WUX – Wrapped User Experience

Tutorial

SUMMARY

1. INTRODUCTION

WUX is not really a framework. It provides the basic functionality for the implementation of wrappers that can be assimilated to the simple or complex components of a user interface.

It uses **JQuery** as a library for DOM manipulation, event management and for all the most common utilities necessary to create cross-browser **HTML 5** pages and the **Bootstrap** framework for all aspects of presentation and style according to the **CSS 3** standard.

The use of JQuery allows you to take advantage of a vast set of plug-ins and third-party[1] components, both commercial and open source.

The use of the **Typescript**[2] language is recommended for the development of **WUX** components and user interfaces. The use of Typescript in general allows you to develop Javascript code using the paradigms used in object-oriented programming languages such as Java. In addition, development environments can offer better support in writing code in order to improve the robustness and maintainability of particularly complex interfaces.

The advantage of using **WUX** is to have a basic model for the creation of even very complex user interfaces through the composition of reusable components according to the OOP (Object Oriented Programming) paradigm and is particularly suitable for the development of management software in **HTML5**.

[1] E.g. https://js.devexpress.com/
[2] https://www.typescriptlang.org/

2. OVERVIEW

The implementation of the **WUX** components requires the extension of the **WUX.WComponent** class which has a series of methods to be implemented for life cycle management, state and event management.

A simple example of implementation is the following:

```
class HelloWorld extends WUX.WComponent {

    protected render() {
        return '<h1>Hello world</h1>';
    }

}
```

As you can see from the example code, **WUX** is inspired by **React**[3] and other methods have the same name and meaning as those found in React.Component.

To show the component on the page just write the following Javascript code:

```
<script type="text/javascript">

    WuxDOM.render(new HelloWorld(), 'view-root');

</script>
```

Where 'view-root' is the id of the HTML element (generally a div element) present on the page.

As in React, the WUX components also manage two attributes for managing the data model (*props and state*) which can be "typed" in the class declaration using Typescript Generics[4].

We will extend the example shown above by having the component show the greeting to a specified name with the possibility of indicating the tag in which this greeting must be enclosed.

[3] https://reactjs.org/
[4] https://www.typescriptlang.org/docs/handbook/generics.html

```
class HelloWorld extends WUX.WComponent<number, string> {

    constructor(headingTag: number, name: string) {
        super();
        this.setProps(headingTag);
        this.setState(name);
    }

    protected render() {
        return '<h' + this.props + '>Hello ' + this.state + '</h' + this.props + '>';
    }

}
```

To show the new component on the page we write:

```
<script type="text/javascript">

    WuxDOM.render(new HelloWorld(1, 'dev'), 'view-root');

</script>
```

Up to this point we have seen how the component can be implemented simply by returning a string containing the HTML code to be represented.

However, the render method can also return a JQuery object or a WUX component. Eg:

```
class HelloWorld extends WUX.WComponent<number, string> {

    constructor(headingTag: number, name: string) {
        super();
        this.setProps(headingTag);
        this.setState(name);
    }

    protected render() {
        let $h = $('<h' + this.props + '></h' + this.props + '>');
        $h.text('Hello ' + this.state);
        return $h;
    }

}
```

The example component is still "static" with respect to the change of state (name) and property (tag), ie it shows what was set in the constructor during creation.

To make the component dynamic, you need to write the following:

```
class HelloWorld extends WUX.WComponent<number, string> {
    constructor(headingTag: number, name: string) {
        //    id            name          props
        super('helloworld', 'HelloWorld', headingTag);
        this.setState(name);
    }

    protected updateState(nextState: string): void {
        super.updateState(nextState);
        if (this.root) this.root.text('Hello ' + this.state);
    }

    protected render() {
        let $h = $('<h' + this.props + '></h' + this.props + '>')
        $h.text('Hello ' + this.state);
        return $h;
    }

}
```

As you can see, a new method (updateState) has been implemented that uses the **root** element. This element represents the JQuery object obtained as returned by the render method. It will generally[5] be undefined as long as the component is not shown (mounted) on the page, so you need to check that it is available before using it in the pre-mount phase.

Through these simple examples we wanted to provide a quick overview of **WUX** to understand its principles. **WUX** offers several basic components for building a user interface in HTML 5.

[5] In the constructor, if the specified id refers to an existing object, the root element may already be set.

3. LIFE CYCLE OF A COMPONENT

A WUX component can be displayed on a page (mounted) one or more times. The mount operation cascades the following methods:

```
1. protected componentWillMount(): void;
2. protected render(): any;
3. protected componentDidMount(): void;
```

The *mount* is performed on a node of the HTML page which in the component is represented by the *context*.

The *root* element has already been introduced which represents the JQuery object with the actual implementation of the component. The *context*, also a JQuery object, is the container of the root object. Sometimes the root object may coincide with the context: this happens when the component is implemented as an enhancement of the context already present in the page. For example, if you want to implement a component that uses a datepicker[6] and the component id coincides with that of the context, the implementation could be limited to *this.context.datepicker* (). However, it is suggested to always use the *root* element. The mount implementation, in fact, verifies that the component id does not coincide with that of the *context*: in this case root will coincide with context. It will generally be available after invoking the *render* method (which will not return anything in this case), therefore it will be necessary to implement the componentDidMount method as in the following example.

```
class SelectDate extends WUX.WComponent {

    constructor(id: string) {
        super(id);
        this.rootTag = 'input';
    }

    protected componentDidMount(): void {
        this.root.datepicker();
    }

}
```

As you can see, the implementation of the render method is the default one: it constructs (if necessary) an element with type this.rootTag (by default it is the div element). The *componentDidMount* method will enhance the input element.

[6] JQueryUI package date selection component (https://jqueryui.com/datepicker/)

4. STATE MANAGEMENT

WUX components have **props** and **state** attributes for model implementation. These attributes can also contain complex data structures.

When invoking **setState** or **setProps** the component calls the following methods:

```
1. protected shouldComponentUpdate(nextProps: P, nextState: S): boolean;
2. protected componentWillUpdate(nextProps: P, nextState: S): void;
3. protected updateProps(nextProps: P): void; (in caso di setProps)
4. protected updateState(nextState: S): void; (in caso di setState)
5. protected componentDidUpdate(prevProps: P, prevState: S): void;
```

According to what is returned by *shouldComponentUpdate*, the invocation of the other methods is continued. If the *forceUpdate* method is called, the invocation of *shouldComponentUpdate* is omitted and the other methods are still invoked (except *updateProps* and *updateState*). This forcing causes the component to be reassembled after invoking the *componentWillUpdate* method.

5. EVENT MANAGEMENT

The WUX components expose the following functions for event management:

```
1. on(events: string, handler: (e: any) => any): this;

2. off(events?: string): this;

3. trigger(eventType: string, ...extParams: any[]): this;
```

These functions are very similar to the corresponding functions present in JQuery.

WUX components support the following events:

- mount
- unmount
- statechange
- propschange

For example:

```
let txtTest = new WUX.WInput('txtTest', WUX.WInputType.Text);

txtTest.on('statechange', (e: WUX.WEvent) => {
       console.log('WInput:statechange', e.component.getState());

});
```

In the creation of a component, attention must be paid to the management of the state, especially when it could be modified through user action. In this case it is suggested to call the trigger function. Below is an example taken from the implementation of the WSelect component:

```
protected componentDidMount(): void {
    this.root.on('change', (e: JQueryEventObject) => {
        this.trigger('statechange', this.root.val());
    });

}
```

As can be seen, the new state is also passed to the trigger function and must be updated without invoking the methods presented in the chapter on state management.