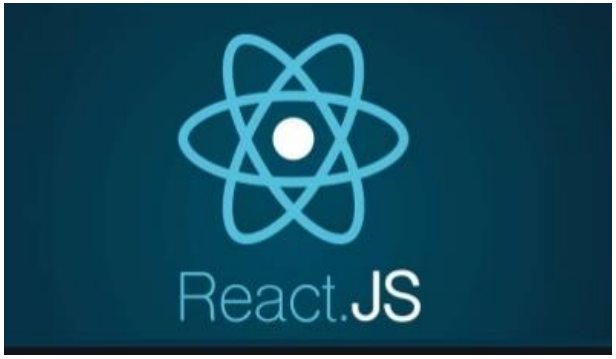


Presentation by Uplatz

Contact Us: https://training.uplatz.com/

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

Using ReactJS with TypeScript:

ReactJS component written in TypeScript

- Actually you can use ReactJS's components in Typescript as in facebook's example. Just replace 'jsx' file's extension to 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
 render: function() {
 return <div>Hello {this.props.name}</div>;
 }
});
ReactDOM.render(<HelloMessage name="John" />,
mountNode);
```

- But in order to make full use of Typescript's main feature (static type checking) should be done couple things:

1) convert React.createClass example to ES6 Class:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
 render() {
 return <div>Hello {this.props.name}</div>;
 }
}
ReactDOM.render(<HelloMessage name="John" />,
mountNode);
```

2) next add Props and State interfaces:

```
interface IHelloMessageProps {
```

```
 name:string;
}
interface IHelloMessageState {
 //empty in our case
}
class HelloMessage extends
React.Component<IHelloMessageProps,
IHelloMessageState> {
 constructor(){
 super();
 }
 render() {
 return <div>Hello {this.props.name}</div>;
 }
```

}

ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);

- Now Typescript will display an error if the programmer forgets to pass props. Or if they added props that are not defined in the interface.

Section 3.2: Installation and Setup

- To use typescript with react in a node project, you must first have a project directory initialized with npm.
- Toinitialize the directory with npm init

Installing via npm or yarn

- You can install React using npm by doing the following:

npm install --save react react-dom

- Facebook released its own package manager named Yarn, which can also be used to install React.
- After installing Yarn you just need to run this command:

yarn add react react-dom

- You can then use React in your project in exactly the same way as if you had installed React via npm.
- Installing react type definitions in Typescript 2.0+
- To compile your code using typescript, add/install type definition files using npm or yarn.

npm install --save-dev @types/react @types/react-dom

or, using yarn

yarn add --dev @types/react @types/react-dom

- Installing react type definitions in older versions of Typescript
- You have to use a separate package called tsd

tsd install react react-dom --save

- Adding or Changing the Typescript configuration
- To use JSX, a language mixing javascript with html/xml, you have to change the typescript compiler configuration.
- In the project's typescript configuration file (usually named tsconfig.json), you will need to add the JSX option as:

“**compilerOptions": { "jsx": "react" },**

- That compiler option basically tells the typescript compiler to translate the JSX tags in code to javascript function calls.
- To avoid typescript compiler converting JSX to plain javascript function calls, use

"compilerOptions": {

"jsx": "preserve"

},

Stateless React Components in TypeScript

- React components that are pure functions of their props and do not require any internal state can be written as JavaScript functions instead of using the standard class syntax, as:

```
import React from 'react'
const HelloWorld = (props) => (
 <h1>Hello, {props.name}!</h1>
);
```

- The same can be achieved in Typescript using the React.SFC class:

```
import * as React from 'react';
class GreeterProps {
 name: string
}
const Greeter : React.SFC<GreeterProps> = props
=>
 <h1>Hello, {props.name}!</h1>;
```

- Note that, the name React.SFC is an alias for React.StatelessComponent So, either can be used.

Stateless and property-less Components

- The simplest react component without a state and no properties can be written as:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

- That component, however, can't access this.props since typescript can't tell if it is a react component. To access its props, use:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () =>
<span>Hello, World!</span>
```

- Even if the component doesn't have explicitly defined properties, it can now access props.children

- since all components inherently have children.
- Another similar good use of stateless and property-less components is in simple page templating.
- The following is an examplinary simple Page component, assuming there are hypothetical Container, NavTop and NavBottom

components already in the project:

```
import * as React from 'react';
const Page: React.SFC<{}> = props => () =>
 <Container>
 <NavTop />
 {props.children}
 <NavBottom />
 </Container>
```

```
const LoginPage: React.SFC<{}> = props => () =>
 <Page>
 Login Pass: <input type="password" />
 </Page>
```

- In this example, the Page component can later be used by any other actual page as a base template.

State in React

Basic State

- State in React components is essential to manage and communicate data in your application.
- It is represented as a JavaScript object and has component level scope, it can be thought of as the private data of your component.

- In the example below we are defining some initial state in the constructor function of our component and make use of it in the render function.

```
class ExampleComponent extends React.Component
{
 constructor(props){
 super(props);
 // Set-up our initial state
 this.state = {
 greeting: 'Hiya Buddy!'
 };
 }
 render() {
 // We can access the greeting property through
```

```
this.state
 return(
 <div>{this.state.greeting}</div>
 );
 }
}
```

Common Antipattern

- You should not save props into state. It is considered an anti-pattern. For example:

```
export default class MyComponent extends
React.Component {
 constructor() {
 super();
 this.state = {
 url: ''
```

```
}
this.onChange = this.onChange.bind(this);
}
onChange(e) {
this.setState({
url: this.props.url + '/days=?' + e.target.value
});
}
componentWillMount() {
this.setState({url: this.props.url});
}
render() {
return (
<div>
```

```
<input defaultValue={2} onChange={this.onChange} />
URL: {this.state.url}
 </div>
 )
 }
}
```

- The prop url is saved on state and then modified. Instead, choose to save the changes to a state, and then build the full path using both state and props:

```
export default class MyComponent extends
React.Component {
 constructor() {
 super();
 this.state = {
 days: "
```

```
}
 this.onChange = this.onChange.bind(this);
 }
 onChange(e) {
 this.setState({
 days: e.target.value
 });
 }
 render() {
 return (
 <div>
 <input defaultValue={2} onChange={this.onChange}
/>
```

```
URL: {this.props.url + '/days?=' + this.state.days}
 </div>
 )
 }
}
```

- This is because in a React application we want to have a single source of truth - i.e. all data is the responsibility ofone single component, and only one component.
- It is the responsibility of this component to store the data withinits state, and distribute the data to other components via props.
- In the first example, both the MyComponent class and its parent are maintaining 'url' within their state.

- If we update state.url in MyComponent, these changes are not reflected in the parent.
- We have lost our single source of truth, and it becomes increasingly difficult to track the flow of data through our application.
- Contrast this with the second example - url is only maintained in the state of the parent component, and utilised as a prop in MyComponent - we therefore maintain a single source of truth.

Using setState() with a Function as updater

// This is most often used when you want to check or make use

// of previous state before updating any values.

//

```
this.setState(function(previousState, currentProps) {
 return {
 counter: previousState.counter + 1
 };
});
```

- This can be safer than using an object argument where multiple calls to setState() are used, as multiple calls may
- be batched together by React and executed at once, and is the preferred approach when using current props to set state.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

- The above example is called a stateless component as it does not contain state (in the React sense of the word).

Stateless Functional Components

- In many applications there are smart components that hold state but render dumb components that simply receiveprops and return HTML as JSX.
- Stateless functional components are much more reusable and have a positive performance impact on your application.

They have 2 main characteristics:

1. When rendered they receive an object with all the props that were passed down
2. They must return the JSX to be rendered

this.setState({ counter: this.state.counter + 1 });

- These calls may be batched together by React using Object.assign(), resulting in the counter being incremented by 1 rather than 3.
- The functional approach can also be used to move state setting logic outside of components. This allows for isolation and re-use of state logic.

```
// Outside of component class, potentially in another file/module
function incrementCounter(previousState, currentProps) {
 return {
 counter: previousState.counter + 1
 };
}
```

// Within component

this.setState(incrementCounter);

Calling setState() with an Object and a callback function

// 'Hi There' will be logged to the console after setState completes

//

this.setState({ name: 'John Doe' }, console.log('Hi there'));

State, Events And Managed Controls:

- Here's an example of a React component with a "managed" input field.
- Whenever the value of the input field changes, an event handler is called which updates the state of the component with the new value of the input field.
- The call to setState in the event handler will trigger a call to render updating the component in the d

- Its very important to note the runtime behavior.
- Every time a user changes the value in the input field
- handleChange will be called and so
- setState will be called and so
- render will be called

Pop quiz, after you type a character in the input field, which DOM elements change

1. all of these - the top level div, legend, input, h1
2. only the input and h1
3. nothing
4. whats a DOM

React Lifecycle

Lifecycle of Components

- Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.
- The three phases are: Mounting, Updating, and Unmounting.

Mounting

- Mounting means putting elements into the DOM.
- React has four built-in methods that gets called, in this order, when mounting a component:
- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

- The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor

- The **constructor()** method is called before anything else, when the component is initiated, and it is the natural place to set up the initial **state** and other initial values.
- The **constructor()** method is called with the **props**, as arguments, and you should always start by calling the **super(props**) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (**React.Component**).

Example:

- The constructor method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
    );
  }
}
```

getDerivedStateFromProps

- The **getDerivedStateFromProps**() method is called right before rendering the element(s) in the DOM.
- This is the natural place to set the **state** object based on the initial **props**.
- It takes state as an argument, and returns an object with changes to the state.
- The example below starts with the favorite color being "red", but the **getDerivedStateFromProps()** method updates the favorite color based on the **favcol** attribute:

Example:

- The getDerivedStateFromProps method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
    );
  }
}
```

render

- The render() method is required, and is the method that actual outputs HTML to the DOM.

Example:

A simple component with a simple render() method:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

componentDidMount

- The componentDidMount() method is called after the component is rendered.
- This is where you run statements that requires that the component is already placed in the DOM.

Example:

- At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
```

```
setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header />,
document.getElementById('root'));
```

Updating

- The next phase in the lifecycle is when a component is updated.
- A component is updated whenever there is a change in the component's state or props.
- React has five built-in methods that gets called, in this order, when a component is updated:

getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

- The render() method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

- Also at updates the getDerivedStateFromProps method is called.
- This is the first method that is called when a component gets updated.
- This is still the natural place to set the state object based on the initial props.
- The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

- If the component gets updated, the getDerivedStateFromProps() method is called:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
```

```
<h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button"
onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}
ReactDOM.render(<Header favcol="yellow"/>,
document.getElementById('root'));
```

shouldComponentUpdate

- In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.
- The default value is true.

- The example below shows what happens when the shouldComponentUpdate() method returns false

Example:

- Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
```

```
render() {
    return (
      <div>
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
      <button type="button"
onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}
ReactDOM.render(<Header />,
document.getElementById('root'));
```

Example:

- Same example as above, but this time the shouldComponentUpdate() method returns true instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
```

```
render() {
    return (
      <div>
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
      <button type="button"
onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

ReactDOM.render(<Header />,
document.getElementById('root'));
```

render

- The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.
- The example below has a button that changes the favorite color to blue:

Example:

- Click the button to make a change in the component's state:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
```

```
changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
      <button type="button"
onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}ReactDOM.render(<Header />,
document.getElementById('root'));
```

getSnapshotBeforeUpdate

- In the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.
- If the **getSnapshotBeforeUpdate()** method is present, you should also include the **componentDidUpdate**() method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

- When the component is mounting it is rendered with the favorite color "red".
- When the component has been mounted, a timer changes the state, and after one second, the favorite color becomes "yellow".

- This action triggers the update phase, and since this component has a **getSnapshotBeforeUpdate**() method, this method is executed, and writes a message to the empty DIV1 element.
- Then the **componentDidUpdate**() method is executed and writes a message in the empty DIV2 element:

Example:

- Use the getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
```

```
componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " +
prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
```

```
 render() {
    return (
      <div>
        <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    );
  }
}

ReactDOM.render(<Header />,
document.getElementById('root'));
```

componentDidUpdate

- The componentDidUpdate method is called after the component is updated in the DOM.
- The example below might seem complicated, but all it does is this:
- When the component is mounting it is rendered with the favorite color "red".
- When the component has been mounted, a timer changes the state, and the color becomes "yellow".
- This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Example:

The componentDidUpdate method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
```

```
render() {
    return (
      <div>
      <h1>My Favorite Color is
{this.state.favoritecolor}</h1>
      <div id="mydiv"></div>
      </div>
    );
  }
}

ReactDOM.render(<Header />,
document.getElementById('root'));
```

Unmounting

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

componentWillUnmount

- The componentWillUnmount method is called when the component is about to be removed from the DOM.

Example:

- Click the button to delete the header:

```
class Container extends React.Component {
```

```
constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
```

```
{myheader}
      <button type="button"
onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}
class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be
unmounted.");
  }
  render() {
```

```
return (
     <h1>Hello World!</h1>
   );
  }
}
ReactDOM.render(<Container />,
document.getElementById('root'));
```

Summary of LifeCylce methods:

- **componentWillMount** is executed before rendering, on both the server and the client side.
- **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur.
- This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as setTimeout or setInterv

- We are using it to update the state so we can trigger the other lifecycle methods.
- **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from setNewNumber when we updated the state.
- **shouldComponentUpdate** should return true or false value. This will determine if the component will be updated or not.
- This is set to true by default. If you are sure that the component doesn't need to render after state or props are updated, you can return false value.
- **componentWillUpdate** is called just before rendering.
- **componentDidUpdate** is called just after rendering.

- **componentWillUnmount** is called after the component is unmounted from the dom. We are unmounting our component in main.js.
- In the following example, we will set the initial state in the constructor function.
- The setNewnumber is used to update the state. All the lifecycle methods are inside the Content component

```
import React from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
     this.state = {
       data: 0
    }
```

```
this.setNewNumber = this.setNewNumber.bind(this)
  };
  setNewNumber() {
    this.setState({data: this.state.data + 1})
  }
  render() {
    return (
      <div>
        <button onClick =
{this.setNewNumber}>INCREMENT</button>
        <Content myNumber =
{this.state.data}></Content>
      </div>
    );
  } }
```

Uplatz

```
class Content extends React.Component {
  componentWillMount() {
    console.log('Component WILL MOUNT!')
  }
  componentDidMount() {
    console.log('Component DID MOUNT!')
  }
  componentWillReceiveProps(newProps) {
    console.log('Component WILL RECIEVE PROPS!')
  }
  shouldComponentUpdate(newProps, newState) {
    return true;
  }
  componentWillUpdate(nextProps, nextState) {
    console.log('Component WILL UPDATE!');
```

```
}componentDidUpdate(prevProps, prevState) {
    console.log('Component DID UPDATE!')
  }
  componentWillUnmount() {
    console.log('Component WILL UNMOUNT!')
  }
  render() {
    return (
      <div>
        <h3>{this.props.myNumber}</h3>
      </div>
    );
  }
} export default App;
```

main.js

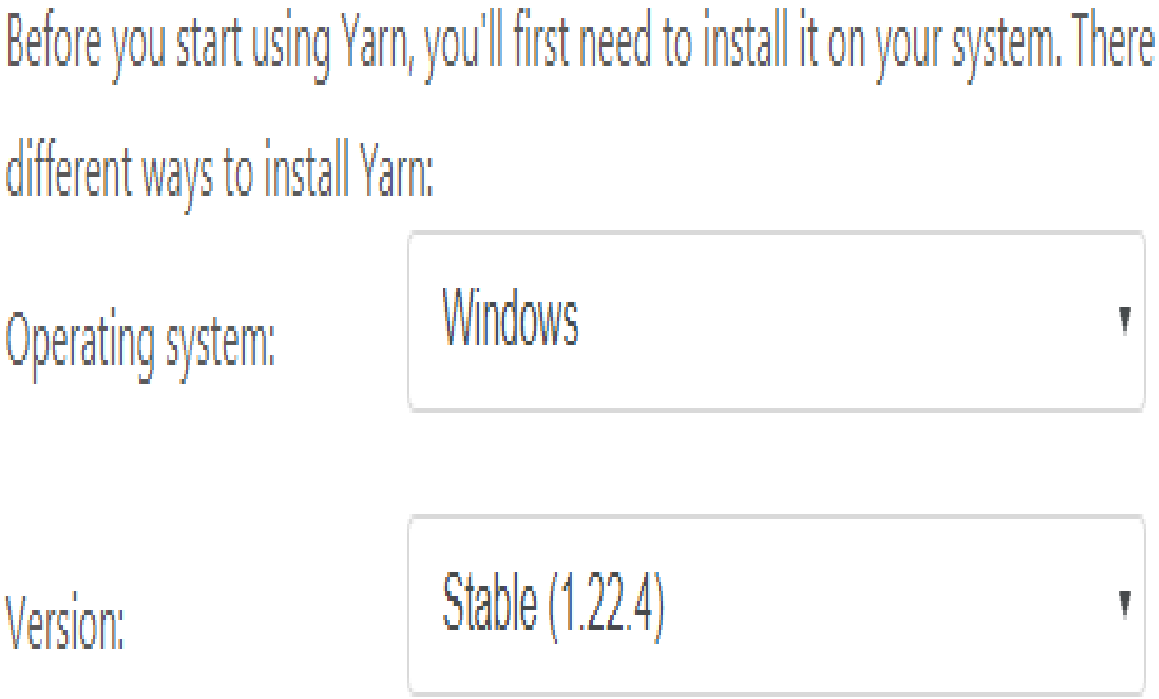
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>,
document.getElementById('app'));
setTimeout(() => {
ReactDOM.unmountComponentAtNode(document.g
etElementById('app'));}, 10000);
```

Uplatz

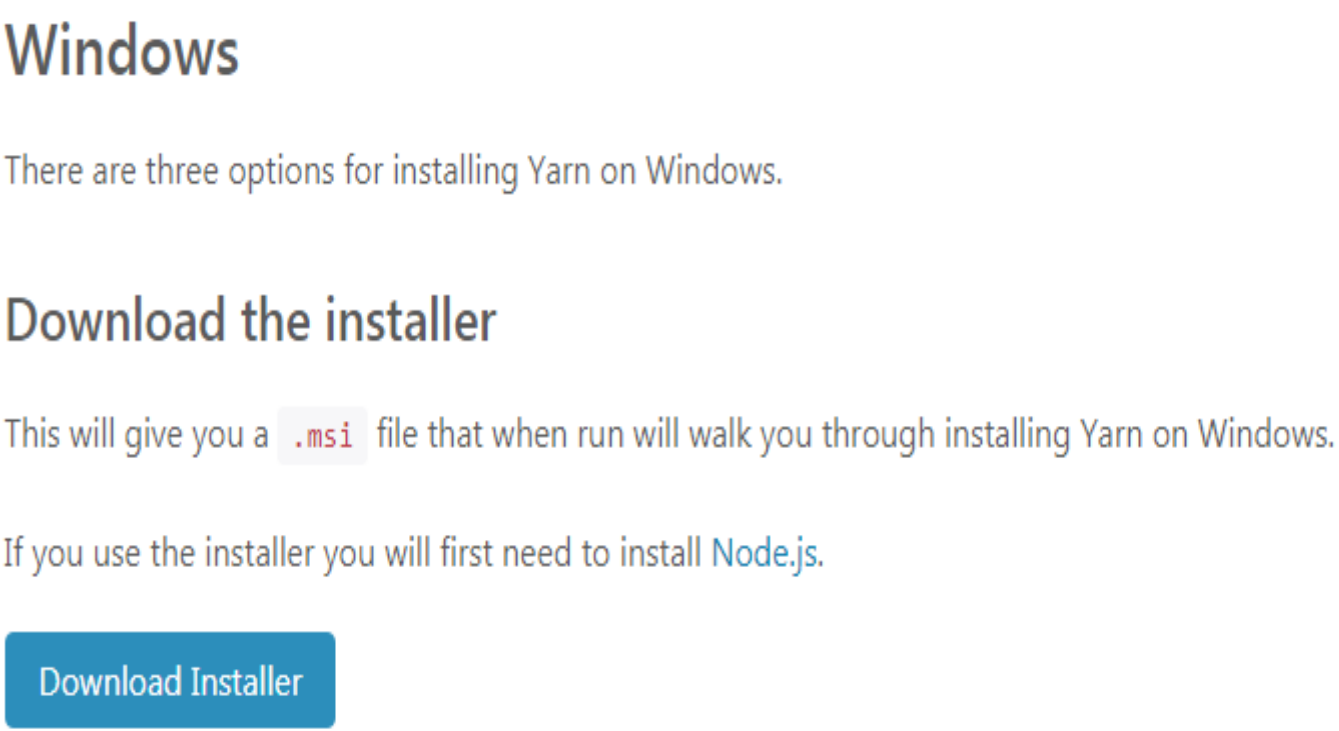
Installation of Yarn:

Step1:

https://classic.yarnpkg.com/en/docs/install/#windows-stable



Step2:
Install the msi package



Summary:

- ReactJS component written in TypeScript
- Installation and Setup
- Stateless React Components in TypeScript
- Stateless and property-less Components
- Life-cycle methods
- Installation of Yarn

Thank You.....

If you have any quries please write to info@uplatz.com".

Uplatz