

# 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

	Topic	Slide No.
1	Idea	3
2	Demonstration	4
3	Code	5
4	Techniques	8
5	Conclusion	10
6	References/Links	11

# 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.

```
594     AppContext context = {
595         users,
596         trains,
597         tickets,
598         .currentUser = NULL};
599
600     ReturnCode ret;
601
602     while (true)
603     {
604         if (!context.currentUser)
605         {
606             ret = authScreen(&context);
607         }
608         else
609         {
610             ret = mainScreen(&context);
611         }
612
613         if (ret == EXIT)
614         {
615             break;
616         }
617     }
618
619     printf("Exiting...\n");
620
621     listFree(users);
622     listFree(trains);
623     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.

```

521 ReturnCode mainScreen(AppContext *context)
522 {
523     int choice;
524
525     while (true)
526     {
527         clearScreen();
528         printf(
529             "[ MENU ]\n"
530             "1. Create an account\n"
531             "2. Log in\n"
532             "3. Exit\n");
533
534         printf("Enter your choice: ");
535         choice = getInt();
536
537         switch (choice)
538         {
539             case 1:
540             {
541                 createAccountActivity(context);
542                 break;
543             }
544             case 2:
545             {
546                 User *res = loginActivity(context);
547                 if (res)
548                 {
549                     return SUCCESS;
550                 }
551                 break;
552             }
553             case 3:
554             {
555                 return EXIT;
556
557                 break;
558             }
559         }
560     }
561 }

```

# 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.

```
451 ReturnCode mainScreen(AppContext *context) 482
452 { 483
453     clearScreen(); 484
454     int choice; 485
455 486
456     printf( 487
457         "[ MENU ]\n" 488
458         "1. Book a ticket\n" 489
459         "2. List booked tickets\n" 490
460         "3. Cancel a ticket\n" 491
461         "4. Pay\n" 492
462         "5. Display details\n" 493
463         "6. Log out\n" 494
464         "7. Exit\n"); 495
465     printf("Enter your choice: "); 496
466     choice = getInt(); 497
467 498
468     switch (choice) 499
469     { 500
470     case 1: 501
471     { 502
472         // Book a ticket 503
473         bookTicket(context); 504
474         break; 505
475     } 506
476     case 2: 507
477     { 508
478         // List booked tickets 509
479         listBookedTickets(context); 510
480         break; 511
481     } 512
482     case 3: 513
483     { 514
484         // Cancel a ticket 515
485         cancelTicket(context); 516
486         break; 517
487     } 518
488     case 4: 519
489     { 520
490         // Pay 521
491         pay(context); 522
492         break; 523
493     } 524
494     case 5: 525
495     { 526
496         // Display user details 527
497         displayUserDetails(context); 528
498         break; 529
499     } 530
500     case 6: 531
501     { 532
502         // Log out 533
503         logout(context); 534
504         break; 535
505     } 536
506     case 7: 537
507     { 538
508         // Exit 539
509         return EXIT; 540
510     } 541
511     default: 542
512     { 543
513         printf("Invalid option chosen!\n"); 544
514     } 545
515     } 546
516     return 0; 547
517 } 548
```

# 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.

```
8 #define GENERATE_CRUD(type)
9     typedef struct type##Node type##Node;
10
11     struct type##Node
12     {
13         type data;
14         type##Node *next;
15     };
16
17     typedef struct
18     {
19         type data;
20         CompareFunction compare;
21     } type##Condition;
22
23     void *type##Create(void *value)
24     {
25         type##Node *node = CREATE(type##Node, 1);
26         node->next = NULL;
27         node->data = *((type *)value);
28
29         return node;
30     }
31
32     void *type##GetData(void *node)
33     {
34         type##Node *n = (type##Node *)node;
35         return ((void *)&n->data);
36     }
37
38     void *type##GetNext(void *node)
39     {
40         type##Node *n = (type##Node *)node;
41         return ((void *)n->next);
42     }
43
44     bool type##SetData(void *node, void *value)
```



# 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.

```

81 void *on##type##Event(EventType type, void *container, void *data)
82 {
83     switch (type)
84     {
85     case INSERT:
86     {
87         List *list = (List *)container;
88         type##Node *node = (type##Node *)data;
89         BasicMetadata *metaData = (BasicMetadata *)list->context->meta;
90         | | | | | | | | | | | | | | | | | | | | | | | | | | | |
91         metaData->counter++;
92         node->data.id = metaData->counter;
93         | | | | | | | | | | | | | | | | | | | | | | | | | | | |
94         break;
95     }
96     case UPDATE:
97     {
98         break;
99     }
100    case DELETE:
101    {
102        break;
103    }
104    }
105    return NULL;
106 }

```

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!**