onnet

2021
VIE

Odoo widget

www.on.net.vn
www.arrowhitech.com

onnet
BY AHT TECH JSC

Table of contents

onnet
BY AHT TECH JSC

Introduction

For **v13.0 and below** Odoo uses **Backbone JS** to handle its front end. **Backbone** is a modern JavaScript library that is based on the **model-view design paradigm**. So, it is unsure of a **structured user interface** for a large project (By providing building blocks like widgets).

Introduction

As Odoo’s definition “**The Odoo Web Library (OWL)** is a smallish (~<20kb gzipped) UI framework intended to be the basis for the **Odoo Web Client**. The **OWL** is a modern framework, written in Typescript, taking the best ideas from React and Vue in a simple and consistent way.”

OWL uses Odoo’s powerful template engine “**Qweb**” to handle front end HTML fragments and pages.

OWL uses **components** as it’s building blocks, the same as **React JS**. Simply, components are JavaScript **classes and functions** to describe how UI(or a part of UI) should appear on the screen. It helps to keep a structured view(tree view) to the building blocks. Even though Owl uses JS classes to build components rather than JS functions. And it is super dynamic too (by the help of Qweb engine).

Introduction

Why OWL ?

Odoo is **extremely modular**. This means, for example, that the core parts of Odoo are not aware, **before runtime**, of what files will be loaded/executed, or what will be the state of the UI. Because of that, Odoo cannot rely on a standard build toolchain. Also, this implies that the core parts of Odoo need to be extremely generic. In other words, Odoo is **not** really **an application with a user interface**. It is an application which **generates a dynamic user interface.** And most frameworks are not up to the task.

OWL Components

OWL components are the building blocks for user interface. They are designed to be:

1. **declarative**: the user interface should be described in terms of the state of the application, not as a sequence of imperative steps.
2. **composable**: each component can seamlessly be created in a parent component by a simple tag or directive in its template.
3. **asynchronous** rendering: the framework will transparently wait for each sub components to be ready before applying the rendering. It uses native promises under the hood.
4. **uses QWeb as a template system**: the templates are described in XML and follow the QWeb specification. This is a requirement for Odoo.

OWL components are defined as a **subclass of Component**. The rendering is exclusively done by a **QWeb template** (which needs to be preloaded in QWeb). Rendering a component generates a virtual dom representation of the component, which is then patched to the DOM, in order to apply the changes in an efficient way.

OWL Components: Define a component

This example shows how a **component** should be defined: it simply **subclasses** the **Component** class. If no static template key is defined, then **Owl** will use the **component's name as template name**. Here, a state object is defined, by using the **useState** hook. It is not mandatory to use the state object, but it is certainly encouraged. The result of the **useState** call is **observed**, and **any changes** to it will cause a **rerendering**:

```
const { useState } = owl.hooks;
class ClickCounter extends owl.Component {
  state = useState({ value: 0 });
  increment() {
    this.state.value++;
  }
}
```

OWL Components: Static properties

- **`template`** (string, optional): if given, this is the name of the QWeb template that will render the component. Note that there is a helper xml to make it easy to define an inline template.
- **`components`** (Object, optional): if given, this is an object that contains the classes of any sub components needed by the template. This is the main way used by Owl to be able to create sub components.
- **`props`** (Object, optional): if given, this is an object that describes the type and shape of the (actual) props given to the component. If Owl mode is dev, this will be used to validate the props each time the component is created/updated.
- **`defaultProps`** (Object, optional): if given, this object define default values for (top-level) props. Whenever props are given to the object, they will be altered to add default value (if missing). Note that it does not change the initial object, a new object will be created instead.
- **`style`** (string, optional): it should be the return value of the css tag, which is used to inject stylesheet whenever the component is visible on the screen.

OWL Components: Methods

- **mount**(target, options) (async): this is the main way a component is added to the DOM: the root component is mounted to a target HTMLElement (or document fragment). Obviously, this is asynchronous, since each children need to be created as well. Most applications will need to call mount exactly once, on the root component.
- **unmount**(): in case a component needs to be detached/removed from the DOM, this method can be used. Most applications should not call unmount, this is more useful to the underlying component system.
- **render**() (async): calling this method directly will cause a rerender. Note that this should be very rare to have to do it manually, the Owl framework is most of the time responsible for doing that at an appropriate moment. Note that the render method is asynchronous, so one cannot observe the updated DOM in the same stack frame.
- **shouldUpdate**(nextProps): this method is called each time a component's props are updated.
- **destroy**(). As its name suggests, this method will remove the component, and perform all necessary cleanup, such as unmounting the component, its children, removing the parent/children relationship. This method should almost never be called directly (except maybe on the root component), but should be done by the framework instead.

OWL Components: Methods

- **willStart**(): willStart is an asynchronous hook that can be implemented to perform some action before the initial rendering of a component. It will be called exactly once before the initial rendering. It is useful in some cases, for example, to load external assets (such as a JS library) before the component is rendered. Another use case is to load data from a server.
- **mounted**(): is called each time a component is attached to the DOM, after the initial rendering and possibly later if the component was unmounted and remounted. At this point, the component is considered active. This is a good place to add some listeners, or to interact with the DOM, if the component needs to perform some measure for example.
- **willUpdateProps**(nextProps): The willUpdateProps is an asynchronous hook, called just before new props are set. This is useful if the component needs to perform an asynchronous task, depending on the props (for example, assuming that the props are some record Id, fetching the record data).
- ...

OWL Components: Lifecycles

Order	Methods	Description
1	`setup()`	setup
2	`willStart()`	async, before first rendering
3	`mounted()`	just after component is rendered and added to the DOM
4	`willUpdateProps()`	async, before props update
5	`willPatch()`	just before the DOM is patched
6	`patched()`	just after the DOM is patched
7	`willUnmount()`	just before removing component from DOM
8	`catchError()`	catch errors

Q&A