**Restaurant POS Document**

### Workflow: From Order Creation to Payment and Delivery

1. Order Creation
   * Customer: Places an order with the waitstaff.
   * Waitstaff: Enters the order into the Point of Sale (POS) system. The order can consist of special recipes or customized items.
   * System: Saves the order details in the Orders collection. The order status is set to "Pending."
2. Order Processing
   * Waitstaff: Confirms the order and sends it to the kitchen.
   * System: Updates the order status to "Pending" in the kitchen's queue.
3. Order Acceptance or Rejection
   * Kitchen Staff: Reviews the pending orders.
   * Kitchen Staff:
     + Accepts the order: Updates the order status to "In Preparation."
     + Rejects the order: Updates the order status to "Cancelled" and adds a rejection message.
4. Order Preparation
   * Kitchen Staff: Prepares the order according to the items listed.
   * Kitchen Staff: Marks the order as ready once completed.
   * System: Updates the order status to "Ready."
5. Order Delivery
   * Waitstaff: Receives notification that the order is ready.
   * Waitstaff: Delivers the prepared food to the customer.
6. Payment Processing
   * Customer: Requests the bill from the waitstaff after finishing the meal.
   * Waitstaff: Generates the bill using the POS system.
   * Customer: Pays the bill.
   * Cashier: Processes the payment (cash, card, or digital payment).
   * System: Updates the order status to "Completed."
7. Order Completion and Reporting
   * System: Records the payment details and updates sales and profits data in the Sales collection.
   * Manager: Reviews daily/weekly/monthly sales and profit reports through the dashboard.

### Parties Involved and Responsibilities

1. Customer
   * Places the order.
   * Requests the bill.
   * Makes the payment.
2. Waitstaff
   * Takes the order from the customer.
   * Enters the order into the POS system.
   * Confirms the order and sends it to the kitchen.
   * Delivers the food to the customer.
   * Generates the bill.
   * Facilitates the payment process.
   * Updates the order status to "Completed."
3. Kitchen Staff
   * Reviews pending orders.
   * Accepts or rejects orders.
   * Updates the order status to "In Preparation" or "Cancelled."
   * Prepares the food according to the order.
   * Marks the order as "Ready" when completed.
4. Cashier
   * Processes the payment from the customer.
   * Updates the payment status in the system.
5. Manager
   * Monitors overall operations.
   * Reviews sales and profit reports.
   * Manages inventory, staff, and other administrative tasks.

### Summary

1. Order Creation: Customer -> Waitstaff -> System
2. Order Processing: Waitstaff -> System
3. Order Acceptance/Rejection: Kitchen Staff -> System
4. Order Preparation: Kitchen Staff -> System
5. Order Ready Notification: Kitchen Staff -> Waitstaff
6. Order Delivery: Waitstaff -> Customer
7. Payment Processing: Customer -> Waitstaff -> Cashier -> System
8. Order Completion and Reporting: System -> Manager

This workflow ensures that the process is streamlined and that all parties involved have clear responsibilities, resulting in efficient order management, preparation, and payment processing.

### 1. **Authentication Pages**

* Login Page: For users to log in to the system.
* Forgot Password Page: To handle password recovery.

### 2. **Dashboard Pages**

* Admin Dashboard: Overview of restaurant performance, sales, and inventory status.
* Manager Dashboard: Detailed insights into sales, profits, inventory, and staff performance.
* Waitstaff Dashboard: Overview of orders and table assignments.
* Kitchen Dashboard: List of orders in different statuses (Pending, In Preparation).

### 3. **Order Management Pages**

* Order Entry Page: For waitstaff to enter new orders, both special and customized.
* Order List Page: View and manage all orders, with filtering options for status (Pending, In Preparation, Ready, Cancelled, Completed).
* Order Detail Page: Detailed view of individual orders, including status updates and messages.

### 4. **Menu Management Pages**

* Food Item List Page: View, add, edit, and delete food items.
* Food Item Detail Page: Detailed view of a food item, including ingredients and pricing.
* Special Recipe List Page: View, add, edit, and delete special recipes.
* Special Recipe Detail Page: Detailed view of a special recipe, including ingredients, steps, and pricing.

### 5. **Inventory Management Pages**

* Ingredient List Page: View, add, edit, and delete ingredients.
* Ingredient Detail Page: Detailed view of an ingredient, including name, category, amount, and price per unit.
* Inventory Status Page: Overview of current inventory levels and alerts for low stock.

### 6. **Sales and Profits Pages**

* Sales Report Page: View sales data, with options to filter by hourly, daily, weekly, or monthly views.
* Profit Report Page: View profit data, with options to filter by hourly, daily, weekly, or monthly views.
* Sales Detail Page: Detailed view of individual sales transactions.

### 7. **User Management Pages**

* User List Page: View, add, edit, and delete users.
* User Detail Page: Detailed view of a user, including role and permissions.
* Profile Page: View and edit the logged-in user's profile.

### 8. **Settings and Configuration Pages**

* General Settings Page: Configure restaurant-specific settings (e.g., business hours, tax rates).
* Role Management Page: Define and manage user roles and permissions.
* Notification Settings Page: Configure notification preferences for different users.

### 9. **Miscellaneous Pages**

* Help and Support Page: Access help documentation and support contact information.
* About Page: Information about the system and the company.
* Terms and Conditions Page: Display terms and conditions of use.
* Privacy Policy Page: Display the privacy policy.

### Summary

Authentication Pages:

* Login Page
* Forgot Password Page

Dashboard Pages:

* Admin Dashboard
* Manager Dashboard
* Waitstaff Dashboard
* Kitchen Dashboard

Order Management Pages:

* Order Entry Page
* Order List Page
* Order Detail Page

Menu Management Pages:

* Food Item List Page
* Food Item Detail Page
* Special Recipe List Page
* Special Recipe Detail Page

Inventory Management Pages:

* Ingredient List Page
* Ingredient Detail Page
* Inventory Status Page

Sales and Profits Pages:

* Sales Report Page
* Profit Report Page
* Sales Detail Page

User Management Pages:

* User List Page
* User Detail Page
* Profile Page

Settings and Configuration Pages:

* General Settings Page
* Role Management Page
* Notification Settings Page

Miscellaneous Pages:

* Help and Support Page
* About Page
* Terms and Conditions Page
* Privacy Policy Page

**Database Schemas**

**Ingredients schema**

const mongoose = require('mongoose');

const ingredientSchema = new mongoose.Schema({

name: {

type: String,

required: true,

trim: true

},

category: {

type: String,

required: true,

trim: true

},

amount: {

type: Number,

required: true,

min: 0

},

pricePerUnit: {

type: Number,

required: true,

min: 0

}

});

const Ingredient = mongoose.model('Ingredient', ingredientSchema);

module.exports = Ingredient;

**Foods schema**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const foodSchema = new Schema({

name: {

type: String,

required: true,

trim: true

},

type: {

type: String,

required: true,

trim: true

},

ingredients: [{

ingredient: {

type: Schema.Types.ObjectId,

ref: 'Ingredient',

required: true

},

quantity: {

type: Number,

required: true,

min: 0

}

}],

cost: {

type: Number,

required: true,

min: 0

},

sellingPrice: {

type: Number,

required: true,

min: 0,

validate: {

validator: function(v) {

return v > this.cost;

},

message: props => `Selling price (${props.value}) should be higher than cost (${this.cost})!`

}

},

createdAt: {

type: Date,

default: Date.now

},

updatedAt: {

type: Date,

default: Date.now

}

});

// Pre-save hook to calculate cost based on ingredients

foodSchema.pre('save', async function(next) {

if (this.isModified('ingredients')) {

let totalCost = 0;

for (const item of this.ingredients) {

const ingredient = await mongoose.model('Ingredient').findById(item.ingredient);

if (ingredient) {

totalCost += ingredient.pricePerUnit \* item.quantity;

}

}

this.cost = totalCost;

}

this.updatedAt = Date.now();

next();

});

const Food = mongoose.model('Food', foodSchema);

module.exports = Food;

**Special Recipe schema**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const specialRecipeSchema = new Schema({

name: {

type: String,

required: true,

trim: true

},

origin: {

type: String,

required: true,

trim: true

},

description: {

type: String,

required: true,

trim: true

},

ingredients: [{

ingredient: {

type: Schema.Types.ObjectId,

ref: 'Ingredient',

required: true

},

quantity: {

type: Number,

required: true,

min: 0

}

}],

steps: [{

type: String,

required: true

}],

cost: {

type: Number,

required: true,

min: 0

},

sellingPrice: {

type: Number,

required: true,

min: 0,

validate: {

validator: function(v) {

return v > this.cost;

},

message: props => `Selling price (${props.value}) should be higher than cost (${this.cost})!`

}

},

createdAt: {

type: Date,

default: Date.now

},

updatedAt: {

type: Date,

default: Date.now

}

});

// Pre-save hook to calculate cost based on ingredients

specialRecipeSchema.pre('save', async function(next) {

if (this.isModified('ingredients')) {

let totalCost = 0;

for (const item of this.ingredients) {

const ingredient = await mongoose.model('Ingredient').findById(item.ingredient);

if (ingredient) {

totalCost += ingredient.pricePerUnit \* item.quantity;

}

}

this.cost = totalCost;

}

this.updatedAt = Date.now();

next();

});

const SpecialRecipe = mongoose.model('SpecialRecipe', specialRecipeSchema);

module.exports = SpecialRecipe;

**Users schema**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const userSchema = new Schema({

idNumber: {

type: String,

required: true,

unique: true,

trim: true

},

name: {

type: String,

required: true,

trim: true

},

pin: {

type: String,

required: true

},

role: {

type: String,

required: true,

enum: ['Owner', 'Manager', 'Waitstaff', 'Cashier', 'KitchenStaff', 'InventoryManager', 'Accountant']

},

createdAt: {

type: Date,

default: Date.now

},

updatedAt: {

type: Date,

default: Date.now

}

});

// Pre-save hook to update the updatedAt field

userSchema.pre('save', function(next) {

this.updatedAt = Date.now();

next();

});

const User = mongoose.model('User', userSchema);

module.exports = User;

**Order Schema**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const orderItemSchema = new Schema({

itemType: {

type: String,

required: true,

enum: ['Food', 'SpecialRecipe']

},

item: {

type: Schema.Types.ObjectId,

required: true,

refPath: 'items.itemType'

},

quantity: {

type: Number,

required: true,

min: 1

}

}, { \_id: false });

const orderSchema = new Schema({

items: [orderItemSchema],

status: {

type: String,

required: true,

enum: ['Pending', 'In Preparation', 'Ready', 'Cancelled', 'Completed'],

default: 'Pending'

},

customerDetails: {

type: String,

required: true,

trim: true

},

tableNumber: {

type: Number,

required: true,

min: 1

},

totalAmount: {

type: Number,

required: true,

min: 0

},

rejectionMessage: {

type: String,

trim: true

},

createdAt: {

type: Date,

default: Date.now

},

updatedAt: {

type: Date,

default: Date.now

}

});

// Pre-save hook to update the updatedAt field and calculate the totalAmount

orderSchema.pre('save', async function(next) {

this.updatedAt = Date.now();

let total = 0;

for (const item of this.items) {

if (item.itemType === 'Food') {

const food = await mongoose.model('Food').findById(item.item);

if (food) {

total += food.sellingPrice \* item.quantity;

}

} else if (item.itemType === 'SpecialRecipe') {

const specialRecipe = await mongoose.model('SpecialRecipe').findById(item.item);

if (specialRecipe) {

total += specialRecipe.sellingPrice \* item.quantity;

}

}

}

this.totalAmount = total;

next();

});

const Order = mongoose.model('Order', orderSchema);

module.exports = Order;

**Sales Schema:**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const salesSchema = new Schema({

orderId: {

type: Schema.Types.ObjectId,

ref: 'Order',

required: true

},

date: {

type: Date,

required: true,

default: Date.now

},

totalAmount: {

type: Number,

required: true,

min: 0

},

profit: {

type: Number,

required: true,

min: 0

}

});

const Sales = mongoose.model('Sales', salesSchema);

module.exports = Sales;

**Aggregation Pipeline Example: Daily Sales and Profits**

const mongoose = require('mongoose');

const Sales = mongoose.model('Sales');

const getDailySalesAndProfits = async () => {

const dailySalesAndProfits = await Sales.aggregate([

{

$group: {

\_id: { $dateToString: { format: "%Y-%m-%d", date: "$date" } },

totalSales: { $sum: "$totalAmount" },

totalProfit: { $sum: "$profit" }

}

},

{

$sort: { \_id: 1 }

}

]);

return dailySalesAndProfits;

};

getDailySalesAndProfits().then(data => console.log(data));

#### Aggregation Pipeline Example: Weekly Sales and Profits

javascript

Copy code

const getWeeklySalesAndProfits = async () => {

const weeklySalesAndProfits = await Sales.aggregate([

{

$group: {

\_id: { $isoWeek: "$date" },

totalSales: { $sum: "$totalAmount" },

totalProfit: { $sum: "$profit" }

}

},

{

$sort: { \_id: 1 }

}

]);

return weeklySalesAndProfits;

};

getWeeklySalesAndProfits().then(data => console.log(data));

#### Aggregation Pipeline Example: Monthly Sales and Profits

javascript

Copy code

const getMonthlySalesAndProfits = async () => {

const monthlySalesAndProfits = await Sales.aggregate([

{

$group: {

\_id: { $dateToString: { format: "%Y-%m", date: "$date" } },

totalSales: { $sum: "$totalAmount" },

totalProfit: { $sum: "$profit" }

}

},

{

$sort: { \_id: 1 }

}

]);

return monthlySalesAndProfits;

};

getMonthlySalesAndProfits().then(data => console.log(data));