2.3. Mono-Repository und Multi-Root Workspaces: Organisation von Fullstack-Projekten

Stellen Sie sich vor, Sie arbeiten an einem größeren Bauvorhaben – einem Komplex aus Wohnhaus, Bürogebäude und Tiefgarage. Alle drei Gebäude gehören zusammen und teilen sich Infrastruktur wie Strom, Wasser und Heizung. Würden Sie für jedes Gebäude separate Baupläne an unterschiedlichen Orten aufbewahren? Nein – Sie würden alle Pläne in einem zentralen Ordner organisieren, aber dennoch klar strukturiert, damit jeder Handwerker (Elektriker, Installateur, Zimmermann) schnell die für ihn relevanten Unterlagen findet.

Genau dieses Prinzip wenden wir in der Softwareentwicklung mit **Mono-Repositories** (kurz: **Monorepo**) und **Multi-Root Workspaces** an.

2.3.1. Was ist ein Mono-Repository?

Ein **Mono-Repository** ist ein einzelnes Git-Repository, das **mehrere zusammenhängende Projekte oder Komponenten** enthält – zum Beispiel eine Mobile-App, eine Web-Anwendung und einen Backend-Server.

Unterschied zu Multi-Repository-Ansatz

Aspekt	Multi-Repository	Mono-Repository
Struktur	Jedes Projekt hat eigenes Git-Repo	Alle Projekte in einem Git-Repo
Dependencies	Versionierung über Package-Manager	Direkter Zugriff auf gemeinsame Module
Code-Sharing	Über externe Packages	Direkt über relative Pfade
Versionskontrolle	Unabhängige Versionen	Koordinierte Versionen
CI/CD	Separate Pipelines	Zentrale Pipeline mit selektiven Builds

Beispielstruktur eines Fullstack-Monorepos

```
yourapp/
├── mobile/                 # Flutter-App für iOS/Android
│   ├── lib/
│   ├── test/
│   └── pubspec.yaml
│
├── admin-web/              # React-Web-Admin
│   ├── src/
│   ├── tests/
│   └── package.json
│
├── server/                 # Spring Boot Backend
│   ├── src/main/java/
│   ├── src/test/java/
│   └── pom.xml
│
├── shared-resources/       # Gemeinsame Ressourcen
```

```
│   ├── api-contracts/       # OpenAPI-Spezifikationen
│   ├── design-tokens/       # Design-System
│   └── documentation/
│
├── docs/                    # Projekt-Dokumentation
├── infra/                   # Infrastruktur (Docker, Terraform)
├── .github/workflows/       # CI/CD Pipelines
│   ├── mobile.yml
│   ├── web.yml
│   └── server.yml
│
├── .gitignore
└── README.md
```

> 💡 **Merksatz:** Ein Mono-Repository organisiert mehrere zusammengehörige Projekte in einem einzigen Repository – ähnlich wie ein Aktenordner mit verschiedenen Registern für unterschiedliche Themen.

2.3.2. Vorteile und Herausforderungen von Mono-Repositories

Vorteile ✅

1. Zentrale Code-Verwaltung

- Alle Projektteile an einem Ort
- Einheitliches Versionskontrollsystem
- Atomic Commits über Projektgrenzen hinweg

2. Code-Sharing und Wiederverwendung

- Gemeinsame Utilities, Modelle und Konfigurationen
- Keine doppelte Implementierung
- Konsistente Namenskonventionen

```
// Beispiel: Gemeinsames Datenmodell
// shared-resources/models/User.ts
export interface User {
  id: string;
  username: string;
  email: string;
}

// admin-web/src/services/UserService.ts
import { User } from '../../../shared-resources/models/User';

// mobile/lib/models/user.dart
// Kann aus derselben OpenAPI-Spec generiert werden
```

3. Koordinierte Entwicklung

- API-Änderungen werden sofort in allen Clients sichtbar
- Breaking Changes sind leichter zu erkennen
- Synchrone Feature-Entwicklung über alle Plattformen

4. Vereinfachte Dependency-Verwaltung

- Zentrale Verwaltung von Tool-Versionen
- Einheitliche Entwicklungsumgebung
- Reduzierte Konflikte bei Abhängigkeiten

5. Konsistente Standards

- Gemeinsame Code-Formatierung (ESLint, Prettier, Checkstyle)
- Einheitliche Commit-Konventionen
- Zentrale Dokumentation

Herausforderungen ⚠️

1. Repository-Größe

- Großes Repository kann Klonen/Fetchen verlangsamen
- *Lösung:* Git Shallow Clones, Git LFS für große Dateien

2. Build-Performance

- Vollständiger Build dauert lange
- *Lösung:* Selektive Builds nur für geänderte Komponenten

3. Zugriffsrechte

- Alle Entwickler haben Zugriff auf allen Code
- *Lösung:* Code-Owner-Dateien, Branch-Protection

4. Komplexität für neue Entwickler

- Große Codebasis kann überwältigend sein
- *Lösung:* Gute Dokumentation, schrittweises Onboarding

> 🔍 **Praxistipp:** Die meisten Herausforderungen lassen sich durch klare Strukturierung, gute Tooling-Unterstützung und definierte Best Practices lösen.

Quellen

- Multi-root Workspaces - VS Code Documentation
- Managing Projects in VSCode: Workspaces and Folder Structures

---

2.3.3. Multi-Root Workspaces in VS Code

Multi-Root Workspaces sind eine VS Code-Funktion, die es ermöglicht, **mehrere Ordner in einer einzigen VS Code-Instanz** zu öffnen. Dies ist ideal für die Arbeit mit Mono-Repositories.

Was ist ein VS Code Workspace?

Ein Workspace ist eine Konfigurationsdatei (`.code-workspace`), die folgendes definiert:

- **Welche Ordner** geöffnet werden sollen
- **Workspace-spezifische Einstellungen**
- **Empfohlene Extensions**
- **Tasks und Launch-Konfigurationen**

Workspace-Strategien für Fullstack-Projekte

Für ein Fullstack-Monorepo empfiehlt es sich, **mehrere Workspace-Dateien** zu erstellen:

1. Full Workspace (`yourapp-full.code-workspace`)

Öffnet **alle Projektkomponenten** gleichzeitig – ideal für Cross-Platform-Entwicklung.

```
{
  "folders": [
    { "path": "." },              // Root-Verzeichnis
    { "path": "mobile" },         // Flutter-App
    { "path": "admin-web" },      // React-Admin
    { "path": "server" }          // Spring Boot
  ],
  "settings": {
    "files.exclude": {
      "**/.git": true,
      "**/node_modules": true,
      "**/build": true,
      "**/dist": true,
      "**/target": true
    }
  }
}
```

Anwendungsfall:

- Feature-Entwicklung über alle Plattformen
- API-Änderungen mit direkter Integration in Clients
- Full-Stack-Debugging

2. Focused Workspaces (technologiespezifisch)

Mobile Workspace (`yourapp-mobile.code-workspace`)

```
{
  "folders": [{ "path": "mobile" }],
  "settings": {
    "files.exclude": {
      "../admin-web": true,
      "../server": true,
      "../infra": true
    }
  }
}
```

Web Workspace (`yourapp-web.code-workspace`)

```
{
  "folders": [{ "path": "admin-web" }],
  "settings": {
    "files.exclude": {
      "../mobile": true,
      "../server": true,
      "../infra": true
    }
  }
}
```

Server Workspace (`yourapp-server.code-workspace`)

```
{
  "folders": [{ "path": "server" }],
  "settings": {
    "files.exclude": {
      "../mobile": true,
      "../admin-web": true,
      "../infra": true
    }
  }
}
```

Vorteile von Focused Workspaces:

- **Performance:** Nur relevante Dateien werden indiziert
- **Fokus:** Reduzierter visueller Clutter
- **Extension-Verwaltung:** Nur notwendige Extensions aktiv
- **Team-Spezialisierung:** Frontend-Entwickler nutzen Web-Workspace, Mobile-Entwickler nutzen Mobile-Workspace

> 💡 **Merksatz:** Mehrere Workspace-Dateien erlauben es, denselben Code aus verschiedenen Perspektiven zu betrachten – wie verschiedene Ansichten desselben Gebäudeplans (Grundriss, Elektrik, Sanitär).

Workspace öffnen

```
# Full Workspace öffnen
code yourapp-full.code-workspace

# Focused Mobile Workspace öffnen
code yourapp-mobile.code-workspace

# Server Workspace öffnen
code yourapp-server.code-workspace
```

Quellen

- Multi-root Workspaces - VS Code Documentation

2.3.4. Zentrale VS Code-Konfiguration

Um die verschiedenen Workspaces effizient zu nutzen, wird eine **zentrale `.vscode/`-Konfiguration** im Repository-Root angelegt.

Verzeichnisstruktur

```
yourapp/
├── .vscode/
│   ├── tasks.json          # Zentrale Tasks für alle Projekte
│   ├── launch.json         # Debug-Konfigurationen
│   ├── settings.json       # Globale Einstellungen
│   └── extensions.json     # Empfohlene Extensions
```

Tasks: Projektübergreifende Aufgaben

Die `tasks.json` definiert **Tasks für alle Projektteile** mit korrektem `cwd` (Working Directory).

```
{
  "version": "2.0.0",
  "tasks": [
    // Mobile Tasks
    {
      "label": "mobile: pub get",
      "type": "shell",
      "command": "flutter pub get",
      "options": { "cwd": "${workspaceFolder:yourapp}/mobile" },
      "problemMatcher": []
    },
    {
      "label": "mobile: run",
      "type": "shell",
      "command": "flutter run",
      "options": { "cwd": "${workspaceFolder:yourapp}/mobile" }
    },

    // Web Tasks
    {
      "label": "web: install",
      "type": "shell",
      "command": "npm ci",
      "options": { "cwd": "${workspaceFolder:yourapp}/admin-web" }
    },
    {
      "label": "web: dev",
      "type": "shell",
      "command": "npm run dev",
      "options": { "cwd": "${workspaceFolder:yourapp}/admin-web" },
      "isBackground": true,
```

```
      "problemMatcher": "$tsc-watch"
    },

    // Server Tasks
    {
      "label": "server: mvn verify",
      "type": "shell",
      "command": "mvn -B clean verify",
      "options": { "cwd": "${workspaceFolder:yourapp}/server" }
    },
    {
      "label": "server: run",
      "type": "shell",
      "command": "mvn spring-boot:run",
      "options": { "cwd": "${workspaceFolder:yourapp}/server" }
    }
  ]
}
```

Task ausführen:

1. `Ctrl+Shift+P` → **Tasks: Run Task**
2. Task auswählen (z.B. `web: dev`)

> 🔍 **Tipp:** Tasks mit `isBackground: true` laufen im Hintergrund weiter – ideal für Development-Server.

Launch Configurations: Cross-Platform Debugging

Die `launch.json` ermöglicht **simultanes Debugging** mehrerer Komponenten.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "mobile: Flutter",
      "request": "launch",
      "type": "dart",
      "cwd": "${workspaceFolder:yourapp}/mobile",
      "program": "lib/main.dart"
    },
    {
      "name": "web: Vite dev server",
      "type": "node",
      "request": "launch",
      "cwd": "${workspaceFolder:yourapp}/admin-web",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"]
    },
    {
      "name": "server: Spring Boot",
```

```
      "type": "java",
      "request": "launch",
      "mainClass": "com.example.Application",
      "projectName": "server",
      "cwd": "${workspaceFolder:yourapp}/server"
    }
  ],
  "compounds": [
    {
      "name": "Full Stack: Web + Server",
      "configurations": [
        "web: Vite dev server",
        "server: Spring Boot"
      ]
    }
  ]
}
```

Compound Launch Configuration ermöglicht es, mehrere Debugger gleichzeitig zu starten:

- `Run and Debug` → **Full Stack: Web + Server**
- Beide Komponenten werden parallel gestartet
- Breakpoints funktionieren in beiden Codebases

Globale Einstellungen (settings.json)

```
{
  "files.exclude": {
    "**/.git": true,
    "**/node_modules": true,
    "**/build": true,
    "**/dist": true,
    "**/target": true
  },
  "editor.formatOnSave": true,
  "dart.analysisExcludedFolders": ["**/build/**"],
  "java.configuration.updateBuildConfiguration": "interactive",
  "eslint.useFlatConfig": true
}
```

Empfohlene Extensions (extensions.json)

```
{
  "recommendations": [
    // Dart/Flutter
    "Dart-Code.dart-code",
    "Dart-Code.flutter",
```

```
    // Java/Spring Boot
    "redhat.java",
    "vscjava.vscode-java-pack",
    "vmware.vscode-boot-dev-pack",

    // Web/TypeScript
    "esbenp.prettier-vscode",
    "dbaeumer.vscode-eslint",

    // Allgemein
    "eamodio.gitlens",
    "github.vscode-github-actions",
    "ms-azuretools.vscode-docker"
  ]
}
```

> ⚠️ **Achtung:** Extensions werden workspace-übergreifend vorgeschlagen. Mit Focused Workspaces können Sie selektiv nur relevante Extensions aktivieren.

Quellen

- Multi-root Workspaces - VS Code Documentation

2.3.5. Shared Resources: Code-Sharing im Monorepo

Ein großer Vorteil von Mono-Repositories ist die Möglichkeit, **gemeinsame Ressourcen** zwischen verschiedenen Projektteilen zu teilen.

Typische Shared Resources

```
shared-resources/
├── api-contracts/           # OpenAPI-Spezifikationen
│   └── openapi.yaml
│
├── design-tokens/           # Design-System (Farben, Typografie)
│   ├── tokens.json
│   └── README.md
│
├── documentation/           # Cross-Platform Dokumentation
│   ├── architecture.md
│   └── development-guide.md
│
└── scripts/                # Build & Deployment Scripts
    ├── deploy.sh
    └── version-bump.sh
```

Beispiel 1: API-Contracts (OpenAPI)

Definiere einmal, nutze überall:

```yaml
# shared-resources/api-contracts/openapi.yaml
openapi: 3.0.3
info:
  title: YourApp API
  version: 1.0.0
paths:
  /users/{id}:
    get:
      summary: Get user by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: User found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
```

```
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        username:
          type: string
        email:
          type: string
```

Nutzung in verschiedenen Komponenten:

Web-Client generieren (TypeScript/Axios):

```
# Ausführen von: yourapp/admin-web/
npm install @openapitools/openapi-generator-cli --save-dev

npx openapi-generator-cli generate \
  -i ../shared-resources/api-contracts/openapi.yaml \
  -g typescript-axios \
  -o src/generated/api
```

Mobile-Client generieren (Dart):

```
# Ausführen von: yourapp/mobile/
flutter pub global activate openapi_generator_cli

openapi-generator generate \
  -i ../shared-resources/api-contracts/openapi.yaml \
  -g dart \
  -o lib/generated/api
```

Server-Stubs generieren (Spring Boot):

```
<!-- Konfiguration in: yourapp/server/pom.xml -->
<plugin>
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  <version>7.0.0</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
```

```
      <configuration>
        <inputSpec>${project.basedir}/../shared-resources/api-
contracts/openapi.yaml</inputSpec>
        <generatorName>spring</generatorName>
        <configOptions>
          <interfaceOnly>true</interfaceOnly>
        </configOptions>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
# Ausführen von: yourapp/server/
mvn clean compile
```

> **Merksatz:** API-First bedeutet: OpenAPI-Spec als Single Source of Truth, aus der alle Clients und Server-Stubs generiert werden.

Beispiel 2: Design Tokens

Zentrale Definition:

```
// shared-resources/design-tokens/tokens.json
{
  "colors": {
    "primary": "#007AFF",
    "secondary": "#5856D6",
    "error": "#FF3B30"
  },
  "typography": {
    "fontFamily": "Inter",
    "baseFontSize": 16
  }
}
```

Nutzung in Web (CSS Variables):

```
// admin-web/src/styles/tokens.css
:root {
  --color-primary: #007AFF;
  --color-secondary: #5856D6;
  --font-family: 'Inter', sans-serif;
}
```

Nutzung in Mobile (Flutter Theme):

```
// mobile/lib/theme/app_theme.dart
import 'package:flutter/material.dart';

class AppTheme {
  static const Color primary = Color(0xFF007AFF);
  static const Color secondary = Color(0xFF5856D6);
}
```

Beispiel 3: Gemeinsame Dokumentation

Zentrale Architektur-Dokumentation:

```
// shared-resources/documentation/architecture.md

# System-Architektur

## Komponenten-Übersicht

- **Mobile App:** Flutter (iOS/Android)
- **Admin Web:** React + TypeScript + Vite
- **Backend:** Spring Boot + PostgreSQL

## Deployment-Architektur

[Deployment-Diagramm hier]
```

Diese Dokumentation ist für **alle Teams** einsehbar und wird **zentral** gepflegt.

Quellen

- Multi-root Workspaces - VS Code Documentation
- OpenAPI Generator

2.3.6. CI/CD für Mono-Repositories

Eine große Herausforderung bei Mono-Repositories ist die **effiziente CI/CD-Pipeline**, die nur die geänderten Teile baut und testet.

Path-basierte Trigger

GitHub Actions ermöglicht **selektive Workflows** basierend auf geänderten Dateipfaden.

1. **Mobile Workflow (`.github/workflows/mobile.yml`)**

```
name: Mobile CI
on:
push:
    paths:
    - "mobile/**"           # Nur triggern bei Änderungen in mobile/
    - "shared-resources/api-contracts/**"  # Oder bei API-Änderungen
pull_request:
    paths:
    - "mobile/**"
    - "shared-resources/api-contracts/**"

jobs:
test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - uses: subosito/flutter-action@v2
        with:
        flutter-version: 'stable'
        cache: true

    - name: Install dependencies
        run: flutter pub get
        working-directory: mobile

    - name: Run tests
        run: flutter test --coverage
        working-directory: mobile

    - name: Build APK
        run: flutter build apk
        working-directory: mobile
```

2. **Web Workflow (`.github/workflows/web.yml`)**

```
name: Web CI
on:
```

```
push:
    paths: ["admin-web/**", "shared-resources/api-contracts/**"]
pull_request:
    paths: ["admin-web/**", "shared-resources/api-contracts/**"]

jobs:
build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - uses: actions/setup-node@v4
        with:
        node-version: '20'
        cache: 'npm'
        cache-dependency-path: 'admin-web/package-lock.json'

    - run: npm ci
        working-directory: admin-web

    - run: npm run lint
        working-directory: admin-web

    - run: npm test -- --ci
        working-directory: admin-web

    - run: npm run build
        working-directory: admin-web
```

3. **Server Workflow (`.github/workflows/server.yml`)**

```
name: Server CI
on:
push:
    paths: ["server/**", "shared-resources/api-contracts/**"]
pull_request:
    paths: ["server/**", "shared-resources/api-contracts/**"]

jobs:
build:
    runs-on: ubuntu-latest
    permissions:
    contents: read
    packages: write

    steps:
    - uses: actions/checkout@v4

    - uses: actions/setup-java@v4
        with:
        distribution: 'temurin'
        java-version: '21'
```

```
          cache: 'maven'

      - name: Build and Test
          run: mvn -B clean verify
          working-directory: server

      - name: Build Docker Image
          uses: docker/build-push-action@v6
          with:
          context: server
          push: true
          tags: ghcr.io/${{ github.repository }}/server:${{ github.sha }}
```

Vorteile dieser Struktur

- ✅ **Effiziente Builds:** Nur geänderte Komponenten werden gebaut
- ✅ **Parallele Ausführung:** Mobile, Web und Server-Builds laufen parallel
- ✅ **Schnelles Feedback:** Entwickler warten nicht auf irrelevante Builds
- ✅ **Resource-Optimierung:** CI-Minuten werden gespart

> 🔍 **Vertiefung:** Für noch komplexere Setups kann man Tools wie **Nx**, **Turborepo** oder **Bazel** verwenden, die automatisch Dependency-Graphen analysieren und nur betroffene Projekte bauen.

Quellen

- GitHub Actions - Workflow Triggers

2.3.7. Best Practices und Entwicklungsworkflow

Empfohlener Entwicklungsworkflow

1. **Initial Setup**

```
# Repository klonen
git clone https://github.com/yourorg/yourapp.git
cd yourapp

# Full Workspace öffnen
code yourapp-full.code-workspace
```

2. **Abhängigkeiten installieren**

Terminal 1: Server

```
cd server
mvn clean install
```

Terminal 2: Web

```
cd admin-web
npm install
```

Terminal 3: Mobile

```
cd mobile
flutter pub get
```

> 💡 **Tipp:** VS Code Tasks automatisieren diese Schritte – einfach `Ctrl+Shift+P` → **Tasks: Run Task** → `mobile: pub get` etc.

3. **Feature-Entwicklung**

Szenario: Neues User-Management-Feature über alle Plattformen

Schritt 1: API-First – OpenAPI-Spec definieren

```
# shared-resources/api-contracts/openapi.yaml
paths:
/users:
    post:
```

```
    summary: Create new user
    requestBody:
        content:
        application/json:
            schema:
            $ref: '#/components/schemas/CreateUserRequest'
```

Schritt 2: Backend implementieren

```
// server/src/main/java/com/example/api/UserController.java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody CreateUserRequest
request) {
        // Implementation
    }
}
```

Schritt 3: Web-Client generieren und nutzen

```
// admin-web/src/services/UserService.ts
import { UsersApi } from '../generated/api';

export class UserService {
private api = new UsersApi();

async createUser(username: string, email: string) {
    return await this.api.usersPost({ username, email });
}
}
```

Schritt 4: Mobile-Client generieren und nutzen

```
// mobile/lib/services/user_service.dart
import 'package:yourapp/generated/api.dart';

class UserService {
final UsersApi _api = UsersApi();

Future<User> createUser(String username, String email) async {
    return await _api.usersPost(
    CreateUserRequest(username: username, email: email)
    );
```

```
  }
  }
```

4. **Git-Workflow**

```
# Feature-Branch für alle Plattformen
git checkout -b feature/user-management

# Commits pro Komponente (atomare Commits)
git add shared-resources/api-contracts/
git commit -m "api: add user management endpoints"

git add server/
git commit -m "backend: implement user management"

git add admin-web/
git commit -m "web: add user management UI"

git add mobile/
git commit -m "mobile: add user management screens"

# Push und Pull Request
git push origin feature/user-management
```

Code-Organisation Best Practices

Namenskonventionen - Konsistente Namensgebung über Plattformen:

Konzept	Backend (Java)	Web (TypeScript)	Mobile (Dart)
Entität	`User`	`User`	`User`
Service	`UserService`	`UserService`	`UserService`
Controller/Screen	`UserController`	`UserListPage`	`UserListScreen`

Feature-basierte Organisation

```
```
server/
└── src/main/java/com/example/
    └── features/
        ├── auth/
        │   ├── AuthController.java
        │   ├── AuthService.java
        │   └── AuthRepository.java
        └── user/
            ├── UserController.java
            ├── UserService.java
            └── UserRepository.java
```
```

```
admin-web/
└── src/
    └── features/
        ├── auth/
        │   ├── AuthService.ts
        │   └── LoginPage.tsx
        └── user/
            ├── UserService.ts
            ├── UserListPage.tsx
            └── UserDetailPage.tsx
```

Performance-Optimierung

1. **VS Code Performance bei großen Repos**

   Problem: Mit allen Projekten geöffnet kann VS Code langsamer werden.

   Lösungen:

   1. **Focused Workspaces verwenden**
   - Öffnen Sie nur `yourapp-mobile.code-workspace` für Mobile-Entwicklung
   - Wechseln Sie zu `yourapp-server.code-workspace` für Backend-Arbeit
   2. **File Watcher Limits erhöhen**

```
// settings.json
{
"files.watcherExclude": {
    "**/node_modules/**": true,
    "**/build/**": true,
    "**/target/**": true,
    "**/.dart_tool/**": true
}
}
```

   3. **Extension-Management**
   - Deaktivieren Sie nicht benötigte Extensions workspace-spezifisch
   - Beispiel: Flutter-Extension nur in Mobile-Workspace aktivieren

Zusammenarbeit im Team

1. **Code-Ownership**

```
# CODEOWNERS Datei im Repository-Root
# Backend Team
/server/                @backend-team
```

```
# Frontend Team
/admin-web/             @frontend-team

# Mobile Team
/mobile/                @mobile-team

# Shared Resources - Approval von allen Teams erforderlich
/shared-resources/      @backend-team @frontend-team @mobile-team
```

2. **Pull Request Templates**

```
## Beschreibung
<!-- Was wurde geändert und warum? -->

## Betroffene Komponenten
- [ ] Mobile App
- [ ] Web Admin
- [ ] Backend Server
- [ ] Shared Resources

## Testing
- [ ] Unit Tests hinzugefügt/aktualisiert
- [ ] Integration Tests laufen
- [ ] Manuell getestet auf [Platform]

## Breaking Changes
<!-- Gibt es Breaking Changes? Welche Migrationsschritte sind nötig? -->
```

> ⚠️ **Achtung:** Änderungen an `shared-resources/api-contracts/` erfordern besondere Sorgfalt, da sie alle Plattformen betreffen!

Quellen

- Managing Projects in VSCode: Workspaces and Folder Structures
- Multi-root Workspaces - VS Code Documentation

2.3.9. Praxis: Neue Projekte in Unterordnern anlegen

In einem Mono-Repository kannst du für verschiedene Plattformen (z.B. Mobile, Backend, Web) jeweils ein eigenes Projekt im passenden Unterordner anlegen. Hier findest du Schritt-für-Schritt-Anleitungen für die gängigsten Technologien:

Flutter-Projekt im Ordner `mobile` erstellen

1. **Wechsle ins Verzeichnis:**

```
cd mobile
```

2. **Neues Flutter-Projekt initialisieren:**

```
flutter create .
```

(Das `.` sorgt dafür, dass das Projekt im aktuellen Ordner angelegt wird.) 3. **Abhängigkeiten installieren und Projekt starten:**

```
flutter pub get
flutter run
```

Spring Boot-Projekt im Ordner `backend` erstellen

1. **Wechsle ins Verzeichnis:**

```
cd backend
```

2. **Projekt mit Spring Initializr generieren:**

- Online: https://start.spring.io/ (ZIP herunterladen und hier entpacken)
- Oder per CLI:

  ```
  spring init --dependencies=web,data-jpa --build=maven .
  ```

(Passe die Dependencies nach Bedarf an.) 3. **Projekt bauen und starten:**

```
mvn clean install
mvn spring-boot:run
```

React Native (mit TypeScript) im Ordner `web-app` erstellen

1. **Wechsle ins Verzeichnis:**

```
cd web-app
```

2. **Neues React Native-Projekt mit TypeScript anlegen:**

```
npx react-native init MyApp --template react-native-template-typescript
```

(Ersetze `MyApp` ggf. durch den gewünschten Projektnamen. Das Projekt wird im Unterordner `MyApp` angelegt.) 3. **Optional: Projektdateien in den aktuellen Ordner verschieben und `MyApp`-Ordner löschen.** 4. **Abhängigkeiten installieren und App starten:**

```
npm install
npx react-native run-android # oder run-ios
```

Hinweise:

- Lege die jeweiligen Projekte immer in den passenden Unterordnern an (`mobile`, `backend`, `web-app`).
- Die Build- und Konfigurationsdateien (z.B. `pubspec.yaml`, `pom.xml`, `package.json`) gehören in den jeweiligen Projektordner.
- Die parallele Entwicklung und Verwaltung erfolgt über den Multi-Root-Workspace in VS Code.

> **Tipp:** Mit dieser Struktur kannst du verschiedene Technologien sauber getrennt, aber zentral versioniert und verwaltet entwickeln.

Zusammenfassung

Kernkonzepte

Mono-Repository:

- Ein Git-Repository für mehrere zusammenhängende Projekte
- Ermöglicht Code-Sharing, koordinierte Entwicklung und zentrale Verwaltung
- Beispiel: `mobile/`, `admin-web/`, `server/` in einem Repository

Multi-Root Workspaces:

- VS Code-Feature zum Öffnen mehrerer Ordner in einer Instanz
- Verschiedene Workspace-Dateien für unterschiedliche Szenarien
- Full Workspace vs. Focused Workspaces

Zentrale Konfiguration:

- `.vscode/tasks.json`: Projektübergreifende Tasks
- `.vscode/launch.json`: Cross-Platform-Debugging
- `.vscode/settings.json`: Globale Einstellungen
- `.vscode/extensions.json`: Empfohlene Extensions

Shared Resources:

- API-Contracts (OpenAPI) als Single Source of Truth
- Design Tokens für konsistentes UI
- Gemeinsame Dokumentation

CI/CD:

- Path-basierte Triggers für selektive Builds
- Separate Workflows für jede Komponente
- Effiziente Nutzung von CI-Ressourcen

Wann sollten Sie ein Mono-Repository verwenden?

✅ **Geeignet für:**

- Fullstack-Projekte mit mehreren Clients (Web, Mobile) und einem Backend
- Teams, die eng zusammenarbeiten
- Projekte mit vielen gemeinsamen Abhängigkeiten
- Koordinierte Feature-Entwicklung über Plattformen

❌ **Nicht ideal für:**

- Vollständig unabhängige Projekte ohne Code-Sharing
- Sehr große Projekte mit hunderten von Microservices
- Teams mit strikter Codebasis-Trennung

Checkliste für Ihr Semesterprojekt

- ☐ Repository-Struktur mit `mobile/`, `admin-web/`, `server/`, `shared-resources/` angelegt
- ☐ Workspace-Dateien erstellt: `yourapp-full.code-workspace`, `yourapp-mobile.code-workspace`, etc.
- ☐ `.vscode/tasks.json` mit allen wichtigen Tasks konfiguriert
- ☐ `.vscode/launch.json` mit Debug-Konfigurationen eingerichtet
- ☐ `.vscode/extensions.json` mit empfohlenen Extensions erstellt
- ☐ OpenAPI-Spec in `shared-resources/api-contracts/` definiert
- ☐ GitHub Actions Workflows für `mobile`, `web`, `server` konfiguriert
- ☐ `.gitignore` für alle Technologien (Flutter, Node, Maven) angepasst
- ☐ `CODEOWNERS` Datei für Code-Ownership erstellt
- ☐ Pull Request Template angelegt

> 💡 **Abschließender Merksatz:** Ein gut strukturiertes Mono-Repository mit Multi-Root Workspaces ist wie ein professioneller Werkzeugkoffer – alles an einem Ort, aber perfekt organisiert, sodass jeder

> Handwerker schnell die richtigen Werkzeuge findet.

Quellen

- Multi-root Workspaces - VS Code Documentation
- Managing Projects in VSCode: Workspaces and Folder Structures

---