Design Patterns - Modelet e Dizajnit

Patterns te perdorura ne projekt

Hyrje

Design Patterns (Modelet e Dizajnit) jane zgjidhje te provuara per probleme te zakonshme ne dizajnimin e softuerit. Ato ndihmojne ne krijimin e kodit te mirembajteshem, te riperdorshem, dhe te shkallezueshem. Ne projektin tone kemi implementuar disa patterns kryesore qe pershkruhen me poshte.

1. Singleton Pattern

Lokacioni: src/services/.ts, src/lib/prisma.ts*

Cfare eshte? Singleton Pattern siguron qe nje klase te kete vetem nje instance ne te gjithe aplikacionin dhe ofron nje pike globale aksesi per te.

Pse e perdorim?

1. Siguron qe te gjithe komponentet perdorin te njejten instance te sherbimit
2. Parandalon krijimin e shume lidhjeve me databazen (Prisma)
3. Kursen memorien duke shmangur duplikimin e objekteve
4. Lejon testimin duke eksportuar edhe klasen

Shembull nga AuthService.ts:

```
class AuthService {
  async login(email: string, password: string) { /* ... */ }
  async logout(sessionId: string) { /* ... */ }
  hashPassword(password: string): string { /* ... */ }
  verifyPassword(password: string, storedHash: string): boolean { /* ... */ }
}

// Eksportojme nje instance singleton
export const authService = new AuthService()

// Eksportojme edhe klasen per testim
export { AuthService }
```

Shembull nga prisma.ts (Database Singleton):

```
const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined
}

// Perdor instance ekzistuese ose krijo te re
export const prisma = globalForPrisma.prisma ?? new PrismaClient()

// Ruaj ne global per te shmangur shume instanca ne development
if (process.env.NODE_ENV !== 'production') {
  globalForPrisma.prisma = prisma
}
```

2. Service Layer Pattern

Lokacioni: src/services/.ts*

Cfare eshte? Service Layer Pattern krijon nje shtrese abstraksioni qe enkapsulon te gjithe logjiken e biznesit, duke e ndaree ate nga API routes dhe komponentet e UI.

Pse e perdorim?

1. Ndan logjiken e biznesit nga prezantimi (UI) dhe aksesi i te dhenave
2. Ben kodin me te lehte per tu testuar (unit testing)
3. Lejon riperdorimin e logjikes ne shume vende
4. Thjeshton API routes - ato thjesht delegojne tek services

Shembull nga ProjectService.ts:

```
class ProjectService {
  // Krijon projekt me transaction per konsistence
  async createProject(input: CreateProjectInput): Promise<Project> {
    const project = await prisma.$transaction(async (tx) => {
      // Krijo projektin
      const newProject = await tx.project.create({
        data: { title, description, teamLeaderId: createdById }
      })

      // Shto team leader si anetar automatikisht
      await tx.projectUser.create({
        data: {
          projectId: newProject.id,
          userId: createdById,
          role: 'team_leader',
          inviteStatus: 'accepted'
        }
      })
      return newProject
    })

    // Regjistro aktivitetin
    await this.logActivity(createdById, 'create_project', 'project', project.id)
    return project
  }
}
```

3. Provider Pattern (React Context)

Lokacioni: src/contexts/.tsx*

Cfare eshte? Provider Pattern perdor React Context API per te shperndare state dhe funksione ne te gjithe pemen e komponenteve pa pasur nevoje per prop drilling.

Pse e perdorim?

1. Shmang 'prop drilling' - kalimin e props nepermjet shume niveleve
2. Centralizon menaxhimin e state per notifications dhe invites
3. Ben state globalisht te aksesueshem ne cdo komponent
4. Lejon polling automatik per te dhena te reja (cdo 30 sekonda)

Shembull nga NotificationContext.tsx:

```
export function NotificationProvider({ children }: { children: ReactNode }) {
  const [notifications, setNotifications] = useState<Notification[]>([])
  const [unreadCount, setUnreadCount] = useState(0)

  const fetchNotifications = useCallback(async () => {
    const response = await fetch('/api/notifications?limit=50')
    if (response.ok) {
      const data = await response.json()
      setNotifications(data.notifications)
      setUnreadCount(data.unreadCount)
    }
  }, [])

  // Polling cdo 30 sekonda per notifications te reja
  useEffect(() => {
    const interval = setInterval(fetchNotifications, 30000)
    return () => clearInterval(interval)
  }, [fetchNotifications])

  return (
    <NotificationContext.Provider value={{
      notifications, unreadCount, fetchNotifications, markAsRead
    }}>
      {children}
    </NotificationContext.Provider>
  )
}

// Custom hook per perdorim te lehte
export function useNotifications() {
  return useContext(NotificationContext)
}
```

4. Observer Pattern (Event-Driven Notifications)

Lokacioni: src/services/NotificationService.ts

Cfare eshte? Observer Pattern lejon objektet te njoftojne objekte te tjera kur ndodhin ndryshime ne gjendjen e tyre. Ne rastin tone, services njoftojne NotificationService kur ndodhin evente te rendesishme.

Pse e perdorim?

1. Njofton perdoruesit automatikisht kur ndryshon statusi i taskeve
2. Dergon njoftime kur afrohen deadline-t e projekteve
3. Informon anetaret kur dikush pranon ose refuzon ftesen
4. Krijon sistem komunikimi te decentralizuar

Shembull - Kur ndryshon statusi i task:

```
// Ne TaskService kur ndryshon statusi
async changeTaskStatus(taskId: string, newStatus: TaskStatus, changedById: string) {
  // Ndrysho statusin ne database
  const task = await prisma.task.update({
    where: { id: taskId },
    data: { status: newStatus }
  })

  // OBSERVER: Njofto te gjithe anetaret e projektit
  if (newStatus === 'done') {
    await notificationService.notifyTaskCompleted(
      recipientIds,    // Observers - anetaret qe do njoftohen
      changer,         // Kush e perfundoi
      taskInfo,        // Informacion per task
      project          // Projekti perkates
    )
  } else {
    await notificationService.notifyTaskStatusChanged(
      recipientIds, changer, taskInfo, project, newStatus
    )
  }
}
```

5. Repository Pattern

Lokacioni: src/services/.ts (implicit)*

Cfare eshte? Repository Pattern ofron nje abstraksion mbi aksesimin e te dhenave, duke fshehur detajet e queries nga pjesa tjeter e aplikacionit.

Pse e perdorim?

1. Izolon logjiken e aksesit te te dhenave nga logjika e biznesit
2. Ben me te lehte ndryshimin e database (p.sh. nga PostgreSQL ne MongoDB)
3. Centralizon queries - me e lehte per tu optimizuar
4. Lejon mocking te lehte per unit testing

Shembull nga CourseService.ts:

```
class CourseService {
  // Abstraksion per aksesimin e te dhenave
  async getCourseById(courseId: string): Promise<Course | null> {
    return prisma.course.findUnique({
      where: { id: courseId },
      include: {
        professor: { select: { id: true, fullName: true, email: true } },
        _count: { select: { enrollments: true, projects: true } }
      }
    })
  }

  async getEnrolledCourses(studentId: string) {
    const enrollments = await prisma.courseEnrollment.findMany({
      where: { studentId },
      include: { course: { include: { professor: true } } }
    })
    // Kthen domain objects, fsheh detajet e query
    return enrollments.map(e => ({ ...e.course, enrolledAt: e.enrolledAt }))
  }
}
```

6. Factory Pattern

Lokacioni: src/services/.ts (mapToType methods)*

Cfare eshte? Factory Pattern enkapsulon logjiken e krijimit te objekteve, duke e centralizuar transformimin e te dhenave nga databaza ne domain objects.

Pse e perdorim?

1. Centralizon transformimin e te dhenave nga Prisma ne tipet tona
2. Siguron konsistence ne strukturen e objekteve te kthyera
3. Thjeshton menaxhimin e null values (konverton ne undefined)
4. Lejon ndryshime te lehta ne strukture pa prekur shume kod

Shembull nga TaskService.ts:

```
// Factory method per transformim te objekteve
private mapToTaskType(task: PrismaTask): Task {
  return {
    id: task.id,
    projectId: task.projectId,
    title: task.title,
    description: task.description ?? undefined,  // null -> undefined
    priority: task.priority as TaskPriority,     // Type casting
    status: task.status as TaskStatus,
    assigneeId: task.assigneeId ?? undefined,
    ordinal: task.ordinal,
    createdById: task.createdById,
    createdAt: task.createdAt,
    updatedAt: task.updatedAt,
    dueDate: task.dueDate ?? undefined
  }
}
```

7. Controller Pattern (API Routes)

*Lokacioni: src/app/api/**/*.ts*

Cfare eshte? Controller Pattern trajton kerkesat HTTP dhe delegon logjiken tek Service Layer. Ne Next.js, Route Handlers veprojne si controllers.

Pse e perdorim?

1. Ndan trajtimin e HTTP nga logjika e biznesit
2. Centralizon autentifikimin dhe validimin e kerkesave
3. Standardizon formatin e pergjigjeve (JSON)
4. Menaxhon error handling ne nje vend

Shembull nga /api/projects/route.ts:

```
export async function GET(request: Request) {
  try {
    // 1. Autentifikimi
    const user = await getCurrentUser()
    if (!user) {
      return NextResponse.json({ error: "Not authenticated" }, { status: 401 })
    }

    // 2. Merr parametrat
    const { searchParams } = new URL(request.url)
    const status = searchParams.get("status")

    // 3. Delego tek Service Layer
    const projects = await projectService.getProjectsByUser(user.id, status)

    // 4. Kthe pergjigjen
    return NextResponse.json({ projects })
  } catch (error) {
    // 5. Error handling
    return NextResponse.json({ error: "Failed to fetch" }, { status: 500 })
  }
}
```

8. Module Pattern

Lokacioni: src/services/index.ts, src/components/index.ts

Cfare eshte? Module Pattern organizon kodin ne module te pavarura dhe ofron nje pike te vetme eksporti per secilin modul.

Pse e perdorim?

1. Thjeshton importet - nje import per te gjitha services
2. Fsheh implementimin e brendshem te modulit
3. Lejon riorganizimin e brendshem pa ndryshuar importet
4. Krijon API te qarte per cdo modul

Shembull nga services/index.ts:

```
// Barrel export - eksporton te gjitha services nga nje skedar
export { authService, AuthService } from './AuthService'
export { projectService, ProjectService } from './ProjectService'
export { taskService, TaskService } from './TaskService'
export { notificationService } from './NotificationService'
export { courseService } from './CourseService'
export { teamService } from './TeamService'
export { fileService } from './FileService'
export { dashboardService } from './DashboardService'

// Perdorimi:
import { authService, projectService, taskService } from '@/services'
```

9. Facade Pattern

Lokacioni: src/services/DashboardService.ts

Cfare eshte? Facade Pattern ofron nje nderface te thjeshte per nje sistem kompleks, duke fshehur kompleksitetin e nenshtresave.

Pse e perdorim?

1. Thjeshton API per dashboard - nje thirrje merr te gjitha te dhenat
2. Fsheh kompleksitetin e queries te shumefishta
3. Optimizon performancen me Promise.all (paralel)
4. Ofron nderfaqe te qarte per frontend

Shembull nga DashboardService.ts:

```
class DashboardService {
  // FACADE: Nje metode qe mbledh te dhena nga shume burime
  async getDashboardData(userId: string): Promise<DashboardData> {
    // Ekzekuton queries ne paralel per performance
    const [stats, recentProjects] = await Promise.all([
      this.getDashboardStats(userId),      // Statistika
      this.getRecentProjects(userId),      // Projektet e fundit
    ])

    return { stats, recentProjects }
  }

  // Facade per profesor dashboard
  async getProfessorDashboardData(professorId: string) {
    const [stats, courses, recentActivity] = await Promise.all([
      this.getProfessorDashboardStats(professorId),
      this.getProfessorCourses(professorId),
      this.getProfessorRecentActivity(professorId),
    ])

    return { stats, courses, recentActivity }
  }
}
```

Permbledhje e Design Patterns

Pattern	Lokacioni	Qellimi Kryesor
Singleton	Services, Prisma	Nje instance ne gjithe app
Service Layer	src/services/	Ndan logjiken e biznesit
Provider	src/contexts/	State global pa prop drilling
Observer	NotificationService	Njoftimet event-driven
Repository	Services	Abstraksion i aksesit te dhenave
Factory	mapToType methods	Krijim i standardizuar objektesh
Controller	src/app/api/	Trajtim i kerkesave HTTP
Module	index.ts files	Organizim dhe barrel exports
Facade	DashboardService	Interface e thjeshte per sisteme komplekse