324-07A API Versionierung

Inhalt

Einführung in das Thema Versionierung einer Java / Spring-Boot REST-Schnittstelle

Was ist API-Versionierung?

API-Versionierung ist ein entscheidender Aspekt der Softwareentwicklung, der es ermöglicht, Änderungen an einer REST-Schnittstelle vorzunehmen, ohne bestehende Clients zu beeinträchtigen.

Warum ist Versionierung wichtig?

Backward Compatibility: Bestehende Clients können weiterhin funktionieren, auch wenn neue Features hinzugefügt werden.

Schrittweise Migration: Entwickler können in ihrem eigenen Tempo auf neue API-Versionen migrieren.

Parallele Entwicklung: Verschiedene Teams können gleichzeitig an unterschiedlichen API-Versionen arbeiten.

Risikoreduzierung: Breaking Changes werden kontrolliert eingeführt, ohne die Produktionsumgebung zu gefährden.

Übersicht über die verschiedenen Methoden zur Versionierung

URI-basierte Versionierung (URL Path Versioning)

Die Versionsnummer wird direkt in den URL-Pfad eingebettet, sodass der Server anhand des Pfadmusters die gewünschte API-Version auswählt.

Beispiele:

- GET /api/v1/tasks
- GET /api/v2/tasks
- POST /api/v1/tasks

Query Parameter Versionierung

Die Version wird als Query-Parameter übergeben, wodurch sich Clients flexibel für unterschiedliche Versionen entscheiden können, ohne die Core-URL zu ändern.

Beispiele:

- GET /api/tasks?version=1
- GET /api/tasks?version=2
- POST /api/tasks?v=1.1

Header-basierte Versionierung (Custom Header)

Die Version wird über einen benutzerdefinierten HTTP-Header übertragen, sodass die URL unverändert bleibt und die Versionierung vollständig im Header erfolgt.

Beispiele:

- X-API-VERSION: 1
- API-Version: 2.0
- Custom-Version: v1.5

Content Negotiation (Accept Header)

Die Version wird über den Accept-Header mit speziellen Media-Types kommuniziert, wodurch Clients durch Setzen des Headers steuern, welche Version sie erhalten.

Beispiele:

- Accept: application/vnd.todoapp-v1+json
- Accept: application/vnd.todoapp-v2+json
- Accept: application/vnd.company.app+json;version=1.0

Bewertung der verschiedenen Methoden hinsichtlich ihrer Vor- und Nachteile

Bewertungsmatrix

Kriterium	URI-Versioning	Query Parameter	Custom Header	Content Negotiation
Einfachheit	5/5 Sterne	4/5 Sterne	3/5 Sterne	2/5 Sterne
Zuverlässigkeit	4/5 Sterne	3/5 Sterne	4/5 Sterne	5/5 Sterne
Flexibilität	3/5 Sterne	3/5 Sterne	4/5 Sterne	5/5 Sterne
Browser-Testbarkeit	5/5 Sterne	5/5 Sterne	2/5 Sterne	1/5 Sterne
Caching-Kompatibilität	5/5 Sterne	4/5 Sterne	2/5 Sterne	3/5 Sterne
REST-Konformität	3/5 Sterne	2/5 Sterne	3/5 Sterne	5/5 Sterne
Dokumentations-Freundlichkeit	5/5 Sterne	3/5 Sterne	3/5 Sterne	2/5 Sterne
Skalierbarkeit	3/5 Sterne	2/5 Sterne	4/5 Sterne	5/5 Sterne

Gesamt-Score

1. URI-Versioning 30/40 (SIEGER)
2. Content Negotiation 29/40
3. Custom Header 27/40
4. Query Parameter 26/40

Detaillierte Analyse

URI-basierte Versionierung

Vorteile:

- Sehr einfach zu implementieren und zu verstehen
- Sofort im Browser testbar ohne zusätzliche Tools
- Klare Trennung verschiedener API-Versionen
- Hervorragende Caching-Eigenschaften
- Einfache Dokumentation und API-Discovery

Nachteile:

- "URI Pollution" - URLs werden durch Versionsnummern verschmutzt
- Bei vielen Versionen wird die URL-Struktur unübersichtlich
- Nicht vollständig REST-konform (Resourcen sollten versionsneutral sein)

Query Parameter Versionierung

Vorteile:

- Einfach zu implementieren
- Browser-kompatibel
- Flexible Kombinationen mit anderen Parametern

Nachteile:

- URI Pollution
- Verwechslungsgefahr mit funktionalen Parametern
- Weniger professionell wirkend

Header-basierte Versionierung

Vorteile:

- Saubere URLs ohne Versionsinformationen
- Flexible Implementierung
- Granulare Kontrolle über Versionierung

Nachteile:

- Schwieriger im Browser zu testen
- Höhere Komplexität für Client-Entwickler

Content Negotiation (Accept Header)

Vorteile:

- HTTP-Standard-konform
- Sehr flexibel und erweiterbar
- Professioneller Ansatz

Nachteile:

- Höchste Implementierungskomplexität
- Schwer im Browser testbar

Begründung für die Wahl der gewählten Methode

Entscheidung: URI-basierte Versionierung

Nach sorgfältiger Abwägung aller Faktoren haben wir uns für die URI-basierte Versionierung entschieden.

Ausschlaggebende Faktoren

1. Einfachheit und Verständlichkeit
    - Neue Teammitglieder verstehen das Konzept sofort
    - Minimaler Lernaufwand für Frontend-Entwickler
    - Selbsterklärende URLs: /api/v1/tasks vs /api/v2/tasks
2. Entwickler-Experience
    - Direktes Testen im Browser möglich
    - Einfache Debugging-Möglichkeiten
    - Klare Fehlermeldungen bei falschen URLs
3. Maintenance und Support
    - Einfache Code-Organisation durch separate Controller
    - Klare Trennung der Verantwortlichkeiten
    - Straightforward Dokumentation

Abgewogene Nachteile

- URI Pollution: Akzeptabel für unser Projekt-Scope
- REST-Konformität: Pragmatismus über Purismus
- Skalierbarkeit: Ausreichend für absehbare Versionsanzahl

Schritt-für-Schritt-Anleitung zur Implementierung der gewählten Methode

Projektvorbereitung

Zunächst wird die bestehende Codebase analysiert. Die ursprüngliche Funktionalität wird unverändert beibehaltet.

```
@RestController
@SpringBootApplication
public class DemoApplication {
   @GetMapping("/")
   public List<Task> getTasks() { ... }

   @PostMapping("/tasks")
   public String addTask(@RequestBody String taskdescription) { ... }

   @PostMapping("/delete")
   public String delTask(@RequestBody String taskdescription) { ... }

   @PostMapping("/toggle-status")
   public String toggleTaskStatus(@RequestBody String taskdescription) { ... }
}
```

Backend-Implementierung

Schritt 1: Versionierte Endpunkte hinzufügen

In DemoApplication.java ergänzen:

```
// ========== ZUSÄTZLICHE API-VERSIONIERUNG ==========
// API Version 1.0 - Strukturierte Endpunkte
@CrossOrigin
@GetMapping("/api/v1/tasks")
public List<Task> getTasksV1() {
   System.out.println("API v1.0 - GET '/api/v1/tasks'");
   return getTasks(); // Verwendet die originale Funktion
}

@CrossOrigin
@PostMapping("/api/v1/tasks")
public String addTaskV1(@RequestBody String taskdescription) {
   System.out.println("API v1.0 - POST '/api/v1/tasks'");
   return addTask(taskdescription); // Verwendet die originale Funktion
}

@CrossOrigin
@PostMapping("/api/v1/tasks/delete")
```

```
public String deleteTaskV1(@RequestBody String taskdescription) {
  System.out.println("API v1.0 - POST '/api/v1/tasks/delete'");
  return delTask(taskdescription); // Verwendet die originale Funktion
}

@CrossOrigin
@PostMapping("/api/v1/tasks/toggle")
public String toggleTaskV1(@RequestBody String taskdescription) {
  System.out.println("API v1.0 - POST '/api/v1/tasks/toggle'");
  return toggleTaskStatus(taskdescription); // Verwendet die originale Funktion
}
```

Schritt 2: Erweiterte Version 2.0 implementieren

In DemoApplication.java ergänzen:

```
// API Version 2.0 - Mit Verbesserungen
@CrossOrigin
@GetMapping("/api/v2/tasks")
public List<Task> getTasksV2() {
  System.out.println("API v2.0 - GET '/api/v2/tasks' (mit Sortierung)");
  List<Task> allTasks = taskRepository.findAll();

  // V2 Feature: Nach Priorität sortieren
  allTasks.sort((t1, t2) -> {
    if ("hoch".equals(t1.getPriority())) return -1;
    if ("hoch".equals(t2.getPriority())) return 1;
    if ("mittel".equals(t1.getPriority())) return -1;
    if ("mittel".equals(t2.getPriority())) return 1;
    return 0;
  });
  return allTasks;
}

@CrossOrigin
@PostMapping("/api/v2/tasks")
public String addTaskV2(@RequestBody String taskdescription) {
  System.out.println("API v2.0 - POST '/api/v2/tasks' (mit besserer Validierung)");
  ObjectMapper mapper = new ObjectMapper();
  try {
    Task task = mapper.readValue(taskdescription, Task.class);

    // V2 Feature: Bessere Validierung
    if (task.getTaskdescription() == null || task.getTaskdescription().trim().isEmpty()) {
      System.out.println(">>>Error: Leere Task-Beschreibung nicht erlaubt!");
      return "Error: Task description required";
    }

    // Rest wie original
```

```
    return addTask(taskdescription);

  } catch (JsonProcessingException e) {
    e.printStackTrace();
    return "Error: Invalid JSON";
  }
}
```

Frontend-Implementierung

Schritt 1: State für API-Versionierung hinzufügen

In App.js Constructor ergänzen:

```
constructor(props) {
  super(props);
  this.state = {
    // ... bestehende States beibehalten ...

    // API-VERSIONIERUNG: Neue State-Variable
    apiVersion: "original" // "original", "v1", "v2"
  };
}
```

Schritt 2: URL-Hilfsfunktion implementieren

In App.js ergänzen:

```
// API-VERSIONIERUNG: Hilfsfunktion für URLs
getApiUrl = () => {
  switch(this.state.apiVersion) {
    case "v1": return "http://localhost:8080/api/v1";
    case "v2": return "http://localhost:8080/api/v2";
    default: return "http://localhost:8080"; // original
  }
}
```

Schritt 3: Version-Wechsel-Handler

In App.js ergänzen:

```
// API-VERSIONIERUNG: Version wechseln
handleVersionChange = version => {
  this.setState({ apiVersion: version }, () => {
    // Tasks neu laden wenn Version gewechselt wird
    this.componentDidMount();
  });
}
```

Schritt 4: Bestehende API-Calls anpassen

In App.js handleSubmit anpassen

```
handleSubmit = event => {
  event.preventDefault();
  console.log("Sending task description to Spring-Server: " + this.state.taskdescription);

  // API-VERSIONIERUNG: URL anpassen
  const baseUrl = this.getApiUrl();
  const endpoint = this.state.apiVersion === "original" ? "/tasks" : "/tasks";

  fetch(baseUrl + endpoint, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      taskdescription: this.state.taskdescription,
      priority: this.state.priority,
      dueDate: this.state.dueDate,
      completed: this.state.completed
    })
  })
  // ... rest der Implementierung bleibt gleich
}
```

Schritt 5: UI für Versionsauswahl

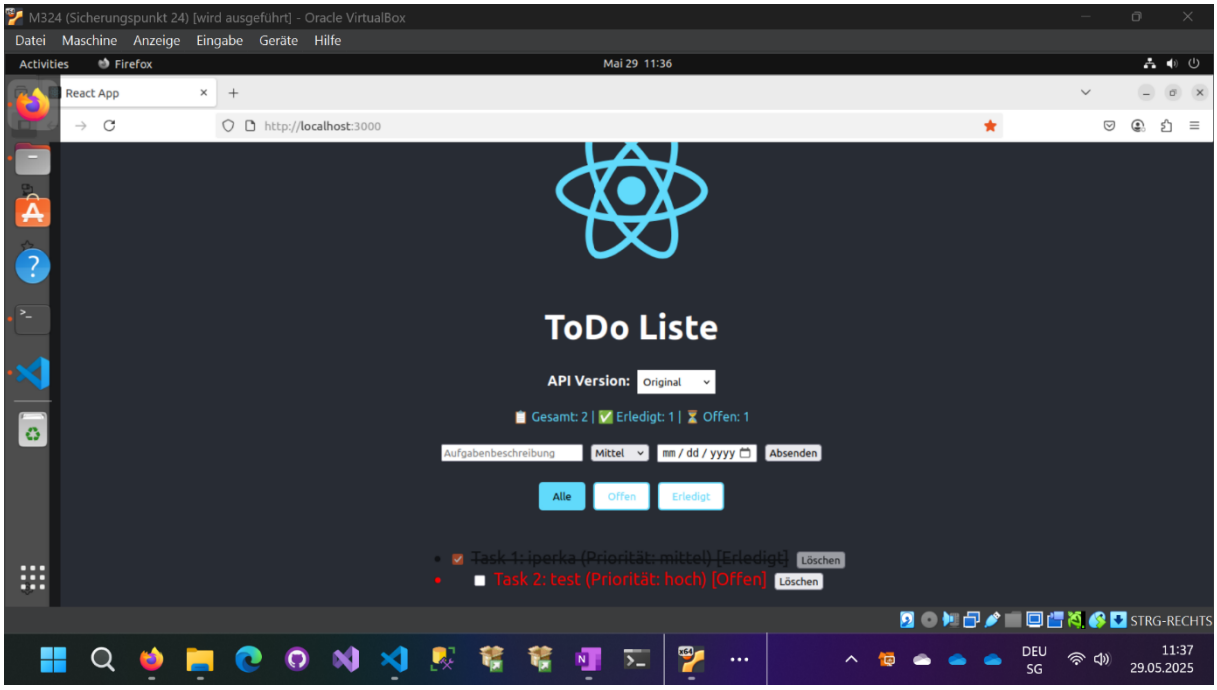
In render() method in App.js ergänzen:

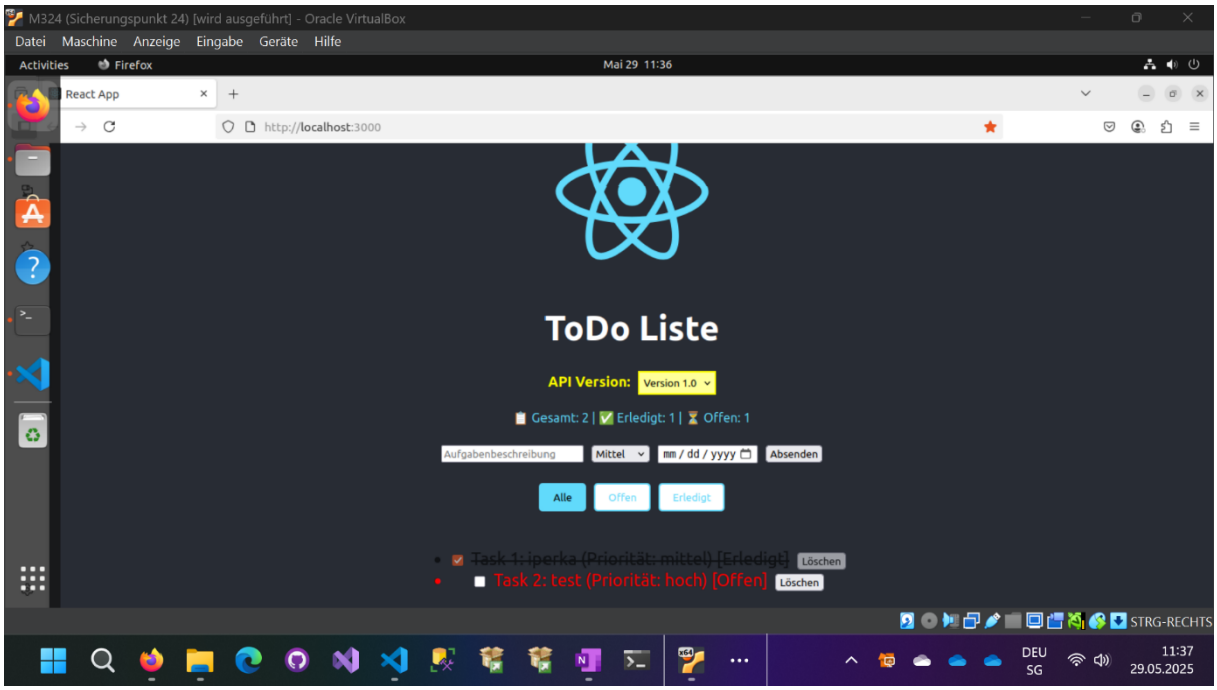
```
{/* API-VERSIONIERUNG: Neue Auswahl */}
<div style={{marginBottom: '20px'}}>
  <label style={{
    color: this.state.apiVersion === 'original' ? 'white' :
        this.state.apiVersion === 'v1' ? 'yellow' : 'lightblue',
    fontWeight: 'bold',
    fontSize: '18px'
  }}>
    API Version:
  </label>
  <select
    value={this.state.apiVersion}
    onChange={(e) => this.handleVersionChange(e.target.value)}
    style={{
      marginLeft: '10px',
      padding: '5px',
      backgroundColor: this.state.apiVersion === 'original' ? '#ffffff' :
              this.state.apiVersion === 'v1' ? '#ffff99' : '#99ccff',
      border: '2px solid ' + (this.state.apiVersion === 'original' ? '#ffffff' :
                  this.state.apiVersion === 'v1' ? '#ffff00' : '#0099ff')
    }}
```

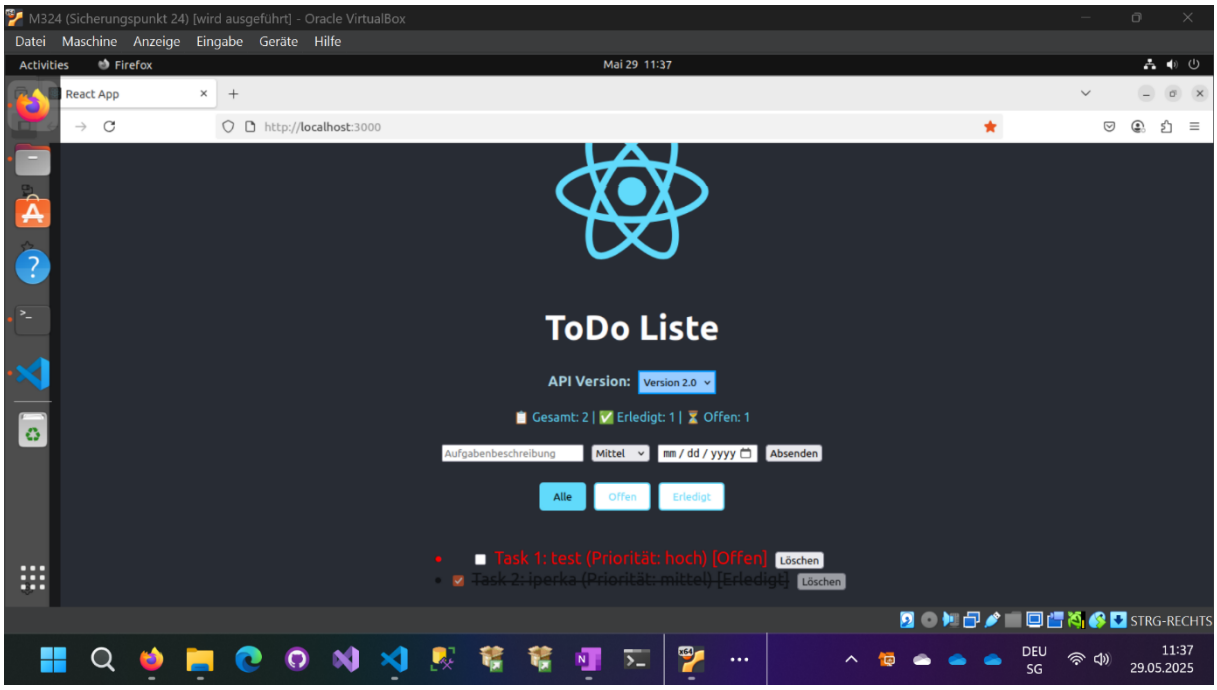
```
  >
    <option value="original">Original</option>
    <option value="v1">Version 1.0</option>
    <option value="v2">Version 2.0</option>
  </select>
</div>
```

Endresultat

Das Endresultat sieht wie folgt aus.







Zusammenfassung und Schlussfolgerungen

Lessons Learned

Was gut funktioniert hat:

1. URI-basierte Versionierung erwies sich als optimal für unser Projekt
2. Code-Wiederverwendung reduzierte Entwicklungszeit erheblich
3. Farbkodierung im Frontend bietet excellente User Experience
4. Schrittweise Implementierung ermöglichte risikoarme Entwicklung

Verbesserungspotential:

1. Automatisierte Tests für alle API-Versionen implementieren
2. API-Dokumentation mit Swagger/OpenAPI ergänzen
3. Deprecation-Strategie für Legacy-Versionen definieren
4. Monitoring der API-Nutzung pro Version einführen

Fazit

Die Implementierung der URI-basierten API-Versionierung in unserem ToDo-Projekt war ein voller Erfolg. Wir haben eine robuste, wartbare und benutzerfreundliche Lösung geschaffen, die:

- Alle originalen Funktionen erhält
- Schrittweise Modernisierung ermöglicht
- Excellente Developer Experience bietet
- Zukunftssichere Architektur etabliert

Das gewählte Vorgehen beweist, dass pragmatische Entscheidungen oft zu den besten Ergebnissen führen. Die URI-basierte Versionierung mag nicht die "purste" REST-Lösung sein, aber sie ist definitiv die praktischste für unseren Anwendungsfall.

Die erfolgreiche Implementation zeigt, dass API-Versionierung nicht komplex sein muss, um effektiv zu sein. Durch die bewusste Entscheidung für Einfachheit über Perfektion haben wir eine Lösung geschaffen, die sowohl für Entwickler als auch für Endnutzer optimal funktioniert.