
}
```

Run the migrations and generate the client:

```
npm run prisma:migrate
npm run prisma:generate
```

4. Worker Code Structure

4.1 Suggested Directory Layout

```
src/
  index.ts              # Worker entrypoint: routing & dispatch
  prismaClient.ts       # Prisma client helper using Hyperdrive adapter

  authPrivate.ts        # Cookie-based session auth (Nuxt)
  authPublic.ts         # API key auth (third parties)

  routes/
    private/
      me.ts             # GET /internal/me
      products.ts       # GET /internal/products, etc.
      orders.ts         # POST /internal/orders, etc.
      auth.ts           # POST /internal/auth/login, logout

    public/
      products.ts       # GET /public/v1/products
      orders.ts         # e.g. GET /public/v1/orders (scoped)
```

4.2 Prisma Client Helper (Workers + Hyperdrive)

The helper below creates a PrismaClient instance using the Hyperdrive binding. For simplicity, it creates a new client per request. Hyperdrive handles pooling at the network edge.

```
// src/prismaClient.ts
import { PrismaClient } from "@prisma/client"
import { PrismaPg } from "@prisma/adapter-pg"

export interface Env {
  HYPERDRIVE: { connectionString: string }
  JWT_SECRET: string
}

export function createPrisma(env: Env): PrismaClient {
  const adapter = new PrismaPg({
    connectionString: env.HYPERDRIVE.connectionString,
  })

  return new PrismaClient({ adapter })
}
```

4.3 Worker Entrypoint (Hostname-Based Routing)

```
// src/index.ts
import type { Env } from "./prismaClient"
import { authenticatePrivate } from "./authPrivate"
import { authenticatePublicApiKey } from "./authPublic"
import { handlePrivateApi } from "./routes/private/index"
import { handlePublicApi } from "./routes/public/index"

function json(body: unknown, status = 200): Response {
  return new Response(JSON.stringify(body), {
    status,
    headers: { "Content-Type": "application/json" },
  })
}

export default {
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise<Response> {
    const url = new URL(request.url)
    const hostname = url.hostname

    if (hostname === "api.example.com") {
      const session = await authenticatePrivate(request, env)
      if (!session) return json({ error: "Unauthorized" }, 401)
```

```
      return handlePrivateApi(request, env, ctx, session)
    }

    if (hostname === "public-api.example.com") {
      const apiContext = await authenticatePublicApiKey(request, env)
      if (!apiContext) return json({ error: "Invalid API key" }, 401)
      return handlePublicApi(request, env, ctx, apiContext)
    }

    return json({ error: "Unknown host" }, 404)
  },
} satisfies ExportedHandler<Env>
```

5. Authentication Flows

5.1 Private API (Nuxt) – Session JWT in Cookie

The private API is used only by your own Nuxt app. Authentication is via a JWT stored in an HttpOnly cookie named session on the api.example.com hostname (or .example.com domain).
Login flow:
1) User submits email/password from the Nuxt app to POST /internal/auth/login on api.example.com.
2) Worker verifies credentials with Prisma.
3) Worker signs a JWT with sub = user.id and role = user.role using env.JWT_SECRET.
4) Worker returns a response with Set-Cookie: session=...; HttpOnly; Secure; SameSite=Lax; Domain=.example.com.
5) Subsequent requests from Nuxt include this cookie and are authenticated by authenticatePrivate.

```
// src/authPrivate.ts
import * as jose from "jose"
import type { Env } from "./prismaClient"

export type Session = { userId: string; role: string }

export async function authenticatePrivate(request: Request, env: Env): Promise<Session | null> {
  const cookieHeader = request.headers.get("Cookie") || ""
  const match = cookieHeader.match(/session=([^;]+)/)
  if (!match) return null

  const token = decodeURIComponent(match[1])

  try {
    const { payload } = await jose.jwtVerify(
      token,
      new TextEncoder().encode(env.JWT_SECRET),
    )
    return {
      userId: String(payload.sub),
      role: String(payload.role ?? "user"),
    }
  } catch {
    return null
  }
}
```

5.2 Public API – API Keys

The public API is accessed by third-party clients via API keys. Keys are created and managed via your internal dashboard (using the private API). Only a hash of each key is stored in the database to avoid leaking secrets.
• Request header format: Authorization: ApiKey <key>.
• Worker hashes the presented key and looks it up in the ApiKey table.
• If found and not disabled, the request is associated with the owning user/account and the scopes stored on the record (for example, ["read_products"]).

```
// src/authPublic.ts
import { createPrisma, Env } from "./prismaClient"

export type ApiContext = { apiKeyId: string; userId: string; scopes: string[] }

export async function hashApiKey(raw: string): Promise<string> {
  // Example: HMAC-SHA256 with a server-side secret, bcrypt, or argon2.
  // Implementation detail left to the developer.
  return raw // placeholder
}

export async function authenticatePublicApiKey(
  request: Request,
  env: Env,
): Promise<ApiContext | null> {
```

```
  const auth = request.headers.get("Authorization") || ""
  const m = auth.match(/^ApiKey\s+(.+)$/i)
  if (!m) return null

  const apiKey = m[1].trim()
  if (!apiKey) return null

  const keyHash = await hashApiKey(apiKey)

  const prisma = createPrisma(env)
  const record = await prisma.apiKey.findUnique({
    where: { keyHash },
  })

  if (!record || record.disabledAt) return null

  return {
    apiKeyId: record.id,
    userId: record.userId,
    scopes: record.scopes,
  }
}
```

6. Example Route Handlers

6.1 Private: List Products

The private products endpoint is used by the Nuxt app to display products in the internal UI (for example, with admin-only fields or draft listings).

```
// src/routes/private/products.ts
import { createPrisma, Env } from "../../prismaClient"
import type { Session } from "../../authPrivate"

function json(body: unknown, status = 200): Response {
  return new Response(JSON.stringify(body), {
    status,
    headers: { "Content-Type": "application/json" },
  })
}

export async function listPrivateProducts(
  request: Request,
  env: Env,
  session: Session,
): Promise<Response> {
  const url = new URL(request.url)
  const page = Number(url.searchParams.get("page") || "1")
  const pageSize = Number(url.searchParams.get("pageSize") || "20")

  const prisma = createPrisma(env)

  const [items, total] = await Promise.all([
    prisma.product.findMany({
      skip: (page - 1) * pageSize,
      take: pageSize,
      orderBy: { createdAt: "desc" },
    }),
    prisma.product.count(),
  ])

  return json({ data: items, total, page, pageSize })
}
```

6.2 Public: List Products

The public products endpoint returns only the fields meant for third-party consumption and enforces whatever "public visibility" rules you decide (for example, only published products).

```
// src/routes/public/products.ts
import { createPrisma, Env } from "../../prismaClient"
import type { ApiContext } from "../../authPublic"

function json(body: unknown, status = 200): Response {
  return new Response(JSON.stringify(body), {
    status,
    headers: { "Content-Type": "application/json" },
  })
}

export async function listPublicProducts(
  request: Request,
  env: Env,
  api: ApiContext,
): Promise<Response> {
  const url = new URL(request.url)
  const page = Number(url.searchParams.get("page") || "1")
  const pageSize = Number(url.searchParams.get("pageSize") || "20")

  const prisma = createPrisma(env)

  const [items, total] = await Promise.all([
    prisma.product.findMany({
```

```
      skip: (page - 1) * pageSize,
      take: pageSize,
      orderBy: { createdAt: "desc" },
      where: {
        // add public visibility filters here (e.g. published = true)
      },
      select: {
        id: true,
        title: true,
        description: true,
        priceCents: true,
        // omit any internal-only fields
      },
    }),
    prisma.product.count({
      where: {
        // same filter as above
      },
    }),
  ])

  return json({ data: items, total, page, pageSize })
}
```

7. Nuxt Integration

7.1 Runtime Config and Environment

Nuxt needs to know where to call the private API. This is done via runtime config. In production, NUXT_PUBLIC_API_BASE_URL should be set to https://api.example.com.

```
// nuxt.config.ts
export default defineNuxtConfig({
  runtimeConfig: {
    public: {
      apiBaseUrl: process.env.NUXT_PUBLIC_API_BASE_URL || "https://api.example.com",
    },
  },
})
```

7.2 Composable for Private API Calls

This composable wraps $fetch and always sends cookies (credentials: "include") so that the Worker can authenticate the user via the session cookie.

```
// composables/useApi.ts
export const useApi = () => {
  const config = useRuntimeConfig()
  const baseURL = config.public.apiBaseUrl

  const get = <T>(url: string, options: any = {}) =>
    $fetch<T>(url, {
      baseURL,
      credentials: "include",
      ...options,
    })

  const post = <T>(url: string, body: any, options: any = {}) =>
    $fetch<T>(url, {
      method: "POST",
      baseURL,
      body,
      credentials: "include",
      ...options,
    })

  return { get, post }
}
```

7.3 Nuxt Page Example: Products List

```
// pages/products.vue
<script setup lang="ts">
const { get } = useApi()

const { data: products, error } = await useAsyncData("products", () =>
  get("/internal/products", {
    query: { page: 1, pageSize: 20 },
  }),
)
</script>

<template>
  <div>
    <h1>Products</h1>
    <div v-if="error">Failed to load products</div>
    <ul v-else-if="products">
      <li v-for="p in products.data" :key="p.id">
        {{ p.title }} - {{ p.priceCents / 100 }} USD
      </li>
    </ul>
  </div>
```

```
</template>
```

7.4 Nuxt Login Flow (High-Level)

1) User fills in email/password in a Nuxt form.
2) Nuxt posts to POST /internal/auth/login on api.example.com using useApi().post.
3) Worker verifies credentials, sets the session cookie, and returns a 200 OK.
4) On success, Nuxt navigates the user to a protected page. Subsequent calls automatically include the cookie.

8. Local Development & Deployment

8.1 Local Development Workflow

1) Configure DATABASE_URL in .env to point at a development RDS instance (or local Postgres).
2) Run npm run prisma:migrate to apply schema changes.
3) Run npm run prisma:generate to update the client.
4) Start the Worker locally with npx wrangler dev.
5) In the Nuxt project, set NUXT_PUBLIC_API_BASE_URL to the local wrangler dev URL (or use host rewrites).
6) Run npm run dev in the Nuxt project and develop against the Worker.

8.2 Deployment Checklist

Database:
• Production RDS instance created and secured.
• Prisma migrations run against production (manually or via CI).

Cloudflare:
• Hyperdrive configured and bound as HYPERDRIVE.
• Worker deployed with compatible compatibility_date and nodejs_compat flag.
• Routes configured for api.example.com and public-api.example.com.
• Secrets (JWT_SECRET, any hashing secrets) configured as Worker environment variables.

Nuxt:
• NUXT_PUBLIC_API_BASE_URL set to https://api.example.com in production env.
• Nuxt deployed (for example, to Vercel, Netlify, or Cloudflare Pages).

Security & Observability:
• Rate limiting and logging implemented for both private and public APIs.
• Access logs and metrics monitored for anomalies.
• Database performance (connections, slow queries, storage) monitored in RDS.

This guide should give a developer enough information to build and operate the API service using Cloudflare Workers, Hyperdrive, and Prisma, and to integrate it cleanly with a Nuxt 3 frontend and a public third-party API surface.