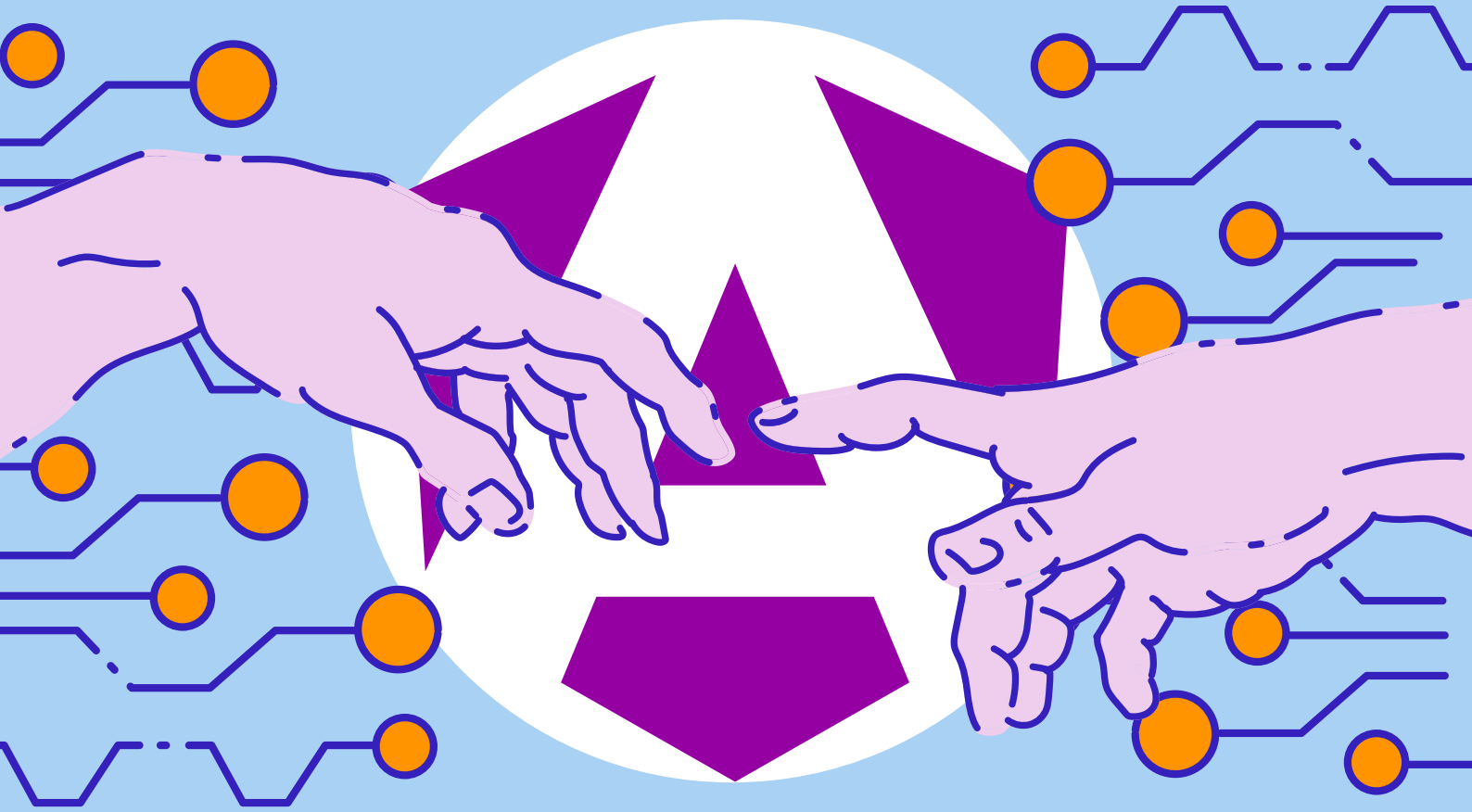
devmio

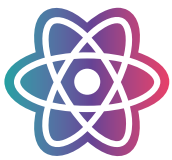
iJS magazine
JavaScript & Web Development

Understanding The Angular Renaissance



Content

Angular



React

JavaScript
Practices & Tools

General Web
Development

Node.js

All Angular 17 features at a glance

Welcome to the Angular Renaissance

There's a lot going on with Angular. Some people can lose track of things, or even become scared of getting overwhelmed. This article examines if this concern is justified or if the Angular team's plan to usher in an Angular renaissance can succeed.

by Karsten Sitterberg

First of all: Anyone who has been putting off Angular upgrades or is only planning to update at the end of an LTS cycle should closely watch current developments and evaluate if incremental updates in small steps instead of a large leaps makes more sense in terms of effort and risk. Of course, we'll take a look at the Angular Core changes as well as TypeScript and Angular Material.

TypeScript

The new Angular version also supports a new version of TypeScript. In Angular 17, TypeScript version 5.2 or higher must be used. Since Angular 16 still required TypeScript 5.0, let's take a look at some interesting new features from both TypeScript 5.1 and TypeScript 5.2.

Implicit *return* when *undefined* is returned

If a JavaScript function doesn't contain a *return* statement, it still always implicitly has *undefined* as its return value. This fact wasn't reflected in TypeScript yet. If *undefined* was specified as the return type for a function, a *return* statement always had to be used explicitly too. Additionally, a function that explicitly returned *undefined* wasn't compatible with a *void* function. See Listing 1.

This is now improved in TypeScript 5.1, so the above example no longer throws compiler errors. A function with *undefined* as the return type also no longer requires an explicit *return*, similar to the *demo1()* function in Listing 2. Alternatively, an explicit but *empty* return can also be used, as in the *demo2()* function.

Independent getter and setter types

Previously, TypeScript get and set functions that belonged together (for example, if they had the same name) had to have compatible types. For a setter that can be given the type *string|number|boolean*, previously you could only specify a subtype like *string* as the corresponding getter. In TypeScript 5.1, these types can be assigned completely independently, but they must be assigned explicitly. See Listing 3.

Decorator Metadata

In TypeScript, a decorator is a function that is given a class, property, or method in order to extend the passed entity's behavior. TypeScript 5.2 adds an exciting feature specifically for the Angular framework: the ability to add metadata to a decorator. This feature was already in the *Experimental Decorators* originally used by Angular, but

Listing 1: *undefined* and *void* were previously incompatible in TypeScript

```
declare function onEvent(f: () => undefined): undefined;

onEvent(function f() { })

// Argument of type '() => void' is not assignable to parameter of
                                        type '() => undefined'.
```

Listing 2: No more return needed for the *undefined* return type

```
function demo1(): undefined {
// no returns
}
function demo2(): undefined {
return;
}
```

now it's also in the stable TypeScript specification. For example, in Listing 4, the decorator *setMetadata* is defined. A decorator is a function that's given the decorated entity as a *target* and the associated *context*. The *metadata* property was added to this context object in TypeScript 5.2. The metadata can be understood as a key-value store. Its corresponding TypeScript type is called *record*. The metadata key is the name of the decorated entity; in Listing 4, for example, the property *demoAmount* is decorated. Therefore, it becomes a metadata key. Listing 4 shows how metadata can be written by accessing *context.metadata*. The decorated entity's name is in *context.name*. In Listing 4, the string *Demo data* is stored in the metadata for each property decorated with *@setMetadata*. Stored metadata can be read from the class with the TypeScript metadata symbol (*Symbol.metadata*).

To better configure a decorator, you can implement a decorator factory. It will then return the actual decorator. This is shown in Listing 5 using the *DemoMeta* decorator factory, which receives the metadata key to be set and the associated metadata value as parameters and returns the configured decorator. The *DemoMeta* factory is used in Listing 5 to add the metadata entry *'entity'* with the value *'3270'* to the *DemoClass* class. The metadata entry *'demo'* with the value *'42'* is added to the *demoMethod()* method. Then, the class metadata is accessed again with *Symbol.metadata*.

Listing 3: Independent getter and setter types.

```
interface MyElementStyling {
set style(newValue: string);
get style(): CSSStyleDeclaration;
}
```

Listing 4: Set and read metadata in TypeScript

```
interface Context {
name: string;
metadata: Record<PropertyKey, unknown>;
}
function setMetadata(_target: any, context: Context) {
context.metadata[context.name] = 'Demo-Data';
}
class MyClass {
@setMetadata
demoAmount = 123;
@setMetadata
demoAction() { }
}
const ourMetadata = MyClass[Symbol.metadata];
console.log(JSON.stringify(ourMetadata));
// { "demoAmount": "Demo-Data", "demoAction": "Demo-Data" }
```

After our look at TypeScript, next we will turn to the Angular CLI. This is typically used to build Angular applications.

Angular CLI

The Angular team provides the Angular CLI for developers to manage Angular projects. An important step in Angular CLI 17 is the Angular build system update. In Angular CLI 17, newly created projects are now built with the new build system based on *esbuild* as the standard for the first time. This is primarily meant to speed up the build and optimize the build result or make it smaller.

The new name *Vite* (pronounced like the French. "vite" = "fast") based live development server promises a better, faster development cycle. For example, Vite lets you apply changes to the global CSS without needing a live reload.

For apps to be built entirely with the esbuild-based build system in the future, the esbuild builder needed some enhancements. For example, basic support for checking Angular CLI bundle budgets was added to help developers keep track of the size of JavaScript and CSS files generated in the build. Basic support for building WebWorkers has also been added. These are a browser-native feature, but still must be considered in the Angular build, since worker scripts are typically written with TypeScript. The old builder was already able to do this, and now the new esbuild builder follows suit.

Since they're built in Angular, SPAs have one disadvantage. Once the initial web page has loaded, some scripts must first be loaded, which start the application. Depending on the internet connection and the page size, this takes a moment. Along with the fact that scripts have to be executed in order to display the content of the page, this is bad for the page's SEO ranking.

Previously, the *@nguniversal* package could be used to make an Angular application SEO-fit. *@nguniversal* can (pre-)render an Angular application on the server side. The page content is visible immediately after the HTML

Listing 5: Example Decorator Factory in TypeScript

```
function DemoMeta(key: string, value: string) {
return (_: any, context: Context) => {
context.metadata[key] = value;
};
}

@DemoMeta('entity', '3270')
class DemoClass {
@DemoMeta('demo', '42')
demoMethod() {}
}

DemoClass[Symbol.metadata].entity; // '3270'
DemoClass[Symbol.metadata].demo; // '42'
```

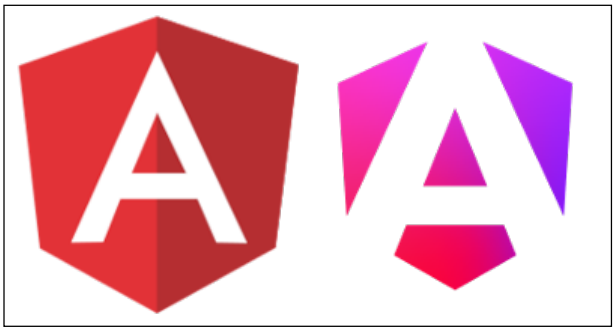


Figure 1: Old Angular-Logo on the left and new Angular-Logo on the right.

has been loaded, since it no longer has to be generated dynamically in the client. This leads to a fast page load, better usability, and a better SEO ranking. Previously, the pre-rendering packages were in a separate namespace *@nguniversal*, although these packages were also maintained by the Angular team. In Angular 17, most of the code has been moved from *@nguniversal* to the *@angular/ssr* package in the *@angular* namespace. However, apart from renaming, the functionality is essentially the same.

The topic of SSR (Server Side (pre-) Rendering) has become more important to the Angular team, as can be seen from the above change and the fact that it has become easier to integrate packages. When creating a new Angular 17 project, you'll be asked if server-side rendering should be activated. Alternatively, the new option *ng new --ssr* can be used to generate a project with preconfigured SSR. In this project, Angular 16's hydration is also directly activated. This will ensure a seamless transition from the server-side generated page to the fully functional Angular app without flickering.

Breaking Changes in Angular CLI 17

Angular 17 keeps its own dependencies up to date. At least TypeScript version 5.2 and zone.js version 0.14.0 must now be used for Angular and Angular CLI. The minimum required node.js version is upgraded to 18.13.0.

Some defaults have also been replaced in Angular CLI. For example, when a new application is generated, the Angular router is now initialized by default. This wasn't previously the case. If no routing is needed, the command line option *--no-routing* must be specified.

As of Angular CLI 17, all applications are generated as *standalone* applications by default, such as an application without *@NgModule()*.

Another changed default value concerns the Angular interceptors. Angular HTTP interceptors can be generated with the *ng g interceptor* command. Previously, class-based interceptors were generated with this command. In Angular CLI 17, functional interceptors are generated instead. If a class-based interceptor is generated, the *--no-functional* option must be added to the command.

Angular

Angular 17 comes with a lot of new features. The Angular team calls it one of the "biggest" releases in its history. Besides the version updates for zone.js and TypeScript, it also offers many smaller and larger innovations. Two innovations related to the new APIs regarding Signals have already been added in Angular 16.2.0. Namely, it adds two new component lifecycle hooks. These are special because they don't have to be implemented with an interface like conventional lifecycle hooks. They can be called as a callback function. This is similar to the *computed()* function in Signals. Listing 6 shows an example with the new hooks. The two hooks are called *afterRender()* and *afterNextRender()* and are executed, according to their name, after Angular renders the respective component. The differ-

Listing 6: Implementing the new lifecycle hooks

```
@Component({})
export class DemoComponent {
val = 42;
constructor() {
afterRender(() => {
console.log('afterRender', this.val);
});
afterNextRender(() => {
console.log('afterNextRender', this.val);
});
}
}
```

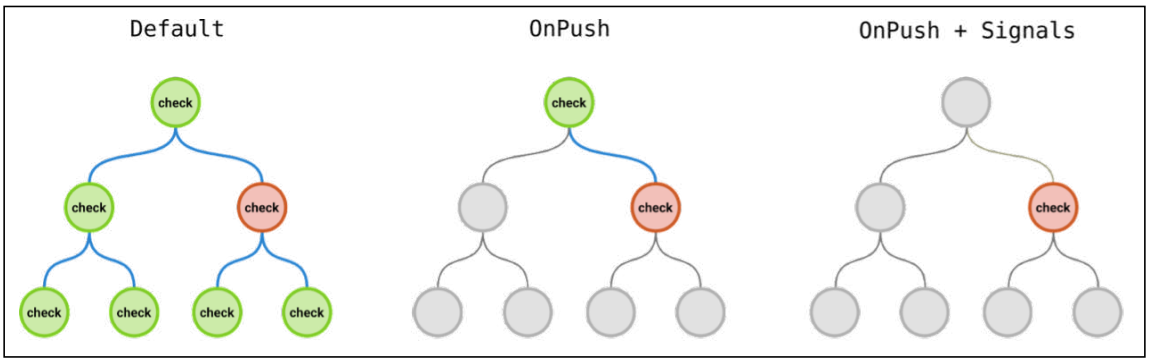


Figure 2: Comparing change detection mechanisms

ence between the two hooks is that *afterNextRender()* is only called once, after the next rendering process, while *afterRender()* remains active as long as the component is active. However, these two hooks are still listed as *Developer Preview* and are subject to change.

Signals

An important detail in Angular 17 is that the signals introduced in Angular 16 are now stable. The goal of introducing Signals was to make zone.js optional by introducing a new, signal-based change detection mechanism. In Angular 17, the team has come a step closer to this goal. Signals enable *local change detection*, a change detection process that only refers to the components that have changed (signal) values. **Figure 1** shows an example of the difference between the change detection mechanisms. In the standard case, the complete component graph is checked for each asynchronous event. With OnPush-ChangeDetection, components that triggered the change detection or have changed *@Input()* values and their respective parent components (up to the AppComponent) are checked. With Signals, you can now check only the components that have changed signal values.

Listing 7: Activating the ViewTransitions in the AppModule or app.config

```
provideRouter(
routes,
withViewTransitions({
onViewTransitionCreated: transitionInfo => {
}
})
),
```

Listing 8: Previous "Control Flow" with Angular Directives

```
<div><span *ngIf="show">Content here</span></div>
```

Listing 9: "@if" with the new Control Flow syntax in Angular 17

```
<div>@if (show) { <span>Content here</span> }</div>
```

Animations

Animations can be reloaded in Angular 17 with lazy loading. The browser must first load smaller bundles, making the initial application smaller, leading to shorter application loading and start times. To activate lazy loading for animations, the *provideAnimationsAsync()* function must be used. Here, the *BrowserAnimationModule* should no longer be imported and the *provideAnimations()* function should no longer be used.

The special thing about an SPA is that, as a rule, an HTML document is only loaded when the application starts. When switching to a different route, only components in the browser DOM are exchanged. Animations cannot simply be executed on page load, as is the case with server-side applications. Instead, animations must be better adapted to SPA logic, such as route changes. This is what the new View Transitions DOM API [1] is designed for. This API provides the callback *document.startViewTransition()*, which tells the browser that a view transition will start. Logic can be passed to the function that defines how the browser DOM should look at the end of the transition. The transition animation can be created using special CSS pseudo selectors. Angular uses this new API in the Angular 17 router so that a view transition is started for each navigation. However, since the *View Transition* feature is still experimental and has only been implemented in Chrome and Edge, router navigation in browsers such as Firefox or Safari continues to take place synchronously. To activate ViewTransitions, the *withViewTransitions()* function can be added to the router at startup. In Listing 7, the *withViewTransitions()* is also given a configuration object in which an *onViewTransitionCreated()* callback can be defined if additional (animation) logic is needed.

New Control Flow Syntax

The biggest and most far-reaching innovation for developers, and the biggest innovation in Angular in terms of template syntax, is the new control flow syntax. In Angular 17, this syntax is still in a developer preview. But in the future, it will completely replace the previ-

ous directives *ngIf*, *ngFor* and *ngSwitch*. The logic of these directives will become part of the framework itself, instead of "normal" directives. This will allow the Angular team to better optimize the respective logic. Signals and the goal of making zone.js optional with the help of Signals were also a key reason for these innovations, since the old control flow directives were based on and needed *zone.js*.

Besides the control flow logic, another feature that can be implemented thanks to the new syntax is a new form of lazy loading at component level. This feature, also known as "deferrable views", is discussed in more detail below.

First, let's familiarize ourselves with the basic syntax. Until now, *control flow* in Angular templates was handled by structural directives like *ngIf* and *ngFor*. See Listing 8. In the future, structural directives can still be used and built, but *ngIf*, *ngFor* and *ngSwitch* will be implemented with the new syntax. The example in Listing 8 can be easily rewritten in the new syntax, as shown in Listing 9. You'll see that the *if* is no longer an HTML attribute, but an independent syntax construct marked with an @. The condition is specified in round brackets, just like in TypeScript. This is followed by the template to be displayed in single curly brackets.

Incidentally, the new syntax has changed slightly since the first RFC. In the original proposal, the example in Listing 9 would look a little more complex: (*{#if show} ... {/if}* statt *@if (show) { ... }*).

The new syntax has a clear advantage over the old structural directives, especially for *if-elseif-else* conditions. Previously, separate templates had to be created in an " " element for the *else* case. *if-elseif-else* constructs weren't possible at all. *if-else* constructs had to be nested instead. Listing 10 shows the new syntax, where both are easily possible. The syntax is reminiscent of JavaScript. It's important here that the keywords *@if* and *@else* are always preceded by an @.

The **ngFor* directive will also be replaced with the new control flow syntax *@for*. Listing 11 shows the corresponding syntax. You will immediately see that in the *for* statement, *item of items* is now written. It is no longer *let item of items*, as was needed with **ngFor*. You must now specify a *track* option that takes over the role of the *trackBy* option from **ngFor*. There is an important difference here. The new *@for-track* option isn't optional, it must always be specified. It's used internally by Angular to optimize rendering the list so that the list does not have to be completely re-rendered all the time, especially if just individual list entries are changed. A completely new algorithm for list rendering was made possible by the new *@for* syntax. It also helps here and shows significant performance improvement in benchmarks compared to the old algorithm. The value specified in the *track* expression must identify the respective collection entry. If an array of objects is iterated over, the respective object ID should be used, as seen in Listing 11. If you are iterating over an array of primitive values, you can simply use the value itself (*track item*).

An innovation that we did not know from **ngFor* until now is *@for* can be extended by *@empty{}*. The template code in the *@empty* expression is always displayed if the transferred list (here *items*) has no entries or is *null* or *undefined*.

Several implicit variables can also be used within the *@for* expression to display the current loop iteration index (via *$index*), for example. See Listing 11. You can also access the loop length (*$count*) to check if the current loop index is even (*$even*) or odd (*$odd*), or if it is the first (*$first*) or last (*$last*) loop element.

Previously, three directives were used to implement a switch case statement: *ngSwitch* and the structural directives **ngSwitchCase* and **ngSwitchDefault*. With the new syntax, *@switch*, *@case* and *@default* can now be used, as seen in Listing 12. Of course, you can still switch via primitive values or enum values, for instance. Just like the other control flow statements, normal Angular template code can be used as the cases content. You can also nest the control flow statements.

Listing 10: *@if-@else* with Angular 17 Control Flow syntax in the template

```
@if (show) {
<div>case content 1</div>
} @else if (otherCond) {
<div>case content 2</div>
} @else {
<div>other content</div>
}
```

Listing 11: Example of a loop with the new *@for* Syntax in Angular 17

```
@for (item of items; track item.id) {
<p>Element Nummer {{$index}}:
{{item.name}}</p>
} @empty {
<p>No items</p>}
```

Listing 12: Example of a switch case with the new *@switch* syntax in Angular 17

```
@switch (value) {
@case (1) {
<div>case one</div>
}
@case (2) {
<p>case two</p>
}
@default {
<span>Default-case</span>
}
}
```

Listing 13: Example of the new *@defer* expression in Angular 17

```
@defer (when isLoaded) {
<app-main-content/>
} @loading {
<p>Loading...</p>
} @placeholder {
<icon>pending</icon>
} @error {
Failed to load
}
```

The last element of the new control flow syntax offers new syntax and completely new functionality. *Deferred Views* (with @*defer*) enable lazy loading at the component level without needing the router or complex renderer constructs. An example of a deferred view can be seen in Listing 13. Here, the component " " is the component that will be reloaded. The *when* expression is optional. If it's omitted, lazy loading begins once the outer component is rendered. Otherwise, the *when* expression can be used to load the @*defer* block's content only after a certain condition occurs. In the simplest case, a Boolean expression (*isLoaded*) can be queried, as seen in Listing 13.

Listing 14: Examples of *on* expressions that load directly or only in the idle state

```
@defer (on immediate) {
//...
@defer (on idle) {
//...
```

Listing 15: Options for loading defer content based on an interaction trigger

```
@defer (on interaction(trigger))
{
<app-my-content/>
}
<button #trigger>Trigger
                    </button>

@defer (on interaction) {
<app-my-content/>
} @placeholder {
<button>Trigger</button>
}

@defer (on hover(trigger)) {
<app-my-display/>
}
<button #trigger>Trigger
                          </button>
```

Listing 16: Viewport triggers are used to load content when the trigger element is visible (in the viewport)

```
@defer (on viewport(trigger)) {
<app-my-content/>
}
<div #trigger style="margin-top: 1500px">Content</div>
```

Listing 17: Timer condition and minimum specifications

```
@defer (on timer(1500ms)) {
<app-my-content/>
}
@loading (after 100ms; minimum 150ms) {
<p>...Loading...</p>
}
@placeholder (minimum 100ms) {
Placeholder
}
```

In addition to the @*defer* block, Listing 13 shows more optional blocks. A template can be specified in @*loading* to represent the *loading status*, like a loading spinner. The loading status is displayed the moment the Boolean condition (here *isLoading*) changes to *true*. When the "" component is loaded and rendered, the loading status is automatically removed. Developers should be especially careful with the loading state and avoid UX pitfalls. One large spinner for the entire page is easier for a user to *digest* than 20 simultaneously active spinners spread across the entire page and all individual components. When using load states, take care to make sure the page doesn't permanently "flicker" due to piecemeal reloading the smallest page elements or that the content doesn't constantly shift back and forth.

Here, @*placeholder* content is displayed until the *isLoaded* condition first has the value *false*. One important note here: @*defer* doesn't work like @*if* in the sense that you can switch back and forth between the values. Once *isLoaded* has the value *true*, the lazy loaded component is displayed. Even if *isLoaded* is manually reset to *false*, the @*defer* content continues to be displayed and doesn't switch back to the @*placeholder* content.

If an error occurs when reloading the @*defer* content, (due to a network interruption, for example), content from the @*error* block is displayed.

Besides the *when* specification with a Boolean expression, you can also specify an *on* condition. Listing 14 shows an example of the *on immediate* and the *on idle* condition. Here, *on immediate* behaves as if the expression were omitted. Once the outer component is rendered, components in the defer block are reloaded. With *on idle*, the components are loaded as soon as the browser is in the *idle* state (technically, this corresponds to the browser's *requestIdleCallback()* function).

In addition to these fairly static specifications, you can also react to dynamic user interactions with the *on interaction* conditions. To react to an interaction, an element that the user should interact with must be specified. The example in Listing 15 shows that either an external element like a button, can be transferred to the *interaction* condition using a template reference variable. Alternatively, this explicit assignment can be omitted. But then a @*placeholder* must be specified, since the placeholder content serves as the interaction element. This is shown in the middle of Listing 15.

This could be used to reload a detailed view at the push of a button, which contains complex parsing logic (or parsing and displaying XML) in the form of components and services that should only reload *on demand*.

The *on hover* condition works similarly to the *on interaction* specification. See Listing 15. The only difference is that instead of an interaction, you only need to hover over the trigger element.

Instead of direct user interaction, you can also load a defer block when another (trigger) element comes into

the visible viewport. for instance, when the user scrolls to the trigger element. See Listing 16. Instead of the external trigger element, the placeholder can also be used as an implicit trigger element.

Last but not least, a defer block can also be reloaded using a time trigger. You must give a trigger a duration to load the component after. For example, Listing 17 specifies that the defer block should start loading the " " component after 1.5 seconds.

Regardless of the *on* condition used, the *@loading* block can be given an *after* and/or a *minimum* duration. In Listing 17, the *after* specification ensures that the loading state is displayed after 100 milliseconds at the earliest. The *minimum* specification makes sure that the loading state is displayed for at least 150 milliseconds. Similarly, the *minimum* specification for *@placeholder* ensures that the placeholder is displayed for at least 150 milliseconds before displaying the loading state.

The new defer syntax also allows direct optimization. An optional *prefetch* statement can also be specified in the defer condition. The logic specified after the prefetch statement corresponds to the defer logic described above. In Listing 18, a component should only be displayed when the state of the variable *isVisible* changes to *true*. The prefetch condition is used so the component can be displayed now without loading time. Once the *prefetchCondition* variable changes to *true*, Angular prefetches the *@defer* block content. If *isVisible* changes to *true*, no more loading time is needed, since the component is already *present* and only needs to be rendered. In the example shown in Listing 18, an *on* condition is used for pre-fetching instead of a *when* condition. In this case, Angular starts reloading the component when the outer component has rendered. Angular CLI 17 will have an automatic migration to transform the old control flow directives to the new syntax. This should simplify changing to the new syntax. But at least in Angular 17, both syntax variants can still be used.

Listing 18: Prefetch conditions

```
@defer (when isVisible; prefetch when prefetchCondition) {
//...
@defer (when isVisible; prefetch on immediate) {
//...
```

Listing 19: Comparing adding items to a *Signal<Array>* between Angular 16/17

```
// With Angular 16
items.mutate(itemsArray => itemsArray.push(newItem));

// From Angular 17:
items.update(itemsArray => [itemsArray, ...newItem]);
```

Breaking Changes

Since Angular 17 is a major release, there's also a few breaking changes. One breaking change concerns the Signal feature introduced in Angular 16. Signals were still in a preview mode in Angular 16, hence the short time until this change. With Signals, it's been shown that the *signal.mutate()* function can cause inconsistencies in the application. Therefore, the option to change a Signal has been removed. Signals should no longer be changed by mutation, but should be used like an immutable data structure instead. Listing 19 shows an example of how a Signal containing an array could previously be changed and how it should be done from Angular 17 onwards.

Besides Signals, there are also breaking changes in the router. So far, this has the properties:

- *canceledNavigationResolution*
- *paramsInheritanceStrategy*
- *titleStrategy*
- *urlUpdateStrategy*
- *malformedUriErrorHandler*
- *urlHandlingStrategy*

With Angular 17, these properties were removed from the router's public API. Instead of using the router, these properties should now be configured with the corresponding configuration parameters in the *provideRouter()* function or the *RouterModule.forRoot()* hook.

Angular 17 in Practice

The changes allow new types of architecture and optimizations which were previously impossible or difficult to implement. An example from practice illustrates this:

A played a leading role in developing an integrated platform to provide effective, secure, fully GDPR-compliant remote training. If you look at a live screenshot (**Figure**

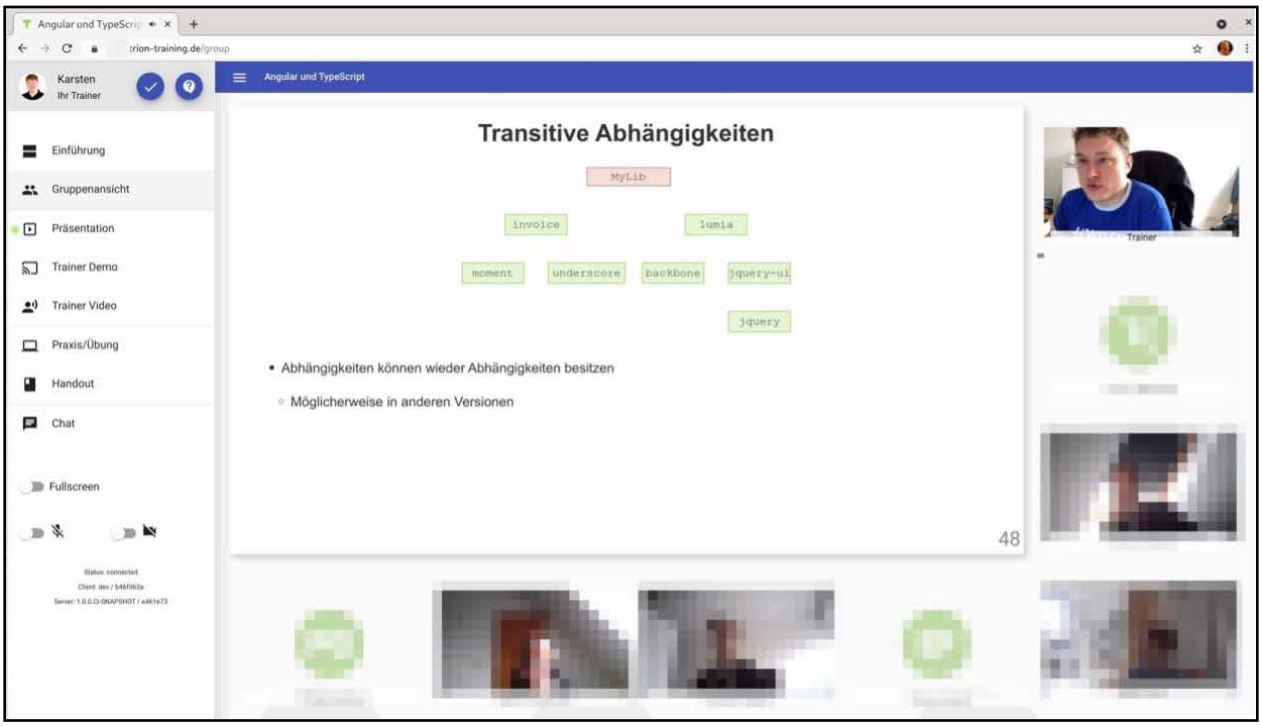


Figure 3: A complex Angular real-time application

3), you'll see that the audio and video control is directly available in the primary view (bottom left switch). This means that the associated components and services cannot be reloaded later with lazy loading, since there's no nested route that would allow the code to be split accordingly.

This is now possible thanks to the new option to carry out code splitting and lazy loading at the component level. This means that initial application loads are considerably faster, since audio/video functionality in particular is complex and accounts for large portions of the code.

It is also advantageous to be able to do without zone.js and implement better controlled and fine-grained change detection using Signals. This leads to an overall improved performance and reduced load on the client system.

On the other hand, this requires corresponding changes that cannot be implemented by an automatic code transformation. All involved developers must learn the new concepts and master them in depth in order to effectively and meaningfully use these new possibilities.

By the way, what about the Angular module system? It still exists today, but the plan is to make modules optional in the future. This is also intended to make it easier for new developers to get started with Angular.

Angular Material

Angular Material 17 introduces both an important and significant change for developers still using old Material components, known as *legacy* since Angular 15. Angular 15 is called legacy because the Angular Material team decided to replace the previously used CSS with the official Google Material Design Components (MDC) style. Because some components needed to be significantly revised, the Angular Material team prefixed the old versions with *Legacy* and deprecated them, but they are still allowed to be used.

In Angular Material 17, all legacy components with a modern MDC counterpart will be removed. This means that applications must be migrated to the new MDC-based Material components by Angular 17.

In the JavaScript universe, there are several ways to work with dates. To be able to use these different date-time libraries in Angular Material as well, Angular Material offers a *DateAdapter*. The DateAdapter is then used by the Datepicker, for example, for date management. The *DateAdapter* is an abstract class that must be implemented to use a specific date-time library. Angular Material includes the built-in *NativeDateAdapter*, which simply uses the native *Date* object available in the browser for data management.

This *NativeDateAdapter* has previously received the *Platform* service via constructor injection. With Angular Material 17, this is no longer the case. In addition, the *NativeDateAdapter* no longer uses constructor injection generally, but the *inject()* function instead. This major change has an impact in particular, if you have created the NativeDateAdapter yourself using a constructor, for example in unit tests. This is no longer possible, instead the NativeDateAdapter must be created using Angular's dependency injection.

Important Changes in CSS

To integrate Angular Material styles into SCSS, the @*import* mechanism of SCSS could be previously used. However, since this mechanism has long since been deprecated on the SCSS side, the @*import* option has now also been removed on the Angular Material side. As of

Angular Material 17, the new SCSS *@use* syntax must be used to import the Material styles.

If you want to use Angular Material with custom styling, you can do so using the Material theming API. Previously, Material did not check whether the custom themes were formally correct but with Angular Material 17, these are now strictly validated. For example, if the *mat.button-typography* mixin is used with a theme in which *typography: null* is set, there will now be an error during the SCSS build.

In addition to importing the global Material styles, it's also possible to load only the styles for individual components in the build. To do this, you can use the SCSS mixin *mat.-theme*, where "" corresponds to the component name, for example *card*, *checkbox*, *radio*, or *list*.

Additionally, you can also load only individual parts of the component styles, such as the button typography (via the mixin *mat.-typography*), the colors (via the mixin *mat.-color*), or the density styles (you could also say *spacing styles*, the mixin is *mat.-density*). With Angular Material 17, even more styles have been added that are available in the component *-theme* mixins, but not in the sub-mixins. To use these styles, you must then use the *mat.-base* mixin. To add the basic styles for all components, there is now also the mixin *mat.all-component-bases*.

Conclusion

Is Angular still known as the framework with long-term investment security? Is Angular's popularity waning or is the framework able to conquer new target groups without alienating existing teams and developers? Or is there even a plan to consciously abandon existing teams and developers in order to modernize Angular?

Angular is being renewed so quickly that it feels almost like a new framework altogether. It will be interesting to see how long the old concepts will continue to be supported. After all, an ng-upgrade transformation of the source code is not enough;users also need to learn new concepts and replace old ones, which can temporarily reduce productivity. To what extent do third-party libraries support old and new Angular concepts? This is an important question, as the Angular team's communication with existing users could be improved. Additionally, users are concerned about Google's history of discontinuing products, which raises doubts about Angular's long-term viability.

The new Angular features are well-designed and enable new architectures and even more high-performance front-end applications. They also significantly improve the developer experience, especially by providing options for expressing if-elseif-else statements.

The rapid development pace of recent Angular versions could be challenging especially for enterprise customers and large projects with long-term maintenance requirements. Existing users may find it difficult to keep up, which could lead to frustration and concerns about investments made, and could ultimately limit Angular's adoption and traction with new users.

The Angular team could certainly work on improving its relationship with users by communicating more sensitively and carefully considering the risks and benefits of new features. Java is a good example of this: it consistently innovates, but with enough lead time and maximum compatibility even with very old codebases. Which is why Java is the preferred language in the enterprise environment. The Angular Ivy migration and Java Loom project would be a good comparison, massive benefits were created through high-quality engineering, without requiring developers to make massive efforts themselves. It's clear that this can't always be achieved, but it's perhaps worth keeping this in mind as a goal.

For Angular beginners, it's best to orient yourself with the latest concepts and avoid outdated literature, blog posts, and training courses.

Karsten Sitterberg holds a master's degree in Physics and is an Oracle-certified Java developer who specialises in modern Web APIs and frameworks such as Angular, Vue, and React. In his workshops, articles, and presentations, he regularly reports on new trends and interesting developments in the world of frontend tech. He is the co-founder of the meetup series "Frontend Freunde," the Java User Group Münster, and the Münster Cloud Meetup.

Links & Ressources

[1] https://developer.mozilla.org/en-US/docs/Web/API/View_Transitions_API

Exploiting TypeScript's type system

TypeScript's Meta-programming Language

TypeScript extends JavaScript with a static type system that can be used to write more than just simple type annotations to variables and parameters. In fact, TypeScript brings a kind of metaprogramming language that can be used to describe types programmatically. This allows complex (JavaScript) code structures to be described in a type-safe manner. This article will introduce this meta-language with some examples.

by Nils Hartmann

JavaScript has a dynamic type system. Variables and function parameters can take on any values in it, as well as any types. Explicit specification of types is not possible. This means you have a great deal of freedom and flexibility, but the approach is also error-prone and can quickly lead to problems during both further development and while using code.

TypeScript addresses this problem and provides a static type system for JavaScript. To deal with the complexity arising from JavaScript's dynamic type system, you can write normal type annotations in TypeScript and describe types programmatically with a kind of meta-programming language. The following will explain this meta-language using a typical JavaScript function from a technical point of view and the correct types will be developed. Source code for this article can be found in a "TypeScript Playground".

Listing 1 shows a function with the technical idea to check an object with values. It is intended to be an example for functions that accept random data and return it in a modified form. The validate function expects two JavaScript objects. The first is an object value, which contains the set of values that will be checked. The second parameter is also a JavaScript object, rules, which contains callback functions that the validate function can use to check individual values of the values object. Implementing the validate function calls the validate function for the corresponding rule function from the *rules* object for each entry in the *values* object. This returns the validation result for each entry (either *true* or *false*, depending on if the value is valid or not). If a function is stored under a key in the values object, its return value will be passed to the *rules* function. If a function is stored under a key in the values object, then its return value will be passed to the *rules* function.

Listing 1

```
function validate(values, rules) {
  // Implementation omitted
}

const person = {
  firstname: "Klaus",
  age: 32,
  sallary() { return 60000 }
}

const personRules = {
  validateFirstname(s) {
    return s.length > 3
  },
  validateSallary(n) {
    return n > 1000;
  }
}

const result = validate(person,
personRules);

if (result.firstname === false) {
  // firstname invalid
}

if (result.sallary === true) {
  // salary valid
}
```

In order for the validate function to know which function in the *rules* object belongs to which field in the values object, the functions should be stored under a key with a similar name to the corresponding key in the values object. The name should follow the pattern *validate-FieldName*, beginning with the string validate, followed by the key name with a capital letter. If there is an entry *firstname* in the rules object set to the type string (or whose value is the type string), then the validate function in the rules object expects a function under the key *validateFirstname* that can accept a parameter of the type string. It will then return a boolean. If this function does not exist in the *rules* object, then the value is considered valid without testing.

Then, the *validate* function returns an object containing the results of the checks. In it, the names of the keys correspond to the keys in the *values* object and the values are the result of the validation (*true* or *false*).

This behavior can be implemented in JavaScript without problem (at least from the point of view of the type system), since the very general data type object can be used both for the parameters and their return value. An object can be created in JavaScript at any time without describing its type or its infrastructure in more detail, or at all. In comparison, in languages like Java or C#, developers may need to implement multiple classes or interfaces in each case Then the validation function would have to know, based on rules, which field in the class for the function parameter should correspond to which field in the rule and return type.

In TypeScript, the type-level rules described above can be described with a Mapped type. This can be used to express rules, like: “Whatever keys are contained in the object passed to the validate function as the first parameter, the return type will be that all keys of the passed object are present, but the values will be of the type boolean.” Or: “The second parameter should be an object containing keys that match the named naming convention - and no keys beyond that.”

Without having to manually write types for each object to be validated, TypeScript knows what function arguments and the return type look like. With this information, TypeScript can (as with an explicitly defined type) verify correct usage and assist with development with code completion, etc. The concept of mapped types is very powerful, which the examples below will introduce step-by-step. Finally, the *validate* function can be described in a completely type-safe manner.

Generics

As in other programming languages, generics are used in TypeScript to describe general types or general function signatures. This allows type variables to be declared on a function which can assume the type of a function argument and be reused in other places in the function signature. Let’s have an example. A fictitious function *saveToDatabase* is to write an arbitrary value into a database. It can also change this value (for example, by incrementing a version counter) and return the updated value to the caller. In principle, the method signature in TypeScript could use the *any* type, to which all values can be assigned (Listing 2). This type can be used for both the function argument and the return type.

The problem is that TypeScript does not know what the exact type is when it comes to the return type. More precisely, the function parameter can be any type, but the return type also corresponds to this type. A type variable can be used to describe a generic function (Listing 3). The type variables are declared in

Listing 2

```
function saveToDatabase(value: any): any { /* Implementation omitted */ }

const result = saveToDatabase({firstname: "Klaus"});
// result is here any, so that no further type checking occurs
result.firstname; // OK
result.lastname;  // No compile error, although lastname is not defined
on the object
```

Listing 3

```
function saveToDatabase<O>(value: O): O { /* Implementation omitted */ };
const result = saveToDatabase({ firstname: "Klaus" });

result.firstname; // OK
result.lastname; // ERR: Property 'lastname' does not exist
```

angle brackets and can be used in the function signature wherever TypeScript expects a type specification. So, here it is in the function parameter and as a return type.

At first, nothing changes for the function caller, since TypeScript automatically assigns the content of the type variable (derived from the value passed for value). Type variables can be used in functions and in the description of type aliases (more examples later), interfaces, and classes.

Listing 4

```
function saveToDatabase<0 extends object>(value: 0): 0 { /* ... */ };
saveToDatabase("Klaus"); // Error: "Klaus" is not an object

function saveToDatabase<0 extends { id: string }>(value: 0): 0 { /* ... */ };
saveToDatabase({
  // Error: id property missing
  firstname: "Klaus"
})
```

Listing 5

```
function addListener<0 extends object>(
  object: 0,
  key: string,
  listenerFn: Function) { }

  const person = {
    firstname: "Klaus",
    age: 32
  }

  addListener(person, "firstname", (newValue) => { /*... */ });
  addListener(person, "age", (newValue) => { /*... */ });

  // This call should result in a compile error:
  addListener(person, "lastname", (newValue) => { /*... */ });
```

Listing 6

```
type Red = "red";
type Black = "black";

// Variables, arguments etc. of type Red can now only contain the
string// "red" accept, everything else leads to compile error
const redColor: Red = "red"; // OK
const blackColor: Red = "black" // ERR

type Colors = Red | Black;
const r: Colors = "red"; // OK
const b: Colors = "black"; // OK
const g: Colors = "grey"; // ERROR
```

The saveToDatabase function can now be called with arbitrary types, objects, primitive types like *string* or *number*, and with *null* or *undefined*. This behavior isn't always desired. In the case of the saveToDatabase function, it's possible that it can only work with objects. For this, restrictions can be described for a type variable with the keyword extends, which must be observed when using the function or the type variable. Listing 4 shows two more examples of possible signatures for the *saveToDatabase* function. The first signature specifies that the function may be passed any object (but not a *string*, *number*, *null*, etc.). The second signature restricts the specification even further and expresses that the passed object must have at least one property id, which must be of type *string*.

The keyof operator

A common use case is to pass an object and a string with a valid key from the object to a function. This constraint can also be mapped with TypeScript, so that a compile error will occur if a string that's passed to the function doesn't match the name of a key.

Listing 5 shows an *addListener* function that should be able to register a listener function (for example, an event handler) for a specified entry in any object. It is passed three parameters: the object, the name of a key from the object, and the actual listener function to be registered for that field.

To make the use of this function type-safe, the type string cannot be used for the second parameter, which allows arbitrary strings. If an object (person) is passed to addListener with the keys *firstname* and *age*, the allowed strings for the second parameter are only first-

name and age. This can be expressed in TypeScript with a string literal type. This type describes a concrete string. To define a type that can accept multiple strings, a union type can be defined for it (Listing 6).

For the above example with the person object, a union type would have to accept the two strings *firstname* and *age*. All other strings are not allowed because they do not represent valid keys in the person object. However, the *addListener* function accepts person objects as well as any object. Therefore, the set of allowed strings results from the passed object and cannot be listed from the outset and described with a type.

Listing 7

```
const person = {
  firstname: "Klaus",
  age: 32
}

type KeysOfKlaus = keyof typeof person;
  // "firstname" | "age"
```

Listing 8

```
function addListener<O extends object, K extends keyof O>(
  obj: O,
  name: K,
  listener: Function
) {
  // ...
}

const person = {
  firstname: "Klaus", age: 32
};

addListener(person, "firstname", v => { /* ... */ }); // OK
addListener(person, "lastname", v => { /* ... */ });  // ERR:
  // Argument of type "lastname" is not   // assignable to parameter of
type "firstname"  // | "age"
```

Listing 9

```
const person = {
  firstname: "Klaus",
  age: 32,
  address: { city: "Hamburg" }
};

type Person = typeof person;
type Age = Person["age"]; // number
type Address = Person["address"]; // { city: string }
```

Instead, the keyof operator can be used. This returns a union type consisting of all of an object's keys (Listing 7). The *addListener* function can use the *keyof* operator to express that the second parameter only accepts keys from the passed object. Anything else will result in a compile error. Furthermore, editors can support code completion and suggest only valid values (Listing 8).

Index Accessed Type

The callback function (third parameter) passed to addListener is called as soon as the value in the object that it was registered to changes. The new value should be passed to it with -addListener_. This is why the callback function must have exactly one argument. This argument needs to be the same type as the corresponding entry in the monitored object. For the key *"firstname"*, which is of type *string*, a *listener* function must be passed with an argument of type string. For the field *"age"*, the callback function needs to have an argument of type number.

In the addListener function shown in Listing 8, the type Function is currently still used for the parameter that the listener function is passed with. This describes an arbitrary function and does not make a statement about function arguments or return values. However,

Listing 10

```
function addListener<O extends object, K extends keyof O>(
  obj: O,
  name: K,
  listener: (newValue: O[K]) => void
) {
  // ...
}

// When using the addListener function, no specification of a type is required:

const person = {
  firstname: "Klaus",
  age: 32,
};

addListener(person, "firstname", (newValue) => { /* ... */ }); // OK

// This call does not work:
// ERR: Argument of type "lastname" is not assignable to parameter of
// type "firstname" // | "age"
addListener(person, "lastname", (newValue) => { /* ... */ });

function ageChangeListener(newAge: number) { /* ... */ }

addListener(person, "firstname", ageChangeListener);
  // ERR: Argument of type '(newAge: number) => void' is not assignable
  // to parameter of   // type '(newValue: string) => void'.
  //      Types of parameters 'newAge' and 'newValue' are incompatible.
```

functions can be described accurately with TypeScript. The callback function for the firstname listener, which expects one argument of type string and returns nothing, can be described with the following: *(newValue: string) => void*.

To correctly type the argument to a callback function for *addListener*, we need the type that the associated entry in the passed object has. Types of entries in an object can be determined with an Index Accessed Type. The notation is identical to the index operator in JavaScript, except you don't specify an object, but a type instead (Listing 9). Since we know both the type of the object (type variable O) and the name of the key (type variable K) from it in the signature of the *addListener* function, we can retrieve the type from the passed object for the passed key. We can use it to describe the signature of the expected listener function (Listing 10). This provides a complete description of the addListener function. The caller of this function has full type safety without having to specify or even describe a single type!

Listing 11

```
type Rules<O extends object> = {
  [K in keyof O]?: (value: O[K]) => boolean;
};
type ValidatedObject<O extends object> = {
  [K in keyof O]: boolean;
};

function validate<O extends object>(
  values: O, rules: Rules<O>
): ValidatedObject<O> {
  return /* ... */;
}

const person = {
  firstname: "Klaus", age: 32
}

const result = validate(person, {
  firstname(v) {
    // v is correct here as string
    return v.length > 3
  },
  age(a) {
    // a is correct as number
    return a > 0;
  }
})

result.age; // OK, boolean
result.lastname; // ERR: lastname does not exist // an result

// Also OK: no rule function for 'age
validate(person, {
  firstname(v) {
    return v.length > 3
  },
});

// The type of the rule object for a concrete // object can also be
determined in order to specify it// explicitly if needed:
type PersonRules = Rules<typeof person>;
const rules: PersonRules = {
  firstname(n) { return n.length > 3; }, // OK
  lastname(l) { return true } // ERR: lastname   // not in 'person'
}
```

Mapped Types

Let's go back to the *validate* function described at the beginning. This function should return an object containing the same keys as the passed values object. However, the values in it should be of type *boolean*. This rule can be expressed in TypeScript with mapped types. A mapped type takes an existing type and maps it to a new type. The existing type can be either a hard-coded type or, as needed in the validate function, a generic type. Regardless, the mapped type's description can iterate over the output type's keys with *keyof* and assign a new type to each entry.

Listing 11 shows a first variant for the type description of the *validate* function with a mapped type. It makes sure that the passed rules object is a subset of the passed values object (by adding the question mark in the mapped type, the entries are made optional). However, the entries set must be a function that takes a type with the same value as the in values object as a parameter (similar to the *addListener* example). The keys should also be renamed (*validateFieldName*), but we will take care of that later. The return type is already described according to requirements. The returned object contains all keys from the values object, but their values are each of the type *boolean*.

Here, TypeScript can ensure correct specification of the rules object and the correct use of the return type, without needing a type specification from the caller of the *validate* function.

Conditional Types

If a function is stored under a key in a passed *values* object, the validate function should call this function when validating and pass its return value to the associated *rules* function. Here, the type in the rules object must correspond with the function's return type in the *values* object (Listing 12).

Conditional types can be used to implement this requirement. This allows existing types to be checked for conditions. A conditional type looks like the ternary operator and returns a different type depending on the check's outcome. The simple example in Listing 13 shows the conditional type's basic operation. The *getLength* function takes a *string* or *null* passed in. If the string is passed at runtime, then its length will be returned (*number*). If *null* is passed, the function will

also return null. The conditional type can be used to describe this return type in exactly the same way.

For the *validate* function, this means that a conditional type can be checked while defining the rule type for each key, whether the associated value in the values object is a function or not. Depending on this, the type can be set in the rule object. Listing 14 shows an example of this behavior. If the type in the values object for an entry is a function, it is assumed in the *rule*s object that it will return a *string*. Therefore, the corresponding *rule* function also processes a *string* as a parameter (we will determine the correct type in the next step). If it isn't a function, the type for the parameter is taken unchanged from the values object, just as before.

You can do more than just check a type with a conditional type. Types contained in the validated type (function parameters, return values, contents of arrays, etc.) can also be extracted from it. This behavior can be used for the validate function to extract the return value of functions in the *values* object. For this, a variable is written in the type to be checked with the *infer* keyword at the position where a type is to be extracted. Provided that the checked type matches the formulated condition, variables marked with *infer* are available in the true branch (after the question mark) and are set by TypeScript to the types present in the checked object at the marked positions. Listing 15 shows a conditional type that checks if the type passed corresponds to any function and, if so, has its return value extracted to the type variable *R* and is returned. Otherwise, the checked type will simply return unchanged.

This allows the validate function's requirements to be implemented by specifying the conditional type that was previously used in the *rules* type. For better readability, Listing 16 defines a separate type for a rule function, accommodating this logic.

Template Literal Types

Now, almost all of the requirements for the validate function's types are implemented. But the expected key names in the rules object aren't correct yet. They shouldn't be called *fieldname*, but *validateFieldname* instead (*validate* and then the name of the key from the values object, starting with a capital letter).

Listing 12

```
const person = {
  firstname: "Klaus",
  salutation: function () {
    "Hello, " + this.firstname;
  }
};

// Expected type of the associated rules object:
type RulesForPerson = {
  firstname?: (newValue: string) => boolean;

  // person.salutation returns a string, // therefore newValue must also
be string here:
  salutation?: (newValue: string) => boolean;
};
```

Listing 13

```
function getLength<0 extends string | null>(s: 0)
  : 0 extends string ? number : null
  { /* ... */ }

const l = getLength("Morning!"); // l is // "number"
const n = getLength(null); // n is "null"
const x = getLength(7); // ERR: 7 is // not a string
```

Listing 14

```
type Rules<0 extends object> = {
  [K in keyof 0]?: 0[K] extends Function
    ? (value: string) => boolean
    : (value: 0[K]) => boolean;
};

// Validated0bjectand validate-Function // unchanged

const person = {
  firstname: "Klaus",
  salutation() {
    return "Hello, " + this.firstname;
  }
}

validate(person, {
  firstname(f) { return f.length > 3 },
  salutation(s) {
    // s is a 'string' here (not 'Function'    // as in 'person')
    return s.length > 7;
  }
});
```

Listing 15

```
type GetReturnType<0> = 0 extends (...args: any) => infer R ? R : 0

function sayHello() { return "Hello" };
type SayHelloReturn = GetReturnType<typeof sayHello>; // string

function sayNothing() { };
type SayNothingReturn = GetReturnType<typeof sayNothing>; // void

type NotAFunction = GetReturnType<string>; // string
```

Listing 16

```
type RuleFunction<O extends object, K extends keyof O> =
  O[K] extends (...args: any) => infer R
          ? (value: R) => boolean
          : (value: O[K]) => boolean;

type Rules<O extends object> = {
  [K in keyof O]?: RuleFunction<O, K>
};

// ...ValidatedObject and validate-// Function unchanged...

const person = {
  firstname: "Klaus",
  salutation() { // returns string
    return "Hello, " + this.firstname;
  },
  sallary() { // returns number
    return 123456;
  }
}

validate(person, {
  salutation(s) {
    // s is 'string' here
    return s.length > 7;
  },
  sallary(n) {
    // n is the correct number here
    return n > 1000;
  }
});
```

Listing 17

```
function sayHello<S extends string>(s: S): `Hello, ${S}!` {
  return `Hello, ${s}!`;
}

const s = sayHello("Susi"); // Typ von S: "Hello, Susi!"

type Spacing = "margin" | "padding";
type Direction = "top" | "right" | "bottom" | "left";
type Unit = "px" | "em" | "rem"

function setSpacing(s: `${Spacing}-${Direction}`, size:
`${number}${Unit}`) {
  /* ... */
}

setSpacing("margin-right", "2rem");   // OK
setSpacing("padding-center", "2rem"); // ERROR: "padding-center" invalid
setSpacing("padding-left", "2pt");    // ERROR: "2pt" invalid
                                      // (Unit 'pt' does not exist)
```

Keep in mind that *the keyof* function returns a set of concrete values of strings (*first-name, age*, etc.). However, these are types, not values. (String) values can be changed in JavaScript with a template string. A placeholder can be defined in a template string for this. A similar concept exists in TypeScript at the type level. Listing 17 shows a very small example to illustrate this syntax. The *sayHello* function is passed an arbitrary string. A "hello" greeting is generated from this string and returned. The return type expresses this too. The function does not return the general type string, but the concrete string that corresponds to the returned value.

A more technical example is next to the *sayHello* function. The *setSpacing* function will receive a spacing and a size specification. This is used to express spacing (*padding* or *margin*), as is typical in CSS, which refers to a page (*top, right*, etc.). Additionally, the distance will be passed with a unit. Both arguments can be described with a template literal type. For spacing, two lists of concrete strings (*spacing* and *page*) will be combined. If one or more union types are used in a template literal type, the generated type consists of a combination of all union type expressions (*margin-top, padding-top, margin-left, padding-left*, and so on). For the size, an arbitrary number is specified with the distances. For this, a union type with the allowed spacing (*px, em, rem*) is also defined for the unit. This is combined with the general number type so that all possible numbers can be used. The *setSpacing* function can only be called now with correct values using these type specifications, even if the parameters are "normal" strings from JavaScript's perspective.

With this, the *validate* function's last requirements can now (almost) be mapped. The name of the keys in the rules object can fundamentally be expressed with the type *validate${K}*. However, the capital letter after the prefix validate is still missing. TypeScript provides some auxiliary types for converting notations, such as the type *Capitalize*. This can be used to convert a string type with the same string with a starting capital letter. The correct type definition is type

```
RuleFnName<S extends string> = ` validate${Capitalize<S>} `_
```

The *as* keyword is used to rename a key in a mapped type. This is used to perform a type cast in TypeScript. In this case, the existing type (the name of the key) will be cast to a new type (renamed key): *[K in keyof O as RuleFnName]?: RuleFunction<O[K]>*.

Unfortunately, it isn't possible to call the *ValidateFnName* function here. This type expects a string for the type variable (S extends string). However, the union type returned by keyof can also contain a number or symbol types. There is a trick you can use to exclude keys of these types. You can rewrite the *keyof* expression into an intersection type, which merges the union type with the individual keys (*keyof O*) and the general type *string*. An intersection type contains only the (compatible) intersections of two types. In our case, this filters out all non-strings. The now complete type definition for the *validate* function can be seen in Listing 18.

Listing 18

```
type RuleFnName<S extends string> = `validate${Capitalize<S>}`
type RuleFunction<T> = T extends (
  ...args: any
) => infer R
  ? (value: R) => boolean
  : (value: T) => boolean;

type Rules<O extends object> = {
  [K in keyof O & string as RuleFn-Name<K>]?: RuleFunction<O[K]>;
};

// ValidatedObject and validate // unchanged

const person = {
  firstname: "Klaus",
  salutation() {
    return "Hello, " + this.firstname;
  },
  sallary() {
    return 123456;
  },
}

const rules: Rules<typeof person> = {
  validateFirstname(a) {
    return a.length > 3;
  },
  validateSallary(n) {
    return n > 100;
  }
}


const result = validate(person, rules);
const firstNameValid: boolean = result.firstname; // OK
const sallaryValid: string = result.sallary; // ERR: Type 'boolean' is not
assignable // to type 'string'
const addressValid = result.address; // ERR: Property 'address' does not
exist // on type 'ValidatedObject<...>'
```

Conclusion

TypeScript is more than a "normal" static type system. With its specialized types and concepts like mapped types, conditional types, template literal types, and generics, types can be dynamically created and used to describe complex code structures in a type-safe manner. This allows us to use all the possibilities of JavaScript's dynamic type system.

Using this "metalanguage" is especially worthwhile for code representing an API with many users. They can benefit from type-safe code without having to write type definitions themselves. There are already some ready-made utility types available for some common requirements, as described in the TypeScript documentation.

Nils Hartmann is a software developer and architect from Hamburg. He programs in Java and JavaScript/TypeScript and supports teams with training and workshops in the development of (React) applications.

Interview with Tao Xin, senior staff software engineer at Google

VanJS: "Enabling everyone to build useful UI apps with a few lines of code, anywhere, any time, on any device"

We spoke with Tao Xin, senior staff software engineer at Google to learn all about VanJS, a minimalist reactive UI framework. It enables you to compose the UI tree in a declarative and intuative way. Let's find out more about this minimalist reactive UI framework.

by Tao Xin

devmio: Thank you for taking the time to answer our questions! First, could you please introduce yourself to our readers? What is your background and how did you end up in your line of work?

Tao Xin: Hi, my name is Tao Xin. I am a senior staff software engineer at Google. In the past decade I've been working for various products of Google – ads, cloud, Play, etc. Despite taking more and more leadership role in my career, I keep a keen passion for the programming itself. I'm interested in a wide range of programming-related topics, such as performance optimizations, designing concise declarative API, programming languages, etc.

Mostly, I can be considered as a backend engineer, as the vast majority of my work in my professional life is backend engineering. Thus indeed I don't have too much front-end background. That said, when it comes to computers, I'm a DIY enthusiast. I enjoy building tools for my personal use. But here is the thing: most tools need some sort to UI to be minimally useful. Consider an example, a tool to track your investment portfolio: Of course this can be done by a CLI program. But the way we can interact with CLI program is quite limited. It's not easy even for rendering a simple table view, let alone charts. And the only way we can provide input to the tool is typing some lengthy, hard-to-memorize strings in the commandline.

Thus, it's fair to say, to write some useful program, particularly "useful" in the sense of day-to-day life (for some sparse use case like scientific calculation, CLI program might be ok), you have to build some UI. And to build some UI, you either learn some proprietary UI framework, or you learn some web framework. Either option likely requires you to at least read through a

neither MFC nor Win32 API was easy 20+ years ago 🙁

book to start with. This situation resonates a very remarkable gap I was facing 20+ years ago when I started learning programming: when I had my first computer, I was fascinated by all the powerful things it can do, and all the apps built on top of it. I couldn't wait to learn programming, Pascal first and then C++, hoping to build something interesting and powerful, but only to realize the only possible thing I can build is a popping black screen where I can output some white characters like

```
Hello World.
```

What a disappointment. Indeed, it took me several years to finally come out with something that is not a popping black screen, way after I felt I was pretty skillful at C++ and programming in general.

Today, even after 20+ years with all the technology advancement, I don't think the situation has substantially changed. We see frameworks after frameworks after frameworks. We see a specialized group of people called FE engineers. But for most other people, the apps they are proficient to build are not much different from a popping black screen with white characters. Well, maybe a white screen with black characters, or a dark blue screen with light gray characters. We've seen a great deal of efforts to bring in more modernized terminals and shells. But my thought is: why not going with the other direction instead? Why not making building UI apps as simple as CLI programs so that most of us don't have to be confined to the limitations of terminals? *btw: even for the quest of modernizing terminals, VanJS is pretty good at it - a web-based Unix terminal with notable improvements [1] can be built under 300 lines with the help of VanJS.*

So this is the motivation behind VanJS. In *https://vanjs.org/*, I mentioned that VanJS is positioned as the scripting language for UI [2]. I do think there is the divergence between more and more powerful interactions we can have with computers, and the abilities where most programmers (except for a small group of people specialized in FE or UI) can utilize the powerful UI for useful tools.

devmio: What exactly is VanJS and what does it aim to accomplish?

Tao Xin: VanJS is designed to bridge in the remarkable gap: on one hand, computers has become universally useful, with apps available to almost all day-to-day use cases we can think of; on the other hand, the programs which most people are comfortable to write, are not much different from a popping black screen with white characters. In the home page of https://vanjs.org/, it has VanJS's mission statement: *Enabling everyone to build useful UI apps with a few lines of code, anywhere, any time, on any device.*

In a nutshell, VanJS is a minimalist (world's smallest) reactive UI framework. It enables you to compose the UI tree in a declarative and intuative way. You can easily define states of your apps and bind the states to UI elements, just as you would do with much more complex frameworks. The amazing part of VanJS is: you don't need any tools to use VanJS. Nothing. You don't need to install any dependencies, not even

```
npm
```

You don't even need an IDE. You can just grab-n-go the VanJS library and build powerful UI apps with it in any environment with barely a browser and a text editor. There is no transpiling, no bundling, no extra binary needed for your code to work. What you write is exactly what is being executed in the browser. This is critical to VanJS's mission: ... *build UI apps ..., anywhere, any time, on any device.* You can even build UI

apps completely on your smartphone (or feature phone perhaps?).

devmio: Where did you get the idea to create VanJS? What were your main inspirations or goals?
Tao Xin: This is a great question! VanJS was originated in the process of building various tools on my own. As someone without much background on FE or UI programming, I felt the only viable way for me to build a UI without spending months on learning is to start from scratch with web programming. As more and more apps being built, I felt that there are common problems where a set of helper functions can make the UI code concise, and DRY. There are problems of composing the UI structure declaratively; There are problems of binding UI to states and reacting to state changes. Eventually these helper functions became the early prototype of VanJS. I had a discussion post [3] that talks about the evolution history in the early days of VanJS.

In terms of API design, VanJS draws inspiration from React. As the most widely-used UI framework, React offers a good example on how a mature reactive UI framework should look like: A declarative way to describe the DOM tree structure; Reusable composed UI elements; One-way data-binding from states to UI. The creation of VanJS is an illustration that these can be achieved in a much simpler way. There are 3 things that VanJS has proved:

- **You don't need JSX and transpiling to describe the DOM tree declaratively:** In VanJS, DOM hierarchy can be declared in pure vanilla JavaScript, with even more concise code! The HTML to VanJS [4] Converter can convert any HTML snippet into the equivalent VanJS code where you can do the comparison. We have examples [5, 6] where apps built with VanJS are 3~4 times shorter than the React counterparts.
- **You don't need complex class hierarchy and lifecycle management for reusable components:** In VanJS, a reusable UI component is just a plain JavaScript function, nothing more.
- **You don't need Virtual DOM and hidden diff algorithm for reactive data binding:** VanJS demonstrates that reactive data binding is possible with pure native methods of DOM elements. Thus, there is no need to introduce a new parallel layer of the virtual DOM tree. No need for any hidden control flow.

devmio: Exactly how lightweight is VanJS and what benefits does that offer?
Tao Xin: As of VanJS 1.0.0, the

```
.min.js.gz
```

bundle is only 0.9kB and the

```
.min.js
```

file is 1.6kB. This is 50~100 times smaller than most popular web frameworks. It is by far the smallest reactive UI framework in the world.

Among all of benefits of being such ridiculously small, the first and foremost one is its utmost simplicity. Unlike other frameworks where you typically need reading a book or joining a bootcamp to start with, the tutorial of VanJS, together with the complete API reference, is just a single web page [7], and can be learned within 1 hour. VanJS can enable people without FE background to build a UI app in just a couple hours.

Another important benefit of being such lightweight is performance. For web application, a smaller bundle means faster page loading, and less bandwidth consumption. And because VanJS is just a very thin layer on top of the standard web API (native code implemented by browsers), the code executes fast as well.

Last but not the least, amid the phenomenal rise of Generative AI and LLM (large language model), VanJS's size might find itself an interest spot in this space. With the token limit being the bottleneck of LLM, VanJS's tiny size means the entire library can be used as part of prompt to LLM. I have tried to paste the entire source code to ChatGPT. What I found amazing is that, ChatGPT can not only understand what the code does, it can also generate sample apps on top of it and adjust the code based on follow-up prompts. So think of that, you don't need to train the language model with any knowledge about VanJS. You don't need to fine tune the model. You don't even need to provide any sample code. Just paste the source code of VanJS and ChatGPT will automatically be able to build apps based on it. My conversation history can be found here [8]. I believe there will be very interesting applications by combining LLM and VanJS.

devmio: Could you show us some sample code so we can see how it looks in action?
Tao Xin: Sure.

```
const Counter = () => {
  const counter = van.state(0)
  return span(
    "❤️ ", counter, " ",
    button({onclick: () => ++counter.val}, "👆"),
    button({onclick: () => --counter.val}, "👇"),
  )
}
```

This is a sample app shown in the home page of *https://vanjs.org/*. The app defines a

```
Counter
```

state, and 2 buttons that can respectively increment and decrement the

```
counter
```

As you can see, the DOM tree is built declaratively with pure JavaScript. You don't need JSX and transpiling for declarative UI programming.

The HTML to VanJS Converter [4] can convert any HTML snippet into the equivalent VanJS code. For instance, for

```
<div>
  <p>👋Hello</p>
  <ul>
    <li>🌍World</li>
    <li><a href="https://vanjs.org/">🍦VanJS</a></li>
  </ul>
</div>
```

the equivalent VanJS code will be:

```
div(
  p("👋Hello"),
  ul(
    li("🌍World"),
    li(a({href: "https://vanjs.org/"}, "🍦VanJS")),
  ),
)
```

In *https://vanjs.org/demo* page, there are lots of interesting apps built with VanJS. You can see many handy utilities can indeed built in just a few lines of code with the help of VanJS:

- Stopwatch [9] - 11 lines of code
- Timer [10] - 18 lines of code
- Stargazers [11] - 13 lines of code
- Diff [12] - 23 lines of code
- Epoch Timestamp Converter [13] - 21 lines of code
- Keyboard Event Inspector [14] - 15 lines of code
- package-lock.jsonInspector [15] - 29 lines of code
- Calculator [16] - 38 lines of code, more than 4 times smaller than equivalent React app

devmio: What instances is VanJS best suited for, and where should users potentially hold off for now?

Tao Xin: The prime use cases of VanJS is for someone without much experience of FE or UI programming, but wants some quick solution to add a usable UI into their app. I believe this type of use cases are profoundly underestimated. Lots of CLI programs can be enhanced with a UI layer, which could greatly improve their usability. Taking a personal tool that I recently built - a code generator, as an example, we can have a quick and dirty CLI script for it. But after adding more and more features, it will be crucial to have a convenient way to tweak different types of code generation parameters, to live preview the generated code when parameter changes, and to copy/paste designated piece of code. All of these functionalities are very difficult, if not impossible to implement with CLI. Thus VanJS bridges the gap between CLI programming and GUI programming. You won't feel substantially different between writing a shell script and building a web UI with VanJS. VanJS empowers all programers. Even people with basic programming skills can have the feeling that the day-to-day apps they're using are within their reach to build.

Even for people who would become professional FE programmer, VanJS can provide a smooth learning curve. VanJS offers a very easy start as its tutorial is so easy to walk through. And the programming experience with VanJS is definitely helpful to strength their understanding of reactive UI programming which will eventual benefit their proficiency of much more complex frameworks like React.

The transpiling-free characteristic of VanJS makes it an ideal tool for REPL. All modern browsers have a developer console that allows you to run arbitrary JavaScript code on the fly, but not the code that requires transpiling. With VanJS, you're able to construct any HTML elements with concise code for the current web page right in the developer console. Another related use case is for browser extensions. The lightweight of the library makes it suitable to be embedded in the content scripts that slightly enhance the appearance and behavior of web pages.

Furthermore, VanJS shines in use cases where size is an essential factor, such as ensuring fast page load in low connecitvity network. Integration with LLM is another interesting area to explore. As mentioned earlier,

because its tiny size, we can paste the entire source code of VanJS into ChatGPT and asks ChatGPT to generate application code based on it without the need of any training or fine tuning. This is not possibile with any much larger libraries as they can't fit in the token limit.

Regarding the use cases where users should hold off for now, I think, objectively, VanJS doesn't have the ecosystem compared to the established counterparts. Thus, even though theoretically VanJS is cappable of many types of applications, you should be aware that you don't have a large collection of reusable components readily available. Other than the limitation on the ecosystem, I think VanJS is suitable for quite many use cases, perhaps more broad than some people might have thought. Some people has said, well, VanJS might excel on simple use cases, but for more complex UI components, VanJS's code is not as readable as React's JSX. I respectively disagree with that, as illustrated in the example above, the VanJS-based DOM tree is structurally equivalent to JSX counter part, only more concise (as we can see examples where VanJS apps can be 3~4 times shorter than React counterparts).

devmio: What is Mini Van and what are its use cases?
Tao Xin: Mini-Van is a slimmed-down version of VanJS that only supports DOM tree composition and manipulation (i.e.: without reactivity). Thus compared to VanJS, Mini-Van is even more lightweight. But more importantly, Mini-Van supports SSR (server-side rendering) and hence can be used as a server-side template engine. The entire https://vanjs.org/ website is built with the help of Mini-Van.

Mini-Van is an exploratory step for VanJS to unify server-side and client-side rendering. With Mini-Van, you can write server-side web pages and components similar to the ones on the client-side. There are 2 remaining matters to address to truly unify server-side and client-side rendering?

- What is an effective way to share UI components between server-side and client-side?
- How to enable hydration?

Currently these can be done with some workarounds. In future, I can consider adding the official support if use cases are well understood.

devmio: Are there any future plans from the roadmap that you would like to share? What do you hope to add in an upcoming release, or what is currently in progress?
Tao Xin: That is a great question! First, I would like to lay out the general principles about future evolution of VanJS:

Reliability is the top priority of the VanJS project. Among my limited bandwidth working on this project, reliability is the thing I will give most attention to. I would like to reassure people that despite VanJS being personal project, it's completely reliable to use, even in production settings. Every single code change in VanJS requires passing 400+ test cases, which cover all the supported usage of the library that I can think of, as well as examples illustrated in the tutorial. You can refer to https://vanjs.org/about#reliability for the details.

VanJS is committed to remain minimalist and performant. I will be very cautious of adding new features to VanJS. There are definetely lots of features can be added into VanJS to make it fancier in certain specific use cases. But there will be a high bar to include them in the core library. Notable things to consider regarding new features are: Will this feature restrict how people optimize their applications? Does this feature introduce performance penalty for use cases where the feature is not being used? For the reasons above, certain features are not included in VanJS, such as lazy-evaluated states (as we want to give users the flexibility [17] in scheduling the expensive evaluation), general-purpose diffing (as we don't want to introduce the performance penalty for the majority of use cases where diffing is not needed), etc.

Optional features can be added via add-ons. Relatively, bar can be lower if features are not added into the core library. Use-case-specific libraries on top of VanJS should be the way to address special needs, as unlike the core library, add-ons are completely optional.

VanJS should be easy to work with other libraries and remain unopinionated on which one to choose. We understand there are a good collection of high-quality libraries in JavaScript ecosystem that people enjoy for different kinds of problems, such as CSS, routing, animation, advanced statement management, etc. Instead of providing an in-house solution for each problem in VanJS to be on par with other libraries, we choose to keep VanJS minimalist and unopinionated so that its users can freely combine it with other available solutions. I think this is the beauty of open source community.

With these guiding principles, these are the things planned in the short-term future:

I will build a set of grab-n-go UI components based on VanJS. It's fairly easy to build reusable UI components with the current library but even some basically components might not seem that obvious to the beginners of FE programing, especially for the CSS tricks to enable the basic constructs such as modal, tabs, banner, message box, etc. I believe this will be beneficial for VanJS to further eliminate the entry barrier of FE programing. The work is ongoing - you can check it out here, and we will be implementing more UI components in the near-term future.

I would like to do some benchmarking for VanJS. Although not validated by benchmarking, I strongly believe that the performance of VanJS is much closer to Vanilla JavaScript than to React, given its simplicity in design and performance-aware implementation. Some benchmarking will prove this point and give people the confidence in case they are still unsure whether adopting VanJS is a performance-wise sound decision.

devmio: Besides VanJS, what current technology, programming language, hardware, etc. are you most excited about and would like to see progress in the coming years?

Tao Xin: The recent development on AI (particular around LLM) is defintely exciting. I'm keen on it and looking forward to foundamental innovation coming out in this field.

Other than that, I'm generally interested in programming languages for a long time. I think in general, there are 2 flavors of design philosophy. One flavor is: making things easy to use, and completely hide the complex stuff from its users. To certain extend, JavaScript is designed in this flavor, as the memory is managed by the runtime. The downside of this flavor is users lose the ability to fine tune things. Thus essentially, we're trading the flexibility in exchange for ease of use. What I'm particularly interested in is the second flavor, which is: maximally preserving the flexibilty of its users and at the same time, still trying to make things as easy as it could. Rust is a perfect example in this flavor. It preserves users' flexibility on memory management but on the other hand, achieves the same level of memory safety (maybe higher) as managed languages.

Recently, Zig has captured lots of my interest. Its way of handling compile-time meta-programming is absolutely innovative, and its insistence on no hidden control flow means giving the maximal transparency and flexibility to its users. Particularly, it preserves one flexibility even Rust doesn't offer - memory allocation. We've seen great projects like Bun which leverages this flexibility for its performance optimizations (btw: Bun, which aims to be a much faster JavaScript runtime, is another project that I am very interested in and wish to succeed). Compared to Rust, Zig takes the subtle balance between memory safety and ease of use. It doesn't try to prohibit writing memory-unsafe code, but instead provides good language constructs to make correct code is much easier to write than incorrect one. To certain extent, this is similar to one main design philosophy of VanJS: unlike most other UI frameworks which prohibit the procedural-styled DOM manipulation, VanJS allows direct DOM manipulation and I believe it could be useful in certain use cases, for performance and/or for convenience. Right now Zig is in a very early stage and it is planned to take a few more years to the 1.0 milestone, but this is really something I am wishing for its progress and good adoption.

Tao Xin is a seasonal programmer with decades of programming experience and currently working at Google as a senior staff software engineer. Throughout the years of programming, he has been working on different types of products, and different types of software systems. Tao has a keen interest in a wide-range of programming topics, and has been always pursing the simplicity in design. Meanwhile, he is a DIY enthusiast and likes building many tools to boost his personal productivity. VanJS is a project that comes out of the quest - how to have a simple, zero-dependency toolkit that even people without any FE or UI background can build good tools with a usable UI in a couple hours?

Links & Literature

[1] https://github.com/vanjs-org/van/tree/main/demo/terminal

[2] https://vanjs.org/about#story

[3] https://github.com/vanjs-org/van/discussions/25#discussioncomment-6043597

[4] https://vanjs.org/convert

[5] https://vanjs.org/demo#blog

[6] https://vanjs.org/demo#calculator

[7] https://vanjs.org/tutorial

[8] https://chat.openai.com/share/d92cfaf6-b78e-45ca-a218-069f76fe1b9f

[9] https://vanjs.org/demo#stopwatch

[10] https://vanjs.org/tutorial#state-typed-child

[11] https://vanjs.org/demo#stargazers

[12] https://vanjs.org/demo#diff

[13] https://vanjs.org/demo#epoch-timestamp-converter

[14] https://vanjs.org/demo#keyboard-event-inspector

[15] https://vanjs.org/demo#package-lock-inspector

[16] https://vanjs.org/demo#calculator

[17] https://github.com/vanjs-org/van/discussions/63#discussioncomment-6331983

An alternative to Selenium and Cypress?

End-to-End-Tests With Playwright

The fact that you need tests in a web application is something that we hopefully no longer need to discuss. But what types of tests are useful? How do I set them up? What and how much should I actually test? We will confront these and many more questions once we get into this topic. In this article, we'll look at a tool for E2E testing with TypeScript Playwright.

by Sebastian Springer

Generally, testing functionality always involves a certain amount of extra work, which should also pay off as quickly as possible. This leads us to the crucial question. Who am I testing for anyway? I can only speak for myself, but I hope you're also a laid-back person who doesn't like doing the same tasks several times in a row.

Let's assume we're entrusted with the task of implementing a web application dealing with managing and evaluating books. So, it's a typical CRUD application with a list of records where you can also add new records, modify, and delete existing ones. You can also rate the books. Doesn't sound tragic overall. But every single time you change or extend the application, you have to try out all existing features at least briefly to make sure you haven't accidentally broken something. This gets annoying very quickly. Once you find yourself in a situation where you repeat a uniform activity over and over again, you should consider if it doesn't just make more sense to invest a little more time and automate the process. Checking an application's features in particular has great automation potential.

We aren't rolling out the entire testing story from scratch here. We're focusing on a very specific type of testing: end-to-end testing an application's features. Since this field is still too vast, we'll focus on one particular tool that we'll use to formulate end-to-end testing, or E2E testing for short. We'll look at Playwright and use TypeScript.

Why E2E tests?

Your application lives from the interaction of the individual units. Especially in the frontend, you often work with tree structures of components and add additional functions that can influence your application's behavior. You need automated tests that check interactions between individual units in your application. These tests are usually called E2E tests and they make sure that processes in your application work without errors from one end (the graphical user interface) to the other (the database in the backend).

Unit tests are the simplest type of tests for an application. They are simple because you can assign them directly to a unit of code, usually a function or method. Then the test just needs to check that piece of code. In most cases, you have a defined input that leads to a certain output. Add to that a few boundary and error cases and the code is ready to be checked. In theory, your application's source code only consists of a lot of atomic code units. But if you achieve a 100% test coverage in your unit tests, this doesn't mean that you can be sure everything works as expected. Your application lives from the interaction between individual units. Especially in the frontend, you often work component tree structures and add additiona functions that can influence your application's behavior. You need automated tests that check interaction between individual units in your application. These are usually called E2E tests and should make sure that the processes in your application work without errors from one end (the graphical user interface) to the other (the database in the backend).

If you consider characteristics of different test categories, a clear picture emerges. Your application should have a large number of unit tests and significantly fewer E2E tests. Unit tests run quickly, are relatively easy to implement, and check details in your application. E2E tests have a significantly longer runtime, are more com-

plex to implement, and allow you to draw conclusions about continuous workflows.

As with almost everything in the web world, there's no one standard solution to fall back on when it comes to E2E testing. Instead, you have your choice of several tools, all of which have proven themselves in practice:

- **Selenium:** In a way, Selenium is the bedrock when it comes to E2E tests. The software has been around since 2004 and was originally developed by ThoughtWorks. Selenium provides the WebDriver APIs that help you implement front-end tests with browser automation interfaces. You aren't tied to any particular programming language when implementing your tests. You can use Java, Python, C#, Ruby, JavaScript, Kotlin, or others.
- **Cypress:** Cypress is much younger than Selenium, which saw the light of day in 2017. This framework is written in JavaScript and is based on Node.js. Cypress works in the context of the browser, in the same environment as your application. So it can intervene very deeply in the automation process. Because of its architecture, you write Cypress tests in JavaScript. Since it's basically independent of the system under testing, you can use it to test any languages and frameworks. You can use the Cypress core free of charge. Additionally, the framework's manufacturer offers a paid Cypress Cloud service, which you can better scale your test automation with.
- **Playwright:** Playwright is an open source developed by Microsoft. It lets you run tests in different environments, such as Chromium, Firefox, or WebKit. The tests run both with and without a graphic interface. Playwright itself is written in TypeScript and needs Node.js to run the tests. However, this doesn't mean you're forced to use TypeScript to write your tests. You can also access the Playwright API with Python, .NET, or Java. Similar to Cypress, Playwright is a relatively young project, but has established itself as a permanent fixture in the web community.

Ultimately, there's no right or wrong tool for testing your application. The different platforms have similar feature sets, at least at the core. So you can choose and use the framework that works best for you.

Installation and commissioning

Playwright supports many programming languages. In this article, we'll focus on the testing framework's TypeScript variant. Like all testing frameworks, it's available as an npm package. This means you can install it in your application with any JavaScript package manager. Besides the pure source code that makes it possible to run tests, the Playwright package includes a command line script you use to work with it. It lets you run tests and can help you create them. If you install Playwright in your project with the npm install playwright command, you still need to put some effort into configuring your test environment. But using the initializer is the better option. This feature is also given to you by npm, Yarn, and pnpm. For npm, you run the npm init playwright@latest command in your application. The command will install Playwright and create the necessary structures in the project, so you can start testing right away. The setup process is interactive. This means that Playwright asks you a series of questions on the console and the answers directly affect your project. The first question is where you want to place your E2E tests. The default answer is tests. No matter what name you choose, Playwright creates the directory and places the first sample test right away. Next to this directory, you'll also find another directory called tests-examples. Here, you'll find a whole series of sample tests in another test file, which you can use as a template for your own tests.

The second question requires you to decide if you want Playwright to generate a GitHub actions workflow. The default here is NO. As you can see, Playwright has integration with CI/CD pipelines firmly in mind. It's obvious that a tool developed by Microsoft also supports Microsoft infrastructure out-of-the-box. However, you don't have to worry if your project relies on a different infrastructure. Playwright is basically independent of the infrastructure used.

The third question tells Playwright if you want to install browsers. Playwright can run tests on one or multiple browsers. By default, Playwright installs Chromium, Firefox and WebKit for you. Alternatively, you can post-install the browsers individually on the command line. For example, you can use npx playwright install chromium to post-install Chromium later. If you omit the browser name, Playwright installs all of the default browsers: Chromium, Firefox and WebKit. With npx playwright install-deps, you can also post-install the system components needed for the browser as well as the browser.

```
> library@0.0.0 test:e2e
> playwright test


Running 6 tests using 5 workers
  6 passed (3.5s)

To open last HTML report run:

  npx playwright show-report
```

Fig. 1: Initial test result

Playwright's setup process creates the playwright.config.ts file in your application's root directory. It contains the configuration for your E2E tests. Playwright also creates the tests and tests-examples directories for you. The tests directory contains a file with two simple sample tests. The tests-examples directory contains a more comprehensive E2E test that shows you how to write tests. It uses a to-do application as an example.

With the initial Playwright configuration, you can use the npx playwright test command to run all tests. The npx command is part of the Node.js platform and searches the system for the specified script. If it isn't available, npx downloads the package and runs the script. However, after you install Playwright in your application, the first case takes effect. It gets a little more convenient if you define the E2E command as an npm script in your package.json file. Listing 1 shows an example.

Using this script, you can run your tests with the command *npm run test:e2e*. While you barely save any characters, you don't need a framework-specific command. This can be helpful when switching between applications that have different frameworks. You can also add additional options to the script without having to explicitly specify them every time you call it. Now, if you run your tests, you'll get an output like the one seen in **Figure 1**.

Playwright ran a total of six tests using five worker processes. All tests were successful and took a total of 3.5 seconds. The number of tests is because there are a total of two tests implemented in the example.spec.ts file in the tests directory and they run on three browsers. You can set how Playwright handles parallelization in playwright.config.ts. The most important settings here are fully-parallel and workers. You can use the npx playwright show-report command to display the HTML report generated for the test run.

Listing 1: Playwright Tests as npm script

```
{
  ...
  "scripts": {
    ...
    "test:e2e": "playwright test",
    "test:debug": "playwright test --project=chromium --debug"
  }
}
```

The first test

First, the good news. If you already have experience with JavaScript testing frameworks like Jasmine, Jest, or Mocha, you won't have a difficult time getting started with Playwright. The TypeScript variant is based on existing structures in the JavaScript testing world. You create a new test with a call to the test function. This anticipates a short description of the test case as the first argument and a callback function containing the actual test. Usually, you can implement this callback function as an async function, since Playwright heavily relies on premises. Tests are much nicer to read with async-await. You can group your tests (calls to the test function) into test suites. You can do this with the test.describe method, which accepts a description and a callback function. You can use the test.beforeAll, test.beforeEach, test.afterEach, and test.afterAll methods to define functions that are executed before all tests, before each test, after each test, and after all tests, respectively. Typically, you use these set-up and tear-down routines to prepare your test environment or clean up after a test run, or to dump commonalities from multiple tests and make your tests more compact.

The book management application has a list of records. The first test case is to check the list display. For this, assume the backend provides three records.

In the example shown in Listing 2, you can see several approaches to locating elements on the current page and checking them. But first, use the page object's goto method to navigate to the appropriate page in your application. In Playwright, it doesn't matter if this is a classic multi-page application or a single-page application with a modern JavaScript framework. The method returns a Promise object that you can wait for with await.

Listing 2: Playwright Test

```
import { expect, test } from "@playwright/test";

test.describe('List', () => {
  test('render', async ({ page }) => {
    await page.goto('http://localhost:5173/list');

    const headline = page.getByRole('heading');
    await expect(headline).toHaveText('Bücherliste');

    const titles = page.locator('table > tbody > tr > td:nth-child(2)');
    await expect(titles).toHaveCount(3);
    await expect(titles).toHaveText(["The Hitchhiker's Guide to the
                                      Galaxy", '1984', 'The Lord of the Rings']);

    const years = page.getByTestId('year');
    await expect(years).toHaveText(['1979', '1949', '1954'])
  })
})
```

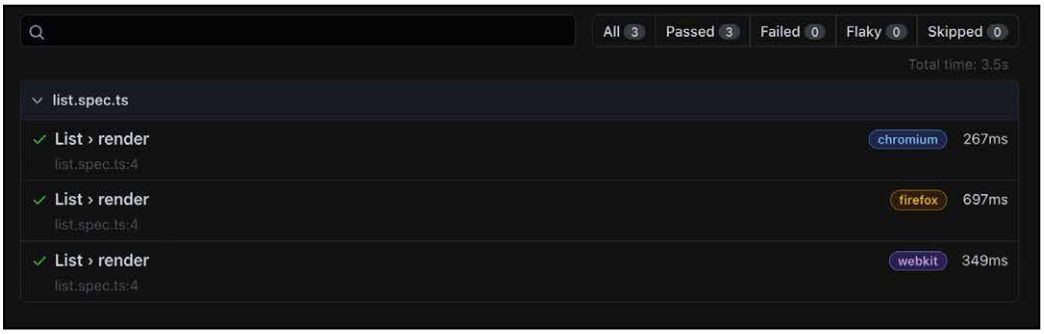


Fig. 2: Successful test report

On the page, first check that the heading has the correct text content. You can access this element using the heading role. This kind of localization is one of the most elegant solutions since it combines an application check with a rudimentary accessibility test. Playwright works with the proven expect syntax that Jasmine and Jest also use. To expect, you pass a locator object, like the one you get from the getByRole method. Then, you can use the matcher functions to formulate your condition. For the heading, you expect the h1 element to contain the text booklist, which you can specify with the toHaveText matcher.

Besides the getByRole locator, Playwright has many more helper functions for locating elements. One is the locator method, you can pass a selector to. This is a string similar to what you use in CSS or JavaScript to locate elements. Be careful not to define overly complex selectors, since they are error-prone. Don't rely on the page structure, since it can change when new features are integrated, causing your test to fail.

Another locator method is *getByTestId*, which you may know already if you've ever worked with the testing-library. This method finds elements based on the value of the data-testid attribute. You can insert this attribute into your page's structure to make it easier to find elements. Since it's a custom attribute, there's no risk of causing an overlap with a standard attribute in the browser.

When checking list items, you'll notice that you can check individual items as well as groups of items. While searching for the title of a book, you'll get three results. With the *toHaveCount* matcher, you can make sure that precisely this number is displayed. If you work with a single element, you can specify the expected text content directly. If you expect more than one element, pass an array with the corresponding values.

On a successful test run, the Playwright script exits to the console and you receive a short success message. If at least one test fails, Playwright delivers the test report to you locally via a web server so you can directly view the results in detail. A successful report looks like **Figure 2**.

Working with the backend

Normally, web applications consist of a front-end and a back-end. You use Playwright to interact with the front-end and use it to test your application. While you can also run API tests with Playwright, this is the exception. When testing, you have the option of testing either your entire application (front-end and back-end) or just the front-end in isolation.

If you're testing the frontend and backend together, you need to make sure that the backend has a defined state for each test run. The easiest way to do this is to virtualize your backend. This way it can be started up in one or more containers and populated with suitable test data. In a pipeline run, first, you start up the back end and then you run the tests. If you have different E2E tests, the state of the database can become a problem. One requirement for tests is that they generally must not build on each other. However, if you run the tests one after the other, a test might create or modify records. You must either undo these operations manually in the clean-up routine, or make sure that the environment and the database are cleaned up in the set-up routine before each test. How you handle this and what additional constructs you will need depends on your environment. However, you should make sure that a test run doesn't require manual intervention under any circumstances.

If you only want to check the front-end, then the situation changes. One reason for this can be that the front-end and back-end are developed independently and the

back-end is already secured with extensive API tests. In this instance, you can replace the back-end with a mock and create a fake back-end that you can control for your tests. For this, Playwright provides the Mock APIs feature. Here, the framework intercepts outgoing requests and answers them with a predefined response.

List implementation in the application catches three different cases. If the backend returns no data, the frontend displays appropriate information. If data is available, it is displayed. If an error occurs during communication to the backend, or if the backend reports an error, then the application displays an error message. The test in Listing 3 checks these three cases using the mock APIs.

You can use the mock APIs with the *page.route* method. Here, you use a string or regular expression to specify which requests should be answered. You have access to the route object in the asynchronous callback function. You can use the fulfill method to formulate the response. If you use the json property, as shown in the example, Playwright ensures that your browser's fetch API receives a valid JSON response. In the rest of the first test, verify that the record is displayed and that neither the empty result message nor an error message is present. Unlike the first test, the mock backend returns only one record, so you can clearly distinguish the result.

In the second test, have the mock backend respond with an empty array and expect the message that will be displayed. You should also check that no data or error message is displayed.

In the final test, you should have the server respond with an error. For this, use the status code 500. This stands for an internal server error. Note that you must use the fulfill method here too. Playwright doesn't provide reject or similar methods. You can only simulate an error with the status code, which you can specify using the status property. Optionally, you can define a response body with the body property. In this test, you can also check if the error is displayed correctly. The other expect calls make sure that no other information is displayed in addition to the error.

Simulating interaction - forms

So far, you've only seen how to perform static checks. The situation becomes much more interesting when you start interacting with your application. In a web application, you have a wide variety of interaction options. These range from simple clicks, to filling in text fields, to uploading files. For these and more, Playwright provides methods on the Locator objects that you can create with getByTestid. In the test shown in Listing 4, you create a new record using a form and verify that the record was created correctly. You could cover the previous tests with ordinary unit tests. But this test checks two features: the create form and the book list.

Listing 3: Using the Mock APIs

```
import { expect, test } from "@playwright/test";

test.describe('List', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('http://localhost:5173/list');
  });

  test('fetch data from Server', async ({ page }) => {
    await page.route(/books/, async route => {
      const json = [{
        "ISBN": "978-3-86680-192-9",
        "title": "The Hitchhiker's Guide to the Galaxy",
        "author": "Douglas Adams",
        "price": 9.99,
        "pages": 224,
        "year": 1979
      }];
      await route.fulfill({ json })
    });

    const titles = page.getByTestId('title');
    await expect(titles).toHaveCount(1);
    await expect(titles).toHaveText("The Hitchhiker's Guide to the Galaxy")

    await expect(page.getByTestId('no-data')).not.toBeVisible();
    await expect(page.getByTestId('error')).not.toBeVisible();
  });

  test('fetch empty data from Server', async ({ page }) => {
    await page.route(/books/, async route => {
      await route.fulfill({ json: [] })
    });
    await expect(page.getByTestId('no-data')).toHaveText('Es sind keine
Datensätze vorhanden');

    const titles = page.getByTestId('title');
    await expect(titles).toHaveCount(0);
    await expect(page.getByTestId('error')).not.toBeVisible();
  });

  test('fetch with error', async ({ page }) => {
    await page.route(/books/, async route => {
      await route.fulfill({ status: 500, body: 'Internal Server Error' });
    });
    await expect(page.getByTestId('error')).toHaveText('An error has
occured');

    const titles = page.getByTestId('title');
    await expect(titles).toHaveCount(0);
    await expect(page.getByTestId('no-data')).not.toBeVisible()
  });
});
```

In the first step, you open the form via the / create path and use the fill method to make entries in the various text fields. Locate these using the respective labels. Then, use the SUBMIT button to submit the form. The form sends a POST request to the server to create the record and redirects the browser to the list view. Here, you can verify that the new record is displayed correctly by searching for each piece of information with the getByText method and checking that each piece of information exists only once.

Outsource auxiliary functions to Page Object Models

E2E tests can become quite extensive, especially when it comes to more comprehensive workflows. Here, it's proven useful to outsource blocks from the actual test code to separate support functions. You can group these support functions into a Page Object Model. In Playwright, you can implement Page Object Models in the form of TypeScript classes. To prevent this class from degenerating into a confusing collection of helper functions, you should collect helper functions of one page or view of your application in one class, if possible. For better reusability between tests, you can also parameterize the helper functions. Listing 5 shows an example of a Page Object Model.

The Page Object Model represents the form page of the application. It simplifies switching to and operating the form. In the constructor, you can define the locator objects that you need in the course of the auxiliary functions. Concrete methods of the class include the goto method that Playwright uses to navigate to the page, the fillForm method to fill in the form, and the submitForm method to submit the form. For the fillForm method, this method is parameterized and given a default value. This lets you control filling the form as needed. With the Page Object Model, the actual test is much more compact, as seen in Listing 6.

In tests using a Page Object Model, instantiate the page object model and pass the reference to the page. Then you can call the helper functions.

Preparing the test environment with fixtures

The concept of fixtures goes in a similar direction. Here, you are trying to keep your tests' source code as tidy and focused as possible. One requirement of any automated test, whether unit or E2E, is that they should be able to run independently of other tests. Therefore, it's very important that each test finds a clean environment and

Listing 4: Forms verification

```
import { expect, test } from "@playwright/test";

test('create a new book', async ({ page }) => {
  await page.goto('http://localhost:5173/create');

  await page.getByLabel('ISBN:').fill('TestISBN');
  await page.getByLabel('Title:').fill('TestTitle');
  await page.getByLabel('Author:').fill('TestAuthor');
  await page.getByLabel('Price:').fill('TestPrice');
  await page.getByLabel('Pages:').fill('TestPages');
  await page.getByLabel('PublicationYear:').fill('TestYear');
  await page.getByRole('button').click();

  await expect(page).toHaveURL(/.*list/);

  await expect(page.getByText('TestISBN')).toHaveCount(1);
  await expect(page.getByText('TestTitle')).toHaveCount(1);
  await expect(page.getByText('TestAuthor')).toHaveCount(1);
  await expect(page.getByText('TestPrice')).toHaveCount(1);
  await expect(page.getByText('TestPages')).toHaveCount(1);
  await expect(page.getByText('TestYear')).toHaveCount(1);
});
```

Listing 5: Definition of a Page Object Models

```
import { Locator, Page } from "@playwright/test";

export class FormPage {
  private ISBN: Locator;
  private title: Locator;
  private author: Locator;
  private price: Locator;
  private pages: Locator;
  private year: Locator;
  private submitButton: Locator;

  constructor(private readonly page: Page) {
    this.ISBN = page.getByLabel('ISBN:');
    this.title = page.getByLabel('Title:');
    this.author = page.getByLabel('Author:');
    this.price = page.getByLabel('Price:');
    this.pages = page.getByLabel('Pages:');
    this.year = page.getByLabel('PublicationYear:');
    this.submitButton = page.getByRole('button');
  }

  async goto() {
    await this.page.goto('http://localhost:5173/create');
  }

  async fillForm(data = { ISBN: 'TestISBN', title: 'TestTitle', author:
      'TestAuthor', price: 'TestPrice', pages: 'TestPages', year: 'TestYear' }) {
    await this.ISBN.fill(data.ISBN);
    await this.title.fill(data.title);
    await this.author.fill(data.author);
    await this.price.fill(data.price);
    await this.pages.fill(data.pages);
    await this.year.fill(data.year);
  }

  async submitForm() {
    await this.submitButton.click();
  }
}
```

leaves the test environment as clean as possible. This avoids the need for a specific call chain of tests. If one test in a chain fails, there's a high probability that all other tests in the chain will fail too. To solve this problem, Playwright provides the Fixtures feature, which lets you prepare your tests.

For the following example, we assume there is a regular backend for tests and it does not have any data yet. Listing 7 shows an example of a fixture for rendering the list display.

To create a fixture, use the extend method of the Playwright test object. You pass an object to it, defining listPage as a method. In the method implementation, you can instantiate your Page Object Model and perform preparatory work like creating records. Include the Page Object Model with the use function. Then, you can perform cleanup routines such as deleting previously created records.

The advantage of using the fixture is that you can extract the preparation code from your test and focus on the workflow that will be checked in the test. You can also use this fixture for multiple tests, reducing duplicates in the code.

Debugging - when things go wrong

If you run your tests headless, you have limited debugging capabilities. But you can change that locally by running your tests with the --debug option. If you select a specific browser with --project, you can easily debug. The --debug option makes Playwright automatically open its Inspector, giving you more tools for debugging.

Another handy tool is the VSCode extension for Playwright. This tool lets you run your Playwright tests directly from VSCode. You can see the test results directly in the development environment without having to switch to the command line first. Another feature from this extension is the ability to debug tests directly in the development environment.

When a test fails, Playwright can also take a screenshot of the state of your application. This can be helpful when looking for a bug.

How does Playwright perform?

Playwright is a very flexible test framework for end-to-end tests in the browser. It doesn't need to hide from its competitors. It's nice that the entry barrier is very low. You can start developing tests right away without much prior knowledge. If you need special tools to test a feature of your application, Playwright usually provides them. One example are locator methods like getByTestId or getByRole. If these aren't enough, you can also use selectors directly to find your elements. For many other use cases like authentication, Playwright either provides concrete tools or at least the documentation guides you step-by-step on how to solve the problem.

This makes Playwright a real alternative to Selenium or Cypress when it comes to automated E2E verification of applications.

Sebastian Springer is a JavaScript developer at MaibornWolff in Munich and is primarily concerned with the architecture of client- and server-side JavaScript. He is a consultant and lecturer for JavaScript and regularly shares his knowledge at national and international conferences.

Listing 6: Integrating the Page Object Model

```
import { expect, test } from "@playwright/test";
import { FormPage } from "./form.po.ts";

test('create a new book', async ({ page }) => {
  const formPage = new FormPage(page);

  await formPage.goto();
  await formPage.fillForm();
  await formPage.submitForm();

  await expect(page).toHaveURL(/.*list/);

  await expect(page.getByText('TestISBN')).toHaveCount(1);
  await expect(page.getByText('TestTitle')).toHaveCount(1);
  await expect(page.getByText('TestAuthor')).toHaveCount(1);
  await expect(page.getByText('TestPrice')).toHaveCount(1);
  await expect(page.getByText('TestPages')).toHaveCount(1);
  await expect(page.getByText('TestYear')).toHaveCount(1);
});
```

Listing 7: Definition and using fixtures

```
import { expect, test as base } from "@playwright/test";
import { ListPage } from "./list.po.ts";
import { FormPage } from "./form.po.ts";

const test = base.extend<{ listPage: ListPage }>({
  listPage: async ({ page }, use) => {
    const listPage = new ListPage(page);
    const formPage = new FormPage(page);
    await formPage.goto();
    await formPage.fillForm();
    await formPage.submitForm();
    await use(listPage);
    await listPage.remove();
  }
})

test.describe('List', () => {
  test('should render the list', async ({ listPage }) => {
    await listPage.goto();

    await expect(listPage.headline).toHaveText('Bücherliste');
    await expect(listPage.titles).toHaveText('TestTitel');
    await expect(listPage.error).not.toBeVisible();
    await expect(listPage.noData).not.toBeVisible();
  });
});
```