<u>PROJECT : Travel Management System</u>

<u>REPORT</u>

<u>LANGUAGE: Kotlin</u>

<u>Group Members:</u>

Kalyan Venkat Madireddy

Rohith Guptha Kona

Vamshi Reddy Konuganti

Tarun Akasapu

Table of Contents:

Introduction to Kotlin Programming Language

The well-known software development company JetBrains, which focuses on integrated development environments, produced the open-source programming language Kotlin. The language was introduced in 2011 and is intended to be expressive, succinct, and safe. It can be compiled to JavaScript or native code and is based on the Java Virtual Machine (JVM).

Static typing, which offers compile-time checks to prevent errors and increase code stability, is one of Kotlin's primary features. The language also allows type inference, which enables the compiler to figure out a variable's data type based on its initialization value.

The language's compatibility with Java is yet another crucial feature. In order to gradually adopt the language without having to rewrite all of their code at once, developers can utilize Kotlin alongside Java. Due to its compatibility, Kotlin can also be used for server-side, web, and desktop application development, in addition to the development of Android apps.

Because of its short syntax, null safety features, and capability to cut down on boilerplate code, Kotlin has become more and more popular. The language has consequently developed into a potent tool for programmers looking to produce high-quality software in a number of scenarios.

Paradigm of the language

- Programming language Kotlin adopts a multi-paradigm approach to software development, giving developers a wide variety of tools and strategies to address challenging issues. Kotlin provides a flexible and adaptable foundation for developing software in a variety of scenarios by mixing components of reactive programming, procedural programming, functional programming, and object-oriented programming.

- The object-oriented programming capabilities of Kotlin, at their core, offer a potent means to encapsulate data and action within reusable code structures, making it simpler to create and maintain applications over time. As a result of Kotlin's functional programming features, such as lambda expressions and immutability, developers can easily design sophisticated algorithms and data transformations with shorter, more expressive code.
- Kotlin also supports procedural programming, giving programmers a systematic way to organize their code using functions and control flow diagrams. This makes it possible to write readable, understandable, and maintainable code.
- Reactive programming, a paradigm that facilitates the handling of asynchronous and event-driven programming using streams of data, events, and signals, is also supported by Kotlin. This makes Kotlin the ideal choice for developing real-time data processing-required apps, including those used in gaming, banking, and IoT.
- Kotlin's multi-paradigm approach to programming offers programmers a complete set of tools and methods to create high-quality, effective software across a variety of areas.

Historical Evolution

2011- JetBrains, a software development firm, announces the invention of Kotlin, a new programming language.

2012- the introduction of Kotlin as an Apache 2.0-licensed open-source project.

2013- the introduction of Kotlin's initial version (1.0), which supported Android and Java development.

2016- Google officially embraces Kotlin on Android, which improved the language's uptake among those working in the mobile development industry.

2017- The release of Kotlin 1.1, which adds support for coroutines and enhances compatibility with Java.

2018- The introduction of Kotlin 1.2 brings experimental support for cross-platform programming and enhances performance.

2019- The release of Kotlin 1.3 introduces new features including inline classes and contracts and enhances support for cross-platform programming.

2020-Kotlin Multiplatform Mobile (KMM), a framework for creating cross-platform mobile apps using Kotlin, is released by JetBrains.

2021-Kotlin 1.5 is published, enhancing support for JVM and Android programming and introducing new features including sealed interfaces.

2022- Kotlin 1.7.0 has been published, which includes stable opt-in requirement annotations, delegation implementation, and an underscore operator for type args.

2023- Kotlin 1.8 has been published, featuring a new -Xdebug compiler option, improved Objective-C/Swift interoperability, and compatibility with Gradle.

Why use Kotlin?

- Simplicity - Easy to learn and write, using a minimal set of straightforward syntactic rules.
- Concurrency- Concurrent and parallel programming are supported natively, allowing for effective utilization of existing hardware.

- Performance- Converts to machine code for quick and effective execution.
- Scalability - Suitable for developing distributed applications and large-scale systems.
- Strong typing - Lowers the possibility of runtime problems by helping to detect mistakes at build time.
- Robustness - Built-in frameworks for error handling and testing increase the dependability of applications.
- Open Source- Development driven by the community, with a multitude of libraries and tools at hand.

Elements - Kotlin

In Kotlin, each variable is an object that may have member functions and properties called on it. Although certain types may have a unique internal representation—for instance, integers, characters, and booleans may be represented at runtime as primitive values—to the user, they may appear to be regular classes.

The fundamental Kotlin types are:

- Numbers
- Booleans
- Characters
- Strings
- Arrays

Numbers:

- Numerous numeric data types, including Byte, Short, Int, Long, Float, and Double, are supported by Kotlin.
- Both signed (positive or negative) and unsigned (positive only) values can be used to represent numeric data types.
- For performing arithmetic operations on numbers, Kotlin offers a variety of operators and functions.
- Numerous formats, including decimal, binary, octal, and hexadecimal, are available for expressing numerical literals.
- The range of values that may be represented by numerical data types is finite, and any attempt to save a value outside of this range will result in an error.
- Because Kotlin supports type inference, the compiler may frequently identify the appropriate data type depending on the context of its application.

  Byte: val byte Var: Byte = 100

  Short: val shortVar: Short = 32767

  Int: val intVar: Int = 2147483647

  Long: val longVar: Long = 9223372036854775807 L

  Float: val floatVar: Float = 3.14159f

  Double: val doubleVar: Double = 3.141592653589793

  Unsigned Int: val unsignedVar: UInt = 4294967295u

Type	Size	Min	Max
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 (231 - 1)
Long	64	-9,223,372,036,854, 775,808 (-263)	9,223,372,036,854, 775,807 (263 - 1)

Booleans:

- In Kotlin, booleans are logical values that may either be true or false.
- In order to manage program flow, booleans are frequently employed in conditional statements and loops.
- The keyword "Boolean" in Kotlin designates the Boolean type.
- Examples

  val flag1: Boolean = true

  val flag2: Boolean = false
- Built in operations on Boolean: || – disjunction (logical OR); && – conjunction (logical AND) ; ! – negation (logical NOT)

Characters :

- Individual Unicode characters, such as letters, numerals, and symbols, are represented by Kotlin characters.
- The term "Char" is used to identify characters.

- Single quotes can be used to represent characters, such as "A" or "n."
- For working with characters, Kotlin offers a variety of operators and methods, including formatting, transforming, and comparison.
- Examples:

  ```
  val letter: Char = 'A'

  val newLine: Char = '\n'
  ```

Strings:

Strings are a basic data type in Kotlin for expressing and manipulating character sequences, and their vast variety of operations makes them appropriate for a wide range of applications.

- Strings in Kotlin represent a character sequence.
- The term "String" denotes strings.
- Kotlin has a number of string operators and methods, such as concatenating, separating, and formatting.
- Double quotes can be used to represent strings, such as "Hello, world!"
- Examples:

  ```
  val str1: String = "Project"

  val str2: String = "Report"

  val str3 = "$str1, $str2!"
  ```

Arrays:

In Kotlin, arrays are a strong data type for storing and accessing collections of items, and their rich set of operations makes them useful for a wide range of programming applications.In Kotlin, arrays are a strong data type for storing and accessing collections of items, and their rich set of operations makes them useful for a wide range of programming applications.

- In Kotlin, an array is a collection of elements of the same type that may be retrieved using an index.
- Arrays can be of fixed or dynamic size, with fixed-size arrays having a fixed size and dynamic-size arrays having an initial set of items.
- Array elements can be accessed using square brackets, such as arr[index].
- arrayOf() and intArrayOf() are two factory methods that may be used to generate arrays.
- Sorting, filtering, and transforming are just a few of the array-related methods available in Kotlin.
- Examples:

  ```
  val splarr = arrayOf(1, 2, 3, 4, 5) // dynamic size

  val splarr = IntArray(3) // fixed size of 3

  splarr[0] = 10 // change element at index 0 to 10
  ```

Syntax Of the language

brief:

- Kotlin code is arranged into packages, which might include functions, classes, and other objects.
- A Kotlin program's entry point is a function called "main()," which is usually found in a file called "Main.kt" or "Main.kts."
- Statements in Kotlin are normally concluded with a semicolon (;), however this is optional.
- Curly braces {} are used to encapsulate chunks of code in Kotlin, such as function bodies or conditional expressions.
- Function arguments are enclosed in parentheses, while function parameters are separated by commas.
- Variables in Kotlin are defined using the "val" or "var" keywords, with "val" suggesting a read-only variable and "var" indicating a mutable variable.
- To access member methods and attributes of objects, such as str.length or array.sum(), Kotlin uses dot notation (.).
- Kotlin has a number of operators for arithmetic, comparison, and logical operations, including +, -, *, ==, &&, and ||.
- If-else statements, for and while loops, and when expressions are among the control structures supported by Kotlin.

Overall, the syntax of Kotlin is intended to be compact and intuitive, with many features borrowed from popular programming languages like Java and Python.

Variables:

Variables are containers in programming that carry data values. Variables in Kotlin are classified into two types: "var" and "val".

1. A "var" variable can have its value changed at any moment.
2. A "val" variable is read-only, which means it cannot be reassigned once initialized.

Rules for writing Variable Names:

- Letters, digits, and underscores can all be used in variable names.
- Variable names must not begin with a number.
- Spaces are not permitted in variable names.
- By convention, variable names should begin with a lowercase letter.
- CamelCase should be used for variable names, where the first word is lowercase and succeeding words begin with a capital letter.
- Acceptable Variable Names in Kotlin includes,

  age

  firstNamer

  myVariable28

- Example of declaring a variable : **val message: String = "Project, Report!"**

  In this example, we've defined a String variable named message and populated it with the value "Project, Report!". Further break down:

  - val: This is a Kotlin keyword that indicates we're declaring an immutable variable, which means its value cannot be changed once it's been initialized. Instead, we would use the var keyword to declare a mutable variable.

- The name of our variable is message. We may select whatever name we choose as long as it adheres to the naming standards and regulations
- String: This defines the sort of value that will be stored in our variable. In this scenario, we have stated that the message shall include a String. = "Project, Report!".

Arithmetic Operators:

Arithmetic operators are used in Kotlin to execute fundamental mathematical calculations. Among the arithmetic operators are:

- Addition (+): The addition operator is used to combine two values. For example, 3 + 5 equals 8.
- The subtraction operator (-) is used to remove one value from another. For example, 7 - 4 yields 3.
- Multiplication (*): This operator is used to multiply two values together. 2 * 6 would, for example, yield 12.
- The division operator (/) is used to divide one value by another. For example, 10/2 would provide 5.
- The modulus operator (%) is used to find the remainder of a division operation. For example, 10% 3 would provide 1 (since 10 divided by 3 yields a residual of 1).

  It's vital to remember that the data types of the operands must match when executing arithmetic operations. If you try to add an integer and a floating-point number, Kotlin will convert the integer to a floating-point number automatically before completing the addition operation.

Assignment Operators:

In Kotlin, assignment operators are used to assign values to variables. The equal sign (=) is the most commonly used assignment operator for assigning a value to a variable. "var x = 5", for example, sets the value 5 to the variable x. Aside from the equal sign, Kotlin also supports a number of compound assignment operators, which combine an arithmetic operator with an assignment operator. These are some examples:

- (+=): The addition and assignment operator is used to add a value to a variable and then assign the result to the variable. For example, "x += 3" is the same as "x = x + 3."
- Subtraction and assignment (-=): This operator is used to subtract a value from a variable and then assign the result to the variable. For example, "x -= 2" is the same as "x = x - 2."
- Multiplication and assignment (*=): This operator is used to multiply a variable by a value and then assign the result to the variable. "x *= 4", for example, is identical to "x = x * 4."
- Division and assignment (/=): This operator is used to divide a variable by a value and assign the result to the variable. For example, "x /= 2" is the same as "x = x / 2."
- Modulus and assignment (%=): This operator is used to discover the remainder of a division operation and assign the result to a variable. For example, "x%= 3" is the same as "x = x% 3."

Comparison Operators:

In Kotlin, comparison operators are used to compare two values and return a Boolean value (true or false) based on the result of the comparison. In Kotlin, the following comparison operators are available:

- Equal to (==): The equal to operator is used to determine whether two values are equal. For instance, "x == y" returns true if x equals y.
- Not equal to (!=): The not equal to operator is used to determine whether two values are not equal. For example, "x!= y" returns true if x and y are not equal.
- Greater than (>): The greater than operator determines if one number is greater than another. For example, the expression "x > y" yields true if x is bigger than y.
- Less than(<): The less than operator is used to determine whether one value is less than another. "x < y," for example, returns true if x is smaller than y.
- larger than or equal to (>=): This operator is used to determine whether one value is larger than or equal to another. "x >= y", for example, returns true if x is larger than or equal to y.
- Less than or equal to (<=): The less than or equal to operator is used to determine if one value is less than or equal to another. "x < = y," for example, returns true if x is less than or equal to y.

Logical Operators:

In Kotlin, logical operators are used to aggregate numerous Boolean statements and evaluate them as one. Kotlin supports the following logical operators:

- AND (&&): If both operands are true, the AND operator returns true. For instance, "x && y" yields true if both x and y are true.

- OR (||): If one or both of the operands are true, the OR operator returns true. For instance, "x || y" yields true if either x or y is true, or if both are true.
- NOT (!): The NOT operator reverses an operand's logical state. For example, "!x" returns true if x is true and false otherwise.

Bitwise Operators:

In Kotlin, bitwise operators are used to execute bitwise operations on individual bits of integer types (Int, Long, Short, and Byte) and return the result as an integer. Kotlin supports the following bitwise operators:

- AND (&): The AND operator performs a bitwise AND operation between two integers' corresponding bits and returns the result.
- OR (|): The OR operator performs a bitwise OR operation between two integers' corresponding bits and returns the result.
- XOR (^): The XOR operator performs a bitwise XOR (exclusive OR) operation between two integer bits and returns the result.
- NOT (~): The NOT operator inverts all of an integer's bits to perform a bitwise NOT (complement) operation on it.
- Left shift (<<):: The left shift operator moves an integer's bits to the left by a given number of locations and fills the empty positions with zeros.
- The right shift operator (>>) shifts the bits of an integer to the right by a given number of places and fills the empty locations with zeros.
- Unsigned right shift (>>>): The unsigned right shift operator moves an integer's bits to the right by a given number of locations and fills the empty positions with zeros while preserving the sign bit.

Elvis Operator:

The Elvis operator (?:) is a Kotlin shorthand operator for handling nullability in expressions. It's used to set a default value for an expression if it's null. A question mark and a colon (?:) are used to symbolize the Elvis operator. In Kotlin, the Elvis operator operates as follows:

If the expression before the Elvis operator is not null, the expression's value is returned.

If the expression preceding the Elvis operator is empty, the value after the Elvis operator is returned.

- Syntax- expression ?: defaultValue

  In this example, "expression" is the nullable expression to be checked for null, and "defaultValue" is the value to be provided if the expression is null.

  Example: val name: String? = null ; Checking if the name is null or not .

Range Operator:

In Kotlin, the range operator (..) is used to produce a range of values between two provided values. It is a quick way to create a range that contains the starting and ending numbers.

The range operator works with loops, conditional statements, and other constructs that operate with value ranges. It is especially handy for iterating through a set of numbers or performing operations on a set of values.

- Syntax: startValue..endValue

  In this case, "startValue" represents the first value in the range, and "endValue" represents the last value in the range. The operator's range covers both the start and finish values.

  Example:

```
for (i in 1..10) {
        println(i)
        }
```

The range operator is used to generate a range of values ranging from 1 to 10. The for loop loops across the range, printing each value to the console.

Unary Operators:

Unary operators in Kotlin are operators that work with a single operand, which can be a variable or a value. These operators modify the operand, such as altering its sign or inverting its boolean value. In Kotlin, there are various unary operators, including:

- Unary Plus (+): This operator represents the identity of a number or variable, and it returns the operand's value unchanged.
- Unary Minus (-): This operator is used to change the sign of a number or variable, converting it from positive to negative or vice versa.
- Increment (++) and Decrement (--): These operators are used to either increase or decrease the value of a variable by one. They can function as prefix or postfix operators.
- Logical Not (!): This operator is used to invert a variable or expression's boolean value. When a variable is true, the logical not operator returns false, and vice versa.
- Bitwise Not (~): This operator inverts the bits of a number or variable, converting every 0 to 1 and every 1 to 0.

Control Statements:

Control statements aid in the control of a program's flow, making it more flexible and efficient.

Conditional Statements: Conditional statements are used to run a given code block only when a certain condition is met. Conditional statements in Kotlin are as follows:

- if statement
- if-else statement
- when statement

if statement: The if statement in Kotlin is a conditional statement that is used to run a given code block only when a specified condition is true. It is the most basic type of conditional statement in Kotlin. The if statement has the following syntax:

```
if (condition) {

   // code to be executed if condition is true

}
```

The condition is an expression that evaluates to a true or false value. If the condition is met, the code block enclosed by curly brackets is executed. If the condition is true, the code block is skipped and the program proceeds to the following statement.

Here's an example of how to use the if statement to determine if a given integer is positive or negative:

```
val num = -5

 if (num < 0) {

       println("$num is negative")
```

The if statement in this example tests to see if the value of the num variable is less than 0. If it is true, the code inside the curly brackets is run and the message "num is negative" is printed by the program.

if-else statement: In Kotlin, the if-else statement is a control flow statement that allows the computer to execute alternative code blocks based on a condition. The if-else statement has the following syntax:

```
if (condition) {
    // code block to execute if condition is true

} else {

    // code block to execute if condition is false

}
```

Condition is the expression to be evaluated in this syntax. If the condition is true, the code block that follows the if statement is executed. If the condition is true, the code block after the otherwise statement is performed.

Example: Checking if a number is even or odd

```
val num = 7

if (num % 2 == 0) {

    println("$num is even")

} else {

    println("$num is odd")

}

// Output: 7 is odd
```

When statement: The when statement in Kotlin is a strong control flow statement akin to the switch statement in other programming languages. It is used to compare an expression to a collection of potential values and execute a block of code for each match.

When statements can be used in two ways:

When used as an expression, the when statement returns a value based on the matched condition.

When used as a statement, the when statement simply executes a block of code for each met condition.

The following is the syntax of the when statement:

```
when (expression) {

    value1 -> statement1
```

```
  value2 -> statement2
  ...
  else -> statementN
}
```

Here, expression is the value to be matched, and the alternative values to match against are value1, value2, and so on. Statement1, statement2, and so on are the code blocks that will be performed for each match. If none of the other requirements are satisfied, the else block is executed.

Example:

```
  val dayOfWeek = "Sunday”
  val dayType = when(dayOfWeek) {
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" -> "Weekday"
    "Saturday", "Sunday" -> "Weekend"
    else -> "Unknown"
  }
  println("The day of week is a $dayType")
```

In this example, we have a when statement that accepts as an input a dayOfWeek variable. The arrow notation (->) is used to describe the many circumstances in which the dayOfWeek variable can be. We have two options in this case: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" and "Saturday", "Sunday". If the dayOfWeek variable matches one of these circumstances, we set the dayType variable to that value, which is then reported to the console. If dayOfWeek does not fit any of the circumstances, the value "Unknown" is assigned to dayType.

Looping statements: Looping statements are used to execute a given code block repeatedly until a specified condition is fulfilled. The looping statements in Kotlin are:

- for loop
- while loop
- do-while loop

for-loop:

In Kotlin, the for loop is used to iterate through a range, an array, a string, or any other object that has an iterator. A for loop in Kotlin has the following general syntax:

```
for (item in collection) {

    // code to execute for each item in the collection

}
```

A collection can be any object that supports an iterator, such as an array or a range, in this context. item is a variable that stores the value of each item in the collection, and the code within the curly brackets is run once for each item in the collection.

Example:

```
val numbers = arrayOf(1, 2, 3, 4, 5)

for (number in numbers) {

    println(number)

}
```

The for loop iterates across the numbers array in this example, and the variable number takes on the value of each entry in turn. Because the println() function is run for each element, the result of this code is:1 2 3 4 5

while loop : A while loop in Kotlin is used to run a piece of code repeatedly as long as a condition stays true. A while loop in Kotlin has the following fundamental syntax:

```
while (condition) {

   // code to be executed repeatedly

}
```

The condition is first evaluated by the loop, and if it is true, the code block is performed. The condition is then re-evaluated, and the loop is repeated until the condition becomes false.

Here are some things to remember while utilizing a while loop:

- Inside the while loop, the condition must be a boolean expression that can be true or false.
- If the condition is originally false, the code within the loop will never be performed.
- It is critical that the condition finally becomes false, or otherwise the loop would run endlessly, resulting in an infinite loop.
- Within a while loop, break and continue statements can be used to quit or skip iterations based on particular criteria.

Example:

```
var i = 0

var count = 0

while (count < 5) {

   if (i % 2 == 0) {

      println(i)

      count++

   }

   i++

}
```

This code defines a variable i with a value of 0 and another variable count with a value of 0. The program then starts a while loop with the constraint that the count is fewer than 5. It checks if the value of i is even during the loop, and if it is, it outputs the value and increments the count variable. Finally, it increases i, and the loop is repeated until count equals 5.

do-while loop : In Kotlin, the do-while loop is used to execute a piece of code repeatedly until a condition is fulfilled. The do-while loop is similar to the while loop in that it runs the block of code at least once, even if the condition is initially false.

```
do {

  // code to be executed

} while (condition)
```

In this syntax, the do statement's code block is run first, followed by the condition check. If the condition is true, the code block is performed again, and the procedure is repeated until the condition is false.

Example:

```
var i = 1

do {

  println(i)

  i++

} while (i <= 10)
```

The variable i is initialized to 1 in this example, and the do block is done first, which prints the value of i and increases it by 1. The while condition is then examined, and if i is less than or equal to 10, the code block is performed once again. This step is repeated until i exceeds 10, at which time the loop is terminated.

Jump Statements: Jump statements are used to move program control from one location to another. The jump statements in Kotlin are as follows:

- break statement
- continue statement
- return statement

break statement: The break statement in Kotlin is used to leave a loop before its regular termination. When the break statement is reached, control is handed to the statement that comes immediately after the loop.

Here's an example of the break statement in a for loop:

```
for (i in 1..10) {

   if (i == 5) {

      break

   }

   println(i)

}
```

In this example, the loop will print integers from 1 to 4 and then exit when the value of i equals 5. The above code will provide the following result: 1 2 3 4

Continue Statement: The continue statement is used inside a loop in Kotlin to skip the current iteration and continue with the next iteration of the loop. When the continue statement is met within a loop, the current iteration of the loop is instantly terminated and the loop jumps to the next iteration. The continue statement is often used when a specific condition is satisfied and you wish to bypass the loop processing for that iteration.

Example:

```
for (i in 1..10) {

   if (i % 2 == 0) {

      continue // Skip even numbers

   }

   println(i)

}
```

The loop in this example iterates from 1 to 10 and verifies if the current value of i is even. If i is an even number, the continue statement is executed, skipping the processing for that iteration and moving on to the next. If i is odd, the println command is run, and the value of i is sent to the console. As a result, the above code would produce:1 3 5 7 9

Return Statement: The return statement in Kotlin is used to terminate a function and return a value to the caller. It may be used in both return-typed functions and non-return-typed functions. Here are some important things to remember when utilizing the return statement:

- The return statement must be followed by an expression of the same type as the function's return type when used in a function with a return type.
- When used in a function that does not have a return type, the return statement can be used to quit the function early but no value is returned.
- The return statement can be used numerous times within a function to leave at various places.
- The return statement can be used within a loop to escape the loop early.
- The return statement can be used to terminate both the nested function and the enclosing function when used within a nested function.

Example:

```
fun add(a: Int, b: Int): Int {

        return a + b

}
```

In this example, the return statement is used to return the sum of a and b to the caller.

Functions:

Functions are an important building component in Kotlin programming because they allow developers to encapsulate functionality and reuse code across several apps.

The keyword "fun" is used to define functions in Kotlin, followed by the function name, input arguments, and output type. Functions can be defined with or without arguments, with or without a return value, and with or without a return value. Here are some major properties of Kotlin functions:

- Functions can have zero or more input arguments, each of which can have an optional type annotation. Parameters may have default values, allowing callers to omit them if necessary.
- Return type: Functions can have any Kotlin type as a return type. If a function does not return a value, the return type should be "Unit".
- Named parameters: Because Kotlin supports named parameters, you may call a function by directly stating the argument names and values.
- Default and variable number of arguments: The "vararg" keyword in Kotlin allows functions to have default values for parameters as well as a variable number of arguments.
- Lambda expressions: Lambda expressions are a succinct way of constructing anonymous functions in Kotlin.
- Function scope: Variables and functions specified in the same file, as well as surrounding classes and packages, can be accessed by functions in Kotlin.
- Higher-order functions: Higher-order functions are functions that accept other functions as parameters or return other functions as output.

Standard Library Functions:

Standard Library Functions are predefined functions in the Kotlin language that may be used for a variety of actions. These functions are part of the Kotlin Standard Library, which is included by default in all Kotlin projects. Some of the most frequent Standard Library Functions are:

- println() - Used to print a message to the console.

- readLine() - Used to read input from the console.
- arrayOf() - Used to create an array of objects.
- listOf() - Used to create a list of objects.
- mapOf() - Used to create a map of key-value pairs.
- filter() - Used to filter elements from a collection based on a condition.
- joinToString() - Used to join elements of a collection into a string with a separator.

These routines are well optimized and may be used in the code with no further setup or configuration. They make the development process simpler and faster by offering widely used functionality that may be easily incorporated into your code.

Lambda Functions: Lambda functions are a type of function in Kotlin that can be defined without a name. They are also known as anonymous functions or function literals. They are useful for defining short, self-contained chunks of code that may be provided as arguments to other functions or saved in variables. The following syntax is used to define lambda functions: { arguments -> body }

Arguments are the function's input parameters, and body is the code that the function runs. The arrow (->) denotes the separation of the arguments from the body.Lambda functions are frequently used in conjunction with higher-order functions, which are functions that accept other functions as arguments and return other functions as outcomes. Many higher-order functions in Kotlin's standard library, such as map(), filter(), and reduce(), can take lambda functions as inputs.

- Lambda functions are short and may be used to reduce the necessity for explicit function definitions in code. They are frequently used for basic one-time activities that do not need a full function description.

Example: val square = { n: Int -> n * n }

This lambda function takes an integer input parameter n and returns its square. It can be called like a regular function. Like this println(square(3)) // Output: 9

Inline Functions: In Kotlin, inline functions are a type of function that is used to improve efficiency by eliminating the cost of function calls. When an inline function is called, its body is copied straight into the calling code, eliminating the cost of generating a new stack frame and supplying parameters. This is especially beneficial for functions that are called often or are contained within tight loops.

Example:

```
inline fun double(n: Int): Int {
    return n * 2
}
```

It's worth noting that inline functions may only be used with functions that accept function parameters, lambdas, or local object expressions as arguments. It's also necessary to exercise caution when employing inline functions with big blocks of code, since they can lead to code bloat and poor performance if not utilized correctly.

Operator Overloading: The ability to define and utilize operators such as +, -, *, /,%, and so on with user-defined types in a custom method is referred to as operator overloading. You may use special member functions or extension functions in Kotlin to overload a number of operators for your classes.The key advantages of operator overloading are that it may make your code more expressive, simpler to comprehend, and less reliant on boilerplate code. It's also a feature that may make coding feel more natural and intuitive.To overload an operator in Kotlin, you must first declare a function with the same name as the operator you wish to overload. To overload the + operator for a custom class, for example, you must specify a function called plus.

Classes :

Classes are templates in Kotlin for building objects that encapsulate data and action. The class keyword is used to declare them, followed by the class name and an optional constructor.Here are some crucial elements to remember regarding Kotlin classes:

- Classes can have properties (variables that hold data) and methods (functions that execute actions on data).
- Access modifiers for properties and methods, such as public, private, protected, or internal, restrict their visibility and are accessible from other classes or modules.
- Secondary constructors in classes allow extra options to initialize the object's properties.
- Inheritance is supported in Kotlin, allowing you to construct new classes based on existing ones and reuse their attributes and methods. To inherit from a class, use the: operator followed by the name of the parent class.
- Classes can implement interfaces, which specify a set of methods that must be implemented by the class. To implement an interface, use the: operator followed by the name of the interface.
- Data classes in Kotlin are classes that are meant to hold data and feature built-in functionality such as equals, hashCode, toString, and copy methods.

Example:

```
class Person(name: String, age: Int) {

    var name = name

    var age = age

    fun sayHello() {

        println("Hello, my name is $name and I'm $age years old.")

    }

}
```

We build a Person class with two fields (name and age) and a sayHello() function that outputs a greeting message in this example. To initialize the properties, we also utilize constructor arguments.

Inheritance:

In object-oriented programming (OOP), inheritance is a technique that allows one class (child class) to inherit properties and behavior from another class (parent class). The child class can utilize the parent class's code and enhance it with its own unique features and behavior. In Kotlin, inheritance is accomplished by using the ": " sign followed by the parent class's name. The "super" keyword allows the child class to access the parent class's properties and functions.Some key aspects to note regarding inheritance in Kotlin are as follows:

- By default, Kotlin classes are final, which means they cannot be inherited until the "open" keyword is used to open them.
- In addition to the inherited properties and functions, the child class can have its own.
- A function in the child class is considered to override the parent function if it has the same name and parameters as a function in the parent class. The term "override" is used to signify this.
- Using the "super" keyword in its own constructor, the child class can invoke the parent class constructor.

Example:

```
open class Animal(val name: String) {

  fun eat() {
    println("$name is eating")
  }
}

class Dog(name: String, val breed: String) : Animal(name) {
  fun bark() {
    println("$name is barking")
  }
}

fun main() {
  val myDog = Dog("Buddy", "Golden Retriever")
  myDog.eat() // inherited function from Animal class
  myDog.bark() // function specific to Dog class
}
```

The Animal class is the parent class in this example, and the Dog class is the child class that inherits from it. In addition to the inherited eat method from the parent class, the Dog class contains the extra properties breed and bark.

Properties:

Properties are a basic component of classes in Kotlin. They are variables that are part of a class and are encapsulated with its functionality. When a property is accessed or updated, it can contain custom getters and setters that allow for computation or validation. There are two kinds of Kotlin properties:

- Member Properties: These are properties that are unique to a particular instance of a class. Each class object contains a copy of the member properties.
- Class Properties: These are properties that all instances of a class share. They are declared in the class declaration using the companion object keyword and accessible with the class name.

Member properties: Member properties are specified within a class and connected with a class instance. They may be accessed and edited using the dot notation, which combines the instance and property names. Here are some essential characteristics of Kotlin member properties:

- They can have default values provided to them in the class's constructor.
- They can have their own getter and setter methods that are called whenever the property is accessed or changed.
- They can be designated as var (mutable) or val (immutable).
- Depending on the visibility needs, they might be secret, protected, internal, or public.

  Example: class Person {

  ```
      var name: String = ""

      var age: Int = 0

  }
  ```

The Person class has two member properties in this example: name, which is immutable and has a default value supplied by the constructor, and age, which is changeable and may be updated using the dot notation.

Class Properties: Class properties in Kotlin are variables that belong to the class rather than its instances. In other programming languages, they are commonly referred to as "static" attributes. The "companion object" keyword is used to specify class properties, which produces an object that is connected with the class and may be accessed directly without the requirement for an instance of the class. Class properties are shared by all instances of the class and may be accessed by using the class name followed by the dot operator and the property name. They may be used to store variables or methods that are shared by all instances of the class.

Example:

```
class MyClass {

    companion object {

        val myProperty = "Hello, World!"

    }

}
```

In this case, myProperty is a MyClass class property that can be accessed using MyClass.myProperty.

Interfaces:

An interface in Kotlin is a blueprint or contract for a class to implement certain functionality. It is a set of abstract methods (methods that do not have any implementation) and/or properties (variables that do not have any value) that a class must implement. Here are some crucial elements to remember regarding Kotlin interfaces:

- The interface keyword is used to declare an interface, which is followed by its name and an optional list of parent interfaces in angle brackets.
- Abstract methods (methods without a body) and default methods (methods with a body) can both have implementations in an interface.
- A class can implement one or more interfaces by using the: InterfaceName syntax, and it must provide implementations for all abstract methods in those interfaces.
- An interface can have attributes that are either abstract (no value) or contain getters and/or setters.
- In Kotlin, interfaces may be used to provide abstraction and polymorphism, allowing for more code flexibility and reusability.

Example: interface Drawable {

```
        fun draw()

}
```

This interface defines a single abstract method draw(), which accepts no parameters and returns null. Any class that implements this interface must offer a draw() function implementation.

Functional Interfaces: In Kotlin, functional interfaces are interfaces with only one abstract method and are used to facilitate the usage of lambdas or function references. Function0, Function1, Predicate, and Consumer are some of the built-in functional interfaces in Kotlin. Function0 represents functions that accept no parameters, whereas Function1 represents functions that take one argument. Function2 is used for functions that accept two parameters, and so on. The Predicate interface represents a function that accepts an input and returns a Boolean value, whereas the Consumer interface represents a function that takes an input but outputs nothing. Functional interfaces can be used to simplify the syntax of code that employs lambdas or function references, allowing for more succinct writing. Instead of establishing an anonymous class to implement an interface, a lambda expression may be used to define the implementation of the abstract method of the interface.

Furthermore, functional interfaces can be used to form more sophisticated functions or predicates.

Packages:

A package is a technique of arranging similar classes, functions, and other code components in Kotlin. A package allows you to organize code that performs the same function or is part of the same module. Here are some important facts to remember regarding packages in Kotlin:

- The package keyword is followed by the package name to define a package.
- Packages can be nested, allowing for hierarchical code structure.
- All Kotlin code is stored in the default package, which has no name by default.
- You can use a particular import statement for each entity or a wildcard import statement to import all entities from the package when importing classes or methods from another package.
- Kotlin has some built-in packages, such as kotlin, kotlin.annotation.

Example:

```
// file name: Example.kt

package com.example



class MyClass {

  // class code goes here

}
```

The MyClass class is defined in the com.example package in this example. Then, using the import statement, you can import this class from another package:

- import com.example.MyClass

Modules: Modules in Kotlin are a technique of arranging code and dependencies into logical chunks that may be independently generated, tested, and deployed. A module may include one or more packages, which may be distributed over numerous modules. A module in Kotlin is defined by a collection of source files that may be built together, as well as any external dependencies. Modules can rely on one another, allowing for a modular design that is simple to maintain and develop.When you start a new Kotlin project, it is immediately formed as a module, and you may add more modules as needed. Modules can be specified in either a build file (for example, a Gradle build file) or in a separate module definition file. Modules in Kotlin can assist to enhance code structure and decrease connection between application components. They also make dependency management easier because each module may have its own set of dependencies that are separate from other modules. Overall, Kotlin modules are a key element that contributes to a modular, scalable design.

Object Declarations: The object keyword is used in Kotlin to generate an object declaration. In Kotlin, an object declaration is used to construct a singleton object. This means that just one instance of the object may exist at any given moment, and it can be used everywhere in the program.Here are some important things to remember while working with object declarations in Kotlin:

- The object keyword is followed by the name of the object to generate an object declaration.
- An object declaration, like a class, can contain properties, methods, and constructors.
- A declaration of an object might extend a class or implement an interface.
- A lazy object declaration is one that is not instantiated until it is accessed for the first time.
- In Kotlin, an object declaration may be used to construct a static utility class since it allows you to aggregate similar methods and properties.

Example:
```
object MySingleton {

   var name: String = "John"

   fun sayHello() {

      println("Hello, $name!")

   }

}
```

MySingleton in this example is an object definition that includes a property name and the method sayHello(). The name attribute is used by the sayHello() method to greet the user. Because MySingleton is a singleton object, it can only have one instance. This implies that simply referencing the MySingleton object, the name property and sayHello() function may be accessible from anywhere in the program.

Kotlin- Writability:, Readability and Reliability:

Writability: Because of its short syntax and expressive features, Kotlin is a highly writable language. Developers may build code that is both efficient and manageable by using current programming ideas such as lambda expressions and extension functions. The language also has a variety of operators, allowing complicated operations to be expressed in a single line of code. The flexibility of Kotlin language to handle both object-oriented and functional programming paradigms improves its writability even further.

Readability: Kotlin places a high value on readability, and its syntax is meant to be compact and expressive, making code more readable and understandable. The syntax of the language is clear and uniform, making it easier for developers to read and comprehend code produced by others. Null-safety and type inference are further characteristics of Kotlin that serve to reduce code complexity and make it more understandable.

Reliability: Because of its robust type system and null-safety features, Kotlin is a very trustworthy language. The type system of the language guarantees that variables are only allocated values of their specified type, lowering the possibility of mistakes due to type mismatches. Null-safety guarantees that null pointer exceptions are caught at build time, decreasing the possibility of runtime mistakes. Furthermore, Kotlin has capabilities like extension functions and data classes that serve to reduce the possibility of problems and mistakes. Overall, Kotlin is a very stable language due to its strong type and null-safety features.

Pro’s and Con’s:

Pro’s:

- Syntax that is both concise and expressive
- Type inference cuts down on boilerplate code.
- Runtime errors are reduced by using null safety mechanisms.
- Java code interoperability
- Outstanding tooling and IDE support
- Support for functional programming through lambdas and higher-order functions
- Asynchronous and non-blocking programming with coroutines
- Extension functions are used to extend the functionality of existing classes.
- Data classes that make it simple to create POJOs
- Operator overloading for bespoke operator behavior Immutable collections for safer and more efficient data processing
- Multiplatform development assistance
- Higher-order functions make programming more versatile and reusable.
- Smart casting for more efficient and succinct type checking

- Simple interaction with Android Studio and programming.

Con's:

- Compilation time is longer as compared to Java.
- When compared to Java, there is a smaller community and less resources.
- For individuals unfamiliar with functional programming ideas, Kotlin's learning curve might be high.
- Because the language evolved swiftly, there is considerable discrepancy in syntax and features.
- The emphasis on immutability and functional programming in Kotlin may not be suitable for many use cases, such as high-performance applications.
- Kotlin is not entirely compatible with Java, necessitating some extra work for projects that use both languages.
- Lack of compatibility for some earlier versions of Android, which may restrict mobile app development acceptance.
- In comparison to Java, there is less support for several frameworks, libraries, and tools.
- The standard library is not as large as that of Java.
- Some more complex systems have little documentation and resources.

How the language made the programs easy/hard to implement:

As a developer who has used Kotlin to create a project, I can certainly tell that the language has greatly simplified the implementation process. One of Kotlin's main advantages is its simple and clear syntax. The grammar of the language is remarkably similar to that of Java, making the switch to Kotlin simple for Java developers. Furthermore, the type inference feature of the language saves a lot of time because the developer does not have to explicitly describe the data type of variables. Another key advantage is Kotlin's compatibility with Java. Using Java libraries in Kotlin projects and vice versa reduces the need for developers to rebuild whole codebases, saving time and effort. Furthermore, Kotlin's smooth interaction with well-known development tools such as IntelliJ IDEA, Android Studio, and Gradle. The language's null safety feature also makes program implementation simple. Null safety assures that the application does not fail owing to the usual null pointer exceptions in Java. The null safety feature in Kotlin allows developers to recognize and handle null values at compile time, resulting in more robust and dependable code. On the negative, Kotlin's tiny developer community implies that resources like libraries and support may be restricted in comparison to more established languages. This might make it difficult for developers to solve complicated challenges.Furthermore, the learning curve for Kotlin might be severe for developers who are unfamiliar with object-oriented programming languages. As a result, the learning curve will be longer and the implementation process will be slower. However, once the developer is comfortable with the syntax and functionalities of Kotlin, the implementation process becomes much easier. Finally, Kotlin's short and clear syntax, Java compatibility, null safety feature, and seamless connection with development tools have simplified the implementation process. However, due to the language's small development community and high learning curve, developers may face certain difficulties. Nonetheless, Kotlin is a robust and dependable programming language for creating contemporary software applications.

Problem Definition:

The system application which we are building must be able to run on the local server so that it could achieve high performance and should be able to run 24/7 to ensure the quality of the delivery. The system should be able to allow the new customers to create their own accounts if they don't have them already and existing customers should be able to login to their existing accounts and depending on the the budget they are constrained with, they should be able to select their preferred mode of travel i.e choose between train, bus and flight the 3 most popular means of transportation. Once selected the application should ask the customer to select the to and from and the date of their travel. The system then shows the possible options from which the customer can select and proceed to the details page, where he should be able to enter all the personal details and the card details. Once verified and processed the customer will be redirected to the thank you page from which he gets the link to download the printable pdf version of the itinerary.

Proposed Method:

The System is built using Kotlin as the main language and the html is used to build the UI for other applications. It is connected to mysql in the background for storing the information about the customers and people who manage the system. This system is built following the incremental methodology principles of the Software development life cycle. Furthermore certain machine learning algorithms can be included in future development of this project to make sure we analyze and cash on the important observations like the favorite destination, current interests, people thoughts and stuff.

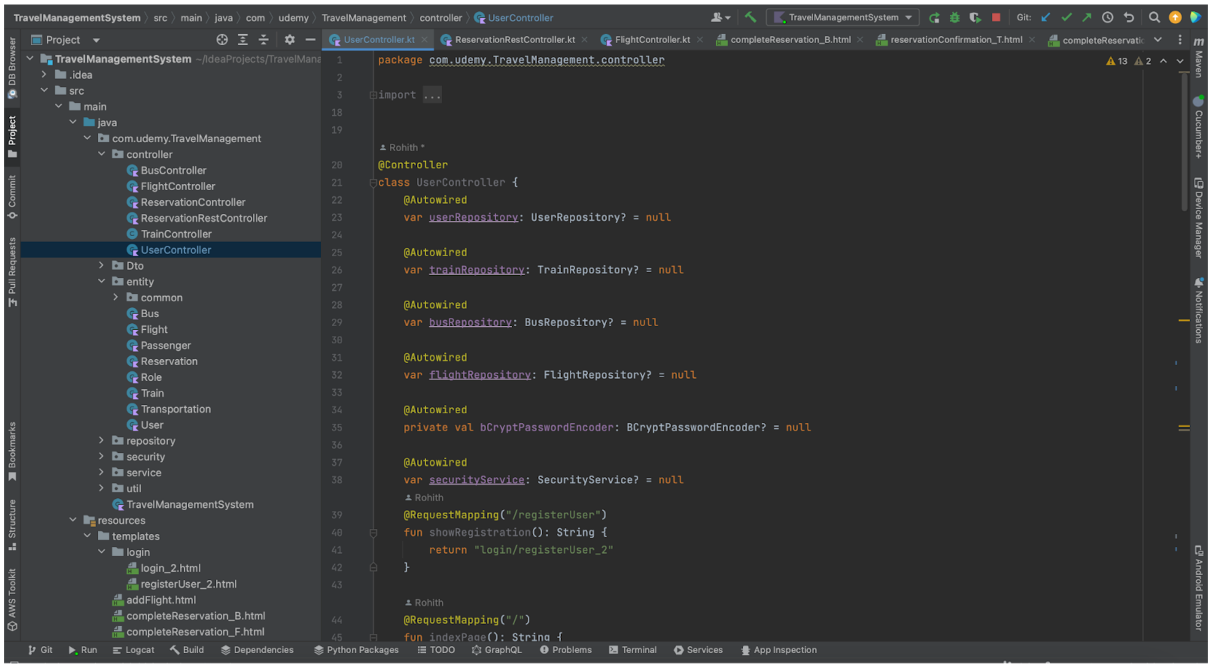
Observations:

- To begin, we discovered that customers tend to search for flights, hotels, and other travel-related services using a mix of dates, price range, and favorite airlines by watching user behavior on the website. We were able to discover user preferences, such as popular destinations and the kind of travel packages they like, based on their search history.
- Second, we were able to detect seasonal trends in travel booking by examining booking patterns across time, such as when users are more likely to plan travel and which locations are most popular at different periods of the year. This data might be utilized to adjust pricing and promotions in order to better meet the demands of users.
- Third, we were able to discover the demographics of the website's users by evaluating user data, such as age, gender, location, and income level. This data might be used to alter the content and promotions of the website to better meet the demands of different user categories.
- Fourth, we discovered that the primary rivals of the website were online trip booking platforms and offline travel firms. We were able to discover areas where we might improve pricing and promotions to better compete with other market participants by examining the rates and availability of flights, hotels, and other travel-related services on the website.
- Finally, based on user search trends and booking history, we identified some popular destinations. We may adapt the website's content and marketing to better satisfy the interests of people who are interested in these popular places by studying this data.
- Overall, these insights might be utilized to improve the website's content, price, and promotions in order to better satisfy its customers' requirements and remain competitive in the travel industry.

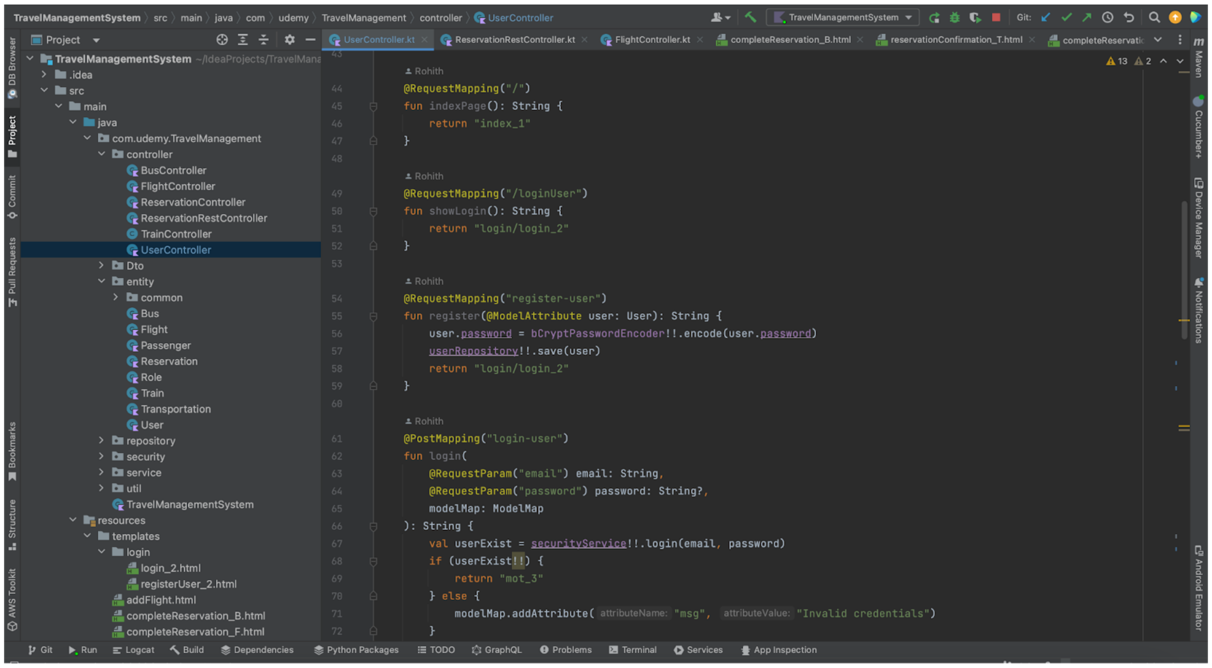
Challenges Faced while Programming:

Setting up Kotlin is easy with a few challenges. You may need to change the SDK and install dependencies from Maven, and using online tutorials and plugins can help. If .jar files are missing from the classpath, download and add them manually. Running on port 8080 may result in errors, but adjusting the application.properties file can usually fix them. Kotlin's null safety and smart casting can make code verbose and hard to read. While it has many open-source libraries, specific ones may not exist, leading to creating custom libraries or examining third-party code. But, Kotlin's static typing and integration with benchmarking and testing tools can be helpful.

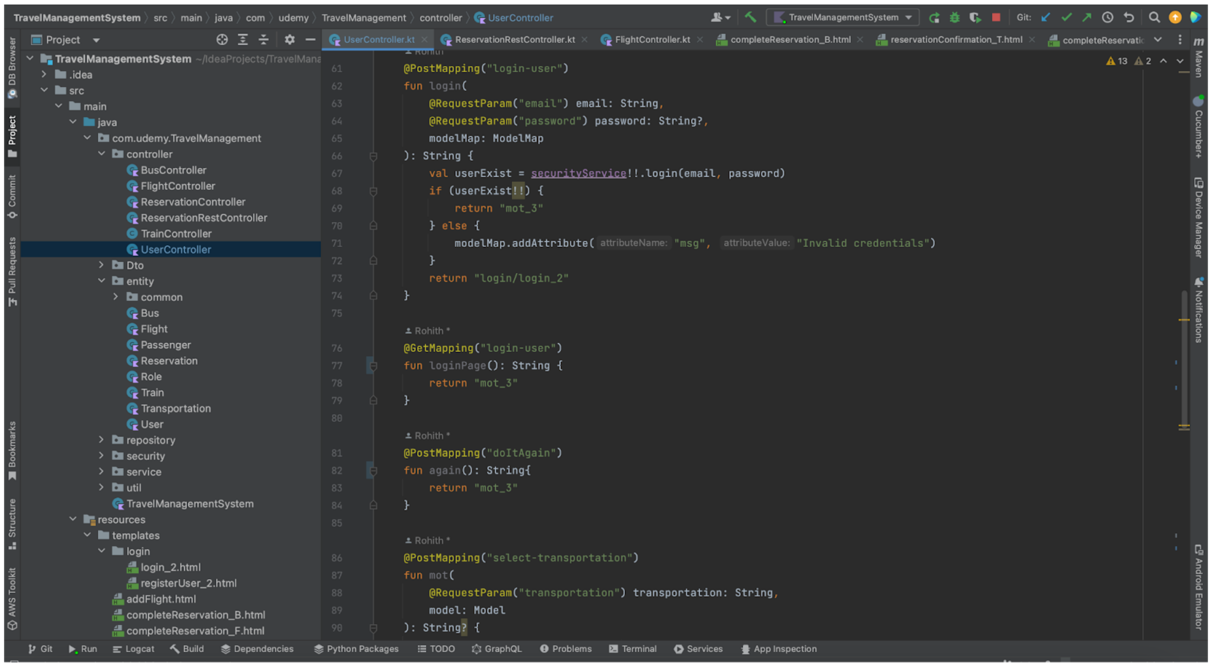
Overview of Code:



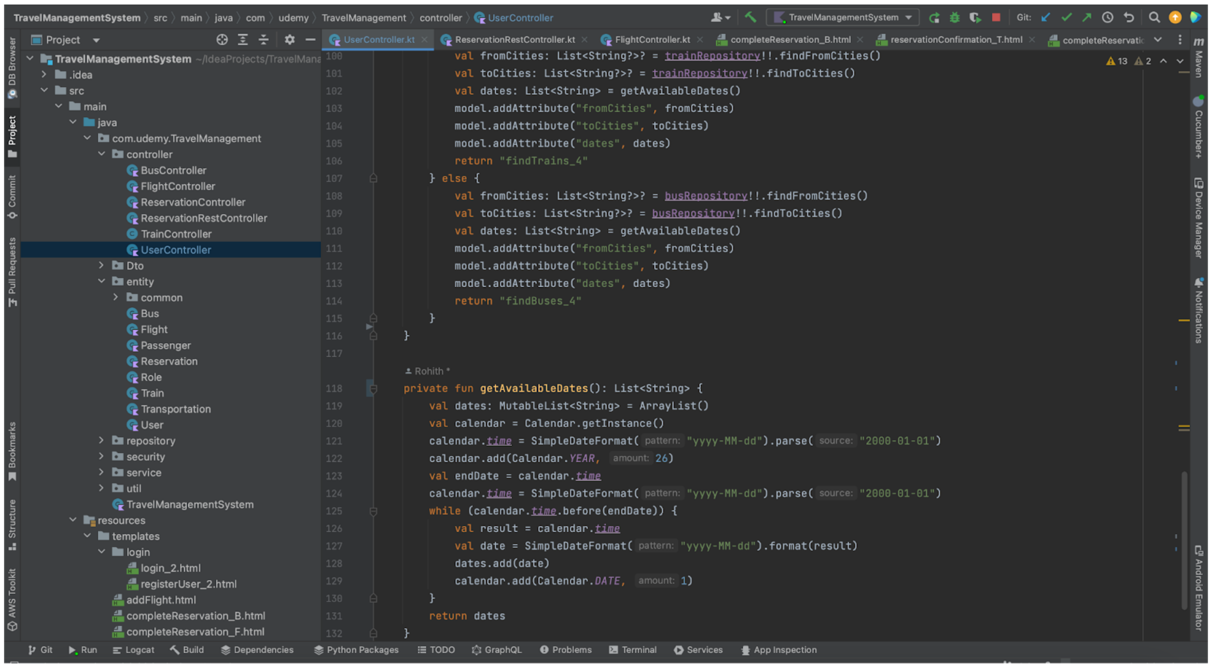
```kotlin
package com.udemy.TravelManagement.controller

import ...


Rohith *
@Controller
class UserController {
    @Autowired
    var userRepository: UserRepository? = null

    @Autowired
    var trainRepository: TrainRepository? = null

    @Autowired
    var busRepository: BusRepository? = null

    @Autowired
    var flightRepository: FlightRepository? = null

    @Autowired
    private val bCryptPasswordEncoder: BCryptPasswordEncoder? = null

    @Autowired
    var securityService: SecurityService? = null
    Rohith
    @RequestMapping("/registerUser")
    fun showRegistration(): String {
        return "login/registerUser_2"
    }

    Rohith
    @RequestMapping("/")
    fun indexPage(): String {
```



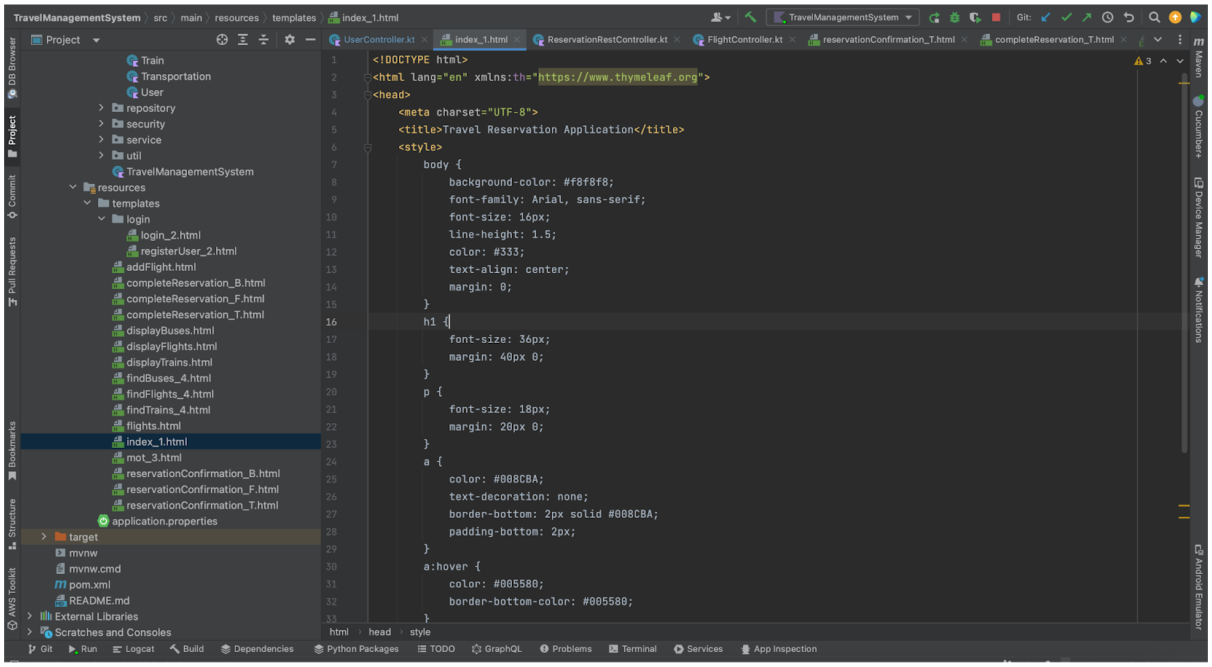
```kotlin
    Rohith
    @RequestMapping("/")
    fun indexPage(): String {
        return "index_1"
    }

    Rohith
    @RequestMapping("/loginUser")
    fun showLogin(): String {
        return "login/login_2"
    }

    Rohith
    @RequestMapping("register-user")
    fun register(@ModelAttribute user: User): String {
        user.password = bCryptPasswordEncoder!!.encode(user.password)
        userRepository!!.save(user)
        return "login/login_2"
    }

    Rohith
    @PostMapping("login-user")
    fun login(
        @RequestParam("email") email: String,
        @RequestParam("password") password: String?,
        modelMap: ModelMap
    ): String {
        val userExist = securityService!!.login(email, password)
        if (userExist!!) {
            return "mot_3"
        } else {
            modelMap.addAttribute( attributeName: "msg",  attributeValue: "Invalid credentials")
        }
```


TravelManagementSystem 〉 src 〉 main 〉 java 〉 com 〉 udemy 〉 TravelManagement 〉 controller 〉 UserController
UserController.kt
ReservationRestController.kt
FlightController.kt
completeReservation_B.html
reservationConfirmation_T.html
Project

```kotlin
    @PostMapping("login-user")
    fun login(
        @RequestParam("email") email: String,
        @RequestParam("password") password: String?,
        modelMap: ModelMap
    ): String {
        val userExist = securityService!!.login(email, password)
        if (userExist!!) {
            return "mot_3"
        } else {
            modelMap.addAttribute( attributeName: "msg",  attributeValue: "Invalid credentials")
        }
        return "login/login_2"
    }

    Rohith *
    @GetMapping("login-user")
    fun loginPage(): String {
        return "mot_3"
    }

    Rohith *
    @PostMapping("doItAgain")
    fun again(): String{
        return "mot_3"
    }

    Rohith *
    @PostMapping("select-transportation")
    fun mot(
        @RequestParam("transportation") transportation: String,
        model: Model
    ): String? {
```


TravelManagementSystem 〉 src 〉 main 〉 java 〉 com 〉 udemy 〉 TravelManagement 〉 controller 〉 UserController
UserController.kt
ReservationRestController.kt
FlightController.kt
completeReservation_B.html
reservationConfirmation_T.html
Project

```kotlin
            val fromCities: List<String?>? = trainRepository!!.findFromCities()
            val toCities: List<String?>? = trainRepository!!.findToCities()
            val dates: List<String> = getAvailableDates()
            model.addAttribute("fromCities", fromCities)
            model.addAttribute("toCities", toCities)
            model.addAttribute("dates", dates)
            return "findTrains_4"
        } else {
            val fromCities: List<String?>? = busRepository!!.findFromCities()
            val toCities: List<String?>? = busRepository!!.findToCities()
            val dates: List<String> = getAvailableDates()
            model.addAttribute("fromCities", fromCities)
            model.addAttribute("toCities", toCities)
            model.addAttribute("dates", dates)
            return "findBuses_4"
        }
    }

    Rohith *
    private fun getAvailableDates(): List<String> {
        val dates: MutableList<String> = ArrayList()
        val calendar = Calendar.getInstance()
        calendar.time = SimpleDateFormat( pattern: "yyyy-MM-dd").parse( source: "2000-01-01")
        calendar.add(Calendar.YEAR,  amount: 26)
        val endDate = calendar.time
        calendar.time = SimpleDateFormat( pattern: "yyyy-MM-dd").parse( source: "2000-01-01")
        while (calendar.time.before(endDate)) {
            val result = calendar.time
            val date = SimpleDateFormat( pattern: "yyyy-MM-dd").format(result)
            dates.add(date)
            calendar.add(Calendar.DATE,  amount: 1)
        }
        return dates
    }
```

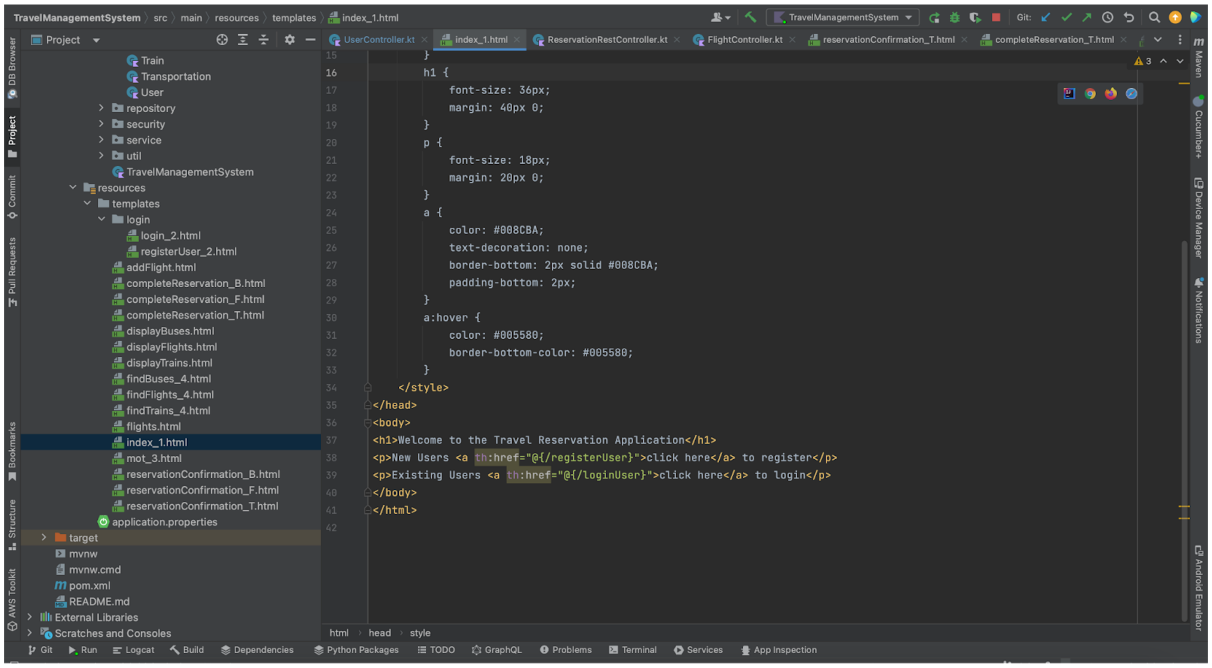


TravelManagementSystem 〉 src 〉 main 〉 resources 〉 templates 〉 index_1.html

UserController.kt | index_1.html | ReservationRestController.kt | FlightController.kt | reservationConfirmation_T.html | completeReservation_T.html

```
Train
Transportation
User
> repository
> security
> service
> util
TravelManagementSystem
v resources
  v templates
    v login
        login_2.html
        registerUser_2.html
      addFlight.html
      completeReservation_B.html
      completeReservation_F.html
      completeReservation_T.html
      displayBuses.html
      displayFlights.html
      displayTrains.html
      findBuses_4.html
      findFlights_4.html
      findTrains_4.html
      flights.html
      index_1.html
      mot_3.html
      reservationConfirmation_B.html
      reservationConfirmation_F.html
      reservationConfirmation_T.html
    application.properties
> target
  mvnw
  mvnw.cmd
  pom.xml
  README.md
> External Libraries
> Scratches and Consoles
```

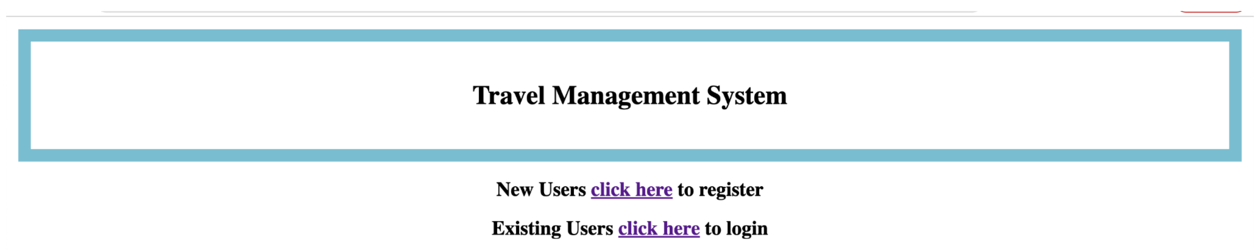
```html
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Travel Reservation Application</title>
    <style>
        body {
            background-color: #f8f8f8;
            font-family: Arial, sans-serif;
            font-size: 16px;
            line-height: 1.5;
            color: #333;
            text-align: center;
            margin: 0;
        }
        h1 {
            font-size: 36px;
            margin: 40px 0;
        }
        p {
            font-size: 18px;
            margin: 20px 0;
        }
        a {
            color: #008CBA;
            text-decoration: none;
            border-bottom: 2px solid #008CBA;
            padding-bottom: 2px;
        }
        a:hover {
            color: #005580;
            border-bottom-color: #005580;
        }
```



```html
        }
        h1 {
            font-size: 36px;
            margin: 40px 0;
        }
        p {
            font-size: 18px;
            margin: 20px 0;
        }
        a {
            color: #008CBA;
            text-decoration: none;
            border-bottom: 2px solid #008CBA;
            padding-bottom: 2px;
        }
        a:hover {
            color: #005580;
            border-bottom-color: #005580;
        }
    </style>
</head>
<body>
<h1>Welcome to the Travel Reservation Application</h1>
<p>New Users <a th:href="@{/registerUser}">click here</a> to register</p>
<p>Existing Users <a th:href="@{/loginUser}">click here</a> to login</p>
</body>
</html>
```

Walkthrough:

Index Page:

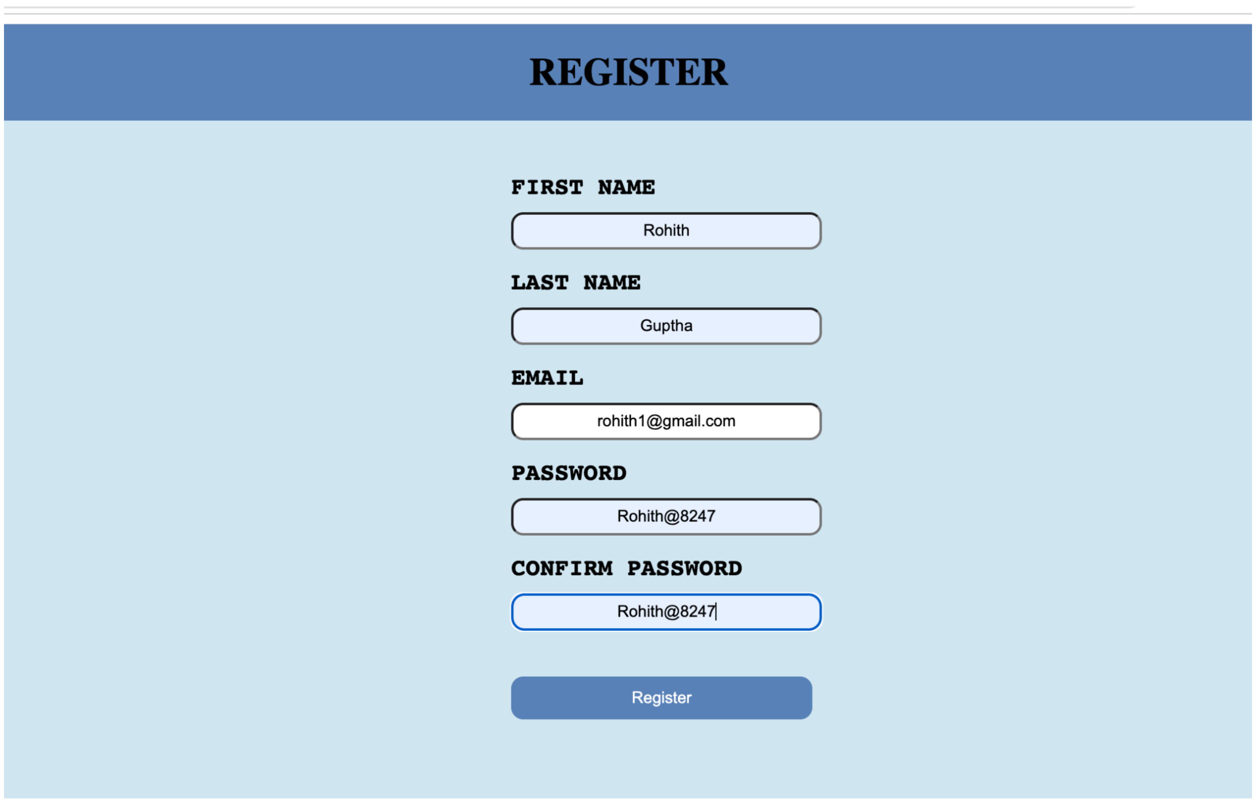


Travel Management System

New Users click here to register

Existing Users click here to login

Register Page:



REGISTER

FIRST NAME
Rohith

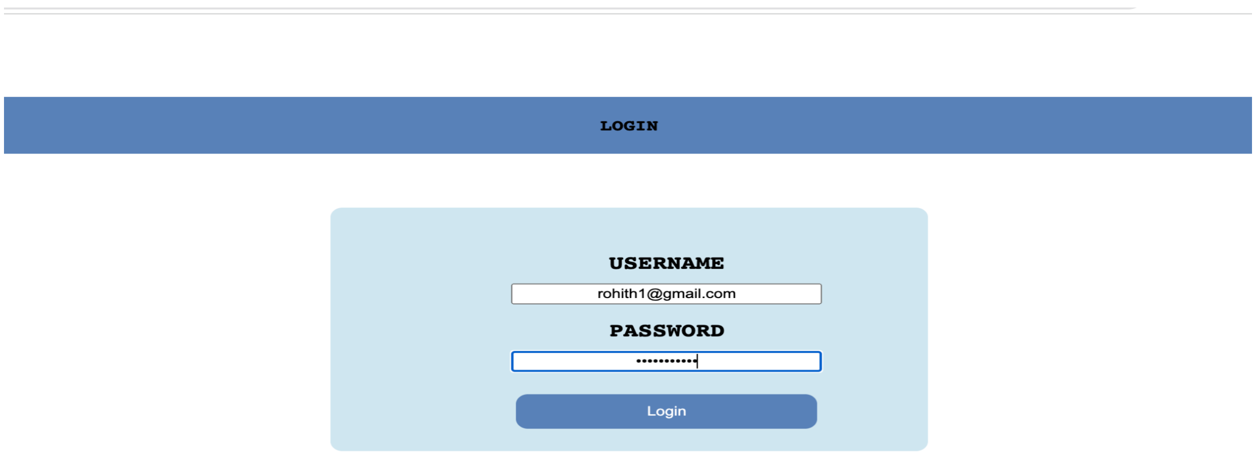
LAST NAME
Guptha

EMAIL
rohith1@gmail.com

PASSWORD
Rohith@8247

CONFIRM PASSWORD
Rohith@8247

Login Page:

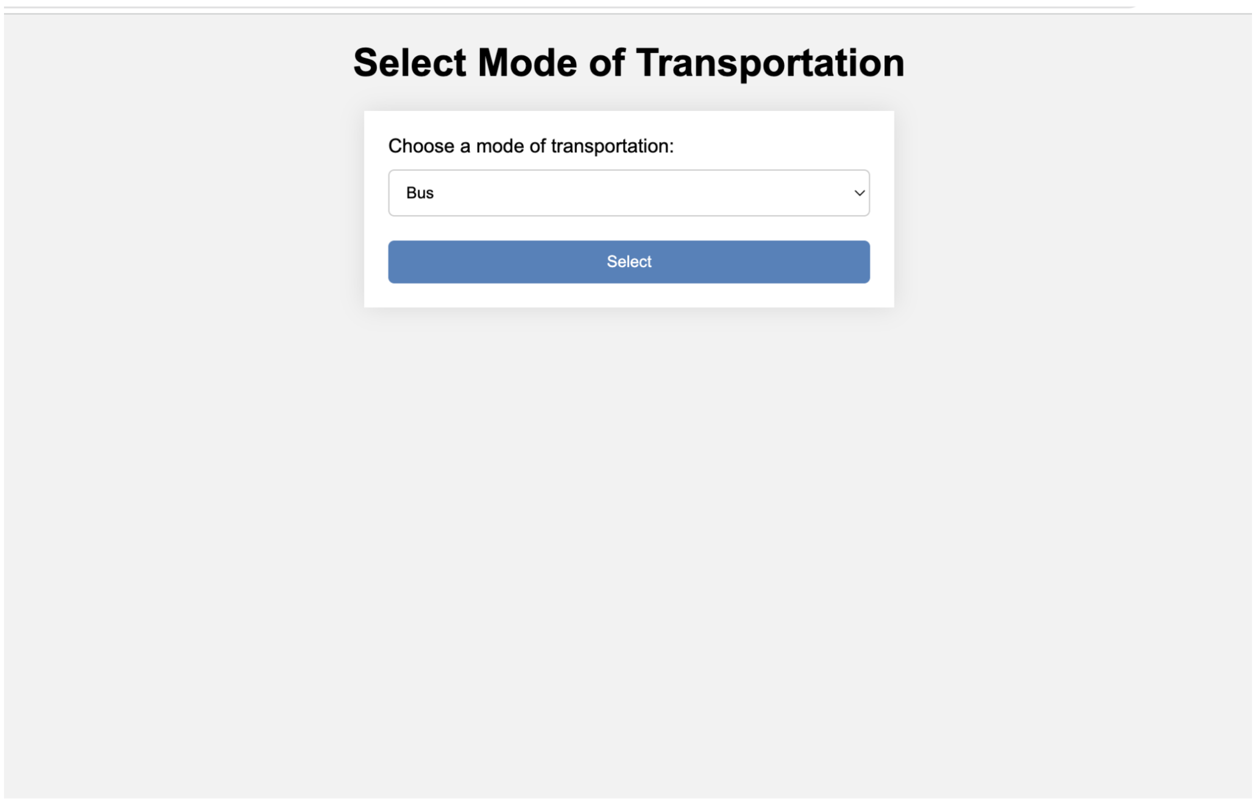


LOGIN

USERNAME
rohith1@gmail.com

PASSWORD
•••••••••••

Selecting Mode of Transportation Page:

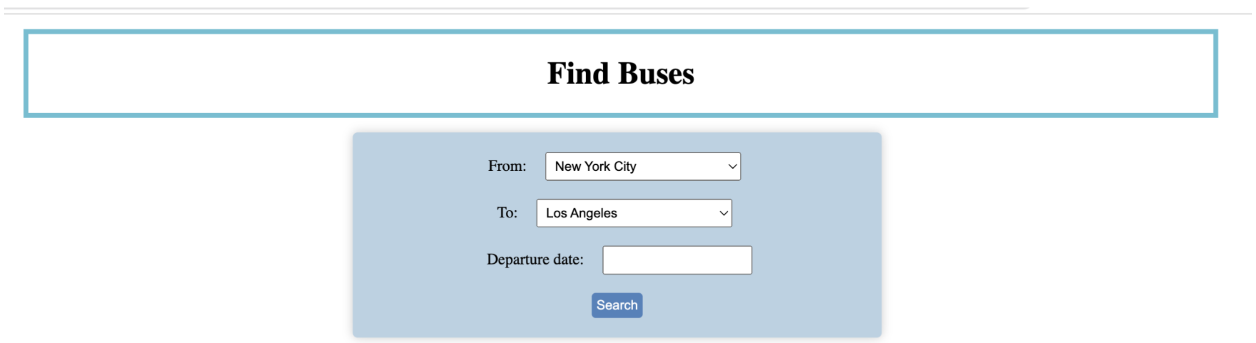


Select Mode of Transportation

Choose a mode of transportation:

Bus

Finding Buses/Trains/Flights:



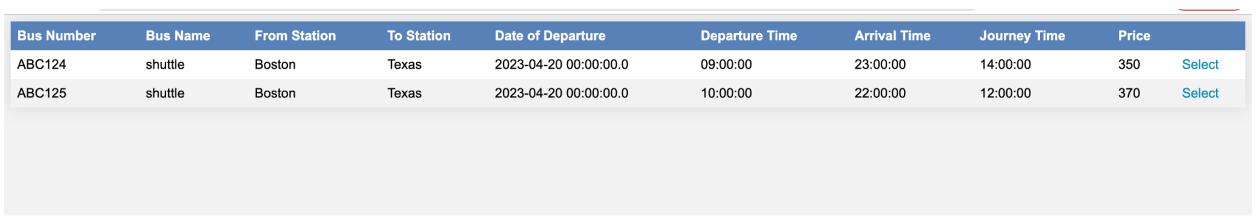
Find Buses

From: New York City

To: Los Angeles

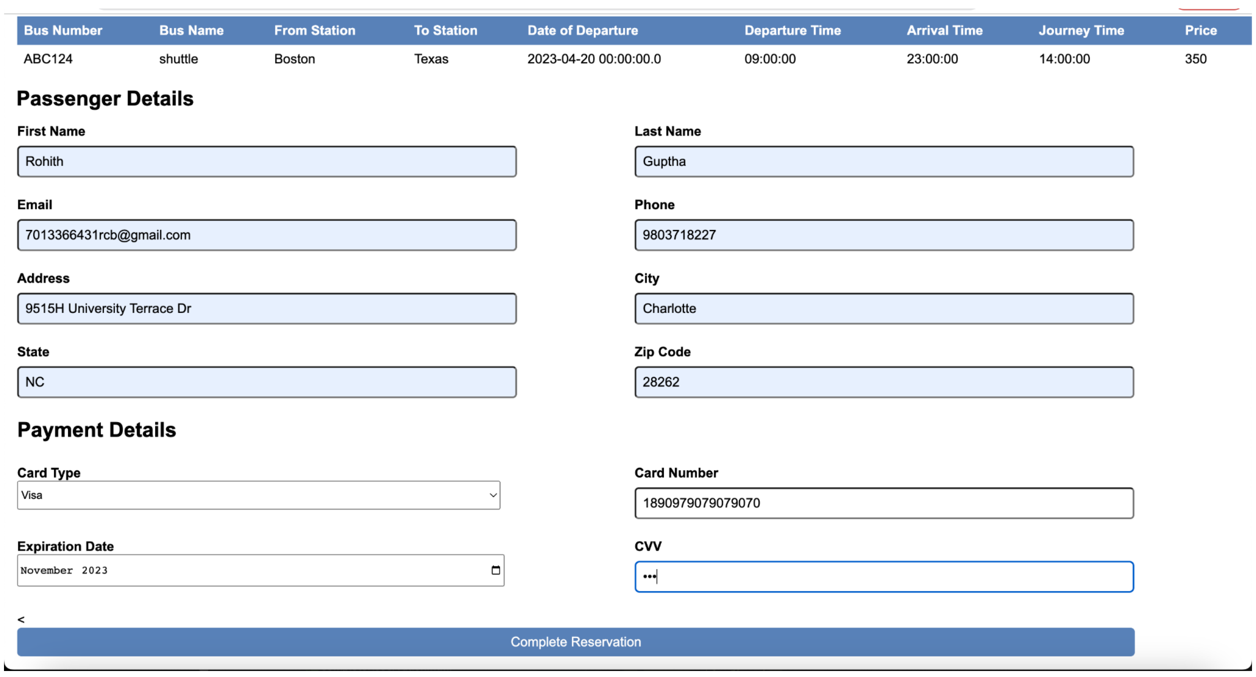
Departure date:

Displaying Buses/Trains/Flights:



Bus Number	Bus Name	From Station	To Station	Date of Departure	Departure Time	Arrival Time	Journey Time	Price	
ABC124	shuttle	Boston	Texas	2023-04-20 00:00:00.0	09:00:00	23:00:00	14:00:00	350	Select
ABC125	shuttle	Boston	Texas	2023-04-20 00:00:00.0	10:00:00	22:00:00	12:00:00	370	Select

Reservation Page:



Bus Number	Bus Name	From Station	To Station	Date of Departure	Departure Time	Arrival Time	Journey Time	Price
ABC124	shuttle	Boston	Texas	2023-04-20 00:00:00.0	09:00:00	23:00:00	14:00:00	350

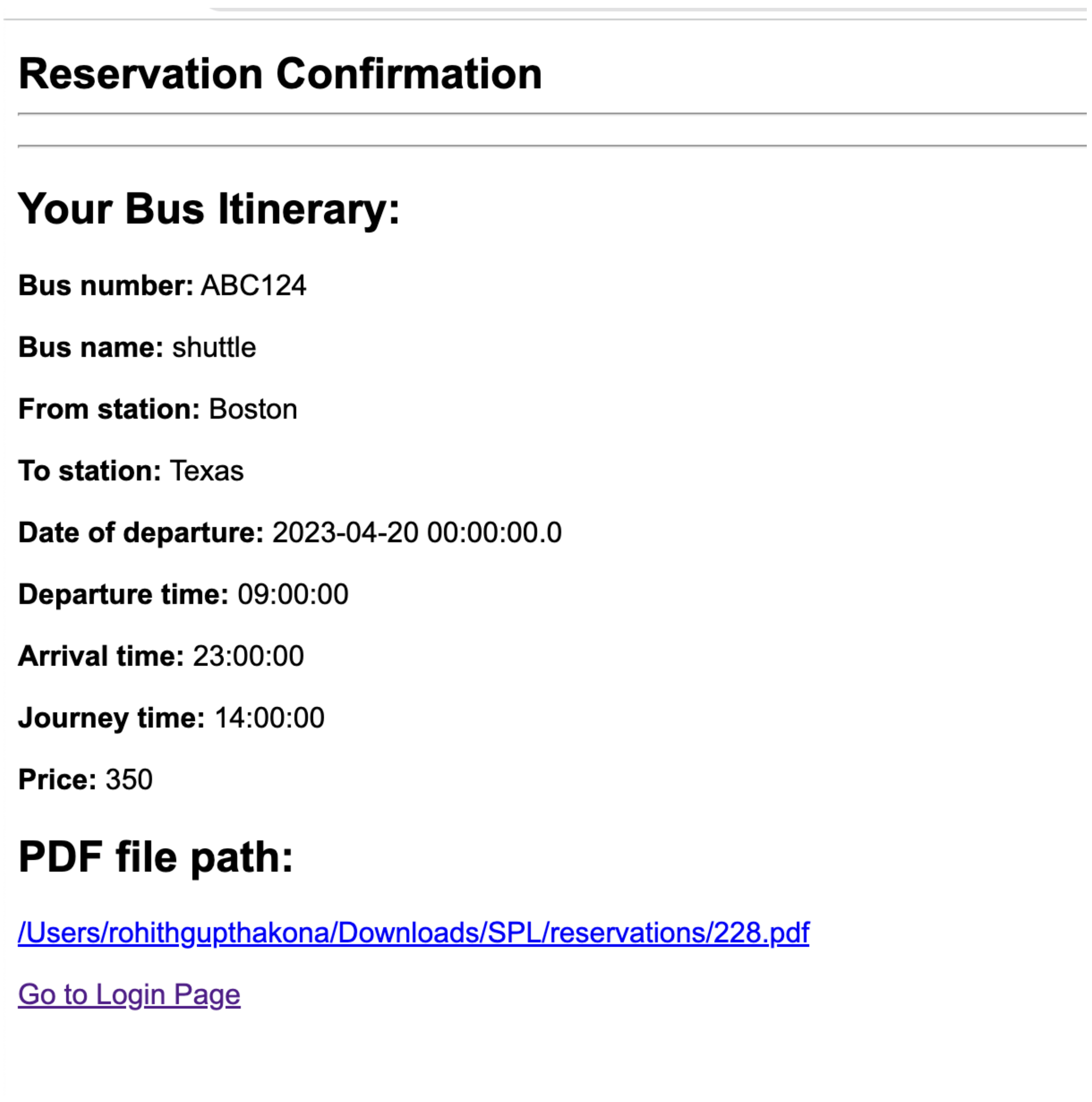
Passenger Details

First Name: Rohith	Last Name: Guptha
Email: 7013366431rcb@gmail.com	Phone: 9803718227
Address: 9515H University Terrace Dr	City: Charlotte
State: NC	Zip Code: 28262

Payment Details

Card Type: Visa	Card Number: 1890979079079070
Expiration Date: November 2023	CVV: •••

Itenary:



Reservation Confirmation

Your Bus Itinerary:

Bus number: ABC124

Bus name: shuttle

From station: Boston

To station: Texas

Date of departure: 2023-04-20 00:00:00.0

Departure time: 09:00:00

Arrival time: 23:00:00

Journey time: 14:00:00

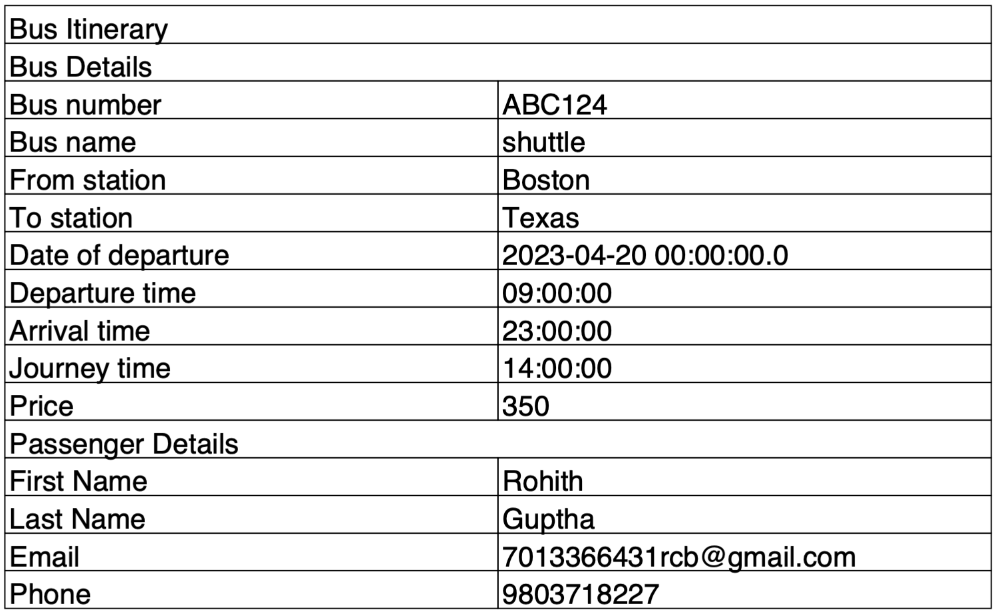
Price: 350

PDF file path:

/Users/rohithgupthakona/Downloads/SPL/reservations/228.pdf

Go to Login Page

Itenary PDF File:



Bus Itinerary	
Bus Details	
Bus number	ABC124
Bus name	shuttle
From station	Boston
To station	Texas
Date of departure	2023-04-20 00:00:00.0
Departure time	09:00:00
Arrival time	23:00:00
Journey time	14:00:00
Price	350
Passenger Details	
First Name	Rohith
Last Name	Guptha
Email	7013366431rcb@gmail.com
Phone	9803718227

Problem Statement:

The problem we are addressing is the inefficiency and complexity of travel planning and management, which can result in wasted time, increased expenses, and decreased traveler satisfaction. Current methods of travel planning often involve waiting in long lines for tickets, managing multiple reservations across different modes of transportation, and manually tracking travel expenses. This can be overwhelming and error-prone, leading to missed opportunities and unnecessary costs. To solve this problem, we are proposing the development of a travel management system software that will simplify and streamline the travel planning and management process. By providing a user-friendly platform for booking and managing travel-related services, the system will allow travelers to easily choose their preferred mode of transportation based on their budget and preferences. This will save time, reduce errors, and enable travelers to make better decisions regarding travel policies and budgets. Overall, our project aims to improve the travel experience for individuals and organizations by providing a more efficient and effective way of managing travel.

Experiments:

- A/B testing of several website layouts, color schemes, and typefaces was one of the studies I carried out. I determined the most appealing design for consumers by analyzing numerous design variants that were effective in engaging users and boosting conversions.
- I experimented with several search algorithms and settings to improve the website's search capability. I discovered the most successful search algorithm for consumers to locate the greatest travel bargains quickly and simply by testing with numerous search parameters such as date ranges, price ranges, and favorite airlines.
- Another experiment I ran was to increase the loading performance of the website by experimenting with various website hosting services, caching methods, and image optimization approaches. I improved user experience and lowered the likelihood of people quitting the website by increasing website performance.
- Furthermore, I experimented with other promotional techniques, such as providing discounts, loyalty awards, or targeted promotions to certain user categories based on demographics, travel history, or search activity. I determined the most effective approaches to boost conversions and improve revenue by tracking the efficacy of these various advertising techniques.
- Overall, I was able to enhance the website to better fulfill the demands of visitors and remain competitive in the travel sector by executing these tests. I ensured that the website remained current and successful in addressing the changing demands and preferences of users by regularly testing and refining its design, functionality, and advertising methods.

Test-Bed Description:

Hardware Configurations:

Any Operating System
Minimum 4GB RAM
Minimum 4GB Disk Space

Software Configurations:

IntelliJ IDE 2022.3.2
Mysql Workbench 8.0
Sublime 2022 Version

Conclusion:

The proposed Travel Management System simplifies the process of booking different modes of transportation in a user-friendly way. Its intuitive interface allows users to save time and avoid errors while booking tickets. The system's ability to generate an itinerary in PDF format adds to its convenience. Overall, the Travel Management System is an efficient and effective solution for travel planning and management.