Tulipan.js: The Javascript Framework

Tulipan.js is a simple minimalistic and progressive JavaScript Framework for building Single Page Applications on the web.

Attention

This Framework is still under development process, some features are still in test, use them with caution.

Introduction

Tulipan.js was made for teaching purposes, but since it uses modern web technologies, it can be used also for production and commercial purposes.

Tulipan.js is a forked and a combination of several Javascript pieces of technologies. Following the Single-Responsability Principle, it combines the best selected tools for each of the features found in a common Single Page Application.

Installation

Drop the following into your page:

```
<script src="//unpkg.com/tulipan"></script>
```

You can also use jsdelivr

```
<script src="https://cdn.jsdelivr.net/npm/tulipan@1.0.2/dist/
tulipan.min.js"></script>
```

or use via npm:

```
npm install tulipan --save
```

Integration

Consider Tulipan.js as a Swiss Tool, it integrates in a convenient way several Javascript libraries and Frameworks, to allow beginners to develop high end modern web applications. The following list shows all technologies integrated and their use.

Library/ Framework	Description
Vue.js	Using the branch 1.0, Vue.js is the base front-end framework for the Model-View-ViewModel pattern
vue-resource	Selected http client, it is used to make web requests across the Internet
Navigo	Library selected to make javascript routing for SPA development
Store.js	Library selected for cross-browser storage
Underscore.js	Library selected for functional programming

Support development

If you liked this, donate to the cause.

License

MIT

Getting Started

You can start coding with Tulipan.js fast and simple, let's dive into the most basic with Tulipan, let's code our first application. A to-do list app.

Data bindings

The first thing to grasp is the binding of the model's data into our HTML. Since Tulipan uses DOM templating rules we can start with the following code.

HTML

```html
<div id="app">
  {{ todo }}
</div>
```

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    todo: 'clean the room!'
  }
})
```

Which will render

```
clean the room!
```

Two-way binding

Now we want to render based on user interaction, we can use an input to bind the user input into the DOM.

```html
<div id="app">
  <p>{{ todo }}</p>
  <input tp-model="todo">
</div>
```

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    todo: 'clean the room!'
  }
})
```

Which will render

·

{{ todo }}

the main difference in this case is that we used a html custom attribute *tp-model*, this attribute is inspected by Tulipan to perform the two-way binding. In the following examples we will use this same format *tp-attr* to perform other actions.

Binding sequences

Sometimes we need to render a list of items, in our case is a list of to-dos.

HTML

```html
<div id="app">
  <ul>
    <li tp-for="todo in todos">
      {{ todo.text }}
    </li>
  </ul>
</div>
```

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    todos: [
      { text: 'Clean the room!' },
      { text: 'Take out trash!' },
      { text: 'Buy groceries' }
    ]
  }
})
```

Which will render

- Clean the room!
- Take out trash!
- Buy groceries

As you can see we can also bind properties from our data, and using the *tp-for* attribute we can tell Tulipan to iterate over a set of elements.

Adding elements dynamically

Let's now try to add new elements with our input.

HTML

```
<div id="app">
  <span>Enter to-do: </span><input tp-model="newTodo" tp-
on:keyup.enter="addTodo">
  <ul>
    <li tp-for="todo in todos">
      {{ todo.text }}
    </li>
  </ul>
</div>
```

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    newTodo: "",
    todos: [
      { text: 'Clean the room!' },
      { text: 'Take out trash!' },
      { text: 'Buy groceries' }
    ]
  },
  methods: {
    addTodo: function(){
      var text = this.newTodo.trim()
      if (text) {
        this.todos.push({ text: text })
        this.newTodo = ''
      }
    }
  }
})
```

Which will render the following

Enter to-do: .

- {{ todo.text }}

This time you can add new to-dos to the list by pressing enter on the text input. A new thing introduced in this exercise was the capability of adding a custom method to our *ViewModel*, and trigger this method through user input.

Let's remove some elements

We now have graps some of the basics in our application, let's put some code to remove those finished to-dos.

HTML

```html
<div id="app">
  <span>Enter to-do: </span><input tp-model="newTodo" tp-on:keyup.enter="addTodo">
  <ul>
    <li tp-for="todo in todos">
      <span>{{ todo.text }}</span>
      <button tp-on:click="removeTodo($index)">X</button>
    </li>
  </ul>
</div>
```

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    newTodo: '',
    todos: [
      { text: 'Clean the room!' },
      { text: 'Take out trash!' },
      { text: 'Buy groceries' }
    ]
  },
  methods: {
    addTodo: function () {
      var text = this.newTodo.trim()
      if (text) {
        this.todos.push({ text: text })
        this.newTodo = ''
      }
    },
    removeTodo: function (index) {
      this.todos.splice(index, 1)
    }
  }
})
```

Which will render

Enter to-do: .

- {{ todo.text }} X

Pretty cool right?

This was the basic on Tulipan.js, in the following sections you will find how to perform more advanced front-end features. So you can create Single Page Applications as soon as you can say Tulipan.

Bindings

User inputs and user interaction with your models are done through bindings, in this guide you will find code examples of bindings between models and form elements suchas inputs.

Text Binding

The most common form of binding, is text binding, this way we can sync our model with the DOM, the standard format in order to bind text or other data simple data type (integer, float, boolean), is to use double mustache *{{ }}* syntax:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    message: 'This is an example text'
  }
})
```

HTML

```
<div id="app">
    <span>{{ message }}</span>
</div>
```

Which will render

```
{{ message }}
```

Keep in mind that you can only bind data within the same context application, DOM child nodes provided by the **el** parameter.

HTML Binding

Sometimes, we retrieve data from our server or any other API service, in the form of raw HTML. and we need to render this HTML instead of showing it as code. In order to do this we can use the triple mustache syntax *{{{ }}}*:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    raw: '<span><b>This is an example text</b></span>'
  }
})
```

HTML

```
<div id="app">
    {{{ raw }}}
</div>
```

Which will render

```
{{{ raw }}}
```

Keep in mind that bindings within the raw HTML are ignored.

Binding Expressions

We can also do some basic JavaScript expressions in order to process our data before rendering.

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    age: 21,
    older: true,
    cocktail: "Vodka Martini"
  }
})
```

HTML

```
<div id="app">
   <span>Age: {{ age }}</span>
   <span> - Is older?: {{ older ? 'YES' : 'NO' }}</span>
   <span> - Favorite Cocktail: {{ cocktail.toUpperCase() }}</span>
</div>
```

Which will render

Age: {{ age }} - Is older?: {{ older ? 'YES' : 'NO' }} - Favorite Cocktail: {{ cocktail.toUpperCase() }}

You can also create custom filters and pipe them in order to perform most advanced data processing.

Class and Style Bindings

We often have to manipulate an element's class and its inline styles. Tulipan.js handle them using **tp-bind** providing special enhancements when is used for class and style, which avoid string manipulation errors. Besides Strings, the expressions can also evaluate to Objects or Arrays.

Binding HTML Classes

We can pass an Object to `tp-bind:class` to change classes on the fly. Observe that the `tp-bind:class` directive can co-exist with the plain `class` attribute:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    showAlertSuccess: true,
    showAlertDanger: false
  }
})
```

HTML

```
<div id="app">
  <div class="static" tp-bind:class="{'alert alert-success' :
showAlertSuccess, 'alert alert-danger' : showAlertDanger}"></div>
</div>
```

Which will render

```
<div calss="static alert alert-success"></div>
```

Keep in mind that if `showAlertSuccess` and `showAlertDanger` changes, the class list will too.

Let's say that `showAlertSuccess` turns into `false` and `showAlertDanger` is `true`, then the class list will become `"static alert alert-danger"`.

You can also bind an Object as follows:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    myObject:{
    'showAlertSuccess': true,
    'showAlertDanger': false
    }
  }
})
```

HTML

```
<div id="app">
  <div class="static" tp-bind:class="myObject"></div>
</div>
```

This will render the same as above.

We can pass an Array to `tp-bind:class` to apply a list of classes:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    showAlertSuccess: 'alert alert-success',
    showActive: 'active'
    showClose: 'close'
  }
})
```

HTML

```
<div id="app">
  <div tp-bind:class="[showAlertSuccess, showActive, showClose]"></div>
</div>
```

Which will render

```
<div calss="alert alert-success active close"></div>
```

Binding Inline Styles

With Tulipan.js you can bind styles using `tp-bind:style` directive, which has pretty easy syntax. Despite it's a JavaScript object it looks like CSS. Let's check out this example:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    activeColor: 'red',
    fontSize: 30
  }
})
```

HTML

```
<div id="app">
  <div tp-bind:style="{ color: activeColor, fontSize: fontSize +
'px' }"></div>
</div>
```

Which will render

```
<div style="color: red; font-size: 30px;"></div>
```

We can bind in a cleaner way using a style object:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    myStyleObject:{
    color: 'red',
    fontSize: '30px'
    }
  }
})
```

HTML

```
<div id="app">
  <div tp-bind:style="myStyleObject"></div>
</div>
```

This will render the same.

We can observe that Object syntax is used in conjunction with computed properties to return Objects.

It is also posible to apply many style objects to the same element using Array syntax for `tp-bind:style` directive:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    myFirstStyleObject:{
      color: 'green',
      fontSize: '50px'
    },
    mySecondStyleObject:{
      fontFamily: 'verdana',
      textAlign: 'center'
    }
  }
})
```

HTML

```
<div id="app">
  <div tp-bind:style="[myFirstStyleObject, mySecondStyleObject]"></div>
</div>
```

Which will render

```
<div style="color: green; font-size: 50px; font-family: verdana; text-
align: center;"></div>
```

Form Input Bindings

Basics Usage

Updating data based on input users is pretty easy using Tulipan.js, with the **tp-model** directive you can create two-way data binding on form input and text area elements. Let's see some examples:

Text

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    message: ''
  }
})
```

HTML

```html
<div id="app">
  <span>This is a message: {{ message }}</span>
  <br>
  <input type="text" tp-model="message" placeholder="you can edit me!">
</div>
```

Which will render

This is a message: {{ message }}

.

Multiline text

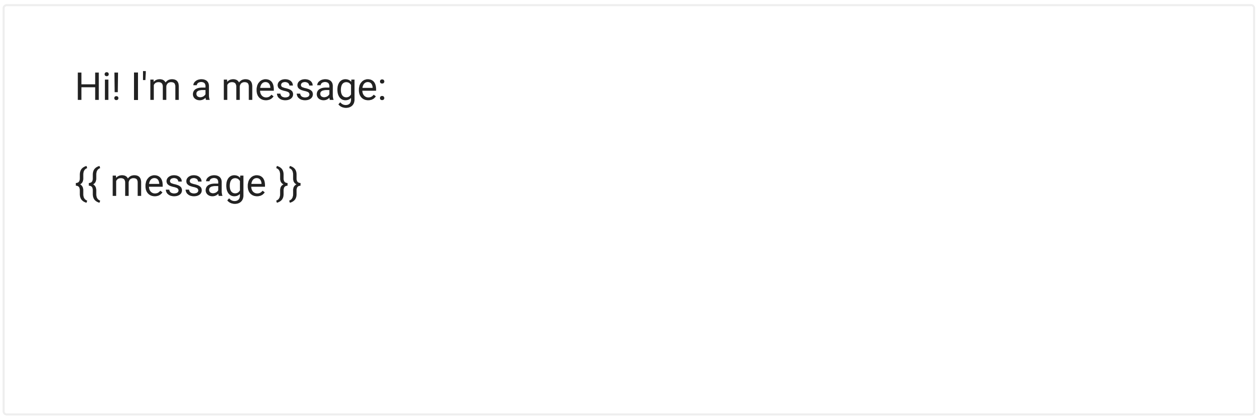
JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    message: ''
  }
})
```

HTML

```html
<div id="app">
  <span>Hi! I'm a message:</span>
  <p>{{ message }}</p>
  <br>
  <textarea tp-model="message" placeholder="You can write a lot of
lines!"></textarea>
</div>
```

Which will render



Hi! I'm a message:

{{ message }}

Checkbox

Single checkbox:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    checked: false
  }
})
```

HTML

```
<div id="app">
  <input type="checkbox" id="checkbox" tp-model="checked">{{checked}}
</div>
```

Which will render

.{{checked}}

Multiple checkbox:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    checkedLanguages: []
  }
})
```

HTML

```
<div id="app">
  <span>You can pick many languages!</span>
  <br>
  <input type="checkbox" id="python" value="Python" tp-
model="checkedLanguages">
  <label for="python">Python</label>
  <input type="checkbox" id="javascript" value="JavaScript" tp-
model="checkedLanguages">
  <label for="javascript">JavaScrpit</label>
  <input type="checkbox" id="go" value="Go" tp-model="checkedLanguages">
  <label for="go">Go</label>
  <br>
  <span>I choose: {{ checkedLanguages | json }}</span>
</div>
```

Which will render

You can pick many languages!
. Python . JavaScrpit . Go
I choose: {{ checkedLanguages | json }}

Radio

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    picked: ''
  }
})
```

HTML

```
<div id="app">
  <span>You can pick only one language!</span>
  <br>
  <input type="radio" id="python" value="Python" tp-model="picked">
  <label for="python">Python</label>
  <br>
  <input type="radio" id="javascript" value="JavaScript" tp-
model="picked">
  <label for="javascript">JavaScript</label>
  <br>
  <input type="radio" id="go" value="Go" tp-model="picked">
  <label for="go">Go</label>
  <br>
  <span>I want to study: {{ picked }}</span>
</div>
```

Which will render

You can pick only one language!
. Python
. JavaScript
. Go
I want to study: {{ picked }}

Select

Single select:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    selected: ''
  }
})
```

HTML

```
<div id="app">
<span>You can select one</span>
<br>
<select tp-model="selected">
  <option selected>PHP</option>
  <option>Ruby</option>
  <option>C#</option>
</select>
<span>Selected: {{ selected }}</span>
</div>
```

Which will render

You can select one
Selected: {{ selected }}

Multiple select:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    selected: ''
  }
})
```

HTML

```
<div id="app">
<span>You can select some of them</span>
<br>
<select tp-model="selected" multiple>
  <option selected>PHP</option>
  <option>Ruby</option>
  <option>C#</option>
</select>
<span>Selected: {{ selected | json}}</span>
</div>
```

Which will render

You can select some of them
Selected: {{ selected | json}}

Value Bindings

You may notice that for radio, checkbox, and select options, the binding values the `tp-model` are strings, or booleans if we're talking about checkboxes. For example:

```
<!-- `picked` is a string "Python" when checked -->
<input type="radio" tp-model="picked" value="Python">
<!-- `toggle` is either true or false -->
<input type="checkbox" tp-model="toggle">
<!-- `selected` is a string "JavaScript" when selected -->
<select tp-model="selected">
  <option value="JavaScript">JavaScript</option>
</select>
```

Anyway, sometimes we want to bind a dynamic property on a Tulipan instance. We can do this using `tp-bind` , besides this option allows binding the input value to non-string values.

Checkbox

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    checked: false,
    a: true,
    b: false
  }
})
```

HTML

```
<div id="app">
  <input type="checkbox" id="checkbox" tp-model="checked" tp-bind:true-
value="a" tp-bind:false-value="b">{{checked}}
</div>
```

Which will render

.{{checked}}

Radio

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    pick: '',
    a: 'Apple',
    b: 'Banana',
    c: 'Cherry'
  }
})
```

HTML

```html
<div id="app">
  <span>Fruits</span>
  <input type="radio" tp-model="pick" tp-bind:value="a">{{a}}
  <input type="radio" tp-model="pick" tp-bind:value="b">{{b}}
  <input type="radio" tp-model="pick" tp-bind:value="c">{{c}}
    <br>
  <span>I want to buy: {{ pick }}</span>
</div>
```

Which will render

```
Fruits: .{{a}} .{{b}} .{{c}}
I want to buy: {{ pick }}
```

Select Options

We can even combine `tp-bind:value` directive with `tp-for` directive to dynamically render the options as follows:

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data: {
    selected:'Python',
    options: [
      {language: 'PHP'},
      {language: 'Python'},
      {language: 'C#'}
    ]
  }
})
```

HTML

```html
<div id="app">
<span>You can select one</span>
<br>
<select tp-model="selected">
  <option tp-for="option in options" tp-
```

```
bind:value="option.language">{{option.language}}</option>
</select>
<span>Selected: {{ selected }}</span>
</div>
```

Which will render

You can select one
Selected: {{ selected }}

Param Attributes

Lazy

You can add the lazy attribute to make sure the `tp-model` syncs the input with the data after change events:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    message: ''
  }
})
```

HTML

```
<div id="app">
  <span>This is a lazy message: {{ message }}</span>
  <br>
  <input type="text" tp-model="message" lazy placeholder="you can edit
me!">
</div>
```

Which will render

This is a lazy message: {{ message }}

.

Number

This attribute allows numbers only as inputs:

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    age: 0
  }
})
```

HTML

```
<div id="app">
  <span>Please insert your age: {{ age }}</span>
  <br>
  <input tp-model="age" number>
</div>
```

Which will render

Please insert your age: {{ age }}

.

Debounce

This param makes it possible to set a minimum delay after each keystroke before the input is synced to the model:

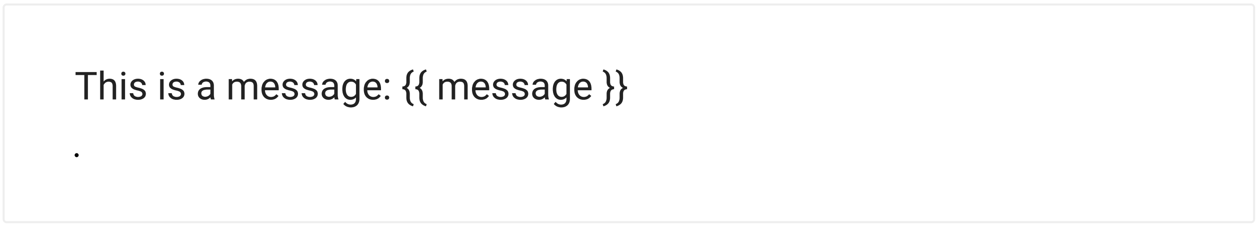
JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    message: ''
  }
})
```

HTML

```
<div id="app">
  <span>This is a message: {{ message }}</span>
  <br>
  <input type="text" tp-model="message" debounce="500" placeholder="you
can edit me!">
</div>
```

Which will render



This is a message: {{ message }}

.

Properties

Tulipan.js lets you use computed properties to make any logic that requires more than one expression. This is very convenient because in order to avoid performance issues Tulipan.js limits binding expressions to only one, which could be restrictive.

Computed Properties

With Tulipan.js we can declare computed properties.

Let's declare this one called `myDadAge` , which is 35 by default, the age my dad was when I was born. He gets older as I do, so to calculate his current age we can use a getter function for the property `vm.myDadAge` :

JavaScript

```
var vm = new Tulipan({
  el: '#app',
  data: {
    myAge: 20
  },
  computed: {
    // a computed getter
    myDadAge: function () {
      // `this` points to the vm instance
      return this.myAge + 35
    }
  }
})
```

HTML

```
<div id="app">
  I am {{ myAge }} years old, and my dad is {{ myDadAge }} years old.
</div>
```

Which will render

```
I am {{ myAge }} years old, and my dad is {{ myDadAge }} years old.
```

Tulipan is aware that `vm.myDadAge` depends on `vm.myAge`, so it will update any bindings that depends on `vm.myDadAge` when `vm.myAge` changes.

Comparing Computed Property with $watch

When you have some data that changes based on other data, one idea could be to use `$watch`, which is an API method provided by Tulipan.js that lets you observe data changes in a Tulipan instance. However, we highly recommend using computed properties instead. Consider the following example:

JavaScript

```
var vm = new Tulipan({
  el: '#app',
  data: {
    temperatureInCelsius: 24,
    temperatureInFahrenheit: 75
  }
})
vm.$watch('temperatureInCelsius', function (val){
  this.temperatureInFahrenheit = val*1.8 + 32
})
```

This code is imperative and repetitive. Let's compare it with a computed property version:

JavaScript

```
var vm = new Tulipan({
  el: '#app',
  data: {
    temperatureInCelsius: 24,
  },
  computed: {
    temperatureInFahrenheit: function(){
        return this.temperatureInCelsius*1.8 + 32
      }
    }
})
```

HTML

```
<div id="app">
  The temperature is {{ temperatureInFahrenheit }}°F.
</div>
```

Which will render

The temperature is {{ temperatureInFahrenheit }}°F.

A lot better, isn't it?

Computed Setters

Computed properties are by default getter-only, but you can also provide a setter when you need it:

JavaScript

```
// ...
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
// ...
```

When `vm.fullName = ‘Daniel Smith’` is called, the setter will be called, and `vm.firstName` and `vm.lastName` will be updated accordingly. Since the get method of `fullName` depends on those, it will be recalculated.

User Interaction

In many cases when we're building a dynamic website it should have the ability to respond to *events*. It could be from a button clicked by the user till submits a form, these can be handled with Tulipan using the **tp-on** directive.

Method Handler

Tulipan provides the `tp-on` directive to listen to DOM events:

JavaScript

```
 var vm = new Tulipan({
  el: '#app',
  data: {
    name: 'Tulipan.js'
  },
  methods: {
    sayHello: function (event) {
      alert('Hello! welcome to ' + this.name + '!');
      alert(event.target.tagName);
    }
  }
})
```

HTML

```
<div id="app">
    <button tp-on:click="sayHello">Touch this</button>
</div>
```

Wich will render

Touch this

We can also bind inline JavaScript statements instead of a method name:

JavaScript

```
new Tulipan({
  el: '#app',
  methods: {
    print: function (msg) {
      alert(msg)
    }
  }
})
```

HTML

```
<div id="app">
  <button tp-on:click="print('Tulipan is great!')">Push me first!</
button>
  <br>
  <button tp-on:click="print('Thanks for using it!')">Push me</button>
</div>
```

Which will render

Push me first!
Push me

Event Modifiers

Sometimes we need to have access to the original DOM event, we do this using the `$event` variable passed into a method, although it could be inconvenient:

JavaScript

```
new Tulipan ({
    el: '#app',
    methods: {
        print: function (msg, event) {
          // now we have access to the native event
```

```
            event.preventDefault()
          }
      }
})
```

HTML

```
<div id="app">
<button tp-on:click="print('hello!', $event)">Touch this</button>
<div>
```

Tulipan.js provides two event modifiers for `tp-on: .prevent` and `.stop`, which replace the use (inside event handlers) of `event.preventDefault()` and `event.stopPropagation()`, respectively:

```
<!-- the click event's propagation will be stopped -->
<a tp-on:click.stop="print('something')"></a>
<!-- the submit event will no longer reload the page -->
<form tp-on:submit.prevent="onSubmit"></form>
<!-- modifiers can be chained -->
<a tp-on:click.stop.prevent="print('something else')">
<!-- just the modifier -->
<form tp-on:submit.prevent></form>
```

Key Modifiers

When listening for keyboard events, we often need to check for key codes. Instead of remembering all of them, a better choice is the use of aliases, so Tulipan.js provides them for the most commonly used keys:

```
<!-- only call vm.submit() when the keyCode is 13 -->
<input tp-on:keyup.13="submit">
<!-- same as above -->
<input tp-on:keyup.enter="submit">
```

Here’s the full list of key modifier aliases:

- enter
- tab

- delete (captures both “Delete” and, if the keyboard has it, “Backspace”)
- esc
- space
- up
- down
- left
- right

Renderings

With Tulipan you can render any data type using the magic words reserved for development. In this guide you can find from simple rendering to conditional rendering.

List Renderings

Tulipan carries custom directives so you can apply a bunch of things to the DOM. One directive is **tp-for** which allows you to render sequential elements.

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    cocktails: [
        "Bronx",
        "Daiquiri",
        "Manhattan",
        "Tom Collins",
        "Piña Colada"
    ]
  }
})
```

HTML

```
<div id="app">
  <ul>
    <li tp-for="cocktail in cocktails">
      {{ cocktail }}
    </li>
  </ul>
</div>
```

Which will render

- {{ cocktail }}

You can also provide an argument to render the index of the element.

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    cocktails: [
        "Bronx",
        "Daiquiri",
        "Manhattan",
        "Tom Collins",
        "Piña Colada"
    ]
  }
})
```

HTML

```
<div id="app">
  <ul>
    <li tp-for="(index, cocktail) in cocktails">
      {{ index }} - {{ cocktail }}
    </li>
  </ul>
</div>
```

Which will render

- {{ index }} - {{ cocktail }}

This way you can use it to perform user actions in a specific element.

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    cocktails: [
        "Bronx",
        "Daiquiri",
        "Manhattan",
        "Tom Collins",
        "Piña Colada"
    ]
  },

  methods: {
    display: function(index){
      alert(this.cocktails[index]);
    }
  }
})
```

HTML

```
<div id="app">
  <ul>
    <li tp-for="(index, cocktail) in cocktails">
      {{ index }} - {{ cocktail }} (<span tp-
on:click="display(index)"><b>Click Me!</b></span>)
    </li>
  </ul>
</div>
```

Which will render

- {{ index }} - {{ cocktail }} (**Click Me!**)

Conditional Rendering

Tulipan.js makes it possible to use conditional rendering to toggle the presence of any element in the DOM based on certain conditions. It uses **tp-if**, **tp-show**, and **tp-else** for this purpose.

tp-if and tp-else

The `tp-if` can be used to conditionally render elements or blocks, you can use it to assign boolean variables to toggle elements in the DOM based on their value as follows:

JavaScript

```
new Tulipan({
  el: '#app',
  data:{
    yes: true
  }
})
```

HTML

```
<div id="app">
  <span>Hi! Can you see me?</span>
  <br>
  <strong tp-if="yes">Yes, I can see you!</strong>
</div>
```

Which will render

Hi! Can you see me?
Yes, I can see you!

The `tp-else` directive can be used to render a block that does not satisfy the condition of the `tp-if` directive. To work, this directive must immediately follow the `tp-if` directive. Let's check out this example:

JavaScript

```
new Tulipan({
  el: '#app',
  data:{
    yes: false
  }
})
```

HTML

```
<div id="app">
  <span>Hi! Can you see me?</span>
  <br>
  <strong tp-if="yes">Yes, I can see you!</strong>
  <strong tp-else>No, I can't see you!</strong>
</div>
```

Which will render

Hi! Can you see me?
Yes, I can see you! No, I can't see you!

Sometimes you may want to toggle more than one element, but `tp-if` has to be attached to a single element. So in those cases, you can use `tp-if` on a `<template>` element as follows:

JavaScript

```
new Tulipan({
  el: '#app',
  data:{
    showTemplate: true
  }
})
```

HTML

```html
<div id="app">
  <template tp-if="showTemplate">
    <h1>Hi! This is a hidden template</h1>
    <p>You can see me because showTemplate is true</p>
  </template>
</div>
```

Which will render

tp-show

The directive `tp-show` is another option for conditionally displaying elements, its usage is pretty much the same as `tp-if` as you may expect:

JavaScript

```javascript
new Tulipan({
  el: '#app',
  data:{
    yes: true
  }
})
```

HTML

```html
<div id="app">
  <span>Did you say hello?</span>
  <br>
  <strong tp-show="yes">Yes, I say hello!</strong>
</div>
```

Which will render

Did you say hello?
Yes, I say hello!

tp-if or tp-show?

Despite the usage is almost the same, `tp-if` and `tp-show` have differences you have to know to use them properly.

When using `tp-if`, Tulipan.js performs a partial compilation/teardown process because the template content inside `tp-if` can also contain data bindings or child components. Such a process ensures that these elements are properly destroyed and re-created during toggles.

The directive `tp-if` is also lazy, which means if the initial condition is false on the initial render, the partial compilation won't start until it becomes true for the first time. The compilation will be cached subsequently.

In contrast, `tp-show` is much simpler since the element is always compiled and preserved.

Overall, `tp-if` has higher toggle costs while `tp-show` has higher initial render costs. It's recommended to use `tp-show` if you have to toggle something frequently, otherwise `tp-if`.

Filters

Data can be unprocessed and raw, and you might want to render data in a different format, filters allow you to process data right before rendering.

Let's take a look at the following example.

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    percents: [38.564, 49.242, 80.231],
    smalls: [0.000231, 0.000034, 0.00000045, 0.2],
    calories: [200, 500, 700, 150]

  },
  methods: {
  },

  filters: {
    sum: function (value) {
      return value.reduce(function(a,b){
        return a + b
      }, 0);
    },

    scientific: function (value) {
      if (value != null){
        return value.toExponential(2);
      }
      return value;
    },

    percent: function (value) {
      if (value != null){
        return value.toFixed(2) + " %";
      }
      return value;
    }
  }
})
```

HTML

```html
<div id="app">
  Percents:
  <ul>
    <li tp-for="value in percents">
      {{ value | percent }}
    </li>
  </ul>
  Scientific:
  <ul>
    <li tp-for="value in smalls">
      {{ value | scientific }}
    </li>
  </ul>
  Sum:
  <ul>
    <li>{{ calories | sum }}</li>
  </ul>
</div>
```

Which will render

Percents:

- {{ value | percent }}

Scientific:

- {{ value | scientific }}

Sum:

- {{ calories | sum }}

As you can see in the JavaScript code, you can define you own filters using custom defined functions in the *filters* properties for the Tulipan instance.

You can also pipe your filters together using the pipe | operator, this will send the output of one filter the input of the next filter.

Web Requests

Our data in a Single Page Application often comes from a back-end service or RESTful API, or any other HTTP service. In order to retrieve information from any web source, Tulipan.js has integrated a small but yet complete library to perform asynchronous requests. Let's practice with a simple GET requests to show dogs images.

GET from Dog API

HTML

```
<div id="app">
  <button tp-on:click="fetchDog()">Click Me!</button>
  <br>
  <img src="{{ dog_url }}" alt="Dog">
</div>
```

JavaScript

```
new Tulipan({
  el: '#app',
  data: {
    dog_url: 'https://images.dog.ceo/breeds/mexicanhairless/
n02113978_2508.jpg'
  },
  methods: {
    fetchDog: function () {
      this.$http.get('https://dog.ceo/api/breeds/image/random')
        .then(function (res){
          this.$set("dog_url", res.data.message);
        }, function(err){
          console.log(err);
        })
    }
  }
})
```

Which will render

```
Click Me!
Dog
```

Routing

Another important feature in Single Page Application development is routing, this client-side routing allows users to navigate through our application where no full page reload will happend, this way you can use *$http* service to fetch partial information from the server, you can also use history buttons without full page reload.

The routing configuration is done in the istance configuration object, but the navigation is done through the *$router.navigate* service.

A simple SPA with Routing

JSONPlaceholder is a Fake Online REST API for Testing and Prototyping and we will use to show its entities in a SPA.

We will use the following resources from JSONPlaceholder.

Resource	Description	url
posts	100 posts	https://jsonplaceholder.typicode.com/posts
comments	500 comments	https://jsonplaceholder.typicode.com/comments?postId=1
todos	200 todos	https://jsonplaceholder.typicode.com/todos
users	10 users	https://jsonplaceholder.typicode.com/users

Since these resources have relations, we can create a simple interface to navigate through these relations, like posts and comments related to posts, or posts related to users.

HTML

```html
<div id="app" class="tabset">
  <!-- Tab 1 -->
  <input type="radio" name="tabset" id="tab1" aria-controls="posts"
checked>
  <label for="tab1" tp-on:click="navigate('')">Posts</label>
  <!-- Tab 2 -->
  <input type="radio" name="tabset" id="tab2" aria-controls="todos">
  <label for="tab2" tp-on:click="navigate('todos')">Todos</label>
  <!-- Tab 3 -->
  <input type="radio" name="tabset" id="tab3" aria-controls="users">
  <label for="tab3" tp-on:click="navigate('users')">Users</label>
</div>

<router-view></router-view>

<section id="posts" class="tab-panel">
  <h2>Posts</h2>
  <ul>
    <li tp-for="post in posts">{{ post.title }}</li>
  </ul>
</section>
<section id="todos" class="tab-panel">
  <h2>Todos</h2>
  <ul>
    <li tp-for="todo in todos">{{ todo.title }}</li>
  </ul>
</section>
<section id="users" class="tab-panel">
  <h2>Users</h2>
  <ul>
    <li tp-for="user in users">{{ user.name }}({{ user.username}}) -
{{ user.email }}</li>
  </ul>
</section>
```

JavaScript

```javascript
// navigation app
new Tulipan({
  el: '#app',
  data: {
  },
  methods: {
    navigate: function (page) {
      this.$router.navigate("/" + page);
    }
  }
```

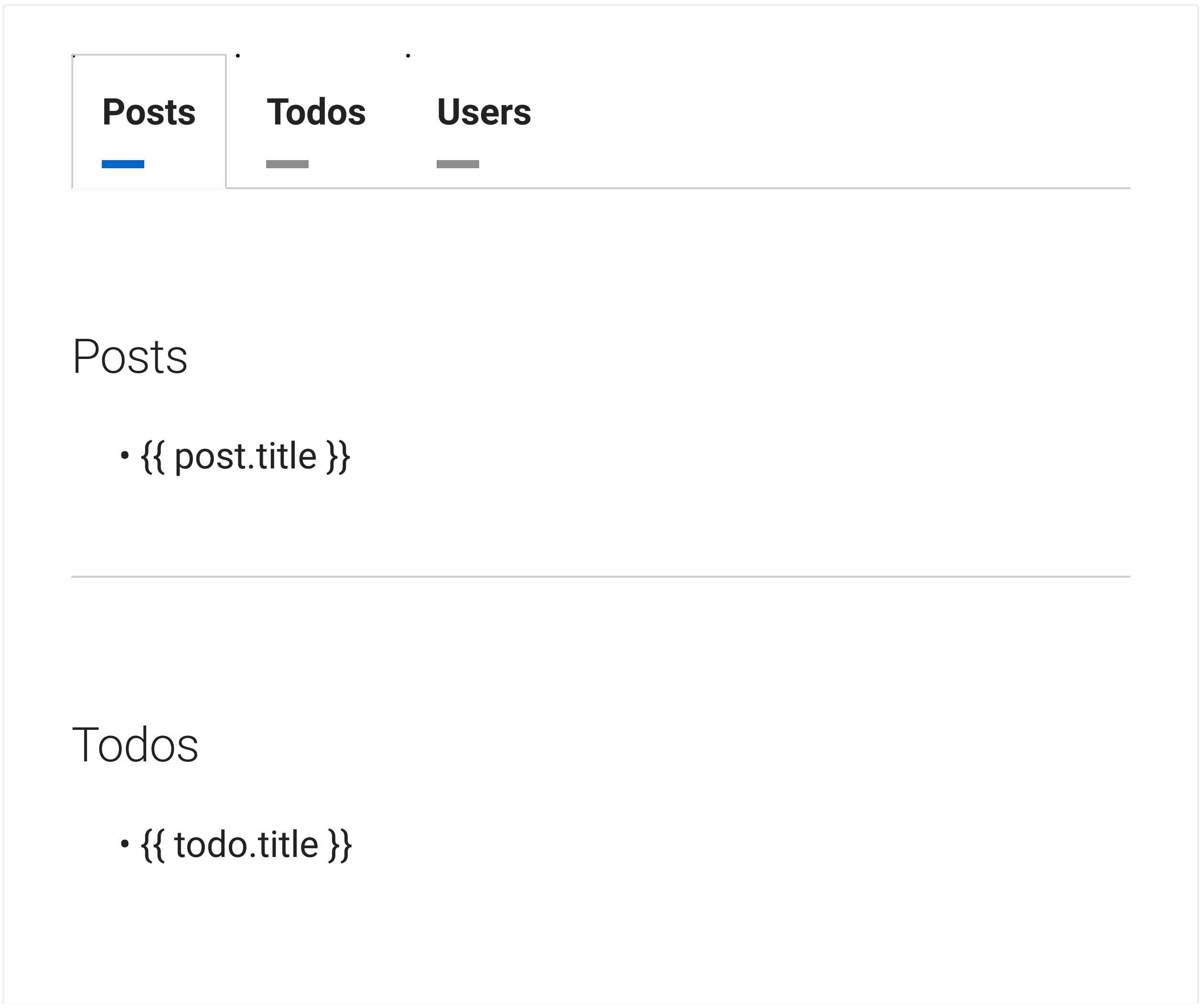
```
})

// posts app
new Tulipan({
  el: '#posts',
  route: "/",
  data: {
    posts: []
  },
  methods: {
    after: function(){
      this.fetchPosts();
    },
    fetchPosts: function () {
      this.$http.get('https://jsonplaceholder.typicode.com/posts')
        .then(function (res){
          this.$set("posts", res.data);
        }, function(err){
          console.log(err);
        })
    }
  }
})

// todos app
new Tulipan({
  el: '#todos',
  route: "/todos",
  data: {
    todos: []
  },
  methods: {
    after: function(){
      this.fetchTodos();
    },
    fetchTodos: function () {
      this.$http.get('https://jsonplaceholder.typicode.com/todos')
        .then(function (res){
          this.$set("todos", res.data);
        }, function(err){
          console.log(err);
        })
    }
  }
})

// users app
new Tulipan({
  el: '#users',
  route: "/users",
  data: {
    users: []
  },
  methods: {
    after: function(){
      this.fetchUsers();
    },
    fetchUsers: function () {
      this.$http.get('https://jsonplaceholder.typicode.com/users')
        .then(function (res){
          this.$set("users", res.data);
        }, function(err){
          console.log(err);
        })
    }
  }
})
```

Which will render

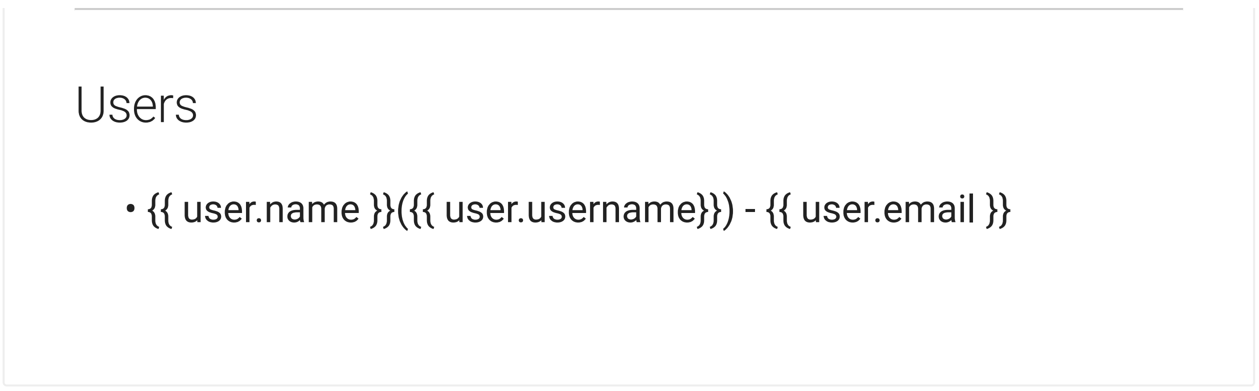


Posts

- {{ post.title }}

Todos

- {{ todo.title }}



Users

- {{ user.name }}({{ user.username}}) - {{ user.email }}

Adding subroutes

Each resource display a list of elements in which each element has its own detailed data and information. The idea now is to display a post body a its comments.

Let's add a new section to show post details

HTML

```
<section id="post-detail" class="tab-panel">
  <h2>Post</h2>
  <b>{{ post.title }}</b>
  <p>{{ post.body }}</p>
</section>
```

Let's create a new Tulipan instance to handle this template

JavaScript

```
new Tulipan({
  el: '#post-detail',
  route: "/posts/:postId",
  data: {
    post: {}
  },
  methods: {
    after: function(params){
      var postId = params.postId;
      this.fetchPost(postId);
    },
    fetchPost: function (postId) {
      this.$http.get('https://jsonplaceholder.typicode.com/posts/' +
postId)
```

```
        .then(function (res){
          this.$set("post", res.data);
        }, function(err){
          console.log(err);
        })
    }
  }
})
```

Let's add a link to navigate to each post

HTML

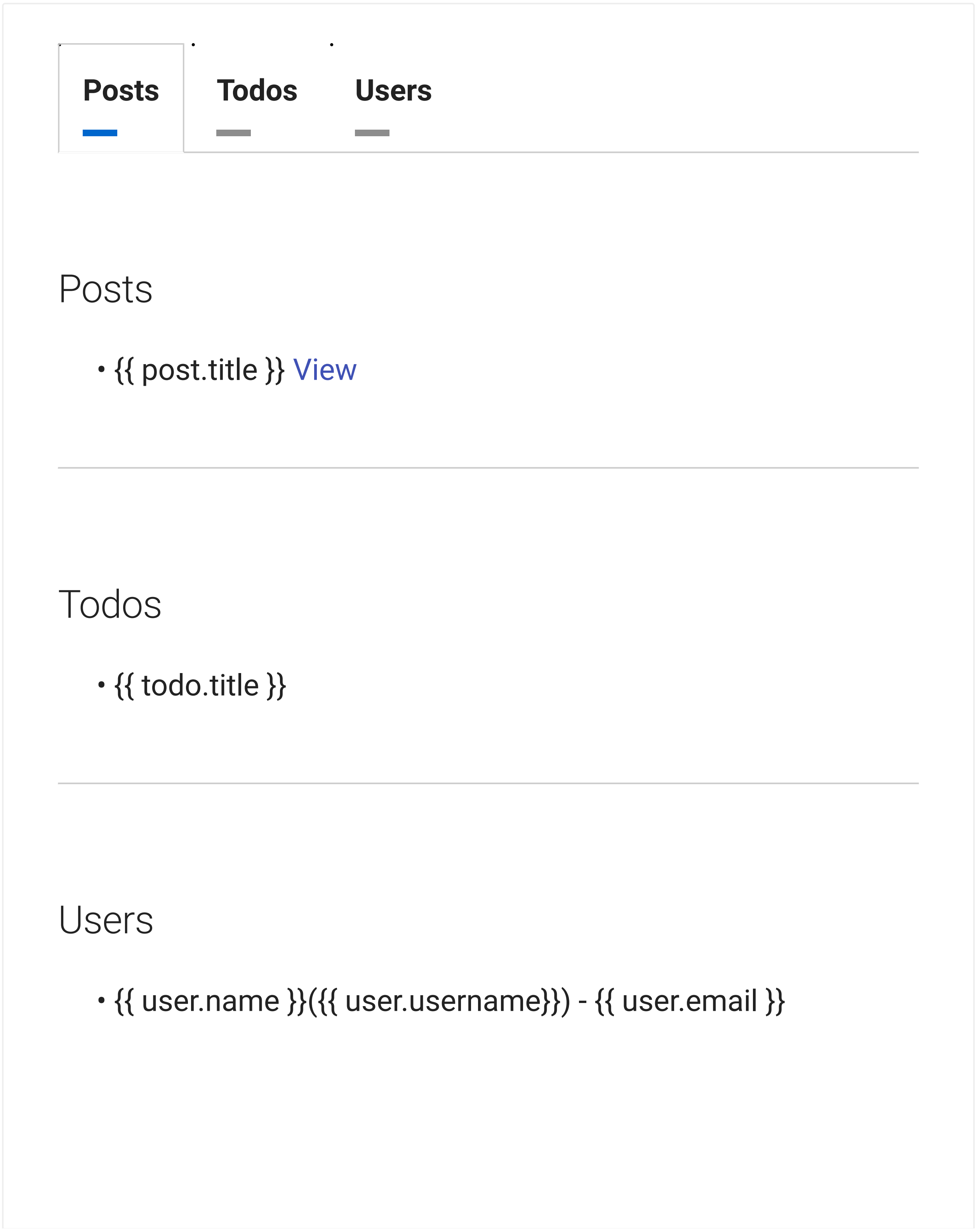
```html
<section id="posts" class="tab-panel">
  <h2>Posts</h2>
  <ul>
    <li tp-for="post in posts">
    {{ post.title }}
    <a href="javascript:void(0)" tp-on:click="viewPost(post.id)">View</a>
    </li>
  </ul>
</section>
```

and a fix to the posts JavaScript application in order to navigate on click.

JavaScript

```javascript
new Tulipan({
  el: '#posts',
  route: "/",
  data: {
    posts: []
  },
  methods: {
    after: function(){
      this.fetchPosts();
    },
    fetchPosts: function () {
      this.$http.get('https://jsonplaceholder.typicode.com/posts')
        .then(function (res){
          this.$set("posts", res.data.slice(0, 20));
        }, function(err){
          console.log(err);
        })
    },
    viewPost(index){
      this.$router.navigate("/posts/" + index);
    }
  }
})
```

All these will render the following.



Posts

- {{ post.title }} View

Todos

- {{ todo.title }}

Users

- {{ user.name }}({{ user.username}}) - {{ user.email }}

Post

{{ post.title }}

{{ post.body }}

Handling Parameters

You can handle parameters inside the *after* special methods buy supplying the first argument.

JavaScript

```
new Tulipan({
  ...
  route: "/posts/:postId",
  data: {
    post: {}
  },
  methods: {
    after: function(params){
      var postId = params.postId;
    },
    ...
  }
})
```

the *params* argument is an object with property names equal to the format provided in the *route* option.

Handling Query Strings

You can also handle query strings inside the *after* special method buy supplying a second argument.

JavaScript

```
new Tulipan({
  ...
  route: "/posts/:postId",
  data: {
    post: {}
  },
  methods: {
    after: function(params, query){
      // If we have /posts/12?sortby=inc as a url then
      var postId = params.postId;
      // query = sortby=inc
    },
    ...
  }
})
```

Storing

Tulipan.js let's you store data in the browser, the basic function is to store key/value pairs that you can associate with your application session or state. Primary use of browser storage is to reduce overloading the server with http requests.

Simple usage

A Tulipan instance exposes store through the *$store* especial property, exposing a simple API for cross-browser local storage:

```
vm = new Tulipan({});

// Store current user
vm.$store.set('user', { name:'Napoleon' });

// Get current user
vm.$store.get('user');

// Remove current user
vm.$store.remove('user');

// Clear all keys
vm.$store.clearAll();

// Loop over all stored values
vm.$store.each(function(value, key) {
  console.log(key, '==', value)
})
```

It is recommender to use it inside your web requests to check for data expiration.