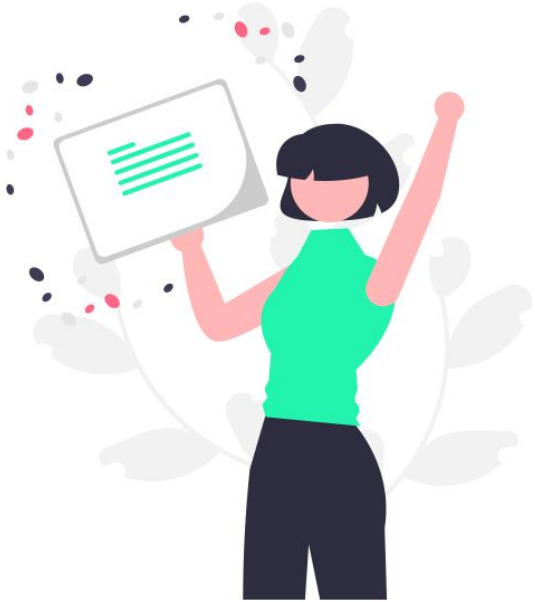
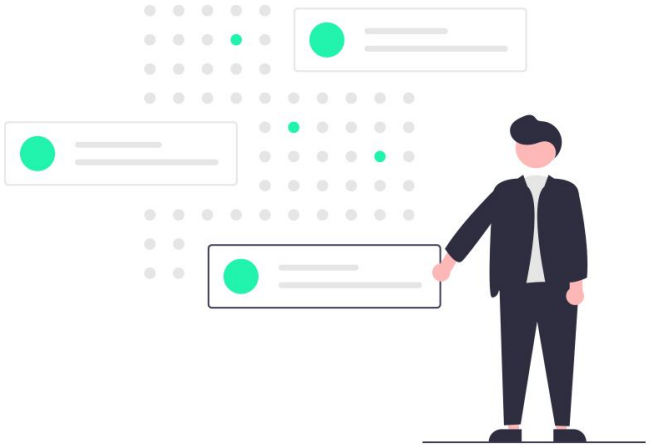
Jederzeit deployen können

Eine alltagstaugliche Implementierung kurzlebiger Feature Toggles

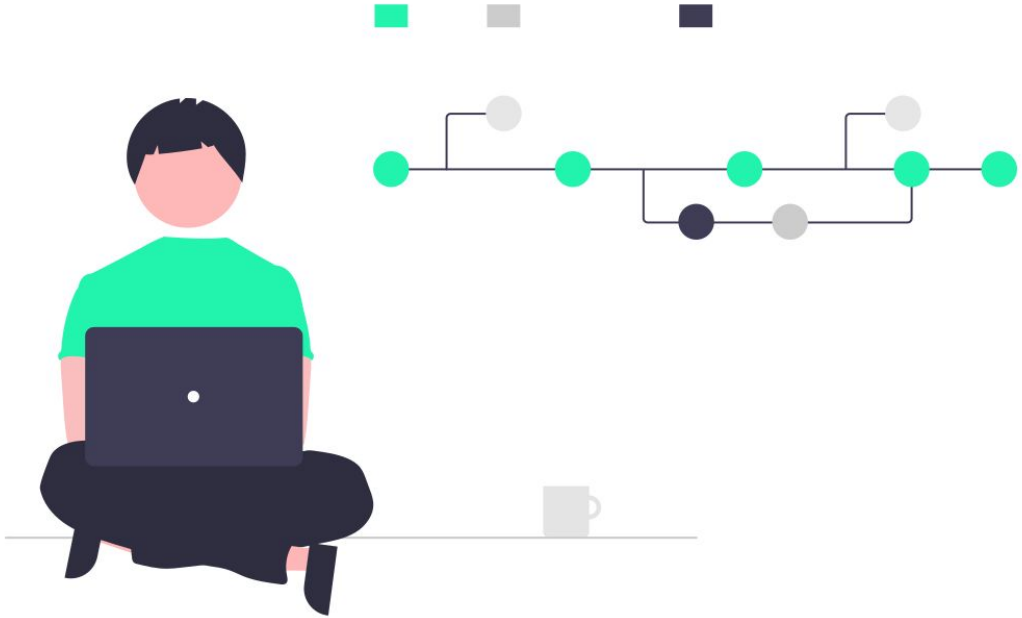
von Miriam Greis und Philipp Krauß

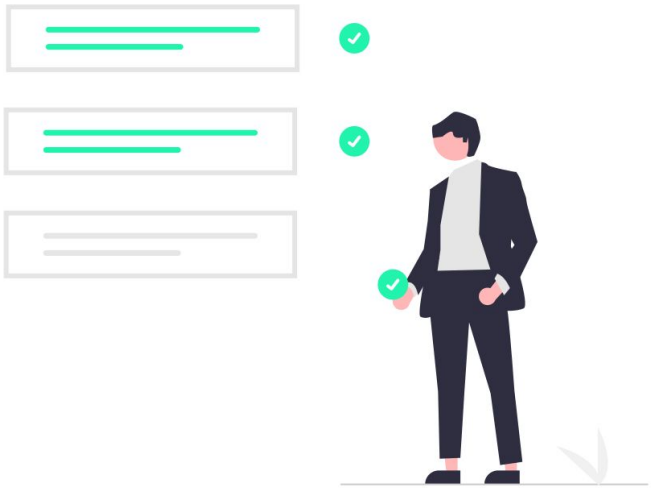


“Das neue Feature ist endlich fertig.”

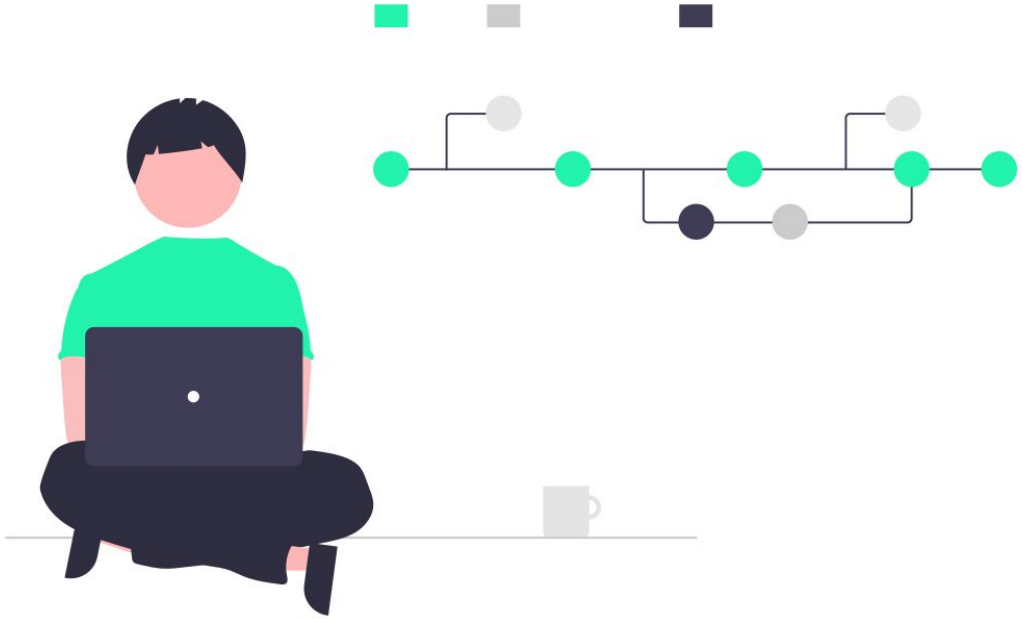


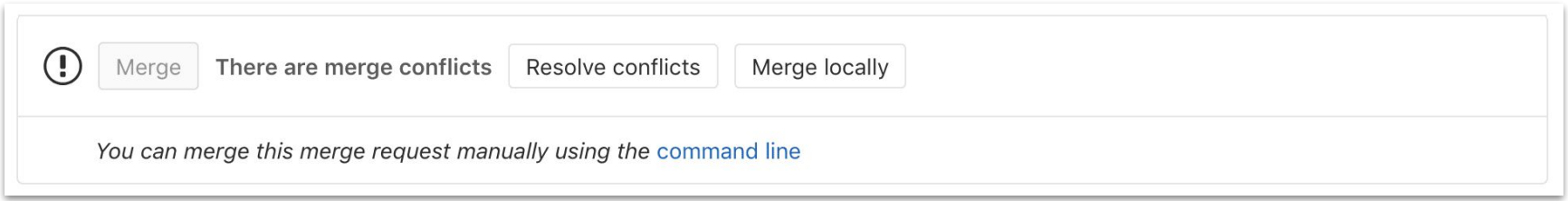
“Die Zustimmung der Rechtsabteilung fehlt leider noch.”

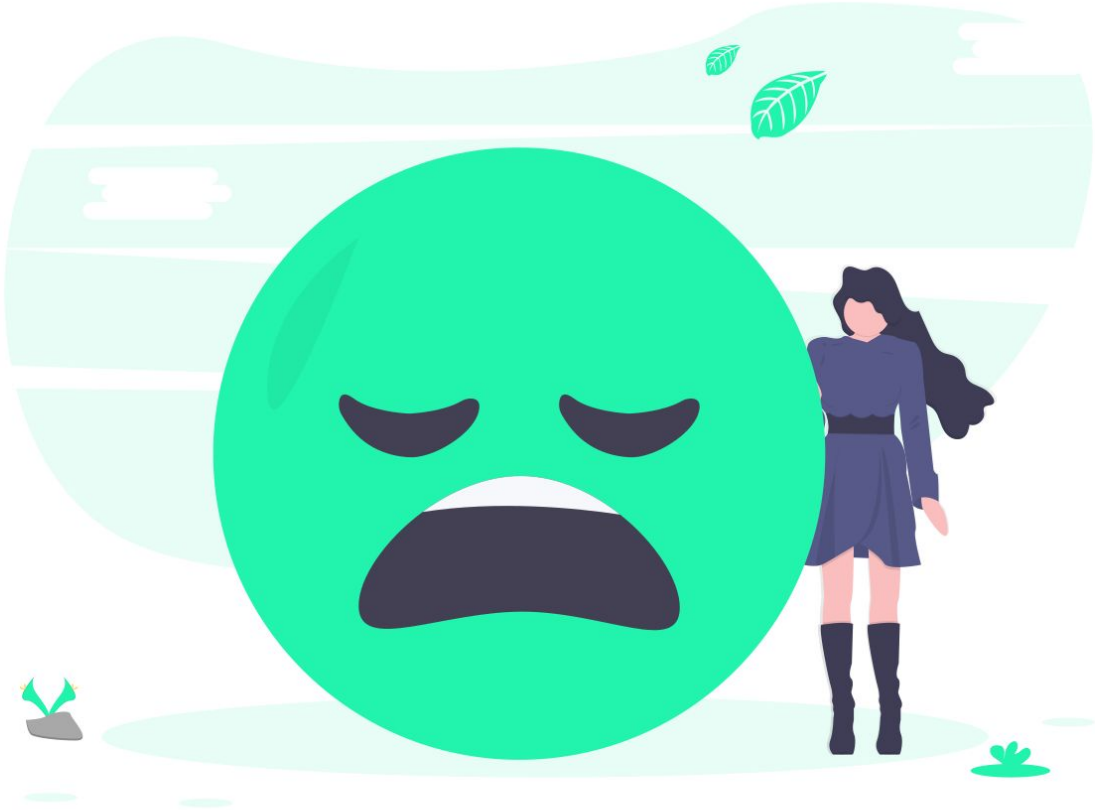




“Das Feature soll morgen live gehen.”

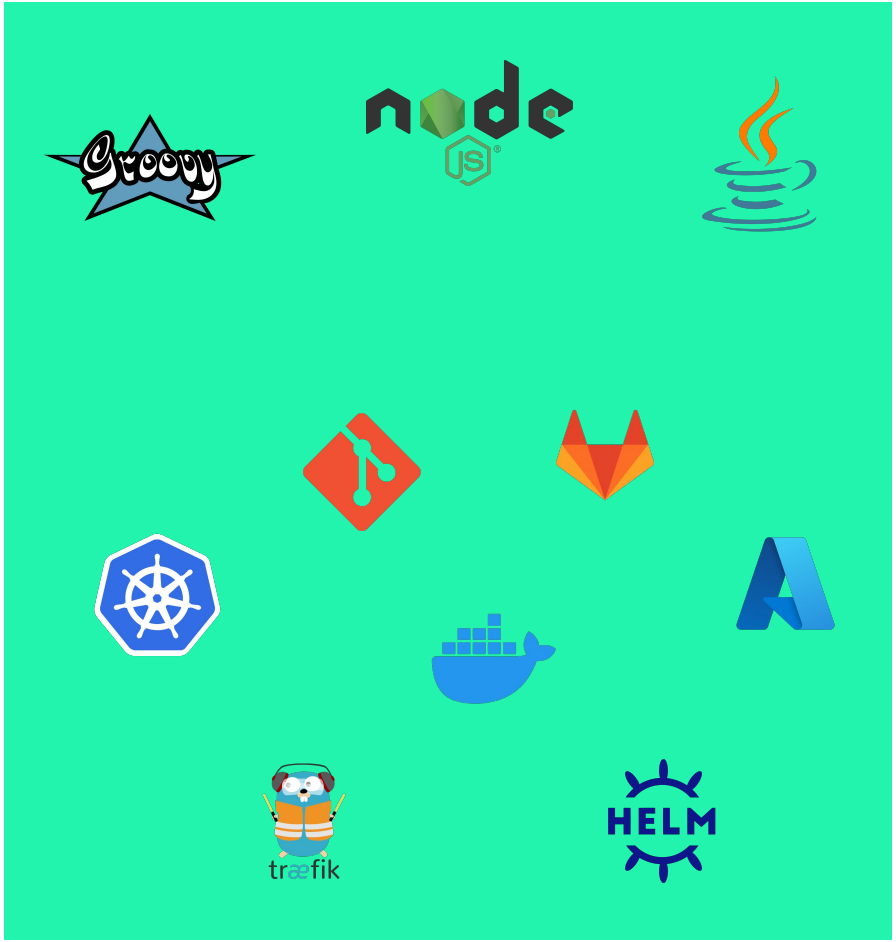



Merge
There are merge conflicts
Resolve conflicts
Merge locally
You can merge this merge request manually using the command line



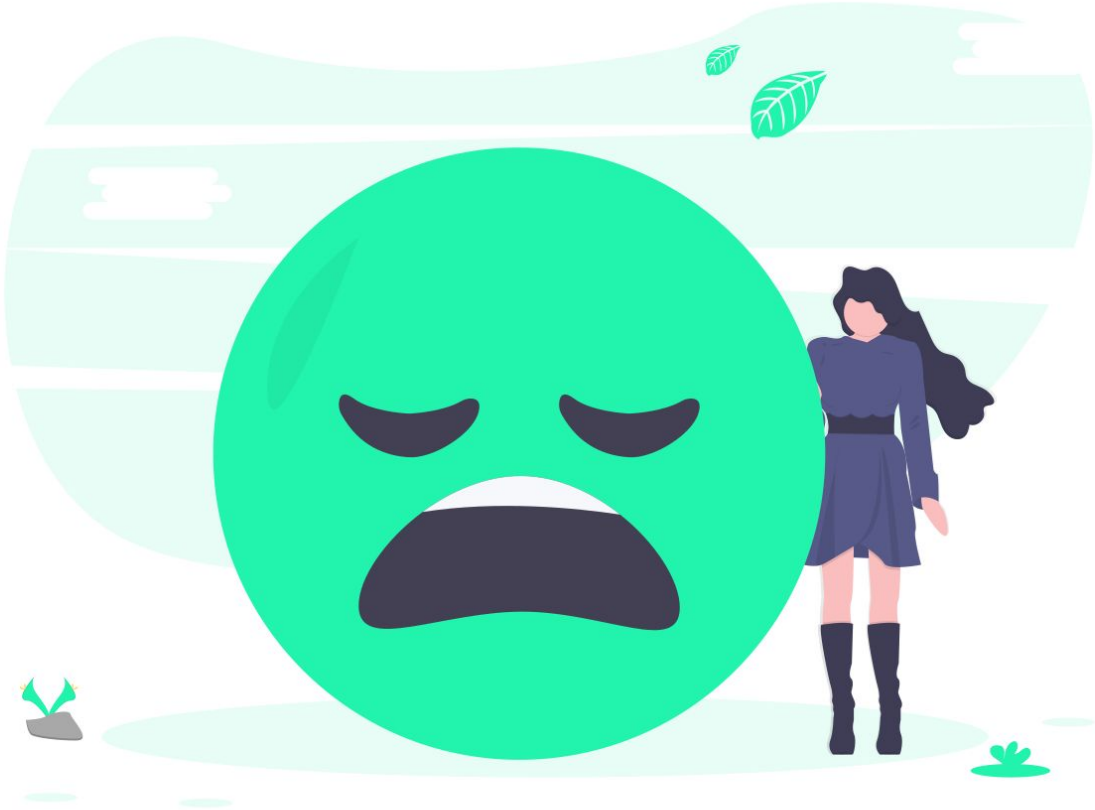
miriam.greis@codecentric.de

philipp.p.krauss@mercedes-benz.com



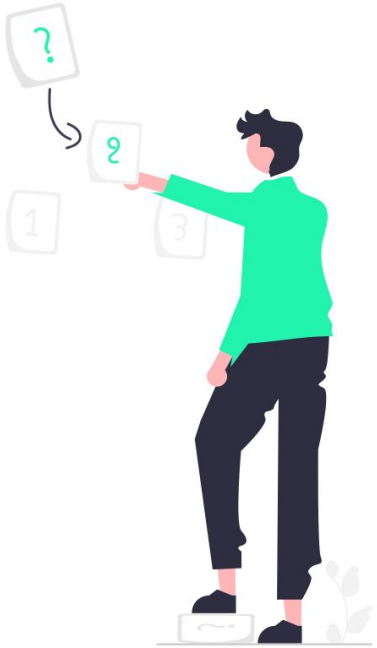
Das Projekt

- Web Application
- Content Management System
- Microservice Backend
- Build, Deploy & Test Pipelines
- Unit, Integration & E2E Tests
- 3 Umgebungen:
  Dev, Stage, Prod

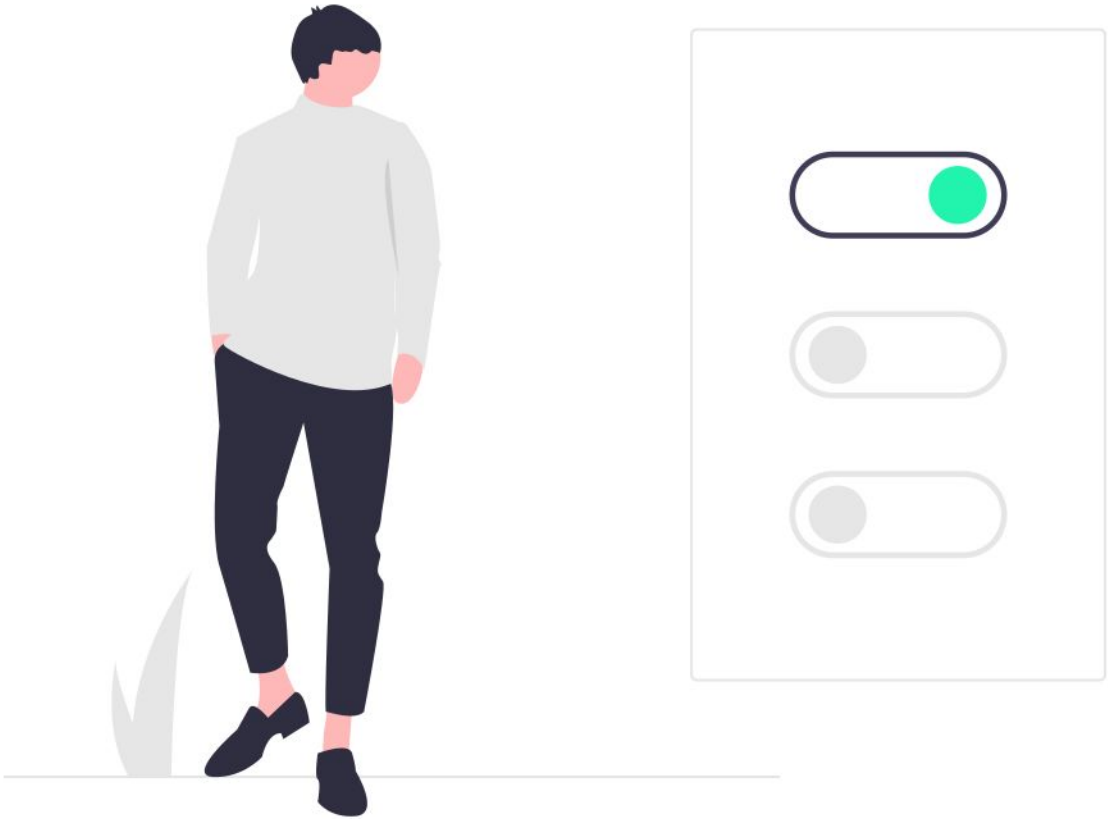


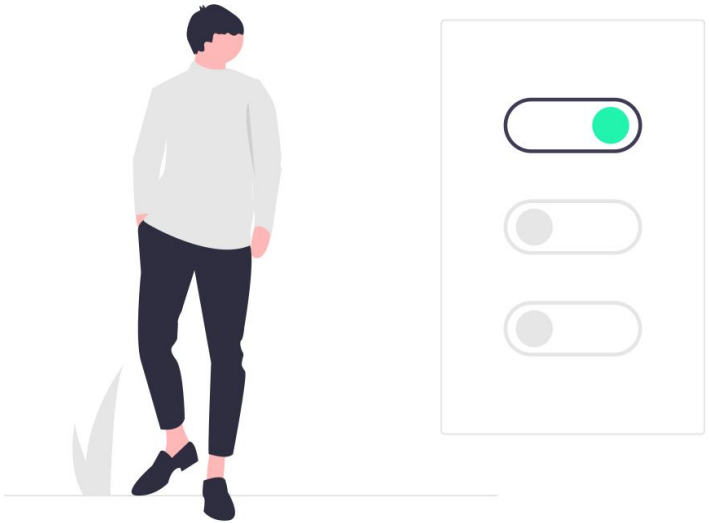
“Wie machen wir das nächstes Mal besser?”

Idee: Lass uns trunk-basierte Entwicklung machen!



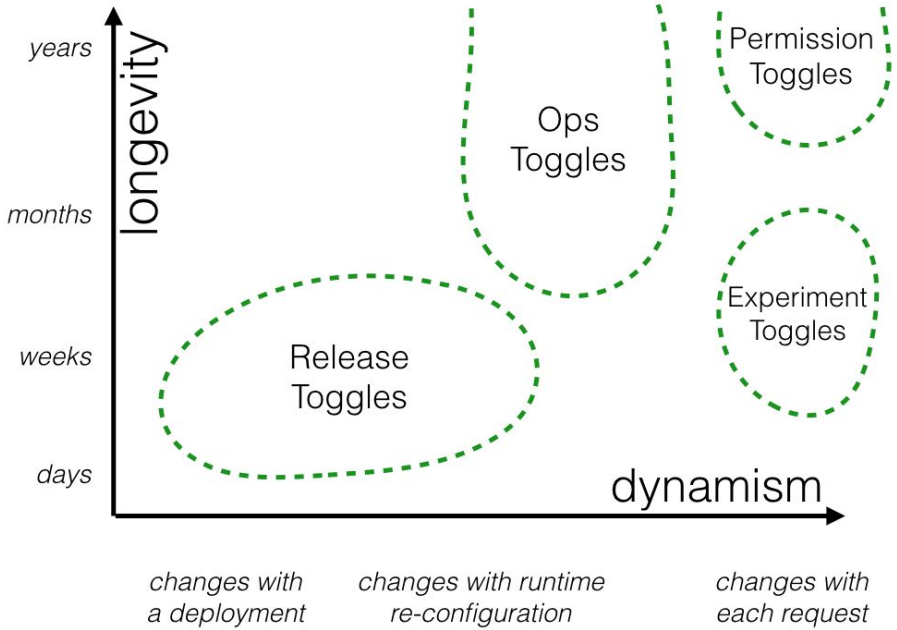
“Aber dann können wir ja nicht mehr jederzeit nach Prod deployen… ?”





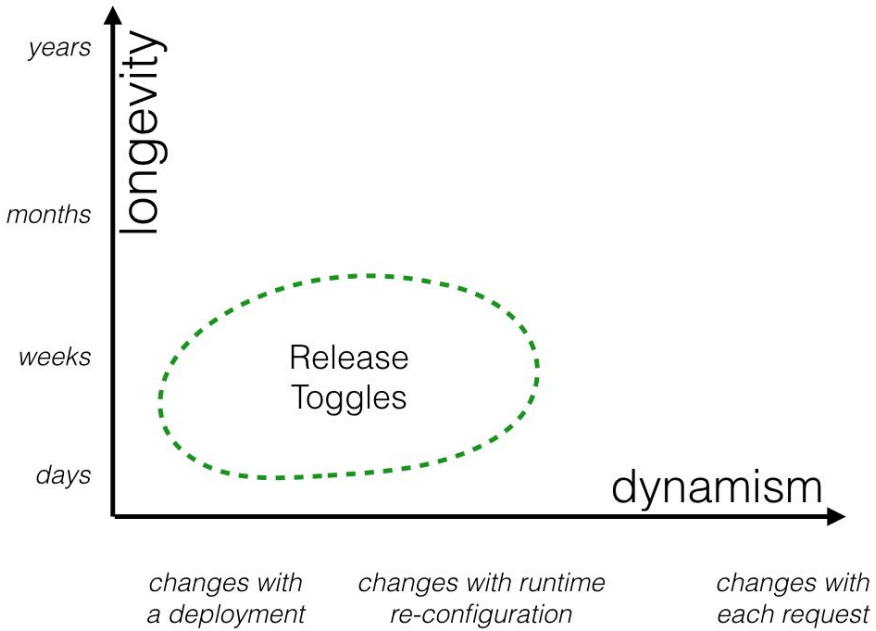
“Aber benutzt man Feature Toggles nicht für etwas ganz anderes?”

Arten von Feature Toggles

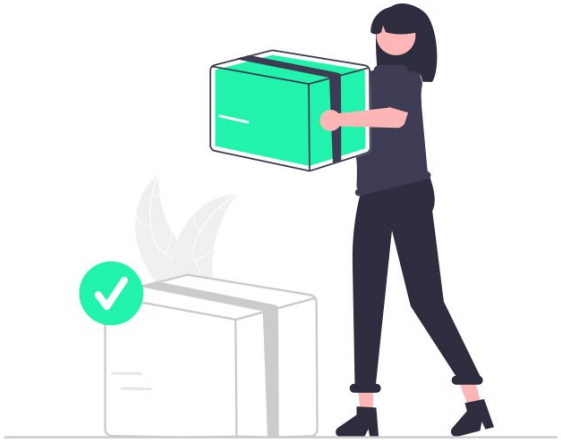


Quelle: https://martinfowler.com/articles/feature-toggles.html

Release Toggles



Quelle: https://martinfowler.com/articles/feature-toggles.html

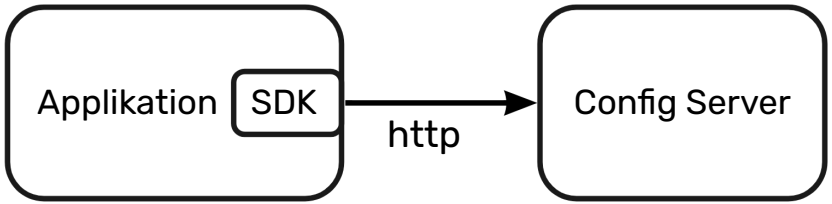


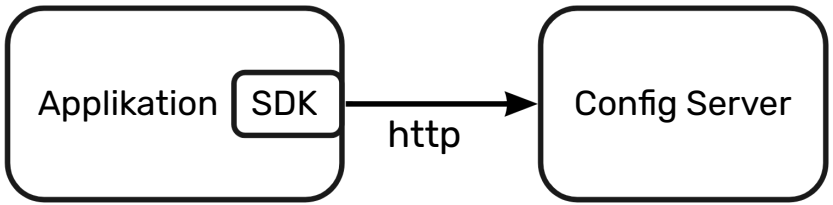
“Wollen wir das selbst bauen? Oder etwas bestehendes verwenden?”

Existierende Lösungen

spring Cloud Config Server

GitLab Feature Flags





Nachteile

Zusätzliche Komponenten nötig

Neue Technologien

Was, wenn der config server ausfällt?

“Wir wollen einen neuen Newsletter Provider nutzen.”

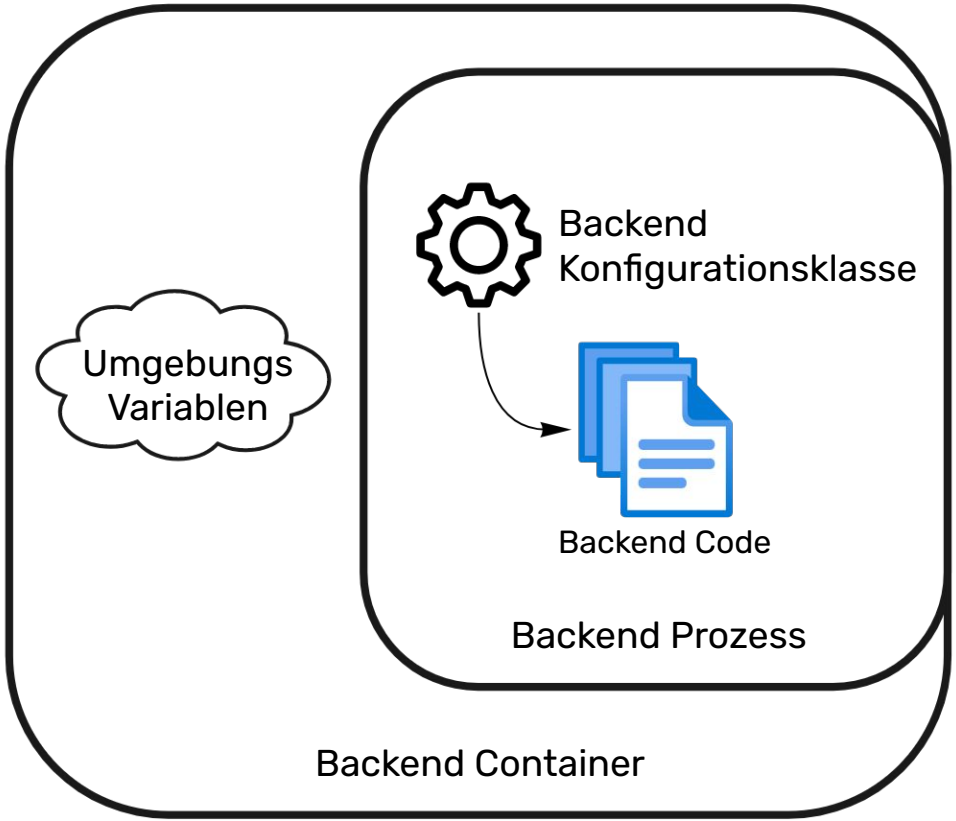
Anforderungen an den Mechanismus

Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Idee: Feature Toggles sind nur Konfiguration

Konfiguration



FeaturesConfig.groovy

```
class FeaturesConfig {
  boolean useNewNewsletterProvider
}
```

Ratpack.groovy

```
serverConfig {
  props(Resources.asByteSource(Resources.getResource('application.properties')))
  env()

  require('/features', FeaturesConfig)
}
```

```
export RATPACK_FEATURES__USE_NEW_NEWSLETTER_PROVIDER="true"
```

NewsletterHandler.groovy

```groovy
class NewsletterHandler extends GroovyHandler {

  @Inject
  FeaturesConfig featuresConfig

  @Override
  protected void handle(GroovyContext context) {
     if (featuresConfig.useNewNewsletterProvider) {
        useNewNewsletterProvider()
     } else {
        useOldNewsletterProvider()
     }
  }
}
```

Anforderungen an den Mechanismus

Einbau

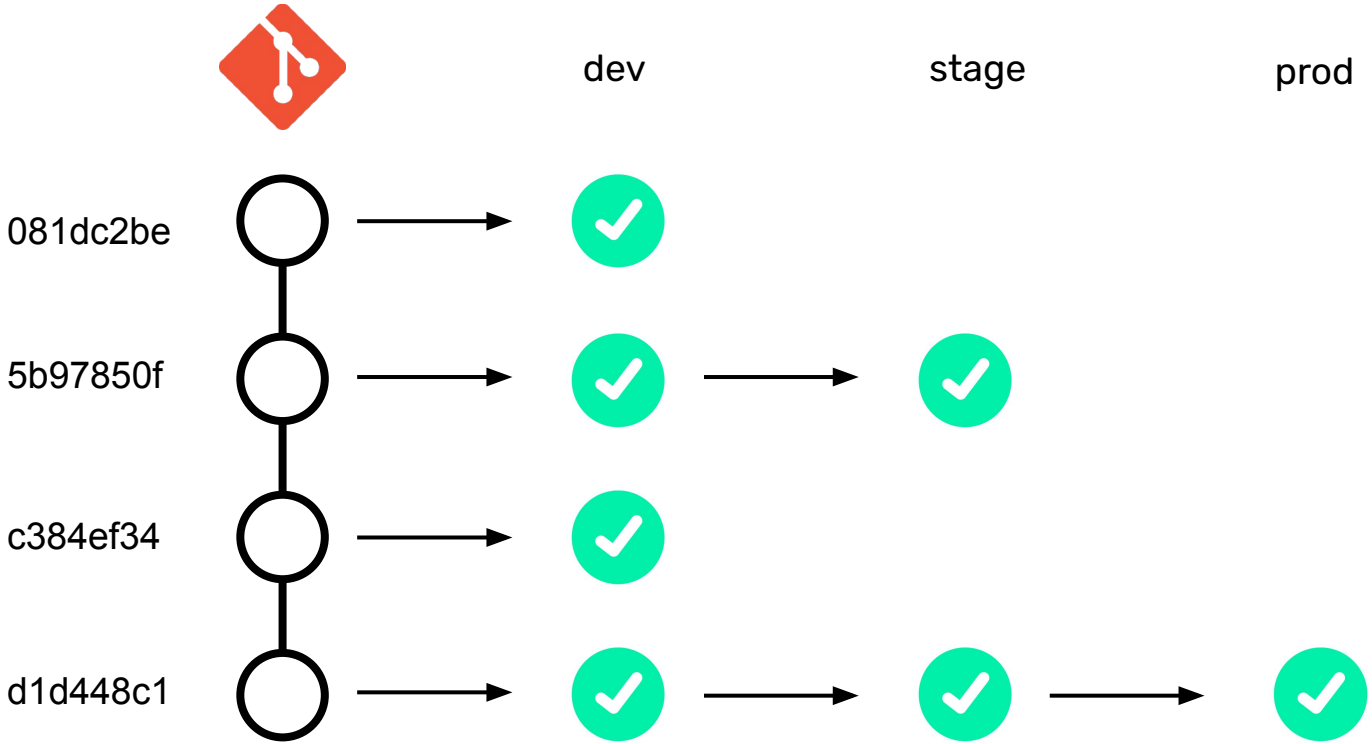
Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

“Aber wie machen wir das in den Unit Tests?”

```
void 'test using new endpoint'() {
  setup:
  FeaturesConfig featuresConfig = new FeaturesConfig(useNewNewsletterProvider: true)
  NewsletterHandler handler = new NewsletterHandler(featuresConfig: featuresConfig)

  when:
  handler.handle()

  then:
  assertNewApiCalled()
}


void 'test using old endpoint'() {
  setup:
  FeaturesConfig featuresConfig = new FeaturesConfig(useNewNewsletterProvider: false)
  NewsletterHandler handler = new NewsletterHandler(featuresConfig: featuresConfig)

  when:
  handler.handle()

  then:
  assertOldApiCalled()
}
```

“Wie wollen wir das Feature Toggle einschalten?”

Deployment Strategie



Anforderungen an den Mechanismus

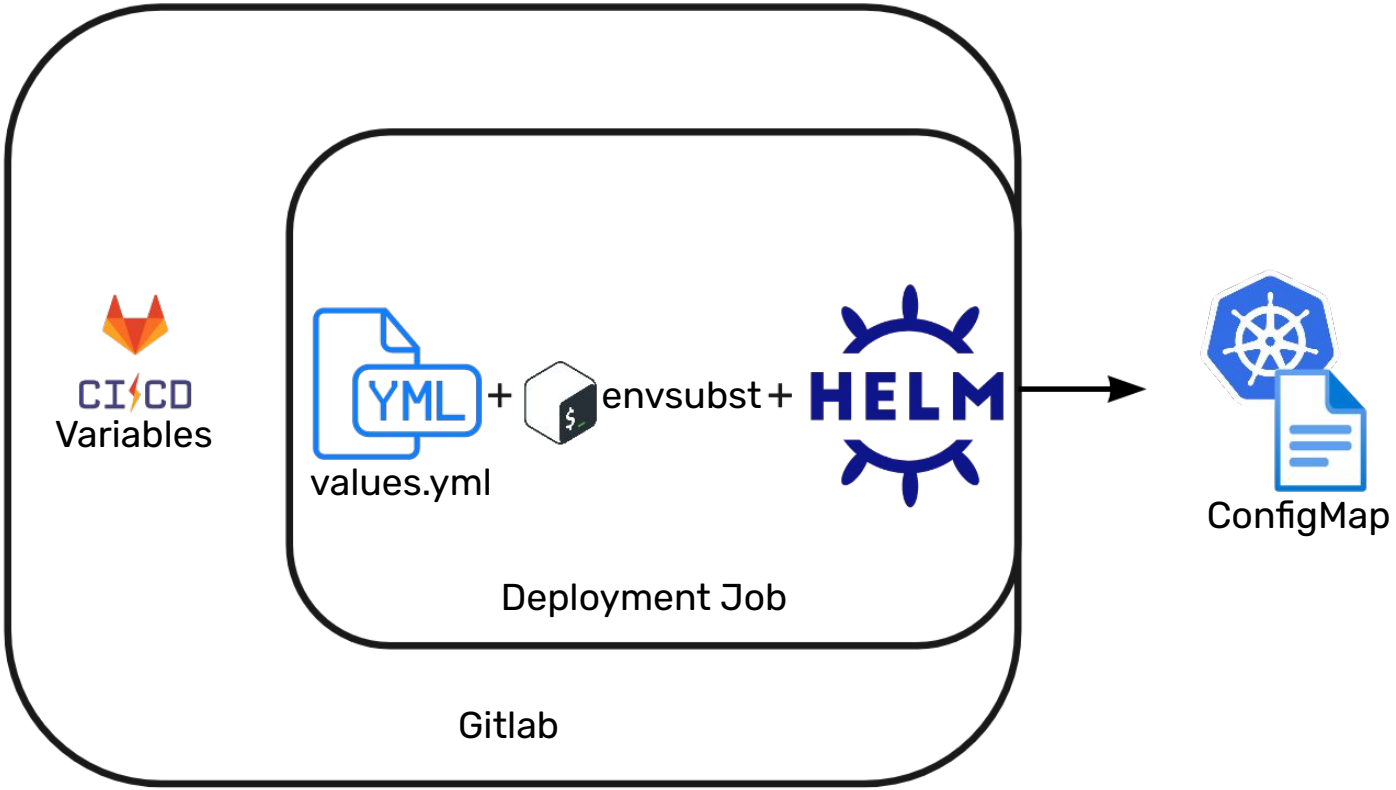
Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Aktivierung

Es soll kein Commit notwendig sein um das Feature Toggle zu aktivieren.

Konfiguration



Type	↑ Key	Value	Protected	Masked	Environments
Variable	USE_NEW_NEWSLETTER_PROVIDER	true	×	×	dev
Variable	USE_NEW_NEWSLETTER_PROVIDER	false	×	×	stage
Variable	USE_NEW_NEWSLETTER_PROVIDER	false	×	×	prod

```yaml
backend:
 config:
   RATPACK_FEATURES__USE_NEW_NEWSLETTER_PROVIDER: "${USE_NEW_NEWSLETTER_PROVIDER}"
```


```yaml
backend:
 config:
   RATPACK_FEATURES__USE_NEW_NEWSLETTER_PROVIDER: "true"
```

Anforderungen an den Mechanismus

Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Aktivierung

Es soll kein Commit notwendig sein um das Feature Toggle zu aktivieren. Ein Deploy ist aber in Ordnung.

Anforderungen an den Mechanismus

Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Aktivierung

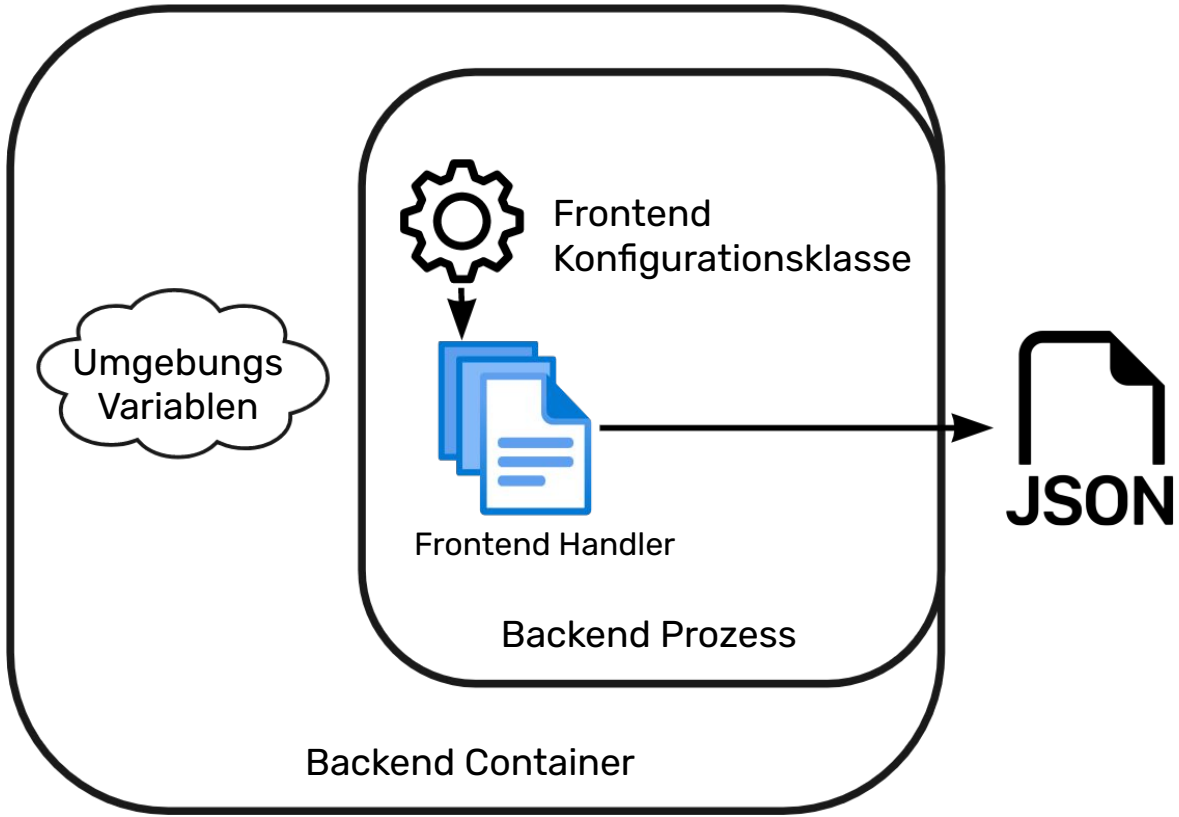
Es soll kein Commit notwendig sein um das Feature Toggle zu aktivieren. Ein Deploy ist aber in Ordnung.

Umgebung

Das unabhängige Aktivieren von Feature Toggles für dev, stage und production ist so einfach wie möglich.

“Wir müssen doch noch den Text im Frontend für den neuen Newsletter Provider anpassen, da sich die rechtlichen Bedingungen ändern.”

Feature Toggles endpoint



FrontendConfig.groovy

```
class FrontendConfig extends
LinkedHashMap {}
```

FrontendConfigHandler.groovy

```
class FrontendConfigHandler extends GroovyHandler {

  @Inject
  FrontendConfig frontendConfig

  @Override
  protected void handle(GroovyContext context){
     context.render json(frontendConfig)
  }
}
```

FrontendConfig.groovy

```
class FrontendConfig extends
LinkedHashMap {}
```

Ratpack.groovy

```
require('/frontend', FrontendConfig)

<...>

prefix('config'){
  get(FrontendConfigHandler)
}
```

FrontendConfigHandler.groovy

```
class FrontendConfigHandler extends GroovyHandler {

  @Inject
  FrontendConfig frontendConfig

  @Override
  protected void handle(GroovyContext context){
     context.render json(frontendConfig)
  }
}
```

values.yaml

```
backend:
 config:
   RATPACK_FRONTEND__USE_NEW_NEWSLETTER_PROVIDER: "${USE_NEW_NEWSLETTER_PROVIDER}"
```

FrontendConfig.groovy

```
class FrontendConfig extends
LinkedHashMap {}
```

Ratpack.groovy

```
require('/frontend', FrontendConfig)

<...>

prefix('config'){
  get(FrontendConfigHandler)
}
```

FrontendConfigHandler.groovy

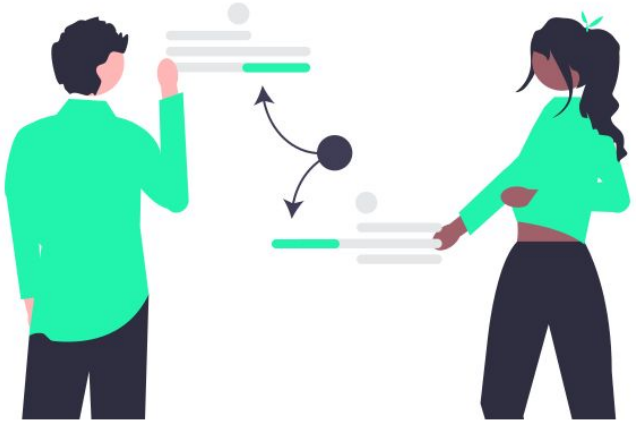
```
class FrontendConfigHandler extends GroovyHandler {

  @Inject
  FrontendConfig frontendConfig

  @Override
  protected void handle(GroovyContext context){
     context.render json(frontendConfig)
  }
}
```

values.yaml

```
backend:
 config:
   RATPACK_FRONTEND__USE_NEW_NEWSLETTER_PROVIDER: "${USE_NEW_NEWSLETTER_PROVIDER}"
```

GET /config

```
{
  "useNewNewsletterProvider": true
}
```



“Warum machen wir das für die Frontend Feature Toggles anders als für die Backend Feature Toggles?”

Gründe für die Unterscheidung zwischen Backend und Frontend Feature Toggles

Geringer Aufwand

Im Backend-Code sind keine Änderungen notwendig um ein Frontend Feature Toggle einzubauen.

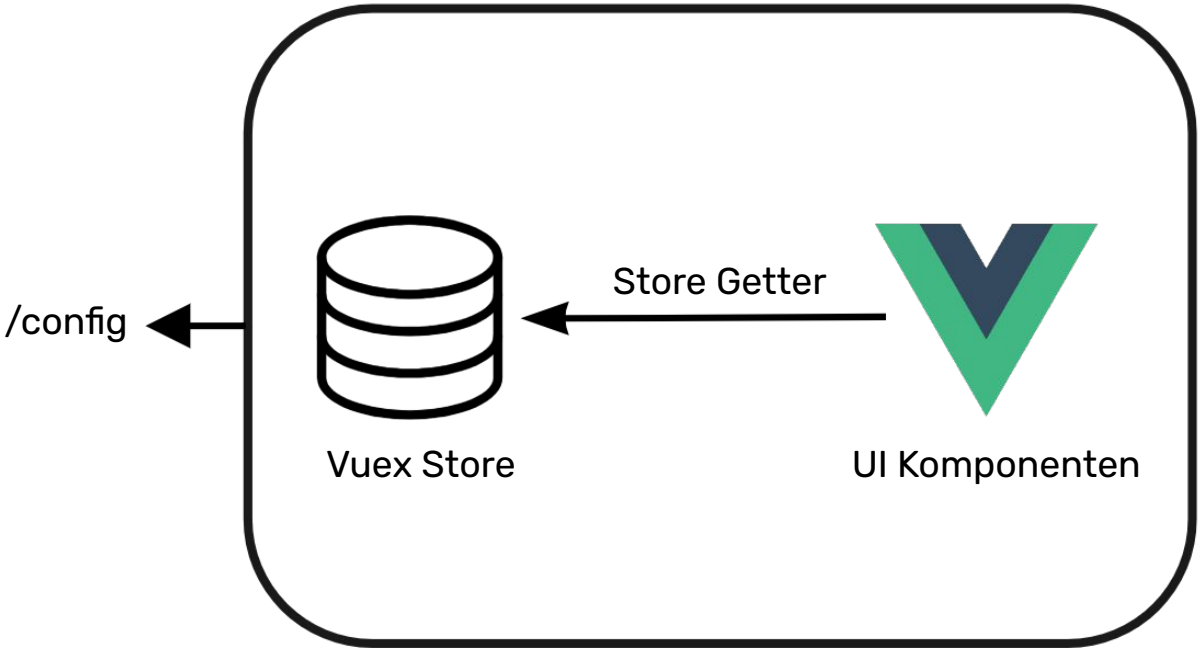
Keine Warnungen

Im Backend-Code wären Frontend Feature Toggles unbenutzte Variablen, die Warnungen erzeugen.

IDE-Funktionalität

Für die Backend Feature Toggles kann die Verwendung der Feature Toggle Variablen über die IDE einfach geprüft werden.

Frontend ruft Feature Toggles über /config ab



featureToggles.json

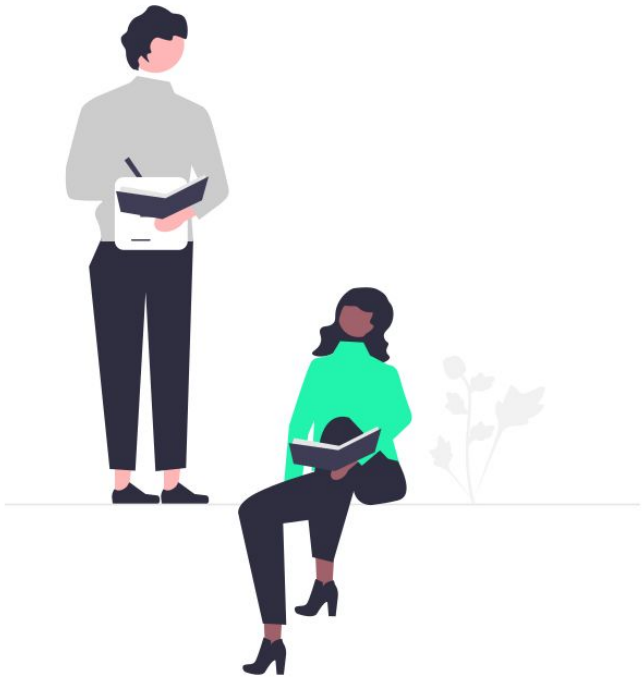
```
["useNewNewsletterProvider" ]
```

ConfigModule.js

```
const featureToggles = require('@/featureToggles.json')

const state = {
  config: {},
}

const mutations = {
  setConfig(state, config) { state.config = config },
}

const actions = {
  async loadConfig({ commit, state }) {
    const response = await axios.get('/config')
    commit('setConfig', response.data)
  },
}

const getters = {}
featureToggles.forEach((featureToggle) =>
  (getters[featureToggle] = (state) =>
    state.config[featureToggle]))
```

NewsletterBox.vue

```
<template>
  <div>
    <new-newsletter-box  v-if="useNewNewsletterProvider " />
    <old-newsletter-box  v-else />
  </div>
</template>

<script>
<...>
computed: {
  ...mapGetters(['useNewNewsletterProvider' ]),
}
<...>
</script>
```



“Gibt es denn keine bessere Lösung als die Frontend Feature Toggles über einen REST Endpunkt zu laden?”

Eine bessere Lösung?

Statisches Rendern

Die Konfiguration könnte beim ersten Laden in der index.html statisch gerendert werden.

Window-Objekt

Ein Skript im Header der index.html könnte die Feature Toggles über das window-Objekt zur Verfügung stellen.

Neue Anforderungen

Solange es keine neuen Anforderungen gibt, ist es nicht nötig den aktuellen Mechanismus abzulösen.

1. CI/CD Variable Gitlab

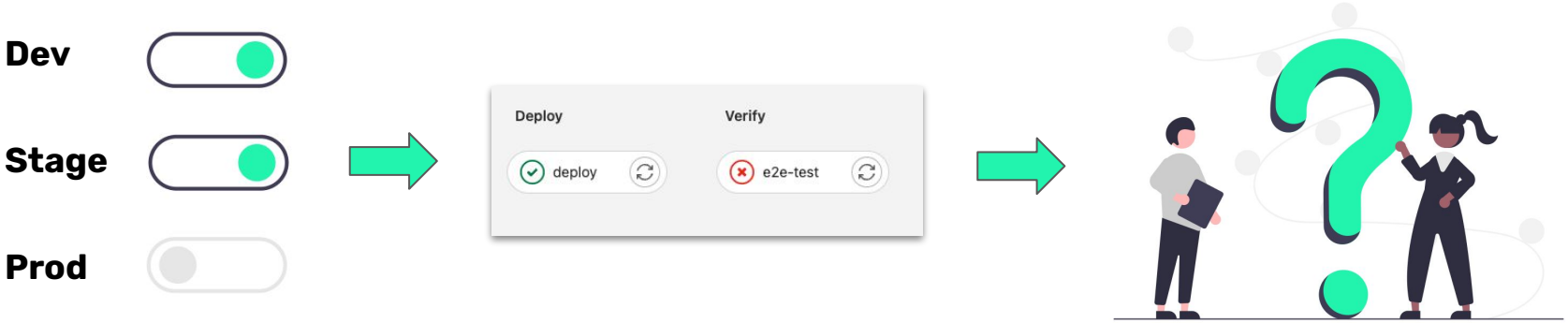
Type	↑ Key	Value	Protected	Masked	Environments	
Variable	USE_NEW_NEWSLETTER_PROVIDER	true	×	×	dev	
Variable	USE_NEW_NEWSLETTER_PROVIDER	false	×	×	stage	
Variable	USE_NEW_NEWSLETTER_PROVIDER	false	×	×	prod	

2. values.yaml

```yaml
backend:
 config:
   RATPACK_FRONTEND__USE_NEW_NEWSLETTER_PROVIDER: "${USE_NEW_NEWSLETTER_PROVIDER}"
```

3. featureToggles.json

```json
["useNewNewsletterProvider"]
```


Dev
Stage
Prod
Deploy
Verify
deploy
e2e-test

configLoader.js

```
let config

async function loadConfig() {
  const targetURL = <...>
  const axios = require('axios')
  await axios.get(`${targetURL}/config`})
    .then((response) => {
      config = response.data
    })
}

function getConfig() {
  return config
}

module.exports = {
  loadConfig,
  getConfig,
}
```

configLoader.js

```
let config

async function loadConfig() {
  const targetURL = <...>
  const axios = require('axios')
  await axios.get(`${targetURL}/config`})
    .then((response) => {
      config = response.data
    })
}

function getConfig() {
  return config
}

module.exports = {
  loadConfig,
  getConfig,
}
```

nightwatch.globals.js

```
const { loadConfig } = require('./configLoader')

module.exports = {
  async before(callback) {
    await loadConfig()
    callback()
  },
}
```

nightwatch.config.js

```
module.exports = {
  <...>
  globals_path: 'nightwatch.globals.js',
  <...>
}
```

util.js

```
const { getConfig } = require('./configLoader')

function isFeatureToggleEnabled(featureToggle) {
  const config = getConfig()
  return config && config[featureToggle]
}
```

NewsletterProvider.test.js

```
if (!isFeatureToggleEnabled('useNewNewsletterProvider')) {
  <...>
}
```

Skip Zeilen

util.js

```
function skipForEnabledFeatureToggle(featureToggle) {
  if (isFeatureToggleEnabled(featureToggle)) {
    return true
  }
  return false
}
```

OldNewsletterProvider.test.js

```
const skipForEnabledFeatureToggle = require('../util').skipForEnabledFeatureToggle

module.exports = {
  '@disabled': skipForEnabledFeatureToggle('useNewNewsletterProvider'),
  <...>
}
```

Skip Test

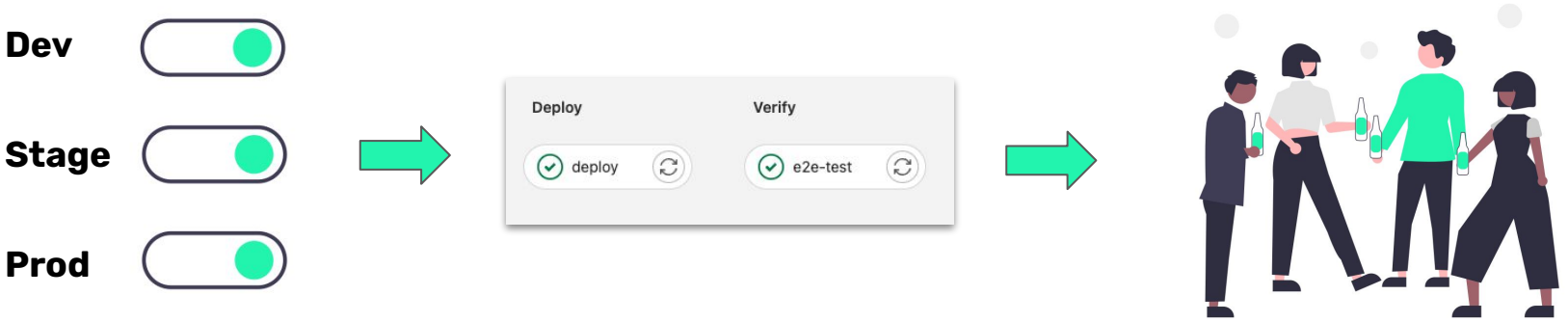
util.js

```
function skipForDisabledFeatureToggle(featureToggle) {
  if (!isFeatureToggleEnabled (featureToggle)) {
    return true
  }
  return false
}
```

NewNewsletterProvider.test.js

Skip Test

```
const skipForDisabledFeatureToggle  = require('../util').skipForDisabledFeatureToggle

module.exports = {
  '@disabled': skipForDisabledFeatureToggle('useNewNewsletterProvider'),
  <...>
}
```


Dev
Stage
Prod
Deploy
Verify
deploy
e2e-test

“Wir sollten noch das Feature Toggle aus dem Code löschen. Wie finden wir allen überflüssigen Code am besten?”

Anforderungen an den Mechanismus

Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Aktivierung

Es soll kein Commit notwendig sein um das Feature Toggle zu aktivieren. Ein Deploy ist aber in Ordnung.

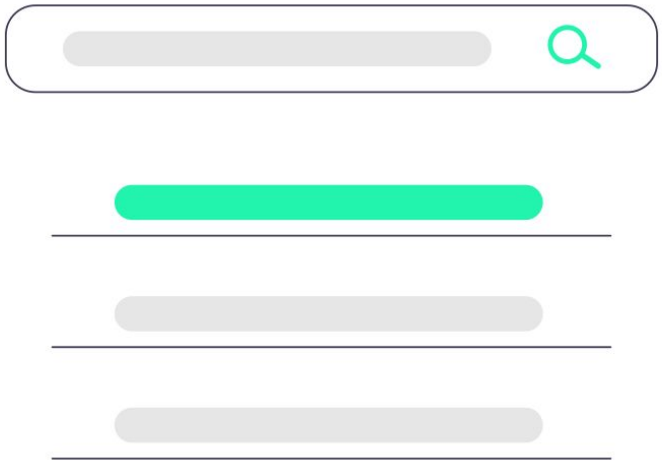
Umgebung

Das unabhängige Aktivieren von Feature Toggles für dev, stage und production ist so einfach wie möglich.

Ausbau

Das Ausbauen von Feature Toggles ist so einfach wie möglich, d.h. es können einfach alle Stellen gefunden und gelöscht werden.

Was muss ausgebaut werden?



Volltextsuche

IDE find usages

Duplikate statt Logik

Kommentare

Tests failed: 4, passed: 1,130 of 1,134 tests – 2 min 19 sec

Test Results 2 min 19 sec

NewsletterSubscriptionForm.spec.js 460 ms

Newsletter.integration.spec.js 21 ms

```
Error: expect(received).toEqual(expected) // deep equality

Expected: true
Received: false
<Click to see difference>
```

Idee: Feature Toggles sollen in den Unit & Integration Tests immer aktiviert sein.

vuex.js

realer Store

```
const featureToggles = require('@/featureToggles.json')

async function getIsolatedStore(config = {}) {
  const defaultConfig = {}
  featureToggles.forEach((featureToggle) => (defaultConfig[featureToggle] = true))
  let store = new Vuex.Store({<...>})
  store.state.config.config = merge(defaultConfig, config)
  return store
}

export { getIsolatedStore }
```

NewsletterSubscriptionForm.spec.js

gemockter Store

```
let store = new Vuex.Store({
  getters: {
    <...>
    useNewNewsletterProvider : jest.fn(),
  },
})
```

Anforderungen an den Mechanismus

Einbau

Das Einbauen neuer Feature Toggles ist so einfach wie möglich, d.h. es erfordert gezielte Änderungen an wenigen Stellen.

Aktivierung

Es soll kein Commit notwendig sein um das Feature Toggle zu aktivieren. Ein Deploy ist aber in Ordnung.

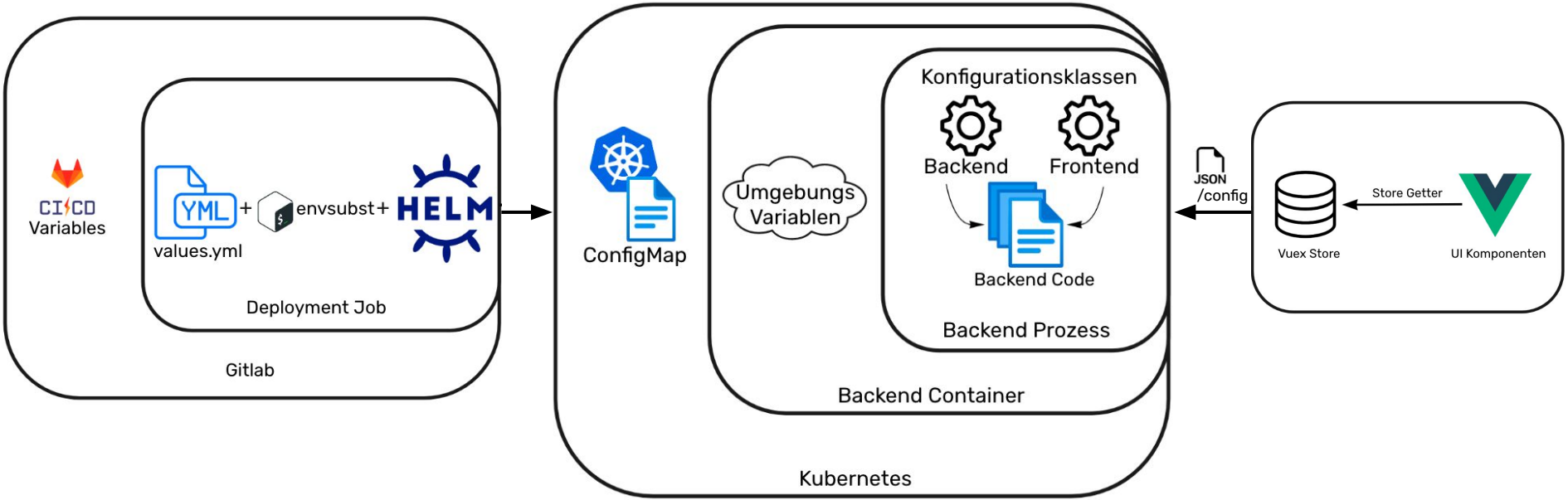
Umgebung

Das unabhängige Aktivieren von Feature Toggles für dev, stage und production ist so einfach wie möglich.

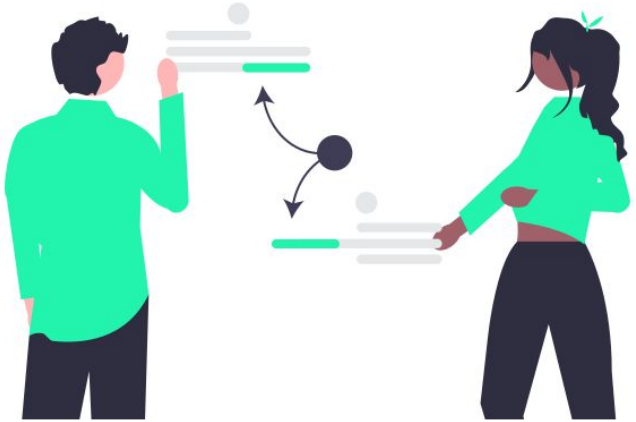
Ausbau

Das Ausbauen von Feature Toggles ist so einfach wie möglich, d.h. mit einer Volltextsuche können alle Stellen gefunden und gelöscht werden.

Die Methode im Überblick



Tips und Tricks



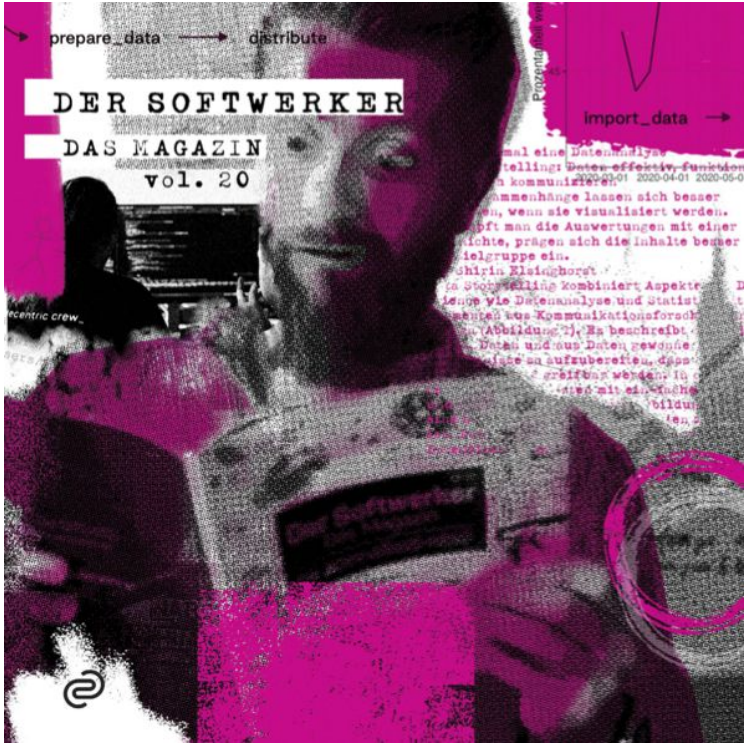
Festlegen auf die Art der Feature Toggles

Es muss nicht immer ein neuer Service sein

Beim Einbau schon an den Ausbau denken

E2E Tests abhängig von Feature Toggles laufen lassen

Feature Toggles in Unit Tests standardmäßig aktivieren



https://www.codecentric.de/wissen/softwerker

miriam.greis@codecentric.de
philipp.p.krauss@mercedes-benz.com