

Spending Tracker

Stefan Atanasovski 183205

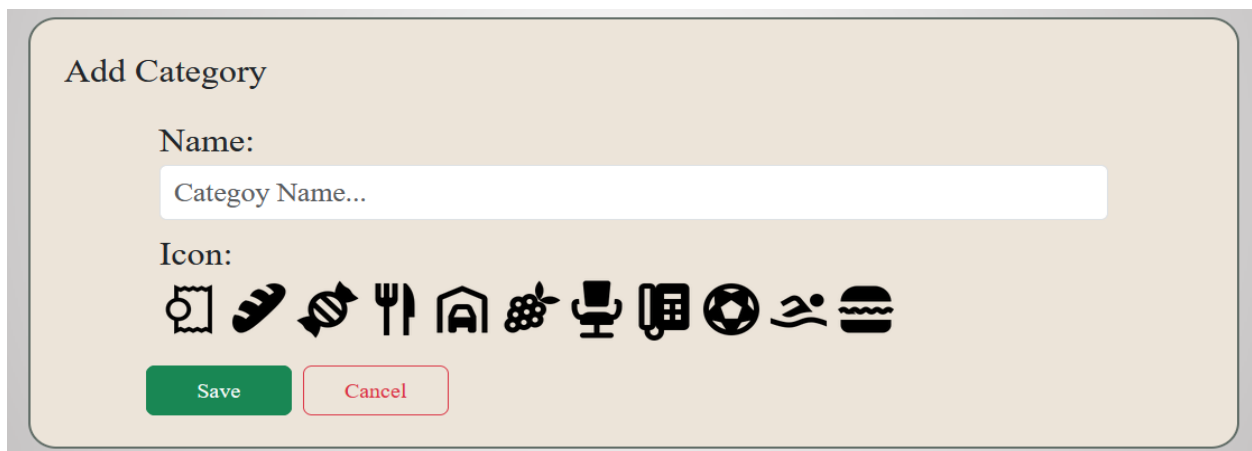
INTRO

The purpose of this application is for the user to keep track of their spending and income. The application offers the user fine visual display of his spendings. The whole project is in-memory (there is no database connected to it so whenever the user hits refresh on the browser every change is lost and the project is initialized again). Images that are used in this project are the ones for the icons the user has to select for the category he creates and are located in the assets/images folder.

BUSINESS LOGIC AND FLOW OF THE APP

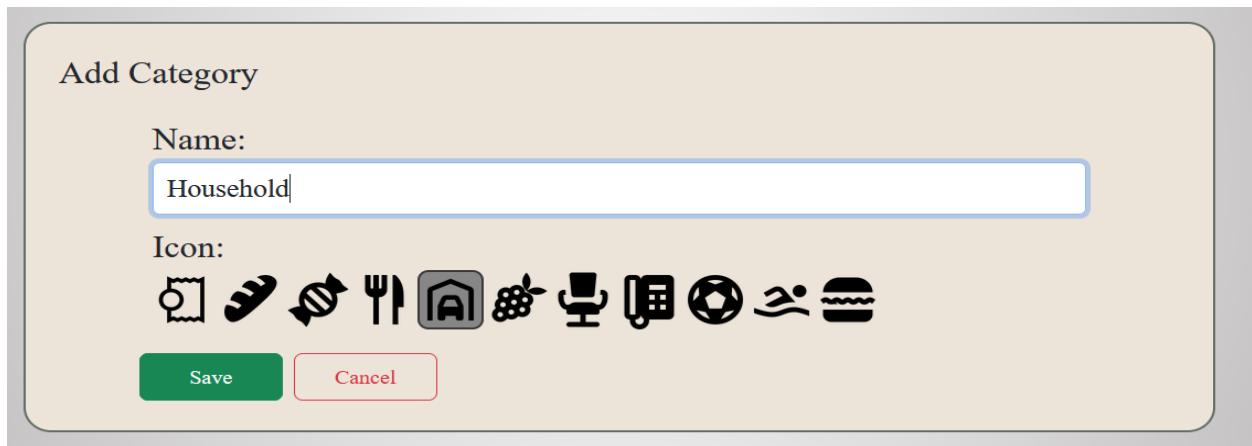
Categories

The first thing the user needs to do is create a category. That can be done in the category section of the menu which view looks like this.



The screenshot shows a light beige rounded rectangle titled "Add Category". Inside, there is a "Name:" label followed by a white text input field containing the placeholder text "Category Name...". Below this is an "Icon:" label followed by a horizontal row of 12 black icons: a gift box, a hand holding a coin, a crossed-out circle, a fork and knife, a house, a bunch of grapes, a chair, a calendar, a soccer ball, a person swimming, and a hamburger. At the bottom are two buttons: a green "Save" button and a red-outlined "Cancel" button.

Here we select a name for our category and select an icon that we want for it and click the save button.



This screenshot is identical to the previous one, but the text input field now contains the word "Household". The "House" icon in the row of 12 icons is highlighted with a grey background, indicating it has been selected.

The category model looks like this:

```
export interface Category{
  id: number,
  name: string,
  icon: Icon | undefined
}
```

And the icon model like this:

```
export interface Icon{
  id: number,
  iconUrl: string
}
```

After the user clicks on the save button the method from the add-category component is called:

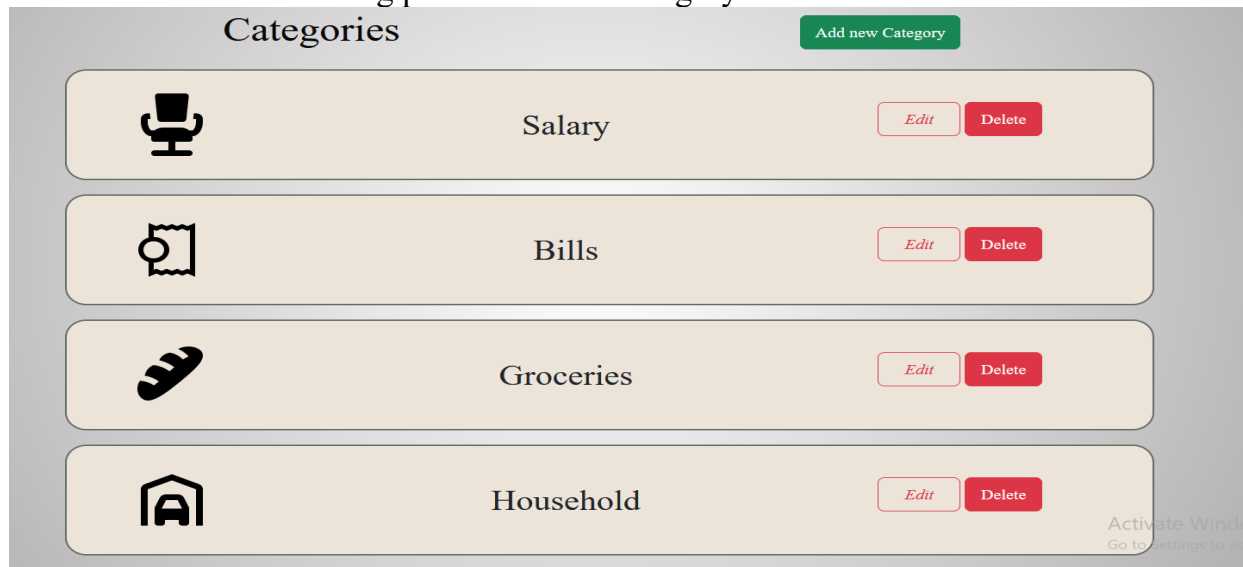
```
saveCategory(): void {
  let iconId: number = Number(this.categoryForm.value.iconId);
  if (this.categoryForm.valid) {
    let c = {
      name: this.categoryForm.value.categoryName,
      icon: this.categoryService.getIcon(iconId) as Icon
    }
    console.log(c);
    this.categoryService.createCategory(c);
    this.onSaveComplete();
  }
}
```

which calls the method from the service that saves the category:

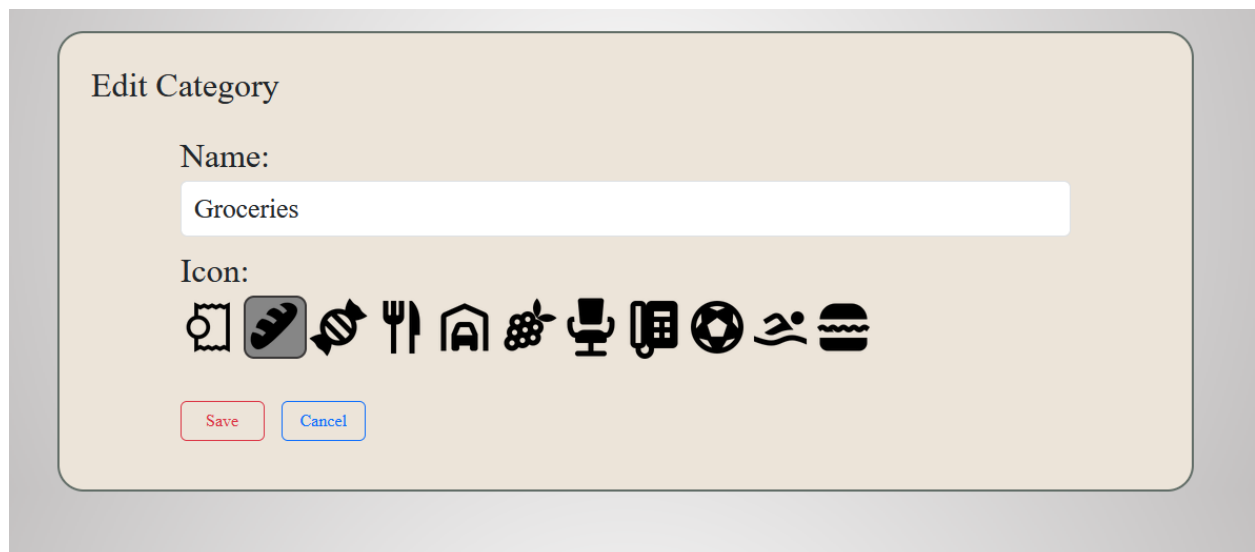
```
createCategory(category: any) {
  category.id = Math.max(...this.getCategories().map(o => o.id)) + 1;
  this.categories.push(category as Category);
  this.categoryUpdated.emit();
}
```

An event is fired so that the category-list-component knows that it has to update the list it displays. The category-list section displays the categories that we have.

We can see in the following picture that the category we added is in it:



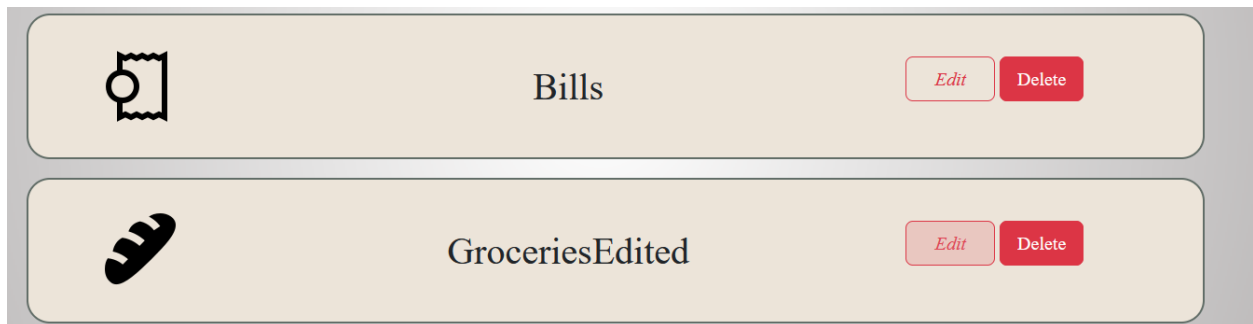
In this view we can see that for each category we have the edit and delete options. The edit options let's us go into edit view so we can change the properties of the category we want. It looks very similar to the add category view with the difference that it pre-selects the values of the category we edit.



When we click on the save button the edit functions is called from service which first finds the selected category from the list of categories, then updates its values and saves it.

```
updateCategory(category: Category) {  
  let objIndex = this.categories  
    .findIndex((obj => obj.id == category.id));  
  this.categories[objIndex] = category;  
  this.categoryUpdated.emit(); }  
}
```

This emits the same event as the save and updates the List we have. If we changed the name of the category in the previous picture, the list view will look like this.



We'll come back to the delete button after the transactions are explained.

Transactions

Now that we have few categories we can add Transactions. This is the main part of the application. To add new transaction we go to the transactions section of the menu and click the add transaction button. The following view will appear:

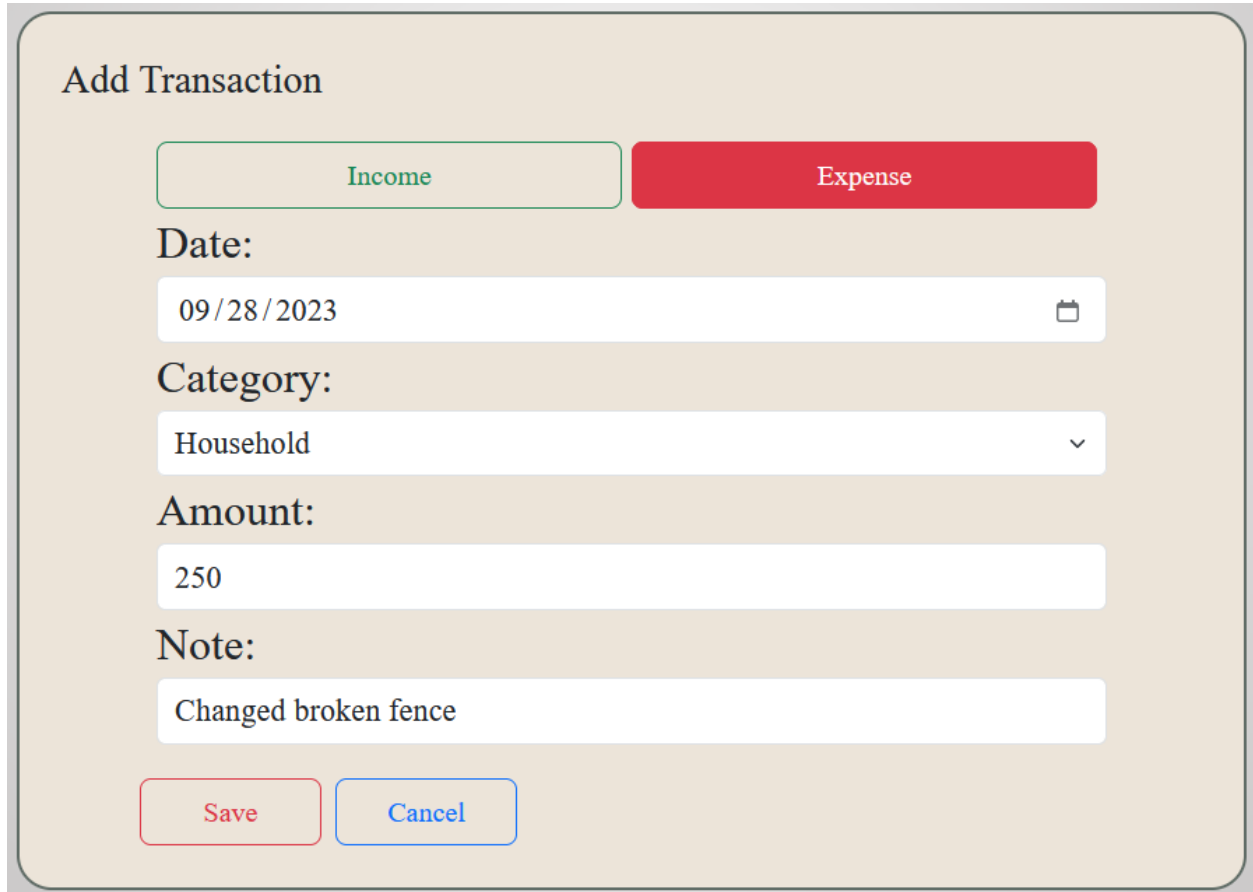
A screenshot of the 'Add Transaction' form. At the top, it says 'Add Transaction'. Below this are two buttons: 'Income' (green border) and 'Expense' (red border). Then, there are four labeled input fields: 'Date:' with a text input showing 'mm/dd/yyyy' and a calendar icon; 'Category:' with a dropdown menu; 'Amount:' with a text input showing 'Amount'; and 'Note:' with a text input showing 'No Note Entered'. At the bottom, there are two buttons: 'Save' (red border) and 'Cancel' (blue border).

There are two types of Transactions, Income and Expense. The transaction model looks like this:

```
export interface Transaction{
  id: number,
  type: number,
  date: Date,
  category: Category | undefined,
```

```
amount: number,  
note: string  
}
```

Let's add a transaction with this values



The image shows a web form titled "Add Transaction". It has two radio buttons for "Income" (selected) and "Expense". Below these are fields for "Date:" (09/28/2023), "Category:" (Household), "Amount:" (250), and "Note:" (Changed broken fence). At the bottom are "Save" and "Cancel" buttons.

Add Transaction

☒ Income ☐ Expense

Date:
09/28/2023

Category:
Household

Amount:
250

Note:
Changed broken fence

When we insert the values in the form and click on the save button the function save from the add-transaction component is called:

```
saveTransaction(){  
  let categoryId: number = Number(this.transactionForm.value.categoryId);  
  if(this.transactionForm.valid){  
    let t = {  
      type: Number(this.transactionForm.value.transactionType),  
      date: this.transactionForm.value.transactionDate,  
      category: this.categoryService.getCategory(categoryId) as Category,  
      amount: Number(this.transactionForm.value.transactionAmount),  
      note: this.transactionForm.value.transactionNote  
    }  
  }  
}
```

```

    }
    this.transactionService.createTransaction(t);
    this.onSaveComplete();
  }
}

```

that calls the method from the transaction service:




```

createTransaction(transaction: any){
  transaction.id = Math.max(...this.getTransactions().map(o => o.id)) + 1;
  this.transactions.push(transaction as Transaction);
  this.totalIncome();
  this.totalExpense();
  this.transactionUpdated.emit();
}

```

That saves the transaction in the list and emits an event so that the list-component knows that it has to update its values.

The list component looks like this:

Transactions					Add new Transaction	
\$3000			\$400			
	Salary	Aug 8, 2023	\$3000	Edit	Delete	
	Phone Bill	Aug 8, 2023	\$150	Edit	Delete	
	Changed broken fence	Sep 28, 2023	\$250	Edit	Delete	

At the top there's the option to add new transaction and we can also see that we have information about how much we have spent and how much we have earned. This updates with every transaction we add or delete.

We can see that we have the options to edit a transaction and to delete a transaction. The edit button opens a view similar to the add view with pre-filled values.

The view looks like this:

A screenshot of a mobile application's 'Edit Transaction' form. The form is set against a light beige background with rounded corners. At the top, the title 'Edit Transaction' is displayed. Below the title are two buttons: 'Income' (light green) and 'Expense' (red). The 'Date:' field shows '09/08/2023' with a calendar icon. The 'Category:' field is a dropdown menu currently showing 'Bills'. The 'Amount:' field contains the value '150'. The 'Note:' field contains the text 'Phone Bill'. At the bottom of the form are two buttons: 'Save' (red) and 'Cancel' (blue).

The save button calls the `updateTransaction` function which first finds the transaction in the list and then edits it.

```
updateTransaction(transaction: Transaction){
  let objIndex = this.transactions.findIndex((obj => obj.id ==
transaction.id));
  this.transactions[objIndex] = transaction;
  this.totalIncome();
  this.totalExpense();
  this.transactionUpdated.emit();
}
```

The create and update transactions call the `totalIncome` and `totalExpense` methods. They calculate the income and the expense from all of the current transaction and update the view in the `transation-list` component so that it can be correct at all times.

Here's the code for the above mentioned functions:

```
totalIncome(): number{
  let total: number = 0;
  this.transactions.forEach((t) => {
    if(t.type == 0){
      total = Number(total) + Number(t.amount);
    }
  } );
  return total;
}
```

```
totalExpense(): number{
  let total: number = 0;
  this.transactions.forEach((t) => {
    if(t.type == 1){
      total = Number(total) + Number(t.amount);
    }
  } );
  return total;
}
```

The delete button removes the transaction from the list.

Now to get back to the delete category button because it has more complex logic. This button calls the deleteCategory function from the sharedService that is dependent on both the category and transaction services. The function looks like this.

```
deleteCategory(id: number) {
  console.log(`In shared service ${id}`);
  let isReferenced = false;
  this.transactionService.getTransactions().forEach(transaction => {
    if (transaction.category?.id === id) {
      isReferenced = true;
    }
  });
  if (isReferenced) {
    let confirmAwnser = confirm("There are existing transactions with this category. Deleting this category will remove them all!! Are you sure? ")
    if (confirmAwnser) {
      this.cascadeTransactions(id);
      this.categoryService.deleteCategory(id);
    }
  } else {
    this.categoryService.deleteCategory(id);
  }
}
```

First when we attempt to delete certain category it checks all the transactions and tries to find a transaction that has that category. If it finds a referenced transaction it ask us for confirmation because if we delete the category all the transactions that it's referenced in will be deleted.

If we confirm then it calls the method `cascadeTransactions` which iterates the transactions that have the mentioned category and deletes them. That method looks like this:

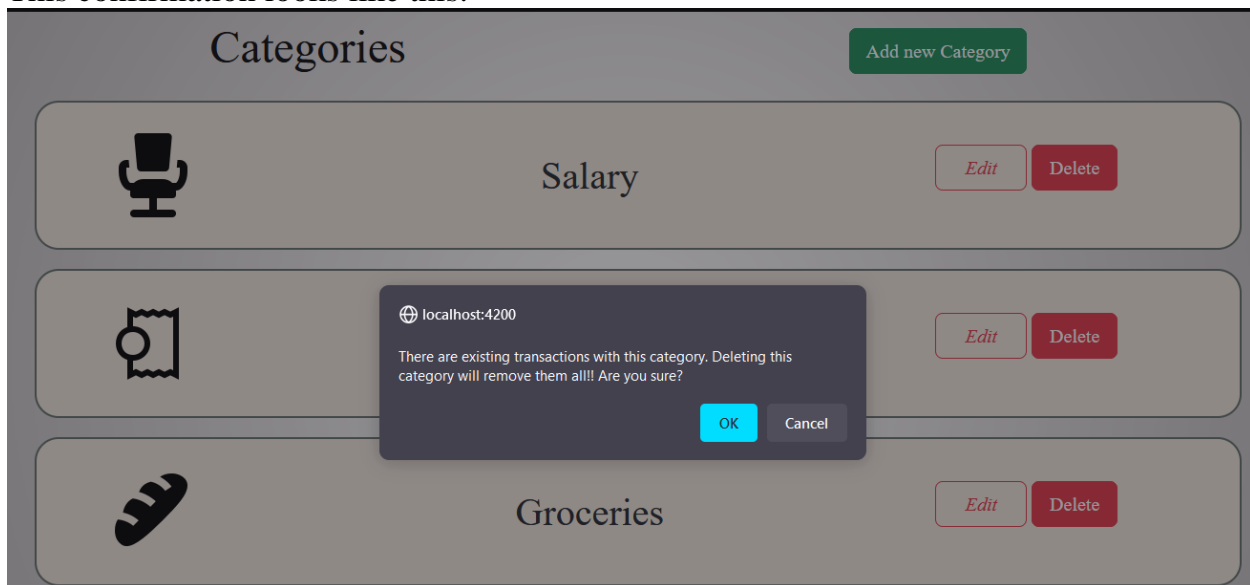
```
cascadeTransactions(id: number) {  
  this.transactionService.getTransactions().forEach(t => {  
    if (t.category?.id == id) {  
      this.transactionService.deleteTransaction(t.id)  
    }  
  })  
}
```

After this is completed and all of the transactions that contain the current category are deleted it calls the `deleteCategory` method from the category service which actually deletes the category. That method looks like this:

```
deleteCategory(id: number) {  
  let objIndex = this.categories.findIndex((c => c.id == id));  
  if(objIndex != -1){  
    this.categories.splice(objIndex,1);  
  }  
  this.categoryUpdated.emit();  
}
```

If the delete method from the shared service does not find any transactions that contain the mentioned category it calls the delete method from the category service without asking for confirmation since that won't do any damage.

This confirmation looks like this:



Spendings

This is how the spending component looks:

Your Spendings

Income:	\$3,000.00
Expense:	\$400.00
<hr/>	
Bills:	\$150.00
Household:	\$250.00
<hr/>	
Balance:	\$2,600.00

Add Transaction

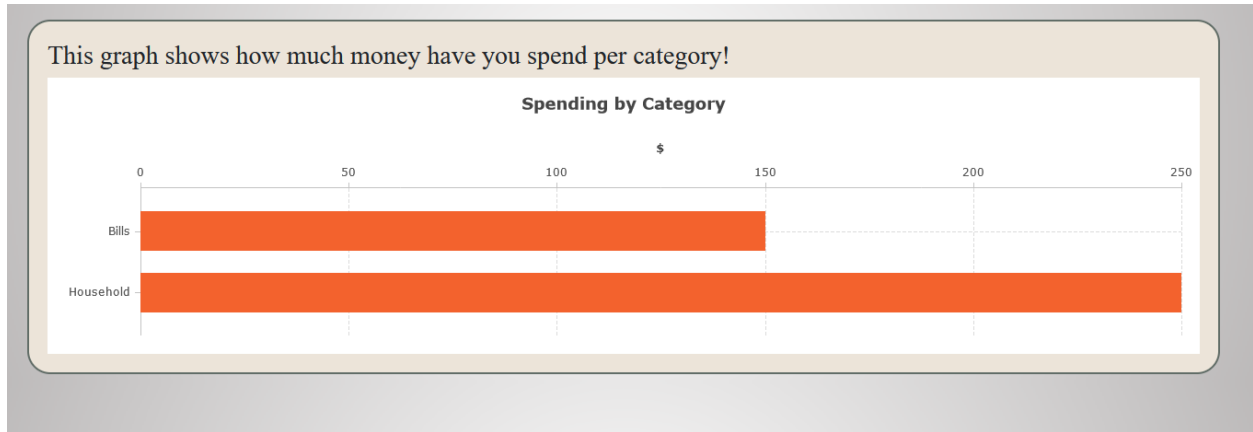
Here we can see that we have information about the income and the expense we had, but also we have information about the balance.

The main thing here is the listing of spendings by category which gives us information for the total spendings for each category that we have a transaction in. It calls the function `calculateExpensesByCategory` from the transaction service which gives us an array of every category and the spending we have in it.

```
calculateExpenseByCategory() {  
  const expenseByCategory = new Map();  
  this.transactions  
    .filter(transaction => transaction.type === 1 && transaction.category)  
    .forEach(transaction => {  
      const categoryName = transaction?.category?.name;  
      const amount = transaction.amount;  
      expenseByCategory.set(categoryName, (expenseByCategory.get(categoryName)  
|| 0) + amount);  
    });  
  
  return Array.from(expenseByCategory, ([category, totalExpense]) => ({  
category, totalExpense }));  
}
```

Graphs

If we go to the graphs section of the menu it will show the view for the Graph component. The component looks like this:



This component shows our spendings by category visually and more clearly with graphs using the same function to display the data that our spending component uses.

.