The Angular 21

Signal Forms Handbook

Written by

Mateusz Stefańczyk

HOUSE OF ANGULAR

Is Reactive Forms finally becoming "legacy"? Why are we all waiting for Signal Forms?

Remember the first time you had to handle a complex form in Angular?

The dance with valueChanges, the manual unsubscriptions, the struggle with typing in older versions, or the "magic" behaviors of dynamic controls. ReactiveFormsModule has served us faithfully for years, but let's be honest: **we always felt it could be simpler.**

With the arrival of **Angular 21**, we are entering a new era. An **era without Zone.js**, **with better performance**, and most importantly, **with code that reads like a good book**, not a washing machine manual.

In this guide, I will show you h**ow Signal Forms change the game**. It isn't just another "new feature." It is a paradigm shift that will make you enjoy writing forms again.

Signal forms, compared to their predecessors, are a breath of fresh air. Familiar concepts such as validators, dirty state, and validity remain, but their implementation has been **completely rewritten using Signals**. However, not everything maps 1:1; we **no longer have concepts like FormGroup or FormControl** as we knew them. On the flip side, "Typed Forms" are now actually typed.

Ready to see the future? Let's dive in.

Introduction to Signal Forms

Signal forms themselves are already a novelty, working based on signals that we already know quite well. Nevertheless, they introduce **new nomenclature and features** to the world of forms that we have no chance of knowing based on previous years with Reactive and Template-Driven Forms. **One of them is a form model**.

Form Model

The form model is a **writable signal that we use to initialize our form**. This is crucial because the **form model directly corresponds to the type of our form**. Forms are now very well typed and directly infer the type from the initializing object.

Another very important point is that any modifications and updates to our form model will be **directly propagated and reflected by the form**. Currently, this initialization signal is the owner of the state the form represents – they remain fully synchronized.

```typescript
export class LoginComponent {
 // Form model
 loginModel = signal({
   email: '',
   password: ''
 })

 // We init form with defined form model
 loginForm = form(this.loginModel)
}
```

This is a fundamental change compared to reactive forms. In the previous approach, the form managed its own state independently – we mapped object fields to form controls, but the form state existed independently from the source object. Any synchronization of the form with an external model required manual value updates.

```typescript
// We map entity properties into form controls, since that
point they are not synchronized

form = fb.group({
 email: [entity.email],
 password: [entity.password]
});
```

Initializing the Form

Creating a new form is done through **the form() function.** This is already characterizing Angular: more and more functionality is implemented in a functional way. The first argument of the function is our previously mentioned form model.

It will give type to our form, and based on it, the Form Tree will be initialized - a hierarchical structure of fields where each object in the model becomes a node with its

own children, and each primitive value becomes a terminal field (leaf). Thanks to this, navigation through the form naturally corresponds to navigation through data.

```typescript
import { form } from '@angular/forms/signals';

loginForm = form(this.loginModel);

// Navigation through dot - like a regular object
loginForm.email      // email field
loginForm.password   // password field
```

Typing - End of Compromises

Typed Reactive Forms introduced in Angular 14 were a step in the right direction. However, in practice, their **typing has limitations** that can be frustrating on a daily basis. Signal forms, designed from the ground up with TypeScript in mind, solve these problems.

Problem 1: Nullable Everywhere

In Reactive Forms, each FormControl by default has type T | null:

```typescript
const emailControl = new FormControl('');
// Type: FormControl<string | null>

emailControl.value; // string | null - always nullable!
```

We can use nonNullable, but it requires explicit declaration for each control:

```typescript
const emailControl = new FormControl('', { nonNullable: true });
// Only now the type is FormControl<string>
```

Signal forms - type comes directly from the model:

```typescript
const model = signal({ email: '' });
const myForm = form(model);

myForm.email().value(); // string - no null!
```

Problem 2: get() Method Loses Types

This is one of the most irritating aspects of typed Reactive Forms:

```typescript
const form = new FormGroup({
  user: new FormGroup({
    email: new FormControl(''),
    name: new FormControl('')
  })
});

// Even though form is typed...
const email = form.get('user.email');
// ...email is of type: AbstractControl<unknown, unknown> |
null


// We have to cast manually
```

```
const emailTyped = form.get('user.email') as
FormControl<string | null>;
```

The get() method takes a string, but TypeScript cannot verify that the path is correct.

Signal forms - full navigation typing:

```
TypeScript
const model = signal({
  user: { email: '', name: '' }
});
const myForm = form(model);

// Full typing at every level
myForm.user.email().value(); // string

// Typo? Compilation error!
myForm.user.emial; // ❌ Property 'emial' does not exist
```

Problem 3: FormArray Loses Structure

FormArray in typed forms can be problematic:

```
TypeScript
const users = new FormArray([
  new FormGroup({
    name: new FormControl(''),
    email: new FormControl('')
  })
]);
```

```
// When accessing through at()...
users.at(0); // AbstractControl - we lose FormGroup structure information!
users.at(0).get('name'); // again AbstractControl | null
```

Signal forms preserve full structure:

```typescript
interface User {
  name: string;
  email: string;
}

const model = signal<{ users: User[] }>({
  users: [{ name: 'Jan', email: 'jan@example.com' }]
});

const myForm = form(model);

// Full typing preserved!
myForm.users[0].name().value();  // string
myForm.users[0].email().value(); // string

// Iteration is also typed
for (const [index, userField] of myForm.users) {
  userField.name().value();  // TypeScript knows it's a string
}
```

Problem 4: Dynamic Forms

Adding controls at runtime is a typing nightmare:

```typescript
const form = new FormGroup({
  name: new FormControl('')
});

form.addControl('email', new FormControl(''));

// TypeScript still thinks form only has 'name'
form.controls.email; // ❌ Property 'email' does not exist
```

Signal forms - model is the source of truth:

```typescript
const model = signal<{ name: string; email?: string }>({
  name: ''
});
const myForm = form(model);

// Adding a field = updating the model
model.update(m => ({ ...m, email: 'new@example.com' }));

// Typing automatically accounts for optional field
if (myForm.email) {
  myForm.email().value(); // string
}
```

How Does It Work Under the Hood?

The heart of the type system is **FieldTree<TModel>** - a type that recursively maps model structure to form structure:

- For objects - each property becomes a form field
- For arrays - elements accessible by index with preserved type
- For primitives - terminal field without children

Thanks to this, TypeScript always knows what type each field has - **no manual assertions, no casting, no guessing**.

Summary

Aspect	**Typed Reactive Forms**	**Signal Forms**
Nullable by default	Yes (T | null)	No - depends on model
Navigation (get() / dot)	Loses types	Full typing
Arrays	at() returns AbstractControl	Preserves structure
Dynamic fields	Require type assertion	Model as source of truth
Refactoring	Partially safe	Fully safe

Typed Reactive Forms were a compromise - typing was added to the existing API. Signal forms were designed from scratch **with TypeScript as a priority**. The difference is noticeable from the first line of code.

Validation

Similar to Reactive Forms, we have access to predefined validators. However, the way of applying them is completely different - i**nstead of passing validators when creating a control, we call functions pointing to the field and validator**.

```typescript
import { form, required, minLength, email, pattern } from
'@angular/forms/signals';

const loginForm = form(this.loginModel, (login) => {
  required(login.email);
  email(login.email);
  required(login.password);
  minLength(login.password, 8);
});
```

All validators are passed in one place - as the second parameter of the form() function. This centralizes validation logic, which has its pros and cons. On one hand, we have a **full picture of the rules** in one place. On the other hand, the validator **is not applied directly next to the field definition**, as it used to be in Reactive Forms:

```typescript
// Reactive Forms - validator at field
new FormControl('', [Validators.required, Validators.email])

// Signal Forms - validators in separate section
form(model, (f) => {
  required(f.email);
  email(f.email);
});
```

Everyone will have to subjectively assess how this affects form readability. For small forms, the difference is cosmetic. For large ones, centralization can be an advantage.

Predefined Validators

Signal forms provide a set of built-in validators:

```typescript
required(path);                 // required field
min(path, minValue);            // minimum numeric value
max(path, maxValue);            // maximum numeric value
minLength(path, length);        // minimum length
maxLength(path, length);        // maximum length
pattern(path, regex);           // regex pattern
email(path);                    // email format
```

Custom Validators

Creating custom validators is simpler than ever:

```typescript
import { form, validate, customError } from '@angular/forms/signals';

const registrationForm = form(this.model, (f) => {
  // Custom validator - function receives context with value
  validate(f.username, ({ value }) => {
    const username = value();
    if (username.includes(' ')) {
      return customError({ kind: 'no-spaces', message: 'Name cannot contain spaces' });
    }
    return undefined; // no error
  });

  // Validator with access to other fields
  validate(f.confirmPassword, ({ value, valueOf }) => {
    if (value() !== valueOf(f.password)) {
      return customError({ kind: 'password-mismatch', message: 'Passwords are not identical' });
    }
```

```
    return undefined;
  });
});
```

Validator context (ctx) gives access to:

- value() - current field value
- valueOf(path) - value of any other field
- state - full field state (touched, dirty, etc.)
- stateOf(path) - state of any other field

Validator Reactivity - Automatic Dependency Tracking

Here lies one of the biggest advantages of signal forms. Validators work inside a reactive context, which means Angular automatically tracks all read signals.

Let's look at a password comparison validator:

```typescript
validate(f.confirmPassword, ({ value, valueOf }) => {
  if (value() !== valueOf(f.password)) {
    return customError({ kind: 'password-mismatch' });
  }
  return undefined;
});
```

This validator will run when:

- confirmPassword changes (because we call value())
- password changes (because we call valueOf(f.password))

So the validator reacts to changes of every read signal, not just the field it's assigned to.

Why Is This Revolutionary?

Think about the classic "passwords must match" scenario:

1. User enters password in password → confirmPassword validator runs → error (confirm is empty)
2. User enters the same in confirmPassword → validator runs → OK
3. User goes back and changes password → confirmPassword validator automatically runs → error (they no longer match)

In Reactive Forms, point 3 required manual work:

```typescript
// Reactive Forms - need to manually link
this.form.get('password').valueChanges.subscribe(() => {
  this.form.get('confirmPassword').updateValueAndValidity();
});
```

In signal forms, this happens automatically. Zero subscriptions, zero manual updateValueAndValidity() calls.

Performance Consideration

Since the validator reacts to all read signals, it's worth reading only what's really needed:

```typescript
// ⚠️ Reads entire form - will run on EVERY change
validate(f.someField, ({ stateOf }) => {
  const everything = stateOf(f).value(); // entire form!
  // ...
});



// ✅ Precise dependencies - will run only when one of two fields changes
validate(f.someField, ({ value, valueOf }) => {
  const mine = value();
  const related = valueOf(f.otherField);
```

```
  // ...
});
```

Asynchronous Validation

For validation requiring server requests, we have validateAsync and validateHttp:

```
TypeScript
import { validateHttp } from '@angular/forms/signals';


const form = form(this.model, (f) => {
  validateHttp(f.username, {
    request: ({ value }) =>
      value() ? `/api/check-username?name=${value()}` : undefined,
    onSuccess: (result) =>
      result.taken ? customError({ kind: 'taken', message: 'Name taken' }) :
undefined,
    onError: () =>
      customError({ kind: 'server-error', message: 'Error checking availability' })

  });

});
```

Asynchronous validation runs only when synchronous validation passes successfully.

Conditional Functions

Analogously to validators, we have functions allowing dynamic control of field state:

```typescript
import { form, disabled, hidden, readonly } from
'@angular/forms/signals';


const orderForm = form(this.model, (order) => {
  // Field disabled conditionally
  disabled(order.discountCode, ({ valueOf }) =>
    valueOf(order.orderType) === 'wholesale'
  );


  // Field hidden conditionally

  hidden(order.companyName, ({ valueOf }) =>
    valueOf(order.customerType) !== 'business'
  );



  // Read-only field

  readonly(order.totalPrice);

});
```

Key difference from Reactive Forms: these functions are reactive. Changing orderType automatically enables or disables the discountCode field, without manually subscribing or calling enable()/disable().

What Does disabled/hidden/readonly Mean?

Fields in these states are skipped when determining the parent state:

- A hidden field with an error doesn't make the form invalid
- A disabled field marked as dirty doesn't affect the parent's dirty
- Read-only field doesn't participate in validation

Reusability - Schema

Schemas are a complete novelty. A schema allows defining a set of rules once and applying them in multiple places:

```typescript
import { schema, required, email, minLength } from '@angular/forms/signals';


// Define once

const addressSchema = schema<Address>((addr) => {
  required(addr.street);
  required(addr.city);
  required(addr.zipCode);
  pattern(addr.zipCode, /^\d{2}-\d{3}$/);
});


const contactSchema = schema<Contact>((contact) => {
    required(contact.email);
  email(contact.email);
  minLength(contact.phone, 9);

});
```

Applying Schemas

```typescript
import { form, apply, applyEach } from '@angular/forms/signals';

// Apply to nested object
const customerForm = form(this.customerModel, (customer) => {
  required(customer.name);
  apply(customer.billingAddress, addressSchema);
  apply(customer.shippingAddress, addressSchema);
  apply(customer.contact, contactSchema);
});

// Apply to each array element
const orderForm = form(this.orderModel, (order) => {
  applyEach(order.addresses, addressSchema);
});
```

Conditional Schemas

We can apply schemas conditionally:

```typescript
import { applyWhen, applyWhenValue } from '@angular/forms/signals';

const form = form(this.model, (f) => {
  // Schema applied when condition is met
  applyWhen(f.payment,
    ({ valueOf }) => valueOf(f.paymentMethod) === 'card',
    cardPaymentSchema
  );

  // Schema applied based on field value (with type narrowing!)
  applyWhenValue(f.document,
    (doc): doc is Invoice => doc.type === 'invoice',
    invoiceSchema
```

```
  );
});
```

Schemas are a powerful tool for organizing application architecture. You define rules for Address once - you're sure that every form with an address validates it identically.

Field Directive - One Way for Everything

In Reactive Forms we had to remember about different directives:

```HTML
<!-- Reactive Forms - different directives -->
<input [formControl]="emailControl">
<input formControlName="email">
<div formGroupName="address">...</div>
<div formArrayName="items">...</div>
```

Signal forms simplify this to one [field] directive:

```HTML
<!-- Signal Forms - always [field] -->
<input [field]="myForm.email">
<input [field]="myForm.address.street">
<input [field]="myForm.items[0].name">
```

Typing in Template

The Field directive is strictly typed. When you try to bind a number type field to an input expecting string:

```HTML
<!-- myForm.age is FieldTree<number> -->
<input type="text" [field]="myForm.age">
<!-- ❌ Type 'FieldTree<number>' is not assignable to type
'FieldTree<string>' -->
```

This is something unseen before in Angular forms - type errors detected in the template!

Automatic State Binding

The Field directive automatically synchronizes state between the field and the UI control:

```TypeScript
// Control can declare these inputs - Field will automatically fill them
@Component({...})
export class MyInput {
  value = model<string>('');          // value - required
  disabled = input<boolean>(false);    // is disabled
  touched = input<boolean>(false);     // is touched
  errors = input<ValidationError[]>([]); // validation errors
  required = input<boolean>(false);    // is required
  // ... and more
}

<my-input [field]="myForm.email"></my-input>
<!-- All states synchronized automatically -->
```

FormValueControl Contract

To create a custom control compatible with [field], you just need to implement a simple contract:

```typescript
import { FormValueControl } from '@angular/forms/signals';

@Component({
  selector: 'my-custom-input',
  template: `...`
})
export class MyCustomInput implements FormValueControl<string> {
  // Only required field
  readonly value = model<string>('');

  // Optional - Field will automatically bind if they exist
  readonly disabled = input<boolean>(false);
  readonly errors = input<ValidationError[]>([]);
  readonly touched = input<boolean>(false);
}
```

Migration - compatForm

If you have an existing application with Reactive Forms, you probably won't rewrite everything at once (and rightly so). Fortunately, Angular anticipated this scenario and provides compatForm(), a function that allows mixing both worlds.

```typescript
import { compatForm } from '@angular/forms/signals';
import { FormControl, Validators } from '@angular/forms';

// Existing FormControl with validators
const ageControl = new FormControl(5, Validators.min(3));

// Model mixing signal forms with Reactive Forms
```

```
const model = signal({
  name: 'Jan',          // regular signal forms field
   age: ageControl       // existing FormControl
});


const myForm = compatForm(model);
```

How Does It Work?

compatForm automatically "unwraps" values from FormControl:

```
TypeScript
myForm.age().value();    // 5 (number, not FormControl!)
myForm.name().value();   // 'Jan'

// If you need access to the original FormControl:
myForm.age().control();  // FormControl<number>
```

Bidirectional Synchronization

State is synchronized both ways:

```
TypeScript
// Change through FormControl
ageControl.setValue(10);
myForm.age().value();       // 10

// Change through signal forms
myForm.age().value.set(15);
ageControl.value;           // 15
```

```
// Touched/dirty also propagates
ageControl.markAsTouched();
myForm.age().touched();     // true
myForm().touched();         // true (propagation to parent)
```

Validators Are Respected

Validators defined on FormControl work normally:

```
TypeScript
const control = new FormControl(1, Validators.min(5));
const model = signal({ age: control });
const myForm = compatForm(model);

myForm.age().valid();   // false
myForm().valid();       // false (propagation)

control.setValue(10);
myForm.age().valid();   // true
```

Limitation: No Rules on FormControl Fields

You cannot apply signal forms rules (like required(), validate()) directly to fields that are FormControl - TypeScript will block it:

```
TypeScript
compatForm(model, (f) => {
  required(f.name);     // ✅ OK - regular field
  required(f.age);      // ❌ Compilation error - age is FormControl

  // But you can read FormControl values in validators of other fields:
  validate(f.name, ({ valueOf }) => {
```

```
    return valueOf(f.age) < 18
      ? customError({ kind: 'too-young' })
      : undefined;
  });
});
```

This makes sense - FormControl validation should stay with that FormControl. Mixing two validation systems on one field is asking for trouble.

Submit and Reset

Submitting the Form

Signal forms provide a submit() function that handles typical submission flow:

TypeScript

```
import { submit } from '@angular/forms/signals';

async function onSubmit() {
  await submit(myForm, async (form) => {
    // 1. At this point all fields are already marked as touched
    // 2. If form is invalid - this function will NOT be called
    // 3. form().submitting() === true during execution

    const response = await api.save(form().value());

    // We can return server errors
    if (response.error) {
      return [{
        field: myForm.email,
        error: customError({ kind: 'server', message: response.error })
      }];
    }
```

```
    return undefined; // success
  });
}
```

What submit() does under the hood:

1. Marks all fields as touched (to show errors)
2. Checks valid() - if false, aborts and doesn't call action
3. Sets submitting to true
4. Calls the action
5. Applies any server errors to the appropriate fields
6. Sets submitting to false

Submitting State

You can use submitting() to block UI:

HTML
```
<button [disabled]="myForm().submitting()">
  {{ myForm().submitting() ? 'Sending...' : 'Submit' }}
</button>
```

submitting propagates down - if the parent is submitting, the children are too:

TypeScript
```
myForm().submitting();          // true
myForm.email().submitting();    // true
```

Resetting the Form

The reset() method clears interaction state (touched, dirty):

```typescript
myForm.email().reset();  // resets single field
myForm().reset();        // resets entire form and all children
```

Optionally you can pass a new value:

```typescript
myForm().reset({ email: '', password: '' });
```

Note: reset() doesn't change value if you don't pass it - it only resets UI state.

Debouncing

For fields where we don't want to react to every keystroke (e.g. search, async validation), we have debounce():

```typescript
import { form, debounce } from '@angular/forms/signals';

const searchForm = form(this.model, (f) => {
  // Update model only 300ms after last change
  debounce(f.query, 300);
});
```

You can also pass your own debounce function:

```typescript
debounce(f.query, (ctx, abortSignal) => {
  return new Promise(resolve => {
    const timeout = setTimeout(resolve, 500);
    abortSignal.addEventListener('abort', () => clearTimeout(timeout));
  });
});
```

Debouncing is inherited - if you set it on parent, children will also be debounced (unless they override with their own).

Custom Controls - End of ControlValueAccessor

In Reactive Forms creating a custom form control required implementing ControlValueAccessor - an interface with four methods, magical provider with forwardRef, and manual calling of onChange/onTouched. Every Angular developer knows this boilerplate:

```typescript
// Reactive Forms - ControlValueAccessor 😵
@Component({
  selector: 'my-input',
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => MyInputComponent),
      multi: true
        }
  ]
})
export class MyInputComponent implements ControlValueAccessor {
  private onChange: (value: string) => void = () => {};
  private onTouched: () => void = () => {};
```

```
  writeValue(value: string): void { /* ... */ }
  registerOnChange(fn: (value: string) => void): void { this.onChange = fn; }
  registerOnTouched(fn: () => void): void { this.onTouched = fn; }
  setDisabledState(isDisabled: boolean): void { /* ... */ }
}
```

Signal Forms reduce this to one line.

FormValueControl - Minimalistic Contract

To create a control compatible with [field] directive, you just need to implement FormValueControl<T> interface:

TypeScript

```
import { Component, model } from '@angular/core';
import { FormValueControl } from '@angular/forms/signals';

@Component({
  selector: 'my-input',
  template: `
    <input
      [value]="value()"
      (input)="value.set($event.target.value)"
    />
  `
})
export class MyInputComponent implements FormValueControl<string> {
  readonly value = model('');
}
```

That's all. One model() signal and the control is ready to use:

```HTML
<my-input [field]="myForm.email"></my-input>
```

The [field] directive automatically synchronizes the value between form and control. Change in form → value() update. Change in control → form model update.

Optional Inputs - Automatic State Binding

FormValueControl provides several optional inputs. If you declare them, [field] directive will automatically fill them:

```TypeScript
@Component({
 selector: 'my-input',
 template: `
  <div class="input-wrapper" [class.has-error]="invalid()">
   <input
    [value]="value()"
    [disabled]="disabled()"
    [attr.name]="name()"
    (input)="value.set($event.target.value)"
    (blur)="touched.set(true)"
   />
        @if (invalid() && touched()) {
    <div class="errors">
     @for (error of errors(); track error.kind) {
      <span>{{ error.message }}</span>
     }
    </div>
   }
  </div>
 `
})
export class MyInputComponent implements FormValueControl<string> {
 // Required
```

```
  readonly value = model('');

  // Optional - Field will automatically bind if they exist
  readonly disabled = input(false);
  readonly touched = model(false);  // model() allows two-way binding
  readonly errors = input<ValidationError[]>([]);
  readonly invalid = input(false);
  readonly name = input('');
  readonly required = input(false);
  readonly readonly = input(false);
}
```

Full list of optional inputs:

- disabled - whether the field is disabled
- readonly - whether the field is read-only
- touched - whether the user interacted with the field (can be model() for two-way binding)
- dirty - whether the value was changed
- invalid - whether validation failed
- pending - whether async validation is in progress
- errors - list of validation errors
- name - field name in form
- required - whether the field is required
- min, max, minLength, maxLength, pattern - values from validators

You declare only those you need. The rest is ignored.

FormCheckboxControl - For Checkboxes

For checkbox-type controls, there is a separate FormCheckboxControl contract:

```typescript
import { Component, model } from '@angular/core';
import { FormCheckboxControl } from '@angular/forms/signals';

@Component({
  selector: 'my-checkbox',
  template: `
    <label>
      <input
        type="checkbox"
        [checked]="checked()"
        (change)="checked.set($event.target.checked)"
      />
      <ng-content></ng-content>
    </label>
  `
})
export class MyCheckboxComponent implements FormCheckboxControl {
  readonly checked = model(false);
}
```

Usage:

```html
<my-checkbox [field]="myForm.agreeToTerms">
  I accept the terms
</my-checkbox>
```

Controls as Directives

Control doesn't have to be a component - it can be a directive on a native element:

```typescript
@Directive({
  selector: 'input[myCustomInput]',
  host: {
    '[value]': 'value()',
    '(input)': 'value.set($event.target.value)',
    '(blur)': 'onBlur()'
  }
})
export class MyCustomInputDirective implements FormValueControl<string> {
  readonly value = model('');
  readonly touched = model(false);

  onBlur() {
      this.touched.set(true);
  }
}

<input myCustomInput [field]="myForm.email" />
```

The [field] directive will automatically detect FormValueControl and connect it to the form.

Signal Forms eliminate ceremony. Instead of implementing an interface with four methods and configuring providers, you declare one signal, and the control works.

Before You Start - A Few Notes

Status: Experimental

Signal forms are marked as @experimental 21.0.0. What does this mean in practice?

- API may change in future versions (though the core will likely remain stable)
- Edge cases and bugs may appear
- Documentation is still in development

Does this mean they're not worth using?

In my opinion, they are **worth it**, especially in new projects. But in critical production applications, consider whether you're ready for potential API migrations.

Import

Signal forms live in a separate entry point:

```typescript
import { form, required, validate, ... } from '@angular/forms/signals';
```

Don't mix with imports from @angular/forms (unless using compatForm).

Summary

Signal forms are not an evolution of Reactive Forms - they are a rethought from scratch implementation of forms in Angular. Key changes:

- **Model as source of truth** - form and data are always synchronized
- **Real typing** - TypeScript knows everything, without compromises
- **Reactivity out of the box** - validators react to dependency changes without manual binding
- **One API** - [field] directive instead of a zoo of directives
- **Schemas** - reusable validation rules
- **Simple Controls** - FormValueControl instead of ControlValueAccessor

Should you migrate existing applications? If you have time and budget - yes. If not, compatForm allows introducing signal forms gradually, form by form.

And new projects? There's no dilemma here. Signal forms are the future of forms in Angular.

You've got the theory. Now, let's talk about production.

Everything you just read sounds great on screen. But **implementing Signal Forms in a complex, living business application is a completely different beast**.

As you start refactoring, questions will pop up that tutorials don't cover:

- *How do I handle complex async validations with debouncing without killing the backend?*
- *How do I write reusable Control Components that handle errors, disabled states, and dirty checking consistently?*
- *How do I safely migrate a "legacy" form with 50+ fields without stopping feature development?*

You could spend the next **40 hours digging through GitHub** issues and documentation, learning from your own mistakes...

Or you can join our Workshop:
[Scalable Architecture & Modern Reactivity + Signal Forms]

Because **Signal Forms rely heavily on the rest of your application's structure** (such as the new Resource API or Signal Inputs), we realized that teaching forms in isolation isn't enough.

That is why we teamed up with **Google Developer Experts Mateusz Stefańczyk and Fanis Prodromou** to create a complete **2-Day Workshop**.

We structured it logically so you can modernize your entire stack:

Day 1: Scalable Architecture (with GDE Fanis Prodromou)

Before you build the forms, you need a solid foundation. Fanis covers the "Big Picture":

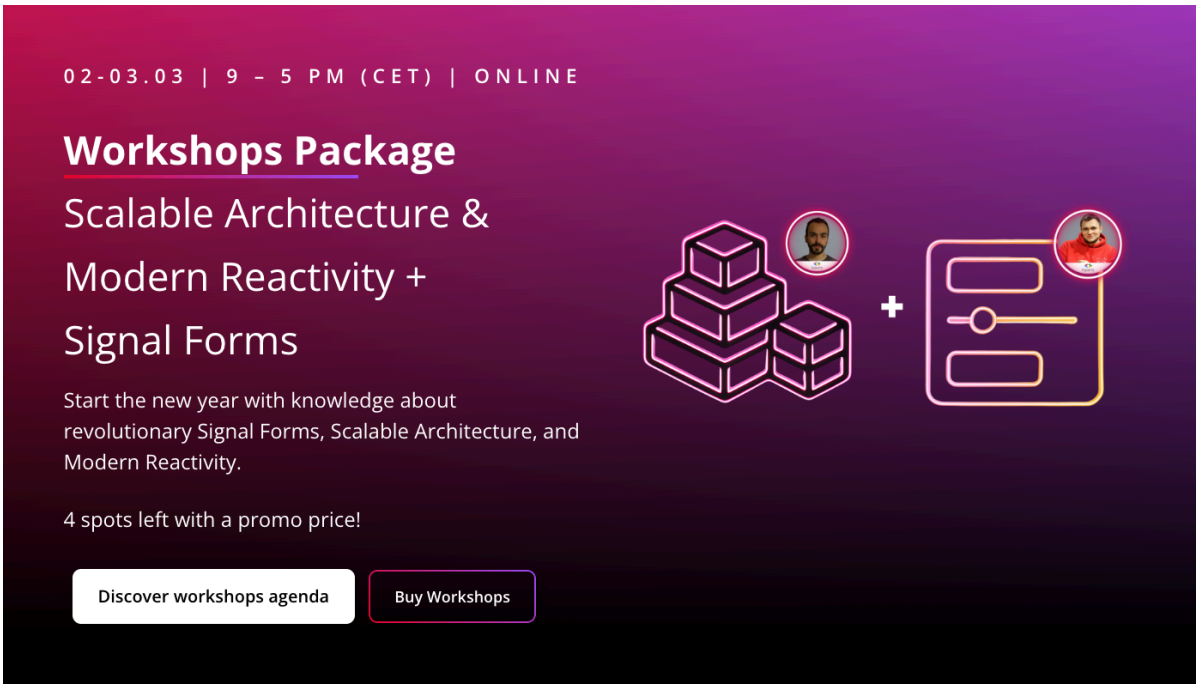
- **Modern Reactivity & Moduliths**: How to organize your workspace with Nx.
- **Resource API**: How to fetch data cleanly (so your forms have data to work with).
- **Signal Graph**: Understanding state management beyond just inputs.

Day 2: Signal Forms Deep Dive (with GDE Mateusz Stefańczyk)

Once the architecture is ready, we dive into the specifics of Signal Forms:

- **Build the Form System**: From nested groups to dynamic arrays.
- **Create Custom Controls**: Implementing FormValueControl patterns.
- **Live Refactoring**: Moving legacy code to the new standard.

Joining 2 days, you connect the dots between how you architect the app and how you implement the forms.



Check more details on the **Scalable Architecture & Modern Reactivity + Signal Forms Workshop** [here](#).