PropTypes

Props are a very important mechanism for passing read-only attributes to React components. These attributes are usually required to be of a certain type or form for them to be used properly in the component.

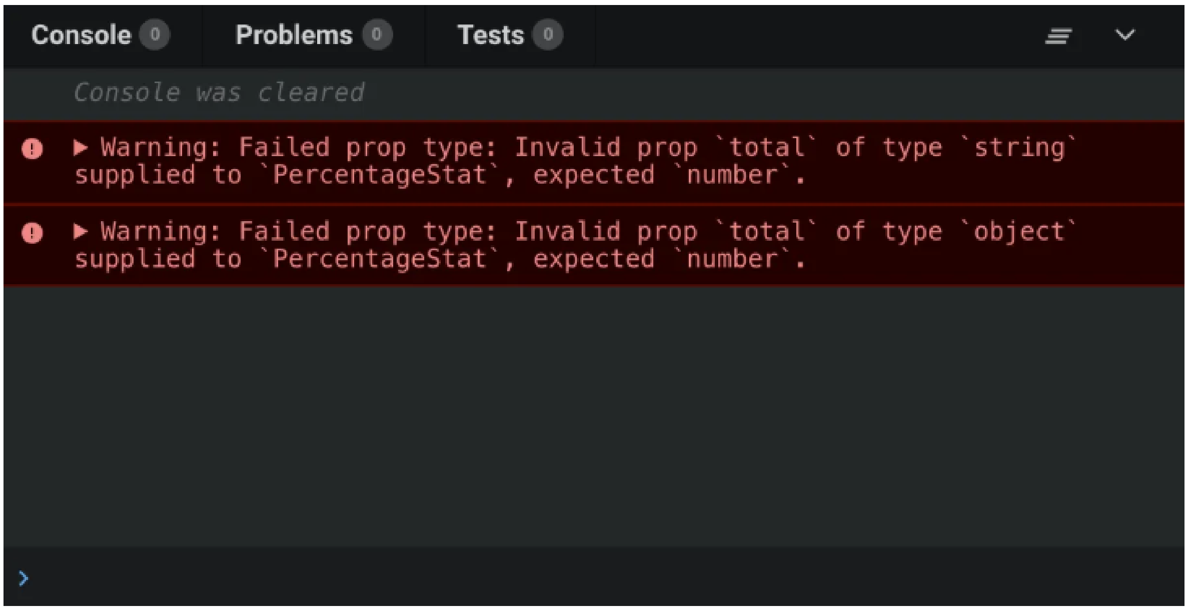
If a prop is passed to a component in a type or form that is unfamiliar, the component may not behave as we intend it to. Thus, a great way of improving React components is props validation.

propTypes:

React provides an internal mechanism for adding typechecking to components. React components use a special property named **propTypes** to set up typechecking.

PropTypes

When props are passed to a React component, they are checked against the type definition configured in the **propTypes** property. When an invalid value is passed for a prop, a warning is displayed on the JavaScript console.



```
Console was cleared
▶ Warning: Failed prop type: Invalid prop `total` of type `string`
supplied to `PercentageStat`, expected `number`.
▶ Warning: Failed prop type: Invalid prop `total` of type `object`
supplied to `PercentageStat`, expected `number`.
```

React.PropTypes:

Prior to React 15.5.0, a utility named PropTypes was available, as part of the React package, which provided a lot of validators for configuring type definitions for component props. It could be accessed with **React.PropTypes**.

However, in later versions of React, this utility has been moved to a separate package named prop-types, so you need to add it as a dependency for your project in order to get access to the **PropTypes** utility:

```
npm install prop-types --save-dev
```

It can be imported into your project files as follows:

```
import PropTypes from "prop-types";
```

You could use it as follows:

```
/** FUNCTIONAL COMPONENTS **/
function ReactComponent(props) {
 // ...implement render logic here
}

ReactComponent.propTypes = {
 // ...prop type definitions here
};
```

PropTypes validators:

As stated in the previous section, the `PropTypes` utility exports a lot of validators for configuring type definitions. Here are the validators for the basic data types:

`PropTypes.any // PropTypes.bool // PropTypes.number // PropTypes.string // PropTypes.func // PropTypes.array // PropTypes.object // PropTypes.symbol`

```
Component.propTypes = {
 anyProp: PropTypes.any,
 booleanProp: PropTypes.bool,
 numberProp: PropTypes.number,
 stringProp: PropTypes.string,
 functionProp: PropTypes.func
};
```

Multiple types:

PropTypes also export validators that can allow a limited set of values or multiple sets of data types for a prop. Here they are:

- **PropTypes.oneOf:** The prop is limited to a specified set of values, treating it like an enum.
- **PropTypes.oneOfType:** The prop should be one of a specified set of types, behaving like a union of types.

```
Component.propTypes = {
 enumProp: PropTypes.oneOf([true, false, 0, "Unknown"]),

 unionProp: PropTypes.oneOfType([
   PropType.bool,
   PropType.number,
   PropType.string,
   PropType.instanceOf(Person)
 ])
};
```

Required props:

Oftentimes you need to control your component's output. You could do that by conditional rendering.

You can prevent the component from rendering whenever a required prop is not passed or is invalid by rendering an alternative output instead. That is done by using the conditional rendering.

```
const SimpleComponent = ({ requiredProps }) => {
 return (
   <>
     {requiredProps ? (
       <h1>We need this {requiredProps} !</h1>
     ) : (
       <h1>Ooops ! we need that props</h1>
     )}
   </>
```

```
  );
};
```

We set an alternative for our output with **requiredProps** .

‹ Previous next ›