Einfach mal guten Code schreiben

Anon
ASM20

Workshop

- Mindset
- Werkzeuge
- Demos / Ausprobieren
- Diskussion

Meta

- Fragen am Ende jeder Slide
- Diskussionen am Ende
- Große Diskussionen ganz am Ende

Vorstellungsrunde

- Warum seid ihr hier?

Motivation

- Lesbarkeit für Andere
  - “[C]ode is read much more often than it is written”
  - In einem Jahr seid ihr selbst der Andere!
- Schnelleres Arbeiten
  - Weniger Konzentration für das Lesen des Codes, mehr für’s Verstehen
- Bugs keine Chance geben
  - Guter Code vermeidet ganze Fehlerklassen
- Nicht für euren Code schämen
  - Seid stolz auf

Was ist guter Code?

- Viele Aspekte
- Einige Teil dieses Workshops
  - Fokus auf Werkzeuge
- Andere nicht
  - Architektur
  - Variablennamen
  - Sprachspezifisches

Was ist guter Code?

- Umsetzung: Grundsätze
  - “Explicit is better than implicit” (PEP 20 - The Zen of Python)
- Oder: Satz an Regeln
  - Jede Regel will Bugs vermeiden, Lesbarkeit erhöhen, etc.
  - Konkret, also Sprach- bzw. Projektspezifisch
- Herangehensweise:
  - **Hab eine Meinung (be opinionated)**
    - Finde Grundsätze / Regeln
  - **Wende diese einheitlich an (be consistent)**
    - Nutze Werkzeuge

Consistency

- Einheitlicher Code = guter Code
  - Folgt den selben Regeln / Grundsätzen
- Arbeiten ohne Umschalten
- Streit vermeiden (Code Reviews, etc.)
- **Einheitlichen Code zu schreiben ist wichtiger als Recht zu haben**
- **Man kann nur dort einheitlich sein, wo man eine Meinung hat**

Coding Guidelines / Style Guides

- Satz an Regeln, die für ein Projekt gelten
- Beispiele
  - Linux kernel coding style
  - PEP 8
  - Angular Style Guide
- Bestehende Styles
  - Sparen Zeit und Nerven
  - Machen es einfach andere Projekte zu lesen
  - Haben viele Diskussionen schon ausgetragen
    - Black #118

Ausnahmen bestätigen die Regel

- Coding Guidelines haben ihre Grenzen
- Im Zweifelsfall Regeln brechen um Grundsätze einzuhalten
- ABER: Vorsicht beim Tooling!

Werkzeuge

- Guten Code erreicht man auch mit gutem Tooling
- Automatisierung
  - Idealerweise automatisch überprüfen / anwenden
- Beim Schreiben
- Beim Build
- Beim Committen
- Lehrt einen immer auf Regeln zu achten

Formatter

- Überprüft das Format des Codes
- Verändert nicht den AST
- Wird meistens auf alle Dateien / ganze Projekte angewendet

Linter

- Analysiert Code und bietet Verbesserungen an
- Formatierung, aber auch potentielle Bugs und verdächtigen Code
- Kategorisieren oft Regeln
  - Level: Warning, Error
  - Kontext: Best Practices, Possible Errors, Style
  - Verhalten: Default, Autofix

Übersicht

Sprache	Formatter	Linter
Go	Gofmt	Golint
Rust	rustfmt	rustc
Python	Black	Flake8, Pylint
JavaScript / TypeScript	Prettier	ESLint
Kotlin	ktlint	

Ältere Sprachen

- Haben oft keinen Standard
- Tooling ist IDE-zentrisch

Regelsatz

- Oft konfigurierbar, aber nicht immer
- So aggressiv wie möglich
- Aber: soll nicht den Entwicklungsprozess beeinflussen

Integration in Editoren

- Highlighting
- Format on Save
- Quick Fixes

Editorconfig

- Überschreibt Einstellungen
- Cross-IDE / -Editor Config
  - Aber auch Editor-spezifische Optionen
- Besonders relevant bei Autocomplete

```
[*]
end_of_line = lf
insert_final_newline = true

# 4 space indentation
[*.py]
indent_style = space
indent_size = 4

# Editor-specific
Ij_kotlin_name_count_to_use_star_import = 3
```

Git Commit Hooks

- Überprüfen beim Einchecken von Code
- Egal ob Editor richtig konfiguriert ist
- Findet Probleme vor dem Review / PR-Builder
- Liegen in `.git/hooks`

pre-commit

- “A framework for managing and maintaining multi-language pre-commit hooks.”
- Oft direkt von den Tools unterstützt
  - Beispiel: Prettier
- Nachteil: Teilweise eigene Config

Demo Stack

- Node.js 12
  - JavaScript lässt besonders viel Freiraum
- Formatter: Prettier
- Linter: ESLint
- Hooks: Prettier + ESLint