Expense Tracker with Persistent Data

Objective

Create a program to manage and persistently track expenses. Use JavaScript basics like arrays, objects, functions, loops, and conditionals, along with `localStorage` for data persistence. The program will allow users to add, view, and analyze expenses through prompts and alerts.

Instructions

Step 1: Initialize the Expense Database

1. Start by defining an array called `expenses` to store expense objects.
2. Each expense object should include:
    - `description` : A string describing the expense.
    - `amount` : A number representing the expense in dollars.
    - `category` : A string for the type of expense (e.g., "Food", "Transport").
    - `date` : A string in the format `"YYYY-MM-DD"` .
3. Create helper functions to manage persistent data:
    - **`loadExpenses()`** : Loads the `expenses` array from `localStorage` . If no data exists, initialize an empty array.
    - **`saveExpenses(expenses)`** : Saves the current `expenses` array to `localStorage` .

Step 2: Implement the Core Functions

Write the following functions, ensuring they use `loadExpenses` to access data and `saveExpenses` to update it:

1. **Add an Expense**
    - Function: `addExpense()`
    - Use prompts to ask the user for:
        - `description` : A brief description of the expense.
        - `amount` : The expense amount in dollars (validate to ensure it's a number).
        - `category` : The category for the expense.
        - `date` : The date in `"YYYY-MM-DD"` format.
    - Add the expense to the `expenses` array and save it.
    - Show an alert confirming the expense was added.
2. **Get Total Expenses**

- Function: `getTotalExpenses()`
- Calculate the total amount of all expenses.
- Display the total using an alert.

3. **Get Expenses by Category**
   - Function: `getExpensesByCategory()`
   - Use a prompt to ask the user for a category (case-insensitive).
   - Filter and return all expenses in the specified category.
   - Display the results using an alert, with each expense shown on a new line.
4. **Get Expenses by Month**
   - Function: `getExpensesByMonth()`
   - Use a prompt to ask the user for a month in `"YYYY-MM"` format.
   - Filter and return all expenses added during that month.
   - Display the results in an alert.
5. **Find the Most Expensive Expense**
   - Function: `findMostExpensiveExpense()`
   - Identify and return the expense with the highest `amount`.
   - Show its details in an alert.
6. **Generate an Expense Report**
   - Function: `generateExpenseReport()`
   - Calculate and display:
     - The total amount of all expenses.
     - A breakdown of expenses by category (e.g., `{"Food": 150, "Transport": 75}`).
   - Format the report in a readable way using alerts.

Step 3: Bonus Features (Optional)

Add the following features for extra practice:

1. **Delete an Expense**
   - Function: `deleteExpense()`
   - Allow the user to delete an expense by entering its description.
   - Update and save the `expenses` array.
2. **Filter Expenses by Amount Range**
   - Function: `filterExpensesByRange()`
   - Allow the user to input a minimum and maximum amount.
   - Return and display expenses within the specified range.
3. **Interactive Menu**
   - Create a main menu using a `while` loop to allow the user to choose different operations (e.g., "1: Add Expense, 2: View Total, 3: Exit").

Example Workflow

1. The user runs the program and chooses an action (e.g., adding an expense, viewing total expenses).
2. The program uses prompts to gather input and alerts to display results.
3. All changes to the `expenses` array are saved to `localStorage`, ensuring data persists even if the program is closed.

Sample Alerts

1. **Adding an Expense**

   ```
   Expense added: Dinner, $40, Food, 2024-01-15
   ```

2. **Total Expenses**

   ```
   Total expenses: $200.00
   ```

3. **Expense Report**

   ```
   Total Expenses: $200.00
   Category Breakdown:
   - Food: $100.00
   - Transport: $50.00
   - Entertainment: $50.00
   ```

Requirements

1. Use `localStorage` to store and retrieve data persistently.
2. Use arrays, objects, loops, and functions to implement the logic.
3. Validate user inputs (e.g., ensure amounts are numbers and dates are valid).
4. Format alerts to be user-friendly and readable.

Estimated Time

- Core functionality: 2 hours
- Bonus features: 30–45 minutes