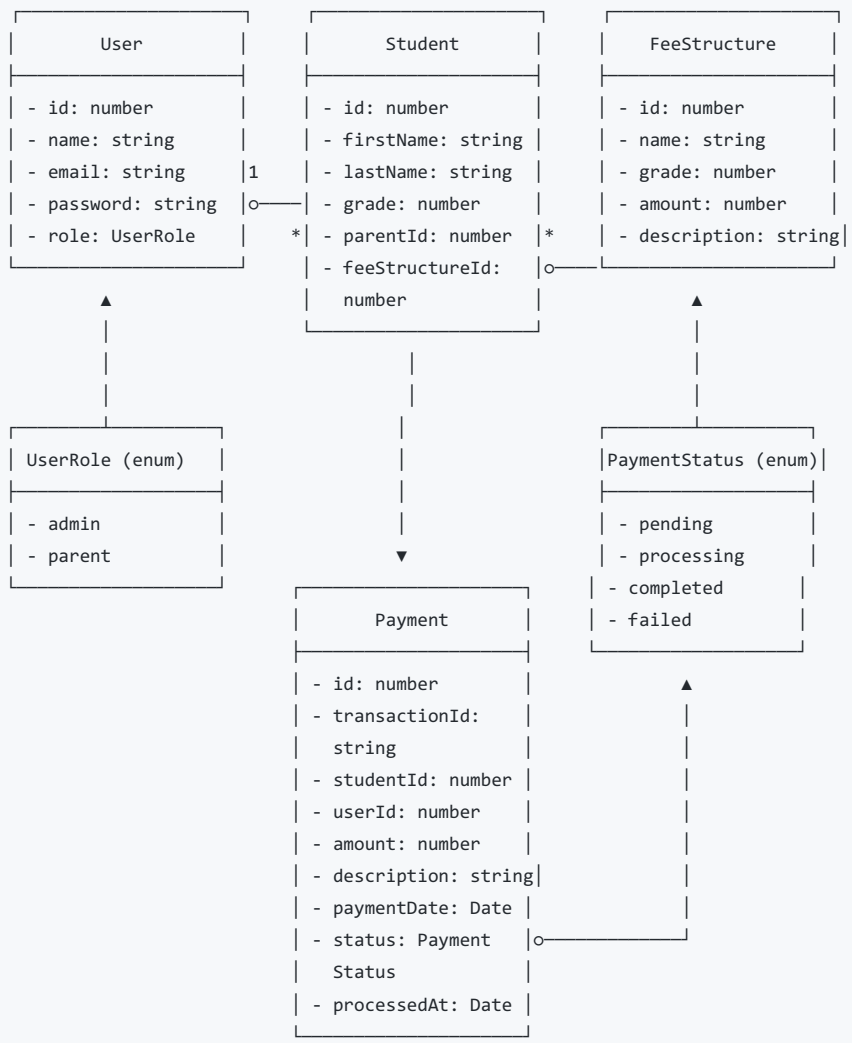
School Fee Management System: Design Principles Documentation

UML Class Diagram



Code Segments Highlighting Design Principles

SOLID Principles

1. Single Responsibility Principle (SRP)

Each class in the system has a single responsibility:

```
// server/storage.ts
// Each method in the storage class has a single responsibility
export class DatabaseStorage implements IStorage {
  async getUser(id: number): Promise<User | undefined> {
    const [user] = await db.select().from(users).where(eq(users.id, id));
    return user || undefined;
  }

  async createStudent(insertStudent: InsertStudent): Promise<Student> {
    const [student] = await db
      .insert(students)
      .values(insertStudent)
      .returning();
    return student;
  }

  // Each method does one thing and does it well
}
```

2. Open/Closed Principle (OCP)

The system is designed to be open for extension but closed for modification:

```
// shared/schema.ts
// Enums allow extending status types without modifying existing code
export const paymentStatusEnum = pgEnum('payment_status', ['pending', 'processing', 'completed', 'failed']);

// New status types can be added without changing the Payment implementation
export const payments = pgTable("payments", {
  id: serial("id").primaryKey(),
  transactionId: text("transaction_id").notNull(),
  studentId: integer("student_id").references(() => students.id).notNull(),
  userId: integer("user_id").references(() => users.id).notNull(),
  amount: decimal("amount", { precision: 10, scale: 2 }).notNull(),
  description: text("description"),
  paymentDate: timestamp("payment_date").defaultNow().notNull(),
  status: paymentStatusEnum("status").default("pending").notNull(),
  processedAt: timestamp("processed_at"),
});
```

3. Liskov Substitution Principle (LSP)

User roles implement the same interface and can be substituted:

```
// server/routes.ts
// Middleware functions that can be substituted based on access level
const requireAuth = (req: Request, res: Response, next: Function) => {
  if (!req.isAuthenticated()) {
    return res.status(401).json({ message: "Unauthorized" });
  }
  next();
};

const requireAdmin = (req: Request, res: Response, next: Function) => {
  if (!req.isAuthenticated() || req.user.role !== "admin") {
    return res.status(403).json({ message: "Forbidden" });
  }
  next();
};

// Routes can use either middleware interchangeably
app.get("/api/students", requireAuth, async (req, res) => {
  // Code that works regardless of which middleware was used
});

app.post("/api/fee-structures", requireAdmin, async (req, res) => {
  // Code that requires admin privileges
});
```

4. Interface Segregation Principle (ISP)

Interfaces are client-specific instead of general-purpose:

```
// server/storage.ts
// IStorage interface is segregated by entity type
export interface IStorage {
  // User operations
  getUser(id: number): Promise<User | undefined>;
  getUserByEmail(email: string): Promise<User | undefined>;
  createUser(user: InsertUser): Promise<User>;

  // Student operations
  getStudent(id: number): Promise<StudentWithRelations | undefined>;
  getStudentsByParent(parentId: number): Promise<StudentWithRelations[]>;

  // Fee Structure operations
  getFeeStructure(id: number): Promise<FeeStructure | undefined>;
  getAllFeeStructures(): Promise<FeeStructure[]>;

  // Payment operations
  getPayment(id: number): Promise<PaymentWithRelations | undefined>;
  getPaymentsByStudent(studentId: number): Promise<PaymentWithRelations[]>;

  // Session store
  sessionStore: session.Store;
}
```

5. Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not concrete implementations:

```
// server/routes.ts
// Routes depend on storage interface, not concrete implementation
export async function registerRoutes(app: Express): Promise<Server> {
  // API endpoints use the storage abstraction
  app.get("/api/students", requireAuth, async (req, res) => {
    try {
      const students = req.user.role === "admin"
        ? await storage.getAllStudents()
        : await storage.getStudentsByParent(req.user.id);
      res.json(students);
    } catch (error) {
      console.error("Error fetching students:", error);
      res.status(500).json({ message: "Failed to fetch students" });
    }
  });

  // Storage implementation can be changed without affecting routes
}
```

DRY (Don't Repeat Yourself) Principle

The code avoids duplication through shared schemas and utilities:

```
// shared/schema.ts
// Schema definitions are shared between client and server
export const insertUserSchema = createInsertSchema(users).omit({ id: true, createdAt: true });
export const selectUserSchema = createSelectSchema(users);

// Types are derived from schemas to maintain consistency
export type InsertUser = z.infer<typeof insertUserSchema>;
export type User = typeof users.$inferSelect;

// client/src/components/add-student-modal.tsx
// Reusing the same schema for form validation
const formSchema = insertStudentSchema.extend({
  firstName: z.string().min(2, "First name must be at least 2 characters"),
  lastName: z.string().min(2, "Last name must be at least 2 characters"),
  grade: z.coerce.number().min(1, "Grade must be at least 1").max(12, "Grade must be at most 12"),
});
```

KISS (Keep It Simple, Stupid) Principle

The code is kept simple and straightforward:

```
// server/utils.ts
// Simple, focused utility function for processing payments
export async function processPayment(paymentId: number) {
  try {
    console.log(`Processing payment ${paymentId}...`);

    // Get the payment from storage
    const payment = await storage.getPayment(paymentId);
    if (!payment) {
      console.error(`Payment ${paymentId} not found`);
      return;
    }

    // Simulate payment processing
    await new Promise(resolve => setTimeout(resolve, 2000));

    // Update payment status
    await storage.updatePaymentStatus(paymentId, "completed", new Date());
    console.log(`Payment ${paymentId} completed successfully`);
  } catch (error) {
    console.error(`Error processing payment ${paymentId}:`, error);
    await storage.updatePaymentStatus(paymentId, "failed");
  }
}
```

Modularity

The code is organized into modular components:

```
// Project structure demonstrates modularity
// server/
//   - auth.ts (Authentication logic)
//   - db.ts (Database connection)
//   - routes.ts (API endpoints)
//   - storage.ts (Data access layer)
//   - utils.ts (Utility functions)

// client/src/
//   - components/ (Reusable UI components)
//   - hooks/ (Custom React hooks)
//   - pages/ (Page components)
//   - lib/ (Utility functions)

// shared/
//   - schema.ts (Shared data models)
```

Dependency Injection

The system uses dependency injection to manage dependencies:

```
// server/index.ts
// Database and storage are injected into routes
import { storage } from "./storage";

async function main() {
  const app = express();
  // ... app configuration

  // Registering routes with Express app instance
  const server = await registerRoutes(app);

  // ... more configuration

  return { app, server };
}
```

Encapsulation

Data and behavior are encapsulated within classes:

```
// server/storage.ts
// DatabaseStorage encapsulates all database operations
export class DatabaseStorage implements IStorage {
  sessionStore: session.Store;

  constructor() {
    // Initialize session store
    const PostgresSessionStore = connectPg(session);
    this.sessionStore = new PostgresSessionStore({
      pool,
      createTableIfMissing: true,
    });
  }

  // Methods encapsulate data access operations
  async getStudentsByParent(parentId: number): Promise<StudentWithRelations[]> {
    // Implementation details are hidden from consumers
  }
}
```

Factory Pattern

The application uses factory patterns to create objects:

```
// client/src/lib/queryClient.ts
// Factory function to create query functions
export const getQueryFn =
  ({ on401 = "throw" }: { on401?: "throw" | "returnNull" } = {}) =>
  async ({ queryKey }: { queryKey: string[] }) => {
    try {
      const res = await fetch(queryKey[0]);
      if (res.status === 401) {
        if (on401 === "returnNull") return null;
        throw new Error("Unauthorized");
      }
      if (!res.ok) {
        throw new Error(`Network response was not ok: ${res.status}`);
      }
      return await res.json();
    } catch (error) {
      console.error(`Failed to fetch ${queryKey[0]}:`, error);
      throw error;
    }
  };
```

Observer Pattern

The system uses events and observers for state management:

```
// client/src/hooks/use-auth.tsx
// Observer pattern using React Query's cache
const loginMutation = useMutation({
  mutationFn: async (credentials: LoginData) => {
    const res = await apiRequest("POST", "/api/login", credentials);
    return await res.json();
  },
  onSuccess: (user: SelectUser) => {
    // Update query cache (observer pattern)
    queryClient.setQueryData(["/api/user"], user);
  },
  onError: (error: Error) => {
    toast({
      title: "Login failed",
      description: error.message,
      variant: "destructive",
    });
  },
});
```

GitHub Repository

School Fee Management System Repository

Note: Replace the GitHub link with your actual repository link after pushing your code.