Cheatsheets for experienced React developers getting started with TypeScript

Basic | Advanced | Migrating | HOC | 中文翻译 | Contribute! | Ask!

HOC Cheatsheet

This HOC Cheatsheet compiles all available knowledge for writing Higher Order Components with React and TypeScript.

- We will map closely to the official docs on HOCs initially
- While hooks exist, many libraries and codebases still have a need to type HOCs.
- Render props may be considered in future

- The goal is to write HOCs that offer type safety while not getting in the way.

HOC Cheatsheet Table of Contents

▶ Expand Table of Contents

Section 1: React HOC docs in TypeScript

In this first section we refer closely to the React docs on HOCs and offer direct TypeScript parallels.

Docs Example: Use HOCs For Cross-Cutting Concerns

▶ Misc variables referenced in the example below

Example HOC from React Docs translated to TypeScript

```
// these are the props to be injected by the HOC interface WithDataProps<T> { data: T; // data is generic
} // T is the type of data // P is the props of the wrapped component that is inferred // C is the actual
interface of the wrapped component (used to grab defaultProps from it) export function withSubscription<T,
P extends WithDataProps<T>, C>( // this type allows us to infer P, but grab the type of WrappedComponent
separately without it interfering with the inference of P WrappedComponent: JSXElementConstructor<P> & C,
// selectData is a functor for T // props is Readonly because it's readonly inside of the class
selectData: ( dataSource: typeof DataSource, props: Readonly<JSX.LibraryManagedAttributes<C, Omit<P,
'data'>>> ) => T ) { // the magic is here: JSX.LibraryManagedAttributes will take the type of
WrapedComponent and resolve its default props // against the props of WithData, which is just the original
P type with 'data' removed from its requirements type Props = JSX.LibraryManagedAttributes<C, Omit<P,
'data'>>; type State = { data: T; }; return class WithData extends Component<Props, State> {
constructor(props: Props) { super(props); this.handleChange = this.handleChange.bind(this); this.state = {
data: selectData(DataSource, props) }; } componentDidMount = () =>
DataSource.addChangeListener(this.handleChange); componentWillUnmount = () =>
DataSource.removeChangeListener(this.handleChange); handleChange = () => this.setState({ data:
selectData(DataSource, this.props) }); render() { // the typing for spreading this.props is... very
complex. best way right now is to just type it as any // data will still be typechecked return
<WrappedComponent data={this.state.data} {...this.props as any} />; } }; // return WithData; } /** HOC
usage with Components */ export const CommentListWithSubscription = withSubscription( CommentList,
(DataSource: DataType) => DataSource.getComments() ); export const BlogPostWithSubscription =
withSubscription( BlogPost, (DataSource: DataType, props: Omit<BlogPostProps, 'data'>) =>
DataSource.getBlogPost(props.id) );
```

Docs Example: Don’t Mutate the Original Component. Use Composition.

This is pretty straightforward - make sure to assert the passed props as `T` due to the TS 3.2 bug.

```
function logProps<T>(WrappedComponent: React.ComponentType<T>) { return class extends React.Component {
componentWillReceiveProps( nextProps: React.ComponentProps<typeof WrappedComponent> ) {
console.log('Current props: ', this.props); console.log('Next props: ', nextProps); } render() { // Wraps
the input component in a container, without mutating it. Good! return <WrappedComponent {...this.props as
T} />; } }; }
```

Docs Example: Pass Unrelated Props Through to the Wrapped Component

No TypeScript specific advice needed here.

Docs Example: Maximizing Composability

HOCs can take the form of Functions that return Higher Order Components that return Components.

`connect` from `react-redux` has a number of overloads you can take inspiration from in the source.

Here we build our own mini `connect` to understand HOCs:

▶ Misc variables referenced in the example below

```
const commentSelector = (_: any, ownProps: any) => ({ id: ownProps.id }); const commentActions = () => ({
addComment: (str: string) => comments.push({ text: str, id: comments.length }) }); const ConnectedComment
= connect( commentSelector, commentActions )(CommentList); // these are the props to be injected by the
HOC interface WithSubscriptionProps<T> { data: T; } function connect(mapStateToProps: Function,
mapDispatchToProps: Function) { return function<T, P extends WithSubscriptionProps<T>, C>(
WrappedComponent: React.ComponentType<T> ) { type Props = JSX.LibraryManagedAttributes<C, Omit<P,
'data'>>; // Creating the inner component. The calculated Props type here is the where the magic happens.
return class ComponentWithTheme extends React.Component<Props> { public render() { // Fetch the props you
want inject. This could be done with context instead. const mappedStateProps = mapStateToProps(this.state,
this.props); const mappedDispatchProps = mapDispatchToProps(this.state, this.props); // this.props comes
afterwards so the can override the default ones. return ( <WrappedComponent {...this.props}
{...mappedStateProps} {...mappedDispatchProps} /> ); } }; }; }
```

Docs Example: Wrap the Display Name for Easy Debugging

This is pretty straightforward as well.

```
interface WithSubscriptionProps { data: any; } function withSubscription< T extends WithSubscriptionProps
= WithSubscriptionProps >(WrappedComponent: React.ComponentType<T>) { class WithSubscription extends
React.Component { /* ... */ public static displayName = `WithSubscription(${getDisplayName(
WrappedComponent )})`; } return WithSubscription; } function getDisplayName<T>(WrappedComponent:
React.ComponentType<T>) { return WrappedComponent.displayName || WrappedComponent.name || 'Component'; }
```

Unwritten: Caveats section

- Don't Use HOCs Inside the render Method
- Static Methods Must Be Copied Over
- Refs Aren't Passed Through