File: /home/oai/share/dayboard/backend/cmd/server/main.go

```
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/google/uuid"

    "dayboard/backend/internal/db"
    "dayboard/backend/internal/store"
    "dayboard/backend/internal/estimate"
    "dayboard/backend/internal/commute"
)

// main is the entrypoint for the DayBoard backend. It sets up the HTTP router
// and starts listening on the port specified in the PORT environment variable.
func main() {
    // Determine the port to listen on. Default to 8080 if not set.
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    // Initialize DB connection. Fatal if cannot connect.
    database := db.New()
    defer database.Close()

    // Use Gin in release mode for production. Gin automatically logs requests.
    gin.SetMode(gin.ReleaseMode)
    router := gin.New()
    router.Use(gin.Logger(), gin.Recovery())

    // Register health check endpoint for uptime monitoring.
    router.GET("/healthz", func(c *gin.Context) {
        c.String(http.StatusOK, "ok")
    })

    // Mount API routes under /api/v1. Handlers are stubbed for now and
    // should be implemented in the internal/http package.
    api := router.Group("/api/v1")
    {
        api.GET("/agenda/today", func(c *gin.Context) {
            // In a production system you'd derive the user ID from the
            // authenticated session. For demonstration we read a query param.
            userParam := c.Query("user_id")
            userID := uuid.Nil
            if userParam != "" {
                if uid, err := uuid.Parse(userParam); err == nil {
                    userID = uid
                }
```

```
    }
    // Determine start and end of today in UTC based on the server's time.
    now := time.Now().UTC()
    y, m, d := now.Date()
    loc := now.Location()
    startOfDay := time.Date(y, m, d, 0, 0, 0, 0, loc)
    endOfDay := startOfDay.Add(24 * time.Hour)
    events, err := store.GetTodayEvents(c.Request.Context(), database, userID, startOfDay, endOfDay)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }
    // Transform events into response objects. Gin will marshal the
    // time.Time fields as RFC3339 strings.
    c.JSON(http.StatusOK, events)
})

api.GET("/subs", func(c *gin.Context) {
    userParam := c.Query("user_id")
    userID := uuid.Nil
    if uid, err := uuid.Parse(userParam); err == nil {
        userID = uid
    }
    subs, err := store.GetSubscriptions(c.Request.Context(), database, userID)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }
    c.JSON(http.StatusOK, subs)
})

api.POST("/subs", func(c *gin.Context) {
    userParam := c.Query("user_id")
    userID := uuid.Nil
    if uid, err := uuid.Parse(userParam); err == nil {
        userID = uid
    }
    var req store.Subscription
    if err := c.BindJSON(&req); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    sub, err := store.CreateSubscription(c.Request.Context(), database, userID, req)
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    c.JSON(http.StatusCreated, sub)
})

api.POST("/estimate/taxes", func(c *gin.Context) {
    // Parse payload {incomeCents,state,filingStatus,payFreq,termWeeks}
    var body struct {
        IncomeCents int    `json:"incomeCents"`
        State       string `json:"state"`
        FilingStatus string `json:"filingStatus"`
        PayFreq     string `json:"payFreq"`
        TermWeeks   int    `json:"termWeeks"`
```

```
        }
        if err := c.BindJSON(&body); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        // Use current year for taxes. In production you might allow specifying.
        year := time.Now().Year()
        res, err := estimate.EstimateTaxes(c.Request.Context(), database, body.IncomeCents, body.State, body.FilingStatus, year,
body.PayFreq, body.TermWeeks)
        if err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        c.JSON(http.StatusOK, res)
    })

    api.GET("/commute/estimate", func(c *gin.Context) {
        origin := c.Query("from")
        destination := c.Query("to")
        // Example surge parameter, default to 1.0 (no surge)
        surge := 1.0
        if s := c.Query("surge"); s != "" {
            if v, err := strconv.ParseFloat(s, 64); err == nil {
                surge = v
            }
        }
        // For demonstration, fetch cost model from DB based on city. Here
        // we simply hardcode a generic model. In production, you would
        // select by city/state.
        baseCents := 200   // $2 base fare
        perMileCents := 150 // $1.50 per mile
        perMinCents := 25  // $0.25 per minute
        est, err := commute.EstimateCommute(c.Request.Context(), origin, destination, baseCents, perMileCents, perMinCents, surge)
        if err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        c.JSON(http.StatusOK, est)
    })

    api.GET("/profile", func(c *gin.Context) {
        userParam := c.Query("user_id")
        userID := uuid.Nil
        if uid, err := uuid.Parse(userParam); err == nil {
            userID = uid
        }
        prof, err := store.GetProfile(c.Request.Context(), database, userID)
        if err != nil {
            c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }
        if prof == nil {
            c.JSON(http.StatusOK, gin.H{})
            return
        }
        c.JSON(http.StatusOK, prof)
    })
```

```
        api.POST("/profile", func(c *gin.Context) {
            userParam := c.Query("user_id")
            userID := uuid.Nil
            if uid, err := uuid.Parse(userParam); err == nil {
                userID = uid
            }
            var prof store.Profile
            if err := c.BindJSON(&prof); err != nil {
                c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
                return
            }
            prof.UserID = userID
            if err := store.UpsertProfile(c.Request.Context(), database, prof); err != nil {
                c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
                return
            }
            c.JSON(http.StatusCreated, prof)
        })
    }

    // Start listening and serving requests. If an error occurs, log and exit.
    if err := router.Run(fmt.Sprintf(":" + port)); err != nil {
        log.Fatalf("failed to run server: %v", err)
    }
}

================================================================================

File: /home/oai/share/dayboard/backend/internal/db/db.go
--------------------------------------------------------------------------------
package db

import (
    "context"
    "database/sql"
    "fmt"
    "log"
    "os"

    _ "github.com/jackc/pgx/v5/stdlib"
)

// DB wraps a sql.DB instance and exposes helper methods for common database
// operations. All queries should be executed via prepared statements to
// mitigate SQL injection vulnerabilities. The connection string should be
// provided via the DATABASE_URL environment variable. The recommended format
// is a PostgreSQL URL, for example:
//   postgres://username:password@host:port/database
// When using Supabase, copy the connection string from your project's settings.
type DB struct {
    *sql.DB
}

// New creates a new DB connection pool. It reads the DATABASE_URL
// environment variable and opens a pooled connection using pgx's stdlib
// driver. If the variable is not set or the connection fails, the
// application will log and exit. The returned *DB should be closed
// gracefully on shutdown.
```

```
func New() *DB {
    dsn := os.Getenv("DATABASE_URL")
    if dsn == "" {
        log.Fatal("DATABASE_URL environment variable not set")
    }
    db, err := sql.Open("pgx", dsn)
    if err != nil {
        log.Fatalf("failed to open database: %v", err)
    }
    // Set connection pool parameters. Adjust these based on your hosting
    // environment's limits (e.g. Supabase free tier supports up to 10 connections).
    db.SetMaxOpenConns(5)
    db.SetMaxIdleConns(2)
    return &DB{db}
}

// Ping verifies a connection to the database can be established. It's a
// convenience method for health checks or startup verification.
func (d *DB) Ping(ctx context.Context) error {
    return d.DB.PingContext(ctx)
}

// Close gracefully closes the underlying sql.DB. Always call this on
// application shutdown to release connections back to the pool.
func (d *DB) Close() error {
    return d.DB.Close()
}
```

```
================================================================================
```

File: /home/oai/share/dayboard/backend/internal/store/store.go

```
--------------------------------------------------------------------------------
package store

import (
    "context"
    "database/sql"
    "errors"
    "time"

    "github.com/google/uuid"
    "github.com/jackc/pgx/v5"

    "dayboard/backend/internal/db"
)

// Event represents a calendar event stored in the database. It mirrors the
// columns of the calendar_events table and is returned to the API caller.
type Event struct {
    ID       uuid.UUID `json:"id"`
    Start    time.Time `json:"start"`
    End      time.Time `json:"end"`
    Title    string    `json:"title"`
    JoinURL  string    `json:"join_url"`
    Location string    `json:"location"`
}
```

```
// Subscription represents a recurring payment. AmountCents and cadence
// determine the billing schedule. NextDue may be nil if unknown.
type Subscription struct {
    ID           uuid.UUID `json:"id"`
    Merchant     string    `json:"merchant"`
    AmountCents  int       `json:"amount_cents"`
    CadenceDays  int       `json:"cadence_days"`
    NextDue      *time.Time `json:"next_due,omitempty"`
    Source       string    `json:"source"`
    IsActive     bool      `json:"is_active"`
}

// Profile holds user-specific settings used for tax and cost estimation.
// All monetary values are stored as cents to avoid floating point errors.
type Profile struct {
    UserID        uuid.UUID
    HomeAddr      string
    OfficeAddr    string
    City          string
    State         string
    HourlyCents   *int
    HoursPerWeek  *int
    StipendCents  *int
    PayFreq       string
    StartDate     *time.Time
    InOfficeDays  int
    FoodCostCents int
}

// GetTodayEvents returns all events for a user that start on the given day.
// The caller is responsible for passing startOfDay and endOfDay in UTC.
func GetTodayEvents(ctx context.Context, d *db.DB, userID uuid.UUID, startOfDay, endOfDay time.Time) ([]Event, error) {
    rows, err := d.QueryContext(ctx, `
        SELECT id, start_ts, end_ts, title, join_url, location
        FROM calendar_events
        WHERE user_id = $1
          AND start_ts >= $2
          AND start_ts < $3
        ORDER BY start_ts ASC
    `, userID, startOfDay, endOfDay)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    var events []Event
    for rows.Next() {
        var e Event
        var id string
        if err := rows.Scan(&id, &e.Start, &e.End, &e.Title, &e.JoinURL, &e.Location); err != nil {
            return nil, err
        }
        uid, _ := uuid.Parse(id)
        e.ID = uid
        events = append(events, e)
    }
    return events, rows.Err()
}
```

```
// GetSubscriptions returns all active subscriptions for a user.
func GetSubscriptions(ctx context.Context, d *db.DB, userID uuid.UUID) ([]Subscription, error) {
    rows, err := d.QueryContext(ctx, `
        SELECT id, merchant, amount_cents, cadence_days, next_due, source, is_active
        FROM subscriptions
        WHERE user_id = $1 AND is_active = true
        ORDER BY next_due ASC NULLS LAST
    `, userID)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    var subs []Subscription
    for rows.Next() {
        var s Subscription
        var id string
        var nextDue pgx.NullTime
        if err := rows.Scan(&id, &s.Merchant, &s.AmountCents, &s.CadenceDays, &nextDue, &s.Source, &s.IsActive); err != nil {
            return nil, err
        }
        s.ID, _ = uuid.Parse(id)
        if nextDue.Valid {
            t := nextDue.Time
            s.NextDue = &t
        }
        subs = append(subs, s)
    }
    return subs, rows.Err()
}

// CreateSubscription inserts a new manual subscription for the user. Plaid-detected
// subscriptions should be inserted via separate routines. Returns the created
// subscription or an error.
func CreateSubscription(ctx context.Context, d *db.DB, userID uuid.UUID, s Subscription) (*Subscription, error) {
    // Basic validation
    if s.Merchant == "" || s.AmountCents <= 0 || s.CadenceDays <= 0 {
        return nil, errors.New("invalid subscription fields")
    }
    id := uuid.New()
    _, err := d.ExecContext(ctx, `
        INSERT INTO subscriptions (id, user_id, merchant, amount_cents, cadence_days, next_due, source, is_active)
        VALUES ($1, $2, $3, $4, $5, $6, 'manual', true)
    `, id, userID, s.Merchant, s.AmountCents, s.CadenceDays, s.NextDue)
    if err != nil {
        return nil, err
    }
    s.ID = id
    s.Source = "manual"
    s.IsActive = true
    return &s, nil
}

// GetProfile retrieves the user's profile. If no profile exists, returns
// (nil, nil) to signal caller to create a default. Do not create default
// profiles automatically here to avoid unexpected writes.
func GetProfile(ctx context.Context, d *db.DB, userID uuid.UUID) (*Profile, error) {
    row := d.QueryRowContext(ctx, `
        SELECT home_addr, office_addr, city, state, hourly_cents, hours_per_week,
```

```
                stipend_cents, pay_freq, start_date, in_office_days, food_cost_cents
            FROM profiles WHERE user_id = $1
        `, userID)
        var p Profile
        p.UserID = userID
        var hourly, stipend sql.NullInt64
        var hours sql.NullInt32
        var start sql.NullTime
        if err := row.Scan(&p.HomeAddr, &p.OfficeAddr, &p.City, &p.State, &hourly, &hours, &stipend, &p.PayFreq, &start, &p.InOfficeDays,
&p.FoodCostCents); err != nil {
            if errors.Is(err, sql.ErrNoRows) {
                return nil, nil
            }
            return nil, err
        }
        if hourly.Valid {
            v := int(hourly.Int64)
            p.HourlyCents = &v
        }
        if hours.Valid {
            v := int(hours.Int32)
            p.HoursPerWeek = &v
        }
        if stipend.Valid {
            v := int(stipend.Int64)
            p.StipendCents = &v
        }
        if start.Valid {
            t := start.Time
            p.StartDate = &t
        }
        return &p, nil
    }

    // UpsertProfile inserts or updates a user's profile. If a profile does not
    // exist, one is created. Otherwise, the existing record is updated.
    func UpsertProfile(ctx context.Context, d *db.DB, p Profile) error {
        _, err := d.ExecContext(ctx, `
            INSERT INTO profiles (
                user_id, home_addr, office_addr, city, state, hourly_cents,
                hours_per_week, stipend_cents, pay_freq, start_date,
                in_office_days, food_cost_cents
            ) VALUES (
                $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12
            )
            ON CONFLICT (user_id) DO UPDATE SET
                home_addr = EXCLUDED.home_addr,
                office_addr = EXCLUDED.office_addr,
                city = EXCLUDED.city,
                state = EXCLUDED.state,
                hourly_cents = EXCLUDED.hourly_cents,
                hours_per_week = EXCLUDED.hours_per_week,
                stipend_cents = EXCLUDED.stipend_cents,
                pay_freq = EXCLUDED.pay_freq,
                start_date = EXCLUDED.start_date,
                in_office_days = EXCLUDED.in_office_days,
                food_cost_cents = EXCLUDED.food_cost_cents
        `, p.UserID, p.HomeAddr, p.OfficeAddr, p.City, p.State, p.HourlyCents,
```

```
        p.HoursPerWeek, p.StipendCents, p.PayFreq, p.StartDate,
        p.InOfficeDays, p.FoodCostCents)
    return err
}



================================================================================

File: /home/oai/share/dayboard/backend/internal/estimate/estimate.go

--------------------------------------------------------------------------------
package estimate

import (
    "context"
    "database/sql"
    "fmt"
    "time"

    "dayboard/backend/internal/db"
)

// TaxResult holds the computed tax amounts and net values for a given
// income, state and filing status. All monetary values are in cents.
type TaxResult struct {
    FederalCents       int `json:"federalCents"`
    StateCents         int `json:"stateCents"`
    FicaCents          int `json:"ficaCents"`
    PerPaycheckNetCents int `json:"perPaycheckNetCents"`
    TermNetCents       int `json:"termNetCents"`
}

// EstimateTaxes estimates U.S. federal, state, and FICA taxes for a given annual
// income (in cents). It looks up the progressive tax brackets stored in
// tax_tables_federal and tax_tables_state. FilingStatus must be either
// "single" or "married"; other values return an error. The year parameter
// allows supporting future/previous tax years. The result includes the
// after-tax take-home per paycheck over the given termWeeks.
func EstimateTaxes(ctx context.Context, d *db.DB, incomeCents int, state string, filingStatus string, year int, payFreq string, termWeeks
int) (*TaxResult, error) {
    // Determine standard deduction based on filing status.
    var stdDeduction int
    switch filingStatus {
    case "single":
        row := d.QueryRowContext(ctx, `SELECT DISTINCT std_deduction_single FROM tax_tables_federal WHERE year = $1 LIMIT 1`, year)
        if err := row.Scan(&stdDeduction); err != nil {
            return nil, fmt.Errorf("failed to fetch std deduction: %w", err)
        }
    case "married":
        // Not implemented: add support for married filing jointly.
        return nil, fmt.Errorf("married filing jointly not yet supported")
    default:
        return nil, fmt.Errorf("unsupported filing status: %s", filingStatus)
    }

    taxableIncome := incomeCents - stdDeduction
    if taxableIncome < 0 {
        taxableIncome = 0
    }
```

```
// Compute federal tax.
var federalTax int
rows, err := d.QueryContext(ctx, `
    SELECT bracket_low, bracket_high, rate_bps
    FROM tax_tables_federal WHERE year = $1
    ORDER BY bracket_low ASC
`, year)
if err != nil {
    return nil, err
}
defer rows.Close()
remaining := taxableIncome
for rows.Next() {
    var low, high, rateBps int
    if err := rows.Scan(&low, &high, &rateBps); err != nil {
        return nil, err
    }
    if remaining <= 0 {
        break
    }
    // Determine portion of income in this bracket.
    upperBound := high
    if high == 0 { // zero or null high implies no upper bound (top bracket)
        upperBound = taxableIncome
    }
    // Determine taxable amount in this bracket.
    segment := min(remaining, upperBound-low)
    federalTax += segment * rateBps / 10000 // rate_bps is basis points
    remaining -= segment
}
// Compute state tax. If state is unknown, assume zero.
var stateTax int
if state != "" {
    rows, err := d.QueryContext(ctx, `
        SELECT bracket_low, bracket_high, rate_bps
        FROM tax_tables_state WHERE year = $1 AND state = $2
        ORDER BY bracket_low ASC
    `, year, state)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    remaining = taxableIncome
    for rows.Next() {
        var low, high, rateBps int
        if err := rows.Scan(&low, &high, &rateBps); err != nil {
            return nil, err
        }
        if remaining <= 0 {
            break
        }
        upperBound := high
        if high == 0 {
            upperBound = taxableIncome
        }
        segment := min(remaining, upperBound-low)
        stateTax += segment * rateBps / 10000
        remaining -= segment
```

```
        }
    }
    // Estimate FICA (Social Security + Medicare) at 7.65% for simplicity.
    ficaTax := incomeCents * 765 / 10000
    // Determine number of paychecks in the term.
    var checks int
    switch payFreq {
    case "weekly":
        checks = termWeeks
    case "biweekly":
        checks = termWeeks / 2
    case "monthly":
        // Approximate 4 weeks per month. Multiply by termWeeks/4.
        checks = termWeeks / 4
    default:
        checks = termWeeks / 2
    }
    totalTax := federalTax + stateTax + ficaTax
    netAnnual := incomeCents - totalTax
    // Net per paycheck. Avoid division by zero.
    perPay := 0
    if checks > 0 {
        perPay = netAnnual / checks
    }
    result := &TaxResult{
        FederalCents:        federalTax,
        StateCents:          stateTax,
        FicaCents:           ficaTax,
        PerPaycheckNetCents: perPay,
        TermNetCents:        netAnnual,
    }
    return result, nil
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

```
================================================================================
```

File: /home/oai/share/dayboard/backend/internal/commute/commute.go

```
--------------------------------------------------------------------------------
package commute

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "os"
    "time"
)
```

```
// Estimate represents the output of a commute cost estimate. Distances and
// durations are included along with low/high cost estimates (in cents).
type Estimate struct {
    DistanceMiles   float64 `json:"distanceMiles"`
    DurationMinutes float64 `json:"durationMinutes"`
    EstCostLowCents int     `json:"estCostLowCents"`
    EstCostHighCents int    `json:"estCostHighCents"`
}


// estimateDistance calls the Google Distance Matrix API to compute the
// distance and duration between two addresses. It returns miles and
// minutes. The API key must be set via MAPS_API_KEY environment
// variable. This function is blocking and should be called from a
// goroutine or asynchronous context if latency is a concern.
func estimateDistance(ctx context.Context, origin, destination string) (float64, float64, error) {
    apiKey := os.Getenv("MAPS_API_KEY")
    if apiKey == "" {
        return 0, 0, fmt.Errorf("MAPS_API_KEY environment variable not set")
    }
    endpoint := "https://maps.googleapis.com/maps/api/distancematrix/json"
    params := url.Values{}
    params.Set("origins", origin)
    params.Set("destinations", destination)
    params.Set("units", "imperial")
    params.Set("key", apiKey)
    reqURL := fmt.Sprintf("%s?%s", endpoint, params.Encode())
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, reqURL, nil)
    if err != nil {
        return 0, 0, err
    }
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return 0, 0, err
    }
    defer resp.Body.Close()
    var dmResp struct {
        Rows []struct {
            Elements []struct {
                Distance struct {
                    Value int `json:"value"` // meters
                    Text  string `json:"text"`
                } `json:"distance"`
                Duration struct {
                    Value int `json:"value"` // seconds
                    Text  string `json:"text"`
                } `json:"duration"`
                Status string `json:"status"`
            } `json:"elements"`
        } `json:"rows"`
        Status string `json:"status"`
    }
    if err := json.NewDecoder(resp.Body).Decode(&dmResp); err != nil {
        return 0, 0, err
    }
    if dmResp.Status != "OK" || len(dmResp.Rows) == 0 || len(dmResp.Rows[0].Elements) == 0 {
        return 0, 0, fmt.Errorf("distance matrix API error: %s", dmResp.Status)
    }
    elem := dmResp.Rows[0].Elements[0]
```

```
    if elem.Status != "OK" {
        return 0, 0, fmt.Errorf("distance matrix element error: %s", elem.Status)
    }
    // Convert meters to miles and seconds to minutes.
    miles := float64(elem.Distance.Value) * 0.000621371
    minutes := float64(elem.Duration.Value) / 60.0
    return miles, minutes, nil
}

// EstimateCommute calculates the commute cost between origin and destination
// given a surge factor. The cost is computed based on a simple model:
// base fare + per-mile * miles + per-minute * minutes. The cost model
// parameters should be stored in a DB table (city_cost_models) and loaded
// by the caller. For demonstration, this function accepts the cost
// parameters directly.
func EstimateCommute(ctx context.Context, origin, destination string, baseCents, perMileCents, perMinCents int, surge float64) (*Estimate,
error) {
    miles, minutes, err := estimateDistance(ctx, origin, destination)
    if err != nil {
        return nil, err
    }
    low := float64(baseCents) + float64(perMileCents)*miles + float64(perMinCents)*minutes
    high := low * surge
    return &Estimate{
        DistanceMiles:    miles,
        DurationMinutes:  minutes,
        EstCostLowCents:  int(low),
        EstCostHighCents: int(high),
    }, nil
}



================================================================================

File: /home/oai/share/dayboard/backend/migrations/0001_create_tables.sql
--------------------------------------------------------------------------------
-- Migration to create base tables for DayBoard.

-- Users table stores application users. OAuth tokens are stored in a
-- separate table to maintain referential integrity and allow multiple
-- tokens per user.
CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email TEXT UNIQUE NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- OAuth tokens store encrypted access and refresh tokens for each
-- provider. A user may have multiple tokens if connected to
-- Google, Microsoft, and Plaid. Encryption should be handled
-- application-side before inserting into this table.
CREATE TABLE IF NOT EXISTS oauth_tokens (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    provider TEXT NOT NULL,
    access_token_enc BYTEA NOT NULL,
    refresh_token_enc BYTEA,
    scopes TEXT[] NOT NULL,
```

```
    expiry TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Calendar events store normalized events for a user's agenda. Events
-- are fetched from external providers and cached here to avoid
-- hitting provider APIs on each request.
CREATE TABLE IF NOT EXISTS calendar_events (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    source TEXT NOT NULL,
    ext_id TEXT NOT NULL,
    start_ts TIMESTAMPTZ NOT NULL,
    end_ts TIMESTAMPTZ NOT NULL,
    title TEXT,
    join_url TEXT,
    location TEXT,
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    UNIQUE (user_id, source, ext_id)
);

-- Subscriptions table stores recurring charges discovered via Plaid
-- transactions or entered manually by the user. Cadence is stored in
-- days for simplicity (e.g., 30 for monthly, 7 for weekly).
CREATE TABLE IF NOT EXISTS subscriptions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    merchant TEXT NOT NULL,
    amount_cents INT NOT NULL,
    cadence_days INT NOT NULL,
    next_due DATE,
    source TEXT NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Transactions table stores raw transaction data from Plaid or CSV
-- imports. This table may be used to re-run recurring detection when
-- algorithms improve. Raw data is stored in JSONB for flexibility.
CREATE TABLE IF NOT EXISTS transactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    source TEXT NOT NULL,
    ext_id TEXT,
    txn_date DATE NOT NULL,
    merchant TEXT,
    amount_cents INT NOT NULL,
    category TEXT,
    raw JSONB
);

-- Profiles table stores per-user settings used by the estimator. Each
-- user has a single profile row.
CREATE TABLE IF NOT EXISTS profiles (
    user_id UUID PRIMARY KEY REFERENCES users(id) ON DELETE CASCADE,
    home_addr TEXT,
    office_addr TEXT,
    city TEXT,
```

```
    state TEXT,
    hourly_cents INT,
    hours_per_week INT,
    stipend_cents INT,
    pay_freq TEXT,
    start_date DATE,
    in_office_days INT DEFAULT 3,
    food_cost_cents INT DEFAULT 1200
);

-- Federal tax tables store brackets and rates for a given year. Rates
-- are stored in basis points (bps) to avoid floating point issues.
CREATE TABLE IF NOT EXISTS tax_tables_federal (
    year INT NOT NULL,
    bracket_low INT NOT NULL,
    bracket_high INT NOT NULL,
    rate_bps INT NOT NULL,
    std_deduction_single INT NOT NULL,
    std_deduction_mfj INT NOT NULL
);

-- State tax tables store brackets and rates by state and year.
CREATE TABLE IF NOT EXISTS tax_tables_state (
    state TEXT NOT NULL,
    year INT NOT NULL,
    bracket_low INT NOT NULL,
    bracket_high INT NOT NULL,
    rate_bps INT NOT NULL,
    std_deduction_single INT NOT NULL
);

-- City cost models store base fare and per-unit costs for the commute
-- estimator. Costs are stored in cents for precision.
CREATE TABLE IF NOT EXISTS city_cost_models (
    city TEXT PRIMARY KEY,
    base_fare_cents INT NOT NULL,
    per_mile_cents INT NOT NULL,
    per_minute_cents INT NOT NULL
);

================================================================================

File: /home/oai/share/dayboard/backend/.env.example
--------------------------------------------------------------------------------
# Environment variables for the DayBoard backend.  Copy this file to
# `.env` and fill in the values marked TODO with the credentials from
# your external services.  The `.env` file should be kept private and
# should not be committed to version control.

# Application server port.  The default is 8080 if not set.
PORT=8080

# PostgreSQL connection string.  Supabase provides a `DB URL` on the
# project settings page.  It looks like:
#   postgresql://postgres:<db_password>@db.<ref>.supabase.co:5432/postgres?sslmode=require
# Replace the placeholder below with your Supabase database URL.
DATABASE_URL=postgresql://postgres:replace_me@dbhost:5432/postgres?sslmode=require
```

```
# Supabase project configuration.  You can find these values in the
# Supabase dashboard under `Project Settings` ? `API`.  The service key
# allows server?side access to your database via the Supabase client
# library.  Do not expose the service key to your client.
SUPABASE_URL=https://replace-me.supabase.co
SUPABASE_SERVICE_KEY=replace_me

# Google OAuth client configuration.  Create a Google Cloud project,
# enable the Google Calendar API and Google Maps Distance Matrix API,
# and create an OAuth 2.0 Client ID (type: Web application).  Add
# `http://localhost:8080/auth/google/callback` (or your production
# domain) as an authorized redirect URI.  Set the client ID and
# secret here.
GOOGLE_CLIENT_ID=replace_me
GOOGLE_CLIENT_SECRET=replace_me
# The redirect URI should match what you configured in Google Cloud.
GOOGLE_REDIRECT_URI=http://localhost:8080/auth/google/callback

# Plaid API configuration.  Sign up for a Plaid developer account at
# https://dashboard.plaid.com/, create a development client, and note
# the client ID and secret.  Use `PLAID_ENV=sandbox` for development
# and set the redirect URI to `http://localhost:8080/auth/plaid/callback`.
PLAID_CLIENT_ID=replace_me
PLAID_SECRET=replace_me
PLAID_ENV=sandbox
PLAID_REDIRECT_URI=http://localhost:8080/auth/plaid/callback

# Google Maps Distance Matrix API key.  Create an API key in your
# Google Cloud project and restrict it to the Distance Matrix API.
MAPS_API_KEY=replace_me

# JWT secret used to sign session tokens.  Generate a random string
# (e.g., using `openssl rand -hex 32`) and put it here.  This is used
# for authenticating requests from the SwiftUI client.
JWT_SECRET=replace_me

================================================================================

File: /home/oai/share/dayboard/client/DayBoardApp.swift
--------------------------------------------------------------------------------
import SwiftUI
import Combine

/// DayBoardApp is the entry point for the SwiftUI application. It sets up
/// a menu bar extra on macOS and an application scene on iOS. The menu
/// bar extra displays the next event, commute estimate, upcoming bills,
/// and pay outlook using data fetched from the backend.
@main
struct DayBoardApp: App {
    @StateObject private var viewModel = DayBoardViewModel()

    var body: some Scene {
        #if os(macOS)
        MenuBarExtra("DayBoard", systemImage: "calendar.badge.clock") {
            ContentView(viewModel: viewModel)
                .onAppear { viewModel.refresh() }
        }
        .menuBarExtraStyle(.window)
```

```
        #else
        WindowGroup {
            ContentView(viewModel: viewModel)
                .onAppear { viewModel.refresh() }
        }
        #endif
    }
}

/// ContentView lays out the user interface for the DayBoard menu bar and
/// application window. It displays the next meeting with a join button,
/// commute estimate, upcoming bills, and pay outlook. Buttons open
/// corresponding screens (not yet implemented).
struct ContentView: View {
    @ObservedObject var viewModel: DayBoardViewModel

    var body: some View {
        VStack(alignment: .leading, spacing: 12) {
            if let next = viewModel.nextEvent {
                VStack(alignment: .leading, spacing: 2) {
                    Text(next.title)
                        .font(.headline)
                    Text(next.start, style: .time)
                        .font(.subheadline)
                    if let url = next.joinURL {
                        Button("Join") {
                            #if os(macOS)
                            NSWorkspace.shared.open(url)
                            #endif
                        }
                    }
                }
            } else {
                Text("No more events today")
            }

            Divider()

            HStack {
                Text("Commute:")
                Spacer()
                Text(viewModel.commuteCost)
            }

            HStack {
                Text("Bills this week:")
                Spacer()
                Text(viewModel.billsThisWeek)
            }

            HStack {
                Text("Pay outlook:")
                Spacer()
                Text(viewModel.payOutlook)
            }

            Divider()
```

```
            Button("Open DayBoard") {
                viewModel.openMainWindow()
            }
        }
        .padding(10)
        .frame(maxWidth: 300)
    }
}

/// DayBoardViewModel orchestrates data fetching from the backend and
/// transforms it into view-friendly formats. It uses Combine to
/// asynchronously load today's agenda, commute estimate, subscriptions,
/// and pay outlook. Real network requests should be implemented here.
final class DayBoardViewModel: ObservableObject {
    // Published properties drive UI updates.
    @Published var nextEvent: DayBoardEvent?
    @Published var commuteCost: String = "?"
    @Published var billsThisWeek: String = "?"
    @Published var payOutlook: String = "?"

    private var cancellables = Set<AnyCancellable>()

    /// The base URL of the DayBoard backend. Replace this with your deployed
    /// backend URL when running in production. When testing locally you can
    /// leave the default value (assuming the Go server runs on port 8080).
    private let baseURL = URL(string: "http://localhost:8080/api/v1")!

    /// A hard-coded user ID for demonstration purposes. In a production app,
    /// you would generate this after user login and persist it (e.g. in the
    /// keychain). The Go backend expects a `user_id` query parameter on
    /// certain endpoints. Replace this with the UUID of the logged-in user.
    private let userID = "00000000-0000-0000-0000-000000000000"

    /// Refresh reloads all data required for the menu bar display.
    func refresh() {
        fetchAgenda()
        fetchSubscriptions()
        fetchCommuteEstimate()
        fetchPayOutlook()
    }

    /// openMainWindow would present the full application window. Stubbed for now.
    func openMainWindow() {
        // TODO: Implement main window presentation.
    }

    // MARK: - Private network calls

    private func fetchAgenda() {
        let url = baseURL.appendingPathComponent("agenda/today")
        var comps = URLComponents(url: url, resolvingAgainstBaseURL: false)!
        comps.queryItems = [URLQueryItem(name: "user_id", value: userID)]
        guard let finalURL = comps.url else { return }
        URLSession.shared.dataTaskPublisher(for: finalURL)
            .tryMap { data, response -> [DayBoardEvent] in
                let decoder = JSONDecoder()
                decoder.dateDecodingStrategy = .iso8601
                let items = try decoder.decode([BackendEvent].self, from: data)
```

```
                return items.map { DayBoardEvent(id: $0.id.uuidString, title: $0.title, start: $0.start, joinURL: URL(string: $0.joinURL ??
""")) }
            }
            .receive(on: DispatchQueue.main)
            .sink(receiveCompletion: { _ in }, receiveValue: { [weak self] events in
                self?.nextEvent = events.first
            })
            .store(in: &cancellables)
    }

    private func fetchSubscriptions() {
        let url = baseURL.appendingPathComponent("subs")
        var comps = URLComponents(url: url, resolvingAgainstBaseURL: false)!
        comps.queryItems = [URLQueryItem(name: "user_id", value: userID)]
        guard let finalURL = comps.url else { return }
        URLSession.shared.dataTaskPublisher(for: finalURL)
            .tryMap { data, response -> [BackendSubscription] in
                let decoder = JSONDecoder()
                decoder.dateDecodingStrategy = .iso8601
                return try decoder.decode([BackendSubscription].self, from: data)
            }
            .receive(on: DispatchQueue.main)
            .sink(receiveCompletion: { _ in }, receiveValue: { [weak self] subs in
                // Compute bills due in the next 7 days.
                let now = Date()
                let weekAhead = Calendar.current.date(byAdding: .day, value: 7, to: now) ?? now
                var totalCents = 0
                for sub in subs {
                    if let due = sub.nextDue {
                        if due <= weekAhead {
                            totalCents += sub.amountCents
                        }
                    }
                }
                self?.billsThisWeek = centsToDollarString(totalCents)
            })
            .store(in: &cancellables)
    }

    private func fetchCommuteEstimate() {
        // For demonstration we need to know the user's home and office addresses. In
        // production you'd fetch the profile first. Here we call the profile endpoint.
        let profileURL = baseURL.appendingPathComponent("profile")
        var comps = URLComponents(url: profileURL, resolvingAgainstBaseURL: false)!
        comps.queryItems = [URLQueryItem(name: "user_id", value: userID)]
        guard let profileFinalURL = comps.url else { return }
        URLSession.shared.dataTaskPublisher(for: profileFinalURL)
            .tryMap { data, response -> BackendProfile? in
                let decoder = JSONDecoder()
                decoder.dateDecodingStrategy = .iso8601
                // Empty profile may return {} which decodes to nil if we use optional.
                return try? decoder.decode(BackendProfile.self, from: data)
            }
            .flatMap { [weak self] profile -> AnyPublisher<commuteEstResponse?, Never> in
                guard let profile = profile, let self = self else { return Just(nil).eraseToAnyPublisher() }
                // Compose commute API call
                let commuteURL = self.baseURL.appendingPathComponent("commute/estimate")
                var comps = URLComponents(url: commuteURL, resolvingAgainstBaseURL: false)!
```

```
            comps.queryItems = [
                URLQueryItem(name: "from", value: profile.homeAddr),
                URLQueryItem(name: "to", value: profile.officeAddr),
            ]
            guard let finalCommuteURL = comps.url else { return Just(nil).eraseToAnyPublisher() }
            return URLSession.shared.dataTaskPublisher(for: finalCommuteURL)
                .map { $0.data }
                .decode(type: commuteEstResponse.self, decoder: JSONDecoder())
                .map(Optional.some)
                .catch { _ in Just(nil) }
                .eraseToAnyPublisher()
        }
        .receive(on: DispatchQueue.main)
        .sink(receiveValue: { [weak self] est in
            guard let est = est else { return }
            let low = centsToDollarString(est.estCostLowCents)
            let high = centsToDollarString(est.estCostHighCents)
            self?.commuteCost = "\(low)?\(high)"
        })
        .store(in: &cancellables)
}

private func fetchPayOutlook() {
    // Fetch profile first to compute income and hours. Then call taxes API.
    let profileURL = baseURL.appendingPathComponent("profile")
    var comps = URLComponents(url: profileURL, resolvingAgainstBaseURL: false)!
    comps.queryItems = [URLQueryItem(name: "user_id", value: userID)]
    guard let profileFinalURL = comps.url else { return }
    URLSession.shared.dataTaskPublisher(for: profileFinalURL)
        .tryMap { data, response -> BackendProfile? in
            let decoder = JSONDecoder()
            decoder.dateDecodingStrategy = .iso8601
            return try? decoder.decode(BackendProfile.self, from: data)
        }
        .flatMap { [weak self] profile -> AnyPublisher<TaxResult?, Never> in
            guard let self = self, let profile = profile else { return Just(nil).eraseToAnyPublisher() }
            // Derive weekly income in cents.
            var incomeCents = 0
            if let hourly = profile.hourlyCents, let hours = profile.hoursPerWeek {
                incomeCents = hourly * hours * 52 // annual income
            } else if let stipend = profile.stipendCents {
                incomeCents = stipend
            }
            let termWeeks = 12 // Example 12-week internship for projection
            let body: [String: Any] = [
                "incomeCents": incomeCents,
                "state": profile.state,
                "filingStatus": "single",
                "payFreq": profile.payFreq ?? "biweekly",
                "termWeeks": termWeeks,
            ]
            guard let url = self.baseURL.appendingPathComponent("estimate/taxes") as URL? else {
                return Just(nil).eraseToAnyPublisher()
            }
            var req = URLRequest(url: url)
            req.httpMethod = "POST"
            req.setValue("application/json", forHTTPHeaderField: "Content-Type")
            req.httpBody = try? JSONSerialization.data(withJSONObject: body)
```

```
                return URLSession.shared.dataTaskPublisher(for: req)
                    .map { $0.data }
                    .decode(type: TaxResult.self, decoder: JSONDecoder())
                    .map(Optional.some)
                    .catch { _ in Just(nil) }
                    .eraseToAnyPublisher()
            }
            .receive(on: DispatchQueue.main)
            .sink(receiveValue: { [weak self] res in
                guard let res = res else { return }
                self?.payOutlook = self?.centsToDollarString(res.perPaycheckNetCents) ?? "$0.00"
            })
            .store(in: &cancellables)
    }

    // Helper to convert cents to a dollar string like "$12.34".
    private func centsToDollarString(_ cents: Int) -> String {
        let dollars = Double(cents) / 100.0
        let formatter = NumberFormatter()
        formatter.numberStyle = .currency
        return formatter.string(from: NSNumber(value: dollars)) ?? "$0.00"
    }
}

// MARK: - Backend models for decoding

/// BackendEvent mirrors the JSON returned by the /agenda/today endpoint.
fileprivate struct BackendEvent: Decodable {
    let id: UUID
    let start: Date
    let end: Date
    let title: String
    let joinURL: String?
    let location: String?
}

/// BackendSubscription mirrors the /subs endpoint.
fileprivate struct BackendSubscription: Decodable {
    let id: UUID
    let merchant: String
    let amountCents: Int
    let cadenceDays: Int
    let nextDue: Date?
    let source: String
    let isActive: Bool
}

/// BackendProfile mirrors the /profile endpoint.
fileprivate struct BackendProfile: Decodable {
    let homeAddr: String
    let officeAddr: String
    let city: String
    let state: String
    let hourlyCents: Int?
    let hoursPerWeek: Int?
    let stipendCents: Int?
    let payFreq: String?
    let startDate: Date?
```

```
    let inOfficeDays: Int
    let foodCostCents: Int
}

/// commuteEstResponse mirrors the response from /commute/estimate.
fileprivate struct commuteEstResponse: Decodable {
    let distanceMiles: Double
    let durationMinutes: Double
    let estCostLowCents: Int
    let estCostHighCents: Int
}

/// TaxResult mirrors the JSON returned by the /estimate/taxes endpoint. Swift's
/// coding keys use camelCase to map to JSON keys returned by the Go backend.
fileprivate struct TaxResult: Decodable {
    let federalCents: Int
    let stateCents: Int
    let ficaCents: Int
    let perPaycheckNetCents: Int
    let termNetCents: Int
}
}

/// DayBoardEvent represents a calendar event normalized for the view.
struct DayBoardEvent {
    let id: String
    let title: String
    let start: Date
    let joinURL: URL?
}

// ============================================================================
```