

the**code**campus</>

Client-Architektur in Angular



Philipp Burgmer

<> Entwickler, Trainer, Speaker
Web-Technologien
TypeScript, Angular </>

W11K GmbH - The Web Engineers

<> Gegründet 2000
Auftrags-Entwicklung / Consulting
Web / Java
Esslingen / Siegburg </>

theCodeCampus

<> Technologie Schulungen seit 2007
Projekt-Kickoffs & Unterstützung im Projekt </>

- <> Architektur
- <> Services & DI
- <> Komponenten
- <> Module & Routing
- <> Datenfluss
- <> Modularisierung
- <> Testen

Architektur

Was ist Architektur?

- <> Welche Komponenten gibt es
- <> Wie spielen diese Zusammen (Relationship)
- <> Wie grenzen sie sich von einander ab (Separation of Concern)

Wie lässt sich Architektur bewerten?

<> Nicht-Funktionale-Anforderungen

- Wartbarkeit / Testbarkeit
- Erweiterbarkeit / Flexibilität
- Wiederverwendbarkeit
- Performance / Skalierbarkeit
- Verständlichkeit / Nachvollziehbarkeit

<> Widersprechen sich teilweise

Architektur in Angular

- <> Style Guide mit Best Practises
- <> Empfehlungen, keine Verpflichtung
- <> Beschreibt Architektur-Bausteine

Services & DI

<> Kapselt irgendwelche Funktionalität

- Sollte UI nicht direkt verändern
- Kombination Service und Komponente / Direktive
- Ausnahme: Title Service

<> Richtige Größe wichtig

- Schwierig zu Schneiden
- Single-Responsibility-Prinzip
- Indiz: Testbarkeit

Dependency Injection

- <> Ermöglicht lose Kopplung
- <> Abhängigkeiten sichtbar machen (Konstruktor-Parameter)
- <> Nicht an konkrete Implementierung binden → *Flexibilität*
- <> Nicht an Umgebung binden → *Wiederverwendbarkeit*
- <> Abhängigkeiten mockbar → *Testbarkeit*

Komponenten

<> Klein halten

- Single Responsibility
- Wiederverwendbarkeit, Testbarkeit

<> Kapselung einhalten

- Verändern nur eigenen DOM
- Component → Service → Component

<> Sind für UI zuständig

- Single-Responsibility
- Datenbeschaffung und Haltung in Services

Demo

<> a.k.a Smart & Dumb / Smart & Presentation

<> **Presentation**

- Kennt Umgebung nicht
- Soll Daten darstellen
- Bekommt Daten per Input übergeben
- Aggregiert und abstrahiert Events per Output

<> **Context**

- Kennt Umgebung
- Besorgt Daten
- Bindet Presentation-Components ein
- Interpretiert Events

Demo

Module & Routing

<> Steuern welcher Code eingebunden wird

- Kein toter Code

<> Kapselung

- Keine globalen Variablen
- Abhängigkeiten erkennbar

<> Ebene über EcmaScript-Modulen

- Steuern was das Framework kennt und auflösen kann

<> Feature-Module

- Anhaltspunkt: Top-Level-Menü-Einträge
- Beinhalten Komponenten und Services

<> Core

- Beinhaltet allgemeine, häufig verwendete Sachen
- Auch Abhängigkeiten zwischen Features erlaubt (z.B. Dashboard)

<> Oft feinere Aufteilung

- Für Build-Optimierung

- <> Heimliches Herzstück von Angular Anwendungen
- <> Gibt mit Modulen zusammen Anwendung eine Struktur
 - Routen in Feature-Modulen definieren, in App aggregieren
 - Kontext-Komponenten im Routing verwenden

Demo

Zustand & Datenfluss

- <> Wer ändert wann welche Daten
- <> Wer wird wie über Änderungen informiert
- <> Großes Problem in AngularJS Anwendungen
- <> Angular ebenfalls keine Vorgaben

Demo

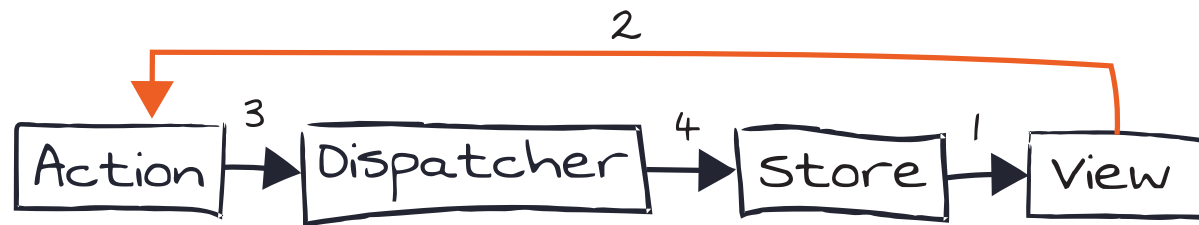
- <> Shared Mutable State
- <> Zwei-Wege-Data-Binding
- <> Daten werden einfach geändert
- <> Wer Binding hat wird aktualisiert
- <> Führt zu viel Polling

<> Immutable Data Structures

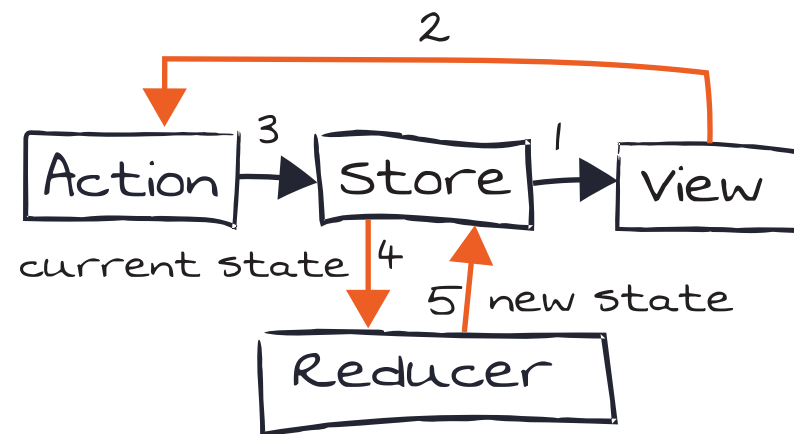
- z.B. über Object.freeze und Bibliothek wie [Immer](#)
- Nur bestimmter Code darf Daten verändern

<> Observables

- In Angular über RxJS schon integriert
- Subscriber bekunden Interesse
- Aktualisierungen können explizit bekannt gemacht werden

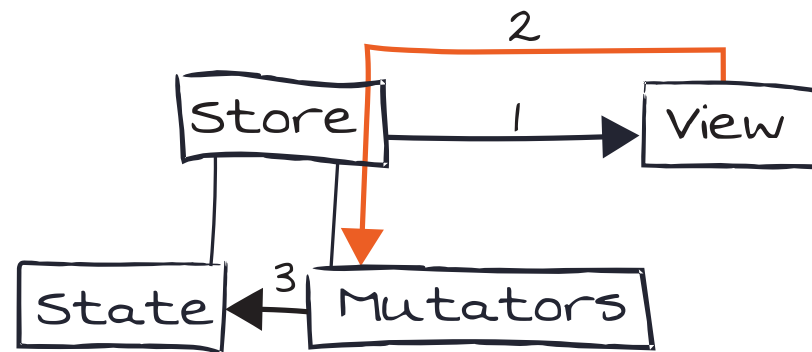


- <> Grundgedanke: Unidirectional Data Flow
- <> Flux ist ein Pattern
 - Verschiedene Implementierungen



<> Redux implementiert Flux

- Etwas andere Namen
- Gleicher Grundgedanke



<> Inspiriert von Flux & Redux

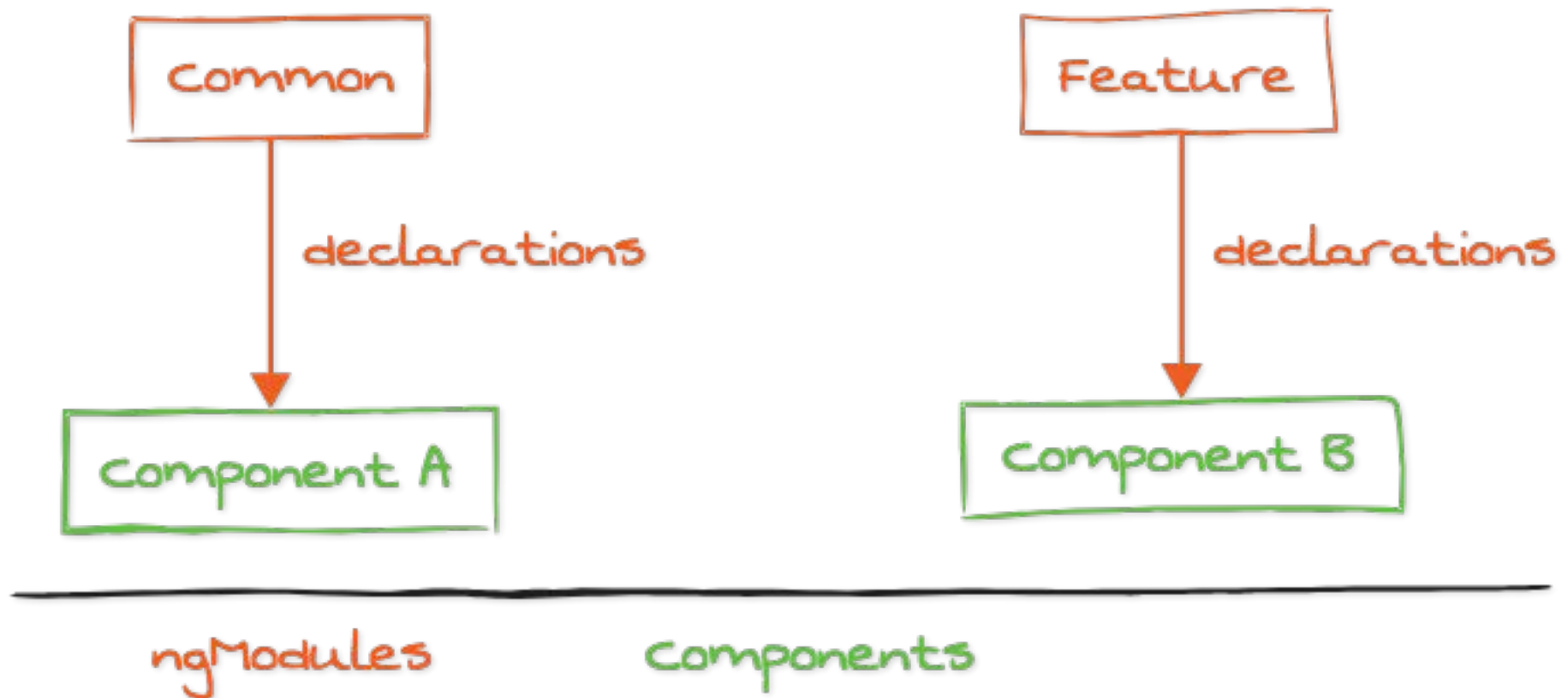
<> Versucht die Probleme von Redux zu lösen

- Alles komplett typisiert (TypeScript)
- Keine Zuordnung über Strings
- Erzwingt Unveränderbarkeit der Daten außerhalb von Mutators

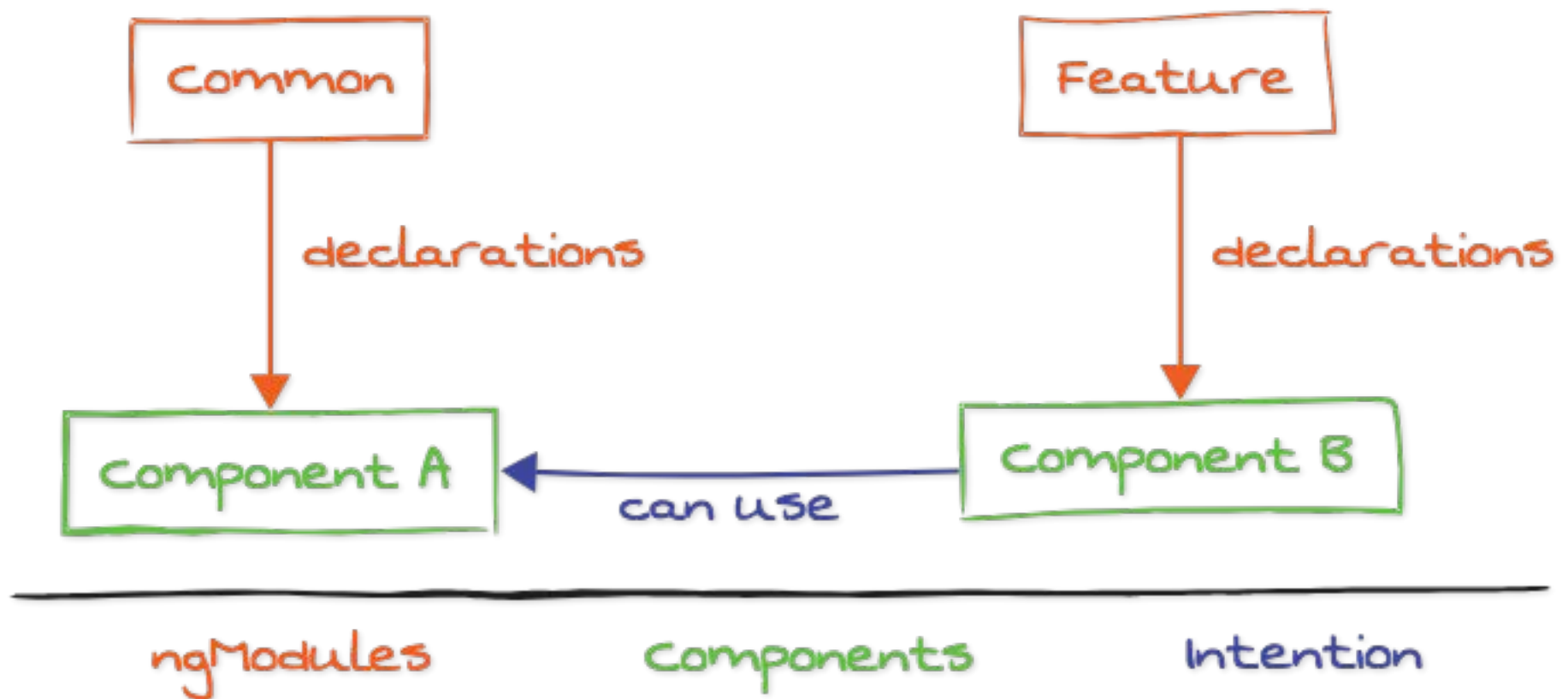
Demo

Angular Module & Kapselung

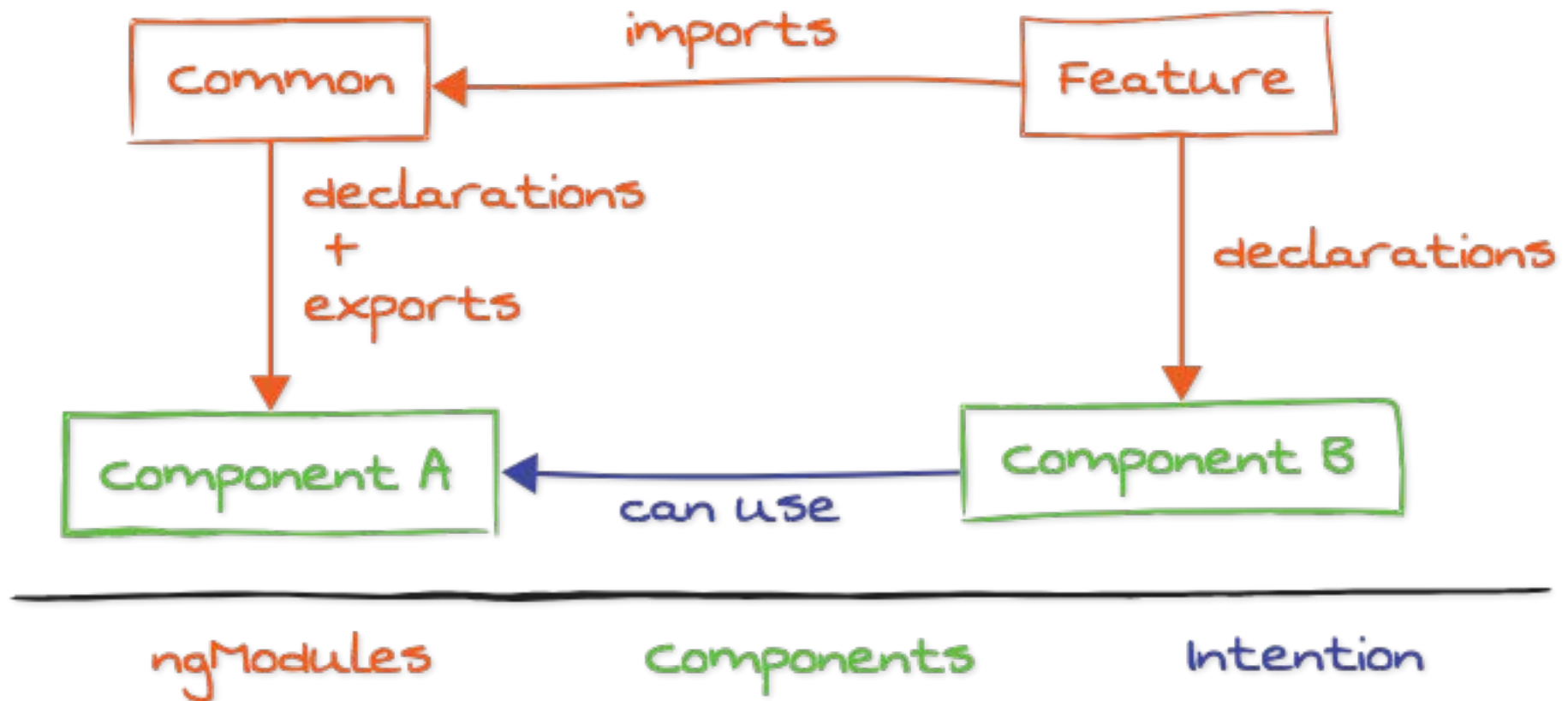
Angular Module & Komponenten



Angular Module & Komponenten



Angular Module & Komponenten



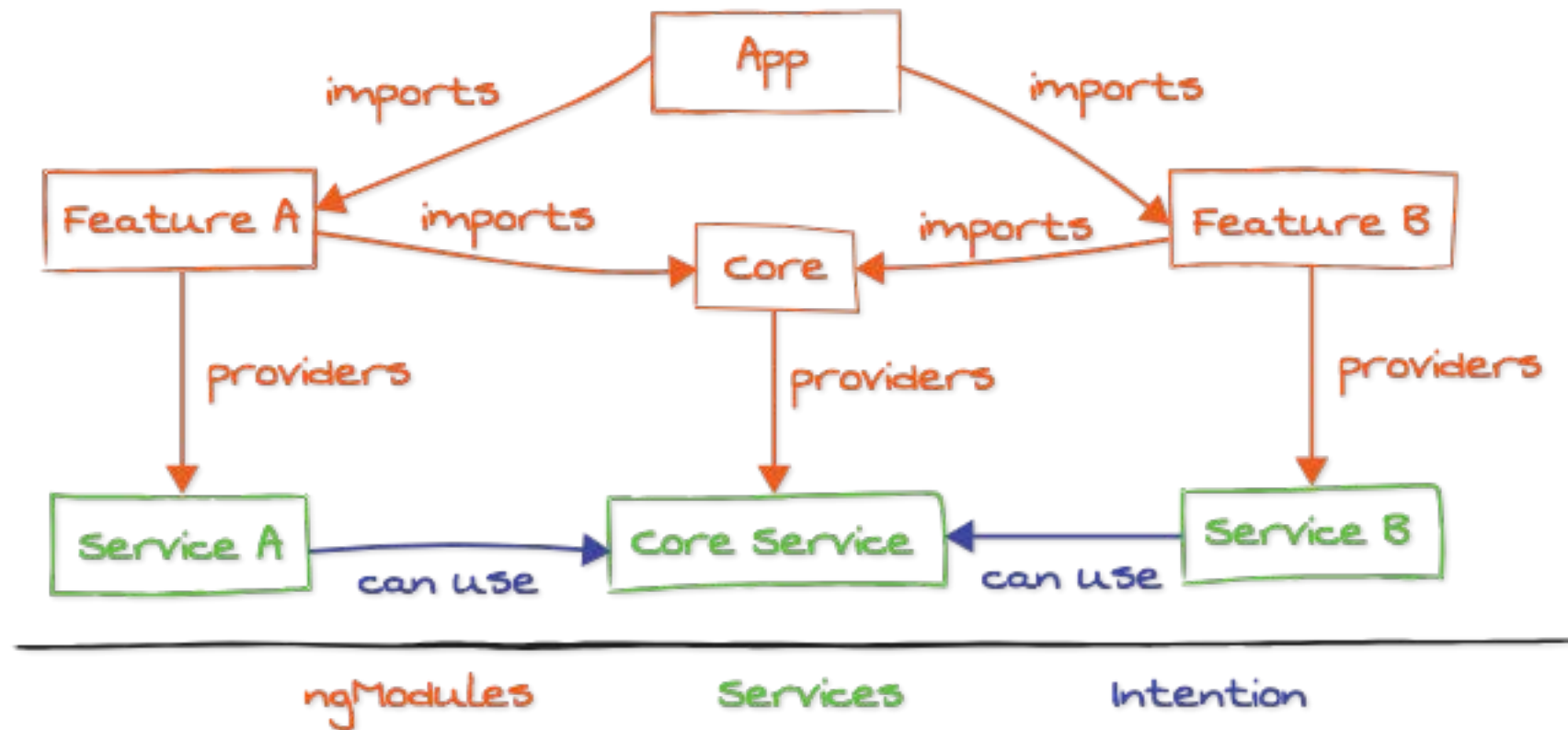
<> Positiv

- Vermeidung Namenskolisionen

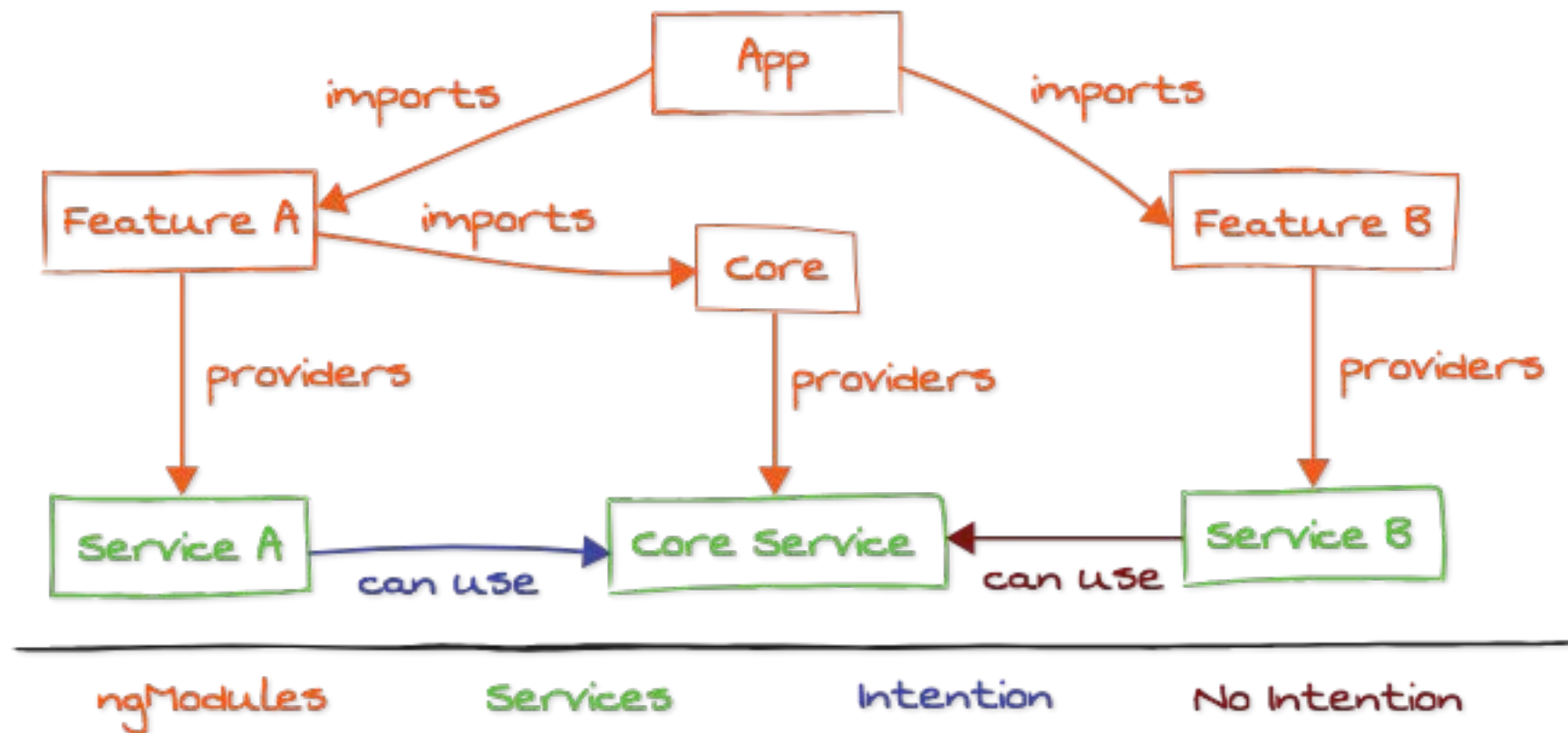
<> Negativ

- Schwierig zu lernen
- Schlecht nachvollziehbar
- Im Code nicht gleich ersichtlich

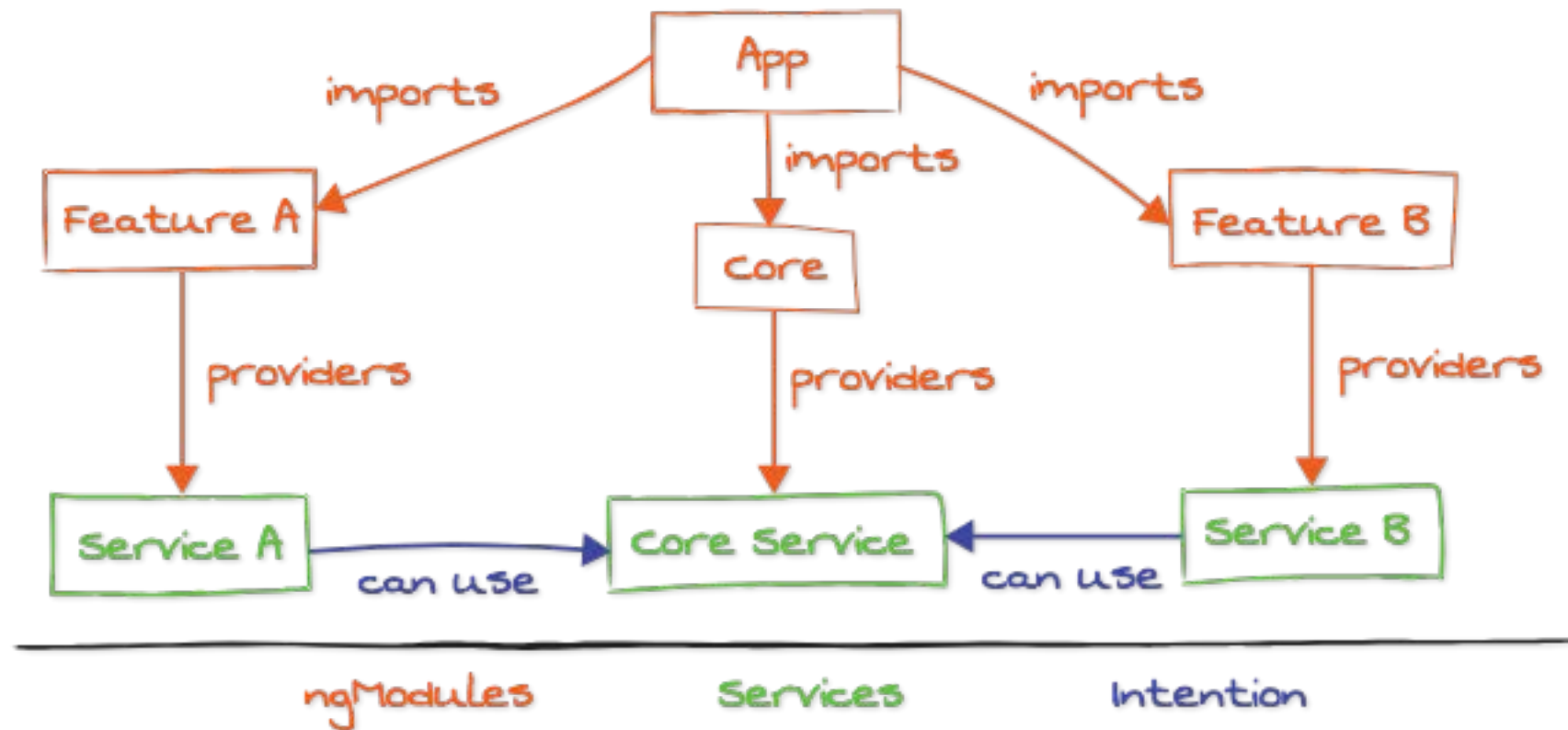
Angular Module & Provider



Angular Module & Provider



Angular Module & Provider



<> Negativ

- Abhängigkeit von Feature zu Core nicht ersichtlich
- Fehler erst zur Laufzeit

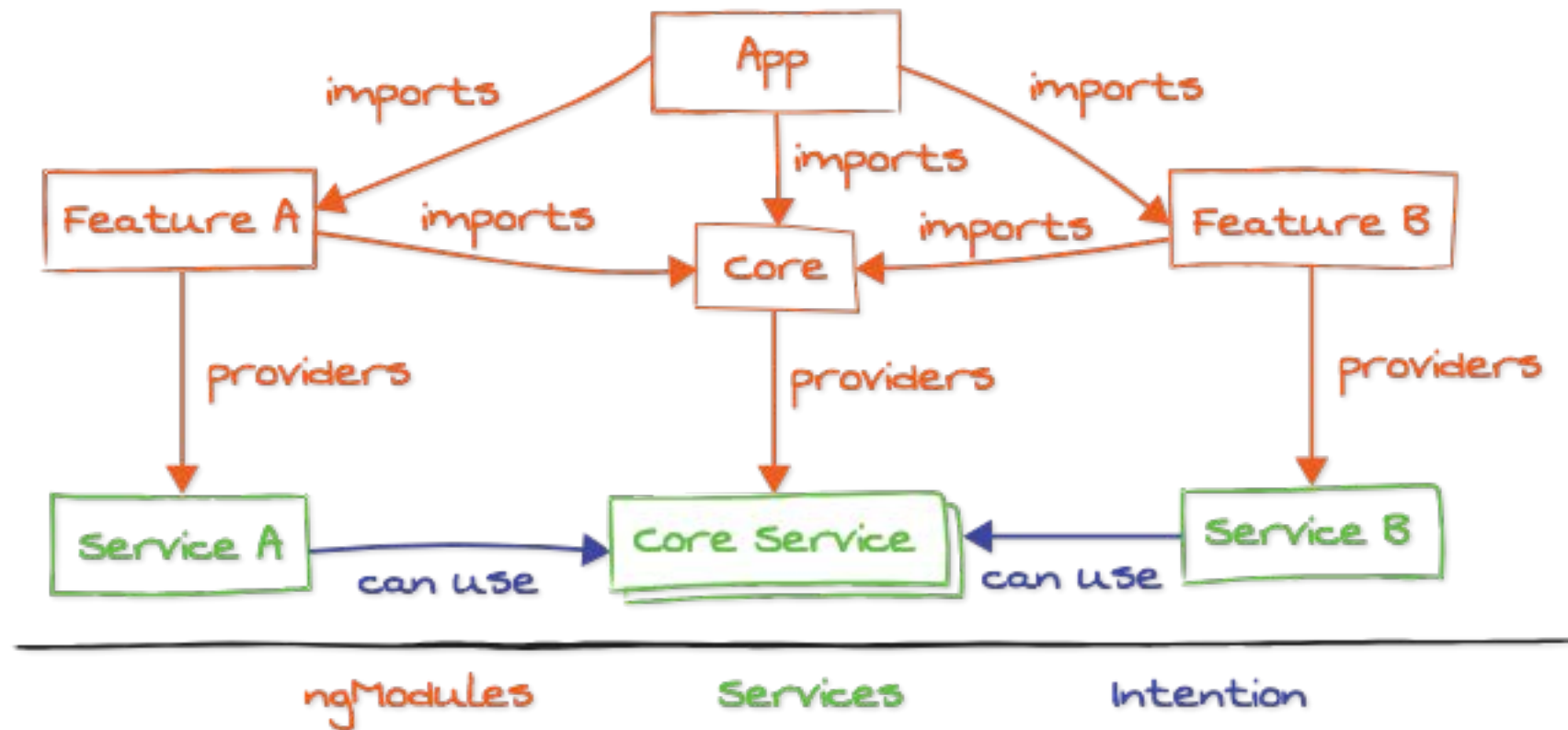
<> Positiv

- Ist in anderen DI Umgebungen auch so
- Eigentlich auch richtig

<> Problem: Unterschiedliches Verhalten für Provider und Komponenten

- Für Zugriff auf Komponenten muss Core in Feature importiert werden

Angular Module & Provider



Modularisierung über Packages

- <> Angular Anwendung Monolith zur Laufzeit
- <> Wiederverwendbare / allgemeine Teile auslagern (Bibliothek)
 - Eigene NPM Packages
 - Private Repos möglich (Nexus, Artifactory, NPM Inc)
- <> Kombination aller Paket- & Modulsysteme
 - NPM Paket installieren
 - EcmaScript Module per absolutem import Statement
 - NgModule zu imports hinzufügen
- <> Re-Exports um interne Strukturen zu verbergen

Demo

- <> Problem 1: Kein Schutz vor tiefem Zugriff
- <> Problem 2: Angular CLI bisher nur für Anwendungen
 - Alternative Builds wie [ng-packagr](#) oder [nrwl/nx](#)
- <> Problem 3: Entwicklung mit aktuellem Stand
 - `npm link`
 - Separate App für Package-Entwicklung



Testen

- <> **Unit-Tests wichtig für langfristige Wartbarkeit**
 - Tests oft guter Indikator für saubere Architektur
 - Lesbarkeit & Verständlichkeit der Tests
 - Tests sind im Idealfall ausführbare Doku
- <> **Test-Driven-Development hilft beim Schneiden**
 - Zusätzlicher Context
 - Single-Responsibility
 - Schlecht testbar? Indiz für *macht zuviel*

<> Dependency Injection und Mock-Support

<> [Angular Testing Guide](#)

<> Test-Code leider sehr aufgebläht

- `it('desc', async(inject([Dependency, (dep) => {}])))`
- Viel Schachtelung, viele Klammern
- Keine Verwendung der Typen bei `inject`

Lazy-Loading

<> Wichtig für Skalierung

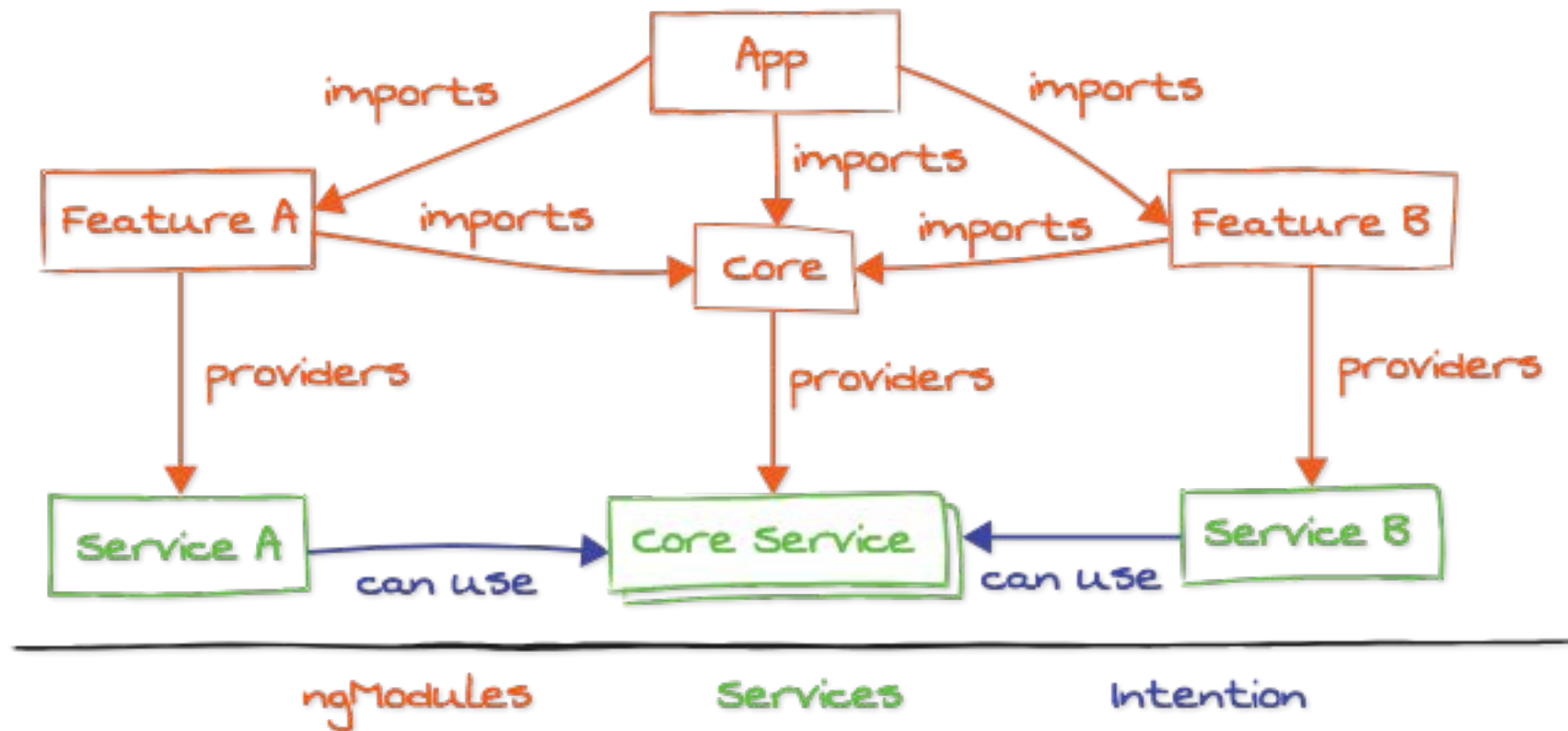
- Code on-demand nachladen
- Initialen Payload klein halten
- Schneller Start bei großen Anwendungen

<> Führt nicht gerade zu mehr Übersichtlichkeit

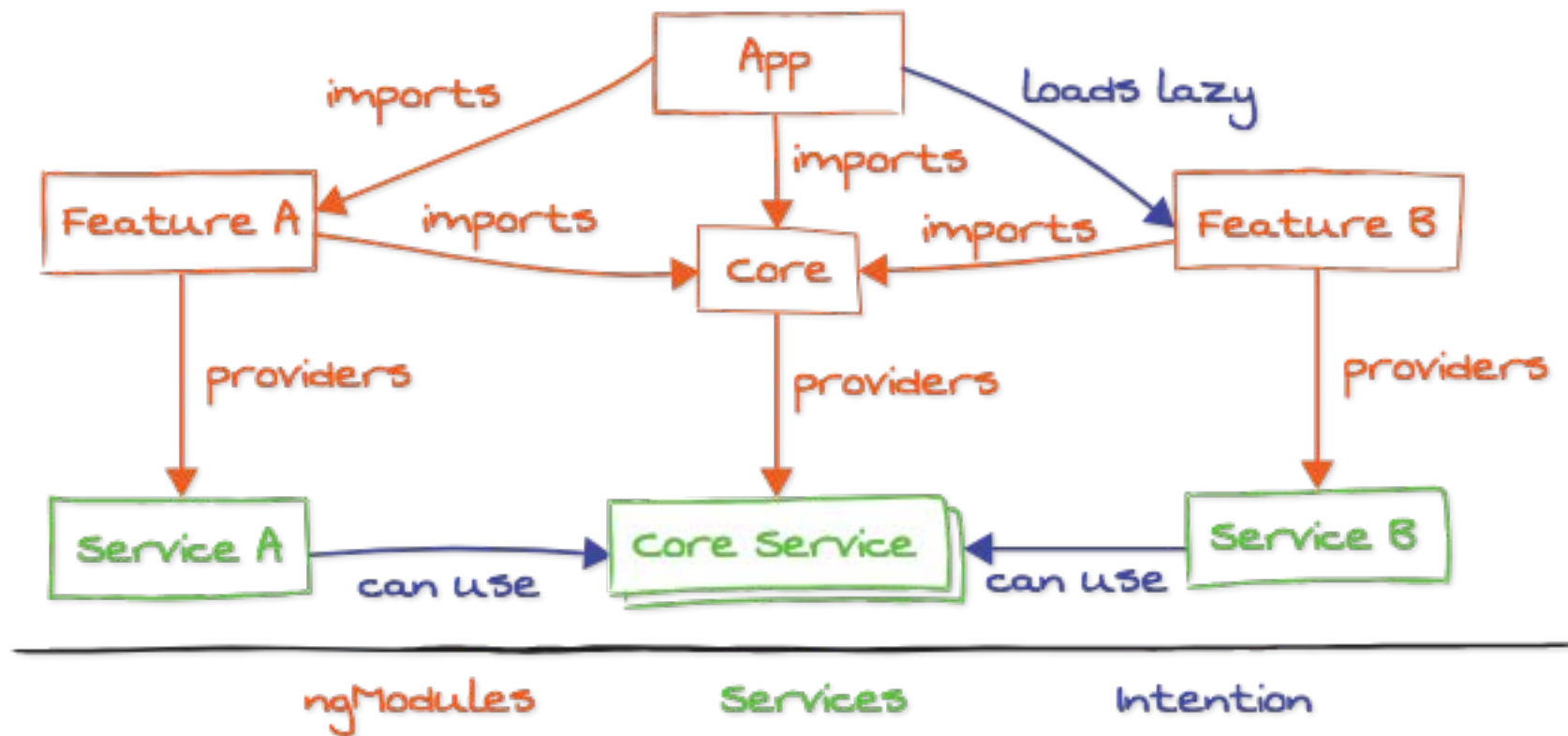
- Provider an Modulen
- "Referenzierung" von Modulen in Strings statt über import Statements

<> Häufiger Fehler: Alles lazy laden

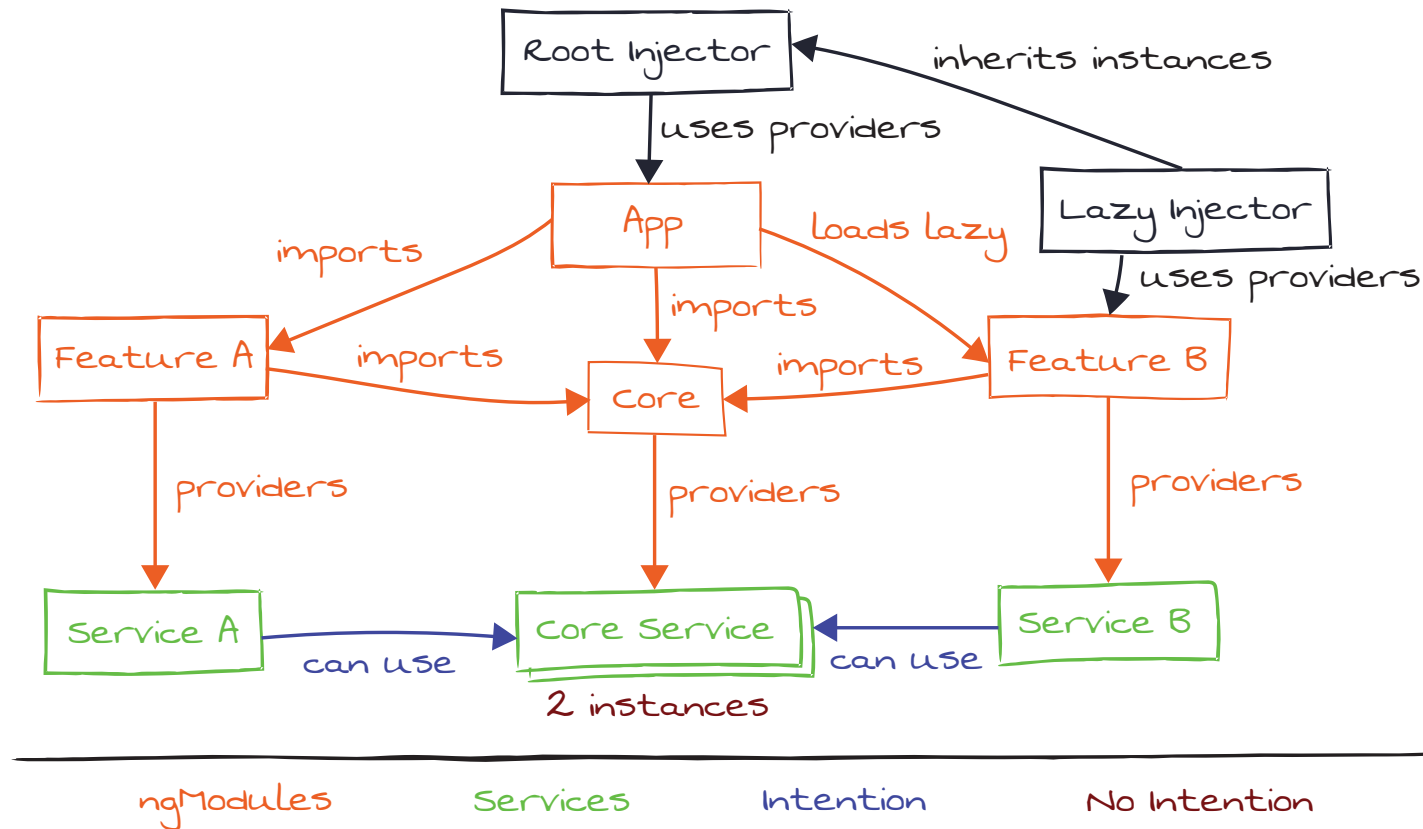
Lazy Loading



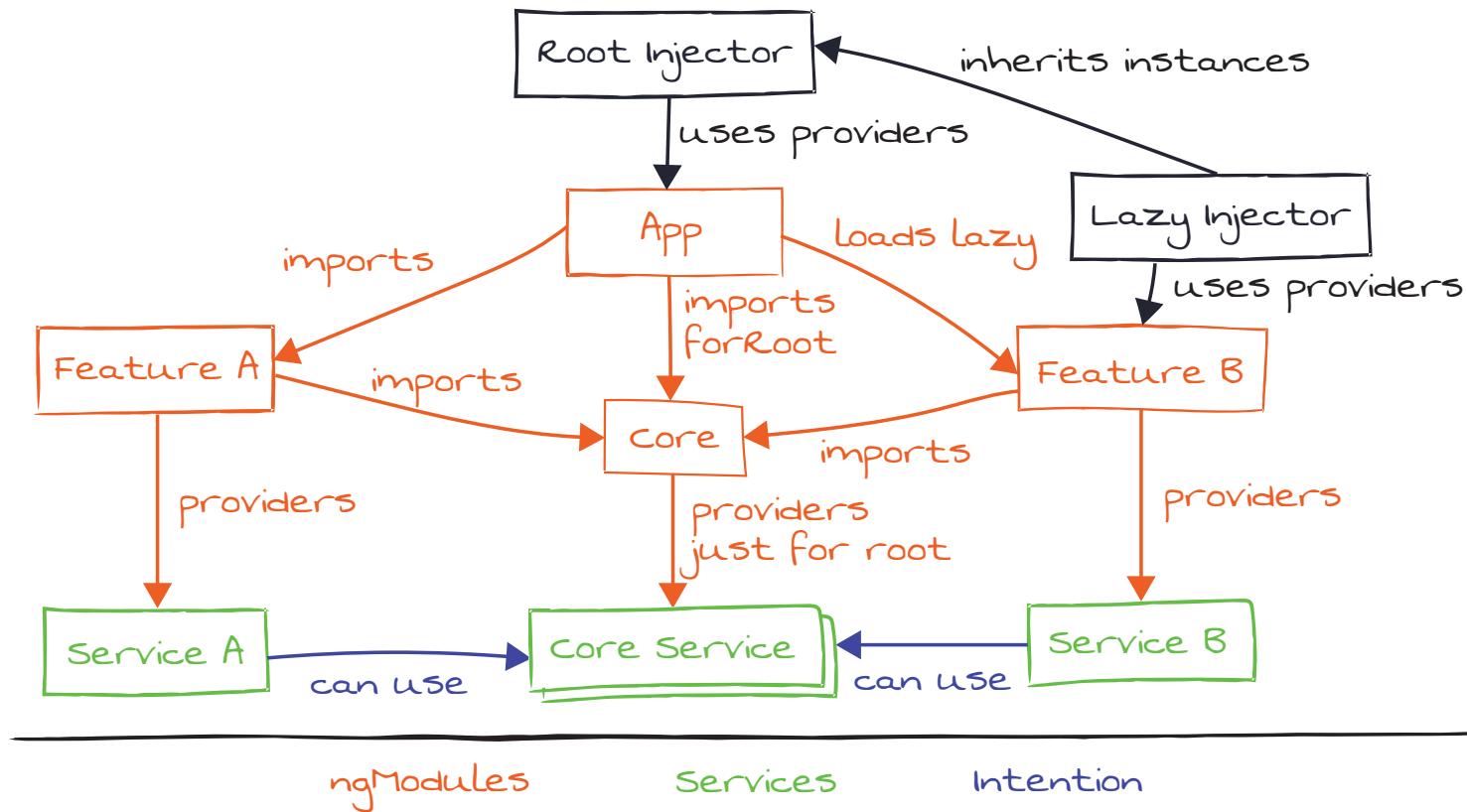
Lazy Loading



Lazy Loading



Lazy Loading



- <> Angular bietet viele Konzepte
- <> Konzepte alleine machen keine gute Architektur
- <> Saubere Architektur möglich
- <> Gefahren lauern im Detail
- <> Über Style-Guide viele Empfehlungen vorhanden

Philipp Burgmer
burgmer@w11k.de
Twitter: @philippburgmer
GitHub: pburgmer

www.thecodecampus.de
@theCodeCampus