Railway Ticket Management System

Team
CS2108 Aditya Nana Bhadane
CS2165 Priyanka Sanjay Mutkule
CS2124 Atmanand Dnyaneshwar Nagpure

Course Faculty
Prof. Swati Khidse

Table of contents

Idea

Introduction

- Railway Ticket Management System is a computer based ticket booking system.
- Allows users to book train tickets with ease.
- Users can find trains, book train tickets, cancel ticket, and make payments.
- Multiple users supported

Features

- User register and login
- Book a ticket, display booked tickets, cancel ticket, make payment
- Display profile

Demonstration

OUTPUT TERMINAL DEBUG CONSOLE

cmd

```
C:\Users\lenovo\Projects\BadRailwayManager>
```

Code

- Main application is divided into screens.
- authScreen() handles user registration and login.
- mainScreen() handles ticket booking and other operations.
- All the data is saved in application's context and is passed to all screens and components of the application.

```c
AppContext context = {
    users,
    trains,
    tickets,
    .currentUser = NULL};

ReturnCode ret;

while (true)
{
  if (!context.currentUser)
  {
    ret = authScreen(&context);
  }
  else
  {
    ret = mainScreen(&context);
  }

  if (ret == EXIT)
  {
    break;
  }
}

printf("Exitting...\n");

listFree(users);
listFree(trains);
listFree(tickets);
```

Code

- Auth screen is divided into 2 options, “Create an account” and “Log in”.
- createAccountActivity() function handles account creation process.
- loginActivity() function handles user login process.
- User can directly exit the application by choosing the “Exit” option.

```c
ReturnCode authScreen(AppContext *context)
{
  int choice;

  while (true)
  {
    clearScreen();
    printf(
        "[ MENU ]\n"
        "1. Create an account\n"
        "2. Log in\n"
        "3. Exit\n");

    printf("Enter your choice: ");
    choice = getInt();

    switch (choice)
    {
    case 1:
    {
      createAccountActivity(context);
      break;
    }
    case 2:
    {
      User *res = loginActivity(context);
      if (res)
      {
        return SUCCESS;
      }
      break;
    }
    case 3:
    {
      return EXIT;

      break;
    }
```

Code

- Main screen provides a list of actions which a logged-in user can perform.
- All actions are handled in their own functions.
- Application context is passed to each action.

```c
ReturnCode mainScreen(AppContext *context)
{
    clearScreen();
    int choice;

    printf(
        "[ MENU ]\n"
        "1. Book a ticket\n"
        "2. List booked tickets\n"
        "3. Cancel a ticket\n"
        "4. Pay\n"
        "5. Display details\n"
        "6. Log out\n"
        "7. Exit\n");
    printf("Enter your choice: ");
    choice = getInt();

    switch (choice)
    {
    case 1:
    {
        // Book a ticket
        bookTicket(context);
        break;
    }
    case 2:
    {
        // List booked tickets
        listBookedTickets(context);
        break;
    }
    case 3:
    {
        // Cancel a ticket
        cancelTicket(context);
        break;
    }
    case 4:
    {
        // Pay
        pay(context);
        break;
    }
    case 5:
    {
        // Display user details
        displayUserDetails(context);
        break;
    }
    case 6:
    {
        // Log out
        logout(context);

        break;
    }
    case 7:
    {
        // Exit
        return EXIT;
    }
    default:
    {
        printf("Invalid option chosen!\n");
    }
    }

    return 0;
}
```

Techniques

Preprocessing

- 3 main entities -
  - User
  - Train
  - Ticket
- Data is stored in linked lists.
- Writing linked list each time for a new entity leads to code duplication.
- **C preprocessing macros** were used to generate CRUD (Create, Read, Update, Delete) functionality automatically.

```c
#define GENERATE_CRUD(type)
  typedef struct type##Node type##Node;

  struct type##Node
  {
    type data;
    type##Node *next;
  };

  typedef struct
  {
    type data;
    CompareFunction compare;
  } type##Condition;

  void *type##Create(void *value)
  {
    type##Node *node = CREATE(type##Node, 1);
    node->next = NULL;
    node->data = *((type *)value);

    return node;
  }

  void *type##GetData(void *node)
  {
    type##Node *n = (type##Node *)node;
    return ((void *)&n->data);
  }

  void *type##GetNext(void *node)
  {
    type##Node *n = (type##Node *)node;
    return ((void *)n->next);
  }

  bool type##SetData(void *node, void *value)
```

Techniques

Event Handling

- List events -
  - Insert
  - Update
  - Delete
- Event handlers can be added to perform operations after a list action is performed.
- For eg., after an element is inserted in the list, the inserted element can be written to a file or saved in a database.

```
void *on##type##Event(EventType type, void *container, void *data)  \
{                                                                   \
  switch (type)                                                     \
  {                                                                 \
  case INSERT:                                                      \
  {                                                                 \
    List *list = (List *)container;                                 \
    type##Node *node = (type##Node *)data;                          \
    BasicMetadata *metaData = (BasicMetadata *)list->context->meta; \
                                                                    \
    metaData->counter++;                                            \
    node->data.id = metaData->counter;                              \
                                                                    \
    break;                                                          \
  }                                                                 \
  case UPDATE:                                                      \
  {                                                                 \
    break;                                                          \
  }                                                                 \
  case DELETE:                                                      \
  {                                                                 \
    break;                                                          \
  }                                                                 \
  }                                                                 \
  return NULL;                                                      \
}                                                                   \
```

Fig. Event handler increments the element counter and gives an ID to the newly inserted element.

Conclusion

- Data structures provide a simple interface and abstract out the complexity.
- Design patterns simplify the application flow and help in creating a consistent, repeatable design process.
- For writing applications rapidly, the structure of code has to be consistent yet flexible.
- Code can be divided into modules to allow multiple developers to work on the project.

References/Links

- Source Code

  - GitHub: https://github.com/proghax333/BadRailwayManager

Thank you!