



# Angular Code Audit

July 2023, Darko V.

**First Digital Trust Limited**

40/F, Tower 1, Lippo Centre, 89 Queensway,  
Admiralty, Hong Kong

[sales@1stDigital.com](mailto:sales@1stDigital.com) | [www.1stDigital.com](http://www.1stDigital.com)

# Table of Contents

- Use Strictly Typed TypeScript
  - Use Strictly Typed Reactive Forms
  - Avoid Dynamic Object Access in Modals
  - Use More TypeScript Built-in Helper Types
- The Angular Way
  - Standardize Event Listening in Modals
- Reduce Code Repetition
  - Standardize Validation Errors
  - Standardize Confirm Modal Component



Use Strictly Typed TypeScript



# Use Strictly Typed TypeScript

## Not using

- One common bad practice in TypeScript development is avoiding the use of strict typing
- When TypeScript code lacks strict typing, its a subject to run-time errors, such as unexpected type mismatches, undefined values, or incorrect function invocations

## Using

- Benefit from catching many errors before the code is even executed.
- Strict typing also improves code readability and maintainability.
- By explicitly defining types, it becomes easier for developers to understand the expected structure of objects, parameters, and function returns

# Use Strictly Typed Forms

- **Type safety:** TypeScript's static typing ensures that only valid data types can be assigned to form controls, preventing type-related errors. This significantly reduces the risk of bugs and runtime exceptions caused by incompatible data assignments.
- **Compile-time validation:** During compilation, Angular's template and component bindings are checked against the defined form model types. This validation alerts developers to any mismatches or missing properties, catching potential errors before the application runs. It helps to maintain consistency and correctness across the form structure.
- **Auto-completion, IDE support:** The use of strict typing enables IDEs and code editors to provide intelligent auto-completion suggestions, type checking, and error highlighting. This enhanced tooling support facilitates faster development, reduces cognitive load, and minimizes the chances of typographical errors.
- **Improved code readability and maintainability:** With explicit type definitions, the codebase becomes more self-documenting, making it easier for developers to understand the expected form structure. This clarity improves collaboration and enables efficient maintenance and refactoring of the code over time.

# Use Strictly Typed Forms

```
createForm() {
  this.form = this.formBuilder.group({
    serviceEntityId: this.formBuilder.control(
      { value: null, disabled: false },
      Validators.required
    ),
    assetMasterId: this.formBuilder.control(
      this.assetMasterId,
      Validators.required
    ),
    incomeLedgerAccountId: this.formBuilder.control(
      { value: null, disabled: true },
      Validators.required
    ),
    incomeDescription: this.formBuilder.control(''),
    alias: this.formBuilder.control('')
  });

  this.form.controls['idontexist'].valueChanges
    .subscribe(() => { /*do something*/ });
  this.form.controls['assetMasterId'].valueChanges
    .subscribe(() => { /*do something*/ });
}
```

```
54
55
56 createForm2() {
57   this.formImproved = new FormGroup({
58     assetMasterId: new FormControl<number | null>(0, Validators.required),
59     serviceEntityId: new FormControl(0, Validators.required),
60     incomeLedgerAccountId: new FormControl<number>(0, Validators.required),
61     incomeDescription: new FormControl<string>('', Validators.required),
62     alias: new FormControl<string>('', Validators.required),
63   });
64
65   this.formImprovedHelperClass = this.formBuilder.nonNullable.group({
66     assetMasterId: this.formBuilder.nonNullable.control(0, Validators.required),
67     serviceEntityId: this.formBuilder.nonNullable.control(0, Validators.required),
68     incomeLedgerAccountId: this.formBuilder.nonNullable.control(0, Validators.required),
69     incomeDescription: this.formBuilder.nonNullable.control('', Validators.required),
70     alias: this.formBuilder.nonNullable.control('', Validators.required),
71   });
72
73   this.formImproved.controls['idontexist'].valueChanges
74     .subscribe(() => { /*do something*/ });
75   this.formImproved.controls['assetMasterId'].valueChanges
76     .subscribe(() => { /*do something*/ });
77   this.formImprovedHelperClass.controls['idontexist'].valueChanges
78     .subscribe(() => { /*do something*/ });
79 }
```

# Use More TypeScript Built-in Helper Types (Utility Types)

## Not using


- Introducing interfaces/classes that are not really needed

## Using

- Code is more understandable
- Sometimes no need of adding new interfaces and classes
- Code is more Strictly typed without a lot of effort
- More compile errors 🎉🎉🎉

Record, Omit, Pick, Readonly...

# Dynamic Object Access in Modals

- ``modalInstance.componentInstance['someProperty'] = ...`` or ``modal.componentInstance.setDetails({...})``
- I provided a example how to handle this on better way 



# Examples



Reduce Code Repetition

# Reduce Code Repetition

- Bad practice
- More than 2 times, should be considered for refactoring
- Components, Helper classes / util functions

# Standardize validation errors

- Current approach provide a lot of logic duplication
- Since no Strictly typed forms, also no type checks and no compile errors
- More readable code
- Easier to wrap-up and make more generic forms.
- No validation error messages repeating
- No validation error show/hide logic repeating

Example on next slide

# Handling validation errors quick example

```

<div class="form-floating">
  <app-amount-input
    [formGroup]="formFiat"
    [decimalPlaces]="currency?.decimalPlaces"
    controlName="amount"
    placeholder="Enter Withdrawal Amount"
    label="Withdrawal Amount"
    [withError]="
      (formFiat.get('amount')?.invalid &&
       formFiat.get('amount')?.dirty) ||
      false
    "
    data-cy="withdrawal-modal-fiat-amount-input"
  >
    <ng-container *ngTemplateOutlet="balance"></ng-container>
    <div
      class="invalid-feedback"
      *ngIf="formFiat.get('amount')?.errors?.['required'] &
    >
      Amount is required
    </div>
    <div
      *ngIf="!formFiat.get('amount')?.errors?.['required'] &
      class="invalid-feedback"
    >
      Invalid Amount
    </div>
    <div
      *ngIf="formFiat.get('amount')?.dirty && formFiat.get(
    >
      <ng-container *ngTemplateOutlet="balanceError"></ng-c
    </div>
  </app-amount-input>
</div>
<div class="mb-13">

```

```

140 </div>
141 <div *ngIf="step === 2">
142   <div class="form-floating">
143     <app-amount-input
144       [formGroup]="formFiat"
145       [decimalPlaces]="currency?.decimalPlaces"
146       controlName="amount"
147       placeholder="Enter Withdrawal Amount"
148       label="Withdrawal Amount"
149       [withError]="
150         (formFiat.get('amount')?.invalid &&
151          formFiat.get('amount')?.dirty) ||
152         false
153       "
154       data-cy="withdrawal-modal-fiat-amount-input"
155     >
156       <ng-container *ngTemplateOutlet="balance"></ng-c
157       <error-component [control]="formFiat.controls['am
158     </error-component>
159   </app-amount-input>
160 </div>
161
162 <div class="form-floating">
163   <app-amount-input
164     [formGroup]="formFiat"
165     [decimalPlaces]="currency?.decimalPlaces"
166     controlName="amount"
167     placeholder="Enter Withdrawal Amount"
168     label="Withdrawal Amount"
169     [withError]="
170       (formFiat.get('amount')?.invalid &&
171        formFiat.get('amount')?.dirty) ||
172       false
173     "
174     data-cy="withdrawal-modal-fiat-amount-input"
175   >

```

# Standardize Confirm Modal Component

- Current approach provide duplication
- Common used in application logic
- Every language has its own alert, confirm, prompt available

# Standardize Event Listening in Modals

- When new developer come, he is not sure if he should make a EventEmitter on dialog or return result as dialog result
- Code is not consistent
- New joiners can easily see and follow the practice in the application
- Code is more readable
- In almost every scenario, custom dialog returns some result

# Examples





Questions?





Thank You!

