Hochschule RheinMain
H. Arega, W. Weitz

Studiengang Medieninformatik
Sommersemester 2024

Webbasierte Anwendungen

Web-Ueb-06 (29.05.2024)

Hinweis: Mit diesem Blatt schließen wir die Basisfunktionalität des "Spring WebMVC"-Teils unseres Projekts ab. Nach Ablauf der Bearbeitungsfrist (wie immer 12 Tage nach Erscheinen dieses Blatts) erfolgt die erste Zwischenbesprechung mit Vorführung des Systems unter Teilnahme aller Teammitglieder und ggf. Live-Coding/Korrektur-Einlagen. Thema ist Umsetzung aller Aufgaben des Spring-Projekts bis einschließlich dieser, bitte beachten Sie die auf den Übungsblätter angegebenen Akzeptanzkritieren (gültig jeweils für die zugehörigen Abgabezeitpunkte).

Bei diesem Übungsblatt geht es zunächst darum, wie man externe REST-Dienste in die eigene Anwendung integrieren und umgekehrt auch eine REST-artige API für eigene Funktionalität nach außen anbieten kann.

Sie können z.B. die VS-Code-Extension *Thunder Client* hier gut zum REST-Testen verwenden (Extension-ID `rangav.vscode-thunder-client`).

Aufgabe 1 (Geolokalisierungs-API von *Nominatim* integrieren)

Wie ein Blick auf Verzeichnisse wie z.B. `https://rapidapi.com/hub` zeigt, gibt es im Internet unglaublich viele Web-Dienste, deren APIs man in eigenen Anwendungen nutzen kann (kostenlos oder bezahlt, mit oder ohne Registrierung usw.).

Ein solcher Dienst ist `https://nominatim.org/` aus dem Umfeld des OpenStreetMap-Projekts, der (in vernünftigem Rahmen) ohne Registrierung nutzbar ist und Geocodierungen in beiden Richtungen zur Verfügung stellt. Er findet also zu einem eingegebenen Ortsnamen plausible Kandidaten (mit Geoinformationen) und kann umgekehrt zu gegebenen Koordinaten ermitteln, wo das ist (Land / Ort etc).

Die genaue API ist auf der genannten Webseite dokumentiert. Für unsere Zwecke sind wir daran interessiert, zu einem Ortsnamen dessen Koordinaten zu ermitteln. *Nominatim* kann in verschiedenen Datenformaten antworten, wir hätten gerne JSON. Außerdem beschränken wir unsere Suche auf Orte in Deutschland.

Beispiel: Eine HTTP-GET-Abfrage auf den `/search`-Endpunkt führt eine Suche nach "Karlsruhe" (Parameter `q` für *Query*) in Deutschland (`countrycodes=de`) mit Antwort im JSON-Format (`format=json`):

```
https://nominatim.openstreetmap.org/search?q=karlsruhe&format=json&
countrycodes=de
```

Testabruf und Ergebnisstruktur

Bitte nutzen Sie den Thunderclient in VSCode, um die angegebene Beispiel-URL abzurufen und das gelieferte Datenformat zu verstehen. Sie sehen, warum die API immer eine *Liste* zurückliefert – es kann sein, dass nur eine (oder keine) Antwortmöglichkeit ermittelt wird, oft sind es aber mehrere (hier Karlsruhe Stadt und Karlsruhe Landkreis – ohne Einschränkung auf `countrycodes=de` wäre auch noch ein Karlsruhe in North Dakota/USA, dabei, was für unseren Mitfahrservice weniger relevant sein dürfte).

GeoService

Bitte legen Sie im Unterpackage `services.geo` Ihres Projekts ein Interface `GeoService` an wie folgt...

```
public interface GeoService {
    List<GeoAdresse> findeAdressen(String ort);
}
```

und für die Antworten einen Record-Typ `GeoAdresse` mit den `String`-wertigen Attributen `name`, `addresstype` und `display_name` sowie den beiden `double`-wertigen Attributen `lat` (*latitude*, geogr. Breite) und `lon` (*longitude*, geogr. Länge). Die Namen entsprechen denen aus der API-Antwort, das macht das Mapping der empfangenen JSON-Daten einfach.

Bitte implementieren Sie die `GeoService`-Schnittstelle im selben Package durch die Klasse `GeoServiceImpl` als Spring-Service. Die Implementierung von `findeAdressen(String ort);` soll folgendes leisten:

- Wenn für den Ort der `null`-Wert übergeben wurde, soll ein Fehler geloggt und als Ergebnis die leere Liste zurückgegeben werden.
- Mit dem Spring `WebClient` [1] werden bei der *Nominatim*-API Geo-Inforamtionen zum übergebenen `ort` im JSON-Format und beschränkt auf Deutschland abgefragt, so dass Sie eine Java-Liste von `GeoAdresse`-Objekten erhalten.
- Da wir nur Ortsnamen und nicht auch Landkreise etc. haben wollen, sollen in der Ergebnisliste der Methode nur Objekte enthalten sein, deren Attribut `addresstype` einen der folgenden Werte hat: `city`, `town`, `village`. Achten Sie bitte darauf, dass die Reihenfolge aus der API-Antwort erhalten bleibt.

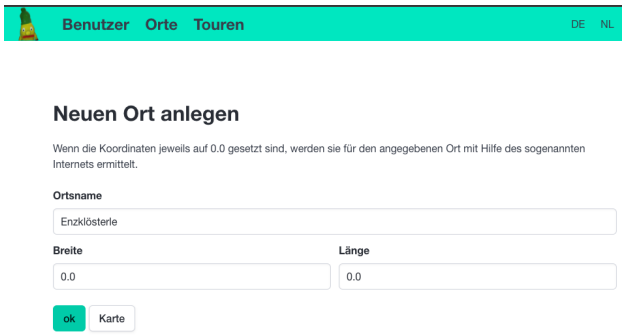
Nach dem `null`-Check lässt sich die gesamte Aufgabe mit einem einzigen (`return`-)Statement lösen (Hinweis: Java Stream-API), länglichere Lösungen sind aber auch zulässig.

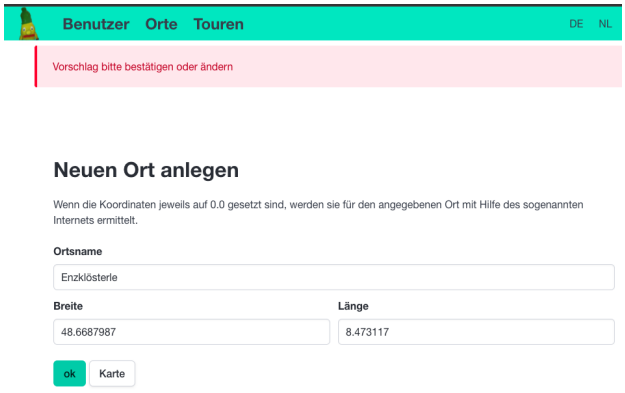
Integration in OrtServiceImpl

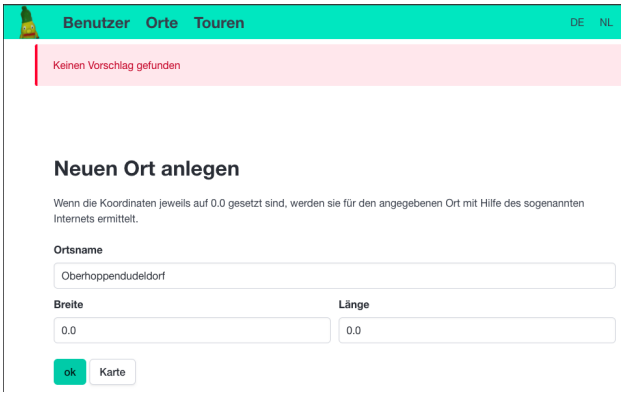
Abschließend wollen wir unseren neuen Service für die Neuerfassung von Orten nutzen. Bisher mussten wir neben dem Namen auch die Geo-Koordinaten von Hand eingeben, das soll jetzt automatisiert werden.

- Bitte ergänzen Sie Ihr `OrtService`-Interface um eine Methode `List<Ort> findeOrtsvorschlaegeFuerAdresse(String ort)` und implementieren Sie sie in Ihrem `OrtServiceImpl` so, dass mithilfe des `GeoService` eine Liste von Vorschlägen für den übergebenen `ort` ermittelt und dann (in gleicher Reihenfolge) eine Liste `Ort`-Objekten zurückgegeben wird, wobei Name und Geoinformationen aus den *Nominatim*-Antworten übernommen werden.
- Verwenden Sie die neue Methode Ihres `OrtService` bitte nun in Ihrem `OrtController`, so dass (nur) beim Anlegen eines neuen Ortes in dem Fall, dass nach Eingabe eines Ortsnamens beide Eingabefeldern für geogr. Breite/Länge unverändert gelassen wurden (also Wert 0.0), alle Formularfelder aus der ersten (Index 0) Antwort von `findeOrtsvorschlaegeFuerAdresse()` befüllt werden, aber noch *kein* Eintrag in die Datenbank gemacht wird. Die oder der Usende soll das Formular mit den ergänzten Eingabefeldern also gezeigt bekommen und kann dann entscheiden, ob er es so abschicken oder noch ändern möchte (siehe Screenshot für den Ort “Enzklösterle” unten, linke Seite).
- Um dem Benutzer einfach Feedback geben zu können, haben Sie bereits in Ihrem Kopfzeilen-Fragment die Möglichkeit vorgesehen, das `info`-Model-Attribut anzuzeigen, wenn es belegt ist. Hier ist eine schöne Möglichkeit, diese Funktionalität zu nutzen. Bitte belegen Sie `info` mit einem Hinweis, wenn (wie eben beschrieben) das Formular automatisch vorbefüllt wurde und bestätigt werden muss, oder für den Fall, dass kein Vorschlag für den eingegebenen Ortsnamen gefunden wurde, dass die Suche erfolglos war (siehe Screenshot für die Eingabe “Oberhoppendudeldorf” unten, rechte Seite).

[1] Dependency org.springframework.boot:spring-boot-starter-webflux nicht vergessen





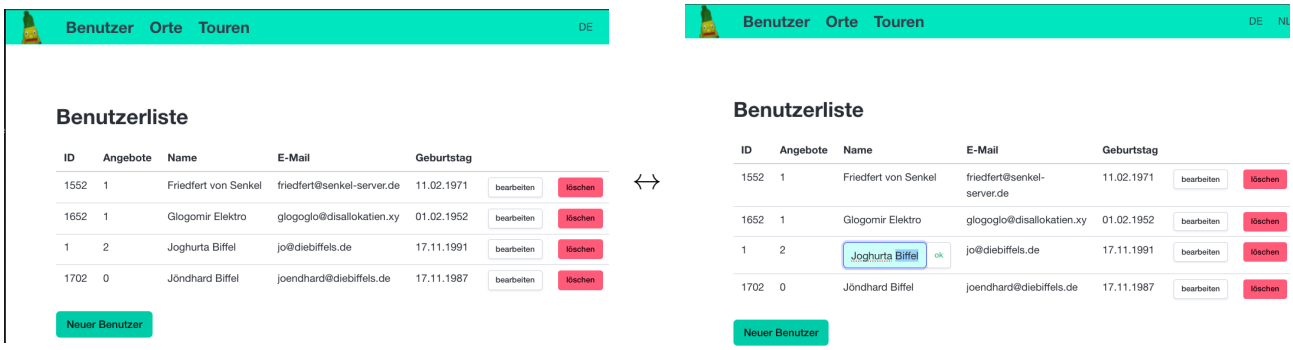


Aufgabe 2 (Inline-Edits in der Benutzerliste (HTMX))

Da die Benutzer der Plattform häufig ihre Namen und Mail-Adressen ändern, möchten wir diese Verbesserungen nun direkt in der Benutzerliste durchführen können, ohne jedes Mal das volle Bearbeitungsformular aufrufen zu müssen. HTMX hilft dabei.

Bitte ergänzen Sie im `HEAD` Ihres `benutzerliste.html`-Template das **Einbinden der HTMX-Bibliothek**. Sie finden Hinweise und aktuelle Codeschnipsel dazu auf https://htmx.org/docs unter “Installing”. Nun können Sie die HTMX-Attribute auf der Benutzerseite-Liste nutzen.

Künftig soll es so sein, dass das **Anklicken eines Namens oder einer E-Mail-Adresse** in der Benutzerliste dieser Tabelleneintrag durch ein **kleines Formular** ersetzt wird, das aus einem Text-`INPUT`-Feld, vorbelegt mit aktuellen Wert des zu bearbeitenden Feldes, und einem “OK-Button” zum Abschicken besteht. Danach erscheint an dieser Stelle wieder die anfängliche, “normale” Ausgabe des Eintrags, man kann **beliebig oft zwischen Anzeige- und Bearbeitungsmodus** je Feld wechseln:



Tabellenzellendarstellung in kleine Thymeleaf-Fragment auslagern

Um leicht zwischen den beiden Darstellungen eines Tabelleneintrags wechseln zu können bereiten wir im `benutzer`-Thymeleaf-Ordner eine neue Datei `benutzerliste-zeile` mit zwei Thymeleaf-Fragmenten vor. **Beide Fragmente** rendern einen HTML-Tabelleneintrag `<TD>` und haben **drei Parameter** `benutzerid`, `feldname` und `wert`.

Das **erste Fragment** `feldausgeben(benutzerid,feldname,wert)` ist für die **"normale Darstellung" einer Tabellenzelle** zuständig und gibt innerhalb des `<TD>` den übergebenen Wert in einem `<SPAN>`-Element aus. Bitte legen Sie dieses Fragment an und bauen Sie in Ihrem bestehenden `benutzerliste.html`-Template die Ausgabe der beiden Felder `name` und `email` so um, dass die Werte nicht direkt ausgegeben, sondern das jeweilige `<td>`-Element durch einen geeignet mit ID, Feldname und -Wert parametrisierten "`th:replace`" Ihres `feldausgeben`-Fragments ersetzt wird.

Tabellenzellen-Fragment in Benutzerliste nutzen

Nun ersetzen wir in unserer Benutzertabelle die bisherige Darstellung der Namen- bzw. E-Mail-Attribute durch Nutzung unseres neuen Fragments. Wenn Sie in `benutzerliste.html` z.B. mit der Laufvariablen `e` über die aufzulistenden Benutzerobjekte iterieren und die Tabellenzelle für die E-Mail-Adresse erzeugt werden soll, verwenden Sie nun anstelle des alten

```
<td th:text='${e.email}'></td>
```

Ihr Fragment zur Ausgabe des `email`-Feldes, z.B. so (Zeilenumbruch bitte wegdenken):

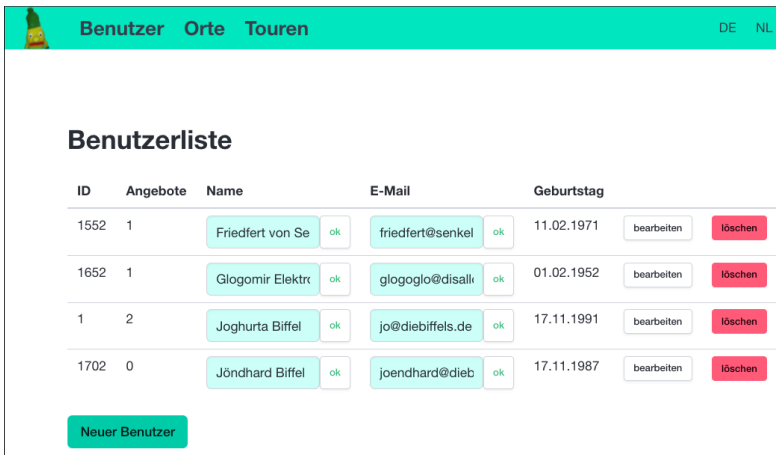
```
<td th:replace="~{benutzer/benutzerliste-zeile ::
            feldausgeben(benutzerid=${e.id},feldname=email,wert=${e.email})}">
</td>
```

Testen Sie danach bitte Ihre Anwendung. Die Benutzerliste sollte äußerlich betrachtet genauso funktionieren wie zuvor.

Tabellenzellen-Bearbeitungsfragment anlegen

Das **zweite Fragment** in `benutzerliste-zeile.html` dient zum Editieren des gewünschten Feldes und heißt `feldbearbeiten(benutzerid,feldname,wert)`. Dazu gibt es innerhalb des von ihm erzeugten `<TD>` ein HTML-FORM aus, das ein INPUT-Feld, dessen Name (`name`-Attribut) fix auf `wert` gesetzt ist und dessen `value`-Attribut mit dem übergebenen `wert`-Fragment-Parameterwert belegt wird (kein Thymeleaf "`th:field`" etc, normales HTML). Beachten Sie bitte, dass das Wort `wert` hier zweimal in verschiedener Bedeutung auftaucht, einmal ans HTML-`name` des `INPUT`-Elements und einmal als Fragment-Parameter. Abschließend soll im FORM ein Button vorgesehen werden.

Bitte **testen** Sie dieses zweite Fragment, indem Sie in Ihrer `benutzerliste.html` in den `th:replace`-Ausdrücken den Fragmentnamen `feldausgeben` durch `feldbearbeiten` ersetzen. Ihre Benutzerliste müsste nun alle Namen und E-Mails als Mini-Formulare darstellen:



Sobald das funktioniert, verwenden Sie in `benutzerliste.html` bitte wieder durchgängig das `feldausgeben`-Fragment für die initiale Darstellung.

Hinweis: Wir verlassen uns in diesem Fall für die Validierung auf die Datenbank-Operationen und legen für das "Miniformular" im `feldbearbeiten`-Fragment mit nur einem Eingabewert *keine* extra Java-Formularklasse

mit eigener Validierung an (obwohl dies möglich wäre), sondern greifen den eigegebenen Wert später im Handler einfach als `@RequestParam` ab.

Tabellenzellen anklickbar machen

Nun erweitern wir unser `feldausgeben`-Fragment um etwas HTMX. Wenn der `<SPAN>` angeklickt wird, soll per HTMX ein HTTP-**GET**-Request an den Endpunkt `/benutzer/{id}/hx/feld/{feldname}` erfolgen, der später implementiert wird. Die Antwort soll das nächstliegende (den Eintrag umfassende) `<TD>`-Element komplett ersetzen, unser Tabelleneintrag wird also durch einen neuen, vom Server geschickten ersetzt.

Analog soll im `feldbearbeiten`-Fragment bei einem Klick auf den Formular-Button ein HTTP-**PUT**-Request auf den Endpunkt `/benutzer/{id}/hx/feld/{feldname}` durchführt werden (hier wird der Inhalt des INPUT-Elements namens `wert` automatisch mit an den Server übertragen, da im selben FORM). Wie eben soll die Serverantwort das umgebende, nächstliegende `<TD>`-Element komplett ersetzen.

Die Platzhalter `id` und `feldname` in den URI-Pfaden sind (wie immer) in beiden Fällen durch die in das jeweilige Fragment hereingereichten Parameterwerte zu ersetzen, so dass in der `email`-Tabellenzelle in der Zeile für den Benutzer mit der ID 123 der Endpunkt `/benutzer/123/hx/feld/email` verwendet wird usw. Sie können sich im Browser den erzeugten HTML-Quellcode ansehen, um zu überprüfen, ob das korrekt umgesetzt ist.

Hinweis: Zum Zusammenbauen von URLs können Sie bei Spring / Thymeleaf Platzhalter im URI-Pfad vorsehen und diese mit Werten füllen. Wie immer muss Attributen, die Ausdrücke enthalten, ein `th:` vorangestellt werden, die Pfadparameternamen und -werte werden in einer Klammer direkt danach aufgezählt. Nehmen wir an, wir haben (z.B. Fragment-Parameter oder in unserem `Model`) die Attribute `myid` mit dem Wert 123 und `myfeldname` mit dem Wert `email` vorbelegt. Dann würde dieser Beispielcode ein `hx-get` mit der oben genannten Beispiel-URL erzeugen:

```
<SPAN th:hx-get="@{/benutzer/{id}/hx/feld/{feldname}(id=${myid},feldname=${myfeldname})}" ... >
```

BenutzerService ergänzen

Wir benötigen sicher eine Funktion, um gewünschte Attribute (hier nur `name` oder `email`) eines Benutzers (mit einer `id`) in der Datenbank auf einen neuen Wert setzen können.

Bitte ergänzen Sie ihr Interface `BenutzerService` und dessen Implementierung um eine neue Methode `Benutzer aktualisiereBenutzerAttribut(long id, String feldname, String wert)`, welche für den `Benutzer` mit der ID `id` das Attribut `feldname` auf den Wert `wert` setzt und das geänderte Benutzerobjekt zurückgibt. Als `feldname` kommen in unserem Fall die Strings `name` oder `email` infrage. Denken Sie an Transaktionssicherheit.

Handler-Methoden im `BenutzerController`

Nun brauchen wir für den in unserem Fragmenten für die HTMX-Anfragen verwendeten URI-Pfad `/benutzer/{id}/hx/feld/{feldname}` **zwei** Handler-Methoden in unserem `BenutzerController`. In beiden Fällen müssen beim Abschluss im `Model` zwei Attribute belegt sein, um das anschließend zu rendernde Folge-Fragment zu füllen: `feldname` (der aus dem Pfadparameter übernommen), `benutzerid` (aus dem `id`-Pfadparameter übernommen) und ein `wert`-String, der in der jeweiligen Handlermethode ermittelt wird.

Beide Handlermethoden geben (wie zu erwarten) als View-ID die Namen eines der beiden Fragmente in `benutzerliste-zeile.html` zurück.

- bei Abruf mit **HTTP GET** wird der `Benutzer` mit der übergebenen `id` aus der Datenbank geholt und das `Model`-Attribut `wert` mit dessen Namen oder E-Mail-Adresse belegt, je nachdem, was über den `feldname`-Pfadparameter gefragt war. Danach gibt die GET-Handlermethode als View-ID das `feldbearbeiten`-Fragment zurück (`benutzer/benutzerliste-zeile :: feldbearbeiten`).

Gemäß der oben gemachten Schritte wird so eine angeklickte Tabellenzelle in “Normaldarstellung” nun per HTMX durch die Bearbeitungs-Darstellung mit Mini-Formular ersetzt (dieses Mini-Formular ist ja gerade der Inhalt der HTTP-Antwort, HTMX baut es nur an der gewünschten Stelle in die Ansicht ein).

- Umgekehrt soll bei einem Abruf mit **HTTP PUT**, also nach Bearbeitung eines Attributwerts und dem “Abschicken” eines Mini-Formulars, das Fragment `feldausgeben` von `benutzerlist-zeile.html` für die “Normalansicht” der Tabellenzelle gerendert werden. Im Gegensatz zum GET-Handler erwarten wir beim PUT, das ja aus einem Bearbeitungs-Miniformular getriggert wird, die Übergabe des eingegebenen Wertes aus dem `wert`-INPUT-Formularfeld. Gemäß der oben gemachten Schritte hat das INPUT-Feld den Namen `wert`, also können wir dessen Inhalt leicht als `@RequestParam("wert")` abgreifen.

  Zur Verarbeitung nutzen wir die neue `BenutzerService`-Methode `aktualisiereBenutzerAttribut()`, um die gewünschte Änderung an Name bzw. E-Mail vorzunehmen.

  Falls die **Attributänderung erfolgreich** war, lassen wir das Fragment `feldausgeben` rendern und der Benutzer sieht seine gerade bearbeitete Tabellenzelle wieder in Normaldarstellung.

  Sollten nach dem Bearbeiten ein leeres Eingabfeld abgeschickt oder anderweitig gegen die Validierungsregeln Ihrer `@Benutzer`-Entität verstoßen werden, bemerken Sie dies spätestens beim Versuch, das Objekt in der Datenbank zu aktualisieren (Exception). Der Benutzer wäre bestimmt nicht traurig, wenn er sich nach einer vergurkten Bearbeitung nicht an den alten (zulässigen) Feldinhalt erinnern müsste, sondern ihn erneut zur Bearbeitung vorgelegt bekäme.

  Sollte daher **bei der Attributänderung eine Exception** auftauchen (z.B. wegen eines Validierungsfehlers beim Persistieren der Änderung), setzen wir im `Model` den `wert` auf den aktuellen Inhalt des zugehörigen `Benutzer`-Objekts (Datenbank fragen, die `id` haben wir ja) und lassen erneut das Fragment `feldbearbeiten` rendern, um dem Benutzer eine Korrektur zu ermöglichen. In diesem Fall wird **keine** Änderung des `Benutzer`-Objekts in der Datenbank persistiert.

Akzeptanzkriterien für diesen Projektstand:

- Beim Anlegen eines neuen Ortes wird der *Nominatim*-Service zur Ermittlung der Geo-Koordinaten herangezogen, wenn nur ein Ortsname angegeben wurde und die Koordinaten-Eingabefelder auf “0.0” gelassen wurden.
- Die so vervollständigten Ortsdaten werden *nicht* gleich in die Datenbank gespeicher, sondern der Nutzer sieht das komplettierte Formular. Wenn er es abschickt, erfolgt die Speicherung des neuen Ortes.
- Falls zum eingegebenen Ortsnamen kein *Nominatim*-Vorschlag gefunden wurde, wird der Benutzer durch eine einfach Ausgabe des `info`-Attributs (in einer früheren Übung bereits erstellte Hinweisbox im Kopfzeilen-Fragment) darauf hingewiesen.
- In der Benuterliste erscheinen nach Anklicken eines Namen oder einer E-Mail-Adressen an der betreffenden Stelle ein Formular mit Eingabefeld (vorbelegt mit dem aktuellen Wert des zu bearbeitenden Attributs) und einem OK-Button (HTMX).
- Abschicken des (geänderten) Wertes führt zur Aktualisierung des zugehörigen `Benutzer`-Objekts in der Datenbank, wenn das Abspeichern erfolgreich war, wird das Mini-Formular wieder durch die Normale Tabellenzellen-Darstellung ersetzt (HTMX)

- Sollte das Abspeichern nicht erfolgreichn gewesen sein, bleibt das Mini-Formular stehen und enthält den aktuellen (unveränderten) Attributwert für den bearbeiteten `Benutzer` aus der DB (HTMX).
- Beliebig viele Namens/E-Mail-Felder in der Benutzerliste können unabhängig voneinander in den Bearbeitungs- bzw. den Normalanzeige-Modus versetzt und bearbeitet werden (HTMX).