LQMS

1) Unit-Tests

Unit-Tests, auch Modultests genannt, bezeichnen ein beliebtes und grundlegendes Verfahren in der Software- und Webentwicklung.

Hierbei werden einzelne, kleinere Komponenten, abgegrenzt vom Rest der Anwendung auf eine korrekte Funktionsweise überprüft, um die Fehlerfindung und Behebung zu erleichtern.

In der Gruppierung der Tests, handelt es sich bei den Unit-Tests um die schnellsten und isoliertesten, was die Implementierungsmöglichkeiten deutlich erhöht.

Exemplarisch wurde in LQMS, eine Unit-Test-Gruppe geschrieben, welche den Zugriff auf die Datenbank während des Ladevorgangs für das Dashboard testet. Dabei werden randomisierte “Tipps und Tricks” zum besseren Lernen aus einer Datenbank-Tabelle ausgewählt und dem Nutzer, über das Dashboard angezeigt. Für die Implementierung des Tests wurde ein Standardwert in der Datenbank und eine voreingestellte ID genutzt. Dabei wird durch Abfragen der Tabelle mit der voreingestellten ID “1” eine exakte Rückgabe, an Stelle einer randomisierten, erwartet, was die Funktionsweise dessen prüft. Für eine leere Rückgabe muss das Programm eine Fehlermeldung zurückgeben, was ebenfalls geprüft werden muss. Dazu wird ein leeres Array, in der Struktur der Rückgabe, im Code simuliert, sodass diese Funktion geprüft wird.

Implementierung zu Unit-Test-Gruppe – 1

- **Code 1:** (Anhang)

Eine weitere Unit-Test-Gruppe, die implementiert worden ist, wird genutzt, um die Speicherung der Daten, die aus einer “Lernsession” resultieren, in der Datenbank zu überwachen. Diese Daten werden teils während der Session und teils nach Session-Ende erhoben und sollen für spätere Auswertung in der Datenbank gespeichert werden. Dabei wird diese durch “jest.mock” simuliert und geprüft, ob der Server auf einen Datenbankfehler konkret mit dem Fehlerstatus 500 reagiert. Außerdem wird überprüft, ob eine Erfolgsmeldung bei erfolgreicher Speicherung zurückgegeben wird. Dadurch stellt die Test-Gruppe allgemein sicher, dass die Session-Daten richtig verarbeitet und Fehler sauber behandelt werden.

Implementierung zu Unit-Test-Gruppe – 2

- **Code 2:** (Anhang)

2) Integration-Test

Bei einem Integration-Test werden mehrere voneinander abhängige Komponenten einer Anwendung hintereinander getestet, um zu prüfen, wie effizient die verschiedenen Aspekte einer Anwendung miteinander arbeiten. Dies ist von besonderer Wichtigkeit bei Projekten mit mehreren Komponenten, welche zeitgleich an einem Programm arbeiten.

Als Beispiel hierfür wurde der Login-Prozess durch Tests nachgestellt:

Zunächst werden Abbilder der zu verwendenden Bibliotheken benutzt und Variablen für die Tests angelegt. Der Test selbst umfasst 5 kleinere Tests, welche die möglichen Fehlschläge und ebenso den Erfolgsfall abbilden. Konkret lauten die Testfälle: **Ungültige Eingaben**, **Ungültiger Benutzer**, **Ungültiges Passwort**, **Login Erfolgreich** und **Serverfehler**.

1.) **Ungültige Eingaben**: Die Eingabe(n) ist/sind *zu kurz*.
→ Benutzername / E-Mail muss mindestens 2 Zeichen enthalten.

2.) **Ungültiger Benutzer**: Der Benutzer existiert nicht.
→ Benutzername / E-Mail existiert in der Datenbank *nicht*.

3.) **Ungültiges Passwort**: Falsches Passwort zu gültigem Benutzer angegeben.
→ Benutzer existiert in der Datenbank, aber das Passwort ist falsch.

4.) **Login erfolgreich**: Gültiges Passwort zu einem gültigen Benutzer angegeben.
→ Benutzer existiert in der Datenbank mit angegebenem Passwort.

5.) **Serverfehler**: Ein unbekannter Fehler aus der Datenbank
→ Ein unerwarteter Fehler ist aufgetreten / Die Datenbank ist nicht mehr erreichbar

Implementierung des Integration-Test- **Code 3:** (Anhang)

3) Szenario-Test

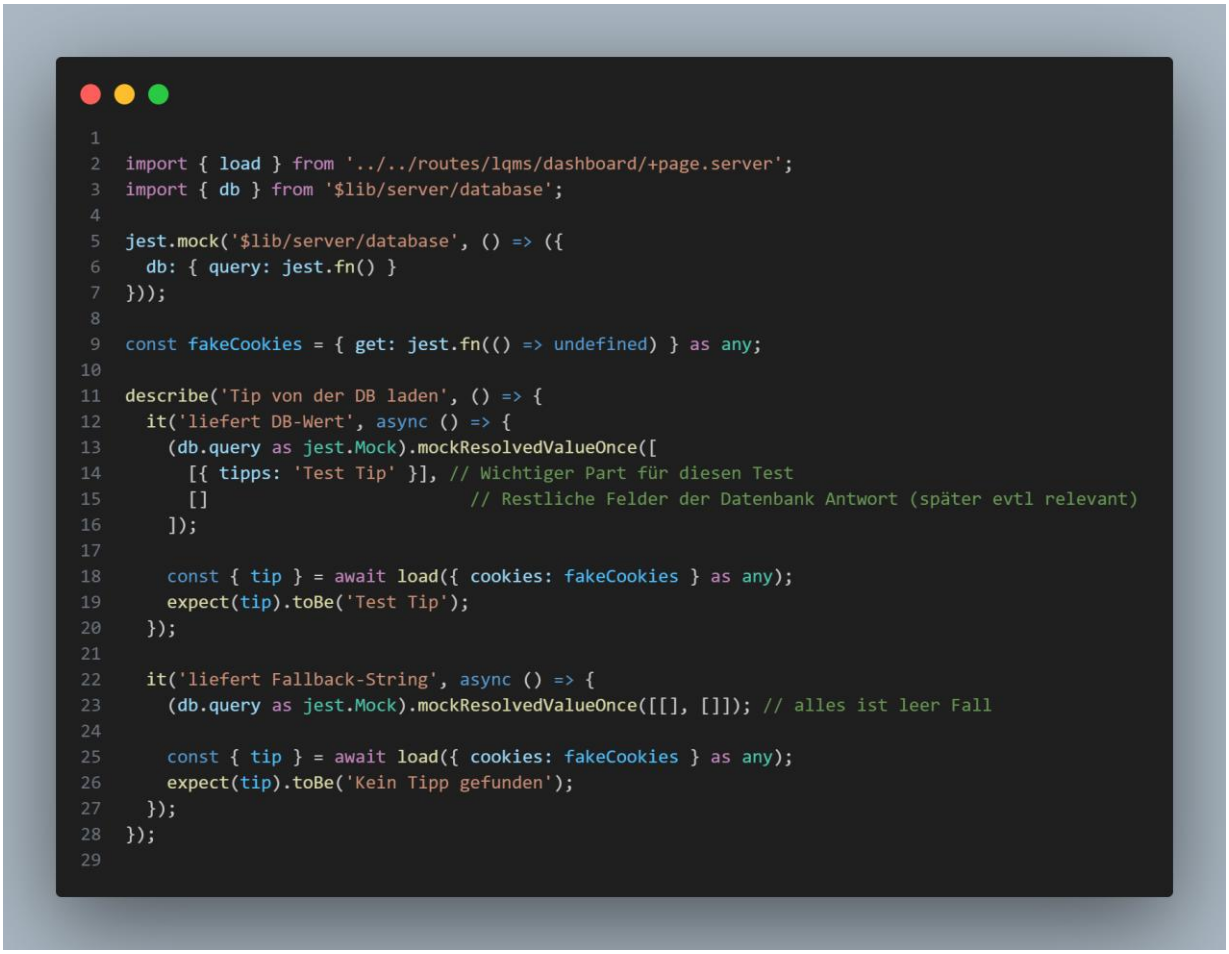
Ein Szenario-Test versucht ein Nutzungs-Szenario der Anwendung abzubilden und dabei alle möglichen Aktionen zu berücksichtigen. Ein gutes Beispiel hierfür ist die Registrierung eines neuen Nutzerkontos. Dabei müssen nacheinander folgende Schritte geprüft werden. Zu Beginn erfolgt die Erstellung eines neuen Nutzers bei der es dann die Anforderungen an die Passwortlänge und die verwendeten Zeichen zu prüfen gilt. Dies kann mit mehreren simulierten Anfragen an die Registrierungs-API geprüft werden, um sämtliche Fehler nachstellen zu können. Analog dazu wird für die E-Mail-Adresse dieselbe Methode genutzt.

Danach erfolgt der Datenbank-Zugriff, wobei geprüft wird, ob die E-Mail-Adresse eines neu anzulegenden Nutzers bereits in der bestehenden Sammlung von Nutzerdaten vorhanden ist. Im nächsten Schritt wird das eingegebene Passwort mithilfe von “bcrypt” gehasht und in die Datenbank eingetragen. Um diesen Prozess zu überprüfen, sollte ein Abruf der neu angelegten Spalte erfolgen, um die Speicherung der Daten zu verifizieren. Zuletzt sollten die im Laufe des Registrierungs-Prozesses, an den Nutzer weitergegebenen, visuellen Ausgaben, ebenfalls getestet werden. Dies wäre wichtig, um sicherzustellen, dass der Nutzer keine überflüssigen Informationen erhält, welche das Nutzer-Erlebnis verschlechtern oder sogar ein Sicherheitsrisiko darstellen könnten.

Tests, die nacheinander ablaufen sollten, um den Registrierungsprozess möglichst realistisch abzubilden:

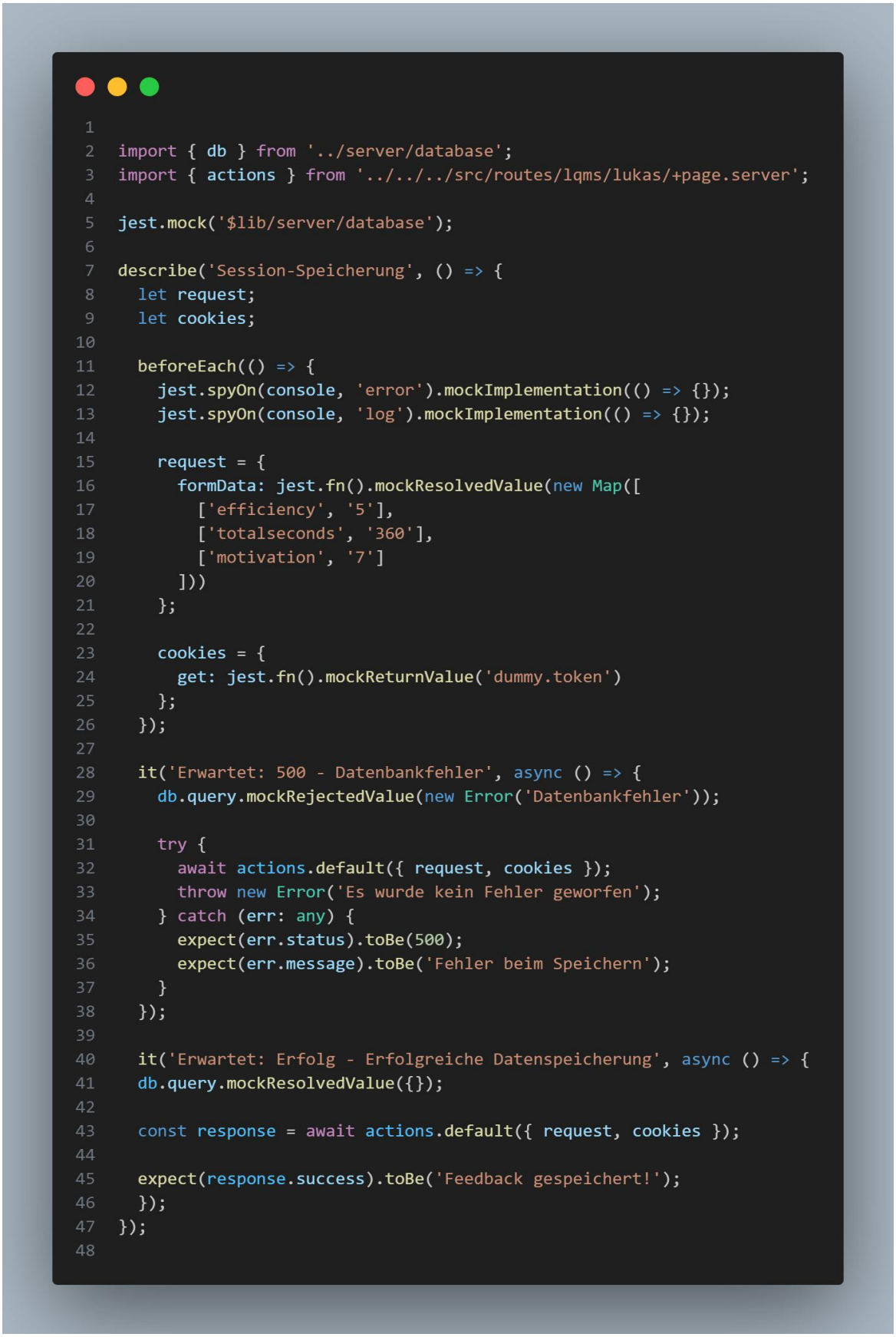
- Eingaben durch den Nutzer
  - Prüfen, ob Bedingungen für Passwort-Länge und Inhalt erfüllt sind
  - Prüfen, ob E-Mail-Format korrekt ist
  - Prüfen, ob E-Mail bereits für ein anderes Nutzerkonto genutzt wurde
- Speichern in der Datenbank
  - Dafür Passwort hashen
  - In Datenbank eintragen
  - Auslesen, ob der Wert eingetragen wurde
- Rückgabe

Rückmeldung an den Nutzer in jedem Teil des Prozesses



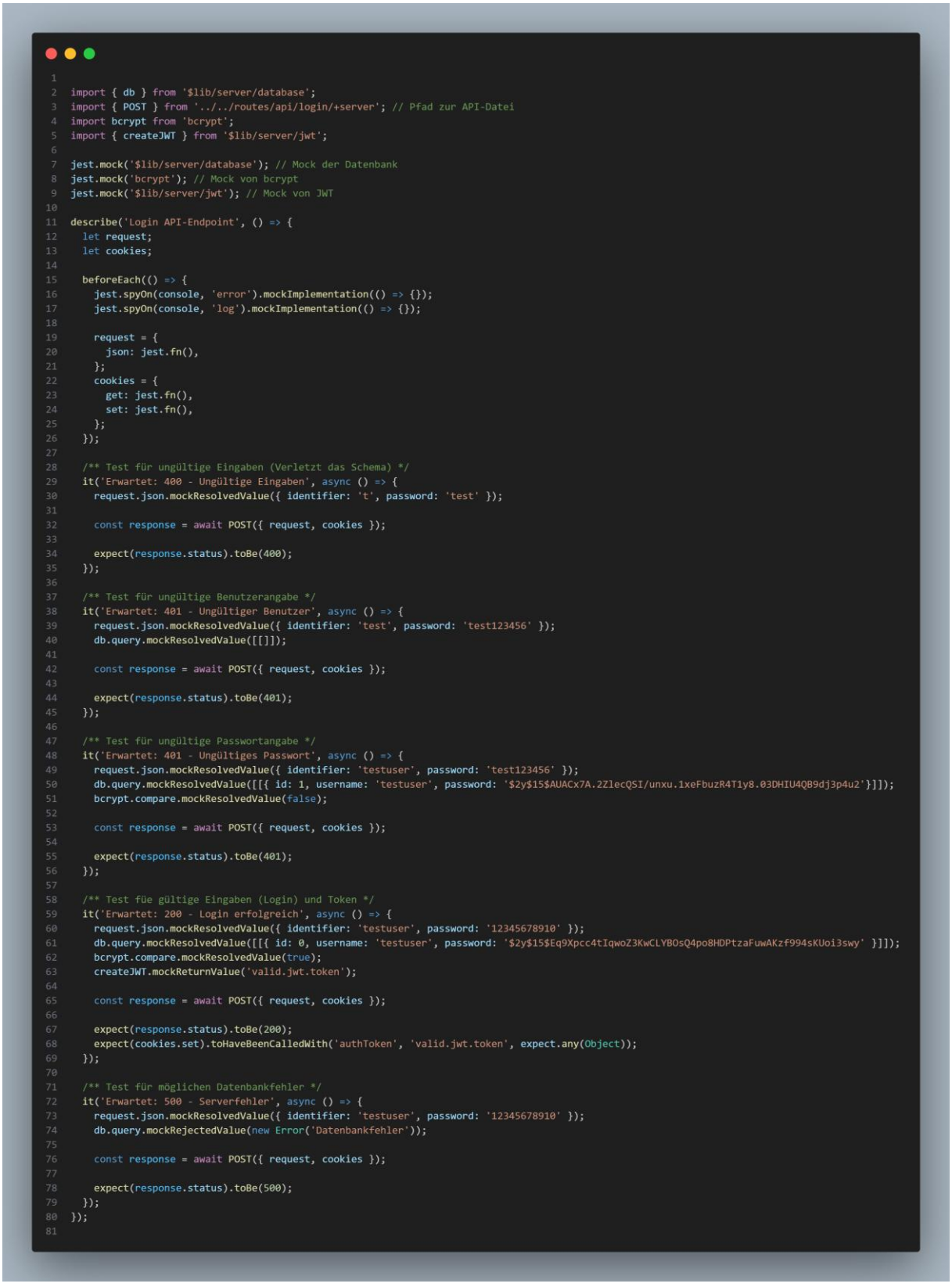
```
1
2  import { load } from '../../routes/lqms/dashboard/+page.server';
3  import { db } from '$lib/server/database';
4
5  jest.mock('$lib/server/database', () => ({
6    db: { query: jest.fn() }
7  }));
8
9  const fakeCookies = { get: jest.fn(() => undefined) } as any;
10
11 describe('Tip von der DB laden', () => {
12   it('liefert DB-Wert', async () => {
13     (db.query as jest.Mock).mockResolvedValueOnce([
14       [{ tipps: 'Test Tip' }], // Wichtiger Part für diesen Test
15       []                       // Restliche Felder der Datenbank Antwort (später evtl relevant)
16     ]);
17
18     const { tip } = await load({ cookies: fakeCookies } as any);
19     expect(tip).toBe('Test Tip');
20   });
21
22   it('liefert Fallback-String', async () => {
23     (db.query as jest.Mock).mockResolvedValueOnce([[], []]); // alles ist leer Fall
24
25     const { tip } = await load({ cookies: fakeCookies } as any);
26     expect(tip).toBe('Kein Tipp gefunden');
27   });
28 });
29
```

Unit-Test-Gruppe – 1: Test für den Datenbank-Zugriff auf der Dashboard Seite



```
1
2  import { db } from '../server/database';
3  import { actions } from '../../../src/routes/lqms/lukas/+page.server';
4
5  jest.mock('$lib/server/database');
6
7  describe('Session-Speicherung', () => {
8    let request;
9    let cookies;
10
11   beforeEach(() => {
12     jest.spyOn(console, 'error').mockImplementation(() => {});
13     jest.spyOn(console, 'log').mockImplementation(() => {});
14
15     request = {
16       formData: jest.fn().mockResolvedValue(new Map([
17         ['efficiency', '5'],
18         ['totalseconds', '360'],
19         ['motivation', '7']
20       ]))
21     };
22
23     cookies = {
24       get: jest.fn().mockReturnValue('dummy.token')
25     };
26   });
27
28   it('Erwartet: 500 - Datenbankfehler', async () => {
29     db.query.mockRejectedValue(new Error('Datenbankfehler'));
30
31     try {
32       await actions.default({ request, cookies });
33       throw new Error('Es wurde kein Fehler geworfen');
34     } catch (err: any) {
35       expect(err.status).toBe(500);
36       expect(err.message).toBe('Fehler beim Speichern');
37     }
38   });
39
40   it('Erwartet: Erfolg - Erfolgreiche Datenspeicherung', async () => {
41   db.query.mockResolvedValue({});
42
43   const response = await actions.default({ request, cookies });
44
45   expect(response.success).toBe('Feedback gespeichert!');
46   });
47 });
48
```

Code 2: Unit-Test-Gruppe – 2: Korrekte Feedbackspeicherung nach Ende der Lernsession



```
import { db } from '$lib/server/database';
import { POST } from '../../routes/api/login/+server'; // Pfad zur API-Datei
import bcrypt from 'bcrypt';
import { createJWT } from '$lib/server/jwt';

jest.mock('$lib/server/database'); // Mock der Datenbank
jest.mock('bcrypt'); // Mock von bcrypt
jest.mock('$lib/server/jwt'); // Mock von JWT

describe('Login API-Endpoint', () => {
  let request;
  let cookies;

  beforeEach(() => {
    jest.spyOn(console, 'error').mockImplementation(() => {});
    jest.spyOn(console, 'log').mockImplementation(() => {});

    request = {
      json: jest.fn(),
    };
    cookies = {
      get: jest.fn(),
      set: jest.fn(),
    };
  });

  /** Test für ungültige Eingaben (Verletzt das Schema) */
  it('Erwartet: 400 - Ungültige Eingaben', async () => {
    request.json.mockResolvedValue({ identifier: 't', password: 'test' });

    const response = await POST({ request, cookies });

    expect(response.status).toBe(400);
  });

  /** Test für ungültige Benutzerangabe */
  it('Erwartet: 401 - Ungültiger Benutzer', async () => {
    request.json.mockResolvedValue({ identifier: 'test', password: 'test123456' });
    db.query.mockResolvedValue([[]]);

    const response = await POST({ request, cookies });

    expect(response.status).toBe(401);
  });

  /** Test für ungültige Passwortangabe */
  it('Erwartet: 401 - Ungültiges Passwort', async () => {
    request.json.mockResolvedValue({ identifier: 'testuser', password: 'test123456' });
    db.query.mockResolvedValue([[{ id: 1, username: 'testuser', password: '$2y$15$AUACx7A.2ZlecQSI/unxu.1xeFbuzR4T1y8.03DHIU4QB9dj3p4u2'}]]);
    bcrypt.compare.mockResolvedValue(false);

    const response = await POST({ request, cookies });

    expect(response.status).toBe(401);
  });

  /** Test füe gültige Eingaben (Login) und Token */
  it('Erwartet: 200 - Login erfolgreich', async () => {
    request.json.mockResolvedValue({ identifier: 'testuser', password: '12345678910' });
    db.query.mockResolvedValue([[{ id: 0, username: 'testuser', password: '$2y$15$Eq9Xpcc4tIqwoZ3KwCLYBOsQ4po8HDPtzaFuwAKzf994sKUoi3swy' }]]);
    bcrypt.compare.mockResolvedValue(true);
    createJWT.mockReturnValue('valid.jwt.token');

    const response = await POST({ request, cookies });

    expect(response.status).toBe(200);
    expect(cookies.set).toHaveBeenCalledWith('authToken', 'valid.jwt.token', expect.any(Object));
  });

  /** Test für möglichen Datenbankfehler */
  it('Erwartet: 500 - Serverfehler', async () => {
    request.json.mockResolvedValue({ identifier: 'testuser', password: '12345678910' });
    db.query.mockRejectedValue(new Error('Datenbankfehler'));

    const response = await POST({ request, cookies });

    expect(response.status).toBe(500);
  });
});
```

Integration-Test: Vollständiger Integrationstest zu den Fällen des Logins