Universal Compatibility of Framework Agnostic Web Components

Philipp Schmutterer

BACHELORARBEIT

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Januar 2023

Advisor:

Ing. Thomas Karl Fischl BSc MSc

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, January 30, 2023

Philipp Schmutterer

Contents

Abstract

In modern frontend web development, most developers use frameworks like Angular, React or Vue. On a basic level, all of these frameworks use so called web components: bundles of HTML, CSS and Typescript which are independent of the rest of the code and each represented by their own custom HTML tag. The idea is to pack chunks of code which are frequently used throughout a project (i.e. Product cards in a webshop) into reusable components. This reduces the amount of code needed, eliminates the necessety for copy-pasting code segements throughout a web page and promotes "dry" code. However, since not every project will be built using the same framework and all these frameworks use components as their basic building blocks, it would be usefull to be able to build framework agnostic web components that can then be used by any framework. This way, components would only need to be created once instead of creating them in each framework separately. The subject of this Thesis is therefore to answer the question of whether it is possible to create framework agnostic web components and use them in multiple frameworks without the need for considerable amounts of extra code and to shed some light on how this universal compatibility is achieved. As an example, StencilJS will be used to create web components and the frameworks Angular, React and Vue will be used to demonstrate the universal compatibility. Apart from small example pages that contain Components created in StencilJs, a part of this Thesis is also going to be a real-world project; A tablet app for Therapists that displays essential patient data and helps with managing patients.

Chapter 1

Introduction

In der Modernen Web-Frontendentwicklung verwenden die meisten Entwickler Frameworks, wie zum Beispiel Angular, React, oder Vue. Grundsätzlich verwenden diese Frameworks alle sogenannte Web-Komponenten: voneinander unabhängige Blöcke von HTML, CSS und Typescript, welche einen eigenen HTML-tag bekommen. Die Idee dahinter ist es, Codeblöcke, die mehrmals in einem Projekt verwendet werden (z.B. Produktkarten in einem Webshop) in wiederverwendbaren Komponenten zusammenzufassen. Das reduziert die Menge an Code in einem Projekt und erleichtert das Schreiben von „dry“ Code. Da man Allerdings nicht jedes Projekt mit dem selben Framework umsetzen wird, und alle diese Frameworks Komponenten als grundlegende Bausteine verwenden, wäre es nützlich, Frameworkagnostische Komopnenten erstellen zu können, welche dann in jedem beliebigen Framework verwendet werden könnten. Dadurch müsste man Web-Komponenten nur einmal erstellen, anstatt für jedes Framework einzeln. Das Ziel dieser Arbeit ist es daher, festzustellen, ob es möglich ist, Frameworkagnostische Webkomponenten zu erstellen und diese in verschiedenen Frameworks ohne viel Boiler Plate Code zu verwenden.Als Beispiel wird Stenciljs verwendet, um Komponenten zu erstellen und die Frameworks Angular, React und Vue werden dienen Dazu, die Universale Verwendbarkeit der Komponenten zu demonstrieren. Neben einigen Testseiten, die die in Stenciljs erstellten Componenten beinhalten wird auch ein Projekt Teil dieser Arbeit sein. Es geht dabei um eine Tablet-app für Therapeuten, welche Essenzielle Patientendaten schnell und einfach zugänglich macht und das Patientenmanagement unterstützt.

Chapter 2

Fundamentals and Related Works

For context it is important to elaborate some basics about web components and the frameworks they are used in. This means going into the basic architecture of each framework, comparing the structure of their web components and those created in StencilJs.//

2.1 Web Components

At a basic level, a web component is a JavaScript file that defines an encapsulated piece of HTML, CSS and JavaScript code that can be interpreted by a web browser and treated as an HTML element like <p> or . This basic form of a web component does not depend on any framework and can be imported either in JavaScript using an import command or in HTML using a <script> tag. In this Thesis the terms "web component" and "component" will be used synonymously.

2.2 Angular

Angular is a framework for frontend web development and was created by Google in 2010. It is described in the official documentation as a Typescript based platform that includes a framework to build web applications, as well as many helpful libraries and tools to streamline the entire process of developing and maintaining a web application [1]. This means that Angular is not just a framework, but also has a large number of tools around the framework itself.

2.2.1 Components in Angular

While basic web components in JavaScript are complicated to implement, Angular components use TypeScript and are divided into multiple files for more structure:

- A Typescript file for the component's class (i.e. myComp.component.ts)
- An HTML file for the visual representation (i.e. myComp.component.HTML)
- an SCSS file for styling (i.e. myComp.component.scss)
- a spec.ts (Typescript) file for testing purposes (i.e. myComp.spec.ts)

These files are linked together using the metadata given in the component.ts file.

2.2.2 Basic Concepts

Services

A service is a part of a component that defines a specific behavior or functionality and is written exclusively in Typescript. Services can be injected into components to provide functionality. This helps to make the code more modular and reusable.

NgModules

A module in Angular represents a collection of components and services that share a certain context.

Decorators

An angular component is implemented as a Typescript class which can contain decorators with a certain type. A decorator tells the angular compiler how to use the following code (for example @NgComponent to tell the compiler that the following class is a component).

Metadata

as mentioned above, metadata tells the compiler what to do with a certain piece of code. To give a more specific example, the @NgComponent decorator's metadata contains the location of the component's HTML and CSS files. This way all files can be linked to a single component.

Templates

Templates are an enhancement of HTML featured in Angular that allows a developer to inline some functionality like hiding UI-Elements, which would normally take several additional lines of Typescript and CSS. It works by placing HTML code in a <template> tag. The UI element(s) can then be altered using event binding.

Event Binding

Event binding is a way of responding to DOM events inside the HTML code. A good example for this is the click event. By using it inside an HTML element like this:

```
<button (Click)="onClick()">Do something</button>
```

the button will call the onClick function in the Typescript class of the component when it is clicked. The data flow of event binding goes only from the HTML to the typescript class or from child component to parent component.

Property Binding

Similar to event binding, but in this case the data flow is reversed (from HTML to Typescript or from parent to child). Data is passed as a property via HTML. The properties must be defined in the Typescript class. For example, text can be passed to a component in order to be displayed with the following code:

HTML element:

```
<my-text [text]="myText"></my-text>
```

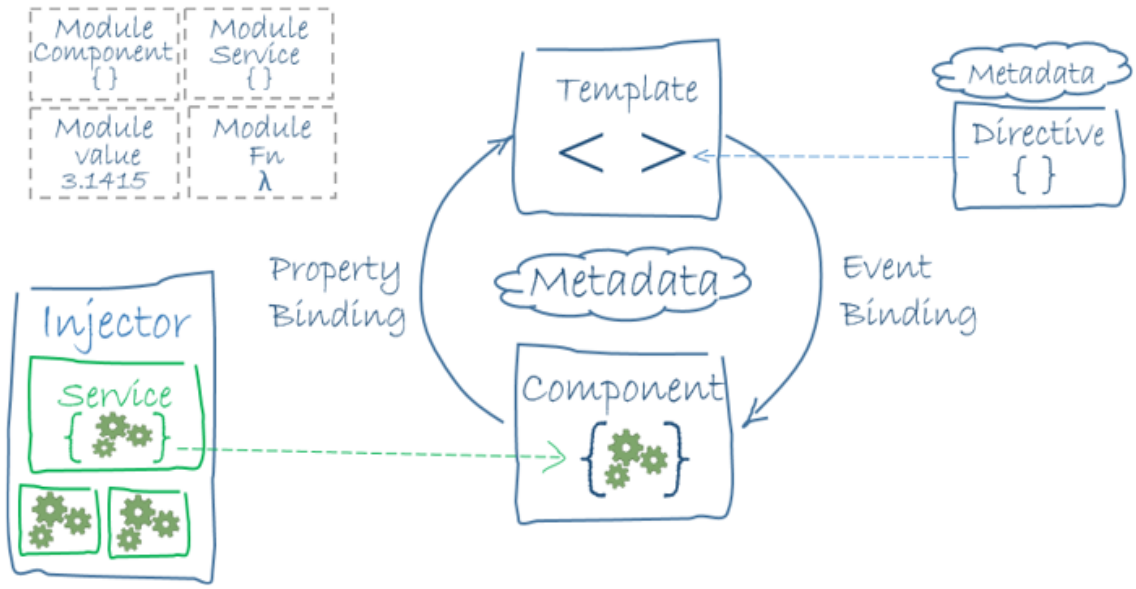
Inside the class:

```
myText:string;
```

Directives

Directives are also Typescript classes that are defined with the @Directive decorator and can be attached to DOM elements in order to apply a certain behavior like changing the background color of a button while it is clicked. Directives are modular and can be used multiple times.

The graph below describes how these concepts work together.



Example of an Angular Component:

Here is an Example of the essential files of an Angular component:

component.html:

```
<h1>{{text}}</h1>
```

component.ts:

```
@NgComponent({
  selector:'my-text',
  templateUrl: './my-text.component.html',
  styleUrls: ['./my-text.component.css']
})

  export class myText{
  myText:string
}
```

The @NgComponent decorator tells the compiler that the following class belongs to a component. The decorator takes an object as a parameter which provides essential metadata for the component. The selector determines the name with which the component can be used in an HTML file, the templateUrl contains the path of the component.html file and the styleUrls is an array in which the paths to all the CSS files belonging to the component. The following class containes the variable that is used to pass text to the h1 element in the component.html file.

2.3 React

React is an open source Library for building user interfaces released by Facebook in 2013 to make the development of their social platform easier. React is not a fully fledged framework, because it does not dictate any design system or even provide an eco-system or built in features. Much of the functionality of a React application must be drawn from third party libraries.

2.3.1 Components in React

In Angular technologies are artificially separated by having a file for HTML and TypeScript for a component. React however separates concerns, meaning that what belongs together is stored together resulting in components that consist of a single file that contains the TypeScript code as well as the CSS styling and the HTML code of a component. To achieve this, React uses JSX or TSX, depending on whether the project is built in JavaScript or TypeScript. In this thesis TSX will be used. Furthermore, in React, components can be implemented as either class components or functional components. Before React 16.8 a class component would be required for state management. Since update 16.8 however the difference between functional and class components is only syntactical and it is recommended to use functional components going forward. In

contrary to Angular, React components are not updated, but created as new instances.

In the following examples the syntactical difference between class and functional components will be demonstrated. In both cases the HTML will look like this:

```
<MyText input="Hello, World!"/>
```

Example of a functional Component:

```
function MyText(props:myTextProps){
  return (<h1>{myTextProps.input}</h1>);
}
```

Example of a Class Component:

```
class MyText extends React.Component{
  render(){
    return(<h1>{this.props.input}</h1>)
  }
}
```

Note that there is no props object explicitely passed to the class. This is because the props object is auto generated from the HTML code.

2.3.2 Basic Concepts

TSX

Without TSX every HTML element has to be created by calling the React.createElement function or as an object. TSX is an abstraction of TypeScript that enhances the syntax and enables the TypeScript compiler to parse HTML code inside a TypeScript file into calls of the React.createElement function. Thus Typescript and HTML code can be stored in the same file. In TSX, a React component has a method called “render” that contains the HTML code to be rendered.

Example for React.createElement call:

```
const element = React.createElement(
  h1,
  {className: 'greeting'},
  'Hello World!'
);
```

Example for element as object:

```
const element = {
  type: 'h1',
  props: {
    classname: 'greeting',
    children: 'Hello, World!'
  }
};
```

Example of render function in TSX:

```
render() {
  return (
    <h1>Hello, World!</h1>
  )
}
```

States

In React, a component maintains its own data using a state. A state is different to props in that the data is private and managed only by the component itself. By giving the component control over its own data it can rerender itself and does no longer need to be reinstantiated every time the data changes. Since React 16.8 the use of a state no longer requires a class component with a constructor, essentially making class components obsolete. An example will be given in the hook section

The count variable is used as the state here. The Array defines the variable and a function that can be called to change the value of count. The useState hook (more on hooks below)is used to add the state to the component and initialize it. When the button is clicked, it will use the setCount function to update the state, which will in turn cause the component to rerender with the new state.

Hooks

Hooks were introduced in React 16.8 and are JavaScript functions that are used to handle state data in components. They extract the stateful logic from components and make it reusable. The basic Hooks are:

- **useState():** is used to update a state, triggering a rerendering of the component. This example demonstrates the use of useState:

```
function Counter(){
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

- **useEffect():** useEffect always runs when any stateful data changes. This behavior can be controlled more accurately by providing the hook with an array of dependencies in which the exact states that the hook should watch for changes are specified. If useEffect is called with no dependency array, it will run when the component is mounted into the UI and when the state changes. An empty dependency array will cause the hook to only run once when the component is initialized since there are no states being watched. If the array has dependencies only those states will be watched for changes and the hook will not necessarily run upon mounting the component.
  This example shows the same component as before with a second state variable and a useEffect hook:

```
function Counter(){
  const [count, setCount] = useState<number>(0);
  const [loaded, setLoaded] = useState<boolean>(false);

  useEffect(
    ()=>{
      fetch('foo').then(()=>setLoaded(true))
    },
    [count] // dependency array
  )
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

- **useContext():** The useContext hook allows for stateful data to be shared or scoped using React's Context API. This means that a variable can be used by multiple disconnected components and if there is a change all the components that use the variable are updated. To do this, a variable or set of variables must be declared in the top most parent component (i.e. the App component if the data is to be used everywhere). Any child component inside the resulting component tree will inherit this variable without the need to pass it via property. By calling the useContext hook the data can be accessed and should it change, the changes will be applied to all child components automatically. The following example demonstrates the use of useContext.

  In App.js:

```
const fruits = { //data for context
  monday: 'Apple',
  tuesday: 'strawberry'
}
//creating context
const FruitContext fruits = createContext(fruits)

function App (props) {
  return (
    //the context is shared to all components inside the
    FruitContext tag.
    <FruitContext.Provider value = {fruit.monday}>
      <todays-fruit/>
    </FruitContext.Provider>
  );
}
```

  In todays-fruit component

```
function todays-fruit () {
  const todaysFruit = useContext(FruitContext)
  return <p>{todaysFruit}</p>
}
```

Other Concepts

Services, decorators and metadata have already been covered in the Angular section and are not much different in React.

2.4 Vue

Vue is a very simple frontend web framework created by ex-Google employee Evan You in 2014 with the goal of building a custom tool around the best parts of Angular. Vue

is designed for creating smaller projects and can be used to create entire applications or just single widgets to be used in other websites.

2.4.1 Copmonents in Vue

Vue uses single file copmonents, meaning that all the code of a component is stored in a single file. While styling can be done externally in both Angular and React, this is not the case in Vue.

The structure of a Vue component is much simpler than in React as it is divided into three HTML tags: template, script and style. The template tag contains the HTML code of the component while the script tag contains any JavaScript or TypeScript and the style tag contains the CSS styling.

Example of a simple Vue component

```
<template>
  <h1>Hello, World!</h1>
</template>

<script lang="ts">
  import { defineComponent } from 'vue';

  export default defineComponent({
    name: 'hello-world',
  });
</script>

<style scoped>
  h1{color: green;}
</style>
```

2.4.2 Basic Concepts

Props

Properties of a component are declared in the script tag as either an array or an object named “props”. If an array is used it contains only the names of the properties as strings. these properties do not have a fixed type and cannot be configured any further. By using an object it is possible to give the property a type and configure it (i.e. make it a required property among other things). In the object the properties are stored as nested objects that contain the property’s configuration.
Example of Properties in a Vue component:

```
.
.
.
```

```
<script lang="ts">
  import { defineComponent } from 'vue';

  export default defineComponent({
    name: 'HelloWorld',
    props: {
      msg: {
        type: string,
        required: true
      }
    }
  });
</script>
.
.
.
```

Property binding

The Vue equivalent of Angular's property binding is the v-bind directive. On a surface level, the difference is only syntactical. By putting `"v-bind:"` before a property of an HTML element, variables can be assigned to the property.
Example of the v-bind directive:

```
<hello-world v-bind:msg="{{message}}"\>
```

Registration

When the Vue compiler comes across a component inside a template tag it needs to know where to find the corresponding component file in order to render the component. Therefore a component must always be registered in its parent component. This is done via an object in a component's script tag that contains the names of all the components used in the component's template. Components can be registered either globally by using the app.component function or locally using the approach described above.

2.5 Stencil

Stencil is an open source library created by the ionic framework team specificly to create web components that could be used in Angular, React and Vue using a single code base as ionic supports all three of these frameworks and provides a library of components out of the box that need to run in each framework. As mentioned, the purpose of Stencil is only to create component libraries and not full websites or apps. There is an Index.html file in a Stencil project, but it is only meant for testing. Therefore Stencil is not a framework. The most important part of Stencil are two features. The first being it's compiler which creates framework agnostic web components, meaning pure JavaScript files and the second being the "output targets" feature which can produce framework native components.

2.5.1 Components in Stencil

Components in Stencil are written in TSX (more on TSX in the React chapter) with a separate CSS file. The structure of the component itself is very simple. The component is just a TypeScript class with a component decorator which holds metadata. Inside the Class most of a component's functionality that is not a private function uses a decorator.

2.5.2 Basic Concepts

Most Common Decorators

In Stencil most functionalities are declared via a decorator. The most common ones are:

- **@Component:** The following class is a component. **Example:**

```
@Component({
  tag: 'hello-world',
  styleUrl: 'hello-world.CSS'
})
```

- **@Method:** The following function is a public Method.
  Example:

```
@Method()
toString():string {
  return 'Hello, World!';
}
```

- **@Prop:** Declares a property of the component that can be used for passing data to the component via HTML.
- **@Event:** Creates an event emitter that takes data as a parameter and can emit DOM events.
  Example:

```
        @Event buttonClicked: EventEmitter<string>

        onButtonClick(string s) {
                buttonClicked.emit(s);
        }
```

- **@Listen** Creates an EventListener that is triggered by a specified event and executes a given function. **Example:**

```
@Listen('HelloButtonClicked')
helloButtonClickHandler(event: CustomEvent<string>) {
  console.log('Hello, World!')
}
```

- **@Watch:** Stencil's equivalent of useEffect. The Watch decorator reacts to changes of states or props and executes a function that is implemented immediately after it. The Watch decorator also takes the old value as well as the new value of the watched data as a parameter. **Example:**

```
@State() displayText: string;

@Watch('displayText')
watchStateHandler(oldValue:string, newValue:string){
  console.log('the old value was: ', oldValue);
  console.log('the new value is: ', newValue);
}

onButtonClick(string s) {
  buttonClicked.emit(s);
}
```

Chapter 3

Solution

The solution to the problem will be demonstrated with an example page using a selection of demo components that will test the functionality of all the concepts described in the Fundamentals and Related Works chapter.

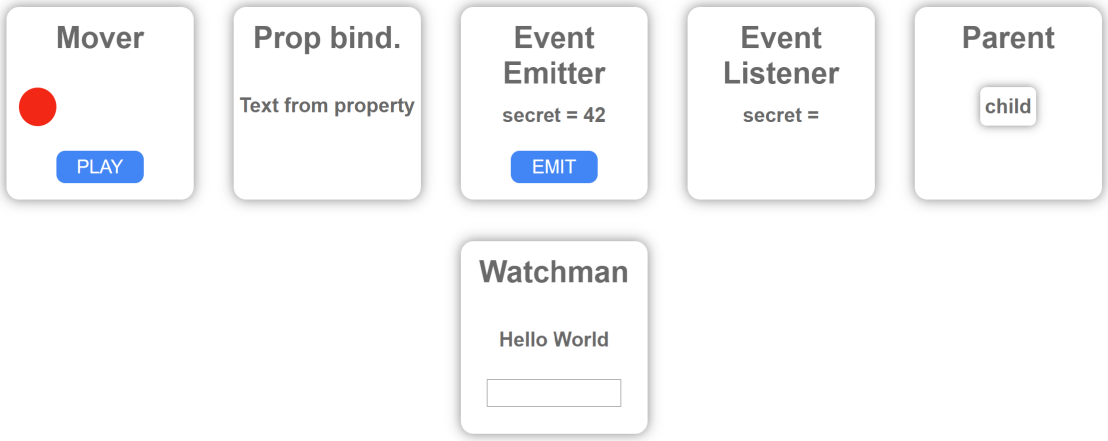
3.1 Compiling Stencil Components

Stencil offers two ways of compiling its components into a format that can be used by other frameworks. The first way is to compile a component into a pure JavaScript file. The second way is to use the Stencil compiler's output target feature which can compile a Stencil component into a framework native component.

3.1.1 Pure JavaScript

The easy and less time consuming way of compiling a Stencil component is to compile it into a standardized JavaScript web component. These component is then ready for any browser to interpret. However if used in a framework, a small amount of boiler plate code is necessary in each of the frameworks.

The picture below shows the demo components that will be used as an example.



About these components:

- **Mover:** A bare bones hello world component that plays a little CSS animation to test whether components and styling are displayed properly.
- **Property binding:** This component uses a prop decorator to receive text via HTML to test the functionality of property binding.
- **Event Emitter:** This component has a secret value in its class and uses an event decorator to emit a DOM event containing this secret which can be processed by other components.
- **Event Listener:** Uses a listen decorator to catch the event emitted by the event emitter component and display the secret in its own template.
- **Parent and Child:** The Parent component contains a child component which is displayed in its template to test component nesting.
- **Watchman:** The watchman component uses a watch decorator to watch its state, a string variable, for changes. Through user input via textbox this state is changed with every keystroke and the displayed text will change along with the state. This component tests the event binding and its equivalents in React and Vue.

A component library created in Stencil can be compiled using the console command `npm run build`.
This will compile all the components in the project into JavaScript and SCSS files

Chapter 4

Figures, Tables, Source Code

Chapter 5

Using Literature and other Resources

[1]

Chapter 6

Printing the Manuscript

Chapter 7

Closing Remarks

Appendix A

Technical Details

Appendix B

Supplementary Materials

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format).

B.1 PDF Files

Path: /

thesis.pdf . . . . . Master/Bachelor thesis (complete document)

B.2 Media Files

Path: /media

*.ai, *.pdf . . . . . Adobe Illustrator files
*.jpg, *.png . . . . . raster images
*.mp3 . . . . . audio files
*.mp4 . . . . . video files

B.3 Online Sources (PDF Captures)

Path: /online-sources

Reliquienschrein-Wikipedia.pdf

Appendix C

Questionnaire

Appendix D

LaTeX Source Code

References

Literature

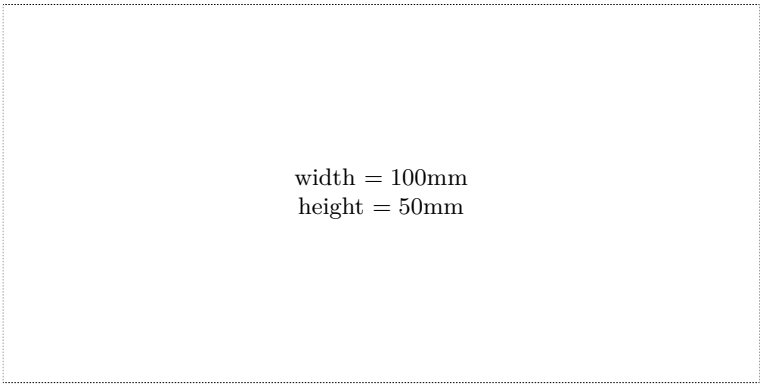
[1] Hubert M. Drake, Milton D. McLaughlin, and Harold R. Goodman. *Results obtained during accelerated transonic tests of the Bell XS-1 airplane in flights to a MACH number of 0.92*. Tech. rep. NACA-RM-L8A05A. Edwards, CA: NASA Dryden Flight Research Center, Jan. 1948. URL: https://www.nasa.gov/centers/dryden/pdf/87528main_RM-L8A05A.pdf (cit. on p. 17).

Online sources

[2] *Reliquienschrein*. Aug. 29, 2022. URL: https://de.wikipedia.org/wiki/Reliquienschrein (visited on 01/13/2023).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —