Project Report for the subject

“Advanced Web Design”

TripCost Calculator:

A consumption and travel expenses calculator

(mac. Калкулатор на потрошувачка и трошоци за патување)

Student: Leart Pajaziti
Index: 141050

Email: pajaziti.leart@students.finki.ukim.mk

Abstract

In an era marked by economic uncertainty and rising fuel prices, managing personal finances has become increasingly critical. This report presents a web application designed to assist users in calculating and estimating car trip expenses, providing a practical tool to navigate the complexities of travel budgeting. The application offers users accurate cost estimates, factoring in variables such as fuel efficiency, distance, and fuel prices. Through a user-friendly interface, the application aims to simplify financial planning for car trips, helping individuals make informed decisions in a challenging economic climate. This report outlines the development process, key features, and potential impacts of the application, highlighting its relevance in today's financial environment.

Table of Contents

Introduction

The current economic climate has introduced significant challenges for individuals and families, with rising fuel prices and economic instability placing additional strain on personal budgets. In response to these pressures, there is a growing need for tools that can provide clarity and control over travel expenses. The development of a web application that estimates car trip costs emerges as a practical solution to this need.

This report explores the creation of a web-based platform designed to help users calculate and manage their travel expenses. By integrating real-time data and user inputs, the application aims to deliver precise estimates of trip costs, including variables such as fuel consumption, distance, and fuel prices. The project's objective is to empower users with actionable insights, enabling them to plan their trips more effectively and make informed financial decisions amidst economic uncertainty.

The introduction of this web application is timely, given the current economic crisis characterized by volatile fuel prices and widespread financial strain. By providing a straightforward and accessible tool for estimating car trip expenses, the application addresses a critical gap in personal finance management, offering users a practical means to better navigate the complexities of travel budgeting in challenging economic conditions. This report will delve into the development process, technological framework, and anticipated benefits of the application, underscoring its significance in enhancing financial planning for car travel.

Development process

This web application was build using React and MaterialUI.

React is a popular JavaScript library developed by Facebook for building user interfaces, especially single-page applications where you need a dynamic and interactive user experience. It allows developers to create reusable UI components and manage the state of an application efficiently.

Material-UI (now known as MUI) is a popular React component library that implements Google's Material Design guidelines. It provides a set of pre-designed and customizable UI components that follow Material Design principles, making it easier to build modern and visually appealing web applications.

For this car trip expense calculator, MUI helped rapidly prototype a visually appealing and user-friendly interface, handle responsiveness and accessibility with minimal effort, and ensure that the app follows modern design standards. This can lead to a more polished and professional user experience while allowing you to focus more on functionality and less on design.

To provide a clearer explanation of the development process, I will break it down based on the four key features of this website.

1. Car selection Feature

This allows the user to simply add his preferred car for the trip by selecting the cars' production year, make, model, and model engine option and get the vehicles' fuel consumption hassle-free.

The dataset is gathered from FuelEconomy.gov API. The official U.S. government site for fuel economy information, it is operated by the Department of Energy and the Environmental Protection Agency. The site provides access to general information, widgets to help car buyers, and fuel economy datasets. This dataset is intended for public access and use. Using axios for http requests and UseEffect hooks we fetch the necessary data.

```
// Fetch years from fueleconomy.gov API on component mount
useEffect(() => {
  const fetchYears = async () => {
      try {
          const response = await axios.get(
              'https://www.fueleconomy.gov/ws/rest/vehicle/menu/year'
          );
          const yearsData = response.data.menuItem.map(item => item.value);
          setYears(yearsData);
          setSelectedYear(yearsData[0]); // Select the first year by default
      } catch (error) {
          console.error('Error fetching years:', error);
      }
  };
  fetchYears();
}, []);
//Fetch makes based on selected year
useEffect(() => {
  const fetchMakes = async () => {
      try {
          if (selectedYear) {
              const response = await axios.get(
                  `https://www.fueleconomy.gov/ws/rest/vehicle/menu/make?year=${selectedYear}`
              );
              const makesData = response.data.menuItem.map(item => item.value);
              setMakes(makesData);
              setSelectedMake(makesData[0]); // Select the first make by default
          }
      } catch (error) {
          console.error('Error fetching makes:', error);
      }
  };
  fetchMakes();
}, [selectedYear]);
```

In the code above we **fetch** the list of years from the API when it mounts and populate the dropdown with those years. Users can then select a year from the dropdown, and the `selectedYear` state will be updated accordingly. Then we fetch the list of makes. Selecting a year dynamically updates the options available for makes based on the API response.

UseState variables: `years`, `makes`, `selectedYear`, and `selectedMake`

Dropdowns: Two Select components from Material-UI are used:

The first dropdown selects the year. It triggers a fetch for makes when the year changes.

The second dropdown selects the make, which is dynamically populated based on the selected year.
HTML and MaterialUI:

```
        {/* CAR SELECTION PART */}
        <Paper sx={{ padding: '40px'}} elevation={12}>
            <Stack spacing={2} direction='column'>
                <Typography variant="h4" color='primary'>Choose car:</Typography>

                <FormControl fullWidth>
                  <InputLabel>Select Year</InputLabel>
                    <Select
                        value={selectedYear}
                        onChange={handleYearChange}
                        label="Select Year"
                    >
                        {years.map((year) => (
                            <MenuItem key={year} value={year}>
                                {year}
                            </MenuItem>
                        ))}
                    </Select>
                </FormControl>

                <FormControl fullWidth style={{ marginTop: '1rem' }}>
                    <InputLabel>Select Make</InputLabel>
                    <Select
                        value={selectedMake}
                        onChange={handleMakeChange}
                        label="Select Make"
                        disabled={!selectedYear} // Disable until year is selected
                    >
                        {makes.map((make) => (
                            <MenuItem key={make} value={make}>
                                {make}
                            </MenuItem>
                        ))}
                    </Select>
                </FormControl>
```

Event Handlers: handleYearChange updates selectedYear, triggering a re-fetch of makes. handleMakeChange updates selectedMake.

```
  // Handle change in selected year
  const handleYearChange = (event) => {
    setSelectedYear(event.target.value);
  };

  // Handle change in selected make
  const handleMakeChange = (event) => {
    setSelectedMake(event.target.value);
  };
```

The second Select for makes is disabled (disabled={!selectedYear}) until a year is selected, ensuring the user selects a year first.

In the same fashion as explained here we fetch Models, Options as well as carMpg by using the cars id number.

2. Route selection Feature

The user is presented with two Textfield elements for entering the Starting City and the Destination City. The input is saved in 2 **usestate variables**: startingCity and destinationCity.
The user then can add the route distance in another Textfield or he can click the button: “Calculate distance using google maps”. Using Google maps api this opens directions and route details for the previously entered cities. The user is allowed to customize his preferred route and then enter the distance in kilometers.

```
  const handleCalculate = () => {
    const formattedStartingCity = encodeURIComponent(startingCity.trim());
    const formattedDestinationCity = encodeURIComponent(destinationCity.trim());

    const googleMapsUrl =
`https://www.google.com/maps/dir/?api=1&origin=${formattedStartingCity}&destination=${formattedDestinationCit
y}`;

    window.open(googleMapsUrl, '_blank');
  };
```

- startingCity and destinationCity are state variables that hold the user input.
- trim() is used to remove any leading or trailing whitespace from the input strings.
- -encodeURIComponent is a JavaScript function that encodes special characters in the input strings to make them URL-safe. This prevents issues with spaces, special characters, or non-ASCII characters in URLs.

Constructing the Google Maps URL:

const googleMapsUrl =
`https://www.google.com/maps/dir/?api=1&origin=${formattedStartingCity}&destination=${formattedDestinationCity}`;

- This line constructs the **URL** that will be used to open Google Maps with directions.
- https://www.google.com/maps/dir/ is the base URL for Google Maps directions.
- **?api=1** is a query parameter indicating that you want to use the Google Maps API for directions.
- origin=${formattedStartingCity} sets the starting point of the route.
- destination=${formattedDestinationCity} sets the endpoint of the route.

By using template literals (backticks `), you can easily insert variables (formattedStartingCity and formattedDestinationCity) directly into the URL string.

Opening the URL in a New Tab:

window.open(googleMapsUrl, '_blank');

- window.open is a JavaScript method that opens a new browser window or tab.

- googleMapsUrl is the URL to be opened.
- '_blank' is the target attribute that specifies to open the URL in a new tab. If omitted or set to '_self', the URL would open in the same tab.

3. Vignettes and tolls Feature

The user is presented with 2 buttons: “Add vignette costs” and “Add toll costs”. This is done to minimize clutter and keep the page clean and simple to use. This also lets the user know that adding vignettes and toll costs is not mandatory but rather optional for calculating the total costs.
When the buttons are clicked the user is greeted with a modal/dialog window where he is allowed to add vignette/toll notes and their respective costs. An add button is present for adding a new row, and a close button for closing this modal/dialog element.

```
// Handle Vignette
const handleOpenVignetteModal = () => {
  setTempVignetteCosts([{ country: '', cost: '' }]);
  setVignetteModalOpen(true);
};

const handleCloseVignetteModal = () => {
  setVignetteCosts(tempVignetteCosts);
  setVignetteModalOpen(false);
};

const handleVignetteCostChange = (index, field, value) => {
  const newVignetteCosts = [...tempVignetteCosts];
  newVignetteCosts[index][field] = value;
  setTempVignetteCosts(newVignetteCosts);
};

const handleAddVignetteRow = () => {
  setTempVignetteCosts([...tempVignetteCosts, { country: '', cost: '' }]);
};
```

Usestate variables:

- vignetteCosts: Stores the final list of vignette costs.
- vignetteModalOpen: Controls the visibility of the modal.
- tempVignetteCosts: Temporarily holds vignette costs while the modal is open.

Opening and Closing the Modal:

- handleOpenVignetteModal(): Initializes the temporary vignette costs and opens the modal.
- handleCloseVignetteModal(): Saves the temporary vignette costs to vignetteCosts, closes the modal, and updates the display of the total vignette cost.

Handling Vignette Cost Changes:

- handleVignetteCostChange(index, field, value): Updates the specific field (country or cost) of a vignette cost at a given index.

Adding More Rows:

- handleAddVignetteRow(): Adds a new row to the temporary vignette costs list.

Calculating and Displaying Total Vignette Cost:

- calculateTotalVignetteCost(): Computes the total vignette cost by summing up the cost fields of all vignette items. It uses parseFloat to convert cost values to numbers and .toFixed(2) to format the total to two decimal places.

HTML and materialUI code:

```
{/* Vignette Costs Modal */}
<Dialog open={vignetteModalOpen} onClose={() => setVignetteModalOpen(false)}>
  <DialogTitle>
    Vignette Costs
  </DialogTitle>
  <DialogContent>
    {tempVignetteCosts.map((vignette, index) => (
      <Grid container spacing={2} key={index} alignItems="center">
        <Grid item xs={5}>
          <TextField
            label="Country"
            variant="outlined"
            fullWidth
            value={vignette.country}
            onChange={(e) => handleVignetteCostChange(index, 'country', e.target.value)}
          />
        </Grid>
        <Grid item xs={5}>
          <TextField
            label="Cost"
            variant="outlined"
            fullWidth
            type="number"
            value={vignette.cost}
            onChange={(e) => handleVignetteCostChange(index, 'cost', e.target.value)}
          />
        </Grid>
        <Grid item xs={2}>
          <Button
            variant="contained"
            color="primary"
            onClick={handleAddVignetteRow}
            style={{ height: '100%' }}
          >
            Add
          </Button>
        </Grid>
```

```
            </Grid>
          ))}
        </DialogContent>
        <DialogActions>
          <Button onClick={handleCloseVignetteModal} color="primary">
            Close
          </Button>
        </DialogActions>
      </Dialog>
```

In the same fashion as explained here we create the modal/dialog for Tolls.

4. Cost Calculation Feature

After the user selects the his car, enters the route distance, fuel cost per litre, vignette and toll expenses he can now view the total cost for the trip in euros.
He is also presented with the “Total Vignette Cost”, “Total Toll Cost” as well as “Total Fuel Cost”.

```
const convertMpgToLpkm = (mpg) => {
  const mpgValue = parseFloat(mpg) || 0;
  return (235.215 / mpgValue).toFixed(2);
};
```

convertMpgToLpkm(mpg) converts MPG to litres per 100 kilometres using the formula:

LPKM = 235.215 / MPG.

```
const calculateTotalFuelCost = () => {
  const distanceInKm = parseFloat(distance) || 0;
  const pricePerLitre = parseFloat(fuelPrice) || 0;
  const fuelConsumption = parseFloat(convertMpgToLpkm(carMpg)) || 0;
  return ((distanceInKm / 100) * fuelConsumption * pricePerLitre).toFixed(2);
};
```

calculateTotalFuelCost() uses the converted litres per 100 km value to compute the total fuel cost based on distance and fuel price.

```
const calculateTotalVignetteCost = () => {
  return vignetteCosts.reduce((total, { cost }) => total + parseFloat(cost || 0), 0).toFixed(2);
};

const calculateTotalTollCost = () => {
  return tollCosts.reduce((total, { cost }) => total + parseFloat(cost || 0), 0).toFixed(2);
};
```

calculateTotalVignetteCost() adds all the costs in the Vignette modal/dialog using the reduce method. “reduce” is a JavaScript array method that iterates over the array and accumulates a single result.
calculateTotalTollCost() adds all the costs in the Toll modal/dialog.

```
const calculateTotalCost = () => {
  const totalVignetteCost = parseFloat(calculateTotalVignetteCost()) || 0;
  const totalTollCost = parseFloat(calculateTotalTollCost()) || 0;
  const totalFuelCost = parseFloat(calculateTotalFuelCost()) || 0;
  return (totalVignetteCost + totalTollCost + totalFuelCost).toFixed(2);
};
```

calculateTotalCost() calculates the sum of all the above mentioned costs.

Future improvements

While TripCost Calculator is a simple and effective cost estimation calculator, there are a lot of future functionalities and possibilities that could provide the user with an even better and detailed user experience all the while maintaining its simple yet user-friendly nature.

- Addition of a **login page**
- **Save Favorite** trips: allows the user to login to his account and save trips in the Favourites page. This helps the user to keep track and compare the expenses of different saved trip combinations.
- Filter/order favorite trips: allows the user to filter or order his saved favourite trips according to Date, Cost, Car etc.

Conclusion

In summary, this web application represents a valuable tool for users seeking to optimize their travel expenses and fuel consumption. Its effective combination of advanced web technologies, accurate calculations, and user-friendly design exemplifies the potential of digital solutions in enhancing personal financial management at a crucial time, given the current economic climate and the volatility in fuel prices.