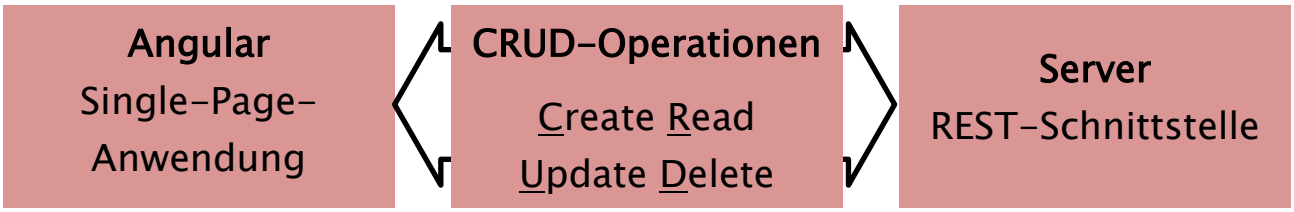
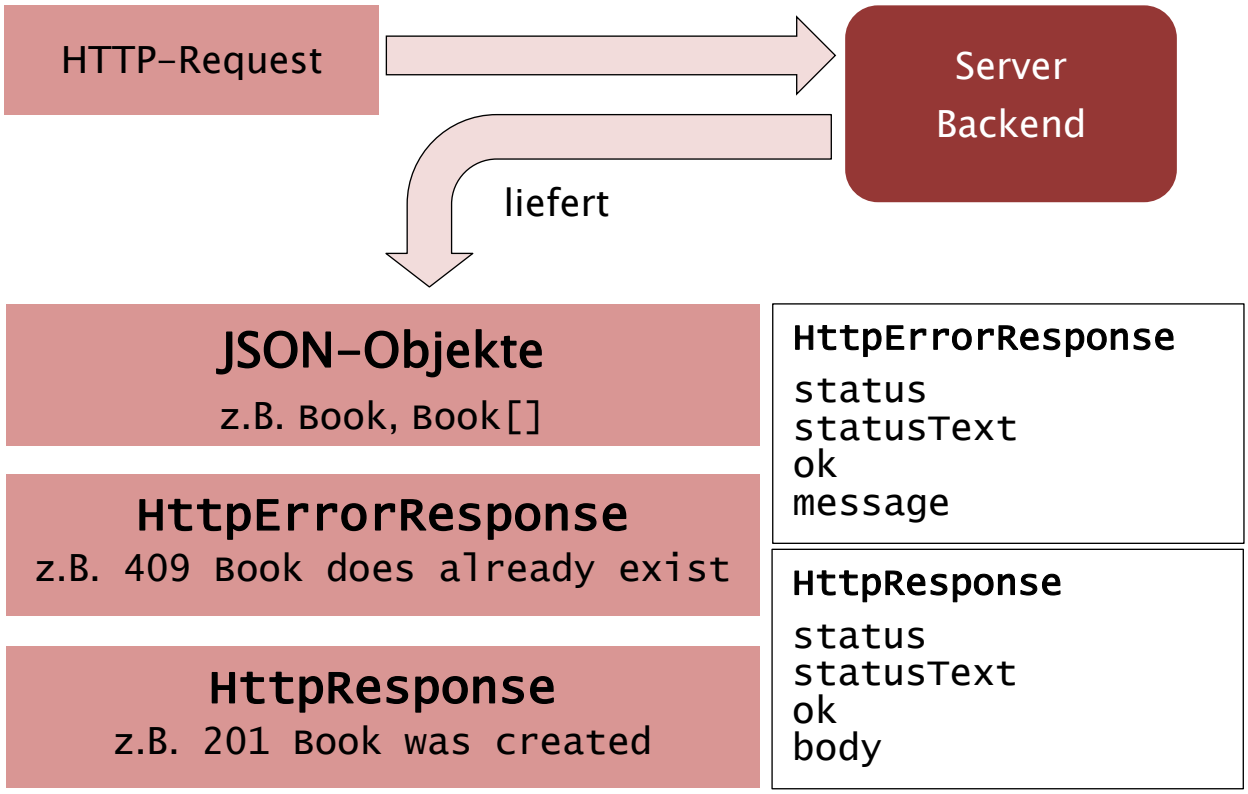
Angular: Zugriff auf Web-Services und Formularverarbeitung

- Über die Klasse `HttpClient` auf einen Web-Service zugreifen und dort die CRUD-Operationen ausführen können
- Bei Zugriff die Fehlerbehandlung richtig einsetzen können
- Wissen was Pipes sind und wie insbesondere mit AsyncPipe gearbeitet werden kann
- Das Konzept der Formularverarbeitung mit Reactive Forms verstehen und in Kombination mit Angular Material Form Controls anwenden können
- Standard-Fehlerbehandlung implementieren können, so dass Fehler angezeigt werden
- Mit dem `FormBuilder` ein Eingabemodell für das Formular erstellen können
- Das Eingabemodell mit dem Formular verbinden können
- Auf die einzelnen Control-Elemente zugreifen, deren Eigenschaften manipulieren und Werte auslesen können
- Eigene Validierer schreiben und einbinden können
- Mit asynchronen Validierern arbeiten und diese implementieren können

HttpClient



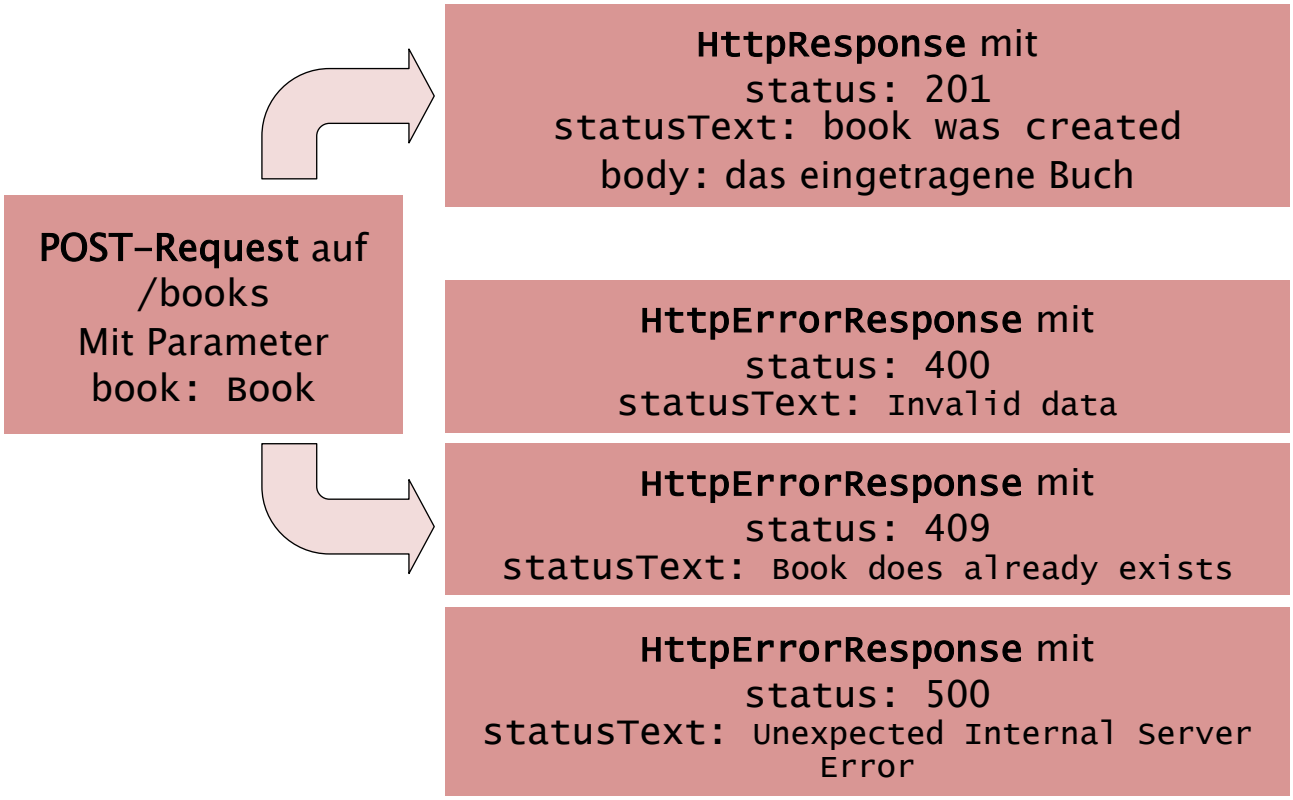
- Klasse `HttpClient` implementiert `Observable`-Pattern um Abfragen an Server Backend zu formulieren, der *asynchron* antwortet



- `HttpClientModule` in `AppModule` von `@angular/common/http` importieren
- In HTTP-Klasse ist `unsubscribe()` nicht notwendig, denn nach Antwort des Servers wird `unsubscribe()` automatisch aufgerufen
- Es ist einstellbar, ob bei Erfolg direkt die Daten (z.B. `Book[]`) oder der `HttpResponse` mit den Daten im Body zurückgeliefert werden (siehe hinten *))

Beispielservice BookMonkey 5 API[1]

Beispiel Hinzufügen eines Buches



[1] Lokale Installation siehe Rohdateien oder https://github.com/angular-buch/api5. Exakte Definition des Services unter https://api5.angular-buch.com/swagger-ui/#/books

Die BookStoreServices-Klasse

```
const URL = 'http://localhost:3000';
@Injectable()
export class BookStoreService {
  constructor(private http: HttpClient) { }
  getAllBooksSearchTerm(searchTerm: string): Observable<Book[]> {
    return
      this.http.get<Book[]>(`${URL}/books/search/${searchTerm}`);
  }
  getBook(isbn: string): Observable<Book> {
    return this.http.get<Book>(`${URL}/books/${isbn}`);
  }
  createBook(book: Book): Observable<Book> {
    return this.http.post<Book>(`${URL}/books`, book);
  }
  createBookResponse(book: Book): Observable<HttpResponse<Book>> {
    return this.http.post<Book>(
      `${URL}/books`, book, { observe: 'response' }*);
  }
  deleteBook(isbn: string): Observable< HttpResponse<Object>> {
    return this.http.delete<Object>(`${URL}/books/${isbn}`,
      { observe: 'response' })**;
  }
}
```

- `HttpClient` muss in den Service injiziert werden
- *) Dadurch wird das `HttpResponse`-Objekt zurück geliefert welches im Body das erfolgreich eingetragene Buch-Objekt enthält
- **) **ACHTUNG**: Web-Service liefert auch dann ein `HttpResponse`-Objekt mit `status: 200`, `statusText: OK` und `body: { success: true }` zurück, wenn zu löschendes Buch nicht existiert

Integration in die Komponente

```
export class AppComponent implements OnInit {
  book: Book | null = null; books: Book[] | null = null;
  error: HttpErrorResponse | null = null;
  response: HttpResponse<any> | null = null;

  constructor(private bs: BookStoreService) {}

  ngOnInit(): void {
    this.bs.getAllBooksSearchTerm('Angular')
      .subscribe(books => this.books = books);

    this.bs.getBook(NOT_EXISTING_ISBN)
      .subscribe({
        next: book => this.book = book,
        error: error => this.error = error
    });

    const newBook1: Book =
      { isbn: '1', title: '11', authors: [ 'A1' ]};
    this.bs.createBook(newBook1)
      .subscribe({
        next: book => this.book = book,
        error: error => this.error = error
      });

    const newBook2: Book =
      { isbn: '2', title: 'T2', authors: [ 'A2' ]};
    this.bs.createBookResponse(newBook2)
      .subscribe({
        next: response => {
          this.response = response;
          this.book = response.body as Book;
        },
        error: error => this.error = error
      });

    this.bs.deleteBook(NOT_EXISTING_ISBN)
      .subscribe(response => this.response = response);
  };
}
```

9783864909467, Angular (4. Auflage)
9783864903571, Angular (1. Auflage)
9783864906466, Angular (2. Auflage)
9783864907791, Angular (3. Auflage)

Error

Status: 404
StatusText: Not Found
ok: false
message: Http failure response for ht

Book

1, T1

Book

2, T2

Response

Status: 201
StatusText: Created
ok: true
body: [object Object]

Response

Status: 200
StatusText: OK
ok: true
body: [object Object]

```
<div *ngIf="book">
  {{ book.isbn}}, {{ book.title }}
</div>

<div *ngFor="let book of books">
  {{ book.isbn}}, {{ book.title }}
</div>

<div *ngIf="error">
  <b>Status:</b> {{ error.status }}<br>
  <b>StatusText:</b> {{ error.statusText }}<br>
  <b>ok:</b> {{ error.ok }}<br>
  <b>message:</b> {{ error.message }}
</div>

<div *ngIf="response">
  <b>Status:</b> {{ response.status }}<br>
  <b>StatusText:</b> {{ response.statusText }}<br>
  <b>ok:</b> {{ response.ok }}<br>
  <b>body:</b> {{ response.body }}
</div>
```

Elegantere Lösung mit AsyncPipe…

```
export class AppComponent implements OnInit {
  $books!: Observable<Book[]>;
  constructor(private bs: BookStoreService) { }
  ngOnInit() {
    this.$books = this.bs.getAllBooksSearchTerm('Angular');
  }
}
```

```
<div *ngFor="let book of $books | async">
  {{ book.isbn }} {{ book.title }}
</div>
```

- Pipes dienen zur Transformation von Werten vor der Ausgabe, z.B. *Upper/LowerCase-*, *Slice*[2]*-*, *JSON-*, *Decimal-*, *Date-*, *Percent-*, *CurrencyPipe*
- *AsyncPipe* übernimmt das Subskribieren und folglich das asynchrone Warten auf die Ergebnisse, die dann angezeigt werden
- Daten werden nicht in Komponente abgespeichert, sondern lediglich das Observable das Daten liefert
- Obervable-Stream laut gängiger Programmierpraxis mit $

[2] Generiert ein Array aus einer Menge von Werten

Formularverarbeitung mit Reactive Forms

- Eingabedaten können zentral validiert und verarbeitet werden
- Auf jede Änderung im Formular kann reagiert werden
- Benutzer bekommt Feedback über seine Eingaben

- Fehlermeldung soll erst dann erscheinen, wenn Eingabefeld geändert wird
- Passwort soll mindestens 8 Zeichen lang sein
- Benutzername, Passwort, Vor- und Nachname müssen eingegeben werden
- E-Mailadressen müssen korrekt eingegeben werden, evtl. Ausgabe von drei Fehlermeldungen
- *Registrieren*-Knopf nur dann aktiv, wenn keine Fehler erkannt wurden
- (Bei sehr kleinen Bildschirmen soll Vor- und Nachname untereinander angezeigt werden)

Voraussetzung

In `AppModule` muss `ReactiveFormsModule` aus `@angular/forms` importiert werden

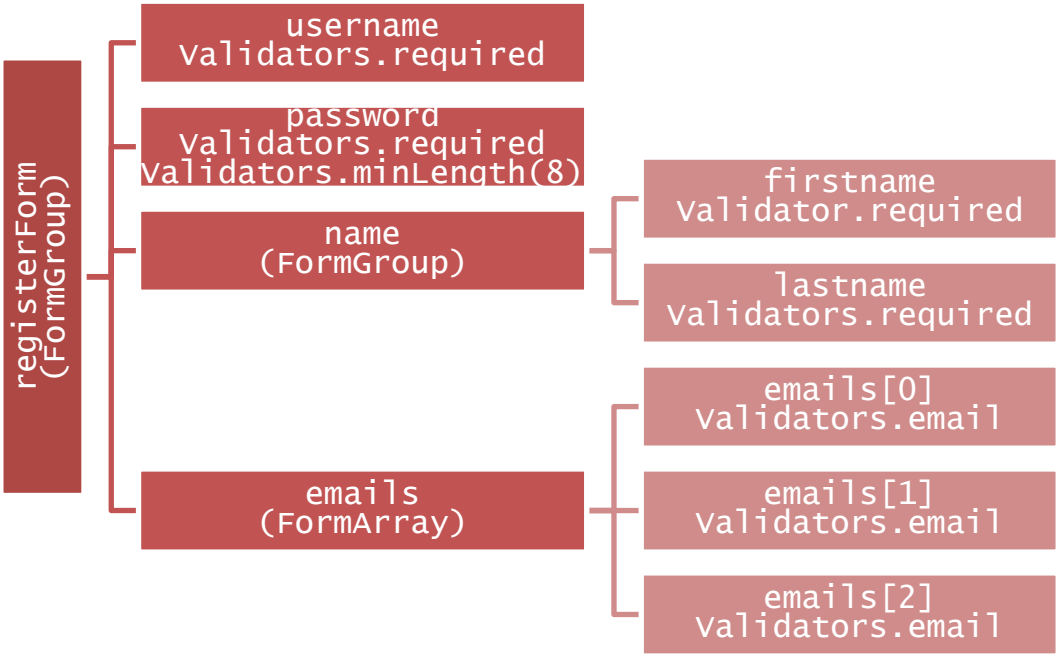
Vorarbeiten

```
export class User {
  constructor(
    public username: string,
    public password: string,
    public name: Name,
    public emails: Array<string>
  ) { }
}

export class Name {
  constructor(
    public firstname: string,
    public lastname: string
  ) { }
}
```

Die Komponente

- In Formularkomponente muss *Modell* erstellt werden welches aus `FormGroups`, `FormControls` und `FormArrays` besteht. Dabei hilft `FormBuilder`
- Diese können Validierer enthalten
- Dem Modell müssen die Daten übergeben werden
- Komponente muss nach Formulareingabe die Daten übernehmen

Das Modell



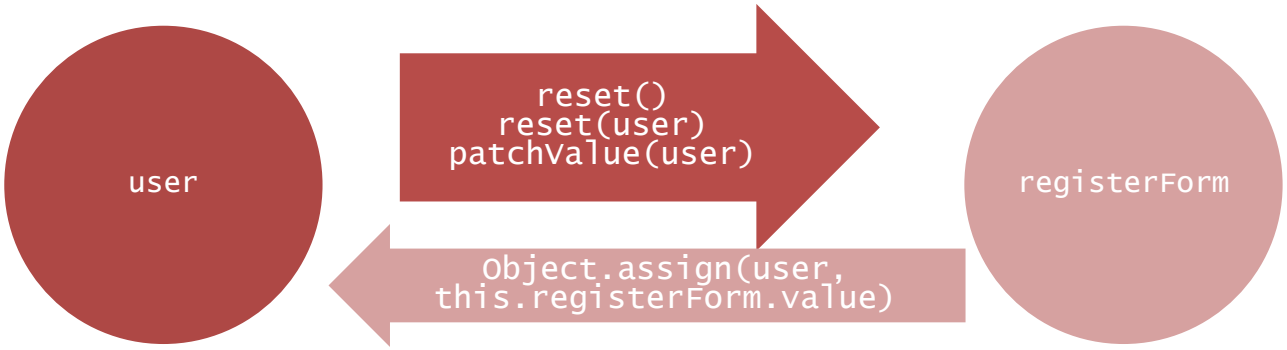

```
export class RegisterFormComponent implements OnInit {
  registerForm!: FormGroup;
  constructor(private fb: FormBuilder) { }
  ngOnInit() {
    const user: User = UserFactory.empty();
    this.registerForm = this.fb.group({
      username: [ user.username, {
        validators: Validators.required
      }],
      password: [ user.password, {
        validators: [ Validators.required,
        Validators.minLength(8) ]
      }],
      name: this.fb.group({
        firstname: [ user.name.firstname, {
          validators: Validators.required
        }],
        lastname: [ user.name.lastname, {
          validators: Validators.required
        }]
      }),
      emails: this.fb.array([
        [ user.emails[0], { validators: Validators.email } ],
        [ user.emails[1], { validators: Validators.email } ],
        [ user.emails[2], { validators: Validators.email } ]
      ])
    });
  }
  get emails(): FormArray {
    return this.registerForm.get('emails')[3] as FormArray;
  }
```

Fortsetzung nächste Seite

[3] Rückgabetyp von `get()` ist `AbstractControl`. Hier wird in `FormArray` konvertiert

```
  register() {
    const user = UserFactory.empty();
    Object.assign(user, this.registerForm.value);
    …
    this.registerForm.reset(UserFactory.empty());
  }
}
```

- `get emails()` ist Getter-Methoden zum einfacheren Zugriff auf das Control-Element
- `FormBuilder` erstellt Model (`registerForm`) mit entsprechenden Formularelementen (`FormControl`, `FormGroup` und `FormArray`)
- Model wird nachfolgend mit Formular verknüpft
- Elemente werden im Formular mit den Eingabefeldern verknüpft (`FormControlName`, `FormGroupName` und `FormArrayName`)
- Zusätzlich verfügbare *Standard-Validierer* `maxLength()` und `pattern()`
- `reset()` ohne Parameter löscht alle Inhalte in Formularelementen
- Mit `this.registerForm.patchValue(user);` können in `ngOnInit()` Daten des Benutzers ins Formular übertragen werden



Das Template

```
<form fxLayout="column" [formGroup]="registerForm"
  (ngSubmit)= "register()" autocomplete="off">

  <mat-form-field fxFlex>
    <input matInput formControlName="username"
      placeholder="Benutzername" type="text">
    <mat-error>Der Benutzername muss eingegeben werden</mat-error>
  </mat-form-field>

  <mat-form-field fxFlex>
    <input matInput formControlName="password"
      placeholder="Passwort" type="password">
    <mat-error
      *ngIf="registerForm.get('password')?.hasError('required')">
      Das Passwort muss eingegeben werden
    </mat-error>
    <mat-error
      *ngIf="registerForm.get('password')?.hasError('minlength')">
      Das Passwort muss mind. 8 Zeichen lang sein
    </mat-error>
  </mat-form-field>

  <div fxFlex fxLayout.lt-sm="column" formGroupName="name">
    <mat-form-field fxFlex>
      <input matInput formControlName="firstname"
        placeholder="Vorname" type="text">
      <mat-error>Der Vorname muss eingegeben werden</mat-error>
    </mat-form-field>
    <mat-form-field fxFlex>
      <input matInput formControlName="lastname"
        placeholder="Nachname" type="text">
      <mat-error>Der Nachname muss eingegeben werden</mat-error>
    </mat-form-field>
  </div>

  <div fxFlex fxLayout.lt-sm="column" formArrayName="emails">
    <mat-form-field fxFlex
      *ngFor="let control of emails.controls; index as i">
      <input matInput [formControlName]="i"
        placeholder="E-Mailadresse" type="text">
      <mat-error>Eine korrekte E-Mailadresse muss …</mat-error>
    </mat-form-field>
  </div>

  <button fxFlex mat-raised-button type="submit"
    [disabled]="!registerForm.valid">Registrieren</button>
</form>
```

FormControl, FormGroup und FormArray

- `value`
  Hat lesenden Zugriff auf den Wert des/r Eingabefeldes/r (anhand von `this.registerForm.value.emails` können eingegebene E-Mailadressen gelesen und geändert werden). Schreibender Zugriff mit `setValue()`
- Enthält Infos zur *Validität* und zum *Status*:

  `dirty`
  wenn Eingabefeld geändert wurde

  `invalid`
  wenn mindestens ein Validierer Probleme erkannt hat

  `errors`
  enthält Info welcher Validierer welchen Fehler erkannt hat
- Mit `get('password')` kann auf die enthaltenen Komponenten zugegriffen werden
- `controls` enthält alle Komponenten als Array
- Mit `hasError('minlength')` kann auf Fehler kontrolliert werden

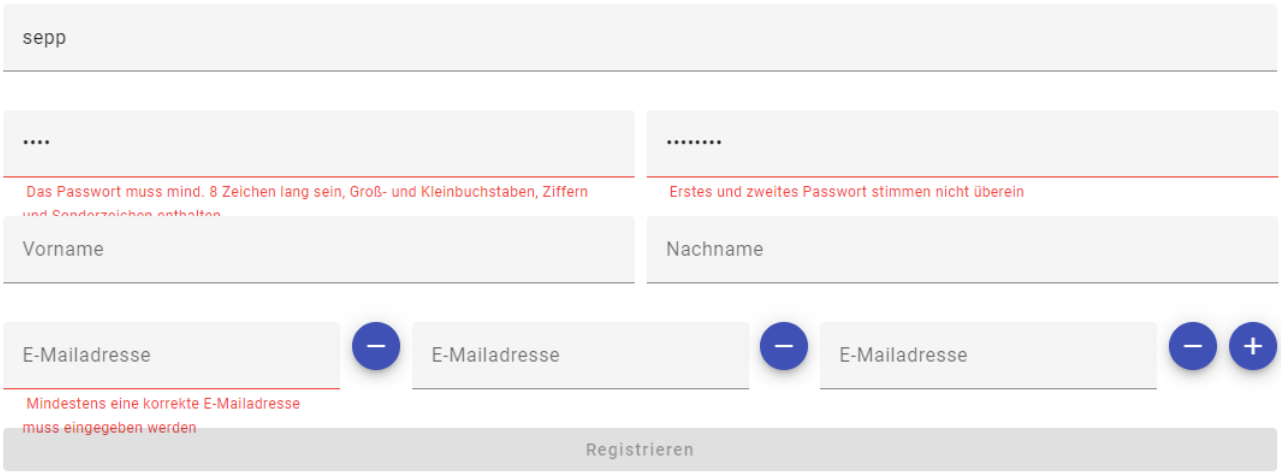
Der globale ErrorStateMatcher-Provider

Sorgt dafür, dass Fehlermeldungen in allen Eingabeelementen bereits sofort -auch während der Eingabe im Element -angezeigt werden (z.B. während zu kurzes Passwort eingegeben wird, wird Fehlermeldung schon sichtbar)

In `AppModule`

```
import { ErrorStateMatcher, ShowOnDirtyErrorStateMatcher }
  from '@angular/material/core';

…
providers: [{
  provide: ErrorStateMatcher,
  useClass: ShowOnDirtyErrorStateMatcher
}]
```

Eigene Validierer schreiben



- Das Passwort muss einem speziellen Muster entsprechen
- Das Passwort muss wiederholt eingegeben werden und übereinstimmen
- Die eingegebenen E-Mailadressen müssen alle korrekt sein
- Es muss mindestens eine korrekte E-Mailadresse eingegeben werden
- E-Mailadressen können hinzugefügt/gelöscht werden
- Leere E-Mailadresse wird bei *submit* aus Array gelöscht
- (Bei mehr als drei E-Mailadressen werden diese untereinander angezeigt)

Validierer allgemein

- In shared vorhandene Klasse enthält Validiermethoden
- Statische Validiermethode erhält als Parameter `AbstractControl`
- Bei *korrekter Validierung* wird `null` zurück geliefert
- Bei *falscher Validierung* wird `ValidationErrors` *<Name der Methode>*: `true` zurück geliefert

```
export class UserValidators {
  static passwordFormat(fc: AbstractControl):
    ValidationErrors | null {
    const passwordPattern = new RegExp('^(?=.*?[A-Z])
      (?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$%^&*-]).{8,}$');
    return passwordPattern.test(fc.value) ? null :
      { passwordFormat: true };
  }

  static passwordSame(fg: AbstractControl):
    ValidationErrors | null {
    const password =
      (fg as FormGroup).get('password')!.value;
    const passwordRepeat =
      (fg as FormGroup).get('passwordRepeat')!.value;
    return (password === passwordRepeat) ? null :
      { passwordSame: true };
  }

  static atLeastOneEmail(fa: AbstractControl):
    ValidationErrors | null {
    const containsValue = (fa as FormArray).controls.some(
      e => e.value ? true : false);
    return containsValue ? null : { atLeastOneEmail: true };
  }
}
```

`ValidationErrors` ist folgendermaßen definiert:

```
type ValidationErrors = {
  [key: string]: any;
};
```

Die Komponente (mit registrierten Validierern)

Es muss eine `FormGroup` mit dem Namen `passwords` erstellt werden, welche die `FormControls` `password` und `passwordRepeat` enthält damit dem Validierer alle beiden `FormControls` übergeben werden können

PROBLEM: Der Fehler `passwordSame` der in `passwords` ausgelöst wird, wird im Eingabefeld `passwordRepeat` nicht angezeigt weil dieses meint es hätte keinen!!!
LÖSUNG: `PasswordCrossFieldErrorMatcher` der in der Komponente angelegt und im Template zugewiesen wird bestimmt, ob Feld Fehler anzeigen soll
Dasselbe Problem löst `EmailCrossFieldErrorMatcher`

```
class PasswordCrossFieldErrorMatcher implements ErrorStateMatcher{
  isErrorState(control: FormControl | null,
    form: FormGroupDirective | NgForm | null): boolean {
    return control?.dirty &&
      form?.form.get('passwords')?.hasError('passwordSame') ?
        true : false;
  }
}
class EmailCrossFieldErrorMatcher implements ErrorStateMatcher {
  isErrorState(…): boolean {
    return control?.dirty &&
      (form?.form.get('emails')?.hasError('atLeastOneEmail') ||
      control?.hasError('email')) ? true : false;
  }
}
export class RegisterFormComponent implements OnInit {
  …
  passwordErrorMatcher = new PasswordCrossFieldErrorMatcher();
  emailErrorMatcher = new EmailCrossFieldErrorMatcher();
  ngOnInit() {
    const user: User = UserFactory.empty();
    this.registerForm = this.fb.group({
      username: …,
      passwords: this.fb.group({
        password: [ user.password, {
          validators: UserValidators.passwordFormat
        }],
        passwordRepeat: [ '' ]
      }, { validators: UserValidators.passwordSame }),
      name: this.fb.group({
        firstname: …, lastname: …
      }),
      emails: this.fb.array([
        [ user.emails[0], { validators: Validators.email } ],
        [ user.emails[1], { validators: Validators.email } ],
        [ user.emails[2], { validators: Validators.email } ]
      ], { validators: UserValidators.atLeastOneEmail })
    });
  }
  get passwords(): FormGroup {
    return this.registerForm.get('passwords') as FormGroup;
  }
  addEmailControl() {
    this.emails.push(this.fb.control(
      '', { validators: Validators.email }));
  }
  removeEmailControl(i: number) {
    this.emails.removeAt(i);
  }
  register() {
    // Streichen von leeren E-Mail-Adressen
    this.registerForm.value.emails =
      this.registerForm.value.emails.filter((email: any)=>email);
  }
}
```

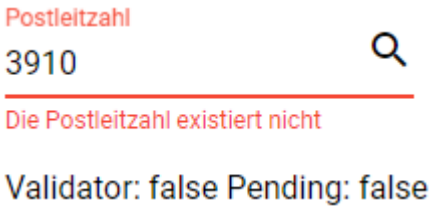
Das geänderte Template

```
<form …>
  …
  <div fxFlex fxLayout.lt-sm="column" formGroupName="passwords">
    <mat-form-field fxFlex>
      <input matInput formControlName='password'
        placeholder="Passwort" type="password">
      <mat-error>
        Das Passwort muss mind. 8 Zeichen lang sein, Groß- und…
      </mat-error>
    </mat-form-field>

    <mat-form-field fxFlex>
      <input matInput formControlName='passwordRepeat'
        placeholder="Passwortwiederholung" type="password"
        [errorStateMatcher]="passwordErrorMatcher">
      <mat-error>
        Erstes und zweites Passwort stimmen nicht überein
      </mat-error>
    </mat-form-field>
  </div>

  <div fxFlex
    [fxLayout]="emails.value.length <= 3 ? 'row' : 'column'"
    fxLayout.lt-sm="column" formArrayName="emails">
    <div fxFlex
      *ngFor="let control of emails.controls; index as i">
      <mat-form-field fxFlex>
        <input matInput [formControlName]="i"
          placeholder="E-Mailadresse" type="text"
          [errorStateMatcher]="emailErrorMatcher">
        <mat-error *ngIf="emails.controls[i].hasError('email')">
          Eine korrekte E-Mailadresse muss eingegeben werden
        </mat-error>
        <mat-error *ngIf="emails.hasError('atLeastOneEmail')">
          Mindestens eine korrekte E-Mailadresse muss …
        </mat-error>
      </mat-form-field>

      <button mat-mini-fab color="primary"
        [disabled]="emails.value.length===1 ? 'true' : 'false'"
        type="button"[4] (click)="removeEmailControl(i)">
        <mat-icon>remove</mat-icon>
      </button>
    </div>
  </div>

    <button mat-mini-fab color="primary" type="button"
      (click)="addEmailControl()">
      <mat-icon>add</mat-icon>
    </button>
  </div>
  …
</div>
```

[4] Damit wird das Abschicken des Formulars an den Server verhindert

Asynchrone Validierer

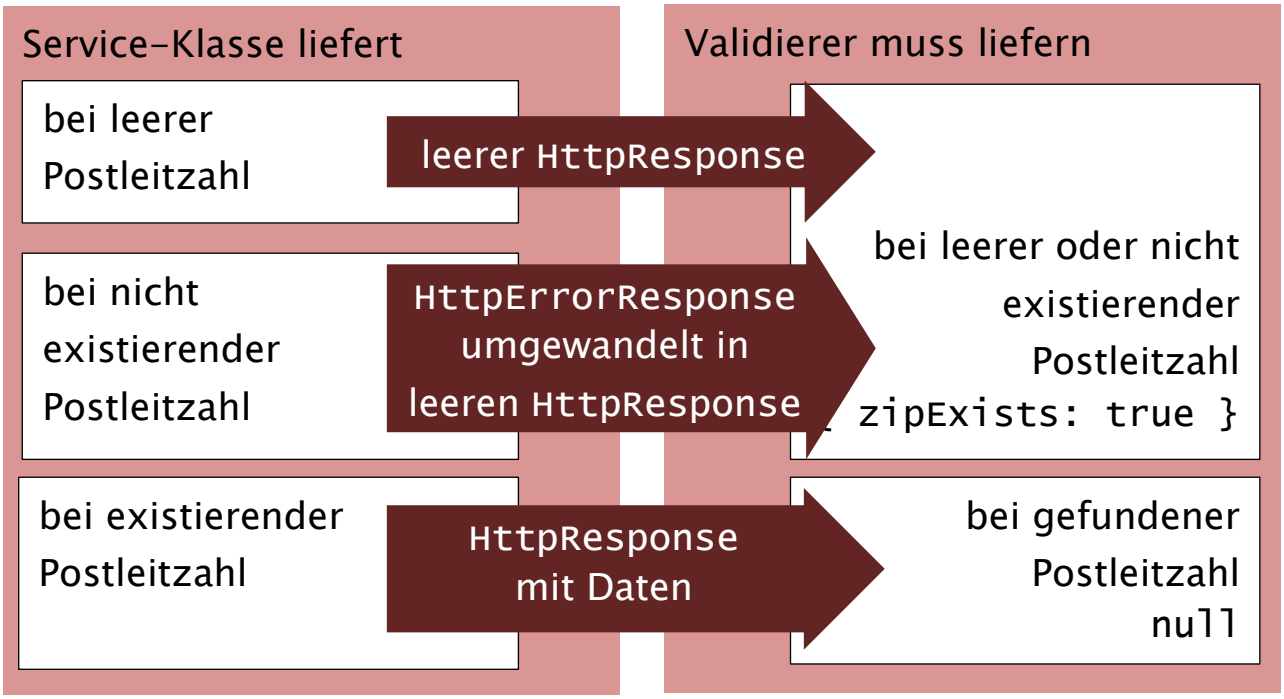
Um Korrektheit von Eingaben zu prüfen, wird externer Dienst abgefragt (z.B. ist Benutzername nicht vergeben, existiert eingegebene Postleitzahl)



- Der Postleitzahlservice (`ZipService`) ermittelt ob Postleitzahl existiert
- Laufende Serviceabfrage wird visualisiert dargestellt ( )

Der Service

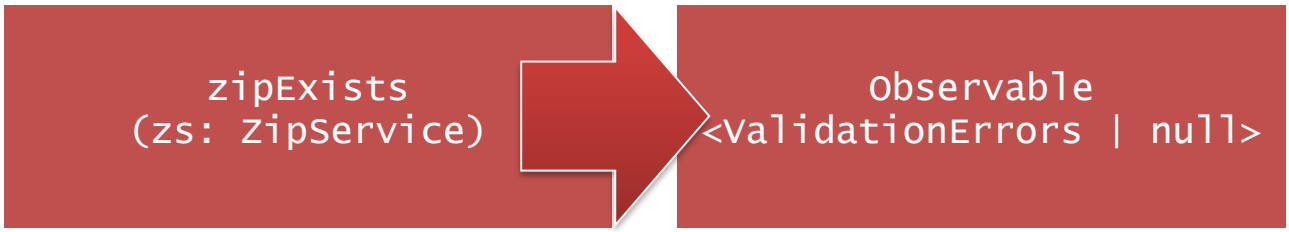
```
…
import { catchError } from 'rxjs/operators';
const URL = 'http://api.zippopotam.us/it';
@Injectable()
export class ZipService {
  constructor(private http: HttpClient) {}
  checkZipExists(value: string): Observable<any> {
    if (!value || !value.length) {
      return new Observable(observer => {
        observer.next(null); observer.complete();
      });
    } else {
      return this.http.get(`${URL}/${value}`)
        .pipe(catchError(this.handleError));
    }
  }
  private handleError(error: HttpErrorResponse): Observable<any> {
    return new Observable(observer => {
      observer.next(null); observer.complete();
    });
  }
}
```



Service liefert bei falscher Postleitzahl einen `HttpErrorResponse`, was dazu führt, dass ein Exception geworfen wird. Validierer kann aber auf Exception nicht reagieren, erwartet aber in diesem Fall einen leeren Response. In `handleError` wird dem entsprochen.
Wird keine Postleitzahl übergeben, wird ebenfalls leerer Response geliefert.

Der asynchrone Validierer

```
…
import { map } from 'rxjs/operators';

export class ZipValidators {
  static zipExists(zs: ZipService[1]) {
    return function(control: AbstractControl):
      Observable<ValidationErrors | null>[2] {
      return zs.checkZipExists(control.value)
        .pipe(map(data => data ? null : { zipExists: true }));
    };
  }
}
```



- [1)]Der Validiermethode wird Service übergeben
- [2)]Damit *Funktionssignatur* für Validierer übereinstimmt, liefert Methode `zipExists()` eine Funktion zurück die der Funktionssignatur des Validierers entspricht. Diese zweite Funktion wird tatsächlich an `FormControl` gebunden
- Methode liefert `null` falls Postleitzahl korrekt ist oder `{zipExists: true }` falls Postleitzahl nicht existiert oder keine Postleitzahl übergeben wurde

Die Komponente

```
…
export class ZipFormComponent implements OnInit {
  zipForm!: FormGroup;
  constructor(
    private fb: FormBuilder, private zs: ZipService
  ) { }
  ngOnInit() {
    this.zipForm = this.fb.group({
      zip: ['',
        { asyncValidators: ZipValidators.zipExists(this.zs)
      }]
    });
  }
}
```

Das Template

```
<form [formGroup]="zipForm" …>
  <mat-form-field>
    <input matInput formControlName="zip"
      placeholder="Postleitzahl" type="text">
    <mat-hint>Gültige italienische Postleitzahl</mat-hint>
    <mat-icon matSuffix>
      <mat-spinner
        *ngIf="zipForm.controls.zip.pending; else not_pending"
        diameter="20">
      </mat-spinner>
      <ng-template #not_pending>
        search
      </ng-template>
    </mat-icon>
    <mat-error>Die Postleitzahl existiert nicht</mat-error>
  </mat-form-field>
  Validator: {{ zipForm.controls.zip.valid }}
  Pending: {{ zipForm.controls.zip.pending }}
</form>
```