

Testkonzept für die Checkout-Seite

1. Einleitung

Dieses Testkonzept beschreibt die Teststrategie für eine Checkout-Seite.
Ziel ist es, eine hohe Testabdeckung mit guter Flexibilität und Wartbarkeit zu erreichen.

2. Testumfang

2.1 Funktionale Bereiche

A) Kundendaten-Eingabe

- **E-Mail-Adresse:**
 - Pflichtfeld-Validierung
 - Format-Validierung (gültige E-Mail-Struktur)
 - Fehlermeldungen bei ungültigen Eingaben
- **Anrede:**
 - Auswahl Herr/Frau
- **Persönliche Daten:**
 - Vorname & Nachname (Pflichtfelder)
 - Straße, Hausnummer, PLZ, Ort (Pflichtfelder)
 - Land-Auswahl
- **Telefonnummer:** Optional, Format-Validierung

B) Zahlungsarten

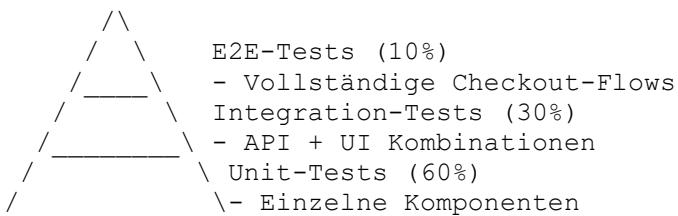
- **Verfügbare Optionen:**
 - Kreditkarte (mit Eingabeformular)
 - SEPA-Lastschrift (mit IBAN-Eingabe)
 - Überweisung (Vorkasse)
- **Zahlungsart-Wechsel:** Dynamische Anpassung der Eingabefelder
- **Button „Jetzt zahlen“:** Wird bei jeder Zahlungsart angezeigt
- **Zahlungsspezifische Validierung:**
 - Kreditkarte:
 - Karteninhaber
 - Kartenzahlung
 - CVV
 - Ablaufdatum
 - SEPA:
 - Kontoinhaber
 - IBAN-Format

C) Bestellabschluss

- **"Zahlungspflichtig bestellen"-Button:**
 - Aktivierung nur bei vollständigen Daten
 - Disabled-State bei fehlenden Angaben
- **Zusammenfassung:** Alle Daten nochmal anzeigen
- **API-Integration:** Erfolgreiche Transaktion erstellen
- **Weiterleitungen:**
 - Success-Seite bei erfolgreicher Zahlung
 - Error-Seite bei Fehlern
 - Failure-Seite bei Abbruch

3. Testarten und Organisation

3.1 Testpyramide



3.2 Unit-Tests (60% der Tests)

Fokus: Einzelne Komponenten isoliert testen

- Validierungsfunktionen (E-Mail, IBAN, Kreditkarte)
- Button-Enable/Disable-Logik

Vorteile:

- Sehr schnell (Millisekunden)
- Einfach zu debuggen
- Hohe Testabdeckung möglich

3.3 Integration-Tests (30% der Tests)

Fokus: Zusammenspiel mehrerer Komponenten

- Formular-Validierung + Fehlermeldungsanzeige
- API-Calls + Response-Handling

Test-Daten: Verwenden von Testdatenbanken oder Mocks

3.4 End-to-End Tests (10% der Tests)

Fokus: Komplette User-Journeys

- Happy Path: Vollständiger Checkout mit Kreditkarte
- Alternative: Vollständiger Checkout mit SEPA
- Error-Handling: Ungültige Zahlungsdaten
- Abbruch-Szenarien

4. Teststrategie für hohe Testabdeckung

4.1 Äquivalenzklassen-Methode

Eingabefelder in Gruppen testen:

Feld	Gültige Klasse	Ungültige Klassen
E-Mail	user@domain.com	user@, @domain, user, leer
PLZ	12345	1234, 123456, abcde, leer
Kreditkarte	4635440000002298	123, abcd, 15 Ziffern
IBAN	DE37503240001000000524	DE123, 12345, XX37...

Vorteil: Reduziert Testfälle, ohne Abdeckung zu verlieren

4.2 Grenzwertanalyse

Teste Grenzwerte bei numerischen Eingaben:

- Menge: 0, 1, 999, 1000 (falls Maximum)
 - Preis: 0,00€, 0,01€, 9999999,99€
 - Textfelder: Leer, 1 Zeichen, Maximum Zeichen, Maximum+1
-

4.3 Zustandsbasiertes Testen

Checkout-Zustände:

1. **Daten-Eingabe:** Teilweise ausgefüllt
2. **Vollständig:** Alle Pflichtfelder ausgefüllt
3. **Validierung:** Fehler angezeigt
4. **Bereit:** Button aktiviert
5. **Verarbeitung:** API-Call läuft
6. **Abgeschlossen:** Success/Error

4.4 Entscheidungstabellen

Jetzt Zahlen-Button-Aktivierung (Beispiel):

Bedingung	Test 1	Test 2	Test 3	Test 4
Alle Pflichtfelder?	✓	✓	X	✓
AGB akzeptiert?	✓	X	✓	✓
Zahlungsart gewählt?	✓	✓	✓	X
Button aktiv?	✓	X	X	X

5. Test-Organisation und Struktur

5.1 Page Object Model (POM)

Vorteile:

- Wiederverwendbarkeit
- Einfache Wartung bei UI-Änderungen
- Klare Trennung von Test-Logik und UI-Interaktion

Struktur:

```
tests/
  └── page_objects/
      ├── checkout_page.py
      ├── cart_component.py
      ├── payment_component.py
      └── customer_form_component.py
  └── test_cases/
      ├── test_checkout_happy_path.py
      ├── test_payment_methods.py
      ├── test_validation.py
      └── test_cart_operations.py
  └── test_data/
      ├── valid_test_data.json
      └── invalid_test_data.json
  └── helpers/
      ├── api_helper.py
      └── data_generator.py
```

Beispiel Page Object:

```
class CheckoutPage:
    def __init__(self, driver):
        self.driver = driver
        self.email_input = (By.ID, "email")
        self.submit_button = (By.ID, "submit-order")

    def enter_email(self, email):
        self.driver.find_element(*self.email_input).send_keys(email)

    def click_submit(self):
        self.driver.find_element(*self.submit_button).click()

    def is_submit_enabled(self):
        return self.driver.find_element(*self.submit_button).is_enabled()
```

5.2 Test-Daten-Management

Zentralisierte Test-Daten:

```
{  
    "valid_customer": {  
        "email": "test@example.com",  
        "firstName": "Max",  
        "lastName": "Mustermann",  
        "street": "Teststraße",  
        "houseNumber": "123",  
        "zipCode": "12345",  
        "city": "Berlin"  
    },  
    "valid_credit_card": {  
        "number": "4635440000002298",  
        "cvv": "123",  
        "expiry": "12/25"  
    }  
}
```

Vorteile:

- Einfache Aktualisierung
 - Konsistenz über alle Tests
 - Schnelle Anpassung bei Datenänderungen
-

5.3 Test-Kategorisierung

Test-Suites:

- **Smoke-Tests:** 5-10 kritische Tests (~5 Minuten)
- **Regression-Suite:** Alle wichtigen Funktionen (~30 Minuten)
- **Full-Suite:** Alle Tests inklusive Edge-Cases (~2 Stunden)

6. Testfälle nach Priorität

6.1 Kritisch (P0) - Müssen immer funktionieren

1. Vollständiger Checkout mit Kreditkarte (Happy Path)
2. Vollständiger Checkout mit SEPA (Happy Path)
3. Vollständiger Checkout mit Überweisung (Happy Path)

6.2 Hoch (P1) - Wichtige Funktionen

1. Alle Zahlungsarten funktionieren
2. Formular-Validierungen zeigen Fehler
3. Fehlerbehandlung bei API-Fehlern

6.3 Mittel (P2) - Sollten funktionieren

1. Länderwechsel aktualisiert Format
2. Links öffnen in neuem Tab
3. Browser-Back-Button funktioniert korrekt

6.4 Niedrig (P3) - Nice-to-have

1. Animationen funktionieren
2. Autocomplete funktioniert
3. Performance unter Last

7. Spezifische Testszenarien

7.1 Zahlungsart: Kreditkarte

Positive Tests:

- ✓ Gültige Kreditkartennummer wird akzeptiert
- ✓ Zahlung wird erfolgreich verarbeitet
- ✓ Weiterleitung zur Success-Seite

Negative Tests:

- ✗ Ungültige Kartennummer zeigt Fehler
- ✗ Abgelaufene Karte wird abgelehnt
- ✗ Falscher CVV zeigt Fehler
- ✗ Nicht unterstützte Kartentypen werden abgelehnt

Testdaten:

- Nummer: 4635 4400 0000 2298
 - CVV: beliebig (3 Ziffern, z.B. 123)
 - Ablaufdatum: beliebig (zukünftig, z.B. 12/26)
-

7.2 Zahlungsart: SEPA-Lastschrift

Positive Tests:

- ✓ Gültige IBAN wird akzeptiert
- ✓ IBAN wird formatiert angezeigt (Leerzeichen)
- ✓ Zahlung wird initiiert

Negative Tests:

- ✗ Ungültige IBAN-Prüfziffer
- ✗ IBAN zu kurz/lang
- ✗ Nicht-deutsche IBAN (falls nur DE unterstützt)

Testdaten:

- IBAN: DE37503240001000000524
- BIC: FTSBDEFAXXX

7.3 Validierungs-Tests

E-Mail-Validierung:

Eingabe	Erwartet	Grund
test@example.com	✓ Akzeptiert	Gültige E-Mail
test@example	X Fehler	Fehlende Domain
@example.com	X Fehler	Fehlender Lokalpart
test@@example.com	X Fehler	Doppeltes @
(leer)	X Fehler	Pflichtfeld

PLZ-Validierung (deutsche PLZ):

Eingabe	Erwartet	Grund
12345	✓ Akzeptiert	Gültige PLZ
1234	X Fehler	Zu kurz
123456	X Fehler	Zu lang
abcde	X Fehler	Keine Ziffern
(leer)	X Fehler	Pflichtfeld

8. API-Integration Testing

8.1 Test-Flow mit API

Setup:

1. Authentifizierung (OAuth Token holen)
2. Transaktion erstellen (POST /Smart/Transactions)
3. UI-Test durchführen
4. Status prüfen (GET Transaction Status)
5. Zahlung prüfen (GET Payment Status)

Assertions:

- Transaction ID wird korrekt zurückgegeben
- Status wechselt von "pending" zu "approved"
- Beträge stimmen überein
- Fehlercodes werden korrekt behandelt

9. Test-Umgebungen

9.1 Umgebungs-Strategie

Umgebung	Zweck	Daten	API
Lokal	Entwicklung	Mock/Stub	Mock
Test	Automatisierte Tests	Test-Daten	Test-API
Staging	Manuelle Tests	Staging-Daten	Staging-API
Production	Smoke-Tests	Prod-Daten	Prod-API

9.2 Konfiguration

Config-File-Ansatz (config.json):

```
{
  "test": {
    "baseUrl": "https://connect-testing.secuconnect.com",
    "clientId": "9da410fe2fe7077a9e040f687ae83f84",
    "clientSecret": "80d8d7785e55e6a0852935396674ed582daf04ff1f038d48a461c94f5219f282"
  },
  "staging": {
    "baseUrl": "https://connect-staging.secuconnect.com",
    "clientId": "staging_client_id",
    "clientSecret": "staging_secret"
  }
}
```

10. Test-Automatisierung Pipeline

10.1 CI/CD Integration

Empfohlener Flow:

Code Push → Build → Unit Tests → Integration Tests → E2E Tests → Deploy
(1min) (2min) (10min) (30min)

Failure-Strategie:

- Unit-Tests fehlgeschlagen → Kein Build
 - Integration-Tests fehlgeschlagen → Kein Deploy zu Staging
 - E2E-Tests fehlgeschlagen → Kein Deploy zu Production
-

10.2 Test-Reporting

Was messen:

- Test-Abdeckung (Code Coverage)
- Pass/Fail-Rate
- Durchschnittliche Testdauer
- Flaky Tests (instabile Tests identifizieren)

Tools:

- Selenium
- Python
- Allure Report (HTML-Reports)

11. Wartbarkeit und Best Practices

11.1 Wartbare Tests schreiben

DRY-Prinzip (Don't Repeat Yourself):

```
# ❌ Schlecht
def test_checkout_visa():
    driver.find_element(By.ID, "email").send_keys("test@test.com")
    driver.find_element(By.ID, "firstname").send_keys("Max")
    # ... 20 weitere Zeilen ...

def test_checkout_mastercard():
    driver.find_element(By.ID, "email").send_keys("test@test.com")
    driver.find_element(By.ID, "firstname").send_keys("Max")
    # ... 20 weitere Zeilen ... (Wiederholung!)

# ✅ Gut
def fill_customer_form(customer_data):
    # Wiederverwendbare Funktion
    pass

def test_checkout_visa():
    fill_customer_form(valid_customer)
    select_payment("visa")
    complete_checkout()

def test_checkout_mastercard():
    fill_customer_form(valid_customer)
    select_payment("mastercard")
    complete_checkout()
```

11.2 Explizite Waits statt Sleeps

```
# ❌ Schlecht (langsam)
time.sleep(5) # Hoffen, dass Seite geladen ist

# ✅ Gut (wartet nur so lange wie nötig)
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.ID, "submit-button"))
)
```

11.3 Aussagekräftige Test-Namen

```
# X Schlecht
def test_1():
    pass

def test_checkout():
    pass

# ✓ Gut
def test_checkout_with_valid_credit_card_succeeds():
    pass

def test_checkout_with_invalid_email_shows_error():
    pass

def test_submit_button_disabled_when_agb_not_accepted():
    pass
```

12. Testmetriken und KPIs

12.1 Wichtige Metriken

Metrik	Zielwert	Bedeutung
Code Coverage	>80%	Anteil getesteter Code
Test Success Rate	>95%	Stabile Tests
Execution Time	<30min	Schnelles Feedback
Defect Detection	>90%	Tests finden Bugs
Flaky Test Rate	<5%	Zuverlässigkeit

12.2 ROI (Return on Investment) von Tests

Was wird verhindert:

- Checkoutfehler (sehr teuer!)
- Manuelle Test-Zeit (skaliert nicht)
- Regressions-Bugs (Vertrauen in Releases)
- Dokumentation (Tests als Spezifikation)

13. Zusammenfassung: Testabdeckung erreichen

13.1 Checkliste für vollständige Abdeckung

Funktional:

- [] Alle Eingabefelder validiert (gültig + ungültig)
- [] Alle Buttons funktionieren
- [] Alle Zahlungsarten getestet
- [] Alle Fehlermeldungen erscheinen
- [] Alle Weiterleitungen funktionieren

Non-Funktional:

- [] Performance: Seite lädt in <3 Sekunden
- [] Responsiv: Mobile, Tablet, Desktop
- [] Browser: Chrome, Firefox, Safari, Edge
- [] Accessibility: WCAG 2.1 AA konform
- [] Security: Keine sensiblen Daten in URL/Logs

API-Integration:

- [] Success-Flow
 - [] Error-Handling
 - [] Timeout-Verhalten
 - [] Retry-Logik
-

13.2 Flexibilität durch Modularität

Vorteile des modularen Ansatzes:

1. **Neue Zahlungsart hinzufügen:** Nur neue Komponente + Tests
2. **UI-Änderung:** Nur Page Object anpassen
3. **API-Änderung:** Nur API-Helper anpassen
4. **Test-Daten ändern:** Nur JSON-Datei aktualisieren

Lose Kopplung: Tests hängen nicht voneinander ab, können einzeln oder parallel laufen.