

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Abkürzungsverzeichnis	II
1 Einleitung	1
2 Grundlagen	2
2.1 Test-Driven Development	2
2.2 Ablauf von Test-Driven Development	3
3 Konzeption und Design der Todo-App	6
3.1 Architektur der Anwendung	6
3.2 Datenmodell und Datenbankdesign	7
3.2.1 Entwurf des Datenmodells	7
3.2.2 Projektanforderungen an die Datenbank	9
3.2.3 Datenbankdesign	10
3.3 User Interface Design	11
3.3.1 Gestaltung der Benutzeroberfläche	11
3.3.2 Berücksichtigung von Usability-Prinzipien	12
4 Implementierung unter Verwendung von TDD	16
4.1 Entwicklungsumgebung und Werkzeuge	16
4.2 Implementierungsschritte	17
4.3 Beispieltests und Testfälle	18
4.4 Unit-Tests	18
4.5 Integrationstests	19
4.6 Akzeptanztests	20

Inhaltsverzeichnis

5	Frontend Entwicklung	22
5.1	Projektanforderung an Frontend-Entwicklung	22
5.2	Technologiestack	23
5.3	Projektstruktur	23
5.4	Routing	24
5.5	Authentifizierung und Autorisierung	24
5.6	HTTP-Anfragen	24
5.7	State Management	25
5.8	Styling	25
	Literaturverzeichnis	26

Abbildungsverzeichnis

2.1	schematischer Ablauf des Test-Driven Development	2
2.2	schematischer Ablauf des traditionellen Testen	3
2.3	Phasen des Test-Driven Development	4
2.4	schematischer Ablauf von Test-Driven Development in der Praxis	5
3.1	User Interface Design der Startseite	11
3.2	User Interface Design der Anmeldeseite	12
3.3	User Interface Design der Registrierungsseite	13
3.4	User Interface Design der Hauptseite	14
3.5	User Interface Design der Detailansicht	15

Abkürzungsverzeichnis

TDD Test-Driven Development

1 Einleitung

2 Grundlagen

2.1 Test-Driven Development

Bei dem Test-Driven Development (TDD), zu Deutsch auch „test-getriebene Entwicklung“, handelt es sich um ein Entwicklungs- und Designverfahren für Software, bei dem Testfälle bereits vor oder spätestens parallel zur Implementierung spezifiziert werden. Die zu erstellende Software wird quasi über Tests entworfen. [1, S. 151]

TDD sollte bereits bei der Anforderungsanalyse angewendet werden. Die Anforderungen sind so zu definieren, dass sichergestellt werden kann, dass die Erfüllung dieser Anforderungen mithilfe von Tests validiert werden können. Dies kann durch Testdefinitionen in Textform gewährleistet werden, in die genau die Vorbedingungen, die Ausführung und die zu erwartenden Ergebnisse beschrieben werden. [2, S.188]

Die Abbildung 2.1 zeigt exemplarisch einen schematischen Ablauf des Test-Driven Development-Ansatzes. Am Anfang wird der Testfall definiert (Test Definition, TD), darauf folgt die Umsetzung und Programmierung (Programming, P). Im Anschluss werden die Testfälle ausgeführt (Test Execution, TE). [1, S. 151]

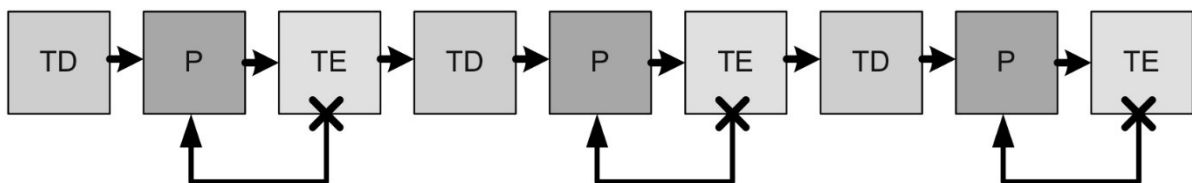


Abbildung 2.1: schematischer Ablauf des Test-Driven Development [1, S. 151]

Im Gegensatz dazu wird beim traditionellen Testen erst parallel zur Entwicklung oder nach Abschluss der Implementierung geeignete Testfälle definiert und ausgeführt, wie ein schematischer Ablauf in Abbildung 2.2 zeigt. [1, S. 150]

2.2. ABLAUF VON TEST-DRIVEN DEVELOPMENT

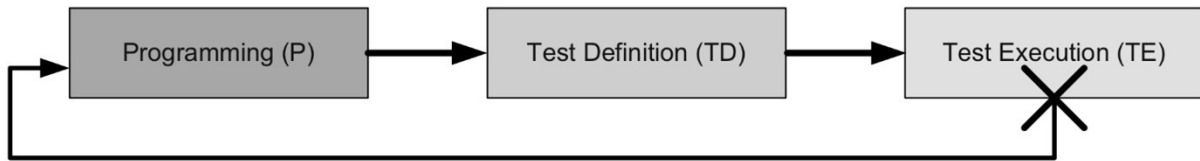


Abbildung 2.2: schematischer Ablauf des traditionellen Testen [1, S.151]

Durch das frühe Definieren und Ausführen von Testfällen im TDD-Ansatz erhalten die Entwickler ein unmittelbares Feedback zur erstellten Lösung und Probleme wie Seiteneffekte können frühzeitig erkannt werden [1, S. 151]. Da der Entwickler sich für die Testerstellung intensiv mit den Anforderungen auseinander setzen muss, erhält er ein verbessertes Verständnis für das zu entwickelnde Produkt und die eigentliche Entwicklungszeit kann deutlich verkürzt werden. Die Anzahl verschleppter und dann aufwändig zu reparierender Fehler kann zudem, durch die von der Entwicklung unabhängigen Erstellung der Testfälle, reduziert werden. Denn, wenn erst nach der Implementierung die Testfälle definiert werden, besteht eine große Wahrscheinlichkeit, dass Denkfehler bei der Implementierung bei der Testerstellung wiederholt werden und somit falsche Tests, die falsche zu testende Software, als korrekt überprüfen [2, S.188].

Die frühzeitig erstellten Tests eignen sich ferner auch zur Automatisierung und des Weiteren kann durch die Testfälle und deren Testergebnisse die Kommunikation verbessert werden. TDD wird meistens im Rahmen der agilen Software-Entwicklung verwendet und ist fester Bestandteil verschiedener Vorgehensmodelle, wie eXtreme Programming und SCRUM [1, S. 151].

2.2 Ablauf von Test-Driven Development

Der entscheidende Bestandteil des Test-Driven Development ist, dass die Testfälle vor oder spätestens parallel zur Implementierung definiert werden. Im Anschluss werden die Softwarekomponenten implementiert bzw. angepasst, bis die definierten Tests erfolgreich durchlaufen. Konkret lässt sich TDD in vier grundlegende Schritte unterteilen, wie sie in Abbildung 2.3 skizziert werden. [1, S. 153]

In dem ersten Schritt „think“ wird die Anforderung ausgewählt, die im nächsten Schritt umgesetzt werden soll. Anhand der ausgewählten Anforderung werden die geeigneten

2.2. ABLAUF VON TEST-DRIVEN DEVELOPMENT

Tests definiert. Hierbei ist sicherzustellen, dass die ausgewählte Anforderung auch tatsächlich durch die Tests abgedeckt wird. Unit-Tests können beispielsweise auf der Implementierungsebene z.B. für Komponenten verwendet werden. [1, S. 153]

In dem zweiten Schritt „red“ werden diese Tests ausgeführt. Da es noch keine Implementierung dazu gibt, schlagen die Tests fehl und sind im Status „red“. [1, S. 153]

Dann erfolgt die Implementierung, bis die erstellten Tests erfolgreich durchlaufen und den Status „green“ erreichen (Schritt drei). Dabei wird die Anforderung schrittweise implementiert und getestet. Wenn die Testfälle nicht erfolgreich durchlaufen, werden Fehler korrigiert, falls die Anforderung bereits umgesetzt wurde oder die Funktionalität implementiert, falls die Anforderung noch nicht umgesetzt wurde. [1, S. 153]

Im letzten Schritt erfolgt die Optimierung und Anpassung des geschriebenen Softwarecodes (vierter Schritt Refactoring). Da durch das Refactoring der Code verändert wird, müssen alle Testfälle im Anschluss jeweils durchgeführt werden, um sicherzustellen, dass alle Tests weiterhin erfolgreich durchlaufen und der Status „green“ nicht mehr verlassen wird. [1, S. 153]

Wenn das Refactoring erfolgreich durchgeführt wurde, ist die Implementierung für die ausgewählte Anforderung abgeschlossen und die nächste Anforderung kann durch die gleiche Schrittfolge umgesetzt werden. [1, S. 153 f.]

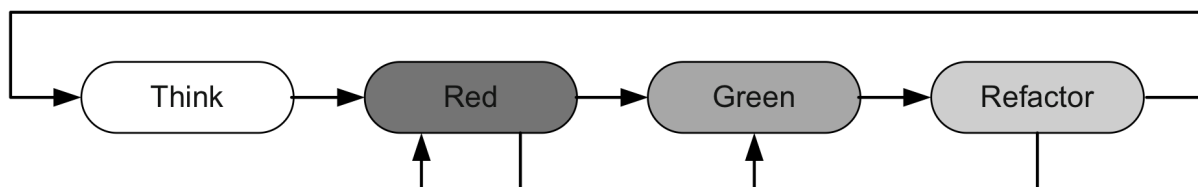


Abbildung 2.3: Phasen Ablauf des Test-Driven Development [1, S. 153]

Die Abbildung 2.4 zeigt einen beispielhaften schematischen Ablaufes von TDD in der Praxis. Dabei wird veranschaulicht, dass für die ausgewählten Anforderungen geeignete Testfälle erstellt werden, um die Erfüllung der Anforderung validieren zu können. In dem Anwendungsbeispiel wird für die Anforderung A drei Testfälle benötigt. Die Testläufe, welche auf der x-Achse dargestellt werden, geben den Status der Testfälle an und wechseln von „red“ in „green“, entsprechend den jeweiligen TDD-Schritten. Dabei wird ersichtlich, dass die Anforderungen schrittweise implementiert und getestet werden. Dieses Beispiel

2.2. ABLAUF VON TEST-DRIVEN DEVELOPMENT

zeigt ebenfalls auf, wie durch die schnelle Rückmeldung, durch die Tests, Seiteneffekte frühzeitig entdeckt werden können. Eine Implementierung, welche für den Testfall C2 bestimmt war, bewirkte, dass nicht nur der Test C2 weiterhin fehlschlägt, sondern auch der Testfall B2. Dieser Testfall wäre wohlmöglich bei einem traditionellen Testanfall erst sehr spät entdeckt worden und müsste aufwendig korrigiert werden. [1, S. 154]

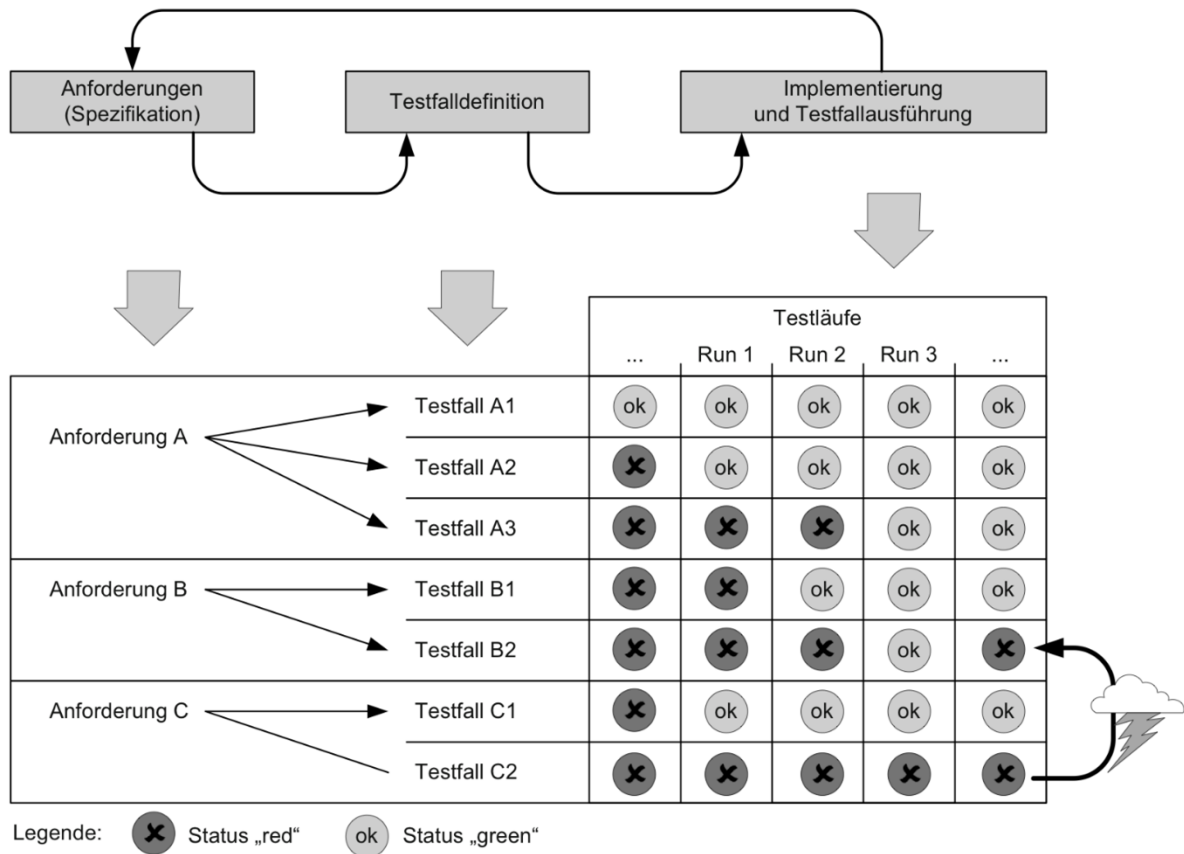


Abbildung 2.4: schematischer Ablauf von Test-Driven Development in der Praxis [1, S. 154]

3 Konzeption und Design der Todo-App

3.1 Architektur der Anwendung

Die Todo-App ist als eine mehrschichtige Architektur konzipiert, die eine klare Trennung der Verantwortlichkeiten zwischen den verschiedenen Schichten sicherstellt. Diese Architektur umfasst die folgenden Schichten:

- **Präsentationsschicht:** Diese Schicht besteht aus REST-Controllern, die HTTP-Anfragen entgegennehmen und HTTP-Antworten zurückgeben. Sie interagiert mit der Service-Schicht, um Geschäftslogik zu implementieren.
- **Service-Schicht:** Diese Schicht enthält die Geschäftslogik der Anwendung. Sie validiert die Daten und ruft die entsprechenden Methoden der Repository-Schicht auf.
- **Repository-Schicht:** Diese Schicht besteht aus JPA-Repositories, die für die Datenzugriffslogik verantwortlich sind. Sie interagiert mit der MySQL-Datenbank, um Daten zu speichern und abzurufen.
- **Sicherheitsschicht:** Diese Schicht nutzt Spring Security, um Authentifizierung und Autorisierung zu implementieren. JWT (JSON Web Tokens) wird verwendet, um die Benutzersitzungen zu verwalten.
- **Datenbank-Schicht:** Diese Schicht besteht aus einer MySQL-Datenbank, in der alle Daten der Anwendung gespeichert werden.

3.2 Datenmodell und Datenbankdesign

3.2.1 Entwurf des Datenmodells

Das Datenmodell der Todo-App umfasst mehrere Entitäten, um die Beziehungen zwischen Benutzern und ihren Aufgaben zu verwalten. Die Hauptentitäten sind User und Todo.

User Entität

Die User-Entität repräsentiert einen Benutzer der Anwendung und enthält folgende Attribute:

- id: Ein eindeutiger Bezeichner für jeden Benutzer.
- username: Der Benutzername, den der Benutzer zur Anmeldung verwendet.
- password: Das verschlüsselte Passwort des Benutzers.

Die User-Entität wird als Java-Klasse implementiert und mit JPA-Anmerkungen versehen, um die Zuordnung zur Datenbank zu erleichtern.

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    // Getter und Setter
}
```

Todo Entität

3.2. DATENMODELL UND DATENBANKDESIGN

Die Todo-Entität repräsentiert eine Aufgabe und enthält folgende Attribute:

- id: Ein eindeutiger Bezeichner für jede Todo.
- title: Der Titel der Todo.
- description: Eine Beschreibung der Todo.
- dueDate: Das Fälligkeitsdatum der Todo.
- completed: Ein boolescher Wert, der angibt, ob die Todo abgeschlossen ist.
- user: Eine Beziehung zum Benutzer, der die Todo erstellt hat.

Die Todo-Entität wird als Java-Klasse implementiert und mit JPA-Anmerkungen versehen, um die Zuordnung zur Datenbank zu erleichtern.

```
@Entity
public class Todo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;

    @Column
    private String description;

    @Column
    private LocalDate dueDate;

    @Column(nullable = false)
    private boolean completed;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;
```

```
// Getter und Setter  
}
```

3.2.2 Projektanforderungen an die Datenbank

Das Projekt stellt spezifische Anforderungen an die Datenbank, die durch MySQL erfüllt werden:

- **Datenintegrität und Konsistenz:** Die Datenbank muss in der Lage sein, Daten in einer strukturierten und konsistenten Weise zu speichern. MySQL gewährleistet dies durch die Verwendung relationaler Tabellen und die Unterstützung von Transaktionen.
- **Hohe Verfügbarkeit:** Das Projekt erfordert eine Datenbank, die rund um die Uhr verfügbar ist, um eine unterbrechungsfreie Nutzung zu gewährleisten. MySQL bietet durch seine Cluster- und Replikationsfunktionen eine hohe Verfügbarkeit.
- **Skalierbarkeit:** Das Projekt muss in der Lage sein, mit wachsendem Datenvolumen und steigender Benutzeranzahl zu skalieren. MySQL kann durch seine Replikationsarchitektur und die Möglichkeit zur Anpassung an größere Datenmengen skaliert werden.
- **Sicherheit:** Der Schutz sensibler Daten ist von entscheidender Bedeutung. MySQL bietet umfassende Sicherheitsfunktionen wie SSL-Verschlüsselung, Datenmaskierung und Authentifizierungs-Plugins, um die Datensicherheit zu gewährleisten.
- **Performance:** Das Projekt erfordert schnelle Datenverarbeitungszeiten, um eine reibungslose Benutzererfahrung zu gewährleisten. MySQL bietet eine hohe Geschwindigkeit und Effizienz bei der Verarbeitung großer Datenbanken.
- **Flexibilität und Integration:** Die Datenbank muss flexibel genug sein, um mit verschiedenen Programmiersprachen und Technologien integriert zu werden. MySQL bietet eine breite Unterstützung für verschiedene Systeme und Schnittstellen.
- **Benutzerfreundlichkeit:** Eine einfache Installation und Verwaltung der Datenbank ist erforderlich, um den Entwicklungsprozess zu optimieren. MySQL ist benutzerfreundlich und bietet zahlreiche Verwaltungstools, die die Bedienung erleichtern.

3.2. DATENMODELL UND DATENBANKDESIGN

Diese Anforderungen des Projekts werden durch die funktionalen und technischen Merkmale von MySQL umfassend abgedeckt, was die Entscheidung für MySQL als Datenbanklösung rechtfertigt.

3.2.3 Datenbankdesign

Das Datenbankdesign umfasst zwei Tabellen: users und todos. Die Struktur der Tabellen ist wie folgt:

Tabelle users

- id (BIGINT, AUTO_INCREMENT, PRIMARY KEY)
- username (VARCHAR, NOT NULL, UNIQUE)
- password (VARCHAR, NOT NULL)

Tabelle todos

- id (BIGINT, AUTO_INCREMENT, PRIMARY KEY)
- title (VARCHAR, NOT NULL)
- description (TEXT)
- dueDate (DATE)
- completed (BOOLEAN, NOT NULL)
- user_id (BIGINT, FOREIGN KEY)

Durch diese Struktur wird sichergestellt, dass jede Aufgabe eindeutig einem Benutzer zugeordnet ist. Diese Beziehung ermöglicht es, dass Benutzer nur ihre eigenen Todos sehen und verwalten können.

3.3 User Interface Design

3.3.1 Gestaltung der Benutzeroberfläche

Die Benutzeroberfläche der Todo-App ist so gestaltet, dass sie intuitiv und benutzerfreundlich ist. Die Anwendung besteht aus mehreren Ansichten, die jeweils spezifische Funktionen bereitstellen:

- **/index:** Die Startseite der Anwendung siehe Abbildung 3.1. Diese Seite bietet zwei Hauptoptionen: „Log in“ und „Sign up“. Dies ermöglicht neuen Benutzern die Registrierung und bestehenden Benutzern die Anmeldung.

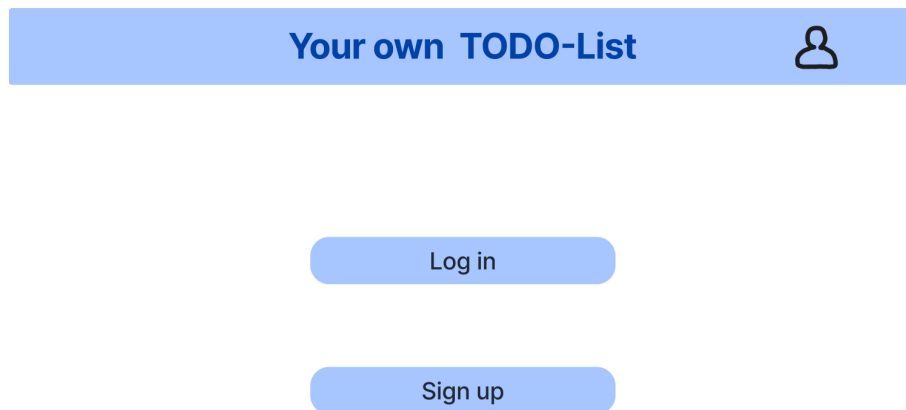


Abbildung 3.1: User Interface Design der Startseite [Eigene Darstellung]

- **/login:** Die Anmeldeseite siehe Abbildung 3.2. Hier können Benutzer ihren Benutzernamen und ihr Passwort eingeben, um auf ihre persönlichen Aufgabenlisten zuzugreifen. Ein Link zur Registrierung ist ebenfalls vorhanden.
- **/signup:** Die Registrierungsseite siehe Abbildung 3.3. Neue Benutzer können hier ein Konto erstellen, indem sie ihren Benutzernamen, ihr Passwort und die Bestätigung des Passworts eingeben. Ein Link zur Anmeldung für bestehende Benutzer ist ebenfalls verfügbar.

3.3. USER INTERFACE DESIGN

The image shows a user interface for a login page. At the top, there is a blue header bar with the text "Your own TODO-List" in white and a user icon on the right. Below the header, the text "Log in" is centered in blue. Underneath, there are two input fields: "username:" and "password:", each followed by a light gray rectangular box with a horizontal line inside. Below the password field is a blue button with the text "Log in" in white. Further down, the text "You don't have an account yet?" is centered, followed by a blue button with the text "Sign up" in white.

Abbildung 3.2: User Interface Design der Anmeldeseite [Eigene Darstellung]

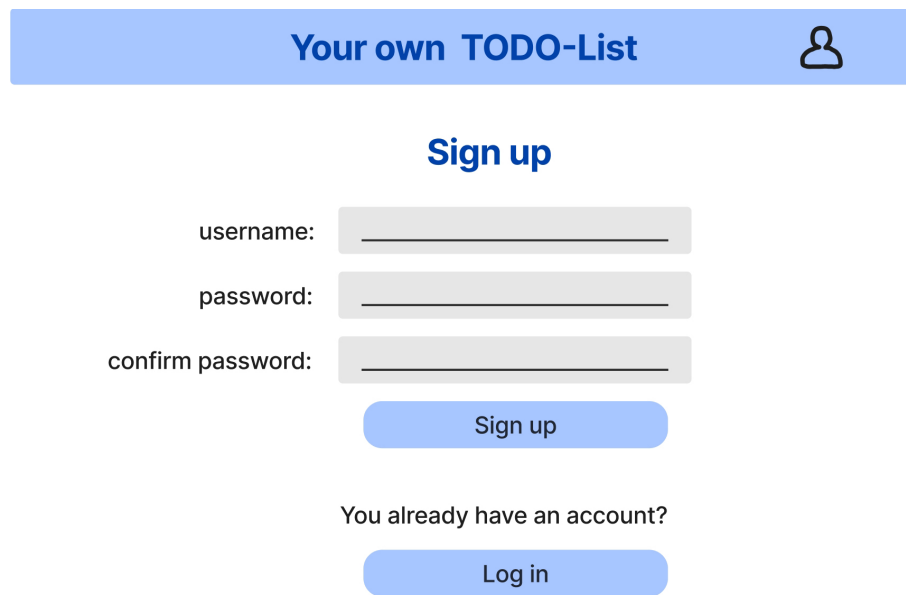
- **/home:** Die Hauptseite nach der Anmeldung siehe Abbildung 3.4. Diese Seite zeigt die Aufgabenliste des angemeldeten Benutzers. Benutzer können neue Aufgaben erstellen, vorhandene Aufgaben bearbeiten oder löschen.
- **/id_Todo:** Die Detailansicht einer spezifischen Todo siehe Abbildung 3.5. Diese Seite ermöglicht es Benutzern, die Details einer ausgewählten Todo zu bearbeiten.

Jede Seite hat ein konsistentes Layout mit einem zentralen Header, der den Titel „Your own TODO-List“ und ein Benutzer-Icon enthält. Das Benutzer-Icon signalisiert, ob ein Benutzer angemeldet ist, indem das Icon ausgefüllt ist oder, ob sich der Benutzer noch nicht angemeldet hat (unausgefülltes Icon). Bei dem drüberhoovern über das ausgefüllt Benutzer-Icon Symbol bietet sich die Möglichkeit zum abmelden („log out“) des angemeldeten Benutzers.

3.3.2 Berücksichtigung von Usability-Prinzipien

Das User Interface Design der Todo-App folgt mehreren grundlegenden Usability-Prinzipien nach Jakob Nielsen [3], um sicherzustellen, dass die Anwendung einfach zu bedienen und effizient ist:

3.3. USER INTERFACE DESIGN



The image shows a registration form for a 'TODO-List' application. At the top, there is a blue header bar with the text 'Your own TODO-List' and a user icon. Below the header, the title 'Sign up' is centered. The form consists of three input fields: 'username:', 'password:', and 'confirm password:'. Each field has a light gray border and a horizontal line for text entry. Below the input fields, there is a blue button labeled 'Sign up'. Underneath the button, the text 'You already have an account?' is displayed, followed by another blue button labeled 'Log in'.

Abbildung 3.3: User Interface Design der Registrierungsseite [Eigene Darstellung]

1. **Sichtbarkeit des Systemstatus:** Die App hält den Benutzer stets über den aktuellen Status und die Ergebnisse ihrer Aktionen informiert. Beispielsweise werden Änderungen an Aufgaben sofort angezeigt und erfolgreiche Anmeldungen führen direkt zur Hauptseite mit der Aufgabenliste.
2. **Übereinstimmung zwischen System und realer Welt:** Die Anwendung verwendet Begriffe und Konzepte, die den Benutzern vertraut sind. Schaltflächen und Symbole sind intuitiv und leicht verständlich, was die Bedienung erleichtert.
3. **Benutzerkontrolle und Freiheit:** Benutzer können Aktionen rückgängig machen. Es gibt deutlich sichtbare Optionen zum Abbrechen von Aktionen, um Fehlaktionen leicht korrigieren zu können.
4. **Konsistenz und Standards:** Das Layout und das Design der Benutzeroberfläche sind auf allen Seiten der Anwendung konsistent. Dies erleichtert den Benutzern das Verständnis und die Navigation durch die App.
5. **Wiedererkennung statt Erinnerung:** Die Benutzeroberfläche macht alle wichtigen Optionen und Funktionen sichtbar, damit Benutzer sie leicht wiedererkennen können, anstatt sich an sie erinnern zu müssen. Wichtige Elemente wie Schaltflächen

3.3. USER INTERFACE DESIGN

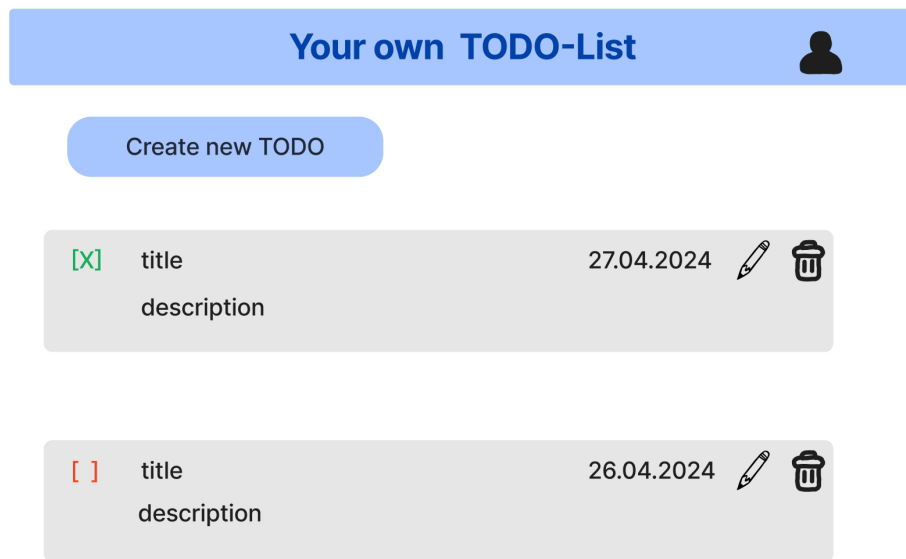
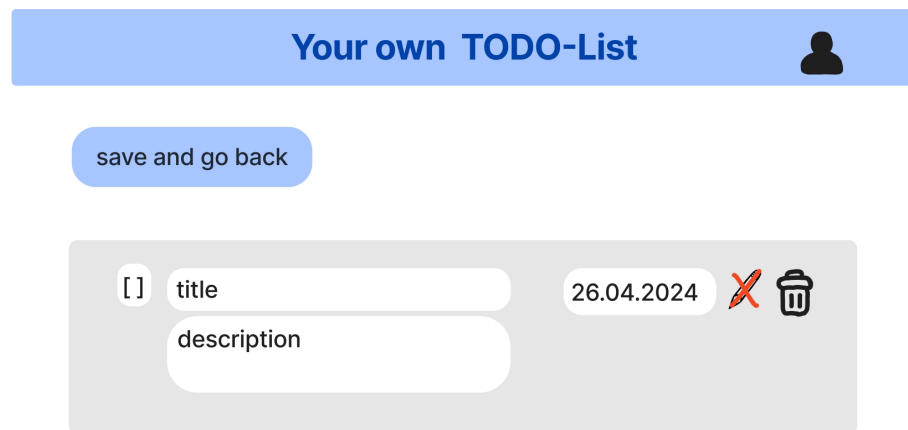


Abbildung 3.4: User Interface Design der Hauptseite [Eigene Darstellung]


chen und Eingabefelder sind prominent platziert und leicht zu finden. Der Einsatz von Farben und Schriftgrößen hilft, die Aufmerksamkeit der Benutzer auf wichtige Aktionen und Informationen zu lenken.

6. **Ästhetik und minimalistisches Design:** Das Design ist einfach und übersichtlich gehalten, um die Benutzerfreundlichkeit zu erhöhen. Unnötige Elemente wurden vermieden, um Ablenkungen zu minimieren und den Fokus auf die Hauptfunktionen der Anwendung zu legen.



3.3. USER INTERFACE DESIGN



The image shows a user interface design for a 'Your own TODO-List' application. At the top is a blue header bar with the text 'Your own TODO-List' and a user icon. Below the header is a blue button labeled 'save and go back'. The main content area is a light gray box containing a form for a todo item. The form has a checkbox labeled '[]', a 'title' input field, a 'description' input field, a date field showing '26.04.2024', a red 'X' icon, and a trash can icon.

Your own TODO-List 

save and go back

☐ title 26.04.2024  

description

Abbildung 3.5: User Interface Design der Detailansicht [Eigene Darstellung]

4 Implementierung unter Verwendung von TDD

4.1 Entwicklungsumgebung und Werkzeuge

Die Entwicklung der Todo-App erfolgte in einer modernen Java-Entwicklungsumgebung, die auf dem Spring Boot Framework basiert. Die wichtigsten verwendeten Tools und Technologien sind:

- **IDE:** IntelliJ IDEA wurde als Integrated Development Environment (IDE) verwendet, da es umfangreiche Unterstützung für Java und das Spring Framework bietet, einschließlich leistungstarker Debugging- und Refactoring-Tools.
- **Build-Tool:** Maven wurde für das Build-Management und die Abhängigkeitsverwaltung eingesetzt. Es ermöglicht die einfache Verwaltung von Bibliotheken und Plugins sowie die Konfiguration von Build-Prozessen.
- **Versionierung:** Git und GitHub wurden für die Quellcodeverwaltung und Versionskontrolle verwendet. GitHub ermöglichte zudem die Zusammenarbeit und den Austausch von Code.
- **Test-Frameworks:** JUnit 5 und Spring Boot Test wurden für die Implementierung und Ausführung von Unit- und Integrationstests verwendet. Mockito diente zur Erstellung von Mock-Objekten für die Isolierung von Testfällen.
- **Datenbank:** MySQL wurde als relationale Datenbank verwendet. Für die Tests wurde eine MySQL-Testdatenbank konfiguriert, um schnelle und isolierte Testausführungen zu ermöglichen.

4.2 Implementierungsschritte

Die Implementierung der Benutzerregistrierung in der Todo-App folgte dem klassischen TDD-Zyklus: **Red-Green-Refactor**.

1. **Red Phase:** Zunächst wurde ein fehlgeschlagener Test (Red) geschrieben, der die Registrierung eines neuen Benutzers beschreibt, ohne dass die Implementierung vorhanden war.

Beispiel: Ein Test für die Registrierung eines neuen Benutzers über den UserController.

```
@Test
public void testRegisterUser_success() throws Exception {
    UserRegistrationRequest request = new UserRegistrationRequest();
    request.setUsername("integrationtestuser");
    request.setPassword("password");
    request.setConfirmPassword("password");

    ResultActions result = mockMvc.perform(post("/register")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(request)));

    result.andExpect(status().isOk());
}
```

2. **Green Phase:** Anschließend wurde der minimal notwendige Code geschrieben, um den Test erfolgreich zu bestehen (Green). In diesem Schritt wird nur so viel implementiert, dass der Testfall erfüllt wird.

Beispiel: Die grundlegende Implementierung des registerUser-Endpunkts im UserController.

```
@PostMapping("/register")
public ResponseEntity<?> registerUser(@RequestBody
    UserRegistrationRequest request) {
    User user = new User();
    user.setUsername(request.getUsername());
}
```

4.2. IMPLEMENTIERUNGSSCHRITTE

```
user.setPassword(passwordEncoder.encode(request.getPassword()));

return ResponseEntity.ok(user);
}
```

3. **Refactor Phase:** Nach dem erfolgreichen Bestehen des Tests wurde der Code optimiert und verbessert, ohne die Funktionalität zu ändern. Dabei wurde auf Sauberkeit, Lesbarkeit und Wartbarkeit des Codes geachtet.

Beispiel: Die vollständige Implementierung des registerUser-Endpunktes im UserController zeigt, dass ein Refactoring erfolgte, nachdem die Implementierung zum erfolgreichen Ausführen von Test, wie zum Beispiel des Testens der Fehlerbehandlung von bereits existierenden Benutzernamen, hinzugefügt wurde.

```
@PostMapping("/register")
public ResponseEntity<?> registerUser(@Valid @RequestBody
    UserRegistrationRequest request, BindingResult result) {
    // Check if username already exists
    if (userService.existsByUsername(request.getUsername())) {
        return ResponseEntity.status(HttpStatus.CONFLICT).body("Username
            already exists");
    }

    // Check if password and confirmation match
    if (!request.getPassword().equals(request.getConfirmPassword())) {

        return ResponseEntity.status(HttpStatus.CONFLICT).body("Password
            and confirm password do not match");
    }

    // Check for validation results
    if (result.hasErrors()) {

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("
            Validation error: " + result.getAllErrors());
    }
}
```

4.3. BEISPIELTESTS UND TESTFÄLLE

```
// Register user
try {
    User user = authenticationService.register(request);
    return ResponseEntity.ok(user);
} catch (Exception e) {
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).
        body("Error registering user");
}
}
```

Die konsequente Anwendung des TDD-Zyklus gewährleistet, dass die Implementierung kontinuierlich durch Tests begleitet und abgesichert wird und somit zu einem stabilen und fehlerarmen Code führen kann.

4.3 Beispieltests und Testfälle

4.4 Unit-Tests

Unit-Tests wurden verwendet, um einzelne Komponenten isoliert zu testen. Ein Beispiel ist der Test der `UserController`-Klasse, um sicherzustellen, dass eine Registrierung eines Benutzers erfolgreich verläuft.

```
@Test
public void testRegisterUserPasswordMismatch() {
    UserRegistrationRequest request = new UserRegistrationRequest("testuser", "
        password1", "password2");

    BindingResult result = mock(BindingResult.class);
    ResponseEntity<?> response = userController.registerUser(request, result);

    assertEquals(HttpStatus.CONFLICT, response.getStatusCode());
    assertEquals("Password and confirm password do not match", response.getBody
        ());
}
```

4.5 Integrationstests

Integrationstests wurden durchgeführt, um das Zusammenspiel verschiedener Komponenten zu überprüfen. Ein Beispiel ist der Integrationstest für die Benutzerregistrierung über den UserController.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class UserRegistrationAcceptanceTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testRegisterUser() {
        String url = "http://localhost:" + port + "/register";
        UserRegistrationRequest request = new UserRegistrationRequest("testuser"
            , "password", "password");

        HttpHeaders headers = new HttpHeaders();
        HttpEntity<UserRegistrationRequest> entity = new HttpEntity<>(request,
            headers);

        // Send HTTP POST request
        ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.
            POST, entity, String.class);

        // Check if the response status code is 409 (CONFLICT)
        assertEquals(HttpStatus.CONFLICT, response.getStatusCode());
    }
}
```

4.6 Akzeptanztests

Akzeptanztests wurden verwendet, um sicherzustellen, dass die Anwendung den Anforderungen der Benutzer entspricht. Diese Tests wurden aus der Sicht des Endbenutzers geschrieben und überprüfen die Funktionalität der gesamten Anwendung.

Ein Beispiel ist der Akzeptanztest für die Benutzerregistrierung über die REST-Schnittstelle.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class UserRegistrationAcceptanceTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testRegisterUser() {
        String url = "http://localhost:" + port + "/register";
        UserRegistrationRequest request = new UserRegistrationRequest("testuser"
            , "password", "password");

        HttpHeaders headers = new HttpHeaders();
        HttpEntity<UserRegistrationRequest> entity = new HttpEntity<>(request,
            headers);

        // Send HTTP POST request
        ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.
            POST, entity, String.class);

        // Check if the response status code is 409 (CONFLICT)
        assertEquals(HttpStatus.CONFLICT, response.getStatusCode());
    }
}
```

Diese strukturierte Vorgehensweise durch TDD führte zu einer robusten und fehlerarmen

4.6. AKZEPTANZTESTS

Implementierung der Todo-App, da jedes Feature durch entsprechende Tests abgesichert und validiert wurde.

5 Frontend Entwicklung

5.1 Projektanforderung an Frontend-Entwicklung

Das Projekt stellt vielfältige Anforderungen an die Entwicklung des Frontends:

- **Einfachheit und Flexibilität:** Vue.js bietet eine sanfte Lernkurve und ist flexibel einsetzbar, was es ideal für verschiedene Projektanforderungen macht.
- **einfache Integrität:** Vue.js ermöglicht eine einfache Integration in bestehende Projekte und Technologiestacks.
- **reaktive Datenbindung:** Durch die reaktive Datenbindung können Änderungen automatisch auf der Benutzeroberfläche reflektiert werden, was die Entwicklung dynamischer Anwendungen erleichtert.
- **Komponentenbasierte Architektur:** Die komponentenbasierte Architektur fördert die Wiederverwendbarkeit und Wartbarkeit des Codes, indem sie die Trennung von Logik und Darstellung erleichtert.
- **Leistungsfähigkeit:** Vue.js bietet eine hohe Leistung und Effizienz bei der Erstellung von Benutzeroberflächen.
- **Große Community und gute Dokumentation:** Eine aktive Community und umfassende Dokumentation erleichtern die Lösung von Problemen und den Zugriff auf Ressourcen.

Die Wahl von Vue.js als Framework erfüllt diese Anforderungen und rechtfertigt daher die Entscheidung für seine Verwendung im Projekt.

5.2 Technologiestack

Das Frontend verwendet folgende Technologien und Frameworks:

- **Vue.js (Version 3.2.13)**: Haupt-Framework für die Erstellung der Benutzeroberfläche.
- **Vue Router (Version 4.4.0)**: Routing-Library für die Navigation innerhalb der Anwendung.
- **Vuex (Version 4.1.0)**: State-Management-Bibliothek für die zentralisierte Speicherung von Daten.

Das Projekt wurde mit Hilfe des Vue CLI (Command Line Interface) initialisiert, um eine standardisierte Projektstruktur und die notwendigen Abhängigkeiten bereitzustellen. Nach der Erstellung des Projekts wurden die zusätzliche Bibliotheken **axios** für HTTP-Anfragen und **vue-router** für die Navigation installiert.

5.3 Projektstruktur

Die Projektstruktur wurde so gestaltet, dass sie eine klare Trennung der einzelnen Komponenten und Funktionalitäten ermöglicht. Die Verzeichnisstruktur ist wie folgt:

- **/src**: Hauptverzeichnis für den Quellcode.
- **/src/components**: Enthält wiederverwendbare Vue-Komponenten.
- **/src/components/HeaderBar.vue**: Die Kopfzeile der Anwendung, die das Logo und die Logout-Option enthält, abhängig davon, ob der Benutzer angemeldet ist.
- **/src/views**: Enthält die Hauptansichten der Anwendung.
- **/src/views/IndexView.vue**: Die Startseite der Anwendung, die Optionen zum Anmelden und Registrieren bereitstellt.
- **/src/views/LoginView.vue**: Das Anmeldeformular, das Benutzernamen und Passwort für die Authentifizierung erfordert.
- **/src/views/RegisterView.vue**: Das Registrierungsformular, das Benutzer zur Erstellung eines Kontos ermöglicht.

5.4. ROUTING

- `/src/views/HomeView.vue`: Die Hauptansicht nach der Anmeldung, die eine Liste von Todos anzeigt.
- `/src/views/NewTodoForm.vue`: Ein Formular zur Erstellung neuer Todos.
- `/src/views/ToDoDetailView.vue`: Ansicht zum Bearbeiten eines bestimmten Todos.
- `/src/router.js`: Konfiguration der Vue Router-Instanz.
- `/src/store.js`: Vuex Store-Konfiguration für das State-Management.
- `/src/App.vue`: Wurzelkomponente der Anwendung.
- `/src/main.js`: Einstiegspunkt, wo die Vue-Instanz erstellt und konfiguriert wird.

5.4 Routing

Der Router wurde in der Datei `src/router.js` konfiguriert. Die Konfiguration umfasst die Definition der verschiedenen Routen, die jeweils einer spezifischen Komponente zugeordnet sind. Dies ermöglicht eine einfache Navigation zwischen den verschiedenen Ansichten der Anwendung.

5.5 Authentifizierung und Autorisierung

Die Webanwendung verwendet JSON Web Tokens (JWT) zur Authentifizierung von Benutzern. Nach erfolgreicher Anmeldung wird ein Token im Local Storage gespeichert und für alle folgenden Anfragen an den Server verwendet. Die Authentifizierung wird durch Vue Router Navigation Guards implementiert, um sicherzustellen, dass nur authentifizierte Benutzer auf geschützte Routen zugreifen können.

5.6 HTTP-Anfragen

HTTP-Anfragen werden mit Axios durchgeführt, um mit dem Backend-Server zu kommunizieren. Diese Anfragen werden in den Vue-Komponenten ausgeführt, um Daten zu

5.7. STATE MANAGEMENT

laden, zu speichern oder zu aktualisieren. Für jede wird der JWT-Token automatisch zum Header hinzugefügt, um sicherzustellen, dass der Server die Anfrage autorisiert.

5.7 State Management

Vuex wird verwendet, um den globalen Zustand der Anwendung zu verwalten. Es speichert den Anmeldestatus und den JWT-Token des Benutzers und bietet zentralisierten Zugriff auf diese Daten. Die State-Management-Logik wurde in der Datei `src/store.js` definiert.

5.8 Styling

Das Styling erfolgt hauptsächlich mit CSS innerhalb der einzelnen Vue-Komponenten. Das Scoped Styling von Vue.js sorgt dafür, dass die Styles nur auf die jeweilige Komponente angewendet werden.

Literatur

- [1] Alexander Schatten u.a. *Best Practice Software-Engineering*. de. Heidelberg: Spektrum Akademischer Verlag, 2010. ISBN: 978-3-8274-2486-0 978-3-8274-2487-7. DOI: 10.1007/978-3-8274-2487-7. URL: <http://link.springer.com/10.1007/978-3-8274-2487-7> (besucht am 18.03.2024).
- [2] Stephan Kleuker. *Qualitätssicherung durch Softwaretests: Vorgehensweisen und Werkzeuge zum Testen von Java-Programmen*. de. Wiesbaden: Springer Fachmedien, 2019. ISBN: 978-3-658-24885-7 978-3-658-24886-4. DOI: 10.1007/978-3-658-24886-4. URL: <http://link.springer.com/10.1007/978-3-658-24886-4> (besucht am 18.03.2024).
- [3] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. en. URL: <https://www.nngroup.com/articles/ten-usability-heuristics/> (besucht am 01.06.2024).