



ReadTheDocs und e2e Tests mit Playwright

Programm für heute



QA|WARE

Vorlesung

1. Vorstellung ReadTheDocs
2. Patterns & Best Practices für e2e Tests
3. Vorstellung Playwright

Übung

1. Deployment einer Dokumentation in ReadTheDocs
2. Einrichten von Playwright und Anbinden an die Pipeline
3. Optional: Last-Testing




QA|WARE

hey there from:

...Roland Hummel!



QA|WARE



Roland Hummel
Software Engineer bei RIO
Rosenheim, Bavaria, Germany

- Software Engineer seit über 15 Jahren
- Technically full-stack
- Starker Fokus auf frontend-stuff (usually referred to as “that JS guy”)
- Find me on:
 - <https://github.com/defaude>
 - <https://linkedin.com/in/roland-hummel-097849242>



QA|WARE

ReadTheDocs

Noch einmal ein Überblick über arc42.

RECAP



QA|WARE

1. Einführung & Ziele

Grundlegende Anforderungen, insbesondere Qualitätsziele

2. Randbedingungen

Regelungen und externe Randbedingungen

3. Kontext & Abgrenzung

Externe Systeme und Schnittstellen

4. Lösungsstrategie

Kernideen und Lösungsansätze

5. Bausteinsicht

Aufbau des Quellcodes, Modularisierung (hierarchisch)

6. Laufzeitsicht

Wichtige Laufzeitszenarien

7. Verteilungssicht

Hardware, Infrastruktur & Deployment

8. Querschnittliche Konzepte

Querschnittsthemen, oft sehr technisch und detailliert

9. Architekturentscheidungen

Wichtige Entscheidungen (nicht anderweitig beschrieben)

10. Qualitätsanforderungen

Qualitätsbaum, Qualitätsszenarien

11. Risiken & Technische Schulden

Bekannte Probleme und Risiken

12. Glossar

Wichtige und spezifische Begriffe ("gemeinsame Sprache")

5C's for Technical Writers

RECAP



QA|WARE

Clarity

Clear text and information.

Concise

Keep it short. Only provide required information.

Consistent

Use the same word throughout the whole document.

Complete

Give all required information but keep **Concise** in mind.

Correct

Mistakes and typos are bad for the users and your trust.

Problem: Wir brauchen eine zentrale Stelle für unsere Dokumentation.



QA|WARE

Challenge: Nicht alle Dokumentation im Code bewegt sich auf der gleichen Flughöhe:

- Methodenkommentare
- Fachliche Dokumentation
- ADRs

Lösung: Hierfür gibt es verschiedene Möglichkeiten:

- Direkt in der Codebase
- In externen Tools:
 - **ReadTheDocs**
 - Confluence
 - Github Wiki

ReadTheDocs baut und hostet Dokumentation aus Repos.



QA|WARE

- ReadTheDocs ist eine Plattform, die automatisch Dokumentation aus einem Git-Repository baut und hostet.
- Unterstützt Sphinx, das v.a. von Python Projekten bevorzugt verwendet wird
- Doku-Versionen lassen sich automatisch pro Branch oder Tag bereitstellen
- Ideal für CI/CD-Dokumentation und Open Source
- Einfach und schnell einzurichten und sehr mächtig
- Perfekt, um verschiedene Tiefen von Dokumentation zu bündeln

ReadTheDocs unterstützt Sphinx.



QA|WARE

Was ist **Sphinx**:

- Sphinx ist ein Generator für strukturierte, versionierte Dokumentation
- Ursprung in der Python-Welt, aber auch für andere Sprachen geeignet
- Unterstützt:
 - reStructuredText und Markdown als Quelle
 - Erweiterungen wie autodoc, napoleon, todo, intersphinx, uvm.
 - Ausgabeformate wie HTML, PDF, EPUB, LaTeX

ReadTheDocs lässt sich in wenigen Schritten einrichten.



QA|WARE

- Voraussetzung ist ein ReadTheDocs Account und ein **öffentliches** Repository
- Direkt von ReadTheDocs aus könnt ihr euer Github-Projekt verbinden
- Wenn alles richtig eingerichtet ist, baut ReadTheDocs die neue Dokumentation automatisch mit jedem Push
- Eure Projektstruktur sollte mindestens folgende Struktur beinhalten:

```
myproject/
├── docs/
│   ├── conf.py → zentrale Konfiguration
│   └── index.rst → Einstiegspunkt für eure Dokumentation
├── requirements.txt → Definiert benötigte Pakete (sphinx, themes, etc.)
├── .readthedocs.yaml → Einstiegspunkt für den readthedocs-build
└── README.md → jedes gute Repository enthält eine README als zentralen Einstieg
;-)
```

conf.py dient als zentrale Konfiguration.



QAWARE

```
# -*- coding: utf-8 -*-
#
# Configuration file for the Sphinx documentation builder.
#
# For the full list of built-in configuration values, see the documentation:
# https://www.sphinx-doc.org/en/master/usage/configuration.html

# -- Project information -----

project = '☕ CoffeeHub'
copyright = '2023, Felix Rampf'
author = 'Felix Rampf'

# -- General configuration -----

extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.viewcode',
    'sphinx copybutton',
    'myst_parser',
]

templates_path = ['_templates']
root_doc = 'index'
exclude_patterns = []

# The name of the Pygments (syntax highlighting) style to use.
pygments_dark_style = 'nord'

# -- Options for HTML output -----

html_theme = 'furo'
html_static_path = ['_static']
```

requirements.txt definiert benötigte Abhängigkeiten.



QAWARE

- nur per pip-install installierbare Pakete können definiert werden
- Beim build der Dokumentation installiert ReadTheDocs automatisch diese Abhängigkeiten.
- Beinhaltet Erweiterung, Themes, etc.

```
furo==2023.3.27
myst-parser==1.0.0
sphinx==6.2.1
sphinx-copybutton==0.5.2
```

ReadTheDocs unterstützt verschiedene Formate.



QA|WARE

Input:

ReadTheDocs basiert nativ auf **reStructuredText** (*.rst), unterstützt aber mit der Erweiterung **myst-parser** auch Markdown (*.md)

Output:

ReadTheDocs bietet:

- PDF
- HTML
- LaTeX
- EPUB

reStructuredText bietet viele Formatierungsmöglichkeiten.



QA|WARE

```
Willkommen zur Dokumentation!  
=====
```

```
.. toctree::  
    :maxdepth: 2  
    :caption: Inhalt:
```

```
    usage  
    api_reference
```

```
Einleitung  
-----
```

```
Dies ist eine Beispielseite mit Fettdruck, Kursiv, und Links wie  
`Python <https://www.python.org>`_.
```

```
.. note::
```

```
    Das hier ist ein Hinweisblock.
```

```
.. code-block:: python
```

```
    def greet(name):  
        return f"Hallo {name}"
```

Markdown ist völlig ausreichend für eure Dokumentation.



QA|WARE

- Wer es doch etwas fancy möchte, findet hier Referenzen zu *.rst:
 - <https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html>
 - <https://docutils.sourceforge.io/docs/user/rst/quickref.html#tables>
- Für verschiedene Sphinx Themes, gibt es hier eine Galerie:
 - <https://sphinx-themes.org/>
- **Wichtig:** Einige Extensions (u.a. openapi-Extension) funktionieren am besten mit *.rst



QA|WARE

e2e Testing Patterns & Best Practices

Es gibt keine einheitliche Benennung bei Testarten.

RECAP



QA|WARE

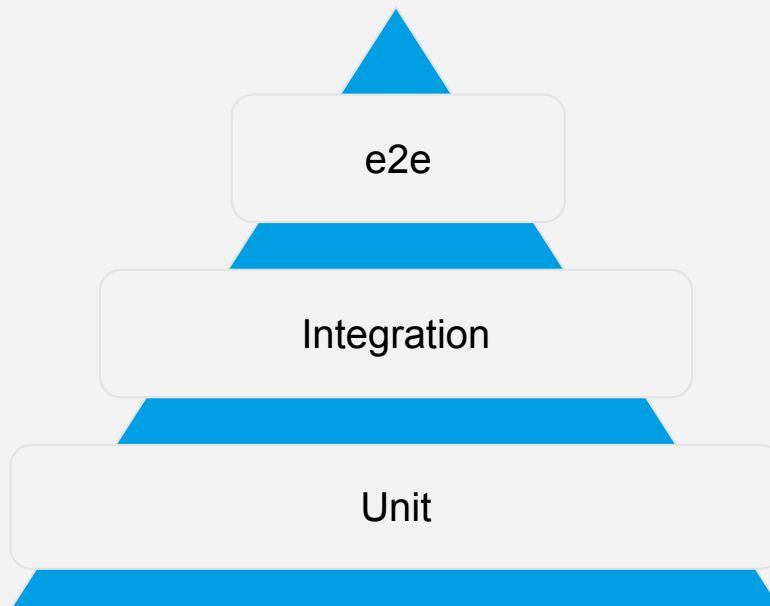
- In diesem Kurs meinen wir mit e2e Tests, die Tests, **die die Funktionalität der Anwendung von der Eingabe über die UI bis hin zur Speicherung im Backend in bpsw. einer Datenbank testen**
- Dazu muss die gesamte Anwendung hochgefahren sein (Frontend + Backend)
- e2e Tests sind damit besonders “teuer” im Vergleich mit anderen Tests
- Es sollten daher nur kritische fachliche **Flows** abgetestet werden

e2eTests im Kontext der Testpyramide.

RECAP



QA|WARE



e2e sind teuer.



QA|WARE

**Bei e2e Tests sollte man immer das
Kosten-Nutzen-Verhältnis im Blick haben.
Testet daher wirklich nur die notwendigen
fachlichen Abläufe ab!**

Container-Presenter-Pattern für bessere Testbarkeit des Frontends.



QA|WARE

■ **Container**-Komponenten (smart):

- Verantwortlich für die Logik, Datenbeschaffung und das State-Management
- Kommunizieren **als einzige** mit APIs, Stores und Routern
- Geben Daten und Callbacks an Presenter weiter

■ **Presenter**-Komponente (dumb):

- Reine UI-Komponente (z.B. Forms, Listen, etc.)
- Bekommen props, rufen callbacks auf
- Keine eigene Logik oder Seiteneffekte

Container-Presenter-Pattern hat viele Vorteile.



QA|WARE

- Dumb Komponenten können isoliert getestet werden (z.B. durch Unit Tests)
- Dumb Komponenten lassen sich mehrfach für verschiedene Daten verwenden
- Trennung von Logik und Darstellung erleichtert Verständnis und Debugging
- Die Smart-Komponenten können einfach in Integrationstests gemockt werden

Beispiel in React: Container.



QAWARE

```
import { useState } from 'react';
import { TodoList } from './TodoList';

export const TodoContainer = () => {
  const [todos, setTodos] = useState<string[]>([]);

  const addTodo = (todo: string) => setTodos([...todos, todo]);

  return <TodoList todos={todos} onAdd={addTodo} />;
};
```

Beispiel in React: Presenter.



QA|WARE

```
type TodoListProps = {
  todos: string[];
  onAdd: (todo: string) => void;
};

export const TodoList = ({ todos, onAdd }: TodoListProps) => (
  <>
    <ul>{todos.map(t => <li key={t}>{t}</li>)}</ul>
    <button onClick={() => onAdd('Neue Aufgabe')}>Hinzufügen</button>
  </>
);
```



QA|WARE

Playwright <3

Playwright ist ein mächtiges Tool für e2e Tests.



QA|WARE

- **Cross-Browser:** Chromium, Firefox, Webkit
- **Multi-Platform:** Linux, Windows, macOS
- **Backed by Microsoft:** Hohe Maintenance
- **Smart Waiting:** Wartet automatisch auf die Synchronisation von verschiedenen UI-Zuständen
- **Tracing & Debugging:** Super mächtiges Tracing-Tool
- **Testparallelisierung out-of-the-box**
- Eignet sich darüber hinaus für die Generierung von Nutzerdokumentation

Aber wie sieht jetzt so ein Test aus?



QA|WARE

```
import { test, expect } from '@playwright/test';
import { LoginPage } from '../pages/LoginPage';

test('Benutzer kann sich erfolgreich einloggen', async ({ page }) => {
  const loginPage = new LoginPage(page);

  await loginPage.login('testuser', 'geheim123');

  // Erwartung: Weiterleitung auf Dashboard
  await expect(page).toHaveURL(/.*dashboard/);
  await expect(page.getByText('Willkommen')).toBeVisible();
});
```

Page Object Pattern als Teststruktur.



QA|WARE

```
class LoginPage {  
    constructor(private page: Page) {}  
  
    async login(user, pass) {  
        await this.page.fill('#username', user);  
        await this.page.fill('#password', pass);  
        await this.page.click('button[type="submit"]');  
    }  
}
```

Page Object Pattern als Teststruktur.



QA|WARE

- Baue eine eigene Page-Klasse pro UI Seite
- Klasse enthält alle Funktionen, die auf dieser Seite auch für die Nutzer:innen möglich sind
- Im Testcode wird dann dieses Page-Objekt verwendet
- Page-Objekt Klasse kapselt dann auch alle UI Selektoren, die für die Seite wichtig sind
- So lassen sich spätere Änderungen einfach und zentral umsetzen

Für Tests sollten bestimmte Selektoren verwendet werden.



QA|WARE

Selektor	Beispiel	Vorteile
<code>getByRole()</code>	<code>page.getByRole('button')</code>	robust und semantisch
<code>getByLabel()</code>	<code>page.getByLabel('Benutzername')</code>	barrierefrei und gut lesbar
<code>getByText()</code>	<code>page.getByText('Login')</code>	klarer Bezug zum Nutzer
<code>data-testid</code>	<code>page.locator('[data-testid="x"]')</code>	für gezielten Zugriff

Nicht verwenden solltet ihr komplizierte Selektoren, wie bpsw. komplexe CSS Pfade: `.foo > .bar:nth-child(2)`



QA|WARE

Demo-Time