L9: Frontend Abstractions

Web Engineering
188.951 2VU SS20

Jürgen Cito

TU WIEN Informatics

L9: Frontend Abstractions

- Overview of abstractions that support building declarative and reactive frontend applications
- Case study demonstrating these abstractions in Vue.js
- Live demonstration of refactoring RecipeSearch from plain JS to Vue.js

Learning Goals

- Understand the interplay between data and design/output and how binding can enable reactive frontends
- Design components that properly encapsulate data, behaviour, design and output
- Ability to map these concepts and abstractions to their concrete counterparts in Vue.js

Concepts and Abstractions in Web Frontend Frameworks

Goal: Enable **declarative** and **reactive** frontends

Routing
(Single Page Applications)

Enables modularization and reusability by encapsulting various aspects of the component

Components

Encapsulation

- Data Encapsulation
- Style Encapsulation
- Behavior Encapsulation

Data Binding

- One-way Binding (Unidirectional)
- Two-way Binding (Bidirectional)

Establishes declarative relationship between components and models

facilitated through

Templates

Declarative Rendering

State Management

Sometimes requires management of (global) shared state across components

Imperative vs Declarative Frontends

Imperative

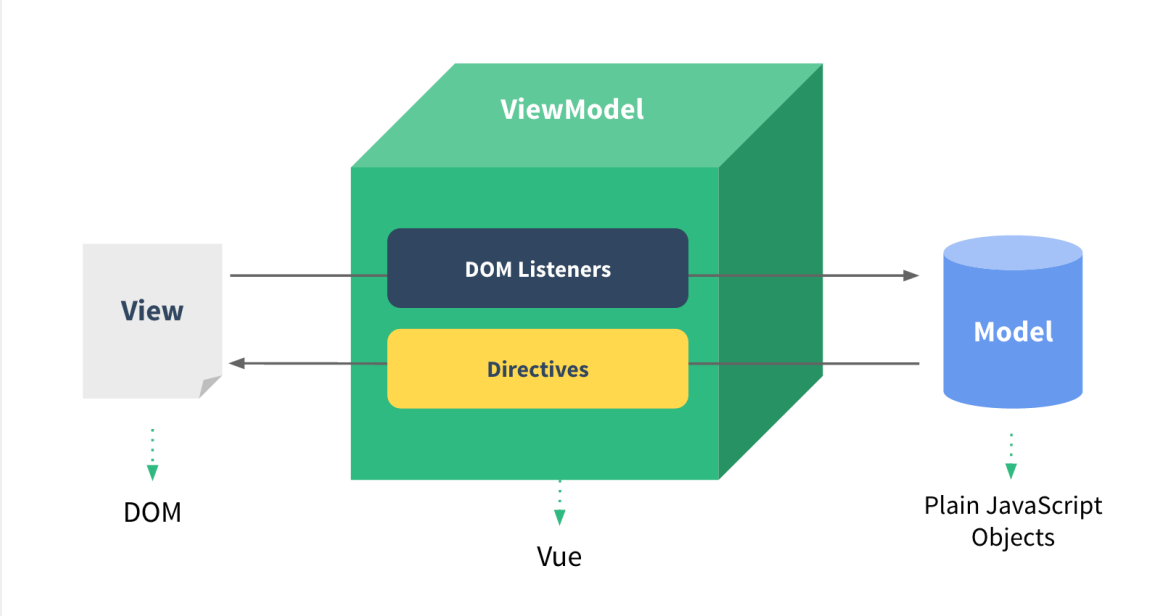
- Need to create new (sometimes ad-hoc) DOM elements and styles as data is introduced
- Requires modifying the DOM as data changes (harder to separate logic from rendering)
- Need to change model as the HTML changes (user input)

Declarative

- Output is represented declaratively with templates
- Binding declares relationships between model and output
- DOM is updated based on model updates “behind the scenes” (instead of imperatively manipulating the DOM)

MVVM (Model-View-ViewModel)

- Data (*Model*) are represented as JavaScript objects
- Complete separation of design/output (*View*) and logic (*ViewModel*)
- *ViewModel* handles relationship between *View* and *Model*
  - Updates the *View* when properties of the *Model* changes
  - Encapsulates methods that modify the *Model*



Recall: Backend Templating

Templates (sometimes also called views) provide separation between program logic and output.

Template engines replace variables in static template files and control structures (conditionals and loops) with values passed from the program.

```
app.set('view engine', 'pug')
...
routes.get('/', async (req, res) => {
  res.render('users', { title: 'Users',
    heading: 'List of users', users: getUsers() });
}
```

PUG Template - users.pug

```
html
  head
    title= title
  body
    h1= heading
    div#container
      - for user in users
        div.user= user.email
```

Output for rendered response

```
<html>
    <head>
        <title>Users</title>
    </head>
    <h1>List of users</h1>
    <div id="container">
        <div class="user">
          jane.doe@tuwien.ac.at
        </div>
        <div class="user">
          jack.bauer@tuwien.ac.at
        </div>
    </div>
</html>
```

Backend vs Frontend Templating

Backend Templates

- The backend receives a request, retrieves/computes data, and generates HTML files
- Templates are static markup files that are expanded based on data/values
  - Template variables are replaced with values
  - Loops: Iterate over lists of values and generate HTML for each instance
  - Conditionals: Generate different HTML depending on values

Frontend Templates

- Conceptually very similar to backend templates (template variables, loops, conditionals)
- Reactive: Values might change based on model changes
  - Model changes can be triggered by user input
  - Model (changes) can be retrieved from backend
- DOM is updated

Frontend Abstractions: Case Study in Vue.js

Vue.js is a frontend JavaScript framework following the MVVM model to build user interfaces and single page applications. It also comes with useful tooling to facilitate creation/maintenance and debugging.

Use as Library

It can be incrementally adopted by using its core functionality around views as a library.

```
<!-- Embed the following script to get started -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<div id="app">
    Hello, {{ course }}
</div>

<script>
  var app = new Vue({
    el: '#app',
    data: {
      course: 'Hello Web Engineering!'
    }
  })
</script>
```

View
(with template variables)

ViewModel
(keeps view and model in-sync)

Model
(holds the data)

Reactive: Changing 'course' in the model will change the view

Frontend Abstractions: Case Study in Vue.js

Use as Framework
It can be also used as a fully-fledged framework to build single page applications in combination with tooling.

Components are then organized in .vue Files and transpiled to JavaScript in a build process.

Use the vue-cli to create boilerplate code/initial structure of your Vue.js project.

App.vue

```
<template>
  <div id="app">
    Hello, {{ course }}
  </div>
</template>
<script>
  export default {
    name: 'App',
    data : function() {
      return {
        course: 'Web Engineering'
      }
    },
  }
</script>
<style>
  #app {
    font-family: Arial;
    color: #2c3e50;
  }
</style>
```

Beware: Data here is a function that returns the model because this is already a component

Components in Vue.js
Top down perspective

Components are reusable building blocks

- Template HTML code in View (<template>)
  - Declares binding to internal model and properties through so called interpolations | Syntax: `{{ data }}`
  - Supports bounded loops and conditional rendering
- Behavior in ViewModel
  - Input parameters (`props`) that become part of the internal model (`data`)
  - Derived, computed values (`computed`)
  - Registered sub-components to use in the template (`components`)
  - Functions to deal with event handling (`methods`)
  - Life-cycle methods (`mounted, created`)
- Encapsulate (scoped) style that are bound to component

RecipeItem.vue

```
<template>
    <article>
      <h2>{{recipe.title}}</h2>
      ...
    </article>
</template>

<script>
    export default {
        name: "RecipeItem",
        props: {
            recipe: {
                title: String,
                thumbnail: String,
                url: String,
                ingredients: Array
            }
        }
    }
</script>

<style scoped>
    .ingredients li {
        text-decoration: underline;
        cursor: pointer;
    }
</style>
```

One-Way Binding

Declares binding to internal model and properties

- Bindings as part of DOM content nodes are declared through interpolation syntax: `{{ data }}`
  (Interpolations are inline expressions, i.e., can be any JavaScript code)

- Bindings as part of attributes are defined using directives
  `<img v-bind:src="standardImage">`
  or
  `<img :src="standardImage">` (syntactic sugar)

- One-way refers to the direction of data-flow
  Values from the model and properties are bound to the template variables to create the output when expanded

RecipeItem.vue

```
<template>
    <article>
      <h2>{{recipe.title}}</h2>
      ...
      <img :src="standardImage">
    </article>
</template>

<script>
    export default {
        name: "RecipeItem",
        props: {
            recipe: {
                title: String,
                thumbnail: String,
                url: String,
                ingredients: Array
            }
        },
        data : function() {
          return {
            standardImage :
             "standard.jpg"
          }
        }
    }
</script>
```

Two-Way Binding

Declares binding to and from internal model (form inputs)

- Model changes are reflected in the view (as in one-way binding)
- Changes in the view are reflected in the model (and consequently to all bindings that have been established on the model)

- Binding through `v-model` directive

  `<input v-model="ingredientInput">`

- Not possible for properties, as direct binding to parent models would cause maintainability nightmares (use events or managed shared state instead)

RecipeSearch.vue

```
<template>
  <section id="search">
   <form …>
     <label for="ingredients">
         Ingredients
     </label>
     <input v-model="ingredientInput"
       type="text" name="ingredients" />
     <button type="submit">Search</button>
   </form>
  </section>
  ...
</template>
```

Computed Properties

Derived properties from model

- Inline expressions should be limited to simple operations
- Computed properties are declarative values with more complicated logic that depend on model values
- Declared as named functions in the `computed` object
- Can be bound in the template (if you want two-way binding, you have to establish an object with a `get` and `set` function)
- Have to be *deterministic* and *synchronous*
- They are *cached* and only lazily re-evaluated when their reactive dependencies change

RecipeSearch.vue

```
computed : {
  ingredients : {
    get:  function() {
      return this.ingredientInput.split(',');
    },
    set: function(ingredients) {
      this.ingredientInput = ingredients.join(',');
    }
  }
}
```

RecipeItem.vue

```
<template>
    <article>
      <h2>{{recipe.title}}</h2>
      ...
      How many? {{ ingredientCount }}!
    </article>
</template>

<script>
  export default {
    name: "RecipeItem",
    props: {
      recipe: {
        title: String,
        thumbnail: String,
        url: String,
        ingredients: Array
      }
    },
    computed : {
      ingredientCount : () => {
        return this.ingredients.length;
      }
    }
  }
</script>
```

Conditional Rendering

Render elements only if expression evaluates to true

- Controlled by directives `v-if, v-else, v-else-if`
- To apply directives to groups of elements either apply to a parent element or to child `<template>` element
- `v-show` directive has similar behaviour
  Difference: Only sets CSS `display: none`, while others actually remove (or never insert) elements in the DOM

RecipeItem.vue

```
<template>
  <article>
    <h2>{{recipe.title}}</h2>
    <div>
      <div>
        <img v-if="recipe.thumbnail"
             :src="recipe.thumbnail">
         <img v-else :src="standardImage">
         <template v-if="recipe.url">
            &rarr;
            <a :href="recipe.url">
              Full Recipe
            </a>
         </template>
    </div>
  </article>
</template>
```

Bounded Loops (List Rendering)

Map elements in an array to HTML elements

- Controlled by directives `v-for` and `v-bind:key`
- Key should be a unique element (ID) that is used by Vue internally for performance reasons

- Iterator also provides the index if needed
  `<li v-for="(item, idx) in items" :key="idx">`

RecipeItem.vue

```
<template>
...
  <h3>Ingredients</h3>
    <ul class="ingredients">
      <li v-for="ingredient in recipe.ingredients"
          :key="ingredient.name">
        {{ ingredient }}
      </li>
    </ul>
...
</template>
```

Events and Methods

Reacting to DOM events and adding event listeners

- Similar event model to plain JavaScript with callback mechanism for event handlers
- Event listeners can be either Inline expressions or calling named function declared in `methods` object
- Binding event listener directly on element
  `<button v-on:click="recipeSearch()">`
  or
  `<button @click="alert('Hello!')">` (syntactic sugar)
- Event Modifiers (e.g., `v-on:submit.prevent`) are directive postfixes that capture common functionality in events (`event.preventDefault()` in this example)

RecipeSearch.vue

```
<template>
  <div id="recipeSearch">
    <section id="search">
      <form role="search"
            v-on:submit.prevent="recipeSearch()">
        <label for="ingredients">Ingredients</label>
        <input v-model="ingredientInput"
               type="text" name="ingredients" />
        <button type="submit">Search</button>
      </form>
    </section>
    <section id="results">
      <recipe-item v-for="(recipe, index) in recipes"
                   :key="index" :recipe="recipe" />
    </section>
  </div>
</template>
<script>
  export default {
    ...
    methods : {
      recipeSearch : async function() {
        this.recipes = ...
      }
    }
    ...
  }
</script>
```

Custom Events

Emitting and handling custom events on components

- Custom events can be emitted with this.**$emit**('ingredientAdd')
- Event Listener has to be registered with exact name (no kebab-casing)

```
<recipe-item
    v-on:ingredientAdd="addIngredientToSearch" />
```

RecipeSearch.vue

```
<recipe-item @ingredientAdd="addIngredientToSearch"
     v-for="(recipe, index) in recipes" :key="index"
     :recipe="recipe" />
```

RecipeItem.vue

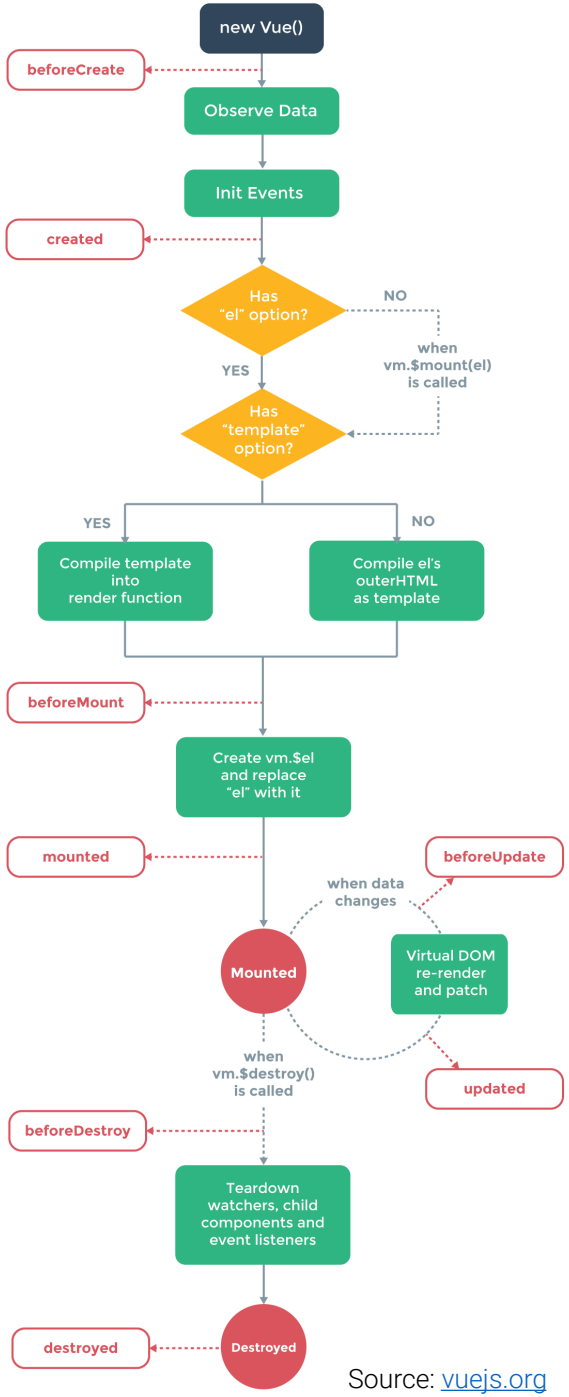
```
<template>
...
  <h3>Ingredients</h3>
    <ul class="ingredients">
      <li @click="addIngredient(ingredient)"
          v-for="ingredient in recipe.ingredients"
        :key="ingredient.name">
        {{ ingredient }}
      </li>
    </ul>
...
</template>
<script>
...
  methods : {
    addIngredient : function(ingredient) {
      this.$emit("ingredientAdd", ingredient);
    }
  }
...
</script>
```

Lifecycle Hooks

Callbacks provided for events in component's life cycle

- `mounted()` Hook
  - Most commonly used hook
  - Is called after DOM has been fully rendered for component (Beware: no guarantee that all child components have been rendered yet)
  - Ability to access templates, reactive component, and manipulate all elements in the DOM

- `created()` Hook
  - DOM has not been loaded yet
  - All options in component are available (data, computed properties, methods, etc.)
  - Used to trigger actions like fetching data from backends



Source: vuejs.org

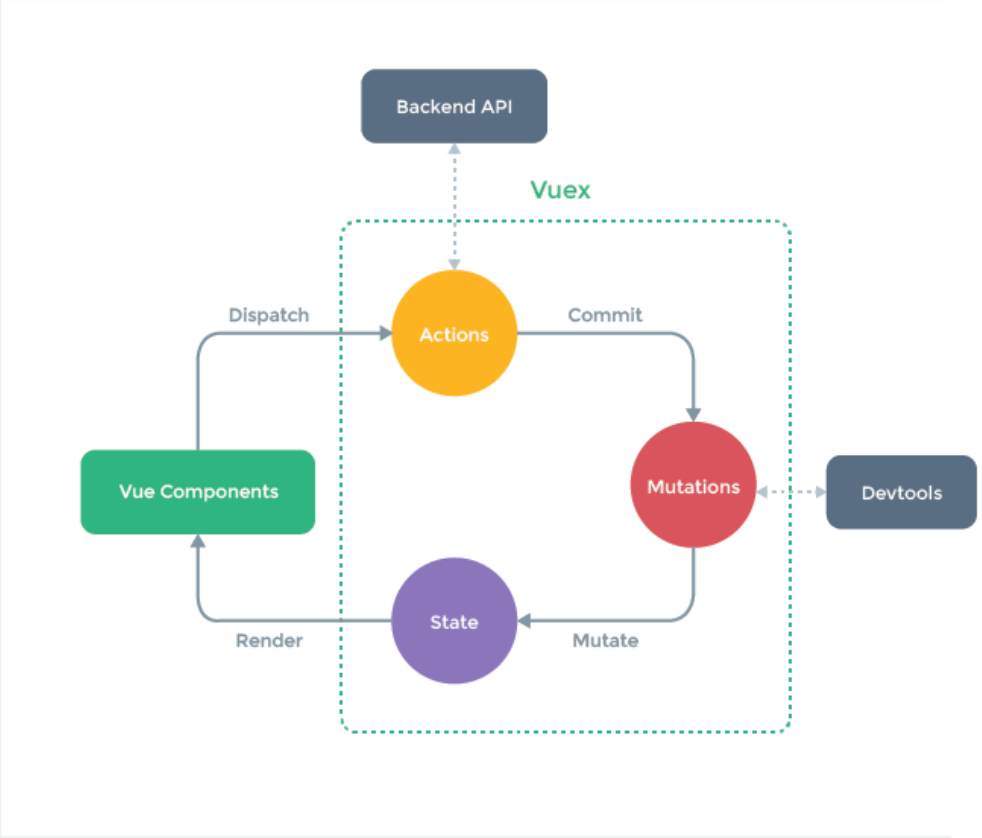
Disclaimer: Not necessary to know whole lifecycle depicted in diagram for the test!

State Management

Managing shared global state in a principled manner

- Vuex is a library that introduces abstractions that create a life-cycle of dealing with reactive global state. This ensures that every state change is tracked to enable better program understanding and debugging.

- There is no direct access to the state data structures. The only way to change the state is by using the following techniques
  - **Committing Mutations:** Predefined synchronous functions called *mutations* are the only way manipulate state. For instance, by calling `store.commit('increment')` you can trigger a predefined increment mutation. Committing a mutations entails that there is a record of the side effects caused by it (state changes).
  - **Actions are functions that can commit mutations:** Actions can have asynchronous operations (e.g., to communicate with the backend). They can be dispatched (i.e., called) in components `this.$store.dispatch('incrementCart')`
- Libraries for reactive state management can be overkill for smaller applications. Encapsulate state as much as possible and weigh the trade-offs of using such a complex approach



Routing

Browser-like navigation for Single-Page Applications

- Simulate standard navigation by manipulating the browser history
- URL fragments allow linking to different logical "pages" while staying on the same browser page

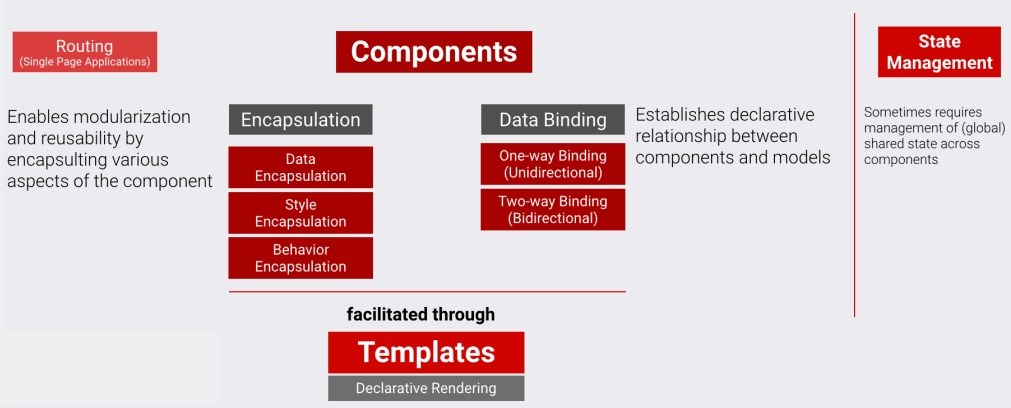
```
https://www.example.com/#/config/437853
```

- Vue Router library
  - Same concept as server-side routing
  - Can pass URL parts as props to components

```
const routes = [
    { path: '/', redirect: '/search'},
    { path: '/search', component: Search},
    { path: '/cart', component: Cart},
    { path: '/checkout', component: Checkout},
    { path: '/config/:artworkId', component: Config, props: true },
    { path: '*', component: PageNotFound},
]
```

Endless Variety of Frontend Frameworks

- Different philosophies
- Different corporate backing
- Same concepts and abstractions



Vue.js

ANGULARJS by Google

Stimulus

React

Polymer

BACKBONE.JS

ember