THE ROLLING CAPSTONES

AI TRIP CREATOR

CODING STANDARDS

TABLE OF CONTENTS

1. Introduction

This document outlines the coding standards to be followed while developing a React application with a Firebase backend and MUI component library with CSS for styling. It also includes guidelines for handling API endpoints within the frontend codebase.

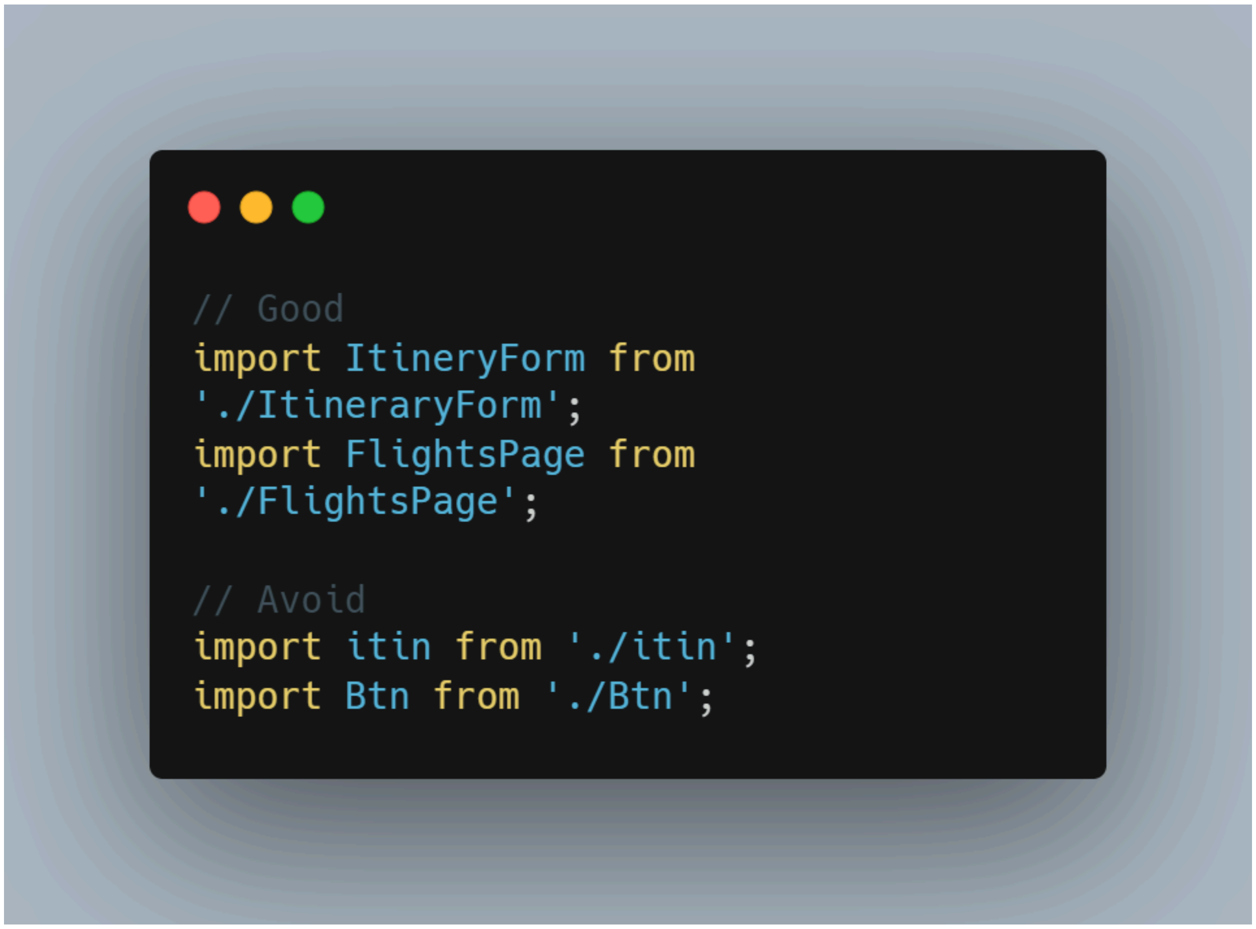
2. Folder Structure

- Organise files and folders logically based on their functionalities.
- Separate concerns by categorizing files into directories such as components, pages, utils, etc.
- Group related components or modules together within their respective directories.

3. Naming Conventions

3.1 React naming Conventions

- Use descriptive and meaningful names for React components.
- Components should be named using PascalCase convention.
- Components representing pages should be suffixed with ”Page”.
- Components representing reusable UI elements should be suffixed with ”Component”.



```
// Good
import ItineryForm from
'./ItineraryForm';
import FlightsPage from
'./FlightsPage';

// Avoid
import itin from './itin';
import Btn from './Btn';
```

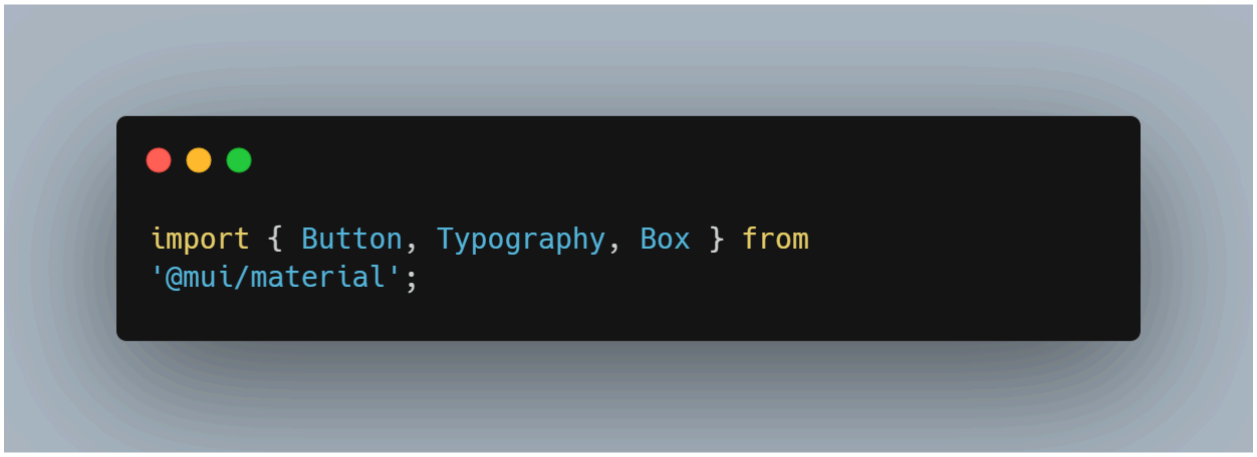
Figure 2: Example of React naming convention

4. CSS, Styling and Component Library

4.1 MUI Component library

4.1.1 Component Importing:

- Import components directly from `@mui/material` to leverage tree-shaking.

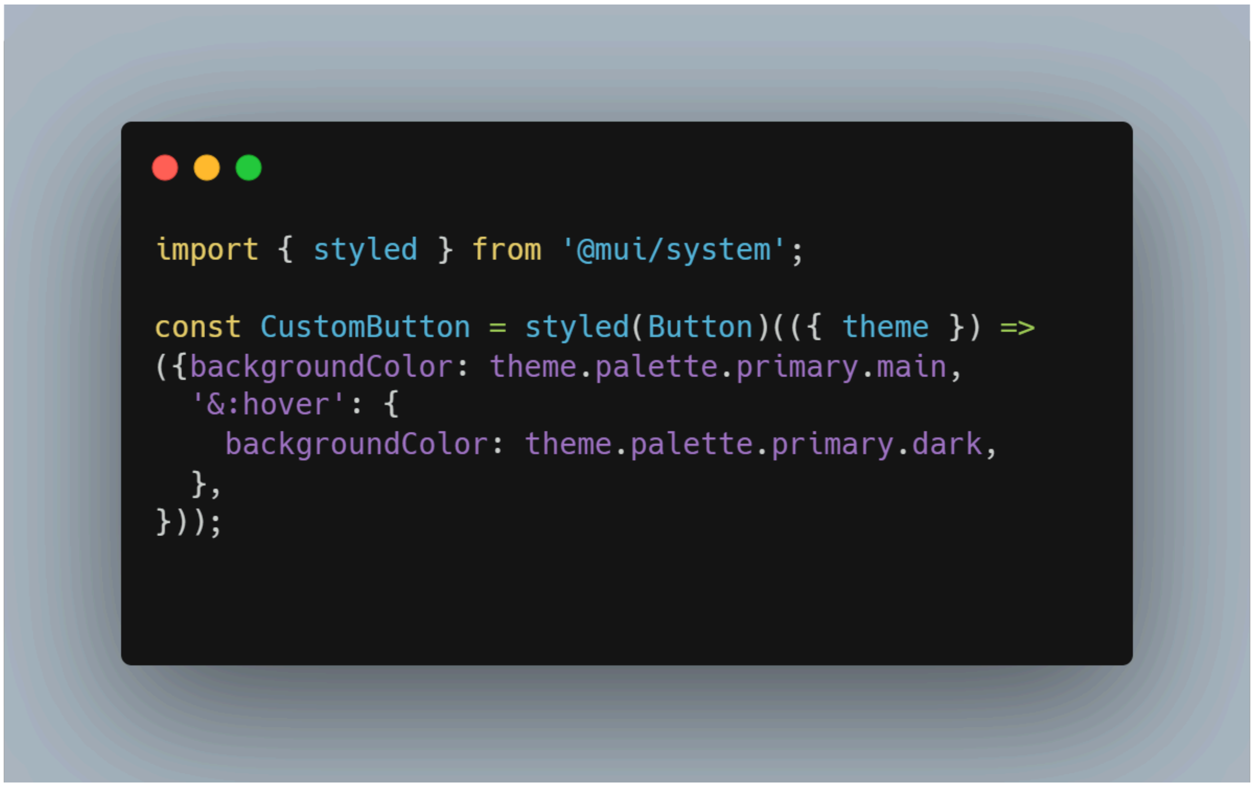


```
import { Button, Typography, Box } from
'@mui/material';
```

Figure 4.1.1: Example of importing components from MUI component library

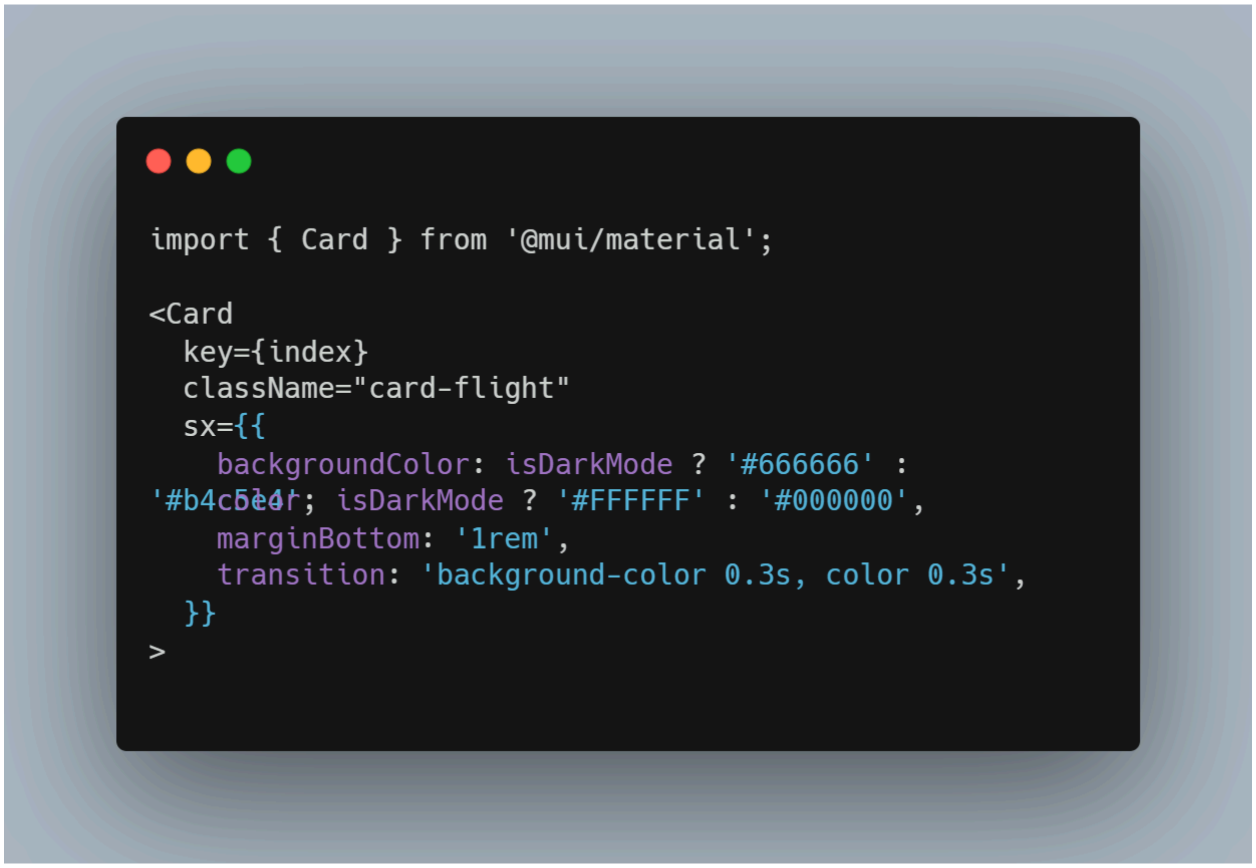
4.1.2 Component Customisation:

- Use `sx` prop for inline styling whenever possible to ensure consistent theming.
- Use `styled` API from `@mui/system` for more complex component customization.



```
import { styled } from '@mui/system';

const CustomButton = styled(Button)(({ theme }) =>
({backgroundColor: theme.palette.primary.main,
  '&:hover': {
    backgroundColor: theme.palette.primary.dark,
  },
}));
```

Figure 4.1.2: Example of component customisation

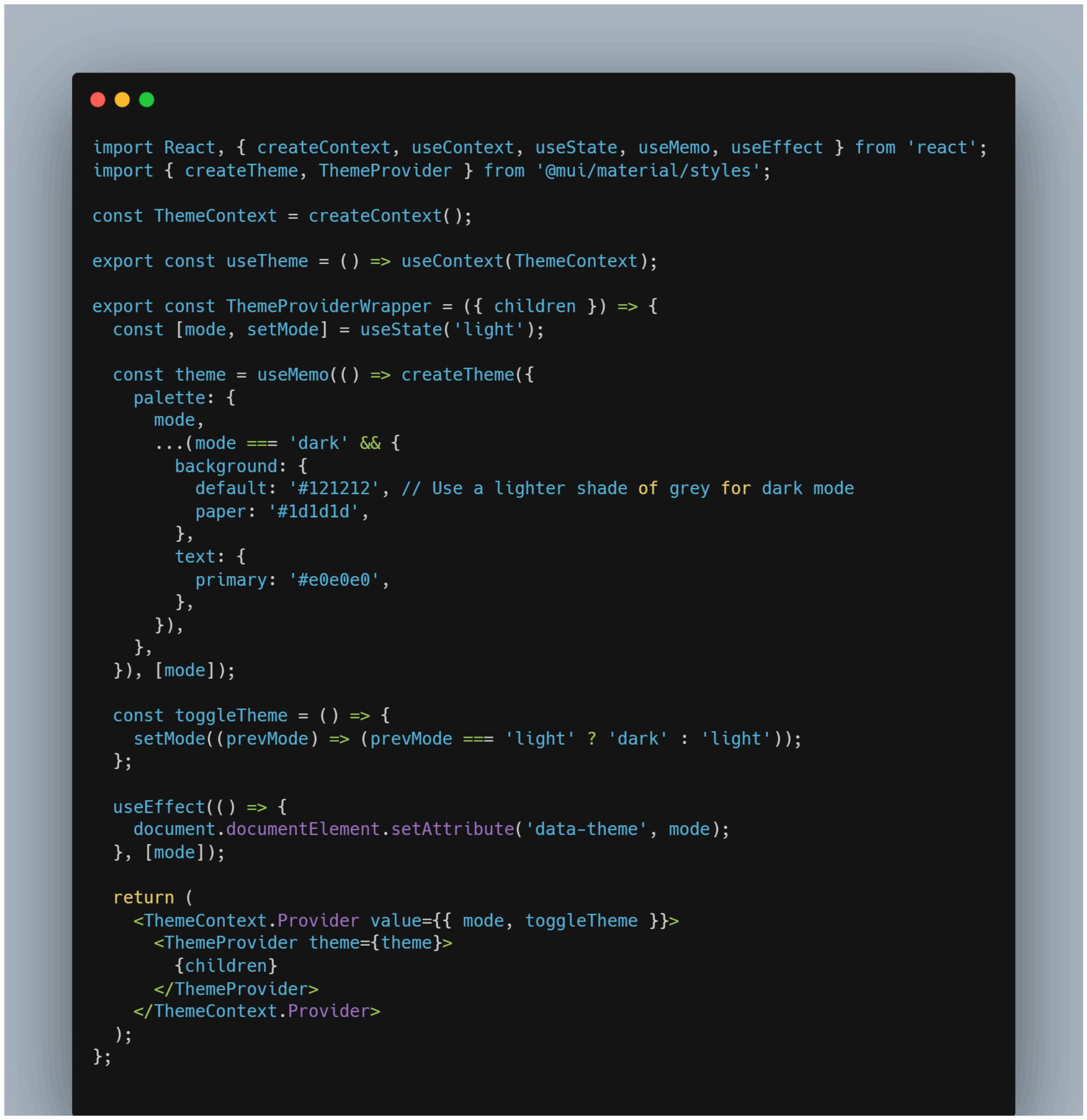


```
import { Card } from '@mui/material';

<Card
  key={index}
  className="card-flight"
  sx={{
    backgroundColor: isDarkMode ? '#666666' :
'#b4c5e4', color: isDarkMode ? '#FFFFFF' : '#000000',
    marginBottom: '1rem',
    transition: 'background-color 0.3s, color 0.3s',
  }}
>
```

4.1.2.1 Another example of component customisation.

4.1.3 Theming:

- Utilize the theme object for consistent styling.
- Define custom themes using `createTheme` and `ThemeProvider`.

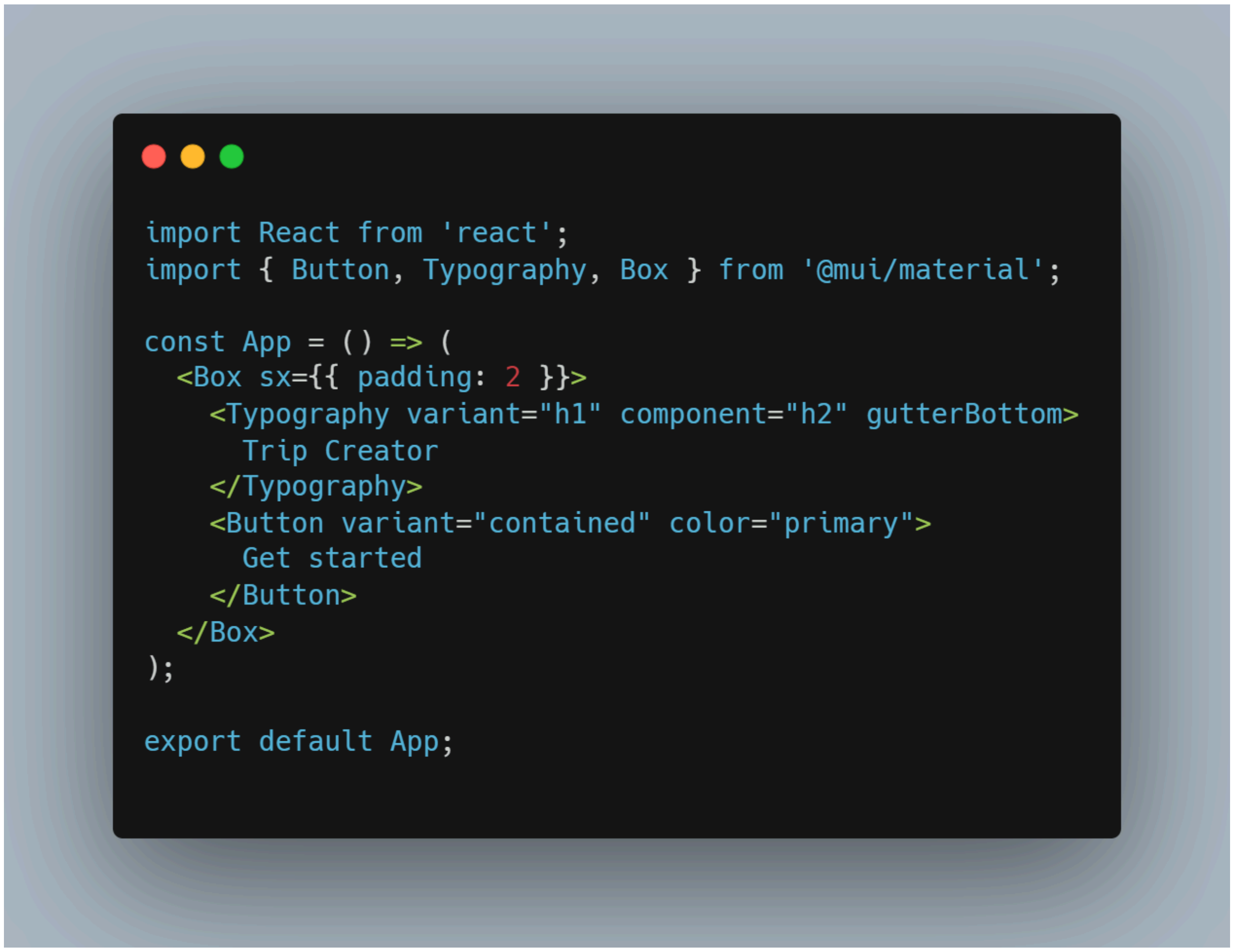


```
import React, { createContext, useContext, useState, useMemo, useEffect } from 'react';
import { createTheme, ThemeProvider } from '@mui/material/styles';

const ThemeContext = createContext();

export const useTheme = () => useContext(ThemeContext);

export const ThemeProviderWrapper = ({ children }) => {
  const [mode, setMode] = useState('light');

  const theme = useMemo(() => createTheme({
    palette: {
      mode,
      ...(mode === 'dark' && {
        background: {
          default: '#121212', // Use a lighter shade of grey for dark mode
          paper: '#1d1d1d',
        },
        text: {
          primary: '#e0e0e0',
        },
      }),
    },
  }), [mode]);

  const toggleTheme = () => {
    setMode((prevMode) => (prevMode === 'light' ? 'dark' : 'light'));
  };

  useEffect(() => {
    document.documentElement.setAttribute('data-theme', mode);
  }, [mode]);

  return (
    <ThemeContext.Provider value={{ mode, toggleTheme }}>
      <ThemeProvider theme={theme}>
        {children}
      </ThemeProvider>
    </ThemeContext.Provider>
  );
};
```

Figure 4.1.3: Example of how we used Theming from the MUI component library and React library

4.1.4 Code Readability:

- Keep component usage clear and concise.
- Avoid deeply nested components.
- Break down large components into smaller, reusable components.



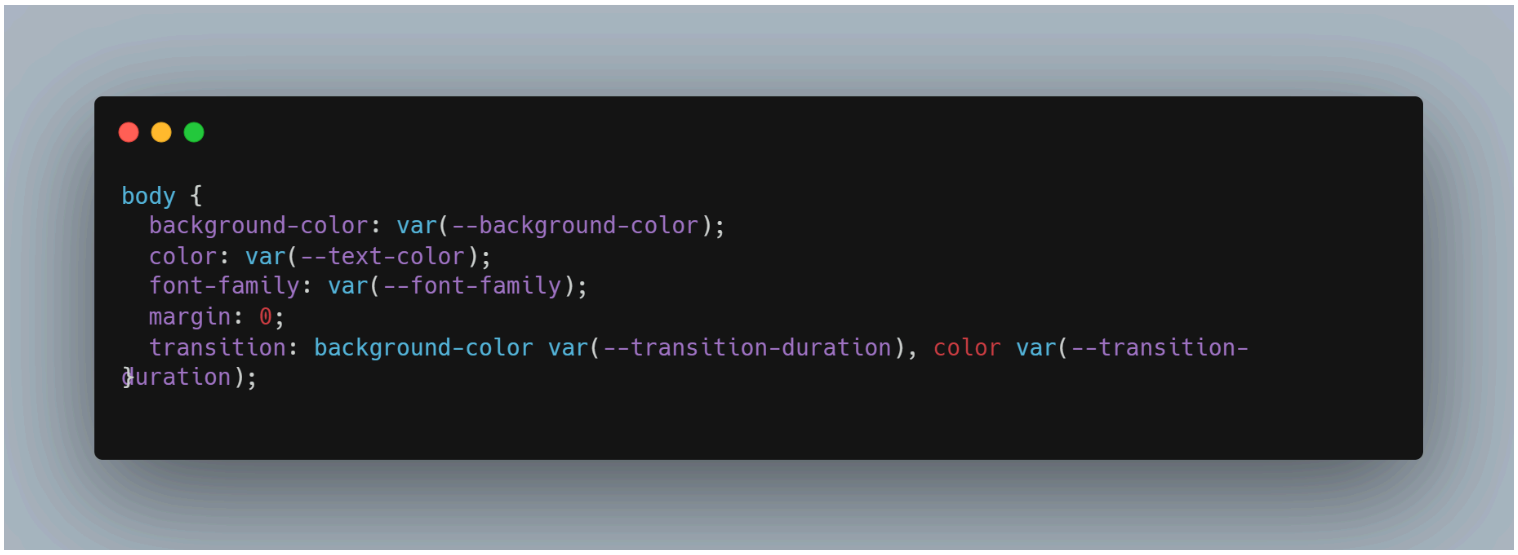
```jsx
import React from 'react';
import { Button, Typography, Box } from '@mui/material';

const App = () => (
  <Box sx={{ padding: 2 }}>
    <Typography variant="h1" component="h2" gutterBottom>
      Trip Creator
    </Typography>
    <Button variant="contained" color="primary">
      Get started
    </Button>
  </Box>
);

export default App;
```

Figure 4.1.4 Example of code readability in the program.

4.1 CSS Styling

4.2.1 General:

- Use a consistent style format
- Avoid overly specific selectors.
- Use class selectors over ID selectors to promote reusability.
- Define a base font size and use relative units.
- Keep CSS files as small as possible.
- Remove unused styles.

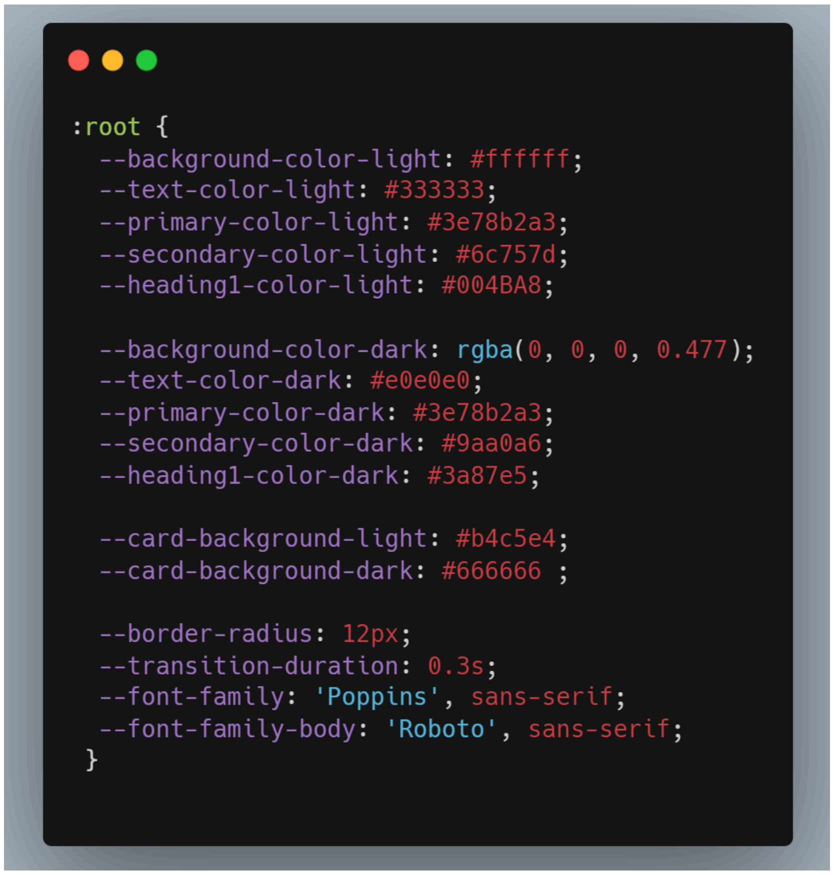


```css
body {
  background-color: var(--background-color);
  color: var(--text-color);
  font-family: var(--font-family);
  margin: 0;
  transition: background-color var(--transition-duration), color var(--transition-
}uration);
```

Figure 4.2.1 Example of general styling in CSS

4.2.2 CSS Variables:

- Use CSS variables for theme colours, fonts, and other repetitive values.



```css
:root {
  --background-color-light: #ffffff;
  --text-color-light: #333333;
  --primary-color-light: #3e78b2a3;
  --secondary-color-light: #6c757d;
  --heading1-color-light: #004BA8;

  --background-color-dark: rgba(0, 0, 0, 0.477);
  --text-color-dark: #e0e0e0;
  --primary-color-dark: #3e78b2a3;
  --secondary-color-dark: #9aa0a6;
  --heading1-color-dark: #3a87e5;

  --card-background-light: #b4c5e4;
  --card-background-dark: #666666 ;

  --border-radius: 12px;
  --transition-duration: 0.3s;
  --font-family: 'Poppins', sans-serif;
  --font-family-body: 'Roboto', sans-serif;
 }
```

Figure 4.2.2 Example of CSS variables

4.2.3 Performance:

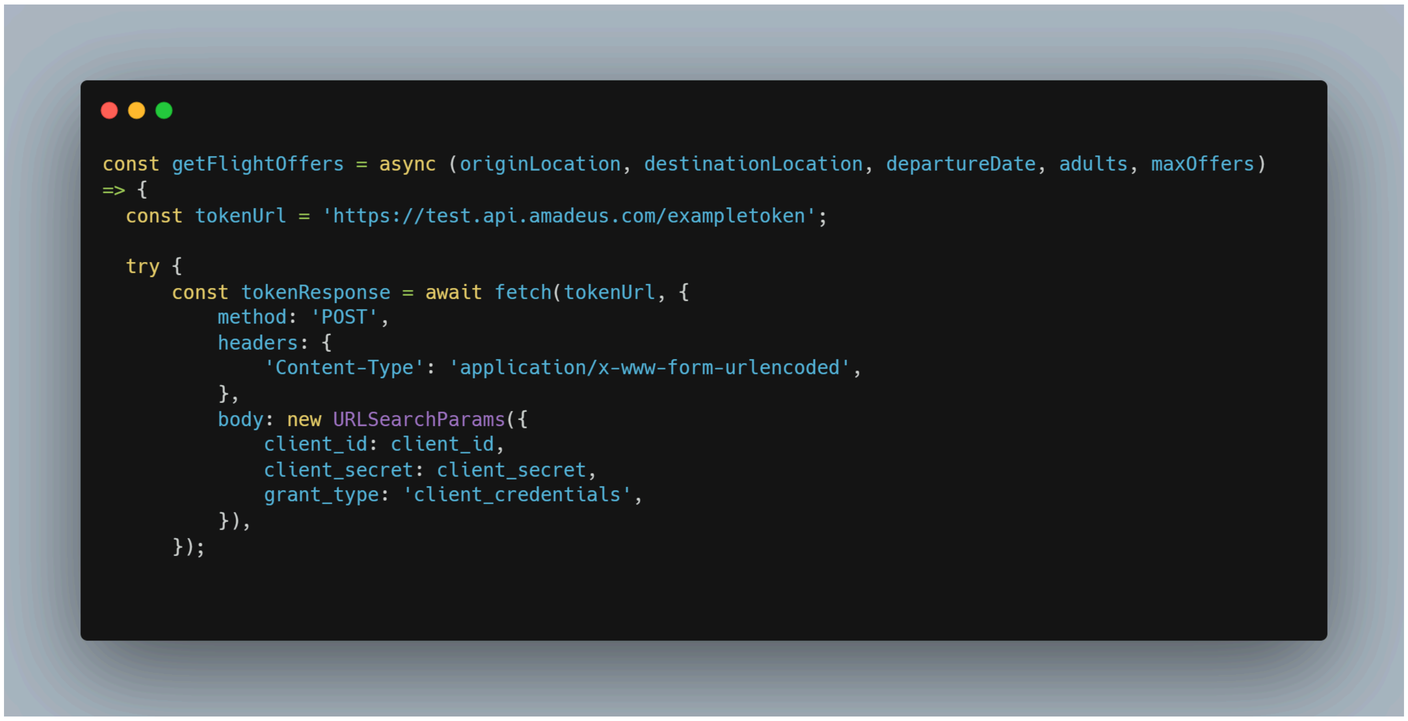
- Minimise the use of complex selectors.
- Avoid using `!important` unless absolutely necessary.

5. Code Formatting

- Use consistent indentation (preferably 2 or 4 spaces) for improved readability. Follow JavaScript best practices for code formatting.
- Utilise ESLint for enforcing code consistency.
- Configure ESLint rules according to project requirements and coding standards.
- Regularly format and lint code to maintain code quality.

6. API Endpoint Handling

- Centralise API endpoint configurations and handling.
- Utilize asynchronous functions for making API requests.
- Abstract API logic into separate modules for reusability and maintainability.



```
const getFlightOffers = async (originLocation, destinationLocation, departureDate, adults, maxOffers)
=> {
  const tokenUrl = 'https://test.api.amadeus.com/exampletoken';

  try {
      const tokenResponse = await fetch(tokenUrl, {
          method: 'POST',
          headers: {
              'Content-Type': 'application/x-www-form-urlencoded',
          },
          body: new URLSearchParams({
              client_id: client_id,
              client_secret: client_secret,
              grant_type: 'client_credentials',
          }),
      });
```

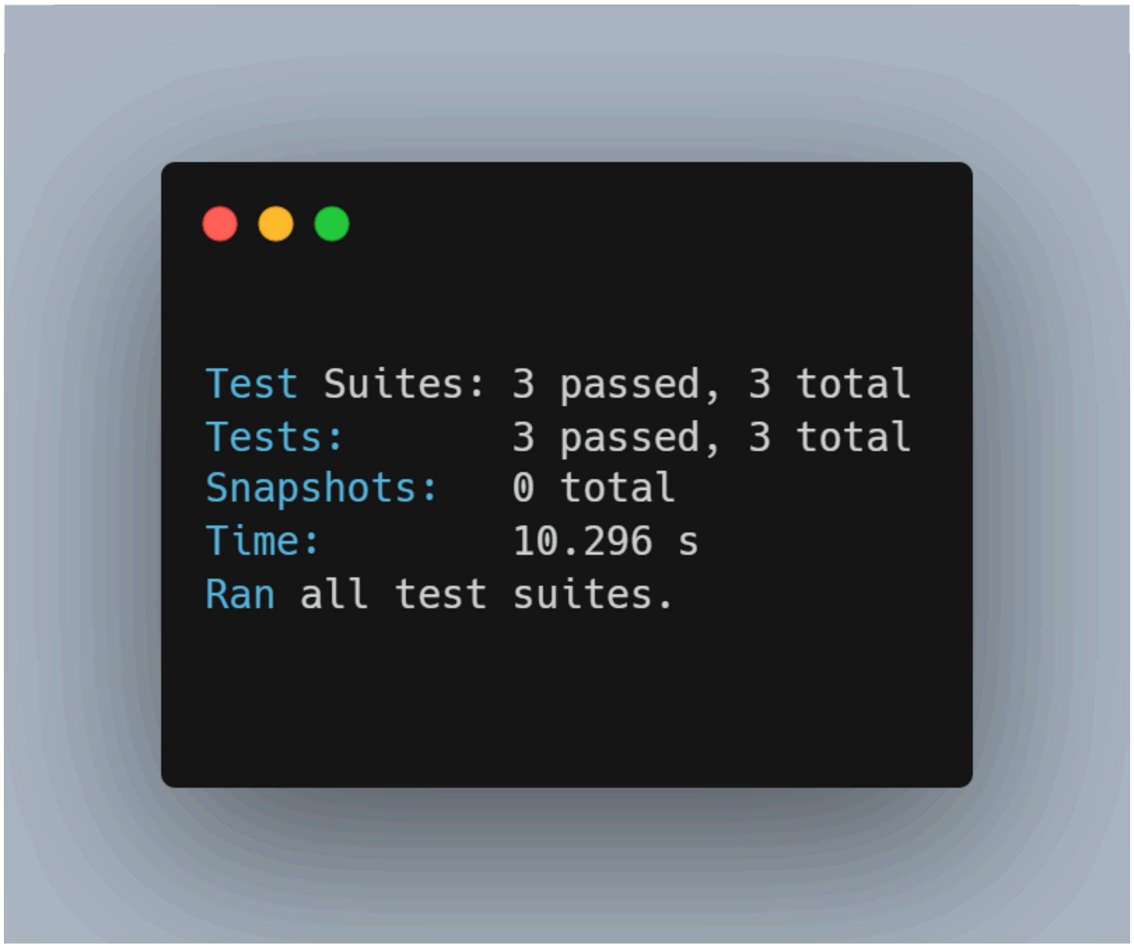
Figure 6: Example of API endpoint handling

7. Version Control

- Utilise version control system Git for tracking changes and collaborating with team members.
- Follow branching strategies Git Flow for managing feature development and releases.
- Write meaningful commit messages that describe the purpose and scope of changes.
- Regularly push changes to remote repositories and pull updates from upstream branches.

8. Testing

- Implement unit tests, integration tests, and end-to-end tests to ensure code reliability and functionality.
- Use testing frameworks Jest for writing tests in React applications.
- Use Cypress for End-to-End Testing.
- Follow test-driven development (TDD) practices when applicable.
- Automate testing processes as much as possible to streamline development workflows.



```
Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        10.296 s
Ran all test suites.
```

Figure 8: Tests passed