

ToDo List

Tenzin & Jussif

Was ist diese App?

- Eine einfache Command Line Todo App
- Verwaltet Aufgaben mit Kategorien und Deadline

Hauptfunktionen

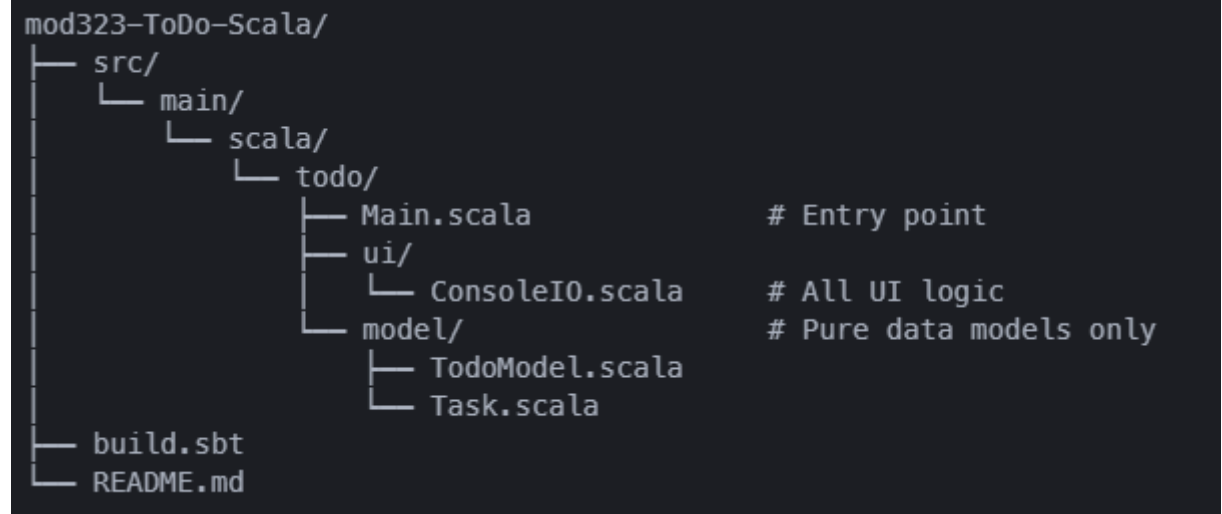
- Aufgaben erstellen, aktualisieren, löschen
- Durchsuchen und filtern
- Statistiken anzeigen
- Überfällige Aufgaben erkennen

Projektarchitektur

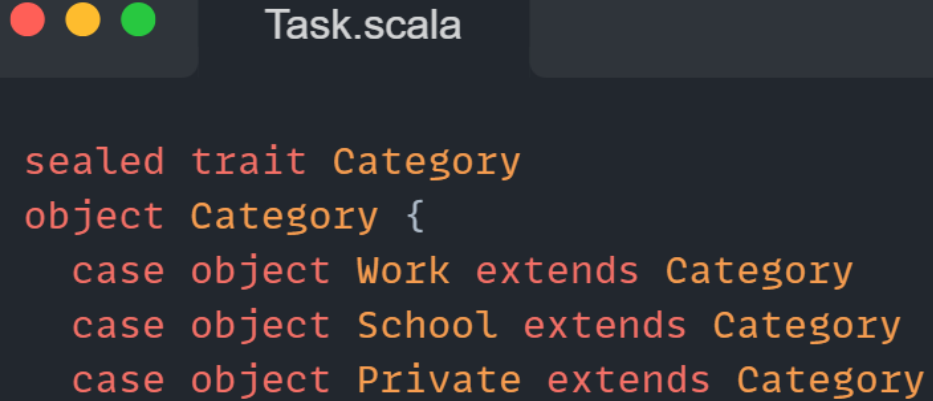
- **Package-Struktur:**

- **Separation of Concerns:**

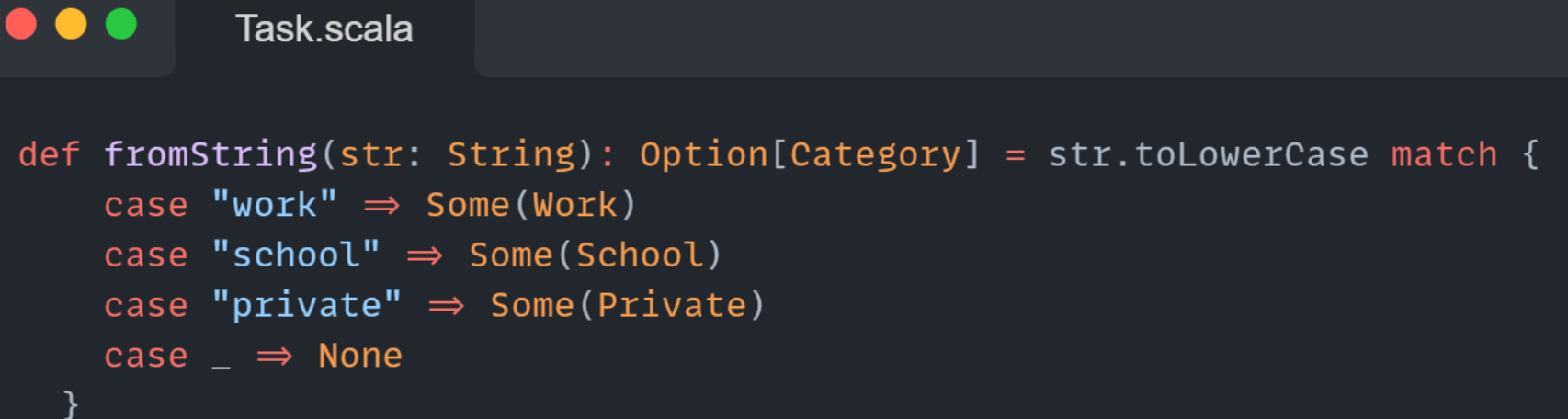
- model/: Reine Logik (keine I/O)
- ui/: Benutzerinteraktion (I/O-Operationen)
- Klare Trennung für bessere maintainability



Interessante Code-Teile

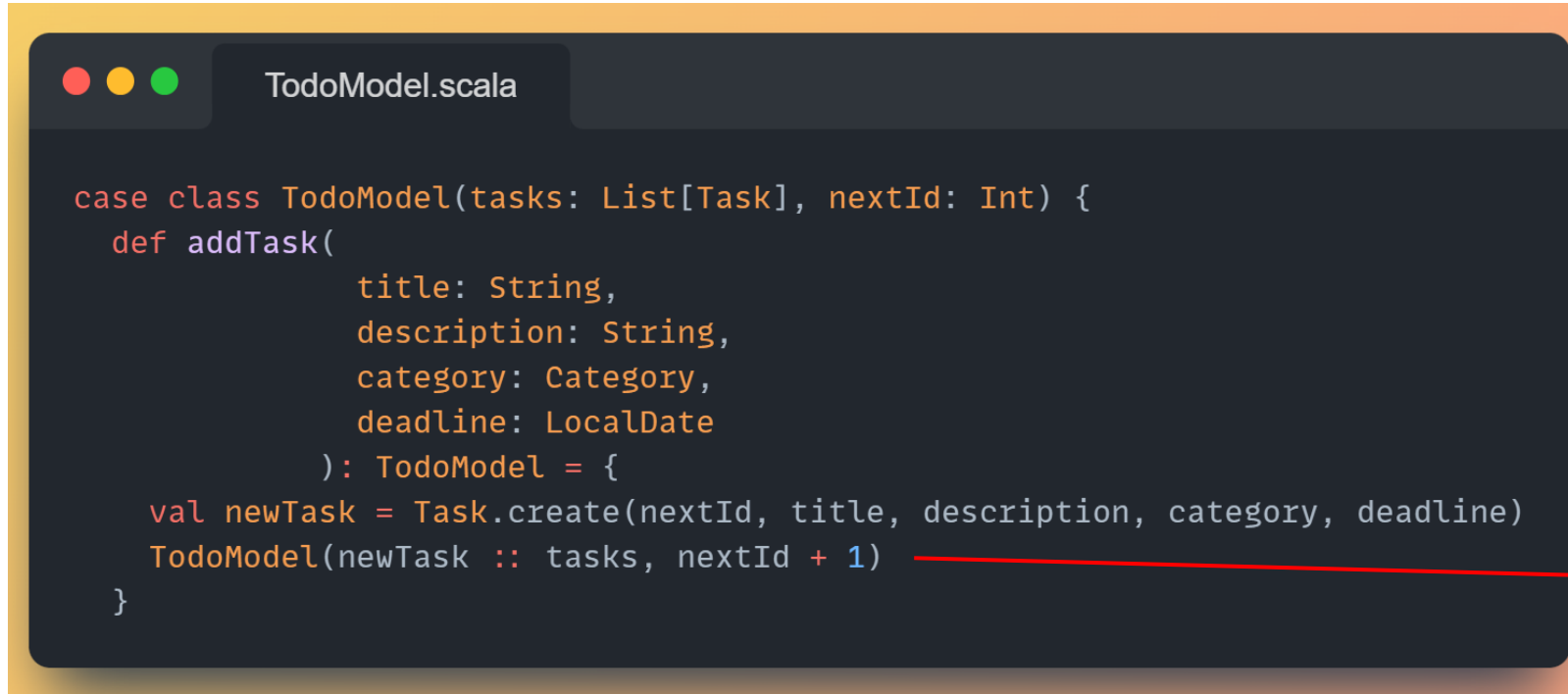


```
sealed trait Category
object Category {
  case object Work extends Category
  case object School extends Category
  case object Private extends Category
```



```
def fromString(str: String): Option[Category] = str.toLowerCase match {
  case "work" => Some(Work)
  case "school" => Some(School)
  case "private" => Some(Private)
  case _ => None
}
```

Interessante Code-Teile



```
case class TodoModel(tasks: List[Task], nextId: Int) {  
  def addTask(  
    title: String,  
    description: String,  
    category: Category,  
    deadline: LocalDate  
  ): TodoModel = {  
    val newTask = Task.create(nextId, title, description, category, deadline)  
    TodoModel(newTask :: tasks, nextId + 1)  
  }  
}
```

// NEUES Model

Interessante Code-Teile

- @tailrec = "Mach aus Rekursion eine Schleife!"

```
ConsoleIO.scala

@tailrec
private def mainMenu(): Unit = {
  println("\n--- Main Menu ---")
  println("1. Add task")
  println("2. List all tasks")
  println("3. Update task status")
  println("4. Remove task")
  println("5. Search tasks")
  println("6. Show statistics")
  println("7. Show overdue tasks")
  println("0. Exit")

  print("Choose an option: ")

  StdIn.readLine() match {
    case "1" => addTaskMenu(); mainMenu()
    case "2" => listTasks(); mainMenu()
    case "3" => updateStatusMenu(); mainMenu()
    case "4" => removeTaskMenu(); mainMenu()
    case "5" => searchTasksMenu(); mainMenu()
    case "6" => showStats(); mainMenu()
    case "7" => showOverdueTasks(); mainMenu()
    case "0" => println("Goodbye!")
    case _ =>
      println("Invalid option, please try again.")
      mainMenu()
  }
}
```

Interessante Code-Teile



```
1 // Tasks with deadline before today and not finished, sorted by deadline
2 def getOverdueTasks: List[Task] = {
3     val today = LocalDate.now()
4     tasks.filter(task => task.deadline.isBefore(today) && task.status != Status.Finished)
5     .sortBy(_.deadline)
6 }
```


Herausforderungen & Lösungen

Herausforderungen:

- Umdenken von imperativer zu funktionaler Programmierung
- Umgang mit unveränderlichen Datenstrukturen
- Fehlerbehandlung ohne Exceptions

Lösungen:

- `Option[T]` statt null
- `Try[T]` für Exception-Handling
- Pattern Matching für Kontrollfluss
- Immutable Collections

Was wir gelernt habe

Technische Erkenntnisse:

- Funktionale Programmierungskonzepte in der Praxis
- Vorteile von unveränderlichen Datenstrukturen
- Scala-spezifische Features: Case Classes, Pattern Matching

Allgemeine Erkenntnisse:

- Wichtigkeit von Code-Organisation
- Agile Entwicklung: "Start small, get it working"

Live Demo