Die wichtigsten Neuerungen in Angular 22 im Überblick

Angular 22: Das Signal-Update

von Manfred Steyer

Angular 22 bringt Signals endgültig in den produktiven Alltag. Das Whitepaper zeigt kompakt, welche neuen Features jetzt wichtig sind – und was sie für Performance, Architektur und moderne Angular-Projekte bedeuten.

Mit Angular 22 erreicht das Entwicklungsteam einen weiteren Meilenstein auf dem Weg zur vollständig Signal-basierten Entwicklung. Zahlreiche Features, die bisher experimentellen Status hatten, werden nun stabil und sind damit bereit für den produktiven Einsatz. Dazu kommen wichtige Ergänzungen für Signal Forms, Verbesserungen an der Templatesyntax und viele spannende Abrundungen.

Der Quellcode der gezeigten Beispiele findet sich unter [1].

OnPush als neue Standardstrategie

Eine Entscheidung, die in der Angular-Community von Anfang an diskutiert wurde, ist nun Realität: *OnPush* ist die neue Standard-Change-Detection-Strategie. Diese Umstellung ergibt mit dem Aufkommen von Signals und zoneless Angular konsequent Sinn: Wer Signals verwendet, erhält präzise Benachrichtigungen über Änderungen, und *OnPush* nutzt diese optimal aus. Das Ergebnis ist eine performante Change Detection, die sich zielgerichtet auf die von Änderungen betroffenen Komponenten konzentriert.

Wer das bisherige Verhalten benötigt, setzt die Strategie manuell auf Eager:

```
changeDetection: ChangeDetectionStrategy.Eager
```

Die Einstellung *Eager* ersetzt die ursprüngliche Einstellung *Default*, die nun deprecated ist. In diesem Fall prüft die Change Detection den gesamten Komponentenbaum auf Änderungen.

Um Breaking Changes zu vermeiden, aktiviert *ng update* bei der Aktualisierung der Angular-Version die Option *Eager*, falls *OnPush* nicht explizit gesetzt wurde.

Resource API wird stabil

Das Resource API war bislang das fehlende Bindeglied im Signals-Ökosystem: Es ermöglicht das reaktive, asynchrone Ableiten von Daten, typischerweise HTTP-Anfragen, die nach der Änderung von Signals ausgelöst werden. Trotz seiner zentralen Rolle blieb es lange im experimentellen Status. Das ändert sich mit Angular 22 schlagartig: *resource*, *rxResource* und *httpResource* sind nun stabil und können bedenkenlos produktiv eingesetzt werden.

Den bequemsten Einstieg bietet die *httpResource*-Funktion. Sie erhält einen Lambdaausdruck, der eine HTTP-Anfrage zurückliefert. Dieser Ausdruck ist reaktiv: Ändert sich ein darin verwendetes Signal, startet die Anfrage automatisch neu (Listing 1).

Listing 1

```
import { httpResource } from '@angular/common/http';
import { ChangeDetectionStrategy, Component, signal } from '@angular/core';
import { Flight } from '../../data/flight';

@Component({
  selector: 'app-flight-search',
  changeDetection: ChangeDetectionStrategy.OnPush,
  [...]
})
export class FlightSearch {
  protected readonly filter = signal({ from: 'Hamburg', to: 'Graz' });

  protected readonly flightsResource = httpResource<Flight[]>(
    () => ({
      url: 'https://demo.angulararchitects.io/api/flight',
      params: {
        from: this.filter().from,
        to: this.filter().to,
      },
    }),
    { defaultValue: [] },
  );

  protected readonly flights = this.flightsResource.value;
  protected readonly error = this.flightsResource.error;
  protected readonly isLoading = this.flightsResource.isLoading;

  protected search(): void {
    this.flightsResource.reload();
  }
}
```

Der Typparameter *Flight[]* legt den erwarteten Antworttyp fest. Das *defaultValue*-Argument stellt sicher, dass die Komponente beim Start nicht mit *undefined* konfrontiert wird. Die *Resource* verwaltet ihren Zustand über Signals: *value* enthält die geladenen Daten, *error* liefert Fehlerinformationen, und *isLoading* zeigt den Ladezustand an.

Im Template lassen sich diese Signals direkt nutzen, etwa um den Ladezustand anzuzeigen und die geladenen Daten zu iterieren (Listing 2).

Listing 2

```
@if (flightsResource.isLoading()) {
  <div>Loading ...</div>
}

@if (flightsResource.error()) {
  <div>Error: {{ flightsResource.error() }}</div>
} @else {
  <div class="row">
    @for (flight of flightsResource.value(); track flight.id) {
      <app-flight-card [item]="flight" />
    }
  </div>
}
```

Race Conditions werden automatisch behandelt: Treffen mehrere Anfragen in schneller Folge ein, wird nur das Ergebnis der jeweils neuesten verwendet und ältere werden sofern noch möglich abgebrochen. Das entspricht der Funktionsweise von *switchMap* in RxJS.

Incremental Hydration als Standard

Mit Angular 22 aktiviert *provideClientHydration()* automatisch die Incremental Hydration. Wer sie nicht benötigt, deaktiviert sie explizit mit dem neuen *withNoIncrementalHydration()*-Feature. Eine mitgelieferte Schematic-Migration hilft beim Update.

Signal Forms wird stabil

Auch Signal Forms kann nun produktiv eingesetzt werden. Der Weg vom experimentellen API zur stabilen Version verlief bemerkenswert schnell. Möglich wurde das durch umfangreiche interne Fallstudien bei Google, in denen typische Formularanwendungen systematisch untersucht wurden.

Das Herzstück von Signal Forms ist die *form*-Funktion. Sie erhält ein Signal mit den Formulardaten sowie ein optionales Schema mit Validierungsregeln (Listing 3).

Listing 3

```
import { linkedSignal } from '@angular/core';
import { form, minLength, required } from '@angular/forms/signals';

@Component({ [...] })
export class FlightEdit {
  private readonly store = inject(FlightDetailStore);

  protected readonly flight = linkedSignal(() =>
    normalizeFlight(this.store.flight()),
  );

  protected readonly flightForm = form(this.flight, (path) => {
    required(path.from);
    required(path.to);
    required(path.date);
    minLength(path.from, 3);
  });
}
```

Das Ergebnis ist ein Field Tree, eine tief verschachtelte Signalstruktur. Darin wird jedes Property als Signal mit Formularstatus repräsentiert, etwa *value*, *dirty*, *invalid* und *errors*. Für die Templatebindung kommt die *FormField*-Direktive zum Einsatz:

```
<input [formField]="flightForm.from" id="flight-from" />
<div>{{ flightForm.from().errors() | json }}</div>
```

@Service-Dekorator

Eine auffällige Neuerung ist der neue *@Service*-Dekorator. Er ersetzt die häufigen Fälle, in denen man bisher *@Injectable()* oder *@Injectable({ providedIn: 'root' })* schrieb, und passt besser zur eigentlichen Intention, einen Service bereitzustellen:

```
import { Service } from '@angular/core';

@Service()
export class FlightClient { [...] }
```

Standardmäßig wird der Service im Root Scope bereitgestellt. Wer das nicht möchte, kann den Service stattdessen manuell bereitstellen, etwa in der *app.config.ts*, auf Komponenten- oder Routenebene. In diesem Fall setzt man *autoProvided* auf *false*:

```
@Service({ autoProvided: false })
export class TabRegistry { [...] }
```

injectAsync

Mit *injectAsync* lassen sich Abhängigkeiten lazy injizieren, also erst dann, wenn sie tatsächlich benötigt werden. Das ist besonders nützlich für Services, die umfangreiche Bibliotheken laden und erst bei einer konkreten Benutzeraktion gebraucht werden (Listing 4).

Listing 4

```
import { injectAsync } from '@angular/core';

@Component({ [...] })
export class CheckinPage {
  private readonly upgradeService = injectAsync(() =>
    import('./upgrade-service').then((m) => m.UpgradeService),
  );

  protected async upgrade(): Promise<void> {
    const flightNumber = this.checkinFormModel().ticketId;
    const upgradeService = await this.upgradeService();
    upgradeService.upgrade(flightNumber);
  }
}
```

Angular Track

JavaScript & Angular Days

Entwickle moderne Angular-Apps: Skalierbar, performant, zukunftssicher!

Nutze RxJS, Signals & NgRx für reaktive Anwendungen. Setze Playwright & Jest für zuverlässige Tests ein. Integriere GenAI, SSR & Micro Frontends für smarte Architektur. Lerne in Angular Schulungen, wie du barrierefreie, robuste Apps entwickelst, die im Einsatz überzeugen.

Lerne von führenden Expert:innen:

- **UI:** Barrierefreiheit (A11y), Design Tokens & Storybook für skalierbare Angular-Apps
- **GenAI in Angular:** Copilot & WebLLM für KI-gestützte Entwicklung trainieren
- **Testing:** E2E- und Unit-Tests mit Playwright & Jest praxisnah umsetzen
- **Reaktive Modelle & State RxJS, Signals & NgRx** sinnvoll kombinieren
- **Architektur:** Monorepos, Micro Frontends & Standalone Components nutzen
- **Performance:** SSR, Incremental Hydration, Deferrable Views mit Angular 20 optimieren

Die Funktion *injectAsync* erhält einen Lambdaausdruck, der ein Promise zurückgibt. Das Ergebnis ist eine Funktion, deren erster Aufruf sich um das Laden des Services kümmert. Der Import des *upgradeService* und damit das Laden des zugehörigen Bundles findet erst beim ersten Aufruf von *upgrade()* statt.

Damit Lazy Loading funktioniert, muss der injizierte Service autoprovided sein, also entweder mit *@Injectable({ providedIn: 'root' })* oder dem neuen *@Service()*-Dekorator dekoriert werden.

injectAsync und Prefetching mit onIdle

Lazy Loading bedeutet zwangsläufig eine Verzögerung beim ersten Aufruf. Um diese zu vermeiden, kann das Bundle bereits im Voraus geladen werden. Genau dafür bietet *injectAsync* die Option *prefetch*: Sie verweist auf eine Funktion, die ein Promise liefert. Sobald das Promise resolved wird, beginnt Angular mit dem Laden des Services.

Dieser Mechanismus lässt sich gemeinsam mit der Hilfsfunktion *onIdle* nutzen. Sie liefert ein Promise und löst es auf, sobald der Browser nichts zu tun hat (Listing 5).

Listing 5

```
import { injectAsync, onIdle } from '@angular/core';

private readonly upgradeService = injectAsync(
  () => import('./upgrade-service').then((m) => m.UpgradeService),
  { prefetch: onIdle },
);
```

Intern delegiert *onIdle* an *requestIdleCallback* und fällt auf *setTimeout* zurück, wenn der Browser dieses API nicht unterstützt.

Debouncing für Signals: debounced

Signals wissen nichts von Zeit: Im Gegensatz zu Observables kennen sie deswegen auch kein Konzept für Verzögerung oder Throttling. Das macht Debouncing in reinen Signal-Chains bisher unmöglich, zumindest dann, wenn man dieses mentale Modell nicht brechen will.

Für Formulare hat das Angular-Team deswegen Debouncing direkt in Signal Forms eingebaut, was einen Großteil der Anwendungsfälle abdeckt. Für alles darüber hinaus gibt es nun die Funktion *debounced*.

Im Gegensatz zu Signals kennen Resources sehr wohl Zeit. Genau hier setzt *debounced* an. Die Funktion erzeugt eine Resource, deren Wert um den angegebenen Zeitraum verzögert aktualisiert wird. Der *status* der Resource zeigt dabei an, ob der Wert noch im Pending-Fenster liegt (Listing 6)

Listing 6

```
import { debounced } from '@angular/core';

const filter = signal('');
const debouncedFilter = debounced(filter, 300); // 300ms

effect(() => console.log(debouncedFilter.value()));
```

Änderungen bei disabled, readonly und hidden

Signal Forms kennt die Hilfsfunktionen *disabled*, *readonly* und *hidden*, um Eingabefelder abhängig vom Formularstatus zu steuern. Die Neuerung: Die Bedingung wird nun über eine *when*-Eigenschaft im Parameterobjekt übergeben. Das macht das API konsistenter und ermöglicht es, im Fehlerfall auch eine erklärende Zeichenkette statt *true* zurückzugeben (Listing 7).

Listing 7

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

disabled(path.delay, {
  when: (ctx) => (ctx.valueOf(path.delayed) ? false : 'not delayed'),
});

readonly(path.delay, {
  when: (ctx) => ctx.valueOf(path.delayed),
});

hidden(path.delay, {
  when: (ctx) => ctx.valueOf(path.delayed),
});
```

Der *ctx*-Parameter stellt dabei kontextuellen Zugriff auf andere Feldwerte bereit, sodass komplexe, feldübergreifende Bedingungen sauber ausgedrückt werden können.

Aktive Routen ermitteln

Mit *isActive* bringt Angular 22 eine neue Möglichkeit, programmatisch zu ermitteln, ob eine Route aktiv ist. Die Funktion liefert ein Signal und wird im Template automatisch neu ausgewertet, sobald sich der Routenzustand ändert (Listing 8).

Listing 8

```
import { isActive, Router } from '@angular/router';

@Component({ [...] })
export class BookingNavigation {
  private readonly router = inject(Router);

  protected readonly flightSearchActive = isActive(
    '/ticketing/booking/flight-search', this.router
  );
  protected readonly passengerSearchActive = isActive(
    '/ticketing/booking/passenger-search', this.router
  );
  protected readonly summaryActive = isActive(
    '/ticketing/booking/summary', this.router,
    { paths: 'exact' }
  );
}
```

Der optionale dritte Parameter nimmt *IsActiveMatchOptions* entgegen und steuert, wie genau der Vergleich ausfallen soll (Tabelle 1).

Option	Werte	Bedeutung		
paths	'exact' \	'subset'	Müssen alle Segmente übereinstimmen, oder reicht ein Subset?	
matrixParams	'exact' \	'subset' \	'ignored'	Vergleich der Matrix-Parameter der übereinstimmenden Segmente
queryParams	'exact' \	'subset' \	'ignored'	Vergleich der Query-Parameter
fragment	'exact' \	'ignored'	Vergleich des URL-Fragments	

TABELLE 1 Ein Vergleich ermittelt, ob eine Route aktiv ist

Standardmäßig gilt für *paths* Subset-Matching: Eine Route gilt als aktiv, wenn sie ein Subset der aktuellen URL ist. Im Beispiel wurde für *summaryActive paths: 'exact'* gesetzt, damit die Route exakt übereinstimmen muss. Im Template nutzt man das Signal direkt für die aktive CSS-Klasse:

```
<a [routerLink]="['./flight-search']" [class.active]="flightSearchActive()">
  Flight
</a>
```

HttpClient: FetchBackend als Standard

Der *HttpClient* setzt ab Angular 22 standardmäßig auf das *FetchBackend*. Das explizite *withFetch()* ist damit deprecated und kann entfernt werden.

Der Hintergrund: Das Fetch API ist heute in allen relevanten Browsern und modernen JavaScript Runtimes verfügbar. Gegenüber *XMLHttpRequest* (XHR) bietet es ein moderneres, Promise-basiertes API, bessere Unterstützung für Streamingszenarien und eignet sich zudem besser als gemeinsame HTTP-Abstraktion für Browser und SSR. Allerdings unterstützt Fetch keine Upload-Progress-Events.

Passend dazu ersetzt Angular 22 die bisherige pauschale *reportProgress*-Option durch zwei dedizierte Varianten, die Upload- und Downloadfortschritt getrennt schalten. Wird *reportUploadProgress* zusammen mit dem *FetchBackend* verwendet, wirft Angular eine Exception. Das ist ein bewusst harter Hinweis, dass in diesem Fall *withXhr()* erforderlich ist.

Wer das ursprüngliche Verhalten benötigt, zum Beispiel für eine Fortschrittsanzeige beim Upload, wechselt zurück zu XHR:

```
provideHttpClient(withXhr());
```

Bei der Versionsaktualisierung ergänzt *ng update* automatisch *withXhr()*. Das verhindert unerwünschte Verhaltensänderungen in bestehenden Anwendungen.

Verbesserungen an der Templatesyntax

Angular 22 bringt eine Reihe nützlicher Erweiterungen für Templateausdrücke. Häufig switcht man nicht auf eine Union selbst, sondern auf eine ihrer Properties, etwa den Discriminator einer diskriminierten Union. TypeScript kann dann zwar die abgefragte Property einengen, aber nicht erkennen, ob die übergeordnete Union dadurch vollständig abgedeckt ist. Angular 22 schließt diese Lücke. Mit *never(<ausdruck>)* lässt sich gezielt angeben, welcher Ausdruck auf vollständige Abdeckung geprüft werden soll (Listing 9):

Listing 9

```
state!: { mode: 'show'; menu: number }
      | { mode: 'hide' };

[...]

@switch (state.mode) {
  @case ('show') { {{ state.menu }} }
  @case ('hide') {}
  @default never(state);
}
```

Die Angabe *never(state)* übermittelt dem Compiler explizit, dass er die vollständige Abdeckung gegen die übergeordnete *state*-Union prüfen soll. Wird sie später um ein weiteres *mode* ergänzt, ohne dass ein entsprechendes *@case* existiert, meldet der Template-Compiler den Fehler.

Angular 22 bringt zudem zwei Präzisierungen beim Umgang mit *null* und *undefined*. Das Verhalten von *?.* im Template entspricht nun exakt JavaScript-Semantik: Bricht die Kette an einer *null*- oder *undefined*-Stelle ab, ist das Ergebnis *undefined*. Wer das alte Angular-spezifische Verhalten benötigt, kann den Ausdruck mit *$null(...)* umschließen:

JavaScript Track

JavaScript & Angular Days

JavaScript 2026: Moderner entwickeln, sauberer strukturieren!

Starte mit einem JavaScript-Crashkurs und entdecke Trend-Themen für Einsteiger:innen, Umsteiger:innen und Fortgeschrittene. Nutze neue Syntax, Toolchains & Browser-APIs für saubere, performante Projekte. Teste automatisiert mit klaren Patterns. Strukturiere smart mit JSON-Modulen & nativen Tools. Jetzt Wissen tanken, das echten Fortschritt bringt.

Lerne von führenden Expert:innen:

- **JavaScript 2026:** Neue Syntax, Toolchains & Browser-APIs einsetzen
- **Automatisiertes Testing:** Komponenten mit klaren Patterns absichern
- **Projektstruktur:** JSON-Module & native Tools sauber strukturieren
- **Komponentenarchitektur:** Provider & Compound Patterns gezielt anwenden
- **Async & Datenflüsse:** Streams, Iterators & immutable Arrays nutzen

```
{{ user?.profile?.name }}
```

Darüber hinaus erlaubt der Typprüfblock nun echtes TypeScript-Narrowing über *?.*. Nach einem Truthiness-Check wird der Typ wie in regulärem TS-Code eingeengt, sodass der Folgezugriff ohne *?.* typsicher ist (Listing 10).

Listing 10

```
@Component({
  template: `
    @if (user?.isMember) {
      {{ user.isMember }}
    }
  `,
})
export class UserComponent {
  user?: { isMember: boolean };
}
```

Schließlich sind in Angular 22 auch *//*- und */* … ***/-Kommentare innerhalb von HTML-Element-Definitionen erlaubt. Das ist praktisch, um lange Attributlisten zu strukturieren (Listing 11).

Listing 11

```
<div
  // primary button
  class="btn btn-primary"
  /*
    Achtung: greift nur, wenn `loading` false ist
  */
  [disabled]="loading()"
></div>
```

@defer: optionaler Timeout für on idle

@defer (on idle) kann jetzt ein Timeout in Millisekunden erhalten, analog zu *IdleRequestOptions.timeout*. So lässt sich verhindern, dass ein *defer*-Block endlos auf eine Idle-Phase wartet, die nie eintritt:

```
@defer (on idle(2000)) {
  <heavy-widget />
}
```

Weiteres im Überblick

- *httpResource und Transfer State:* Ressourcen, die auf dem Server vorgeladen werden, spielen nun nahtlos mit dem HTTP Transfer State zusammen. Das verhindert redundante HTTP-Anfragen beim ersten Rendern im Browser. Der Client verwendet einfach die bereits serverseitig geholten Daten.
- *Web MCP Tools (`provideExperimentalWebMcpTools`, `declareExperimentalWebMcpTool`):* Angular-Anwendungen können KI-Tools direkt in den Injector registrieren. Beim Zerstören des Injectors werden die Tools automatisch wieder ausgetragen, ideal in Kombination mit Route Providern und *withExperimentalAutoCleanupInjectors*.
- *ApplicationRef.bootstrap mit Config: bootstrap()* auf der *ApplicationRef* akzeptiert nun analog zu *createComponent* ein Konfigurationsobjekt. Das ist besonders relevant für Microfrontends, die bedarfsgesteuert in bestimmte Bereiche der Seite geladen und gebootstrappt werden: *appRef.bootstrap(MyComponent, { hostElement: document.querySelector('#root')! })*.



Angular Bootcamp

Generative AI für Angular
AI-Integration im Frontend und Backend

Lerne in 2 Tagen:
- **Generative AI & RAG:** Konzepte & Praxis-Use Cases
- **AI-Backends aufbauen:** LangChain & moderne APIs
- **UX & Streaming-APIs** für Anwendungen
- **Angular-Integration:** TypeScript & Frontend-Workflows
- **Enterprise-Szenarien:** KI in Business-Anwendungen einsetzen

Jetzt entdecken!



JavaScript Bootcamp

Fullstack JavaScript
Von Node.js bis React & Angular

Lerne in 2 Tagen:
- **Fullstack erleben:** Backend-APIs bis Frontend-Frameworks
- **REST & GraphQL:** Konzepte verstehen und richtig einsetzen
- **Datenvalidierung:** Eingaben zuverlässig prüfen
- **Frontend-Integration:** Angular & React anwenden
- **CI/CD & Testing:** Deployment-Workflows direkt anwenden

Jetzt entdecken!

- *Bootstrap unter Shadow Roots:* Angular kann jetzt direkt unter einer Shadow Root gestartet werden. Styles werden im *SharedStylesHost* korrekt an der übergeordneten Shadow Root registriert. Das ist ein weiterer Schritt Richtung sauberer Web-Component- und Microfrontend-Integration.
- *AI-Runtime-Debugging:* Im Dev Mode registriert Angular KI-Debugging-Tools auf der Seite. Dazu gehört angular:di-graph, das den vollständigen Dependency Injection Graph (Element und Environment Injectors) für In-Page-KI-Assistenten bereitstellt. Das ermöglicht zukünftig KI-gestützte Analysen des DI-Graphen direkt im Browser.
- *Node.js 26:* Angular kompiliert und läuft nun offiziell auf Node.js 26.

Fazit

Angular 22 markiert einen wichtigen Reifepunkt in der Signals-Ära: Mit dem stabilen Resource API und Signal Forms stehen nun zwei der zentralen Bausteine für moderne, reaktive Angular-Anwendungen für den Produktiveinsatz bereit. Der neue *@Service*-Dekorator, die Debouncing-Funktion *debounced*, dynamische Schemata und das Submission API runden das Bild ab und zeigen, wie konsequent das Angular-Team an einem kohärenten, ergonomischen API-Design arbeitet.

Die Erweiterungen der Template-Syntax und die neue reaktive Funktion *isActive* vervollständigen eine Version, die in ihrer Breite beeindruckt. Wer in den letzten Monaten mit dem Update gewartet hat, bekommt jetzt ein stabiles Fundament, das den Umstieg auf Signal-basierte Entwicklung auch ohne experimentelle Features erlaubt.



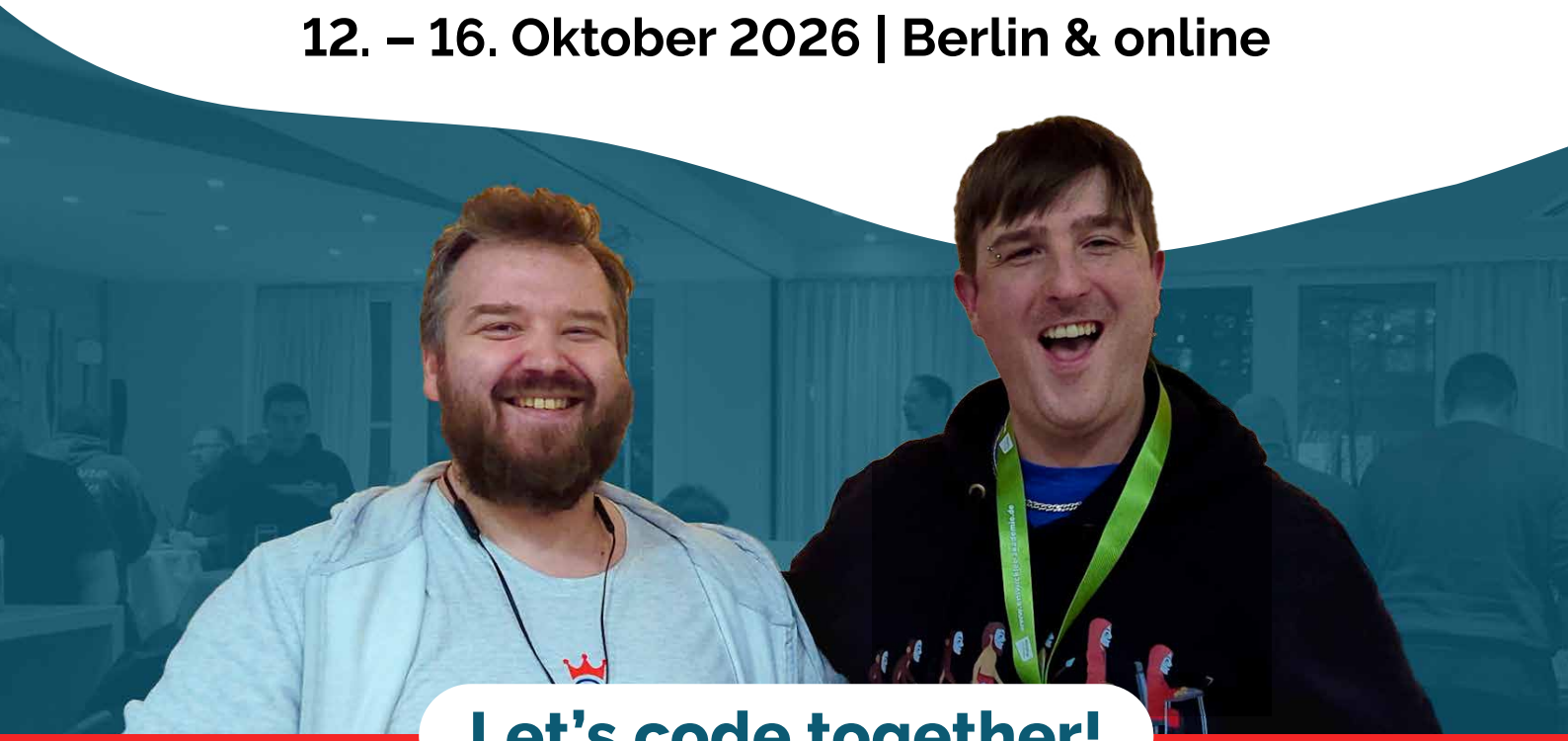
Manfred Steyer ist Trainer und Berater mit Fokus auf Angular und Agentic AI, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java-Magazin und Heise Developer. Unter www.ANGULARarchitects.io bieten er und sein Team tiefgehende Angular-Schulungen und Beratung an.

Links & Literatur

[1] https://github.com/angular-architects/flights42.git

JavaScript & Angular Days

12. – 16. Oktober 2026 | Berlin & online



Let's code together!

Das große Trainingsevent für moderne Webentwicklung mit JavaScript & Angular

Erlebe neue Trends

Erreiche neue technologische Tiefe und lerne zukunftssichere Fähigkeiten. In den ersten drei Workshop-Tagen stellst du dein eigenes Programm zusammen, anschließend geht es zwei Tage zum Deep Dive ins JS- oder Angular-Bootcamp.

Learning by Doing

80 % des Programms besteht aus Workshops. Hier wird nicht nur studiert, sondern direkt ausprobiert. Egal, ob du live in Berlin oder online zugeschaltet bist.

Triff die Community

Wir verbinden dich mit Menschen, die alle eins im Sinn haben: Sich auf hohem Level auszutauschen und neue Kontakte zu knüpfen. Hier kommen alle zusammen: 20+ Toptrainer:innen der Branche und die Teilnehmenden.

JS TRACK

TRACK

+2 Bootcamps



Präsentiert von: entwickler.de akademie entwickler.de

javascript-days.de/berlin