Angular Signal Forms

Build Typed, Validated, Production-Ready Forms with Signals

Model-first form architecture

Schema-driven validation

Custom controls without boiler plate

NIVEK

Meet The Author

Hey there. I'm Kevin Kreuzer, though many people online know me as Nivek. I am a front-end engineer, consultant, streamer, and Google Developer Expert for Angular. Mostly, I am someone who is very into the modern web and still gets excited when Angular gives us a cleaner way to build things.

My story in tech has been a lively one. I have watched web technologies change dramatically, and I have been lucky enough to work right in the middle of that change. Over the years, I have helped companies build, maintain, and modernize web applications, design systems, and internal tools.

What gets me going even more than code is sharing what I learn. I teach on stages, in workshops, through podcasts, videos, articles, and livestreams. In 2019, I was the busiest Angular In-Depth writer, which remains a fun highlight from that part of the journey.

Thanks for picking up this book. I put a lot of care into it, and your support means a lot. But let's not spend too long on me. We have Angular Signal Forms to learn.

Angular v21 edition - Release date: 2026-05-26. Signal Forms are still marked experimental, and several APIs in this book are version-sensitive.

Copyright (c) 2026 Kevin Kreuzer. All rights reserved unless a separate license is provided with the repository.

Table Of Contents

START	Welcome
PART 1	Foundations
01	Form Basics
02	Built-in Validators
PART 2	Validation And Feedback
03	Custom Validators
04	Form Submission
05	Form Metadata
06	Targeted Errors With getError() Angular 22 preview
PART 3	Architecture
07	Subforms And Form Arrays
08	One Form For Create And Edit
09	Data Mapping
PART 4	Integration
10	Custom Form Controls
11	Signal Forms Configuration
PART 5	Migration And Contracts
12	SignalFormControl
13	Legacy Migration With compatForm
14	Standard Schema
END	What Good Signal Forms Feel Like

Welcome

The Angular renaissance continues.

Signals changed the way we think about reactivity. Standalone APIs reshaped how we think about Angular architecture and template context. Signal-based inputs, outputs, and queries, together with the new control flow syntax, deferrable views, and the ongoing zoneless evolution, are all part of Angular's renaissance. They define what modern Angular looks like today — a framework that is more reactive, more ergonomic, and more powerful than ever before.

That is great news.

Still, for a long time, forms were the odd piece out.

Reactive Forms are powerful and proven. Template-driven forms are convenient for small cases. Both are still useful. But neither of them feels native to a signal-first Angular application. You end up with signal state in one part of the component, observable-driven form state in another part, validators expressed through one API, template state read through another API, and a lot of mental switching for something as ordinary as asking, "Is this field valid?"

Signal Forms are Angular's answer to that tension.

They do not just replace one syntax with another. They move the center of gravity. Your form data lives in a model signal. The form API becomes a typed field tree over that signal. Validation is described in schema functions. Field state is read through signals. Custom controls integrate with `model()` properties. Existing Reactive Forms code can move gradually instead of being thrown away.

That is the promise.

API Status And Version Notes

Signal Forms are still marked experimental in the official Angular documentation. That does not make them toy APIs, but it does mean the exact surface can change. Import paths, helper names, option objects, or compatibility utilities may shift between Angular versions.

This book is written around the official Signal Forms documentation that introduced the API in Angular v21 and the related prerelease material available while the book was prepared. The mental model is the durable part: model-first form design, typed field trees, schema-based validation, signal-driven field state, explicit adapters between UI and backend contracts, and careful migration from existing Reactive Forms codebases.

Before copying examples into production, check the documentation for the Angular version your application actually uses.

How To Read The Code

The examples use modern Angular conventions:

- Standalone components.
- `signal()`, `computed()`, `linkedSignal()`, `input()`, `output()`, and `model()`.
- Native control flow with `@if` and `@for`.
- `ChangeDetectionStrategy.OnPush`.
- `form()`, `FormField`, `FormRoot`, validation helpers, and compatibility helpers from `@angular/forms/signals` and `@angular/forms/signals/compat`.

Code samples are there to make the explanation concrete. They are not the main event. When a chapter shows code, read the paragraphs around it. That is where the design decision lives.

> **Snippet conventions**
>
> Most snippets are focused excerpts, not full files. Imports, decorators, providers, and surrounding class code are omitted unless they matter to the idea being taught. When a snippet is intended to name a real file, the file name appears in prose or as the first comment in the snippet, such as `// conference-form.schema.ts`.
>
> Long examples are split into smaller steps so they fit in the PDF and so each block teaches one decision. If a snippet is complete enough to paste, the surrounding text says so explicitly.

Chapter 1: Form Basics

The model signal, the typed field tree, field binding, field state, and reset all start here.

Every Signal Forms form starts with one deliberately simple choice: the value lives in a writable signal.

> **Mental shift**
>
> That sounds small, but it changes the center of gravity. You define a data model, put an initial value into a writable signal, and call `form()` with that signal. The returned object mirrors the model. If the user types into a field, the signal updates. If you reset the form, the signal and field state move together.
>
> The form does not own your data. Your signal owns your data. The form is a typed, reactive view over it.

MENTAL MODEL

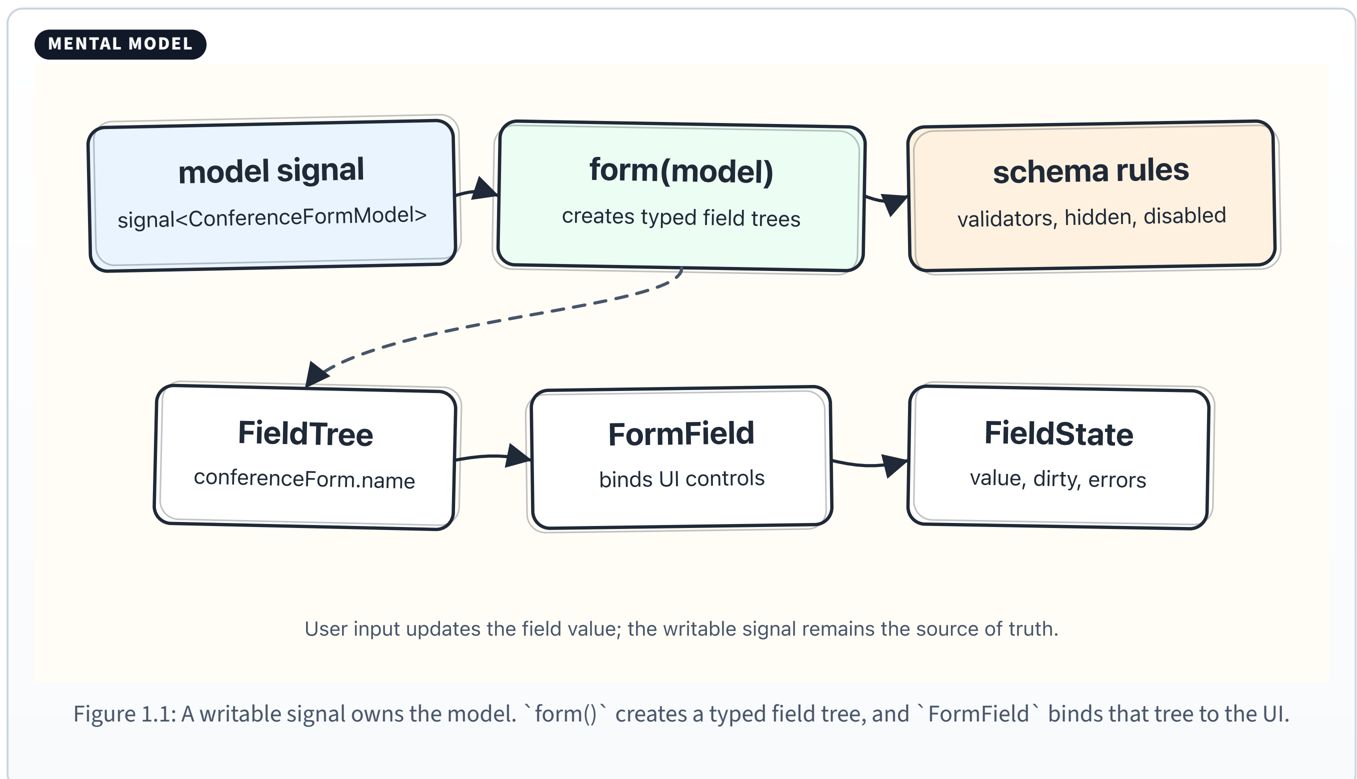

Figure 1.1: A writable signal owns the model. `form()` creates a typed field tree, and `FormField` binds that tree to the UI.

Read the diagram as a map of the pieces we are about to meet. This is only a first pass; field trees, `FormField`, and field state get fuller explanations as the chapter unfolds, and schema rules get their proper introduction in Chapter 2.

The model signal is the writable signal that owns the form's current value. If the form edits a name, a date, or a boolean flag, those values live in the signal. The form API does not replace that signal. It builds on top of it.

`form(model)` creates the form interface for that signal. The result is not a separate copy of the data. It is a typed field tree that follows the same shape as the model, so every field has a stable place in the form.

A field tree (`FieldTree`) is one node in that typed field tree. Think of it as the address of a field, not the current value of the field. You pass a field tree to code that needs to know which field to bind, validate, focus, or compose into a child form.

`FormField` is the directive that connects a UI control to a field tree. When a user types, picks a date, or toggles a checkbox, `FormField` writes that change back through the field tree to the model signal.

`FieldState` is what you read when you want to know what is happening at a field right now. It exposes signals for things like the current value, whether the field is dirty or touched, whether it is valid, and which errors it currently has.

Schema rules are the declarative rules attached to a form: validators, hidden state, disabled state, readonly state, and related field behavior. The first form in this chapter starts without schema rules so the data flow is easy to see. The next chapter gives rules their proper home.

Why This Matters

In Reactive Forms, it is natural to start with the form API itself:

CODE

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

That API is explicit and powerful. But it also means the form object is the place where your value lives. You extract data from the form when you need it.

Signal Forms start from the opposite direction:

CODE

```
const formModel = signal({
  name: '',
});

const fieldTree = form(formModel);
```

Here the writable signal is the source of truth. The form is built around that signal. This makes the form feel closer to the rest of modern Angular, where state is expressed directly and templates react to it.

The Running Example

In this book, we will use one application-shaped example: a form for creating and editing conferences.

The first version is deliberately small. It has a name, a date range, an online toggle, a URL or location field, and a description. As the book progresses, the same form grows validation rules, submission behavior, reusable subforms, API mapping, and a custom location picker.

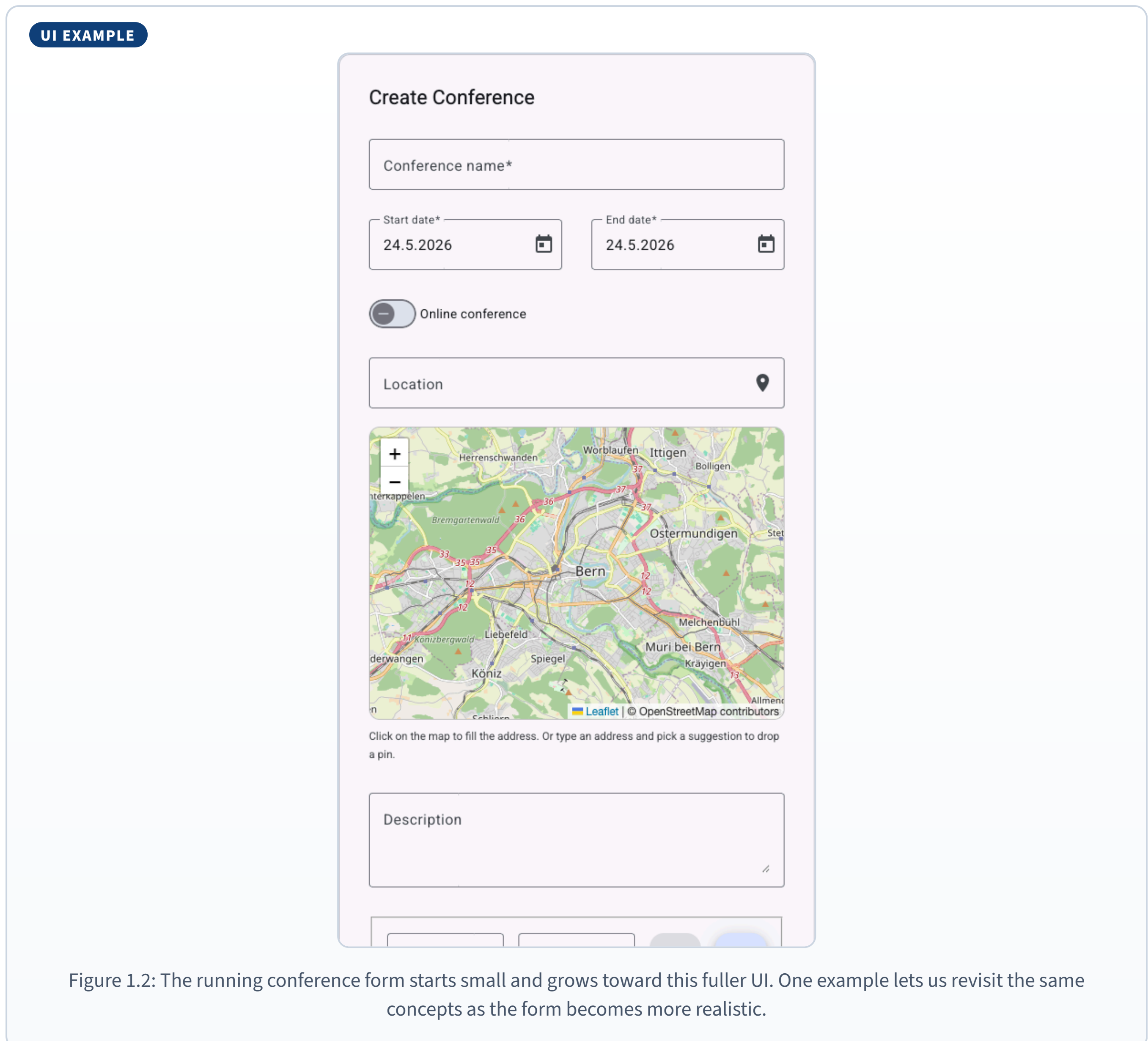

Figure 1.2: The running conference form starts small and grows toward this fuller UI. One example lets us revisit the same concepts as the form becomes more realistic.

We start with the smallest useful question: what shape should the form value have, and which part of the application is that shape serving?

Form Model, Domain Model, API Model

Most applications already have data shapes before a form enters the picture. A backend might store conference dates as ISO strings. A domain model might optimize for business logic. A form has a different job: it collects and edits user input.

A form model is the shape that best serves the input experience. A domain model is the shape that best serves business logic. An API model is the shape that crosses the network. Sometimes they match. Often they do not.

That is why a Signal Form should usually start with a dedicated form model instead of reusing the API model by default.

For the conference form, the UI wants this:

CODE

```
date: {
  start: Date;
  end: Date;
}
```

The API might want this:

CODE

```
startDate: string;
endDate: string | null;
```

Both are reasonable shapes, but they serve different layers. The form can use the UI-friendly shape, then a mapper can translate to the API shape when data is loaded or saved. We will come back to that boundary in the data mapping chapter.

Start With A Form Model

Here is the form model we will use for the first version of the conference form:

CODE

```
export interface ConferenceFormDate {
  start: Date;
  end: Date;
}

export interface ConferenceFormModel {
  name: string;
  date: ConferenceFormDate;
  online: boolean;
  url: string;
  location: string;
  description: string;
}
```

Best practices

This model follows a few deliberate rules:

- Use an explicit `ConferenceFormModel` instead of an anonymous object shape.
- Keep the model focused on one job: creating or editing a conference.
- Match the field types to the UI controls. Datepickers work naturally with `Date` , the online toggle works with `boolean` , and text inputs work with `string` .
- Initialize every field. Empty text inputs start as `''` , the toggle starts as `false` , and the date group has both `start` and `end` .
- Avoid optional properties and `undefined` . If a field should exist in the field tree, it needs a concrete initial value.
- Keep the structure stable. Both `url` and `location` exist even though the UI only shows one of them at a time.
- Group fields that the UI treats as one concept. The date range is one part of the form, so `start` and `end` live together.

With that shape in place, Signal Forms can create a field tree that matches the model. The rest of the chapter builds on that tree: first we create it, then we bind controls to it, and then we read state from it.

Create The Form

Once the model exists, create the initial value:

CODE

```
const conferenceFormInitialState: ConferenceFormModel = {
  name: '',
  date: {
    start: new Date(),
    end: new Date(),
  },
  online: false,
  url: '',
  location: '',
  description: '',
};
```

Then create the signal and the form:

CODE

```
readonly #conferenceFormModel = signal<ConferenceFormModel>(
  conferenceFormInitialState,
);

readonly conferenceForm = form(this.#conferenceFormModel);
```

Core triangle

Signal Forms starts with three pieces:

1. `ConferenceFormModel` tells TypeScript what exists.
2. `signal<ConferenceFormModel>(...)` owns the current value.
3. `form(...)` exposes a field tree for binding and state.

This pattern is the basis of every Signal Form: define the shape, store the current value in a signal, and let `form()` create the typed field tree around it.

FieldTree vs FieldState

This is the first concept that tends to trip people up, so let us slow down.

The form object gives you field trees:

CODE

```
conferenceForm.name;
conferenceForm.date.start;
```

A `FieldTree` is the bindable representation of a field. You pass it to directives and components. It is the thing you use when you want to connect a field to the UI.

CODE
```
<input [formField]="conferenceForm.name" />
```

When you call a field tree as a function, you get field state:

CODE
```
conferenceForm.name();
```

`FieldState` is where the live signals are:

CODE
```
conferenceForm.name().value();
conferenceForm.name().valid();
conferenceForm.name().dirty();
conferenceForm.name().touched();
conferenceForm.name().errors();
```

Parentheses rule

This is where people often get confused: the same pieces can be written with or without parentheses, and the meaning changes depending on where the parentheses go. Choose the field path first, then call it when you want state.

CODE
```
conferenceForm.name; // FieldTree<string>
conferenceForm.name(); // FieldState<string>
conferenceForm(); // FieldState<ConferenceFormModel>
conferenceForm().value().name; // string value from the root state
```

So `conferenceForm.name` and `conferenceForm.name()` are the two expressions you will use most often for one field. `conferenceForm()` is useful when you need root form state, such as the whole value or the form's overall validity. `conferenceForm().name` is not how you navigate to the name field; once you call `conferenceForm()`, you are reading state from the root, not walking the field tree.

The hotel analogy is a useful way to remember that split.

Think of the field tree as a hotel. `conferenceForm.name` is the room address on the floor plan. It tells Angular which room you mean, and it is exactly what a directive needs when it wants to wire an input to that room.

But you are still in the hallway.