WUX – Wrapped User Experience

Tutorial

https://github.com/giosil/wux

SOMMARIO

1. INTRODUZIONE

WUX non è un framework. Esso fornisce le funzionalità di base per l'implementazione di wrapper che possono essere assimilati ai componenti semplici o complessi di una interfaccia utente.

Esso utilizza **JQuery** come libreria per la manipolazione del DOM, la gestione degli eventi e per tutte le utilità più comuni necessarie alla realizzazione di pagine **HTML 5** cross-browser e il framework **Bootstrap** per quanto riguarda tutti gli aspetti di presentazione e stile secondo lo standard **CSS 3**.

L'utilizzo di JQuery consente di usufruire di un vasto insieme di plug-in e componenti di terze parti[1] sia commerciali che open source.

Si raccomanda l'utilizzo del linguaggio **Typescript**[2] per lo sviluppo dei componenti WUX e delle interfacce utenti. L'impiego di Typescript in generale consente di sviluppare codice **Javascript** utilizzando i paradigmi impiegati nei linguaggi di programmazione orientati agli oggetti come Java. Inoltre gli ambienti di sviluppo possono offrire un supporto migliore nella stesura del codice allo scopo di migliorare la robustezza e la manutenibilità di interfacce particolarmente complesse.

Il vantaggio di utilizzare **WUX** è di avere un modello base per la realizzazione di interfacce utente anche molto complesse tramite la composizione di componenti riutilizzabili secondo il paradigma di OOP (Object Oriented Programming) ed è indicato particolarmente per lo sviluppo di gestionali in **HTML5**.

[1] Ad esempio https://js.devexpress.com/
[2] https://www.typescriptlang.org/

2. PANORAMICA

L'implementazione dei componenti **WUX** richiede l'estensione della classe **WUX.WComponent** la quale ha una serie di metodi da implementare per la gestione del ciclo di vita, la gestione degli stati e degli eventi.

Un esempio semplice di implementazione è il seguente:

```
class HelloWord extends WUX.WComponent {

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

}
```

Come si può osservare dal codice di esempio, **WUX** è ispirato a **React**[3] e altri metodi hanno lo stesso nome e significato di quelli presenti in *React.Component*.

Per mostrare il componente sulla pagina è sufficiente scrivere il seguente codice Javascript:

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

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

</script>
```

Dove 'view-root' è l'id dell'elemento HTML (generalmente un elemento *div*) presente nella pagina.

Come in React anche i componenti WUX gestiscono due attributi per la gestione del modello dati (*props* e *state*) i quali possono essere "tipizzati" nella dichiarazione della classe impiegando i Generics[4] di Typescript.

Si estenderà l'esempio sopra mostrato facendo in modo che il componente mostri il saluto ad un nome specificato con la possibilità di indicare il tag nel quale tale saluto deve essere racchiuso.

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

```
class HelloWord 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 + '>';
    }

}
```

Per mostrare il nuovo componente nella pagina si scriverà:

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

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

</script>
```

Fino a questo punto si è visto come il componente possa essere implementato semplicemente restituendo una stringa contenente il codice HTML da rappresentare.

Il metodo render tuttavia può restituire anche un oggetto JQuery o un componente WUX. Ad esempio:

```
class HelloWord 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;
    }

}
```

Il componente di esempio è ancora “statico” rispetto al cambio di stato (nome) e proprietà (tag) ovvero esso mostra quanto impostato in fase di creazione nel *constructor*.

Per rendere dinamico il componente occorre scrivere quanto segue:

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

}
```

Come si può osservare è stato implementato un nuovo metodo (*updateState*) che impiega l'elemento **root**. Tale elemento rappresenta l'oggetto JQuery ottenuto secondo quanto restituito dal metodo *render*. Esso sarà generalmente[5] *undefined* fintanto che il componente non viene mostrato (*mounted*) sulla pagina, pertanto occorre verificare che sia disponibile prima di utilizzarlo in fase precedenti il *mount*.

Attraverso questi semplici esempi si è voluto fornire una rapida panoramica di **WUX** per comprenderne i principi. **WUX** offre diversi componenti di base per la costruzione di una interfaccia utente in HTML 5.

[5] Nel constructor, se l'id specificato si riferisce ad un oggetto esistente, l'elemento root potrebbe essere già valorizzato.

3. CICLO DI VITA DI UN COMPONENTE

Un componente WUX può essere visualizzato in una pagina (montato) una o più volte. L'operazione di *mount* esegue in cascata i seguenti metodi:

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

Il *mount* viene eseguito su un nodo della pagina HTML che nel componente è rappresentato dal *context*.

E' stato già introdotto l'elemento *root* che rappresenta l'oggetto JQuery con l'implementazione vera e propria del compoente. Il *context,* anch'esso oggetto JQuery, è il container del oggetto *root*. Talvolta l'oggetto *root* potrebbe coincidere con il *context:* questo avviene quando il componente è implementato come un *enhancement* del *context* già presente nella pagina. Ad esempio se si vuole implementare un componente che utilizzi un datepicker[6] e l'id del componente coincide con quello del context l'implementazione potrebbe limitarsi a *this.context.datepicker()*. Si suggerisce tuttavia di utilizzare sempre l'elemento root. L'implementazione del mount, infatti, verifica che l'id del componente non coincida con quello del *context*: in tal caso root coinciderà con context. Esso sarà generalmente disponibile dopo l'invocazione del metodo *render* (che non restituirà nulla in tal caso), pertanto occorrerà implementare il metodo componentDidMount come nel seguente esempio.

```
class SelectDate extends WUX.WComponent {

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

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

}
```

Come si può osservare l'implementazione del metodo render è quella di default: costruisce (se necessario) un elemento con di tipo this.rootTag (di default è l'elemento div). Il metodo componentDidMount effettuerà l'enhancement dell'elemento input.

[6] Componente per la selezione di date del pacchetto JQueryUI (https://jqueryui.com/datepicker/)

4. GESTIONE DELLO STATO

I componenti WUX hanno gli attributi **props** e **state** per l'implementazione del modello. Tali attributi possono contenere anche strutture dati complesse.

Quando si invoca il **setState** o il **setProps** il componente richiama i seguenti i metodi:

```
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;
```

Secondo quanto restituito da *shouldComponentUpdate* si prosegue nell'invocazione degli altri metodi. Nel caso in cui venga chiamato il metodo *forceUpdate* viene omessa l'invocazione di *shouldComponentUpdate* e gli altri metodi vengono comunque invocati (tranne *updateProps* e *updateState*). Tale forzatura provoca il rimontaggio del componente dopo l'invocazione del metodo *componentWillUpdate*.

5. GESTIONE DEGLI EVENTI

I componenti WUX espongono le seguenti funzioni per la gestione degli eventi:

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

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

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

Tali funzioni sono del tutto simili alle corrispondenti funzioni presenti in JQuery.

I componenti WUX supportano i seguenti eventi:

- mount
- unmount
- statechange
- propschange

Ad esempio:

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

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

});
```

Nella realizzazione di un componente bisognerà prestare attenzione alla gestione dello stato specie quando esso potrebbe essere modificato attraverso un'azione dell'utente. In tal caso si suggerisce di richiamare la funzione trigger. Di seguito un esempio tratto dall'implementazione del componente WSelect:

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

}
```

Come si può osservare alla funzione trigger viene passato anche il nuovo stato che dovrà essere aggiornato senza l'invocazione dei metodi presentati nel capitolo relativo alla gestione dello stato.