code
tutors

VueJS

VueJS-techniques for implementing

SPA-applications

Getting started

- Create directory c:\course
- Open command prompt and

```
> cd \course
> git clone https://github.com/tutorit/vue241202 trainer
> git clone https://github.com/tutorit/vue241202 mywork
> cd mywork\server
> npm i
> node server.js
```

- Material is available at cloned directory material-folder

- So you cloned the same repository twice
  - Idea is that you only work at mywork-folder
  - The instructor pushes his samples back to the repository
    - And you can always check the latest samples by running

    ```
    > git pull
    ```

    - at trainer-folder

Topics

Modern SPA-applications
- SPA-models
- MVC-variations
- Component-centric UI

VueJS architecture
- Overview of VueJS-application
- Features of VueJS
- Declarative rendering
- Extensions and helpers
- Programming models

Basic use
- Application-instance
- Template syntax
- Data binding
- Using inputs
- Handling events
- Basics of components

Filters
- Built-in filters
- Custom filters

Removed from v3

Directives
- Conditional directives
- Looping
- Other directives
- Custom directives

Components
- Implementing components
- Props and state
- Component hierarchies
- Mixins
- Special cases

Navigation
- Using routing
- Vue-router
- Router parameters
- Nested routing

State management
- Using RESTful interface
- Separation of concerns
- Designing the datamodel
- Pinia

Security of SPA-application

VueJS

Architecture and features

Background

VueJS is a JavaScript Framework for building SPA-applications

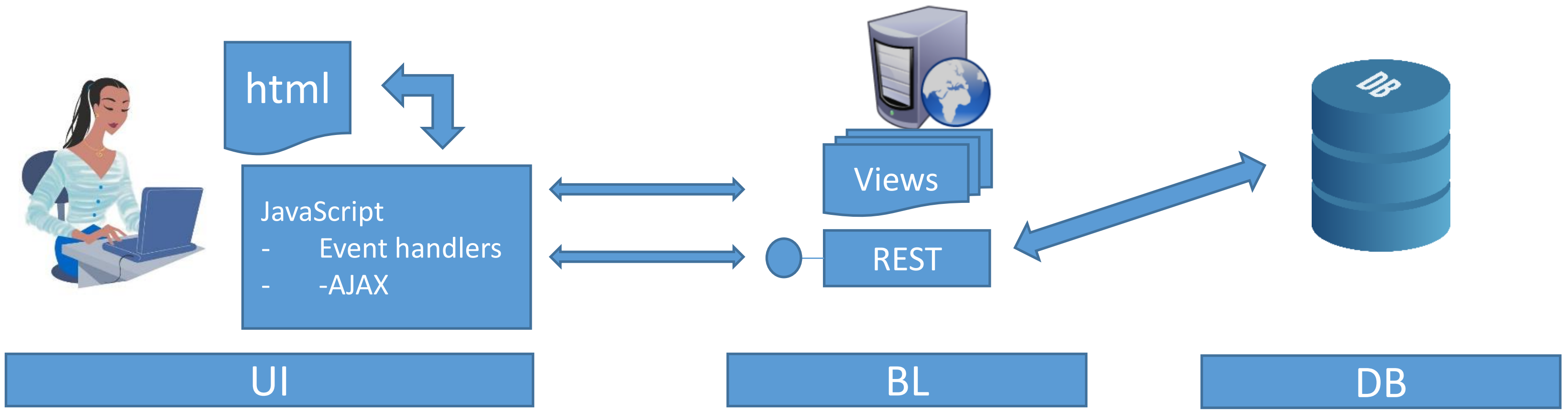
- A challenger to Angular and React
- Borrowing ideas from AngularJS, but also to some extend React
  - Lightweight
  - Easy to get started with
- Originally released in 2014
  - Now at version 3.5.y (9/2024)
  - Some breaking differences between versions 2 and 3

What is SPA

- Single Page Application
  - A single html-page is loaded to the browser
  - JavaScript handles events and modifies UI accordingly
  - JavaScript may also load data from the RESTful services implemented to serve with some AJAX-library

- Big megatrend of web application development today
  - Improved user experience
  - Better scalability
  - More straightforward application architecture
  - Libraries supporting SPA have evolved greatly, most traditional problems are automatically tackled

Single Page Applications



- Application is built with html, css and JavaScript
- JavaScript handles events caused by user actions
  - Loads and updates data with AJAX
  - Changes views
  - Manipulates UI

- Web-server hosts the application
  - Html-page
  - Images
- Views are served by web-server
  - Html-fragments
  - Forms
  - Listings
- Service interface to data
  - Data validations
  - Security

- Data-storage

How do you build your UI

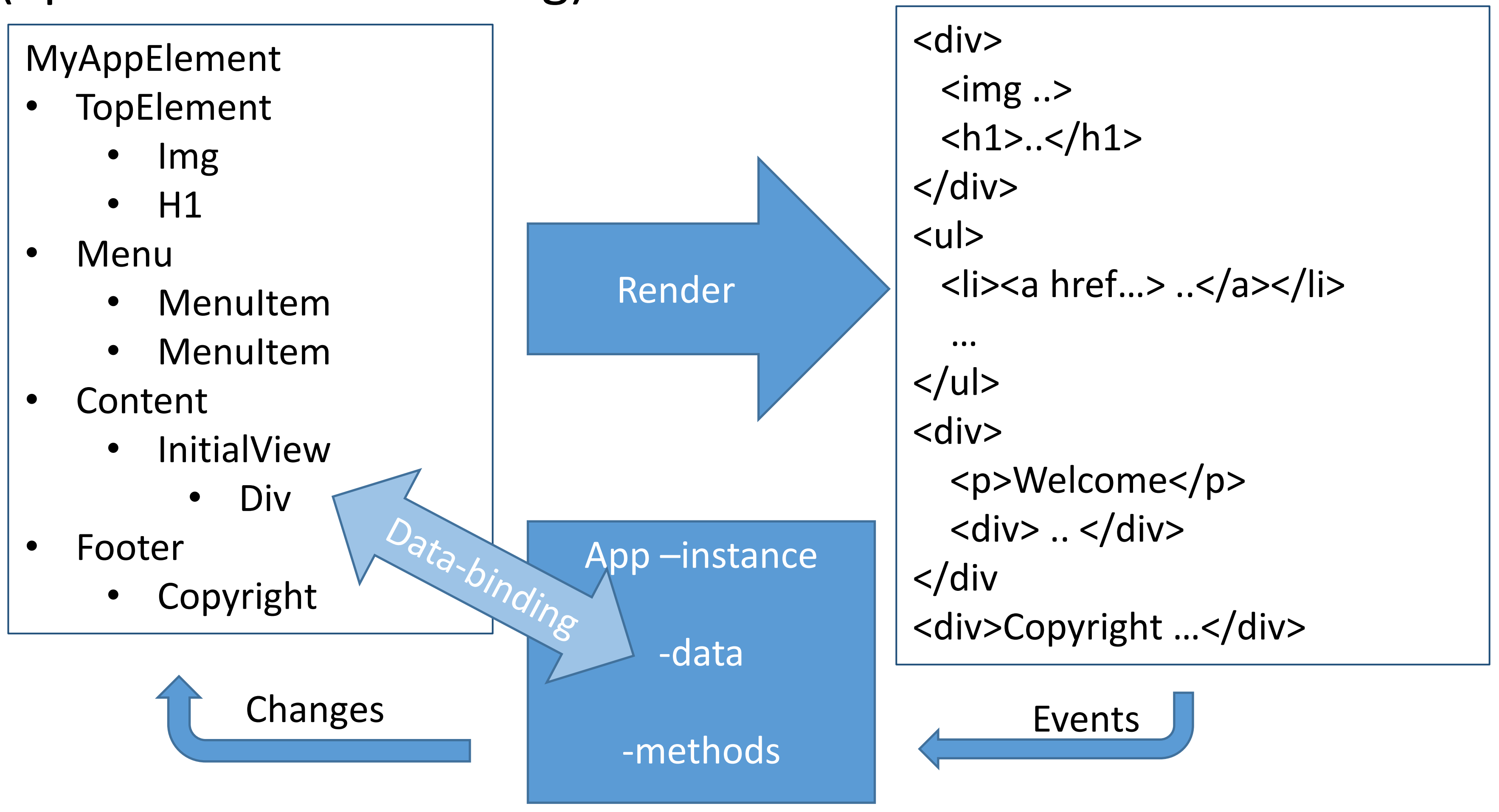
UI is traditionally built from reusable UI-components

- A component may be very simple
    - Display data
    - Allow editing with some intelligence
    - Display list
    - Etc
- Component may be a container
    - Container holds other components
    - Adds intelligence to behavior of the component group
- Container may be a full window
    - Form with components and intelligence for as specific use-case

This is basically also the approach used with VueJS

VueJS Architecture

Idea of virtual DOM is similar to that of React (apart from databinding)



VueJS -features

- VueJS is an UI-library
  - UI-templating (View of MVVM)
    - Raw html as basis
    - Extended with custom elements, Components
    - Control of styles for elements and css effects
    - Directives extend vocabulary as extra attributes
  - Define data model for UI (ViewModel of MVVM)
    - Properties, calculated properties, watchers
    - Event handlers to manipulate data
    - Props to initialize data
- Vue Router add-on
  - Navigate between pages
- Pinia add-on
  - Manage application state
  - Replaces VueX

Plain HTML, no build

- Vue can be used directly on HTML-page
  - To add functionality to the page
  - Like JQuery or any other JavaScript library
  - As contrary to building and application of several components
- For the first examples we use this method
  - Easy and quick to familiarize you with basic concepts and notations
- You just need to load the required js-file

```
<head>
    <script type="importmap">
        {
          "imports": {
            "vue": "/libs/vue.esm-browser.js"
          }
        }
    </script>
</head>
```

Hello world

```
<body>
    <h1>Vue-demonstration</h1>
   <div id="vuecontent">
        <p>{{message}}</p>
        <input v-model="message" />
   </div>
   <script type="module">
      import { createApp } from 'vue’;
      let app=createApp({
         data() {
            return {
               message: 'Hello Vue!'
            }
         }
      });
      app.mount('#vuecontent’);
   </script>
</body>
```

Identify element on page

Text-interpolation

Data-binding with v-model directive

Create App-instance

Properties on data-model are available on page

Exercises

- Study wwwroot/hello.html
  - There is also hello2.html, do not worry about that just yet
- Work with wwwroot/calculator.html
  - Display the sum of figures entered to the two input fields

Components

- Components are reusable UI-elements
- At simplest, they are just objects defining the template and possibly the data for the component

```
const Hello={
    template:`<div><input v-model="message" />{{message}}</div>`,
    data(){
        return {message:"Hello, Vue!"}
    }
}
```

- Now the application can be constructed of the definition above
    - So actually createApp takes the component descriptor object as parameter

```
import { createApp } from 'vue'
const app=createApp(Hello);
app.mount('#vuecontent');
```

App.component

- Api can also be used to create component

```
import { createApp } from 'vue'
const app=createApp({});
app.component("Hello",{
    template:`<div><input v-model="message" />{{message}}</div>`,
    data(){
        return {message:"Hello, Vue!"}
    }
})
app.mount('#vuecontent');
```

- And now the component name can be used as element within the rendered application

```
<div id="vuecontent">
    <Hello />
</div>
```

Exercise

- Study wwwroot/hello-component.html
- Work with calculator-component.html
- Create calc-component that is displayed on the page
  - Just create object that describes calculator and use it to create the application

- Extra
  - Make a copy of calculator-component.html and work with it
  - Use app.component-method to create the actual component and display the calc-element on the page

Computed properties

- Data-model may hold properties whose values are calculated based on other information
  - Either from model or elsewhere
- Add computed-property to the model

Computed properties are defined as functions that return the value for the property
OR
As an object having get and set methods as for Object.defineProperty

```
data: {
    greeting:'Hello',
    name: 'Vue'
},
computed:{
    message:function(){
        return this.greeting+“ "+this.name++“!";
}
```

```
<div id="vuecontent">
    <p>{{message}}</p>
</div>
```

Watchers

- Watcher associated with property is called when property-value changes
  - Watch-property in the model
  - Function that takes the new value and the old value as parameters
- Use to
  - Implement validators against data
  - Update properties asynchronously

```
data: {
    message: 'Hello'
},

watch:{
    message:function(newValue,oldValue){
        console.log("Watch",newValue,oldValue);
        this.message=newValue.split(' ')[0];
    }
},
```

Only allow one word in message, space character is ignored…

Exercise

- Add “result” as computed property to your calculator
- Add “calculation” property to data, “1 + 2 = 3”
    - Add a watch that changes the property when data for calculation changes
    - Display the property on template
    - Note that this also could be implemented (even more straightforwardly) with a computed property

Two programming models, API styles

- Options-API
  - Used in previous examples
  - “Traditional” way of using Vue
    - Vue concepts (application, component, directive) are created using options-object
- Composition-API
  - Now you can study hello2.html
  - Setup method is used to describe aspects of application, component or directive
  - The API is more “functional” style
  - Especially suitable for Single File Components of bigger application
  - We’ll mostly be working with this API-style from now on

```
import {createApp, ref,computed} from 'vue'
const app=createApp({
    setup() {
      const fig1=ref(1);
      const fig2=ref(2);
      const result=computed(() => fig1.value+fig2.value);
      return {
        fig1,
        fig2,
        result
      }
    }
});
```

Earlier calculator application with Composition API

Vue-project

Building an application from single file components

Single File Components (SFC)

- Working with component template string will become tedious in bigger projects
  - It would be easier if the template would be in a html-file
- Single file-components of Vue allow you to combine script, css and html-based template required for the component to a single file
  - But these must be compiled before they can be distributed to the browser

Hello.vue, composition API

```
<script setup>
    import {ref} from 'vue';
    const message=ref("Hello, Vue!");
</script>
<template>
    <p>{{message}}</p>
</template>
<style scoped>
p{
    color:red;
}
</style>
```

Hello.vue, options API

```
<script>
export default {
    data(){
        return {
            message:"Hello,Vue!"
        }
    }
}
</script>
<template>
    <p>{{message}}</p>
</template>
<style scoped>
p{
    color:red;
}
</style>
```

Walkthrough, create project

- Current preference is to use Vite
  - npm create vue@latest
    - Name book-app, no to all other questions
  - cd book-app
  - npm i
  - npm run dev
- Open book-app folder with your editor
  - Study folder contents briefly
  - Simplify the template at App.vue

If you are using Visual Studio Code, you should install Official Vue extension

After npm run dev:

Press
h for help,
o to open application in browser

```
<template>
    <div>
        <hello-world msg="I did it" />
    </div>
</template>
```

Props

- Props are “parameters” passed to a component instance
  - They must be described for component
    - At least name, possibly type and even validator
- The container gives them as attributes to the component element

```
<script setup>
// Props can be defined as an array of strings
defineProps(["greeting","target"])
</script>

<script setup>
// Or for more complex cases, as an object
defineProps({
  greeting: {
    type: String,
    required: true
  },
  target:{
      type:String,
      validator:value => value.length>2
  }
})

</script>
```

```
<hello greeting="Hi" target=“You" />
```

Exercise

- Calculator once more, but now as SFC
- Create Calculator.vue to components-directory
- Add
  - <script setup>…</script>
  - <template>…</template>
- You should have fig1 and fig2 refs and the calculated property result
- Template should display
  - input fields with bindings to fig1 and fig2
  - Result with interpolation

Excercise

- Create CalculatorContainer.vue
- It should just display the calculator

- Also pass optional props fig1 and fig2 to the calculator
  - To actually pass a number you need to prefix the attribute name with a colon

```
<calculator :fig1="17" />
```

Slightly deeper

Lifecycle hooks

Template syntax

Data binding

Using inputs

Handling events

The lifecycle hooks

- The options-object may hold declarations for lifecycle methods
- If composition API is used the lifecycle method must be imported from ‘vue’ and the hook function is passed as parameter to that
  - Lifecycle-methods ar prefixed with ‘on’: onMounted etc

- Hooks
  - beforeCreate – created : The instance is (being) created, not mounted to the dom yet
  - beforeMount – mounted: The instance is (being) mounted to the dom
  - beforeUpdate – updated: The updates are (being) rendered to the dom
  - activated – deactivated: Kept-alive component is activated/deactivated
  - beforeDestroy – destroyed: Component instance is (being) destroyed
  - errorCaptured: Kind of a “catch” for errors occurring in descendants

```
import {onActivated,onMounted,onUpdated} from 'vue';
onActivated(() => console.log("Component activated"));
onMounted(() => console.log("Component mounted"));
onUpdated(() => console.log("Component updated"));
```

Exercise

- Exeriment with some of the Lifecycle hooks
  - Just use console.log to display which ones are executed

Template Syntax

- We have already seen
  - Text interpolation with {{ dataInsertedIntoContent }}
  - And v-model -directive used for data binding with input-elements
- We can also use directives (they always appear as attributes)
  - v-once, element is rendered just once, further updates spipped
  - v-pre, contents of the element are not compiled
  - v-show, conditionally show element
  - v-html and v-text, set innerHtml or innerText
  - v-bind:attribute (:attribute for short)
  - v-on:event (@event for short)

Data binding

- Both the mustache-notation {{}} and v-bind do the databinding
  - Mustache-notation used for content
  - v-bind used in attributes
- For both the value given may be a singe variable or a JavaScript expression that is evaluated
  - Evaluated value is used as content or attribute value
- If v-once directive is used the databinding is only done once upon initialization of the element
  - Affects all the bindings for that element
- If raw html needs to be inserted into contents, it cannot be done with mustache-notation
  - v-html –directive must be used instead

Event handling

- Event handlers are implemented with v-on:event –directive
  - Often the shorthand @event is used
  - Where event is the name of the event (click, change, blur…)
- The handler may be defined
  - By giving the name of function implemented into the vm’s method-property
  - By calling a function declared for component by giving explicit parameter
    - $event in template refers to the original dom-event
- Modifiers may be added: v:on:event.modifier[.modifier]
  - .stop : stopPropagation
  - .prevent : preventDefault
  - .capture : catch the event before children
  - .self : do not process events targeting child elements
  - .once : for components only
  - .passive : As addEventListener - passive

Keyboard events

- Very often we want to process keyboard events only if a specific key was pressed
  - Vue makes this easy through the use of modifiers v-on:keyup.65
- Vue defines aliases for common keys
  - .enter
  - .tab
  - .esc
  - .space
  - .up , .down, .left and .right
- Key names from the standard KeyboardEvent.Key can be used
  - When translated to “Kebab”-case
    - PageDown becomes page-down

Exercise

- Create another version of the Calculator called EventCalculator
  - Add a button to the UI and only calculate the result when button is clicked

- How about calculating when Enter is pressed?

- Show also this calculator on CalculatorContainer

Components

Component hierarchy

Events

Data binding

Mixins and Composables

Special cases

Components

- Components are reusable pieces of UI
  - Declared as Single File Components of by App.component-function
- Each component instance has its own data
  - State of component
  - Items created with ref-function
- Component instances form a hierarchy

```
const inst=getCurrentInstance();
console.log("Component itself, parent and app“,
    inst,inst.parent,inst.root);
```

- The container may pass initialization data to its children
  - Props seen as attributes on the template
- The container may handle the events signaled by its children

Props

- Props are the attributes that are passed to the component instance
- Props must be defined for the component with props-property in the descriptor
  - Array of strings naming the possible props
  - Object where property name identifies the prop and property value identifies the type (by constructor-function, not string)

```
Vue.component("simple-1",{
    data(){
        return {
            greeting:this.initial || "Hello"
        }
    },
    props:["initial"],
    template:"<p>{{greeting}}</p>"
});


Vue.component("simple-2",{
    props:["initial"],
     template:`<p v-on:click="initial='And even changed'">{{this.initial}}</p>`
});



Vue.component("simple-3",{
    props:{ initial:String },
    template:"<p>{{initial}}</p>"
});
```

```
Vue.component("simple-4",{
    props:{
        initial:{
            type:String,
            required:true,
            validator: value => value.length > 10
        }
    },
    template:"<p>{{initial}}</p>"
});
```

<simple-X initial=Pass props as attributes' />

Container and children

Child can

- Signal changes with an event
  - $emit on template
  - `const emit=defineEmits(["resultChange"]);  // at setup you may define emit function`
- Allow databinding with model
- Display content given by container with <slot> -element

```
app.component("child",{
    props:["item","name"],
    model:{
        prop:"item",
        event:"change"
    },
    template:`<div v-on:click="$emit('change','Changed by '+name)">
                <p>{{name}} {{item}}</p>
                <slot></slot>
            </div>`
});


app.component("container",{
    data:function(){return {someValue:'Value from parent'}},
    template:`<div>
        <child name="First child" :item="someValue" v-on:change="v => someValue=v"/>
        <child name="Second child" v-model="someValue" />
        <child name="Third child" v-model="someValue">
            <p><em>Click any of the paragraphs above</em></p>
        </child>
        <p>My value: {{someValue}}</p>
        </div>`
});
```

Exercise

- CalculatorContainer should already display both calculators
- In the EventCalculator
  - Signal “resultChange” when the button is clicked
  - Display the result on CalculatorContainer
- In the original calculator
  - Emit a string that describes the calculation “1+2=3”
  - You might want to implement a watcher for the result

```
watch(result,(newValue,oldValue) => {
    console.log("Watcher",newValue,oldValue)
})
```

Extra exercise

- If you are quick….

- Create yet another calculator: ObjectCalculator
- Modify CalculatorContainer to pass v-model that is an object to the calculator
- The ObjectCalculator will receive modelValue-prop…

```
<script setup>
import {ref} from 'vue'
import Calculator from ‘./ObjectCalculator.vue'

const myCalc=ref({
    fig1:3,
    fig2:4
});

</script>
<template>
<div>
    <object-calculator v-model="myCalc" />
    <p>{{myCalc.fig1}}+{{myCalc.fig2}}</p>
</div>
</template>
```

This is the container

v-for

- The directive v-for is used to repeat an element for each item in a collection
  - The v-bind:key must be specified with a binding to a unique identifier in the data item

```
app.component("for-component",{
    data:function(){ return {
        cars:[
            {id:1,make:"Volvo",model:"V40"},
            {id:2,make:"Toyota",model:"Auris"}
        ],
        selectedId:0
    }},
    template:`<div><select v-model="selectedId">
                <option value="0">Please select</option>
                <option v-for="car in cars" :value="car.id“
                            :key="car.id">
                    {{car.model}}</option>
            </select>
            <p>SelectedId: {{selectedId}}</p></div>`
});
```

v-if

- V-if –directive is used to conditionally render an element
- Using v-if for an element that has v-for is not recommended
  - Though possible

```
app.component("if-component",{
    data:function(){
        return {showIt:true}
    },
    template:`<div>
                <input type="checkbox" v-model="showIt" />
                Show it
                <p v-if="showIt">Here it is</p>
            </div>`
});
```

Exercise

- Create BookList –component that displays holds an array of books in the data
  - Take a peek at server/bookdao.js
  - Display the books in a table: id, title and author columns are enough to start with
- Create Main-component that sets the page-layout and shows the BookList
  - This should be rendered by the Vue-instance

```
<div>
        <header>
            <h1>The app header</h1>
        </header>
        <main>
            <book-list />
        </main>
        <footer>
           Some footer content
        </footer>
</div>
```

Filters

- Filters are functions that m
  - Pipe the data through a filt
  - May take parameters

- Can be registered
  - Globally by Vue.filter –func
  - Locally to one component

This feature is no longer available at version 3

If you are updating an older project to version 3 you need to replace filters most likely with computed properties

```
Vue.filter("upper",function(str){
    return str.toUpperCase();
})

Vue.filter("left",function(str,len){
    return str.substring(0,len);
})

Vue.component("filter-component",{
    filters:{
        year:function(dt){
            return dt.getFullYear();
        }
    },
    template:`<div>
                <p>{{"Hello world" | upper }}</p>
                <p>{{"Hello world" | left(5) }}</p>
                <p>{{new Date() | year}}</p>
            </div>`

});
```

Exercise

- Show price of the book with two decimals and currency sign
- Show published date formatted nicely “4.3.1922”

- Replace headers for columns title and author with input fields
  - Try filtering the table contents: title must contain what is entered into title-input, author must contain what is entered into author-input
- Replace header for id with a combobox with options Title and Author
  - Selection change should change the sort order of the books in the table

Styles

- Most of the styles for the application should of course be declared in global css-files
- It is possible to declare component specific styles in the .vue-files with scoped-option (<style scoped>)
- And you can do data-binding agains style- and class properties of the element

```
app.component("style-component",{
    data:function(){ return {
        colorClass:'ok',
        emStyle:'bold'
    }},
    methods:{
        swapClass() { this.colorClass=this.colorClass=="ok" ? 'bad' : 'ok' },
        swapStyle() {this.emStyle=this.emStyle=='bold' ? 'normal' : 'bold'}
    },
    template:`<div>
             <p :class="colorClass" @click="swapClass">Using css-class</p>
             <p :style="{fontWeight:emStyle}" @click="swapStyle">and style</p>
         </div>`
});
```

Transitions and animations

- Componts may contain transition element
  - That automatically assigns specific css classes to the element when hiding/showing with v-if or v-show
- Transition flows through states
  - css may be applied to different states
  - JavaScript-hook may be applied to different states

```
app.component("transition-component",{
    data(){ return{
        state:'',
        showBig:true
    }},
    template:`<div>
        <p @click="showBig=!showBig">Transition state: {{showBig}} {{state}}</p>
            <transition name="shrink"
                    v-on:enter='state="enter"'
                    v-on:after-enter='state="afterEnter"'
                    v-on:leave='state="leave"'
                    v-on:after-leave='state="afterLeave"'>

                    <p v-if="showBig">Transition element</p>
            </transition>
        </div>`
});
```

Exercise

- Declare .tooSmall and .ratherBig css-classes for the BookList
- Display high prices with ratherBig-class and low prices with tooSmall class
- Use style binding to display low prices in red and high prices in green

```
<style scoped>

.tooSmall{
    font-weight: bold;
}

.ratherBig{
    font-style:italic;
}

</style>
```

Mixins

Mixins are Vue-way of inheritance

- Describe an object that holds items that are common to several components
    - Attach the object to the component descriptor with mixins-property
- There are algorithms to solve problems with overlapping properties within mixins but try to avoid situation

```
let sampleMixin={
    data:function(){
        return { mixinData:'Hello' }
    },
    methods:{
        log(s){
            alert(this.$vnode.tag,s);
        }
    },
    filters:{
        upper: str => str.toUpperCase()
    }
}

Vue.component("mixin-component",{
    name:'MixinComponent',
    mixins:[sampleMixin],
    template:`<div>
                  <p @click="log('clicked')">{{mixinData | upper}} world!</p>
              </div>`

});
```

Version 2 example

At version 3 the preferred way to do this is to use Composables

Composables

- Reusable logic that different components may need should go to composables
- Essentially functions that return an object of reusable items

```
import {ref,onMounted,onUnmounted} from 'vue';

export function myTimer(interval){
    const value=ref(0);
    let timer=0;
    function clear(){
        if (timer) clearInterval(timer);
    }
    onMounted(() => timer=setInterval(() => value.value=value.value+1, interval));
    onUnmounted(() => clear());
    return {value,clear};
}
```

- Component can now use the logic

```
<script setup>
    import {myTimer} from '../MyTimer';
    const {value,clear}=myTimer(1000);
</script>
<template>
    <p @click="clear">Timer {{value}}</p>
</template>
```

Directives

- Vue gives as some built in directives
  - Show or hide based on Boolean value: v-if, v-elseif, v-else and v-show
  - Repeat element: v-for
  - Insert content: v-text and v-html
  - Databinding: v-bind and v-model
  - Event handling: v-on
  - Compiling: v-pre and v-cloak
  - Rendering: v-once
- We can also implement directives for our own purposes
  - Extra attributes that may be attached to components
  - Somehow affect the behavior or the appearance of the component

Custom Directive

- Often components would serve you better....

Object may contain hooks for:
-bind and unbind
-inserted
-update and compenentUpdated

```
app.directive("ul",(el,binding) => {
    let s=el.innerHTML;
    binding.value.forEach(char => {
        s=s.replace(new RegExp(char,"g"),"<u>"+char+"</u>");
    });
    el.innerHTML=s;
});

app.component(“dir-component",{
    directives:{
        border(el,binding){
            el.style.border=binding.value;
        }
    },
    template:`<div v-border="'1px solid black'">
                <p v-ul="['e','o']">Hello world</p>
                <p>Some <span v-ul="['t','l']">what longer</span> text</p>
            </div>`

});
```

Custom directive on setup

- Setup script may have variables prefixed with v
  - These are automatically used as custom directives
  - Just add the needed lifecycle hooks
- Example from documentation
  - Automatically set focus to an input field, not just on page load but also when component is dynamically displayed

```
<script setup>
    import {ref} from 'vue';
    const message=ref("Hello, Vue!");
    const vFocus = {
        mounted: (el) => el.focus()
    }
</script>
<template>
<div>
    <input v-focus v-model="message" />
    <p>{{message}}</p>
</div>
</template>
```

Exercise

- Can you figure out where you might want to use a custom directive or Composable?

Routing

Routing overview

With routing module we automize changing the view based on the url-pattern

- Vue-router -module needs to be loaded and configured
- hashHistory, traditional SPA urls with hashes
  - http://myserver.com/#listview
  - No server configuration
  - We load the same page, just “navigate” to a bookmark
- browserHistory
  - http://myserver.com/listview
  - Requires server configuration so that regardless of the URL the same page is served

First you need to configure routing

- Select navigation mode: createWebHistory, createWebHashHistory
- Each route is described as object
  - Path and component members, name is optional

```
import { createRouter, createWebHistory } from 'vue-router’
import HomeView from './views/HomeView.vue’

// create app

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('./views/AboutView.vue')
    }
  ]
})

app.use(router)
// mount app
```

Code for about-component will be dynamically loaded when the route is visited

And we can modify the main component

- We can use RouterLink element instead of a hrefs to navigate
- We need to have RouterView-element as a placeholder
  - Router will place the component selected by the route to this location

```
<nav>
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/about">About</RouterLink>
</nav>
<RouterView />
```

Exercise

- npm i vue-router
- Configure the routing into the main.js
  - `import Router from 'vue-router’;`
  - Instantiate the router with routes
    - Root should display the BookList
    - /calc should display the calculator
- Extend the App-component with nav-section
  - Links to Booklist and Calculator
  - And remember to add <RouterView />

Dynamic routing

- Route path may contain “parameter-slots” marked with colon
  - /person/:id
- After navigation the parameters are available at
  - $route.params.id
  - $route is a variable that can be used on template
- Or at setup

```
<script setup>
import {useRouter,useRoute} from 'vue-router'

const router=useRouter(); // Not used in this example
const route = useRoute()
let id=route.params.id;

</script>
```

Named route

- When the routing is configured a name can also be given to a route
- For router-link to an object may be given as value
    - Name-property
    - Params-property

```
// Route is configured as:
{path:’/book/:id',name:'bookDetail', component:BookDetail}

// Link could be defined as
<router-link :to="{name:’bookDetail',params:{id:4}}">{{book.id}}</router-link>
```

Programmatic navigation

The component has $router property injected for it

- push(path | object, onComplete?, onAbort?)
- replace – similar to push but the new location is not added to the history
  - Router-link has replace prop for same purpose
- go(n) where n is steps to move up or down in history
  - go(-1)=back()
- back(), forward()

Exercise

- Create BookDetail with input-fields for title and author
  - Show the component with /book/:id path
  - BookDetail should show the information about the selected book
  - And the BookDetail should also hold a Back-button
- Navigate to the BookDetail from the BookList
  - With router-link
  - With programmatic navigation
- Create BookService.js
  - Import bookService to BookList and BookDetail
  - Remove the list of books Booklist, instead use books from service
  - Query the book by id from service at BookDetail

```
import { reactive } from "vue";

export const bookService=reactive({
    books:[
        … original array of books ….
    ],
    get(id){
        return this.books.find(b => b.id==id);
    }
});
```

reactive does pretty much the same as ref but catches all mutations of hierarchical construct

Nested routing

- Basically any component may hold <router-view />
- And the route-configuration may contain children
  - An array of child-routes
- Easiest to
  - Move parameters to the child routes
  - Name the child routes
  - The container can navigate with
    - `:to="{name:'bookDetail',params:{id:personId}}“`
    - `:to="{name:’personExtra',params:{id:personId}}`

```
{path:’/person', component:PersonContainer,children:[
    {path:':id',name:’personDetail', component:PersonDetail},
    {path:':id/extra', name:’personExtra',component:PersonExtra}
]}
```

Exercise

- Create DetailContainer
  - Default view is the current BookDetail
  - Also add PrintableDetail accessible at /book/:id/printable
- DetailContainer should have
  - links to both child views

Named views

- <router-view> may have name-prop
  - And we may have several router views visible, each displaying a separate component
- If that is the case the route configuration must specify which component to display at which router-view

```
var r={path="/named",components:{
    default: ComponentForUnnamedView,
    some: ComponentForViewWithNameSome,
    other: ComponentForViewWithNameOther
}}
```

code tutors

Localization

Translations

Formattings

Localization

- Localization is about
  - Translating the string constants
  - Formatting data according to locale
    - Date-formats
    - Number-formats
    - Currency?
- Vue in itself provides no support for localization
  - vue-localization –module offers some methods for working with translations
  - Vue-cli –tool has its own approach
- ES6 offers Intl-object to support localization
  - Collators
  - Date-formattings
  - Number and currency formattings

Translating strings

- Basic idea is to define a replaceable object that holds the the translations
    - On object for each supported language
- Select which object to use when the language changes
- Instead of using constant strings in the ui use members of the selected translation object
    - The object must be globally available for all components
    - Or you might want to place it into context in the ”main”-component so that the components that need translations may query it

Intl-object (ES6)

- Can you trust that the browsers support this feature or do you need to implement a replacement for browsers that don’t support it?

```
var fiCollator=new Intl.Collator("fi");
console.log(fiCollator.compare("ä","z"));

var fiNumberFormat=new Intl.NumberFormat("fi-FI",
    {minimumFractionDigits:2,maximumFractionDigits:2});
console.log(fiNumberFormat.format(1234567.56789));

var fiCurrency=new Intl.NumberFormat("fi-FI",
                      {style: "currency", currency: "EUR" });
console.log(fiCurrency.format(1234567.89));

var fiDate=new Intl.DateTimeFormat("fi-FI");
console.log(fiDate.format(new Date()));
```

Exercise

Your wwwroot/translations holds couple of translations.json-files

- Load one of them at startup at App.vue setup script
- Figure out mechnism to change the translations on fly

```
import {ref,provide} from 'vue';
import {HTTP} from ‘./http';


const tx=ref({
    title:'SomeApp',
    buttons:{},
    book:{}
})
provide("tx",tx);

HTTP.get("/translations/translations_en.json").then(trans => {
    Object.assign(tx.value,trans);
    console.log(tx)
});
```

Instructor will point you to this

Dependency injection: someone provides, others can inject

Elsewhere (BookList) you can:

```
import {inject} from 'vue';
const tx=inject("tx");
```

```
At vite-config.js:
server:{
      proxy:{
           '/translations':'http://localhost:9000'
      }
  }
```

Thank you!

Any remaining questions?