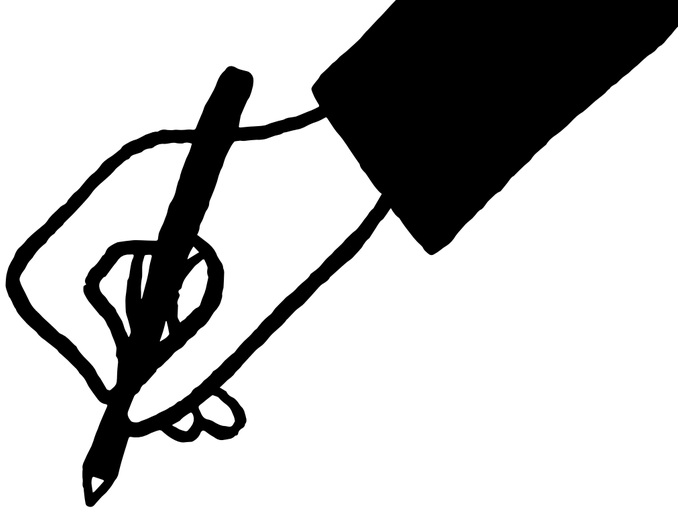


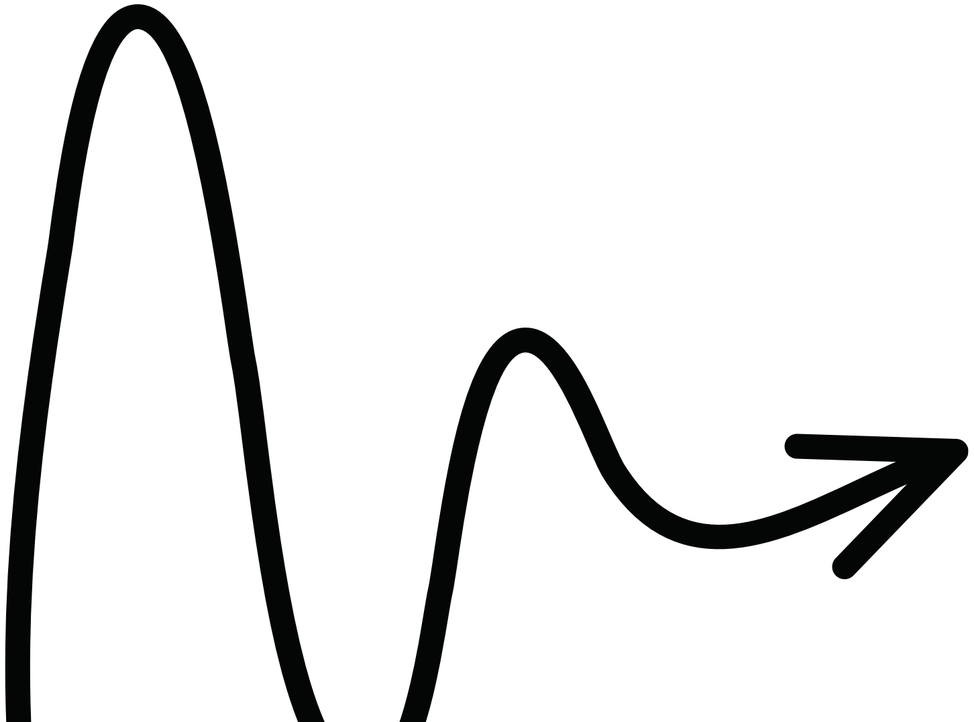
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Working with Data

Advanced Web Design

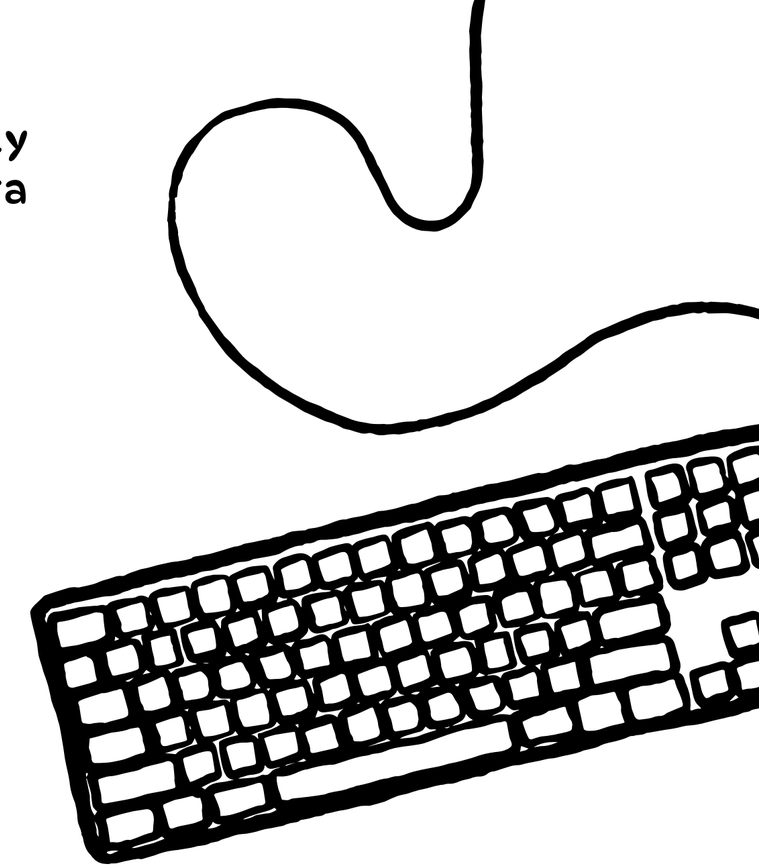
2 Working with Data

- Understanding computed properties
- Understanding computed setters
- Exploring watchers
- Watching nested properties
- Exploring async methods and data fetching
- Comparing methods, watchers, and computed props



→ Computed properties are unique data types that will reactively update only when the source data used within the property is updated. By defining a data property as a computed property, we can perform the following activities:

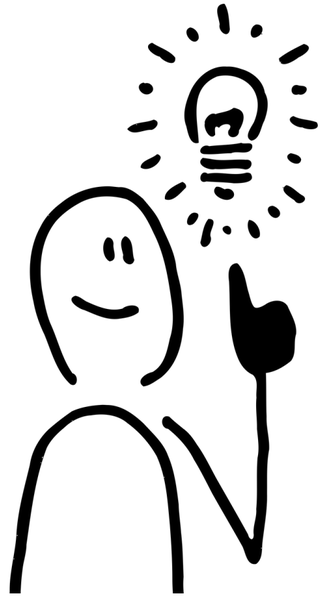
- Apply **custom logic on the original data** property to calculate the computed property's value
- **Track** the **changes** of the original data property to calculate the **updated** value of the computed property
- **Reuse** the computed property as local data anywhere within the Vue component



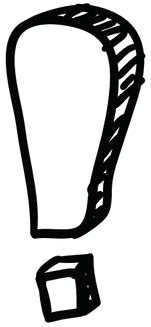
```
export default  {
  computed:  {
   yourComputedProperty() {
    /* need to have return value */
   }
  }
}
```

Understanding computed properties

→ Within the computed property's logic, you can access any component's data property, method, or other computed property using the **this** instance, which is the reference to the Vue component instance itself.



```
export default {
  data() {
   return {
    yourData: "your data"
   }
  },
  computed: {
   yourComputedProperty() {
    return `${this.yourData}-computed`;
   }
  }
}
```



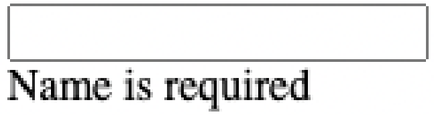
Form validation

In the following example, we have an input field, which attaches to the name data property, and error is a computed property.

If name contains a falsy value (which means name is an empty string, 0, undefined, null, or false), error will be assigned a value of "Name is required".

```
<template>
  <input v-model="name">
  <div>
    <span>{{ error }}</span>
  </div>
</template>
<script>
export default {
  data() {
    return {
     name: '',
    }
   },
  computed: {
    error() {
     return this.name ? '' : 'Name is required'
   }
  }
}
</script>
```

The error computed property updates itself whenever the name value is modified by the user.
Hence when name is empty, the output will be as follows:



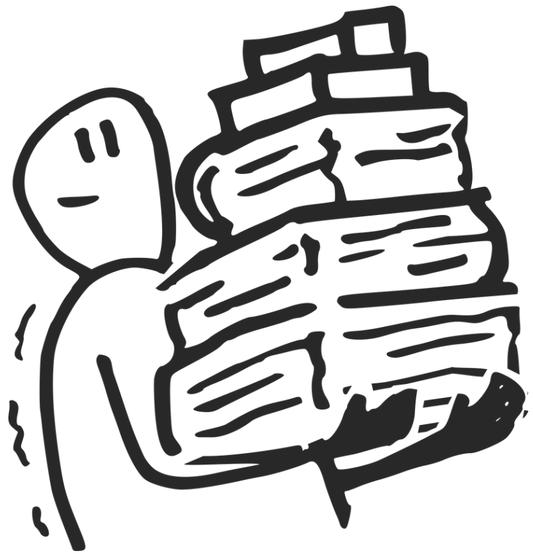
And when name is valid, the output will display just the filled input field:

Combining data props

You can use computed props to combine multiple data properties to generate a **single computed property.**
We combine two pieces of data – title and surname – into one computed string, formalName, and render its value using template.

```
<template>
  <div>{{ formalName }}</div>
</template>

<script>
 export default {
  data() {
   return {
    title: 'Mr.',
    surname: 'Smith'
   }
},
 computed: {
  formalName() {
   return `${this.title} ${this.surname}`;
   }
  }
 }
</script>
```

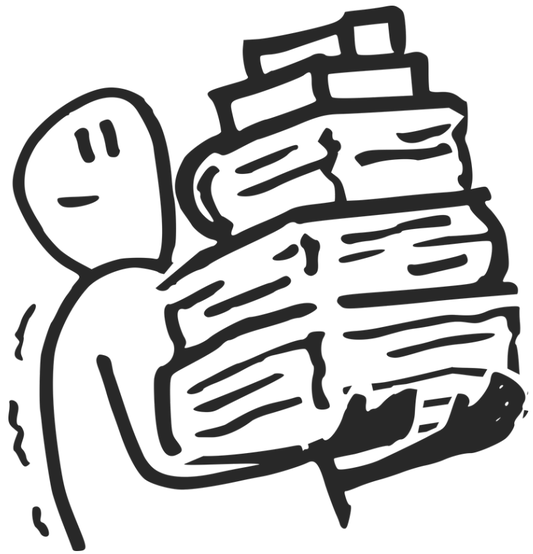


Calculating and displaying complex information

Sometimes there is a need to perform an extra calculation or to extract specific information from one large data object source. Computed properties help to achieve this goal.

In this scenario, you need to perform the following steps:
1. Display the full name of the post's author.
2. Calculate and display the total number of entries included.
3. Display a list of entries that have the feature flag turned on (feature: true).

```
data() {
 return {
  post: {
   fields: {
    author: {
     firstName: 'John',
     lastName: 'Doe'
    },
    entries: [{
     title: "Entry 1",
     content: "Entry 1's content",
     featured: true
    },
    {
     title: "Entry 2",
     content: "Entry 2's content",
     featured: false
    }]
   }
  }
 }
},
```

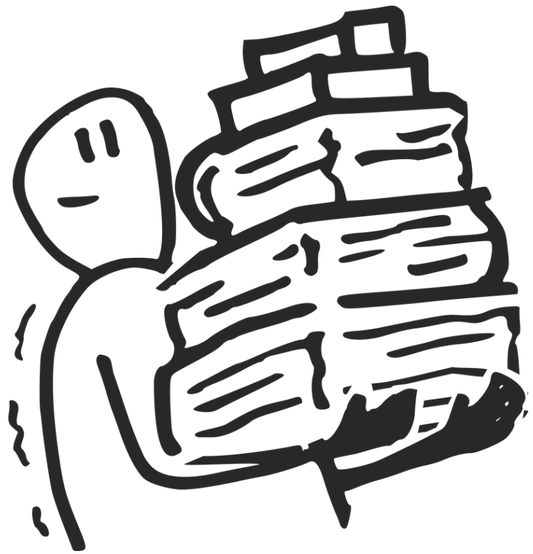


Calculating and displaying complex information

4. By using computed properties, we can decouple the previous post object into several computed data properties while keeping the original post object unchanged, as follows:

fullName for combining firstName and lastName of post.fields.author:

```
fullName() {
const { firstName, lastName } =
this.post.fields.author;
return `${firstName} ${lastName}`
},
```



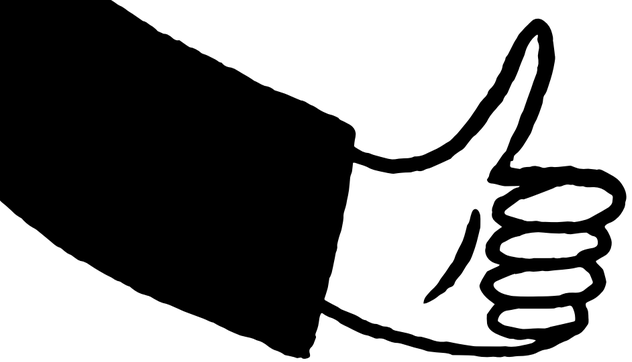
Calculating and displaying complex information

totalEntries contains the length of the post.fields.entries array:

```
totalEntries () {
return this.post.fields.entries.length
},
```

featuredEntries contains the filtered list of post.fields.entries based on the feature property of each entry, by using the filter built-in array method:

```
featuredEntries() {
const { entries } = this.post.fields;
return entries.filter(entry => !!entry.featured)
}
```



Exercise 2.01 – implementing computed data into a Vue component

Understanding computed setters

-> By default, computed data is a getter only, which means it will only output the outcome of your expression.

In some practical scenarios, when a computed property is mutated, you may need to trigger an external API or mutate the original data elsewhere in the project.
The function performing this feature is called a setter.

-> Using a setter in a computed property allows you to reactively listen to data and trigger a callback (setter) that contains the returned value from the getter, which can optionally be used in the setter.

Understanding computed setters

Starting from ES5, you can use the built-in getter and setter to define Object accessors:

get to bind the Object property to a function that returns a value for that property whenever it is looked up

```
const obj = {
get example() {
return 'Getter'
}
}
console.log(obj.example) //Getter
```

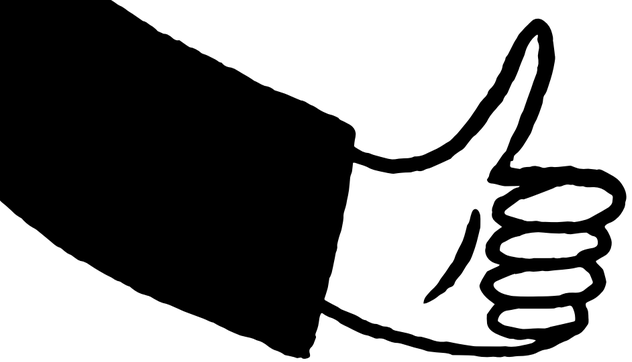
set to bind the specific Object property to a function whenever that property is modified

```
const obj = {
 set example(value) {
  this.information.push(value)
 },
 information: []
}
obj.example = 'hello'
obj.example = 'world'
console.log(obj.information) //['hello', 'world']
```

Understanding computed setters

Based on those features, Vue.js provides us with similar functionalities,
get() as the getter and set() as the setter, for a specific computed property:

```
computed: {
 myComputedDataProp: {
  get() {}
  set(value) {}
 }
}
```



Exercise 2.02 – using computed setters

This exercise demonstrates how we can use computed data to both get and set data reactively in our template by binding computed variables to the v-model

- Vue watchers programmatically observe component data and run whenever a particular property changes.
- Watched data can contain two arguments: oldVal and newVal.
- This can help you when writing expressions to compare data before writing or binding new values.
- Watchers can observe objects as well as other types, such as string, number, and array types.

Exploring watchers

- Vue watchers programmatically observe component data and run whenever a particular property changes.
- Watched data can contain two arguments: oldVal and newVal.
- This can help you when writing expressions to compare data before writing or binding new values.
- Watchers can observe objects as well as other types, such as string, number, and array types.

- In Chapter 1, we introduced life cycle hooks that run at specific times during a component's lifespan.
- If the **immediate** key is set to **true** on a watcher, then when this component initializes, it will run this watcher on creation.
- You can watch all **keys** inside any given object by including the key and value **deep: true** (the default is false).

Exploring watchers

- Vue watchers programmatically observe component data and run whenever a particular property changes.
- Watched data can contain two arguments: oldVal and newVal.
- This can help you when writing expressions to compare data before writing or binding new values.
- Watchers can observe objects as well as other types, such as string, number, and array types.

- In Chapter 1, we introduced life cycle hooks that run at specific times during a component's lifespan.
- If the **immediate** key is set to **true** on a watcher, then when this component initializes, it will run this watcher on creation.
- You can watch all **keys** inside any given object by including the key and value **deep: true** (the default is false).

- To **clean up** your watcher code, you can assign a **handler** argument to a defined component's method, which is considered best practice for large projects.

- **Watchers** complement the usage of computed data since they passively observe values and cannot be used as normal Vue data variables, while computed data must always return a value and can be looked up. Remember not to use arrow functions if you need the Vue context of this.

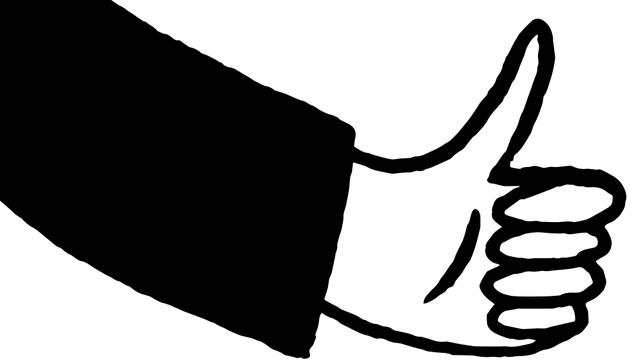
Exploring watchers

```
watch: {
myDataProperty: {
 handler: function(newVal, oldVal) {
 console.log('myDataProperty changed:', newVal, oldVal)
 },
 immediate: true,
 deep: true
 },
}
```

- To **clean up** your watcher code, you can assign a **handler** argument to a defined component's method, which is considered best practice for large projects.

- **Watchers** complement the usage of computed data since they passively observe values and cannot be used as normal Vue data variables, while computed data must always return a value and can be looked up. Remember not to use arrow functions if you need the Vue context of this.

Exploring watchers



Exercise 2.03 – using watchers to set new values

```
data() {
 return {
  organization: {
   name: 'ABC',
   employees: [
    'Jack', 'Jill'
    ]
   }
  }
 },
watch: {
 organization: {
  handler(v) {
   this.sendIntercomData()
  },
  deep: true,
  immediate: true,
 },
},
```

When using Vue.js to watch a data property, you can observe changes belonging to **nested keys** of an object, rather than observing the changes to the object itself. This is done by setting the optional **deep** property to **true**

This code example demonstrates how we watch all available keys inside the organization data object for changes. If the name property inside organization changes, the organization watcher will trigger.

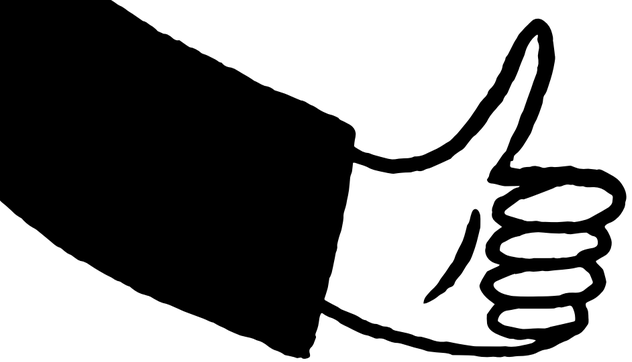
Watching nested properties

```
data() {
 return {
  organization: {
   name: 'ABC',
   employees: [
   'Jack', 'Jill'
   ]
  }
 }
},
watch: {
 'organization.name': {
  handler: function(v) {
   this.sendIntercomData()
  },
  immediate: true,
 },
},
```

If you do not need to observe every key inside an object, it is more performant to assign a watcher to a specific key by specifying it following the syntax **<object>.<key>** string.

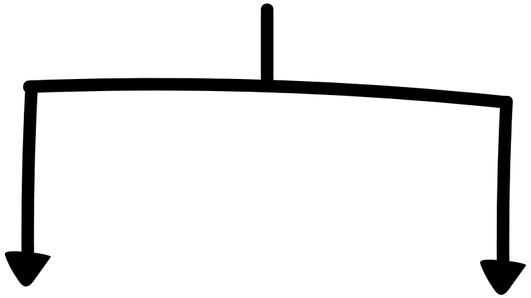
In the following example, the watcher is explicitly observing the name key of the organization object:

Watching nested properties



Exercise 2.04 – watching nested properties of a data object

Exploring async methods and data fetching



async

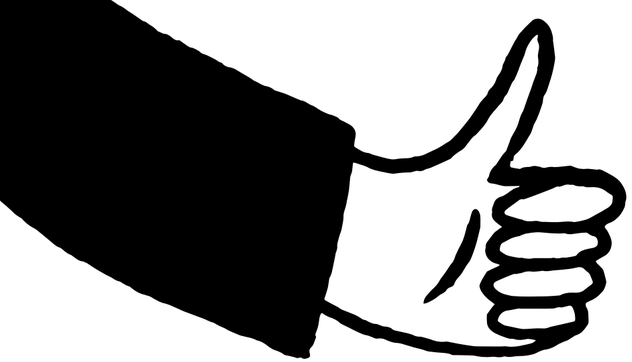
Asynchronous functions in JavaScript are defined by the **async** syntax and return a **Promise**. These functions operate asynchronously via the **Event loop**, using an implicit promise, which is an object that may return a result in the future.

then(), catch()

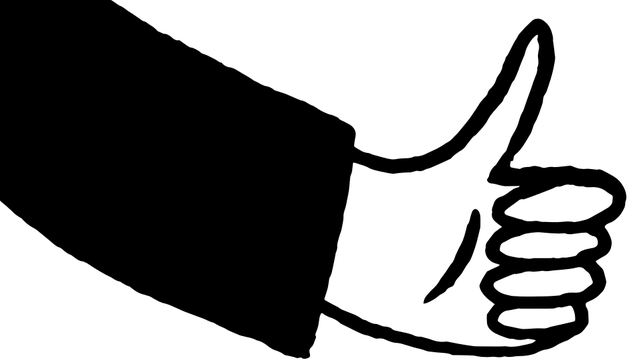
You can use Promise chaining methods, such as the **then() and catch()** functions or try the **await** syntax of ES6 inside these Vue methods and return the results accordingly

```
export default {
 methods: {
  async getAdvice() {
  const response = await fetch('https://api.adviceslip.com/advice')
  return response;
  },
 },
}
```

Axios is a popular JavaScript library that allows you to make external requests for data using Node.js. It has wide browser support making it a versatile library when making HTTP or API requests.

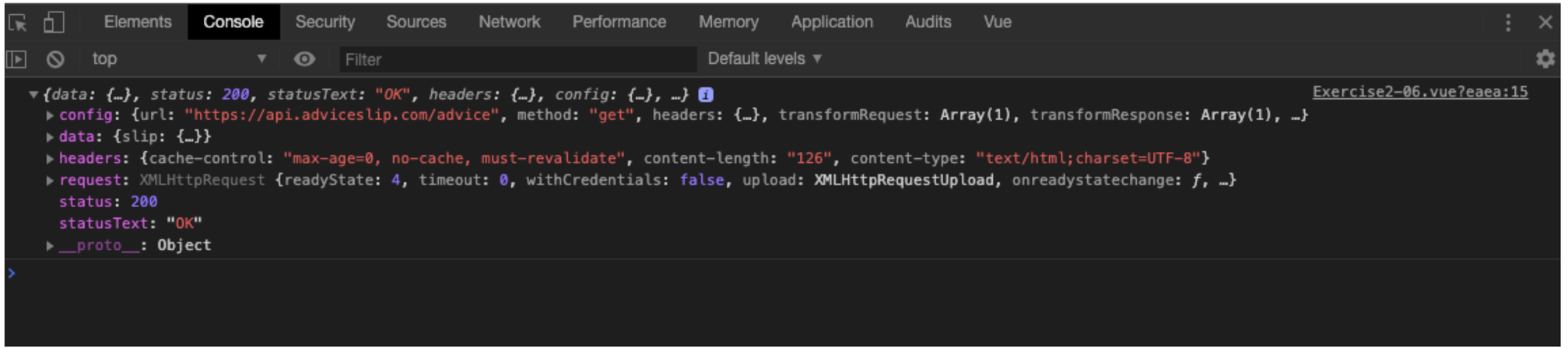


Exercise 2.05 – using asynchronous methods to retrieve data from an API

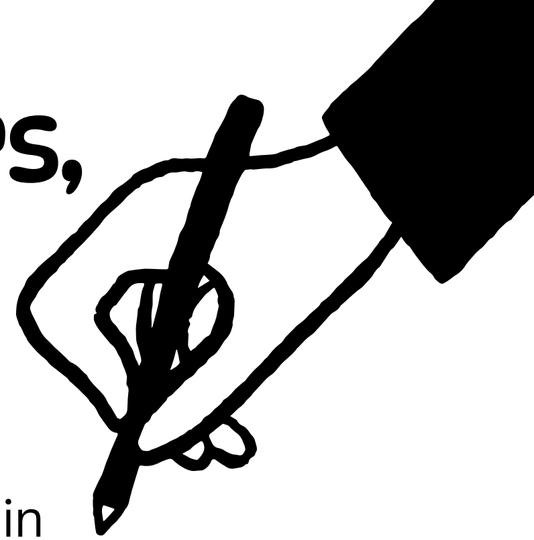


Async fetch

Learn something profound



Comparing methods, watchers, and computed properties



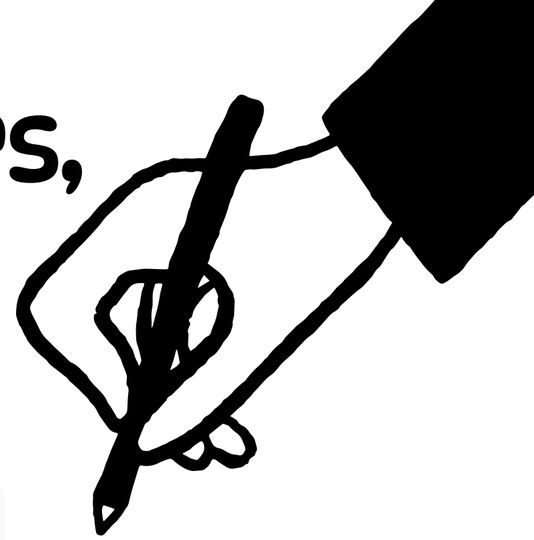
Methods are best used as a handler to an event occurring in the DOM, and in situations where you need to call a function or perform an API call, for example, Date.now(). All values returned by methods are not cached.
For example, you can compose an action denoted by @click, and reference a method:

```
<template>
  <button @click="getDate">Click me</button>
</template>

<script>
 export default {
  methods: {
  getDate() {
  alert(Date.now())
  }
 }
}
</script>
```

Methods should not be used to display computed data, since the return value of the method, unlike computed props, is not cached, potentially generating a performance impact on your application if misused.

Comparing methods, watchers, and computed properties



Computed props are best used when reacting to data updates or for composing complicated expressions in your template.

Their reactive nature makes computed properties perfect for composing new data variables from existing data, such as when you are referencing specific keys of a larger, more complicated object.

```
<template>
 <div>{{ animals }}</div>
</template>

<script>
 export default {
  data() {
   return {
    animalList: ['dog', 'cat']
   }
  },
 computed: {
  animals() {
   return this.animalList.slice(1)
  }
 }
}
</script>
```

Comparing methods, watchers, and computed properties

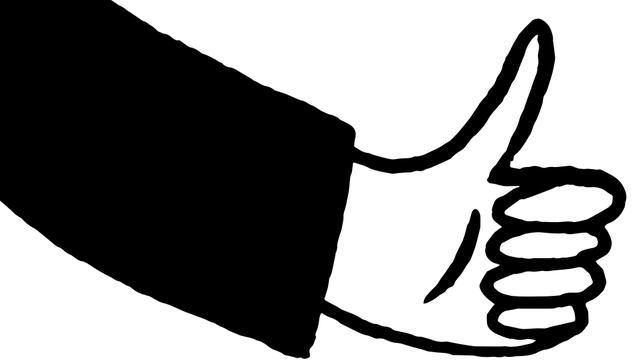
In many cases, using computed props can be overkill, such as when you only want to **watch** a specific data's nested property rather than the whole data object.

Or when you need to listen and perform an action upon any changes of a data property or a specific property key nested inside a data property object, and then perform an action. In this case, data **watchers** should be used.

Because of the unique newVal and oldVal arguments of a watcher, you can watch a variable's changes and perform an action only when a certain value is reached:

```
<template>
<div>
<button @click="getNewName()">
Click to generate name </button>
<p v-if="author">{{ author }}</p>
</div>
</template>

<script>
 export default {
  data() {
   return {
   data: {},
    author: '',
   }
  },
 watch: {
  data: function(newVal, oldVal) {
   this.author = newVal.first
  alert('Name changed from ${oldVal.first} to
${newVal.first}')
  }
 },
methods: {
 async getNewName() {
  await fetch('https://randomuser.me/api/').then(response =>
response.json()).then(data => {
this.data = data.results[0].name
   })
  },
 },
}
</script>
```



Exercise 2.06 – handling the search functionality using a Vue method, a watcher, and computed props

Questions?