Vom Spaghetti-Code zur Zwiebel: Spring Boot gewürzt mit DDD

CodeBuzz 2026

codecamp N
RE•INVENT

Wie modelliert man Fachlichkeit im Backend?

Damit die Fachlichkeit besser sichtbar wird,
um die Wartbarkeit des Backends zu erhöhen.

Agenda

codecamp N
RE•INVENT

1. Problemstellung
2. Fachsprache im Code abbilden
3. Äußere Struktur schaffen
4. Innere Struktur schaffen
5. Kopplung an Technik entfernen
6. Fazit

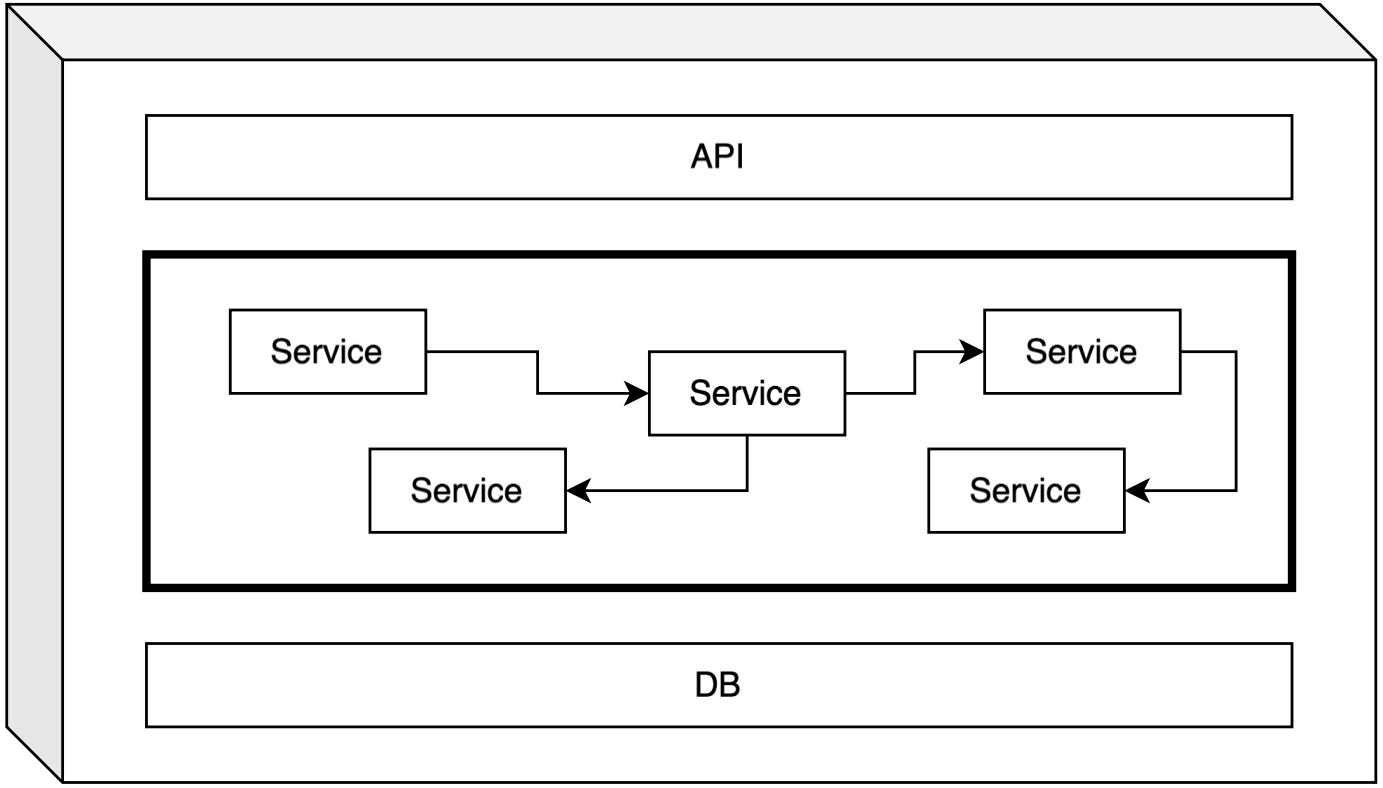
Problemstellung

CodeBuzz 2026

codecamp^N
RE•INVENT

Problemstellung (Ausgangslage)

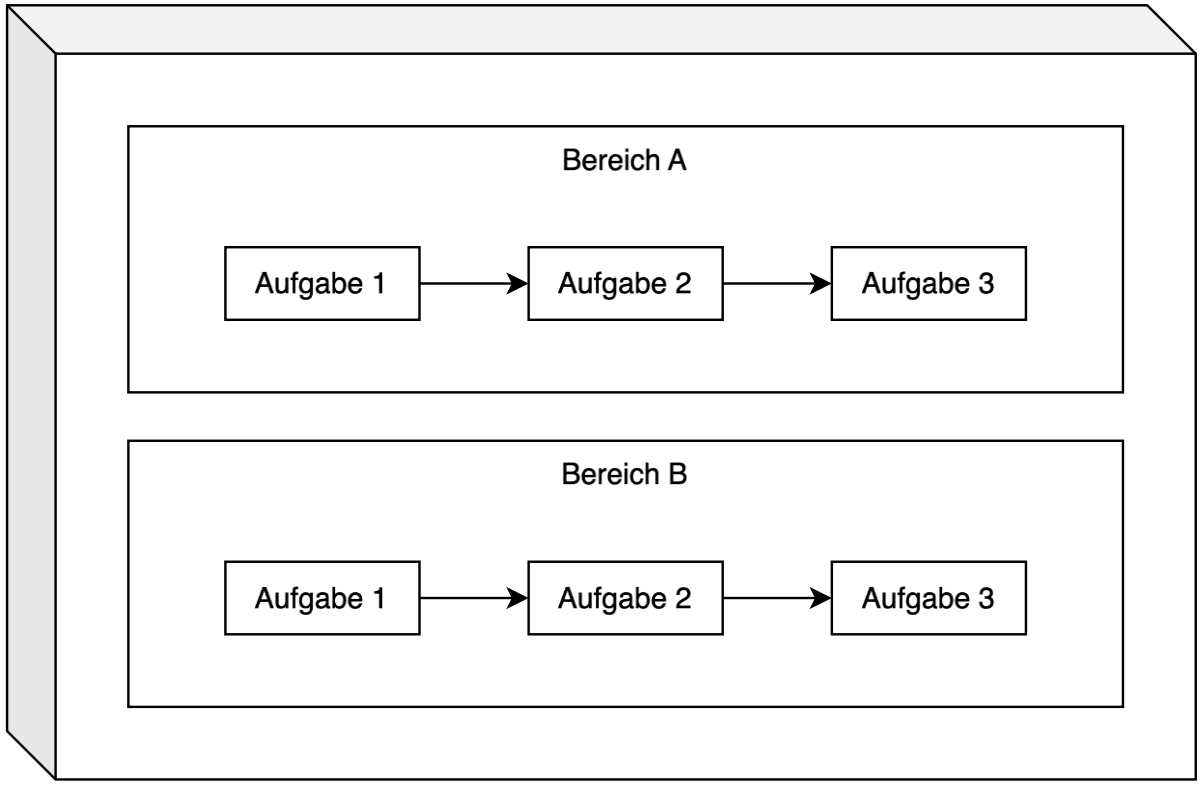
Backend-Anwendungen werden intern schlecht wartbar



- Fachlichkeit in vielen Service-Klassen verteilt
- unübersichtlich
- viele Querverbindungen
- unklare Zuständigkeiten
- technisch getriebener Aufbau & Namen

Problemstellung (Ziel)

Fachlich gut wartbare/verständliche Backend-Anwendung



- Was? Verständliche Namen, klare Zuständigkeiten und Call-Flow
- Wie? Fachlichkeit sichtbar machen und Architektur anpassen
- Ausnahme: einfach CRUD-App

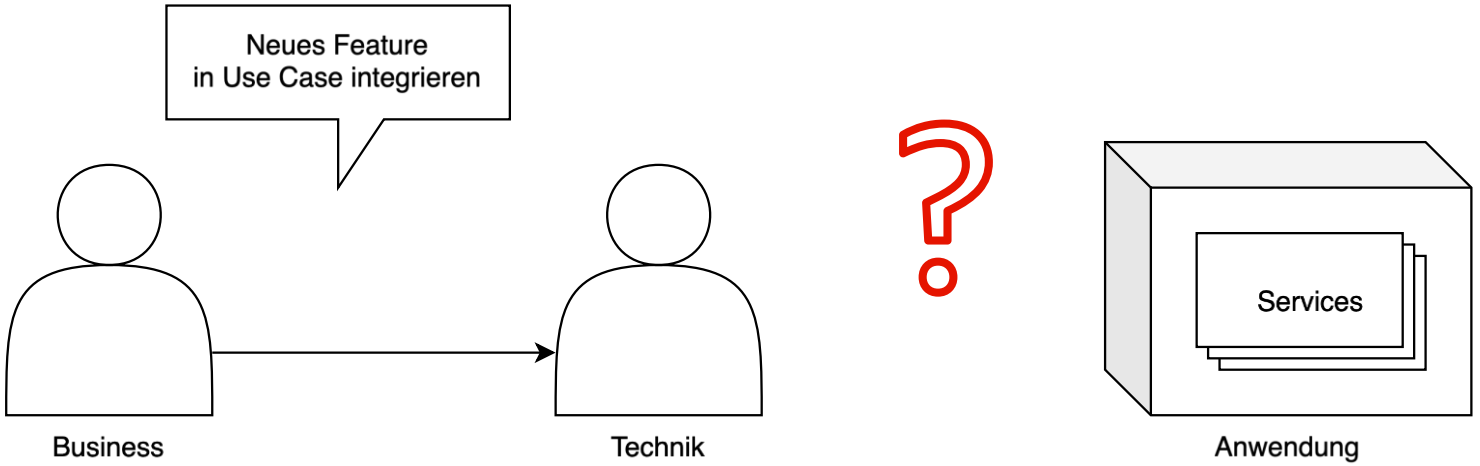
Fachsprache im Code abbilden

CodeBuzz 2026

codecamp^N
RE•INVENT

Fachsprache im Code abbilden

Namensgebung gemäß Fachexperten



- Mit Fachexperten reden (+ Glossar)
- Nomen und Verben fachlich (+ deutsch), z.B. Buch ausleihen, statt update Book
- kein mentales Mapping mehr, Missverständnisse reduzieren
- DDD: "Ubiquitous Language"

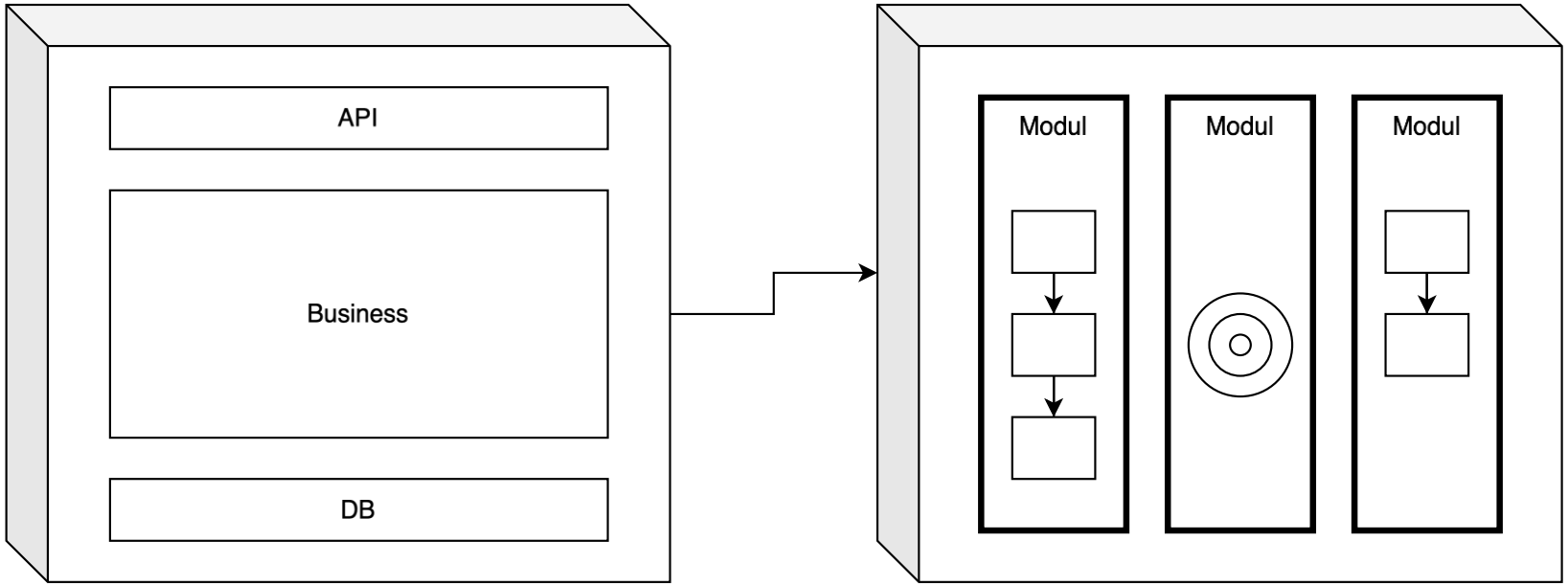
Äußere Struktur schaffen

CodeBuzz 2026

codecamp^N
RE•INVENT

Äußere Struktur schaffen

Fachliche Module schneiden



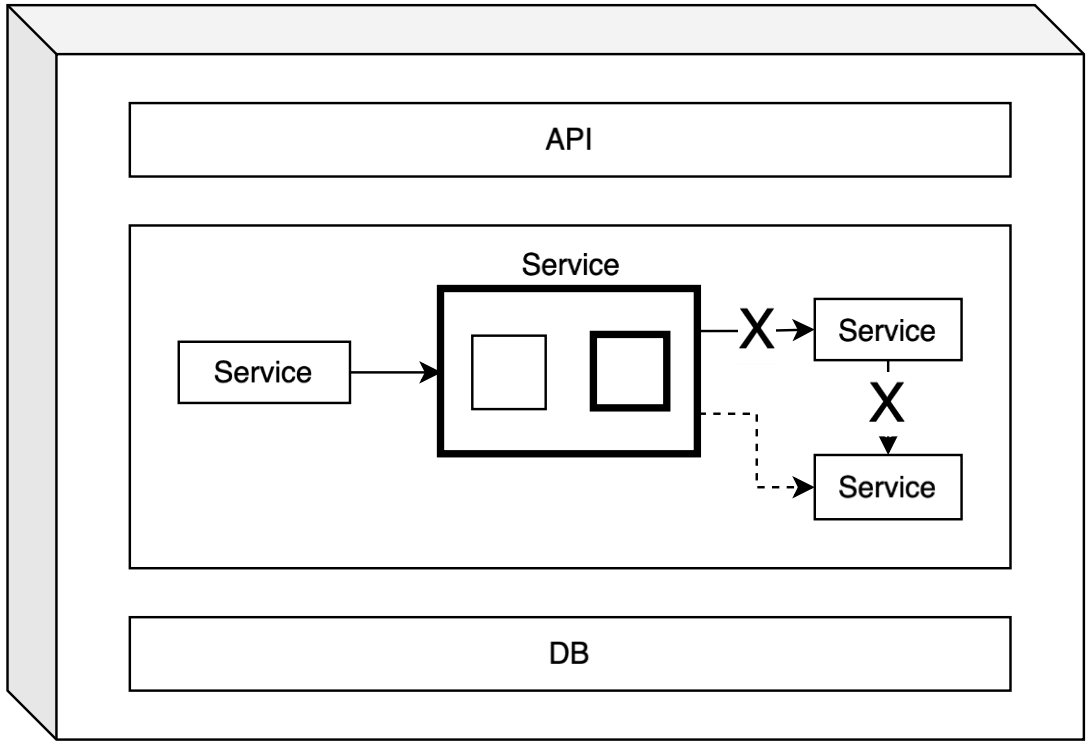
- Gibt es fachliche Grenzen (Bounded Contexts)?
- Eigene top-level Packages, z.B. „Bestellung“ oder „Lieferung“
- Spring Modulith prüft Grenzen
- Microservices?
- DDD: "Strategisches Design"

Innere Struktur schaffen

CodeBuzz 2026

Innere Struktur schaffen

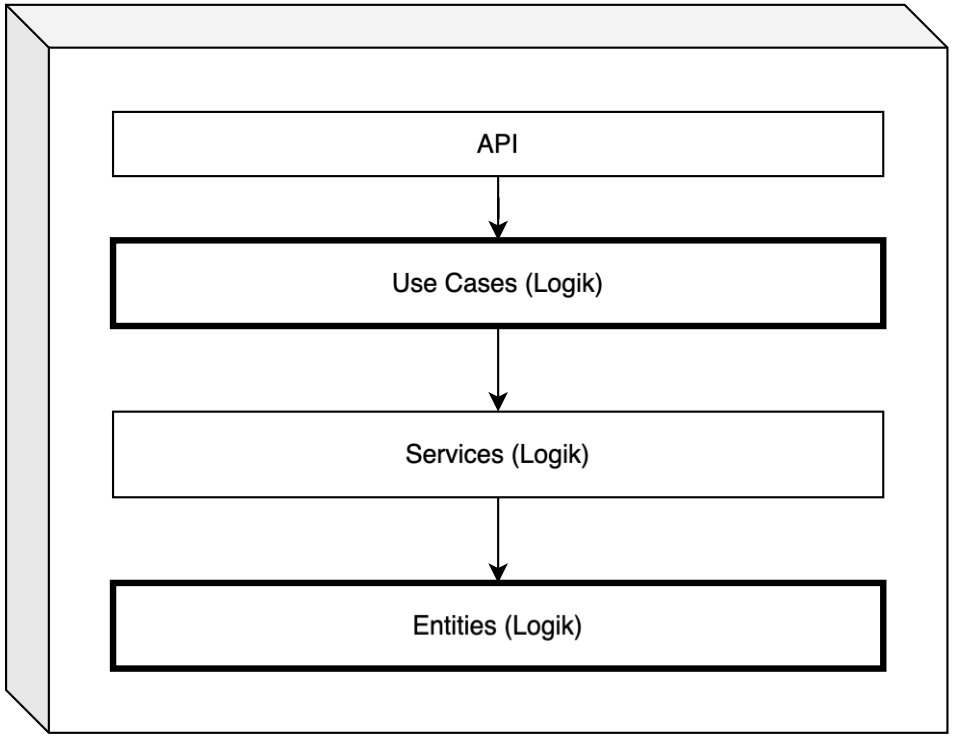
Vorarbeit: Refactoring von bestehenden Service-Klassen



- Vor dem Umbau erst einmal aufräumen & vereinfachen
- Klassen mit ähnlichen fachlichen Aufgaben zusammen ziehen
- Klassen, die nur weiterleiten, entfernen

Innere Struktur schaffen

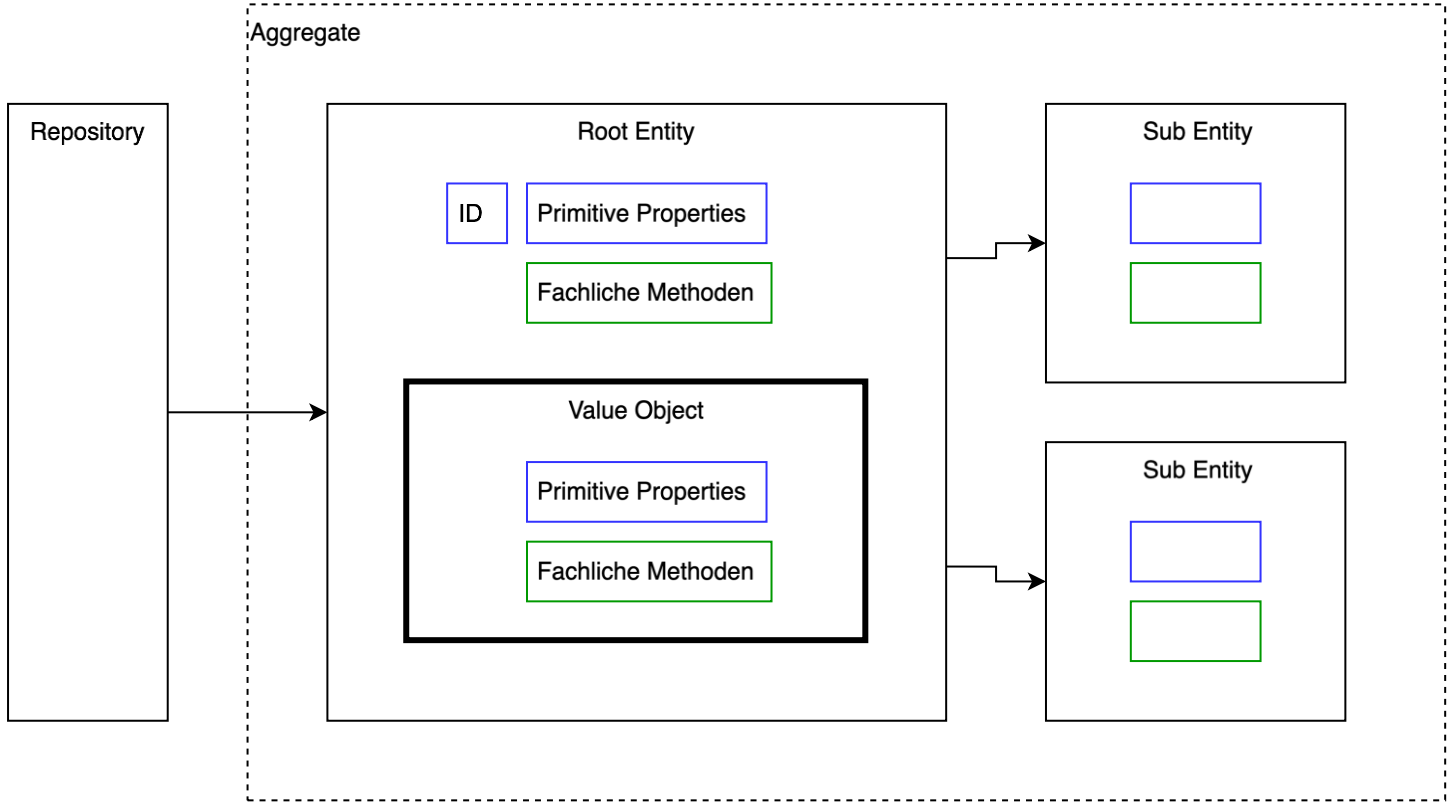
Logik aus Service-Klassen herausziehen



- Orchestrierung in Use Cases verschieben
- Validierungsregeln in Entities verschieben
- übrig bleiben kleine Domain-Services
- Vorteil: klare Zuständigkeiten, nachvollziehbarer Call-Flow

Innere Struktur schaffen

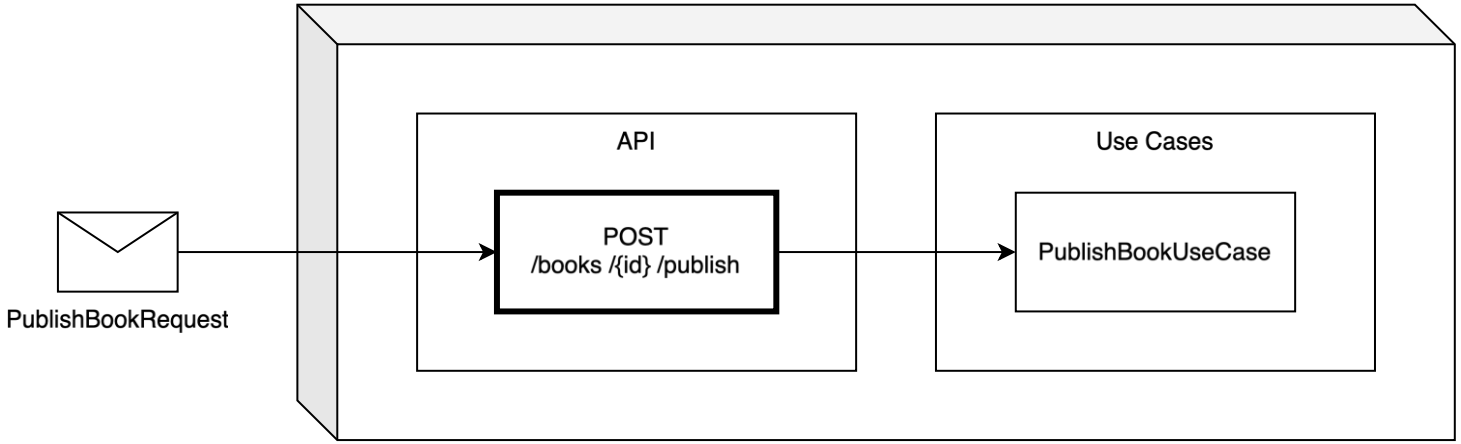
Domain modellieren



- bereits da: Logik in Entity und fachliche Namen für Methoden (keine Setter)
- Einführung von Value Objects, z.B. „Geld“ oder „EmailAdresse“
- Aggregates schneiden
- DDD: "Taktisches Design"

Innere Struktur schaffen

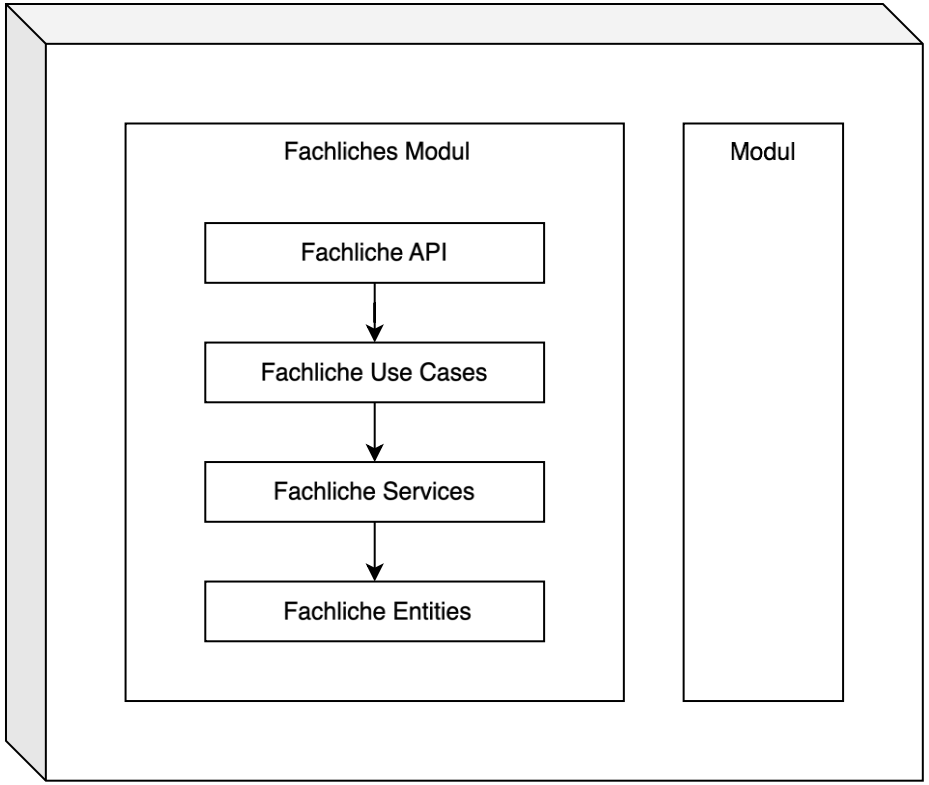
Domain-driven REST-API



- Trigger der Use Cases über Action-Endpunkte
- Verben nutzen satt 100%-RESTful z.B. /publish
- Fachlichkeit statt CRUD
- Prozesse statt Resourcen/Daten
- Commands in CQRS

Innere Struktur schaffen

Zwischenziel erreicht



- Code ist übersichtlicher und fachlich orientiert
- Bis hier genügt es in vielen Projekten
- aber: Kopplung an Infrastruktur noch vorhanden (ist das schlimm?)

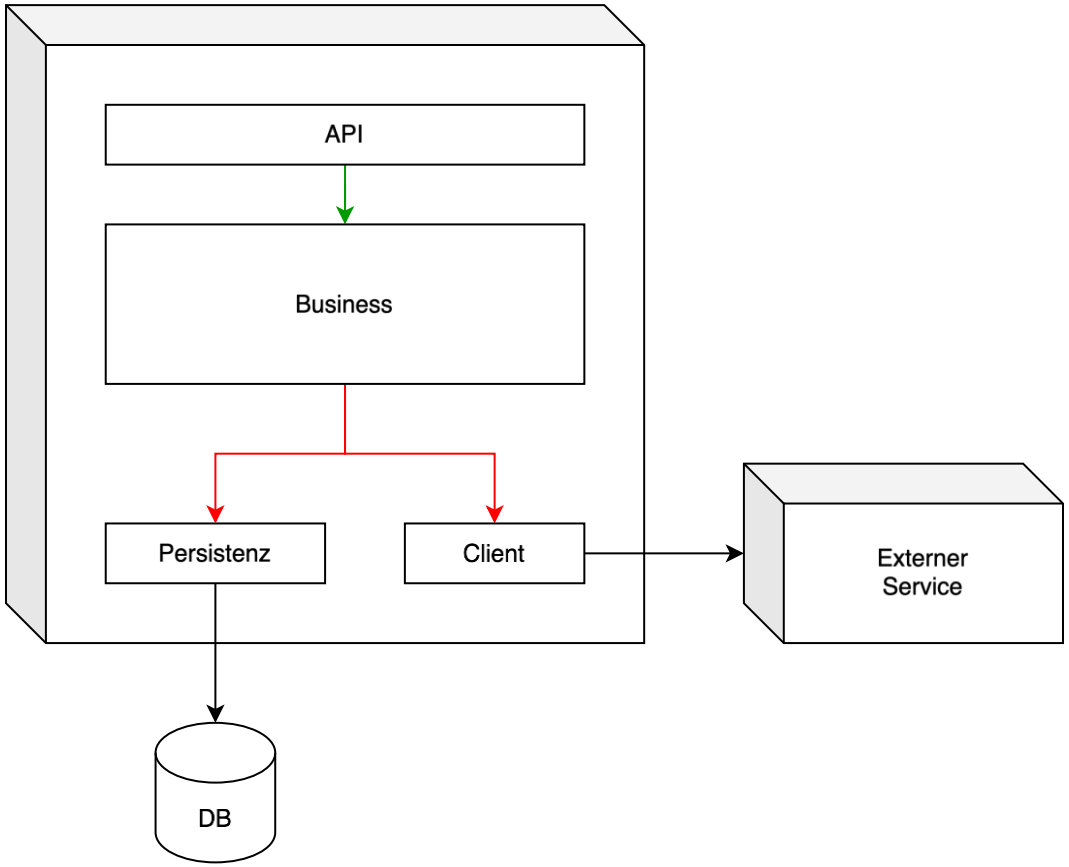
Kopplung an Technik entfernen

CodeBuzz 2026

codecamp^N
RE•INVENT

Kopplung an Technik entfernen

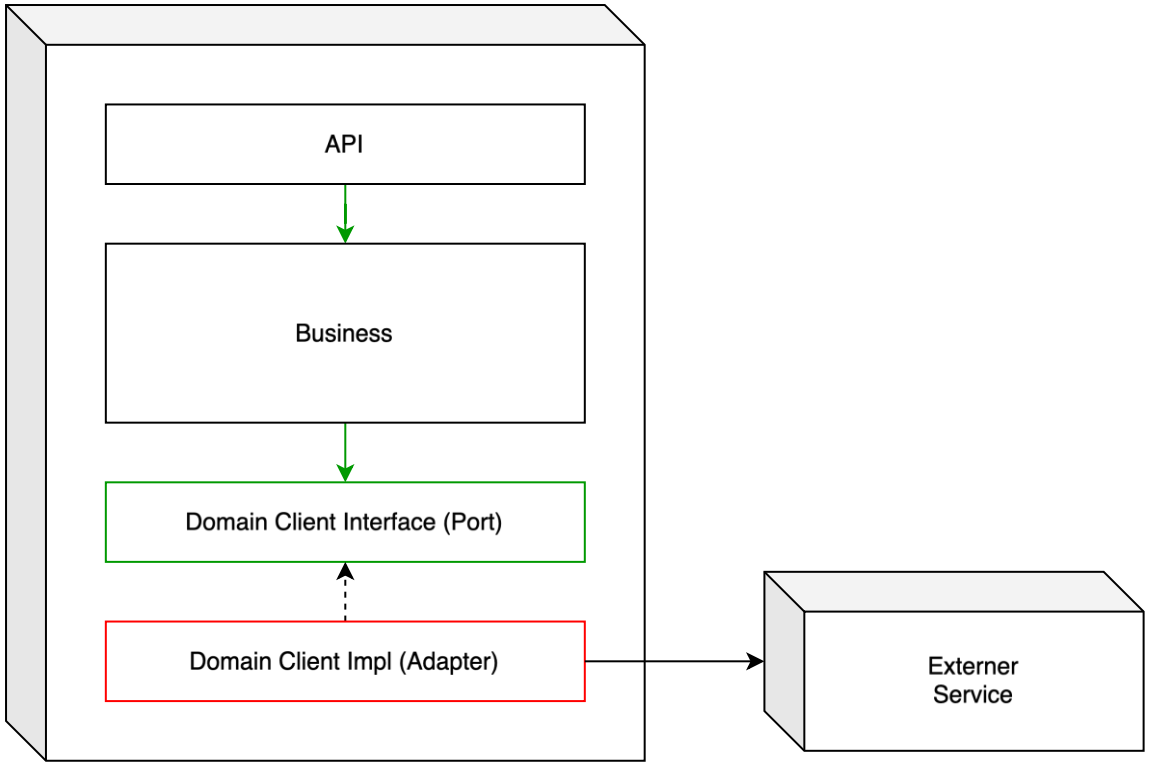
Welche Verbindungen zur Technik gibt es?



- Eingehende Aufrufe sind unproblematisch (API zu Business)
- Ausgehende Aufrufe erzeugen Kopplung an Technik (Business zu Persistenz bzw Service-Client)
- Lösung: Abhängigkeiten umdrehen (DIP)

Kopplung an Technik entfernen

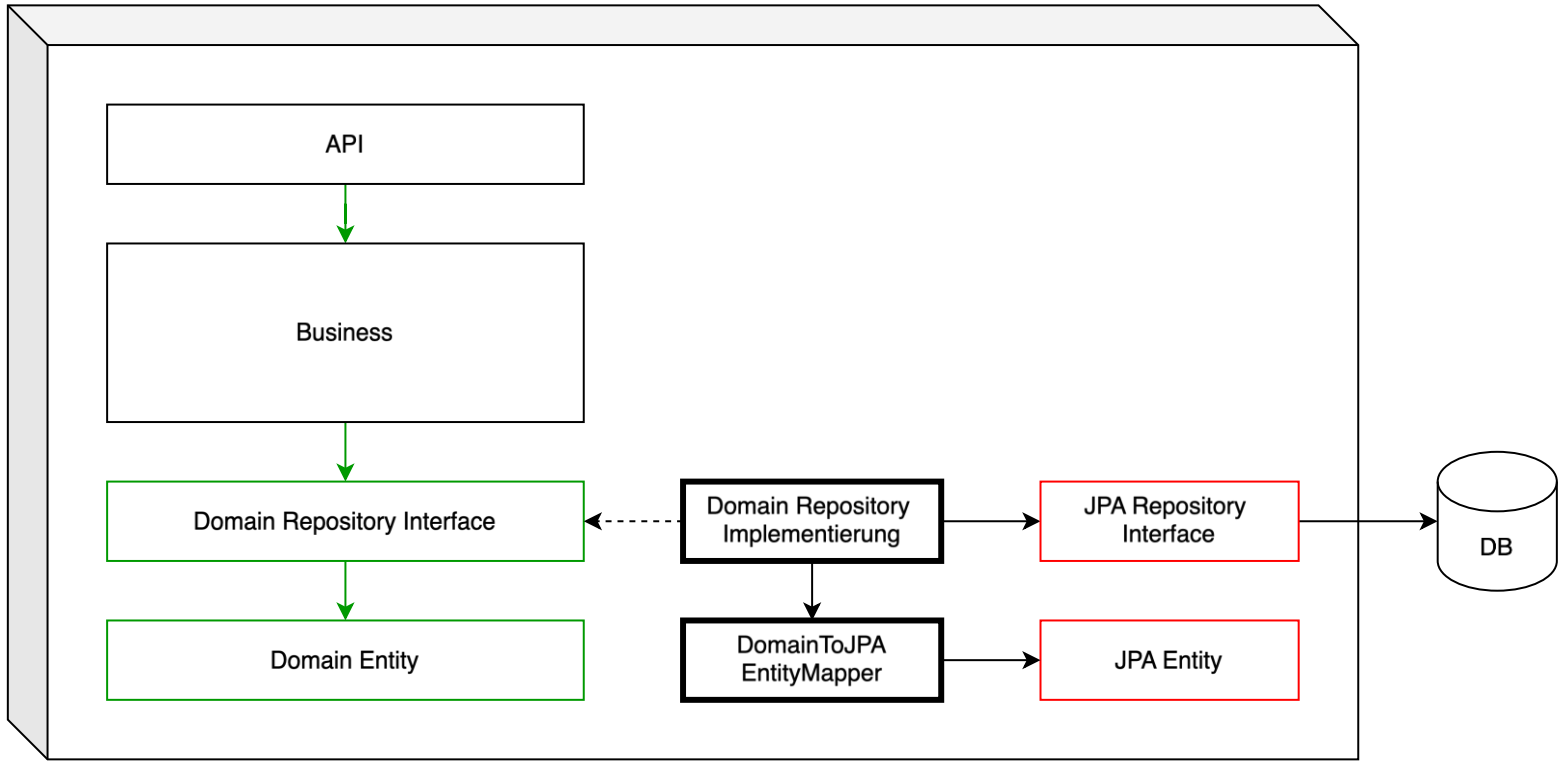
API-Client auftrennen



- Der fachliche Kern sollte nicht direkt von Infrastruktur abhängen
- Entkopplung der API-Clients über Interfaces (Ports)
- Implementierung (Adapters) kann sich beliebig ändern
- Hexagonale Architektur

Kopplung an Technik entfernen

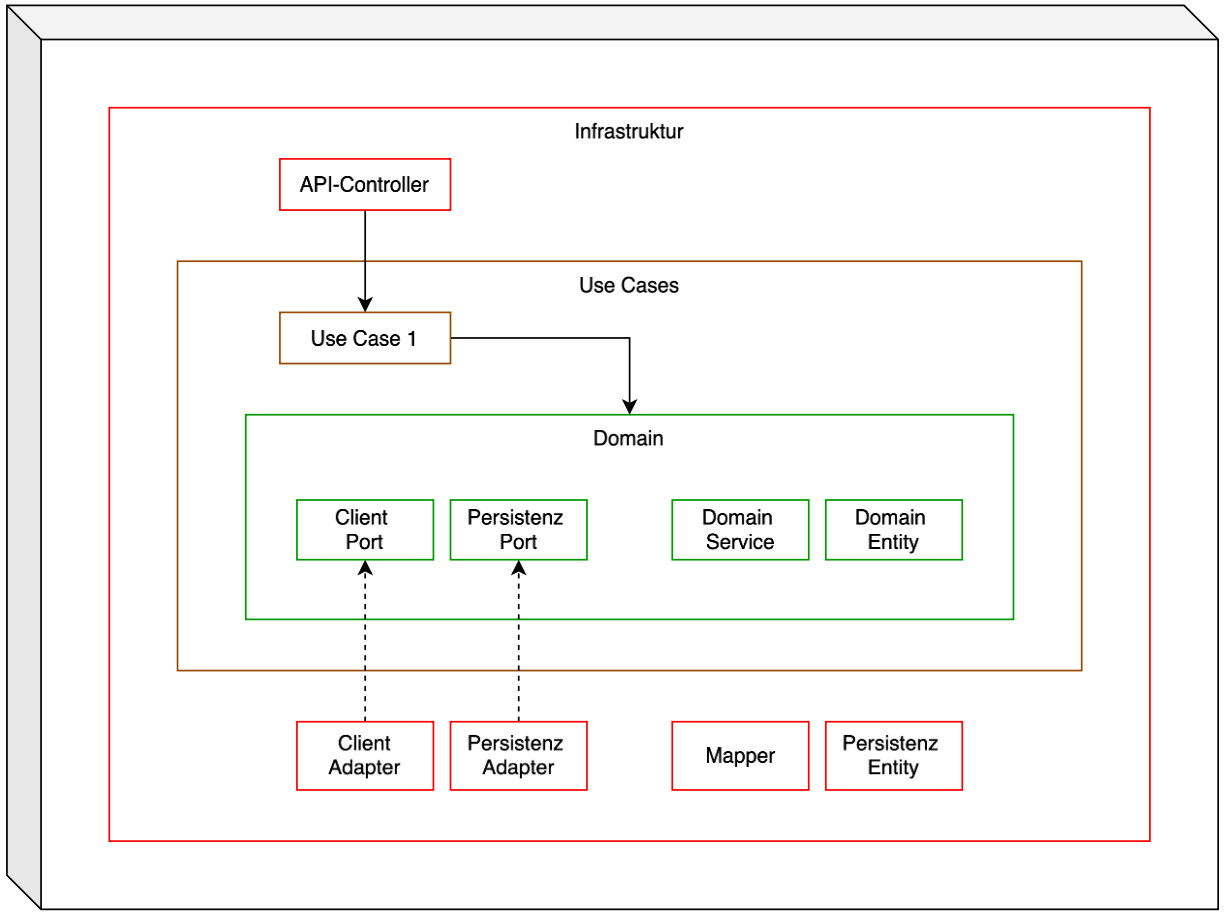
Persistenz auftrennen



- Die Entities hängen noch von der Infrastruktur ab (zb JPA)
- Entities & Repositories auftrennen
- Mapping notwendig!
- Code wird aufgebläht (Dopplungen)

Kopplung an Technik entfernen

Konsequenz: Projektstruktur ändern



- Packages umbenennen, Klassen verschieben
- Ringe statt Schichten
- es ergibt sich ein Onion/Clean-Aufbau samt Hexagonaler Architektur

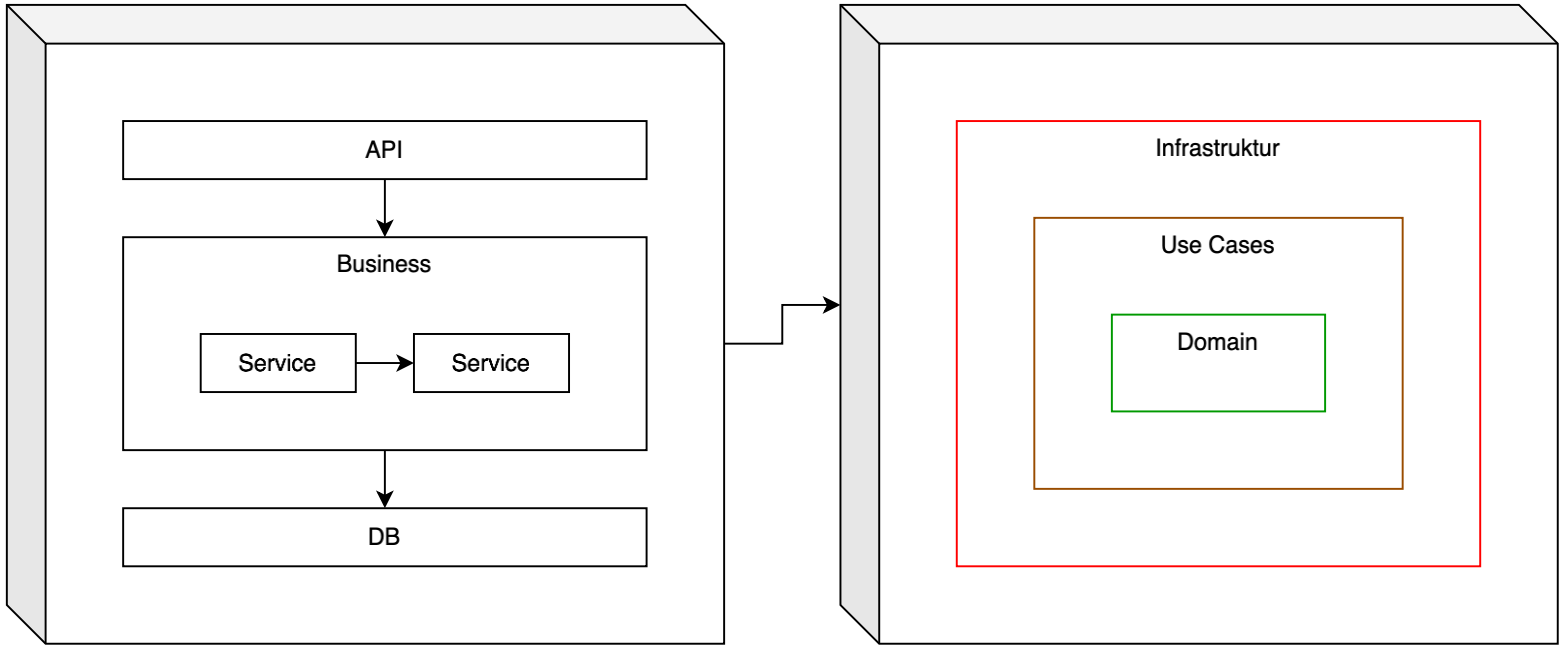
Fazit

CodeBuzz 2026

codecamp^N
RE•INVENT

Fazit

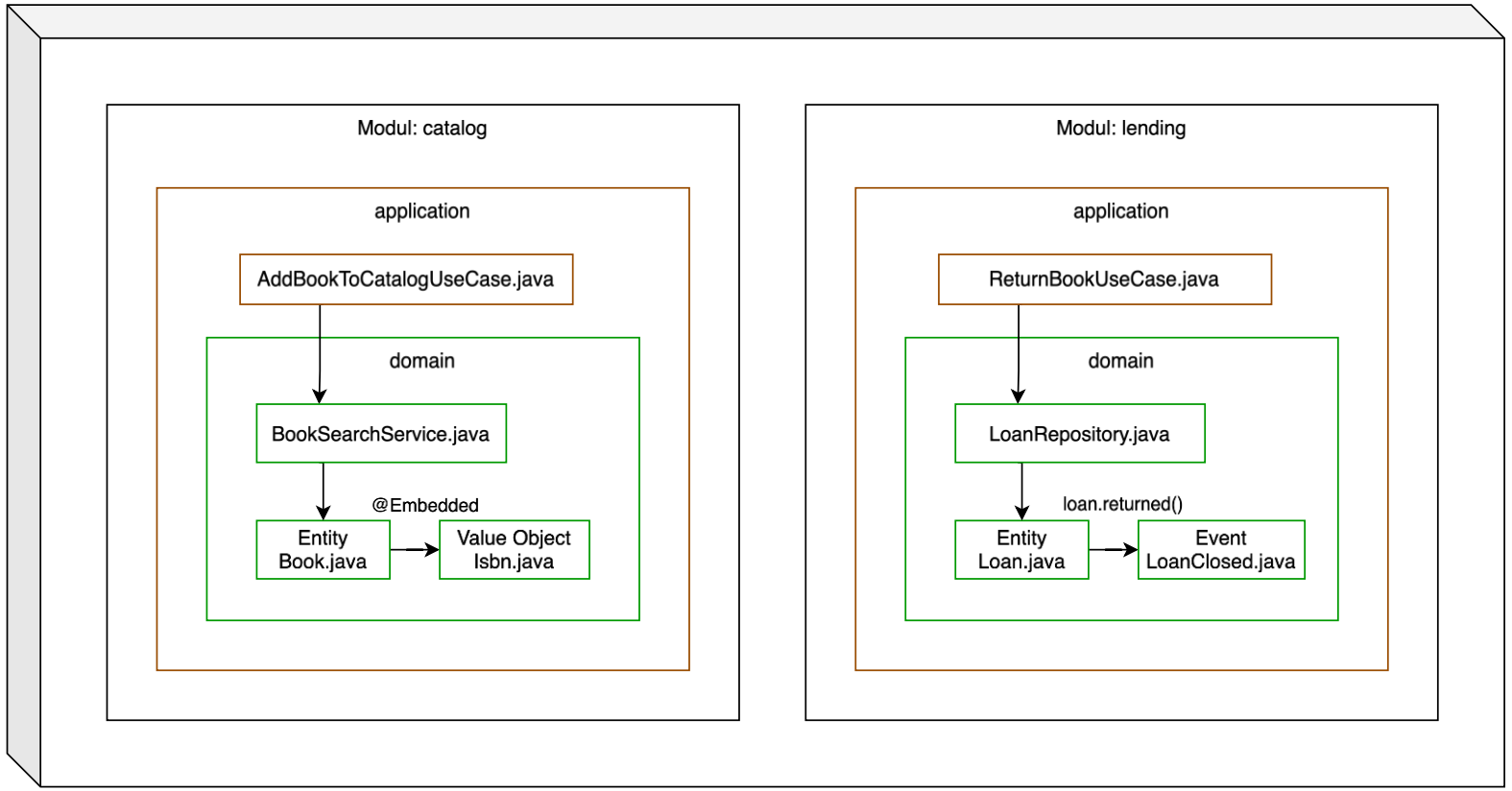
Evolution des Backend-Services



- kein Big-Bang notwendig
- Fachlichkeit in Code sichtbar gemacht
- Architektur leicht angepasst
- Vorteil: besser wartbar für Entwickler, langfristig stabil

Fazit

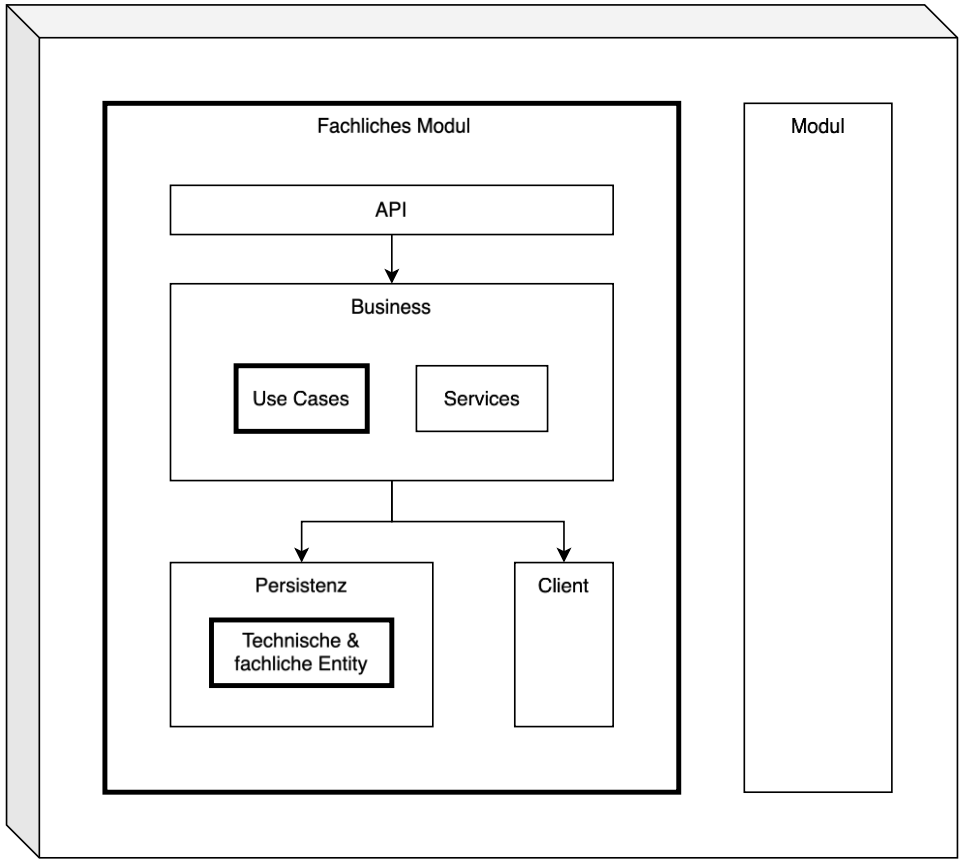
Beispiel: DDD + Clean/Onion-Arc + Java/Spring-Boot



- https://github.com/maciejwalkowiak/implementing-ddd-with-spring-talk
- https://github.com/mattiaciriolo WS/spring-io-conf-25
- Robin: https://github.com/RobinSchneiderCOC/spring-ddd-example

Fazit

Empfehlung: einfach starten



- Module einführen
- Layer beibehalten
- Use Cases herausziehen
- Kopplung an Spring Boot ist in Ordnung, Entity nicht auftrennen, aber mit Logik
- Fachliche Namen für Klassen und Methoden

www.codecamp-n.com