Date Picker

webtool.one

Source files

File name	Description
`main.js`	`DTPicker` prototype. This file should be used to import picker.
`options.js`	Options for picker, incl. params for internal use.
`dt-picker.vue`	The main Vue component.
`src/_state.js`	WebAssembly operations.
`src/accessors.js`	`DTPicker` prototype accessors.
`src/actions.js`	Action methods: clear, confirm, etc.
`src/attrs.js`	Getter for HTML attributes.
`src/constants.js`	Operations to set constant values.
`src/events.js`	Event functionality.
`src/init-check.js`	Some checks are performed on `DTPicker` initialization phase.
`src/observers.js`	Observers to react to HTML updates.
`src/ops.js`	Internal operations.
`src/validate-obj.js`	Helper function to perform initial checks.
`src/helpers.js`	Helper functions.
`src/vue-directives.js`	Vue directives.
`src/svghtml.vue`	Vue component for SVG rendering.
`src/dt-controls-block-icon.vue`	Vue component for picker's controls icons.
`src/dt-level.vue`	Vue component for date level.
`src/dt-selected-block.vue`	Vue component for selected block.
`src/dt-selected.vue`	Vue component for rendering selected date(s).

In addition, there are `icons`, `css`, and `wasm` directories for icons, styles, and compiled WebAssembly, respectively.

Usage

To use the picker, you need a 1) reactive presentation HTML layer and 2) initialized `DTPicker` instance for operations.

With Vue implementation:

```
<template>
    <!-- NOTE: update model on event `update:model`, v-model will not work! -->
    <DatePicker v-bind='props' @update:model='model = $event' />
</template>

<script setup>
    import { ref } from 'vue'
    import DatePicker from 'date-picker/js/dt-picker.vue'

    const initial = { years: [], months: [], days: [] }

    /* Options */
    const props = ref({ model: initial, /* ... options ... */ })

    /* Model with initial values */
    const model = ref(initial)

    /* ... do something with model ... */
</script>
```

With other framework or plain HTML/JavaScript:

```
import DTPicker from 'date-picker/main'

/* Create state object.
   Reference to it shoud be passed to DTPicker constructor.
   It is updated after each picker operation.
   Object properties depend on your approach,
   see `date-picker/dt-picker.vue for details`
*/
const state = { ... }

const options = { ... }

/* Instantiate DTPicker instance: pass options and reference to state. */
const picker = new DTPicker(options, state)

/* ... react to `state` changes and update HTML ... */
```

Please see additional information below.

Architecture

Isolated logic

This functionality is made to be almost framework/presentation layer agnostic. This means that all logic is independent of HTML rendering, which includes the following features:

- HTML attributes (dynamic element classes, IDs, HTML Symbols, etc.) are separated from the presentation layer and provided to it on each update (when `DTPicker` state changes);
- To react to HTML updates standard JavaScript Observer API is used;
- To update the presentation layer after respective state changes, the special method `DTPicker.update` is used. After all, such an approach gives strict control over when display layer update should be performed theoretically providing more performant execution (touch HTML only when all necessary operations are done). Also, the resulting code should be less error-prone;
- Code is divided into separate blocks: WebAssembly operations, HTML attributes, constants, actions, text, etc.

Due to these points to use `DTPicker` in any project you need to:

1. Create a presentation, i. e. HTML, layer: it can be any reactive JavaScript framework or even plain HTML with vanilla JavaScript;
2. Connect presentation layer with logic, i. e. pass state to instantiated `DTPicker` instance and update HTML according to received result.

If you don't mind using Vue.js, it's already implemented for the presentation layer and picker is ready for immediate usage.

If you're adapting the presentation layer for some particular framework, please check `.vue` template files to create proper HTML elements.

Parameters for internal use, i. e. directly in the logic, are separated for convenience: check it also for satisfactory values.

For the styling guide please see the appropriate section of this docs. Please be aware that in HTML there are some elements (not so much so far) that have hard-coded attributes: these should be consistent with styles and `DTPicker.attrs` getter (source file `src/attrs.js`).

HTML elements

In general, HTML elements are intended for internal use only. However, in case you need them, all HTML elements are accessible via `DTPicker.refs.<element name>`, e. g. `DTPicker.refs.root`. For reference please see source files. Also please note, that some of these `DTPicker.refs` might be refreshed after the picker gets updated.

Classes (prototypes)

DTPicker

In Vue.js implementation it's directly accessible from outside the picker as component `ref`, e. g. in your component use: `<imported picker component name>.value.picker`. The picker instance is exposed via `defineExpose`, so to access the picker instance you need to get the component containing it first.

To use `DTPicker` as a standalone object you need to instantiate it and provide proper HTML for the representation layer. The most obvious way to do it is to port HTML from the Vue component, i. e. `dt-picker.vue`.

Please refer to the source files for details.

WebAssembly

The early version of `DTPicker` was implemented with the Day.js library, but after some testing timing results were not very good: changing date took about 40-70 ms (many optimizations, incl. caching, were tried out). Although it's not so awful, finally WebAssembly was used to obtain more performant operations. And now, in the current version timings are more promising: changing date takes only about 2-7 ms.

Date selection

Selection logic

Picker allows us to select all, several, or any date level, i. e. we can select only year, only day, year-month, month-day, etc. Such an approach makes possible usage of picker as, for instance, filtering form field: we will have the ability to filter data by only one date level (for example, we can filter data only by year if month and day aren't of interest for us).

Due to the number of days in months having a variance, displayed day numbers depend on the selected month (in the case of February, a year also matters, because there can be a leap year). If no month is selected, the days' level will display 31 items to give us ability to select any day number.

In case the newly selected month doesn't have an already selected day number, the latter will be unselected. For example, if we select day #31 and then select November, then the day will be unselected because there are only 30 days in November.

This logic also applies to years: for instance, if we select February 29 and the leap year, all will be fine. But, if we change the year to a non-leap year, we will have no selected day and number of days equal to 28.

When clicking on the already selected item, this item will be unselected.

Modes

There are two modes to select dates:

1. **Normal**: only one full date or any one level of date;
2. **Edges**: full start and end dates or any date level of start and end.

All modes allow to select any or all date levels, e. g. only years or only months can be selected.

To enable edges mode supply as initial model two values instead of one, e. g. `years = [2020,2021]`. The active (available for selection) edge is set to 0 by default, i. e. start date. But if the initial value only at index 1 is `null` or `undefined` the active edge will be set to 1, i. e. end date.

To sum it up:

1. If no mode is designated in options and there are no initial values or only one value for each date level present, it will be 'normal' mode. Example of initial values: `{ years: [2020], months: [], days: [11] }`;
2. If no mode is designated in options and any of initial date levels has 2 values, the initial mode will be edges. In such case, when any of the date levels has `null` or `undefined` at index 1, then the edge will be set to 1 (end date), otherwise - to 0 (start date). Example of initial values: `{ years: [2020, 2022], months: [10], days: [11, null] }`;
3. If mode is designated in options then it will be used with one exception: two initial values in any date level will result in edges mode in any case.

If you want the initial mode to be edges, set option `mode = 'edges'`. Note that in this case, all date levels must have at most 2 values. If you need to set an empty level for any edge, set it to `null`, e. g. `{ years: [2020, 2025], months: [null, 10], days: [] }`. If the initial start date is full and the initial end date isn't full, then the active edge will be switched to the end date.

Active mode can be changed at any moment by clicking on the settings button.

Actions

Picker has the following actions:

1. **Clear**. Remove any selected values;
2. **Revert**. Set selected values to the initial state. This works only if there were initial values on picker initialization;
3. **Confirm**. Convenient action to emit special event to confirm selection is done. May be useful for form confirmation;

There are two not-so-obvious actions:

1. **Blocking** entire picker: activated on setting `options.blocked` to `true` in case of component reactivity (for example, when Vue component is used). Can be invoked directly with respective methods, i. e. `DTPicker.block()` and `DTPicker.unblock()`;
2. **Setting stub**: activated when no start and/or end date is provided. May be useful to use stub when loading data. Also, stub can be get/set on-demand via Boolean getter/setter `DTPicker.stub`.

Options

To be initialized, the picker needs minimum (start) and maximum (end) dates to set the range for selection:

`options.minDT = Number | String`

`options.maxDT = Number | String`

Examples: `2020 (Number) | '2020-01-05' (String) | '2020' (String)`

These options are needed to retrieve year only, so the provided string will be parsed for a 4-digit number. In the case of number, it will be used as is. The start and end dates will be respective start-end of provided years, e. g. start date `'2020-01-05'` will be converted to `2020-01-01`. Only 4-digit years are expected.

If no start and end dates are provided, a stub will be used instead of picker.

To ensure proper initial values and parameters, some checks are performed on the picker initialization phase.

Option	Type	Default	Description
`model`	`Object{ years: Array<Number>, month: Array<Number>, days: Array<Number> }`	`{ years: [], months: [], days: [] }`	Initial selected values.
`minDT`	`Number | String`	`null`	Start year.
`maxDT`	`Number | String`	`null`	End year.
`id`	`String`	`'date-picker'`	ID of `DTPicker`.
`blocked`	`Boolean`	`false`	Whether to block (disable) entire component.
`locale`	`String`	`'en'`	Locale: `'ru'` or `'en'`. To add more locales add respective texts to `params` in `options.js`.
`monthNames`	`String: 'name' | 'number'`	`'name'`	How to display months: name or number.
`monthNums`	`Boolean`	`true`	Whether to display month nums. Implicitly disabled if `monthNames = 'number'`.
`clean`	`Boolean`	`true`	Whether to clean output model, i. e. remove any empty (`null`) values. Most useful for edges mode, e. g. `years: [2020, null]` or `[2020]`.
`mode`	`String`	`null`	`'normal' | null | undefined` : single date or 'normal', e. g. `{ years: [2020], months: [2], days: [1] }` (2020-02-01), `'edges'` : interval - , e. g. `{ years: [2020, 2022], months: [1, 2], days: [10, 22] }` (2020-01-10 <-> 2022-02-22).
`changeQ`	`Object{ years: Number, month: Number, days: Number }`	`{ years: 2, months: 2, days: 2 }`	In case of level overflow amount of items to be moved on navigation.
`showModeIcons`	`Boolean`	`true`	Whether to show mode icon in selected area.
`windowResize`	`Boolean`	`false`	Whether to resize picker on window resize event.
`blockedCSSfilter`	`String`	`'opacity(.5) saturate(10%)'`	CSS `filter` property for blocked picker.
`navAnimation`	`Object`	`{ duration: 100, easing: 'ease-out' }`	Animation of moving level on navigation.
`blockAnimation`	`Object`	`{ duration: 100, easing: 'ease-out' }`	Animation for picker blocking.

Option	Type	Default	Description
`resizeAnimation`	`Object`	`{ duration: 100, easing: 'ease-out' }`	Fade animation for correction operations after resize. A one half of full cycle, i. e. in and out.
`levelAnimation`	`Object`	`{ duration: 100, easing: 'ease-out' }`	Animation of correcting level width when picker was resized or days Qty changed.

All animation options comply with the Web Animation API.

In addition, there are some options for internal use. Please see source file `options.js` for details.

Events

The following events are intended for the usage from outside of picker instance. For internal events' details please see source files.

Emits

All events have names constructed from picker ID plus event name: `<picker id>:<event name>`.

Event name	Description	Payload
`confirm`	Emitted when confirming selected data.	Selected data, picker ID, mode in use.
`update:model`	Emitted when selected data changes.	Selected data, picker ID, mode in use.
`compact`	Emitted when compact mode has been changed.	Boolean indicating whether compact mode is enabled.

You can add all emits you need by using `DTPicker.emit` method. Please refer to source files, especially `src/events.js`, for details.

Listeners

Only `<picker id>:resize` event is listened by picker root element.

Adaptability

Triggering resizing

The picker can adapt to the parent element width. Once such a change happens, the picker will recalculate its dimensions and add or remove navigation (previous and next arrows) for each date level, if necessary.

There are two ways to get the picker auto-correcting its dimensions:

1. Set `windowResize = true` option. The resize will be triggered on window resize event;
2. Disable resizing on window event (`windowResize = false`) and trigger resize on-demand. This is accomplished by dispatching event `<picker id>:resize` on picker root element which is accessible on `DTPicker` instance (`DTPicker.refs.root`).

   For example, for Vue.js it can be done with `onMounted` hook:

```
<template>
    <DTPicker v-bind='pickerOptions' ref='picker'>
</template>

<script setup>
import { ref, onMounted } from 'vue'
import DTPicker from './dt-picker.vue'

const picker = ref(null)

const pickerOptions = {
    windowResize: false, /* This is a default */
    id: 'picker-id',
    /* ... other options ... */
}

onMounted( _ => {
    /* Here we use #content as an element with width transition.
       It contains the picker element.
    */
    const content = document.getElementById('content')

    content.ontransitionend = e => {
        /* Check it's a width transition on `content` */
        if (e.target == content && e.propertyName == 'width')
            picker.value.picker.refs.root.dispatchEvent(new Event('picker-id:resize'))
    }

})
</script>
```

IMPORTANT: if any parent element has width transition/animation, `<picker-id>:resize` event **must** be dispatched at the end of this transition/animation with picker's option `windowResize = false`. Otherwise, the picker may attempt to correct dimensions before the parent transition/animation ends and wrong dimensions may be set.

In general, `windowResize` option should be used only when you need to resize the picker exactly on the window resize event and when there is no transition/animation on parent element. For full control of picker resize it's recommended to use `<picker-id>:resize` event only.

Compact mode

Compact mode is auto-enabled when there is no space for a full-sized picker. Set parent element dimensions to get the picker in compact mode.

In general, control of resizing and compact mode is done by `<picker-id>:resize` event. If you change the parent element width programmatically, emit this event to trigger picker auto-resizing.

When auto-correcting its dimensions, the picker also sets property designating whether a compact mode is enabled: `DTPicker.compact` and, if compact mode has been changed, emits event `<picker-id>:compact` with respective payload (`true` or `false`).

> For reference: in the current implementation compact mode will be enabled when the picker's parent element width will be less than ~37em.

A note on setting stub in compact mode: stub for this mode needs to take into account additional top layer for selected values, i. e. picker needs to be aware whether this is a compact mode or not. If no min-max dates are provided in options, the picker can't determine whether should it use compact mode or not, so sets a regular stub intended for a full-sized picker. Alternatively, when min-max values are present, it is clear what size mode is in use, so the picker sets appropriate stubs for compact and full-sized modes.

Styling

Styling is done with beautiful CSS preprocessor Stylus. So you need to use it too in case comprehensive control is needed. Otherwise, simply import precompiled CSS from `date-picker/css/date-picker.css` but be aware that in this case picker will use default values for all its instances.

When using Stylus, to overwrite some or all of the theme variables, import main and size themes as follows (there may be more than one picker instance on one page):

```
/* Theme ID must be set in case of several picker elements. */
DTPickerID = 'picker-id'

/* import main theme to set or overwrite all to defaults. */
@import 'date-picker/css/src/theme-main'

/* Overwrite non-size variables: colors, etc. */
color1 = yellowgreen

/* Import size theme to set or overwrite all variables to defaults. */
@import 'date-picker/css/src/theme-size'

/* Overwrite size variables. For relative font units `rem` is recomended. */
fontSize = 15px

/* import styles to be overwritten. */
@import 'date-picker/css/src/main'
@import 'date-picker/css/src/size'
```

Using media queries:

```
DTPickerID = 'picker-id'
@import 'date-picker/css/src/theme-main'
@import 'date-picker/css/src/theme-size'

/* Change sizes for viewport width <= 500px */
@media (max-width: 500px)
    DTPickerID = 'picker-id' /* Specify picker ID. */
    fontSize = 12px /* Set variables. */
    @import 'date-picker/css/src/size' /* Import size styles. */
```

Some styles are set using animations and cannot be changed by directly applying CSS (for example, a CSS `filter` for a blocked picker).

When using Vue and applying its transition build-in component, it may not work as expected due to the self-updating nature of `DTPicker.attrs`. In such cases wrap the picker component in `div` and apply transition on it, like so:

```
<template>
    <transition name='fade' mode='out-in'>
        <div class='slider-wrapper' :key='pickerKey'>
            <DTPicker v-bind='options' @update:model='model = $event' />
        </div>
    </transition>
</template>

<script setup>
    import {  computed } from 'vue'
    /* ... some code ... */
    const pickerKey = computed( _ => ... )
    /* ... some code ... */
</script>
```

All icons live in the `icons` directory. To change existing ones, simply replace icon files.

Web version

Demo