Skip to main content

Documentation ▾

JSX

Introduction

JSX (https://facebook.github.io/jsx/) is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX rose to popularity with the React (https://reactjs.org/) framework, but has since seen other implementations as well. TypeScript supports embedding, type checking, and compiling JSX directly to JavaScript.

Basic usage

In order to use JSX you must do two things.

1. Name your files with a `.tsx` extension
2. Enable the `jsx` option

TypeScript ships with three JSX modes: `preserve`, `react`, and `react-native`. These modes only affect the emit stage - type checking is unaffected. The `preserve` mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. Babel (https://babeljs.io/)). Additionally the output will have a `.jsx` file extension. The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension. The `react-native` mode is the equivalent of `preserve` in that it keeps all JSX, but the output will instead have a `.js` file extension.

Mode	Input	Output	Output File Extension
`preserve`	`<div />`	`<div />`	`.jsx`
`react`	`<div />`	`React.createElement("div")`	`.js`
`react-native`	`<div />`	`<div />`	`.js`

You can specify this mode using either the `--jsx` command line flag or the corresponding option in your tsconfig.json (./tsconfig-json.html) file.

> *Note: The identifier `React` is hard-coded, so you must make React available with an uppercase R.*

The `as` operator

Recall how to write a type assertion:

```
var foo = <foo>bar;
```

This asserts the variable `bar` to have the type `foo`. Since TypeScript also uses angle brackets for type assertions, combining it with JSX's syntax would introduce certain parsing difficulties. As a result, TypeScript disallows angle bracket type assertions in `.tsx` files.

Since the above syntax cannot be used in `.tsx` files, an alternate type assertion operator should be used: `as`. The example can easily be rewritten with the `as` operator.

```
var foo = bar as foo;
```

The `as` operator is available in both `.ts` and `.tsx` files, and is identical in behavior to the angle-bracket type assertion style.

Type Checking

In order to understand type checking with JSX, you must first understand the difference between intrinsic elements and value-based elements. Given a JSX expression `<expr />`, `expr` may either refer to something intrinsic to the environment (e.g. a `div` or `span` in a DOM environment) or to a custom component that you've created. This is important for two reasons:

1. For React, intrinsic elements are emitted as strings (`React.createElement("div")`), whereas a component you've created is not (`React.createElement(MyComponent)`).
2. The types of the attributes being passed in the JSX element should be looked up differently. Intrinsic element attributes should be known *intrinsically* whereas components will likely want to specify their own set of attributes.

TypeScript uses the same convention that React does (http://facebook.github.io/react/docs/jsx-in-depth.html#html-tags-vs.-react-components) for distinguishing between these. An intrinsic element always begins with a lowercase letter, and a value-based element always begins with an uppercase letter.

Intrinsic elements

Intrinsic elements are looked up on the special interface `JSX.IntrinsicElements`. By default, if this interface is not specified, then anything goes and intrinsic elements will not be type checked. However, if this interface *is* present, then the name of the intrinsic element is looked up as a property on the `JSX.IntrinsicElements` interface. For example:

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: any
    }
}

<foo />; // ok
<bar />; // error
```

In the above example, `<foo />` will work fine but `<bar />` will result in an error since it has not been specified on `JSX.IntrinsicElements`.

> Note: You can also specify a catch-all string indexer on `JSX.IntrinsicElements` as follows:
>
> ```
> declare namespace JSX {
>    interface IntrinsicElements {
>        [elemName: string]: any;
>    }
> }
> ```

Value-based elements

Value based elements are simply looked up by identifiers that are in scope.

```
import MyComponent from "./myComponent";

<MyComponent />; // ok
<SomeOtherComponent />; // error
```

There are two ways to define a value-based element:

1. Stateless Functional Component (SFC)
2. Class Component

Because these two types of value-based elements are indistinguishable from each other in a JSX expression, first TS tries to resolve the expression as Stateless Functional Component using overload resolution. If the process succeeds, then TS finishes resolving the expression to its declaration. If the value fails to resolve as SFC, TS will then try to resolve it as a class component. If that fails, TS will report an error.

Stateless Functional Component

As the name suggests, the component is defined as JavaScript function where its first argument is a `props` object. TS enforces that its return type must be assignable to `JSX.Element` .

```
interface FooProp {
  name: string;
  X: number;
  Y: number;
}

declare function AnotherComponent(prop: {name: string});
function ComponentFoo(prop: FooProp) {
  return <AnotherComponent name={prop.name} />;
}

const Button = (prop: {value: string}, context: { color: string }) => <button>
```

Because an SFC is simply a JavaScript function, function overloads may be used here as well:

```
interface ClickableProps {
  children: JSX.Element[] | JSX.Element
}

interface HomeProps extends ClickableProps {
  home: JSX.Element;
}

interface SideProps extends ClickableProps {
  side: JSX.Element | string;
}

function MainButton(prop: HomeProps): JSX.Element;
function MainButton(prop: SideProps): JSX.Element {
  ...
}
```

Class Component

It is possible to define the type of a class component. However, to do so it is best to understand two new terms: the *element class type* and the *element instance type*.

Given `<Expr />`, the *element class type* is the type of `Expr`. So in the example above, if `MyComponent` was an ES6 class the class type would be that class's constructor and statics. If `MyComponent` was a factory function, the class type would be that function.

Once the class type is established, the instance type is determined by the union of the return types of the class type's construct or call signatures (whichever is present). So again, in the case of an ES6 class, the instance type would be the type of an instance of that class, and in the case of a factory function, it would be the type of the value returned from the function.

```
class MyComponent {
  render() {}
}

// use a construct signature
var myComponent = new MyComponent();

// element class type => MyComponent
// element instance type => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// use a call signature
var myComponent = MyFactoryFunction();

// element class type => FactoryFunction
// element instance type => { render: () => void }
```

The element instance type is interesting because it must be assignable to `JSX.ElementClass` or it will result in an error. By default `JSX.ElementClass` is `{}`, but it can be augmented to limit the use of JSX to only those types that conform to the proper interface.

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // ok
<MyFactoryFunction />; // ok

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // error
<NotAValidFactoryFunction />; // error
```

Attribute type checking

The first step to type checking attributes is to determine the *element attributes type*. This is slightly different between intrinsic and value-based elements.

For intrinsic elements, it is the type of the property on `JSX.IntrinsicElements`

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean }
  }
}

// element attributes type for 'foo' is '{bar?: boolean}'
<foo bar />;
```

For value-based elements, it is a bit more complex. It is determined by the type of a property on the *element instance type* that was previously determined. Which property to use is determined by `JSX.ElementAttributesProperty` . It should be declared with a single property. The name of that property is then used. As of TypeScript 2.8, if `JSX.ElementAttributesProperty` is not provided, the type of first parameter of the class element's constructor or SFC's call will be used instead.

```
declare namespace JSX {
  interface ElementAttributesProperty {
    props; // specify the property name to use
  }
}

class MyComponent {
  // specify the property on the element instance type
  props: {
    foo?: string;
  }
}

// element attributes type for 'MyComponent' is '{foo?: string}'
<MyComponent foo="bar" />
```

The element attribute type is used to type check the attributes in the JSX. Optional and required properties are supported.

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number }
  }
}

<foo requiredProp="bar" />; // ok
<foo requiredProp="bar" optionalProp={0} />; // ok
<foo />; // error, requiredProp is missing
<foo requiredProp={0} />; // error, requiredProp should be a string
<foo requiredProp="bar" unknownProp />; // error, unknownProp does not exist
<foo requiredProp="bar" some-unknown-prop />; // ok, because 'some-unknown-prop' is not a valid identifier
```

> Note: If an attribute name is not a valid JS identifier (like a `data-*` attribute), it is not considered to be an error if it is not found in the element attributes type.

Additionally, the `JSX.IntrinsicAttributes` interface can be used to specify extra properties used by the JSX framework which are not generally used by the components' props or arguments - for instance `key` in React. Specializing further, the generic `JSX.IntrinsicClassAttributes<T>` type may also be used to specify the same kind of extra attributes just for class components (and not SFCs). In this type, the generic parameter corresponds to the class instance type. In React, this is used to allow the `ref` attribute of type `Ref<T>`. Generally speaking, all of the properties on these interfaces should be optional, unless you intend that users of your JSX framework need to provide some attribute on every tag.

The spread operator also works:

```
var props = { requiredProp: "bar" };
<foo {...props} />; // ok

var badProps = {};
<foo {...badProps} />; // error
```

Children Type Checking

In TypeScript 2.3, TS introduced type checking of *children*. *children* is a special property in an *element attributes type* where child *JSXExpression*s are taken to be inserted into the attributes. Similar to how TS uses `JSX.ElementAttributesProperty` to determine the name of *props*, TS uses `JSX.ElementChildrenAttribute` to determine the name of *children* within those props. `JSX.ElementChildrenAttribute` should be declared with a single property.

```
declare namespace JSX {
  interface ElementChildrenAttribute {
    children: {};  // specify children name to use
  }
}
```

```
<div>
  <h1>Hello</h1>
</div>;

<div>
  <h1>Hello</h1>
  World
</div>;

const CustomComp = (props) => <div>props.children</div>
<CustomComp>
  <div>Hello World</div>
  {"This is just a JS expression..." + 1000}
</CustomComp>
```

You can specify the type of *children* like any other attribute. This will override the default type from, eg the React typings (https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/react) if you use them.

```
interface PropsType {
  children: JSX.Element
  name: string
}

class Component extends React.Component<PropsType, {}> {
  render() {
    return (
      <h2>
        {this.props.children}
      </h2>
    )
  }
}

// OK
<Component>
  <h1>Hello World</h1>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element
<Component>
  <h1>Hello World</h1>
  <h2>Hello World</h2>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element or string.
<Component>
  <h1>Hello</h1>
  World
</Component>
```

The JSX result type

By default the result of a JSX expression is typed as `any`. You can customize the type by specifying the `JSX.Element` interface. However, it is not possible to retrieve type information about the element, attributes or children of the JSX from this interface. It is a black box.

Embedding Expressions

JSX allows you to embed expressions between tags by surrounding the expressions with curly braces (`{ }`).

```
var a = <div>
  {["foo", "bar"].map(i => <span>{i / 2}</span>)}
</div>
```

The above code will result in an error since you cannot divide a string by a number. The output, when using the `preserve` option, looks like:

```
var a = <div>
  {["foo", "bar"].map(function (i) { return <span>{i / 2}</span>; })}
</div>
```

React integration

To use JSX with React you should use the React typings (https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/react). These typings define the `JSX` namespace appropriately for use with React.

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent foo="bar" />; // ok
<MyComponent foo={0} />; // error
```

Factory Functions

The exact factory function used by the `jsx: react` compiler option is configurable. It may be set using either the `jsxFactory` command line option, or an inline `@jsx` comment pragma to set it on a per-file basis. For example, if you set `jsxFactory` to `createElement`, `<div />` will emit as `createElement("div")` instead of `React.createElement("div")`.

The comment pragma version may be used like so (in TypeScript 2.8):

```
import preact = require("preact");
/* @jsx preact.h */
const x = <div />;
```

emits as:

```
const preact = require("preact");
const x = preact.h("div", null);
```

The factory chosen will also affect where the `JSX` namespace is looked up (for type checking information) before falling back to the global one. If the factory is defined as `React.createElement` (the default), the compiler will check for `React.JSX` before checking for a global `JSX`. If the factory is defined as `h`, it will check for `h.JSX` before a global `JSX`.

Made with ♥ in Redmond Follow @Typescriptlang (https://twitter.com/typescriptlang)

Privacy (https://go.microsoft.com/fwlink/?LinkId=521839)