# Taking back Software Engineering – Craftsmanship is not enough

Dave Farley

Definition “Software Engineering” nach Farley

“Engineering is the application of an empirical, scientific approach to finding the efficient solution to practical problems.”

## Fundamentals of an Engineering Approach

* Iterative
  + Verfeinerung – nach und nach mehr Detail (Visualisierung: nach und nach wird eine Ebene im Bild ergänz – Striche, ausmalen, 2. Ebene malen, …)
* Employs Feedback
* Incremental
  + Teil für Teil – nach und nach werden Features ergänzt (Visualisierung: Puzzleartig wird das Bild erweitert)
* Experimental
* Empirical

## Scientific Approach

Theory 🡪 Prediction 🡪 Experiment 🡪 Observation 🡪 Theory (Kreislauf)

# Java APIs – the missing Manual

Hendrik Ebbers

## Null Handling

NPE: häufigste Exception (gemäß Untersuchungen)

* Objects.requireNonNull(…)
* Default values setzen, falls null vorhanden
* Optional
  + In Methoden
  + Als Rückgabe
  + Niemals als Parameter nutzen (Optional selbst könnte ja null sein)

## Exception Handling in Streams

* Library VAVR
* Result.of(…): Klasse, die das mapping speichert, falls kein Fehler vorhanden ist, und den Fehler sonst
* Siehe Folien, S. 197/198

## Annotations

Fuck, falscher Vortrag

Folien: <https://dev.karakun.com>

# Warum funktioniert Continuous Delivery nicht?

Eberhard Wolff

<https://ddd-referenz.de/>

<https://puppet.com/resources/whitepaper/state-of-devops-report>

## High frequency deployments

🡪 multiple times a day

* Better lead time
* Better time to restore service
* Lower change failure rate
* Less unplanned work and rework
* Less work on remediation security issues
* Less work on defects identified by end users
* Less customer support work

## Experiments

* Working software should be the foundation for experiments
* What works, what not?
* Metric-based feedback about software
* Conversion funnel
* A/B testing

## Zusammenfassung

* Feedbackzyklen in allen Ebenen
* Continuous Delivery aus Eigeninteresse (verbessert meine Arbeit, selbst wenn time-to-market egal ist)
* Kultur muss sich ändern
* Schrittweise ändern

# Event Streaming and the Future of Applications

Ben Stopford

<https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/>

## Kafka: Messaging

* Scalability
* Event Storage
* Stream Processing

## Notes

* Event stream orders the events
* Stream processor: react, combine, filter and summarize event streams
* Paradigm change in a) storage and b) application communication

1. Storing event streams leads to
   * Customer journey
   * Replayability (e.g. recovering data after programmatic error)
   * Possibility to analyze (trends & behavior)
2. Communication
   * Loose coupling
   * Each app (microservice) fetches only data it is interested in

* Join, filter, transform, enrich (from tables), summarize

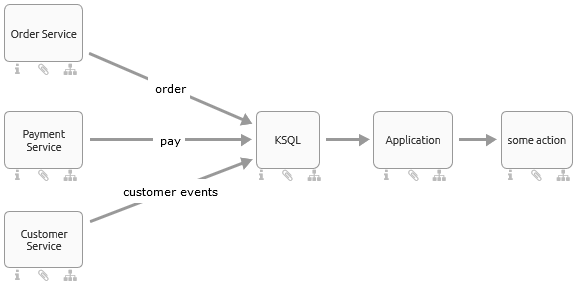
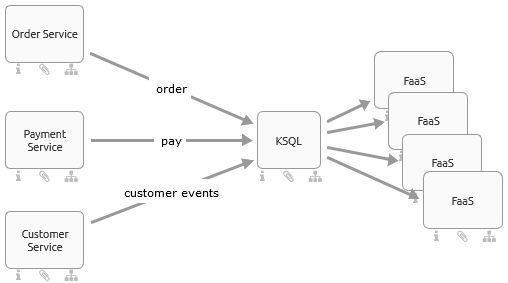


Figure 1: Event streaming

## Serverless

* Auto scaling
* Pay for execution time, not resources
* Event backlog drives autoscaling
* Event driven, but **not** streaming
* Remedy: see Figure 1: Event streaming
  + Each Application (behind KSQL) can be replicated using FaaS



## Summary

* Events decouple architecture
* Events underpin the storage model of truthful/factful systems
* Serverless will likely be event-driven (auto-scaling, pay for use, pluggability)
* KSQL is a kind of “Streaming Database” designed for these aspects of the event-driven world

# Event sourcing

Arno Haase

Nachteil klassischer Architektu (CRUD): Koordination in Db verlegt (Transaktionen, Locks)

## Event sourcing

* Storing events, not state
* Mehrwehrt
  + Metadaten / Kontextinfos (was ist passiert)
  + Reihenfolge & (optional) Zeitpunkt der Änderung
* Replay möglich / time travel
* Zustand kann gecacht werden
* Journal is append-only (Problem: DSGVO!)
* Problem: Parallelisierung von Änderung des **gleichen** Datums nicht möglich

*Bemerkung: Event Storming (für Anforderungs-Analyse) beruht auf den gleichen Events.*

## Commands

* Command
* Validate
* Side effects
* Create event
* Store event
* Apply event

## Resultat

* Kein Transaktions-Kontext
* Idempotenz über id: falls id bereits verarbeitet wurde, keine erneute Verarbeitung
* Scalability: Journal pro Kunde oder Bestellung
* Data centric design**:** kein stateless service
* Entities z.B. gemäß DDD schneiden
* Nachrichten/Commands gehen nur an root Entities
* Abkehr von klassischer kubernetes Architektur: PODs nicht gleichberechtigt
  + Da POD = JVM und Entität in 1 JVM leben muss, muss jeder POD seine eigene Entität verwalten

## Storage: Anforderungen

* Eventual consistency
* Lookup by key, no queries
* No schema: application data = black box
* Cassandra ist exzellent geeignet; prinzipiell geht jedes RDBMS / NoSQL / …
* Wichtig: viel Speicher wird verbraucht 🡪 Lösung suchen, die Speicher günstig anbietet

## Weiteres

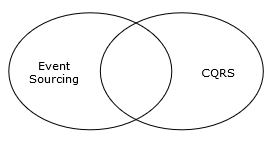
* Hilfreich: alle x Events einen Snapshot speichern
* No schema 🡪 no data migration
* Alte Events löschen: z.B. Snapshot quer über alle Journals 🡪 ab da muss nicht mehr auf den alten Stand zugegriffen werden

## Libraries

* Spring Axon & Spring Eventuate (statische Knoten-Konfiguration)
* Akka (dynamic resharding – scalability mit Kubernetes und PODs!)

## Konsequenzen

* Audit trail
* Rohdaten-Ablage 🡪 Nachvollziehbarkeit, leichte Fehlersuche
* Aus Einzeljournal ergibt sich (wenn nötig) trivial ein Event-Stream 🡪 Vervielfachung / Replikation
* Durch streng sequentielle Verarbeitung keine Race Conditions
* Keine Queries möglich – Lösung dafür: Query View
* Man kann pro Aggregate entscheiden, ob Event Sourcing verwendet wird oder klassisches RDBMS – z.B. Ebvent Sourcing für Order, RDBMS für master data



## Further reading

* Introduction to Event Sourcing: <https://msdn.microsoft.com/en-us/library/jj591559.aspx>
* Events as First Class Citizens (Randy Shoup): <https://hackernoon.com/events-as-first-class-citizens-8633e8479493>
* Life beyond distributed transactions (Pat Helland): <https://www.ics.uci.edu/~cs223/papers/cidr07p15.pdf>

# Continuous Profiling in Production: What, Why and How

Richard Warburton

* Known Knowns
* Known Unknowns
* Unknown Unknowns

## Ambient/Passive/System Metrics

* Preconfigured numerical measure
* CPU time usage / page-load times
* Cheap and sometimes effective

## Logging

* Records arbitrary events emitted by the system being monitored
* Log4j/slf4j/logback/ELK
* Logs of GC events
* Often manual, aids system understanding, expensive

## Coarse grained instrumentation

* Measures time within some instrumented section of the code
* Time spent inside the controller layer of you web-app or performing SQL queries
* More detailed and actionable though expensive

## Production Profiling

* What methods use up CPU time?
* What lines of code allocate the most objects?
* Where are your CPU cache misses coming from?
* Automatic, can be cheap but often isn’t

## How to use continuous profilers

1. Extract relevant time period and apps/machines
2. Choose a type of profile: CPU time/wallclock time/memory
3. View results to tell you what the dominant consumer of a resource is
4. Fix bottleneck
5. Repeat

### Type of profile

* CPU time shows expensive computational hotspots and inefficient algorithms
* Wallclock time shows blocking operations

## Views/Graphs

* Tree view (problematic with Spring stack – deep)
* Flamegraphs (helpful)

## Type of profilers

* Instrumentation (e.g. JVisualVM)
  + high overhead, especially for small methods
* Sampling/statistical profilers
  + Makes use of safepoints (“stop the world”)
  + Problem: JIT compiler inlines code 🡪 sampling safepoints are manipulated
  + Until Java 11, all threads had to wait for the slowest thread reaching a safepoint
* Advanced statistical profiling in Java
  + Not used by current profilers
* Historical data
  + Allows for post-hoc incident analysis
  + Enables correlation with other data/metrics
  + Regression
* Java Perf mapper agent
* Paper from Google about google-wide profiling: <https://ai.google/research/pubs/pub36575>
* <https://jaxenter.de/java-flight-recorder-57537>

## Methodology

* Continuous
* Proactive not reactive
* Systematic not ad hoc

# How modern SQL databases come up with algorithms that you would have never dreamed of

Lukas Eder

* Google “10 SQL tricks” (Vortrag von Lukas Eder)
* Windowing functions haben SQL fundamental verändert
* SQL ist eine 4GL (<https://de.wikipedia.org/wiki/4GL>)
* Konzeptuell ist ein Java Stream das gleich wie SQL: <https://blog.jooq.org/2015/08/13/common-sql-clauses-and-their-equivalents-in-java-8-streams/>
* JOIN ist ein Operator von FROM und kein SQL-Schlüsselwort
* Beobachte Execution Plans!
  + Entscheidende Größe zu beobachten: Kardinalität (bzw. Anzahl der Zeilen)
* Accessing the index is only worth the cost if we filter out a relevant amount of data

## Logische Reihenfolge

### Beispielhaft

1. Load tables
2. Apply predicates
3. Collect groups
4. Aggregate
5. Project
6. Order

### Allgemein

1. FROM, JOIN, APPLY
2. WHERE
3. GROUP BY
4. Aggregations
5. HAVING
6. WINDOW
7. SELECT
8. DESTINCT
9. UNION, INTERSECT, EXCEPT
10. ORDER BY
11. OFFSET
12. LIMIT
13. FOR UPDATE

## Weiterführende Links

<http://bigocheatsheet.com/>

<http://annotatiomania.com/>

# 10 Gründe, warum wir gewisse APIs mögen und uns andere APIs nerven

Lukas Eder

* Wichtigster Konsument einer API ist der Entwickler
* Damit gilt: UX ist wichtig

## Usability – quality components

<https://www.nngroup.com/articles/usability-101-introduction-to-usability/>

* Learnability
* Efficiency
* Memorability
* Errors
* Satisfaction

## Null

* Null is bad when it is unexpected

## 10 Factors

1. Naming
   * ubiquitous language
   * get it right from the beginning
   * talk about it
   * Changing names: change docs and use HTTP 301 Redirect for deprecated docs (no dead links!); don’ t forget external doc (StackOverflow); write good release notes
   * Good example: LocalDate (JSR-310)
2. Simplicity
3. Do one thing, and do it well (UNIX Prinzip)
   * Modular
   * Composition
   * Simple
   * Small
   * Transparent
4. Types
   * Not about type safety
   * Types must be documented at least!
   * All 10 arguments for APIs also apply for types
5. Discoverability
   * Doc is available, but it is not necessary to read it
   * Small number of entry points, e.g. JPA EntityManager
6. Error Handling
   * Transparent
   * Robust
   * Fail in a way easy to diagnose
   * Value developer time over compute time
7. Consistency
   * Leads to efficiency and memorability 🡪 core of usability
   * Take either choice and stick to it
8. Convenience
   * Dogfooding: “eat your own dogfood” (use your own API)
   * E.g. autoclosable Resources from Java 7
9. Compatibility
   * Be downwards compatible
   * Keep old API tests around
10. Documentation
    * Explain how a thing works

# Preventing Null Pointer Exceptions at Compile Time

Werner Dietl

* Use @NonNull or @Nullable for method arguments
* Verification approach: pluggable type-checking with Checker Framework
  + javac -processor NullNessChecker MyClass.java
  + <https://checkerframework.org/>, <https://checkerframework.org/live>
  + IDE support
  + Opposite to Java’s default, makes @NonNull the default (less annotation, safer)
  + Open Source: <https://github.com/typetools/checker-framework>, monthly releases, 75 authors, active

## Summary

* Nullness Checker finds much more than FindBugs, Klint, PMD, Eclipse, IntelliJ.
* Few false positives.
* Annotation effort is fairly small.
* Verification is a different approach than Bug-finding
* More user burden than FindBugs

Using one or the other depends on circumstances.

## Tips

* Start by type-checking part for your code
* Only type-check properties that matter to you
* Use subclasses (not type qualifier) if possible
* Write the spec first (and think of it as a spec)
* Avoid warning suppressions when possible
* Avoid raw types such as List

# Timeless Design Principles

Jason Gorman

## Simple Design

* It works
* Clearly communicate intent
* Free of duplication (unless it makes it hard to understand)
* Build from simplest parts

## Modular Design Principles

S.O.L.I.D.

* Single Responsibility
* Encapsulation (Data Hiding: „Tell, don’t ask”)
* Polymorphism (Dependency Inversion, Dependency Injection)
* Interface Segregation

# Classloading in Action

Felix Becker

<https://github.com/fbe/aacl-talk>

## Classloader

* Load classes and resources from arbitrary sources
* Separate application from data source (platform independent)
* Java classes are being loaded as java byte code
* Resources can be of any type

## Default classloader (OpenJDK 11)

BootstrapClassLoader (core classes) 🡪 PlatformClassLoader (platform classes, e.g. driver) 🡪 AppClassLoader (module/class path)

Arbeitet nach dem Delegation Prinzip: wenn parent nicht laden kann, versuche ich es selbst

## Modules

* Since Java 9 each class is member of a module
* If the class is not being loaded from a module, it’s placed in an “automatic module”
* Classes in automatic modules are allowed to do anything like they did before Java 9
* Rules of the module system only apply to classes loaded from non auto modules

## Classloading vs. Shading

Um mehrere Versionen der gleichen Bibliothek zu laden, ist Classloading deutlich besser als Shading:

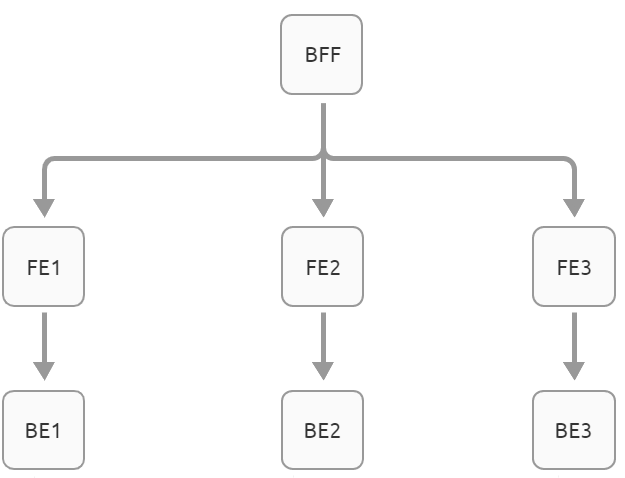
* Shading: Prefix, z.b. „shaded.” vor alle Namespaces; Bibliotheken müssen dafür umgeschrieben werden
* Shading kommt an technische Grenzen
* Wenn Classloader A eine Klasse lädt, kann Classloader 1B dieselbe Klasse erneut laden; sie sind im Sichtbarkeitsbereich des jeweiligen Classloaders. So können zwei verschiedene Versionen derselben Klasse über separate Classloader geladen werden.
* Spring verwendet den Weg das Classloading

<https://github.com/fbe/aacl-talk/>

# Frontend Architektur für Microservices

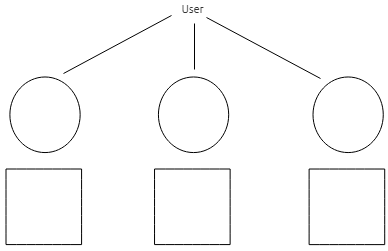
Thomas Kruse

1. BE, das FE zur Verteilung ausliefert (Backend for Frontend, BFF)



* + Wer verantwortlich?
  + Neuer Monolith!

1. Jedes FE verteilt



1. Self Contained Systems (SCS)
   * Integration über Navigation
     1. Jedes System liefert Navigation
     2. Infrastruktur liefert Navigation
   * Navigation

* Statisch eingebaut (build time)
* Dynamisch nachgeladen
  + Rücksprungziele, DeepLinks = API/Contract

## Technologien

* HTML5 Routing (Server side rewrite rules)
* Angular Modulsystem
* Angular Features: Crawlbar (SEO), i18n, Wiederverwendung
* Wartbarkeit dank TypeScript
* Gemeinsames Deployment als Option – oder Container

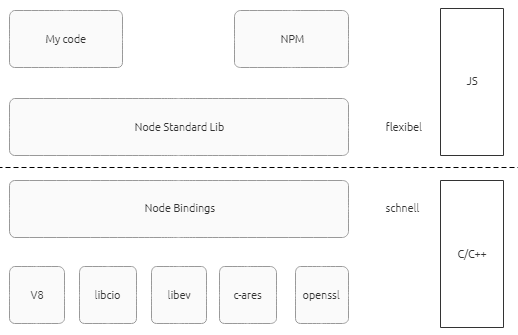
Schneiden: mit DDD

# Einführung in Node.js

Sebastian Springer

<https://github.com/sspringer82>

* Node kann Webseiten ausliefern: Webserver, Reverse-Proxy mgl. 🡪 aber sehr langsam



* Node Webserver ~ 50MB
* Single threaded, single process für eigenen Code

## Standard Lib

<https://nodejs.org/en/docs/>

Node portiert vom Modulsystem require (schlecht) zu import (gut)

* require: const {readFile}: require('fs');
* import: import {readFile} …

## Setup

1. npm init -y erstellt package.json
2. scripts in package.json ergänzen um gute defaults (build, …)
3. echo 'node\_modules' >> .gitignore

## package.json

* main: Einstiegspunkt
* scripts: aliases (Doku hat gute defaults: preinstall, install, postinstall)
* private: true (verhindert versehentliches veröffentlichen)

## Tools

* Verdaccio: npm proxy / lightweight private npm proxy registry (oder: artifactory)
* nvm: Node Version Manager – ermöglicht, mehrere node-Versionen parallel zu installieren (auch für Jenkins verfügbar)
* node: liest Code, wandelt ihn in Bytecode, führt diesen aus und optimiert ihn kontinuierlich 🡪 bei jeder Änderung muss Prozess neu gestartet werden
* nodemon: automatisiert file watching, rebuild & restarts (for dev only, not prod)
* Debugging: node --inspect-brk index.js
  + Chrome: inspect
  + Alternative: von IDE aus debuggen
* yarn: facebook Alternative zu npm
  + Beide sind gleich gut
  + Beide nutzten die gleichen Paketquellen
* Definitely typed: stellt für bekannte Libs Typen zur Verfügung (für Nutzung in TypeScript)
* Coding Standards
  + <https://github.com/airbnb/javascript> (Empfehlung Sebastian)
  + <https://google.github.io/styleguide/jsguide.html>
  + <https://github.com/standard/standard>
  + Linter prüfen Code Qualität
  + Prettier formatieren automatisch gemäß Code Style
* Linters

<https://github.com/douglascrockford>: „It will hurt your feelings!”

jslint <https://www.jslint.com>

jshint <https://jshint.com>

eslint tslint <https://eslint.org> <https://palantir.github.io/tslint>

* Promises ~ Futures
  + Async Flusskontrolle
  + Bessere Schachtelung
  + Mehr Callbacks 😉
* Async – async/await
  + Weiterentwicklung von Promises
  + Noch schlanker
  + Blockiert nicht
* ReactiveX (Microsoft)
  + Observer Pattern
  + Async mit Streams
* APIs
  + Web/Rest: restify, Express, swagger
  + GraphQL
* Tests
  + Mocha
  + Jasmine

## Paketqualität

* npm
  + #Downloads
  + Last publish
* GitHub
  + Commits
  + Stars

## Doc

* <https://jsdoc.app/> (formerly [http://usejsdoc.org](http://usejsdoc.org/))

## CI

* Mocha erzeugt xunit Reports 🡪 Einbindbar in Jenkins, Travis, …

## Multi-Threading

* 1. pm2 (per npm): pm2 start -I 4 index.js
  2. Docker

## Performance/Profiling/Memory Analysis

* Chrome Dev Tools
* Performance Timing API

## Security

* Filter Input, Escape Output
* Validator (<https://www.npmjs.com/package/validator>)
* Escape Db Queries
* NSP (node security project) – ist in npm integriert

# Angular Workshop

Manfred Steyer

## Resources

<https://tinyurl.com/y5mtzwu9>

<https://github.com/manfredsteyer/jax-2019-workshop>

VS Code: Plugin „Angular Essentials“ (John Papa)

Bootstrap: Library für Designs

NX – Nrwl Extensions

* <https://github.com/nrwl/nx>
* <https://nx.dev/>
* U.a. Visualisierung vom Dependency-Graph
* Eigene Linter-Regeln

Unit Tests: karma

## Building-Blocks und Hintergründe

### Erstinstallation

npm i -g @angular/cli

ng new app

cd app

ng serve (nur zur Entwicklung)

### Wissen

* Default Imports: BrowserModule, HttpClientModule, FormsModule
* Access modifier: default ist public; private kann davor geschrieben werden
* Callbacks: Notwendig, da Browser-Tab single threaded ist und sonst die Anwendung blockieren würde.
* Constructor Injection
* Architektur-Ziele
  + Performance
  + Komponenten
  + Vorhersagbarkeit

### Komponenten

Angular bildet Komponenten-Baum ab

Smart Components vs. Dumb Components

### Module

* Modul besteht aus Komponenten
* Komponenten müssen im Modul veröffentlicht werden (export), damit sie in anderen Modulen verwendet werden können
* Zukünftig sollen Module abgeschafft werden (optional gemacht werden)

### Services

* Sind Singletons

### Bindings

Bindings folgen den o.g. Architektur-Zielen.

* Property-Binding: [xxx]
  + Daten fließen von oben nach unten, nie umgekehrt (im Komponenten-Baum)
  + Beim Property-Update darf man kein Event auslösen, da sonst Zyklen erzeugt werden (führt zur Fehlermeldung von Angular)
* Event-Binding: (yyy)
  + One way, bottom 🡪 up
  + Kein Digest (Abgleich des Baums mit der UI) notwendig
* Two-Way-Binding: [(zzz)]
  + Kurzform für ein Property-Binding und ein Event-Binding
  + <input [(ngModel)]="from"> 🡪

<input [ngModel]="from" (ngModelChange)="from = $event">

* + Per Konvention bekommt das generierte Event-Binding immer Change zum Namen des Two-Way-Bindings hinzugefügt

## Routing

* Mapping von URLs (= Routen) auf Komponenten
* URL Matrix Parameter (uralte Konvention, wird durch JS wieder modern – ähnlich wie PUT/DELETE bei Rest)
* Guards
  + CanActivate
  + CanActivateChild
  + CanDeactivate<T>
  + CanLoad
* Routen können anwendungsweit oder pro Modul definiert werden
* Lazy Loading
  + Lazy Module nie importieren – Angular kümmert sich um das Nachladen
  + Preloading: evtl. später benötigte Module werden mit freien Ressourcen später geladen

## Strukturierung

* Das Oberste Modul ist das root-Modul.
* Faustregel: wenn ein Modul 7 ± 2 (aus der Psychologie) Imports verwendet, sollte es in Submodule (bzw. Subsubmodule, usw.) zerschlagen werden.
* Einige Bücher empfehlen ein Core-Modul, das macht aber aus Sicht von klaren Verantwortlichkeiten (vgl. Microservices) und seit Angular 6 keinen großen Sinn mehr.
* Bei Umstrukturierung auch die zugehörigen Routen in die neuen Submodule migrieren

### Mono-Repo

### Schichtentrennung

* Empfohlene Schichten: Feature / UI / Data / Util
* Feature (Smart Component): Darf auf UI, Data und Util zugreifen
* UI (Dumb Component): Darf nicht auf Data zugreifen

## Production Builds

* Bundling
* Minification
* enableProdMode()

### Approaches

* JIT (at runtime)
  + für debugging
  + beherrscht inkrementelles Kompilieren
* AOT (ahead of time)
  + kompiliert HTML Templates zu JS (kleiner und schneller für den Browser)
  + Compiler muss nicht mit ausgeliefert werden
  + Nicht benötigte Komponenten werden herausgenommen
  + ng build --prod