Wrapping and extending Quasar components

Using TypeScript

Evert van der Weit 

QUASAR

Wrapping components

What do I mean by that?

Normal Quasar component usage:

```
<template>
  <q-input
    v-model="form.profession"
    label="Profession"
    outlined
    lazy-rules
    hide-bottom-space
    no-error-icon
    :rules="[(val) => !!val || 'Profession is required']"
  />
</template>
```

Wrapping components - 2

What do I mean by that?

Wrapped component usage:

```
<template>
  <qsr-input
    v-model="form.profession"
    label="Profession"
    :rules="[(val) => !!val || 'Profession is required']"
  />
</template>
```

QsrInput.vue content:

```
<template>
  <q-input
    outlined
    no-error-icon
    lazy-rules
    hide-bottom-space
  />
</template>
```

Extra properties passed to `qsr-input` will be passed down to `q-input` automatically by Vue.

Use the `qsr-input` in your application instead of `q-input`.

Use cases

Why would you want to do this?

- Set 'default' properties for Quasar components
- Add general component logic in one place (QTable/QSelect filter functions)
- Custom design of components (in `` `style` `` blocks)
- Easy refactoring
- Extending component functionality

Challenges when wrapping

Of course there are...

Few things to consider:

- Components with slots
- Using Quasar component methods
- Property validation & IDE hints
- Typed properties, slots & emits
- Automagic availability of components

⚠ Spoiler alert ⚠

The perfect solution does not (yet) exist

But what can we do? There are two main ways of doing this that I will highlight and compare

Option 1

- Components with slots
- Using Quasar component methods
- Property validation & IDE hints
- ☑ Typed properties, ~~*slots*~~ & emits
- Automagic availability of components

Option 2

- Components with slots
- Using Quasar component methods
- Property validation & IDE hints
- Typed properties, slots & emits
- ☑ Automagic availability of components

The next slides will show exactly how we do this for both options.

Components with slots

Many Quasar components have predefined slots, how does that work?

Slots are not automatically passed down to child components like properties are.

Inside the wrapper component we can pass down all slots to the `q-input` component like this[1] :

```
<template>
  <q-input
    -- omitted for clarity --
  >
    <template v-for="(_, slot) in $slots" v-slot:[slot]="scope">
      <slot :name="slot" v-bind="scope || {}" />
    </template>
  </q-input>
</template>
```

This is the same for *Option 1* and *Option 2*

---

1. Copied from a comment in this gist

Using Quasar component methods

How does that work?

> The examples shown will use script setup and TypeScript

Normally we would do something like this:

```
<template>
  <q-input
    ref="inputRef"
    -- ommitted for clarity --
  />
</template>

<script setup lang="ts">
import { QInput } from 'quasar';

const inputRef = ref<QInput | null>(null);

function someMethod() {
  inputRef?.value?.focus();  // This will autocomplete and show documentation on hover
}
</script>
```

Using Quasar component methods - 2

How does that work?

When using a wrapper the parent component looks similar

```
<template>
  <qsr-input
    ref="inputRef"
    -- ommitted for clarity --
  />
</template>

<script setup lang="ts">
import { QInput } from 'quasar';

// The key 'input' we define inside our wrapper component
const inputRef = ref<{ input: QInput } | null>(null);

function someMethod() {
  inputRef?.value?.input.focus();  // This will autocomplete and show documentation on hover
}
</script>
```

Using Quasar component methods - 3

How does that work?

By default Vue will not expose methods when using a ref. Inside `QsrInput.vue` we have to expose it explicitely:

```
<template>
  <q-input
    ref="inputRef"
    -- ommitted for clarity --
  />
</template>

<script setup lang="ts">
import { QInput } from 'quasar';

const inputRef = ref<QInput | null>(null);

defineExpose({
  // Here we define the key mentioned in the previous slide
  input: inputRef,
});
</script>
```

Property validation & IDE hints

The magic of Volar

When using Volar inside VSCode there are some IDE hints:

```
3   <q-form class="q-gutter-md" style="width: 300px">
4     <h4 class="q-mb-md">Awesome form</h4>
5     <q-input
6       v-model="form.name"
7       label="Name"
8       outlined
9       no-error-icon
```

```
5     <q-input
6       v-model="form.name"
7       :label="101"
8       outlined
```

```
[WSL: UBUNTU-20.04]

5     <q-input
6       v-model="form.name"
7       label="Name"
8       outlined
9       no-error-icon
10      lazy-rules
11      hide-bottom-space
12      :rules="[
13        (val) => !!val || 'Name is required',
14        (val) => val.length > 3 || `We don't like short names`,
```

Property validation & IDE hints - 2

The magic of Volar

We don’t want to lose these features when using wrappers!

Inside `QsrInput.vue` we have to extend QInput properties. [1]

```
<script setup lang="ts">
import { QInputProps } from 'quasar';

// eslint-disable-next-line @typescript-eslint/no-empty-interface
interface QsrInput extends QInputProps {
  // Caveat: Every prop used inside this component needs to be defined here like:
  modelValue: QInputProps['modelValue'];
}

// We cannot pass imported interface to defineProps directly [1]
defineProps<QsrInput>()
</script>
```

---

1. Github issue

Typed properties, slots & emits

Kind of overlapping with the previous point

We already saw property typing, and emits are also typed automatically when extending `QInputProps`.

```
61      <qsr-input
62        v-model="form2.profession"
63        label="Profession"
64        :rules="[(val) => !!val || 'Profession is required']"
65        :no-error-icon="false"
66        @blur="onBlur"
67      />
```

Slots are the tough ones though, with `QInput` they have autocomplete and documentation, as well as typed slot parameters

```
39        lazy-rules
40        hide-bottom
41        no-error-ic
42      >
43        <template #prepend>
44          <q-icon name="local_pizza" />
45        </template>
46      </q-input>
```

(property) prepend: unknown

Prepend inner field; Suggestions: QIcon, QBtn

Automagic availability of components

Where paths will diverge 😢

So far everything is similar for *Option 1* and *Option 2* mentioned earlier.

For this point there are two choices:

1. Use a tool to auto-import it when used (like Quasar does internally)
2. Register them globally in your app

Pro's and cons

Auto-import

- Only import when used
- Easier to maintain
- No hints/types on slots

Global components

- Slots can have hints/types
- Extra boilerplate code necessary
- Unused components are included anyways

Option 1: Auto import

Using unplugin-vue-components

We can add a Vite plugin to auto-import all Vue components used in your app.

```
yarn add unplugin-vue-components
```

Inside `quasar.config.js` > build:

```
vitePlugins: [
  [
    'unplugin-vue-components/vite',
    {
      dts: true,
    },
  ],
]
```

Add the generated .dts file to `tsconfig.json` > `include` field:

```
"components.d.ts"
```

Option 2: Global components

Register all wrappers globally

To do this, we register a boot file `wrappers.ts`:

```
import { boot } from 'quasar/wrappers';
import QsrInput from 'src/wrappers/QsrInput.vue';

// This boot file will need to be extended for every extra wrapper that is defined
export default boot(async ({ app }) => {
  app.component('QsrInput', QsrInput);
});
```

Don’t forget to add the boot file inside `quasar.config.js`

Option 2: Global components - 2

Register all wrappers globally

Define TypeScript definitions for your global components, inside `wrappers.d.ts`

```
import { GlobalComponentConstructor, QInputSlots } from 'quasar';
import { _QsrInput } from './types';

declare module '@vue/runtime-core' {
  export interface GlobalComponents {
    QsrInput: GlobalComponentConstructor<_QsrInput, QInputSlots>;
  }
}
```

`_QsrInput` needs to extend `QInputProps`, which is imported here and inside `QsrInput.vue` to avoid duplication.

Wrapping up

Yes, pun intended

Which one to choose?

Can I have the code?

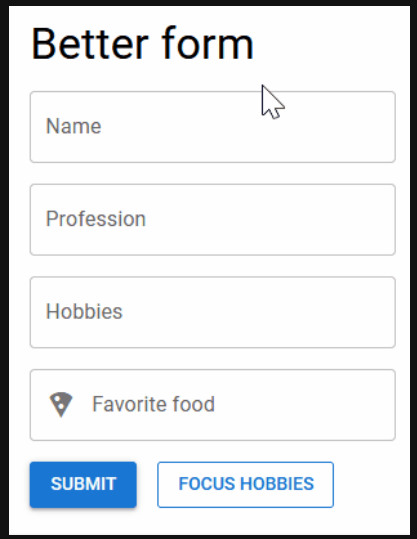
Yes of course! The slides are also in this repository. The main branch contains *Option 1*, and there is a separate branch for *Option 2*

https://github.com/Evertvdw/quasar-conf-wrappers

Bonus - Extending Quasar components

Extra goodies!

There is an example the repository of an Material Design styled QInput field, where you can control the animation speed of the label 😎



Thank You!

Slides & code can be found on github.com/Evertvdw/quasar-conf-wrappers