**Agent Minder Payment System Design Document**

This document outlines the design and implementation of **Agent Minder**, a payment system for managing AI agents for users, tailored for **AI4Families**. The system is a multi-tenant platform supporting three user roles: **System Admin**, **Account Owner**, and **End User**. It handles payments, user management, and group management, integrating with a third-party payment gateway (Stripe) and adhering to modern development practices.

**1. Introduction**

**Purpose**

Agent Minder facilitates the management and payment processing of AI agents for families under organizational accounts (e.g., school districts or churches). It provides role-based access, group management, and a seamless payment experience, charging End Users $8 monthly without proration.

**Requirements**

* **User Roles and Permissions**:
  + **System Admin**: Full access to manage accounts, users, and system settings.
  + **Account Owner**: Manages their organization's settings, users, and data.
  + **End User**: Limited access to select and use AI agents.
* **Core Features**:
  + User management with role-based access control (RBAC).
  + Role-specific dashboards with analytics.
  + Account-specific data management.
  + Notifications for updates and alerts.
  + Reporting and analytics.
  + Audit logging for compliance and security.

**2. System Architecture**

**Tech Stack**

* **Backend**:
  + **Server & Routing**: Hono
  + **Database ORM**: Drizzle with PostgreSQL
  + **Authentication**: Clerk
  + **Validation**: Zod with zValidator
* **Frontend**:
  + **Framework**: Next.js 15 (App Router)
  + **Styling**: Tailwind CSS, shadcn/ui, Radix
  + **Data Fetching**: React Query (TanStack Query)
* **Payment Processing**: Stripe
* **Infrastructure**: Cloudflare Workers (API), Cloudflare Pages (Frontend)
* **Package Manager**: pnpm
* **Runtime**: Bun

**Project Structure**

text

CollapseWrapCopy

apps/

├── api/

│ ├── src/

│ │ ├── modules/

│ │ │ ├── users/

│ │ │ │ ├── users.routes.ts

│ │ │ │ └── users.service.ts

│ │ │ ├── payments/

│ │ │ │ ├── payments.routes.ts

│ │ │ │ └── payments.service.ts

│ │ ├── pkg/

│ │ └── index.ts

└── web/

└── src/

├── app/

│ ├── (dashboard)/

│ │ ├── page.tsx

│ │ └── \_components/

│ └── (auth)/

├── components/

│ ├── ui/

│ └── layout/

└── api/

├── users.api.ts

└── agents.api.ts

packages/

└── db/

├── src/

│ ├── schema.ts

│ ├── types.ts

│ └── index.ts

**3. Database Design**

The database uses PostgreSQL with Drizzle ORM, following a modular schema design.

**Schema Definitions**

typescript

CollapseWrapCopy

*// packages/db/src/schema.ts*

import { pgTable, varchar, text, numeric } from "drizzle-orm/pg-core";

import { newId } from "@repo/id";

*// Users table*

export const users = pgTable("users", {

id: varchar("id", { length: 255 }).primaryKey().$defaultFn(() => newId("user")),

email: text("email").notNull().unique(),

password: text("password").notNull(),

role: text("role").notNull().$type<"system\_admin" | "account\_owner" | "end\_user">(),

accountId: varchar("account\_id", { length: 255 }).references(() => accounts.id),

});

*// Accounts table*

export const accounts = pgTable("accounts", {

id: varchar("id", { length: 255 }).primaryKey().$defaultFn(() => newId("account")),

name: text("name").notNull(),

monthlyFee: numeric("monthly\_fee").notNull(),

workspaceId: text("workspace\_id").notNull(), *// MindStudio integration*

});

*// Groups table*

export const groups = pgTable("groups", {

id: varchar("id", { length: 255 }).primaryKey().$defaultFn(() => newId("group")),

name: text("name").notNull(),

accountId: varchar("account\_id", { length: 255 }).notNull().references(() => accounts.id),

});

*// Subscriptions table*

export const subscriptions = pgTable("subscriptions", {

id: varchar("id", { length: 255 }).primaryKey().$defaultFn(() => newId("subscription")),

userId: varchar("user\_id", { length: 255 }).notNull().references(() => users.id),

groupId: varchar("group\_id", { length: 255 }).notNull().references(() => groups.id),

paymentGatewaySubscriptionId: text("payment\_gateway\_subscription\_id").notNull(),

status: text("status").notNull().$type<"active" | "inactive">(),

});

*// AI Agents table*

export const aiAgents = pgTable("ai\_agents", {

id: varchar("id", { length: 255 }).primaryKey().$defaultFn(() => newId("agent")),

name: text("name").notNull(),

description: text("description"),

});

*// User-Agent Selections table*

export const userAgentSelections = pgTable("user\_agent\_selections", {

userId: varchar("user\_id", { length: 255 }).notNull().references(() => users.id),

agentId: varchar("agent\_id", { length: 255 }).notNull().references(() => aiAgents.id),

primaryKey: ["userId", "agentId"],

});

**Types**

typescript

CollapseWrapCopy

*// packages/db/src/types.ts*

import { InferSelectModel, InferInsertModel } from "drizzle-orm";

import { users, accounts, groups, subscriptions, aiAgents } from "./schema";

export type User = InferSelectModel<typeof users>;

export type NewUser = InferInsertModel<typeof users>;

export type Account = InferSelectModel<typeof accounts>;

export type NewAccount = InferInsertModel<typeof accounts>;

export type Group = InferSelectModel<typeof groups>;

export type NewGroup = InferInsertModel<typeof groups>;

export type Subscription = InferSelectModel<typeof subscriptions>;

export type NewSubscription = InferInsertModel<typeof subscriptions>;

export type AiAgent = InferSelectModel<typeof aiAgents>;

**4. API Design**

The API is built with Hono, using Clerk for authentication and Zod for validation.

**User Management Service**

typescript

CollapseWrapCopy

*// apps/api/src/modules/users/users.service.ts*

import { db, users, accounts, groups, subscriptions } from "@repo/db";

import stripe from "stripe";

import { hashPassword } from "@/pkg/util/security";

import { clerk } from "@clerk/clerk-sdk-node";

const stripeClient = new stripe("your\_stripe\_secret");

export const userService = {

async createAccountOwner({ name, email, password, groups: groupNames, paymentInfo }: {

name: string;

email: string;

password: string;

groups: string[];

paymentInfo: object;

}) {

const account = await db.insert(accounts).values({

name,

monthlyFee: "0", *// Configurable later*

workspaceId: "mindstudio\_workspace\_id",

}).returning();

const user = await db.insert(users).values({

email,

password: hashPassword(password),

role: "account\_owner",

accountId: account[0].id,

}).returning();

await clerk.users.updateUser(user[0].id, { publicMetadata: { role: "account\_owner" } });

for (const groupName of groupNames) {

await db.insert(groups).values({ name: groupName, accountId: account[0].id });

}

const subscription = await stripeClient.subscriptions.create({

customer: (paymentInfo as any).customerId,

items: [{ price: "price\_id\_account\_owner" }],

});

return { account: account[0], user: user[0], subscription };

},

async createEndUser({ email, password, groupId, paymentInfo }: {

email: string;

password: string;

groupId: string;

paymentInfo: object;

}) {

const group = await db.select({ accountId: groups.accountId }).from(groups).where(eq(groups.id, groupId));

const user = await db.insert(users).values({

email,

password: hashPassword(password),

role: "end\_user",

accountId: group[0].accountId,

}).returning();

await clerk.users.updateUser(user[0].id, { publicMetadata: { role: "end\_user" } });

const subscription = await stripeClient.subscriptions.create({

customer: (paymentInfo as any).customerId,

items: [{ price: "price\_id\_end\_user" }], *// $8/month*

});

await db.insert(subscriptions).values({

userId: user[0].id,

groupId,

paymentGatewaySubscriptionId: subscription.id,

status: "active",

});

return { user: user[0], subscription };

},

};

**User Routes**

typescript

CollapseWrapCopy

*// apps/api/src/modules/users/users.routes.ts*

import { Hono } from "hono";

import { auth, requireAuth } from "@/pkg/middleware/clerk-auth";

import { userService } from "./users.service";

import { zValidator } from "@/pkg/util/validator-wrapper";

import { createInsertSchema } from "drizzle-zod";

const userInsertSchema = createInsertSchema(users).omit({ id: true, accountId: true });

const accountInsertSchema = createInsertSchema(accounts).omit({ id: true });

export const userRoutes = new Hono()

.use(auth(), requireAuth)

.post("/accounts", zValidator("json", accountInsertSchema.extend({ email: z.string(), password: z.string(), groups: z.array(z.string()), paymentInfo: z.object({}) })), async (c) => {

const data = c.req.valid("json");

const result = await userService.createAccountOwner(data);

return c.json(result, 201);

})

.post("/users", zValidator("json", userInsertSchema.extend({ groupId: z.string(), paymentInfo: z.object({}) })), async (c) => {

const data = c.req.valid("json");

const result = await userService.createEndUser(data);

return c.json(result, 201);

});

**Payment Processing Service**

typescript

CollapseWrapCopy

*// apps/api/src/modules/payments/payments.service.ts*

import { db, subscriptions } from "@repo/db";

import stripe from "stripe";

import { eq } from "drizzle-orm";

const stripeClient = new stripe("your\_stripe\_secret");

export const paymentService = {

async processMonthlyPayments() {

const activeSubs = await db.select().from(subscriptions).where(eq(subscriptions.status, "active"));

for (const sub of activeSubs) {

try {

await stripeClient.charges.create({

amount: 800, *// $8.00*

currency: "usd",

customer: sub.paymentGatewaySubscriptionId,

});

} catch (error) {

await db.update(subscriptions).set({ status: "inactive" }).where(eq(subscriptions.id, sub.id));

*// Notify user (implementation omitted)*

}

}

},

};

**Webhook Routes**

typescript

CollapseWrapCopy

*// apps/api/src/modules/payments/payments.routes.ts*

import { Hono } from "hono";

import stripe from "stripe";

import { db, subscriptions } from "@repo/db";

import { eq } from "drizzle-orm";

const stripeClient = new stripe("your\_stripe\_secret");

export const webhookRoutes = new Hono()

.post("/stripe", async (c) => {

const sig = c.req.header("stripe-signature");

const body = await c.req.text();

try {

const event = stripeClient.webhooks.constructEvent(body, sig!, "your\_webhook\_secret");

if (event.type === "invoice.payment\_failed") {

const subscriptionId = (event.data.object as any).subscription;

await db.update(subscriptions).set({ status: "inactive" }).where(eq(subscriptions.paymentGatewaySubscriptionId, subscriptionId));

}

return c.json({ received: true });

} catch (err) {

return c.json({ error: "Webhook Error" }, 400);

}

});

**5. Frontend Design**

**Account Owner Dashboard**

tsx

CollapseWrapCopy

*// apps/web/src/app/(dashboard)/account-owner/page.tsx*

import { useQuery } from "@tanstack/react-query";

import { getGroups, getUsers } from "@/api/account.api";

export default function AccountOwnerDashboard() {

const { data: groups, isLoading: groupsLoading } = useQuery({

queryKey: ["groups"],

queryFn: getGroups,

});

const { data: users, isLoading: usersLoading } = useQuery({

queryKey: ["users"],

queryFn: getUsers,

});

if (groupsLoading || usersLoading) return <div>Loading...</div>;

return (

<div className="container mx-auto p-4">

<h1 className="text-2xl font-bold">Account Owner Dashboard</h1>

<section>

<h2 className="text-xl">Groups</h2>

<ul>{groups?.map((group) => <li key={group.id}>{group.name}</li>)}</ul>

</section>

<section>

<h2 className="text-xl">Users</h2>

<ul>{users?.map((user) => <li key={user.id}>{user.email}</li>)}</ul>

</section>

</div>

);

}

**AI Agent Catalogue (End User)**

tsx

CollapseWrapCopy

*// apps/web/src/app/(dashboard)/end-user/\_components/agent-catalogue.tsx*

import { useQuery, useMutation } from "@tanstack/react-query";

import { getAgents, selectAgents } from "@/api/agents.api";

import { Button } from "@/components/ui/button";

export function AgentCatalogue() {

const { data: agents, isLoading } = useQuery({

queryKey: ["agents"],

queryFn: getAgents,

});

const mutation = useMutation({ mutationFn: selectAgents });

if (isLoading) return <div>Loading...</div>;

return (

<div className="container mx-auto p-4">

<h1 className="text-2xl font-bold">AI Agents</h1>

<ul className="space-y-2">

{agents?.map((agent) => (

<li key={agent.id} className="flex items-center justify-between">

<span>{agent.name}</span>

<Button onClick={() => mutation.mutate({ agentId: agent.id })}>Select</Button>

</li>

))}

</ul>

</div>

);

}

**Frontend API Integration**

typescript

CollapseWrapCopy

*// apps/web/src/api/users.api.ts*

import { apiRpc, getApiClient, InferRequestType } from "./client";

const $createAccountOwner = apiRpc.users.accounts.$post;

const $createEndUser = apiRpc.users.$post;

export async function createAccountOwner(params: InferRequestType<typeof $createAccountOwner>["json"]) {

const client = await getApiClient();

const response = await client.users.accounts.$post({ json: params });

return response.json();

}

export async function createEndUser(params: InferRequestType<typeof $createEndUser>["json"]) {

const client = await getApiClient();

const response = await client.users.$post({ json: params });

return response.json();

}

**6. Payment Processing**

* **Integration**: Stripe is used for subscription management and payment processing.
* **Monthly Billing**: A scheduled task (e.g., Cloudflare Workers cron) triggers processMonthlyPayments at month-end.
* **Webhook Handling**: Stripe webhooks update subscription status on payment failures.

**7. Security Considerations**

* **Authentication**: Clerk handles user authentication and RBAC via public metadata.
* **Data Protection**: Sensitive data (e.g., passwords) is hashed; HTTPS is enforced.
* **Payment Security**: Stripe tokenizes payment info client-side, ensuring PCI compliance.

**8. Deployment and Infrastructure**

* **API**: Deployed on Cloudflare Workers for scalability.
* **Frontend**: Hosted on Cloudflare Pages.
* **CI/CD**: GitHub Actions automates linting, testing, and deployment.

**9. Testing Strategy**

* **Unit Tests**: Use Vitest for service logic (e.g., userService).
* **Integration Tests**: Test API endpoints with a mock database.
* **E2E Tests**: Use Playwright for sign-up and payment flows.

**10. Future Enhancements**

* Multi-group support for End Users.
* Advanced analytics dashboards.
* Additional payment gateway options.

This design provides a scalable, secure, and user-friendly payment system for Agent Minder, aligning with the provided development practices and requirements. The pseudo code is ready for post-processing in a coding editor with Claude 3.7 support.