Typescript in React

Learn how Typescript integration helps in a React application.

TypeScript for JavaScript and React have many benefits for developing robust applications. Instead of getting type errors on runtime in the command line or browser, TypeScript integration presents them during compile time inside the IDE. It shortens the feedback loop of JavaScript development. While it improves the developer experience, the code also becomes more self-documenting and readable, because every variable is defined with a type. Also moving code blocks or performing a larger refactoring of a code base becomes much more efficient. Statically typed languages like TypeScript are trending because of their benefits over dynamically typed languages like JavaScript. It's useful to learn more about Typescript whenever possible.

To use TypeScript in React, install TypeScript and its dependencies into your application using the command line. If you run into obstacles, follow the official TypeScript installation instructions for create-react-app:

```
npm install --save typescript @types/node @types/react
npm install --save typescript @types/react-dom @types/jest
```

Next, rename all JavaScript files (*.js*) to TypeScript files (*.tsx*).

```
mv src/index.js src/index.tsx
mv src/App.js src/App.tsx
```

Restart your development server in the command line. You may encounter compile errors in the browser and IDE. If the latter doesn't work, try installing a TypeScript plugin for your editor, or extension for your IDE. After the initial TypeScript in React setup, we'll add type safety for the entire *src/App.tsx* file, starting with typing the arguments of the custom hook:

```
const useSemiPersistentState = (
  key: string,
  initialState: string
) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

src/App.tsx

Adding types to the function's arguments is more about Javascript than React. We are telling the function to expect two arguments, which are JavaScript string primitives. Also, we can tell the function to return an array (`[]`) with a `string` (state), and tell functions like *state updater function* that take a `value` to return nothing (`void`):

```
const useSemiPersistentState = (
  key: string,
  initialState: string

): [string, (newValue: string) => void] => {

  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

src/App.tsx

Related to React though, considering the previous type safety improvements for the custom hook, we hadn't to add types to the internal React hooks in the function's body. That's because **type inference** works most of the time for React hooks out of the box. If the *initial state* of a React `useState` Hook is a JavaScript string primitive, then the returned *current state* will be inferred as a string and the returned *state updater function* will only take a string as argument and return nothing:

```
const [value, setValue] = React.useState('React');
// value is inferred to be a string
// setValue only takes a string as argument
```

src/App.tsx

If adding type safety becomes an aftermath for a React application and its components, there are multiple ways on how to approach it. We will start with the props and state for the leaf components of our application. For instance, the Item component receives a story (here `item`) and a callback handler function (here `onRemoveItem`). Starting out very verbose, we could add the inlined types for both function arguments as we did before:

```
const Item = ({
  item,
  onRemoveItem,

}: {
  item: {
    objectID: string;
    url: string;
    title: string;
    author: string;
    num_comments: number;
    points: number;
  };
  onRemoveItem: (item: {
    objectID: string;
    url: string;
    title: string;
    author: string;
    num_comments: number;
    points: number;
  }) => void;
}) => (

  <div>
    ...
  </div>
);
```

src/App.tsx

There are two problems: the code is verbose, and it has duplications. Let's get rid of both problems by defining a custom `Story` type outside the component, at the top of *src/App.js*:

```
type Story = {
  objectID: string;
  url: string;
  title: string;
  author: string;

  num_comments: number;
  points: number;
};

const Item = ({
  item,
  onRemoveItem,

}: {
  item: Story;
  onRemoveItem: (item: Story) => void;
}) => (

  <div>
    ...
  </div>
);
```

src/App.tsx

The `item` is of type `Story`; the `onRemoveItem` function takes an `item` of type `Story` as an argument and returns nothing. Next, clean up the code by defining the props of Item component outside:

```
type ItemProps = {
  item: Story;
  onRemoveItem: (item: Story) => void;
};
const Item = ({ item, onRemoveItem }: ItemProps) => (

  <div>
    ...
  </div>
);
~~~
```

src/App.tsx

That's the most popular way to type React component's props with TypeScript. From here, we can navigate up the component tree into the `List` component and apply the same type definitions for the props:

```
type Story = {
  ...
};

type Stories = Array<Story>;

...

type ListProps = {
```

```
  list: Stories;
  onRemoveItem: (item: Story) => void;
};

const List = ({ list, onRemoveItem }: ListProps) =>
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

src/App.tsx

The `onRemoveItem` function is typed twice for the `ItemProps` and `ListProps`. To be more accurate, you *could* extract this to a standalone defined `OnRemoveItem` TypeScript type and reuse it for both `onRemoveItem` prop type definitions. Note, however, that development becomes increasingly difficult as components are split up into different files. That's why we will keep the duplication here.

Since we already have the `Story` and `Stories` types, we can repurpose them for other components. Add the `Story` type to the callback handler in the `App` component:

```
const App = () => {
  ...

  const handleRemoveStory = (item: Story) => {
    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });
  };

  ...
};
```

src/App.tsx

The reducer function manages the `Story` type as well, except without looking into the `state` and `action` types. As the application's developer, we know both objects and their shapes passed to this reducer function:

```
type StoriesState = {
  data: Stories;
  isLoading: boolean;
  isError: boolean;
};
type StoriesAction = {
  type: string;
  payload: any;
};

const storiesReducer = (

  state: StoriesState,
  action: StoriesAction
) => {


  ...
};
```

src/App.tsx

The `StoriesAction` type with its `string` and `any` (TypeScript **wildcard**) type definitions are still too broad; and we gain no real type safety through it, because actions are not distinguishable. We can do better by specifying each action TypeScript type as an **interface**, and using a **union type** (here `StoriesAction`) for the final type safety:

```
interface StoriesFetchInitAction {
  type: 'STORIES_FETCH_INIT';
}
interface StoriesFetchSuccessAction {
  type: 'STORIES_FETCH_SUCCESS';
  payload: Stories;
}

interface StoriesFetchFailureAction {
  type: 'STORIES_FETCH_FAILURE';
}

interface StoriesRemoveAction {
  type: 'REMOVE_STORY';
  payload: Story;
}

type StoriesAction =
  | StoriesFetchInitAction
  | StoriesFetchSuccessAction
  | StoriesFetchFailureAction
  | StoriesRemoveAction;


const storiesReducer = (
  state: StoriesState,
  action: StoriesAction
) => {
  ...
};
```

src/App.tsx

The stories state, the current state and the action are types; the returned new state (inferred) are type safe now. For instance, if you would dispatch an action to the reducer with an action type that's not defined, you would get a type error. Or if you would pass something else than a story to the action which removes a story, you would get a type error as well.

There is still a type safety issue in the App component's return statement for the returned List component. It can be fixed by giving the List component a wrapping HTML `div` element or a React fragment:

```
const List = ({ list, onRemoveItem }: ListProps) => (
  <>
    {list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </>
);
```

src/App.tsx

> According to a TypeScript with React issue on GitHub: *"This is because due to limitations in the compiler, function components cannot return anything other than a JSX expression or null, otherwise it complains with a cryptic error message saying that the other type is not assignable to Element."*

Let's shift our focus to the SearchForm component, which has callback handlers with events:

```
type SearchFormProps = {
  searchTerm: string;
  onSearchInput: (event: React.ChangeEvent<HTMLInputElement>) => void;
  onSearchSubmit: (event: React.FormEvent<HTMLFormElement>) => void;
};
const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,

}: SearchFormProps) => (

  ...
);
```

src/App.tsx

Often using `React.SyntheticEvent` instead of `React.ChangeEvent` or `React.FormEvent` is usually enough. Going up to the App component again, we apply the same type for the callback handler there:

```
const App = () => {
  ...

  const handleSearchInput = (
    event: React.ChangeEvent<HTMLInputElement>
  ) => {
    setSearchTerm(event.target.value);
  };

  const handleSearchSubmit = (

    event: React.FormEvent<HTMLFormElement>

  ) => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);

    event.preventDefault();
  };

  ...
};
```

src/App.tsx

All that's left is the InputWithLabel component. Before handling this component's props, let's take a look at the `ref` from React's useRef Hook. Unfortunately, the return value isn't inferred:

```
const InputWithLabel = ({ ... }) => {

  const inputRef = React.useRef<HTMLInputElement>(null!);

  React.useEffect(() => {
    if (isFocused && inputRef.current) {
      inputRef.current.focus();
    }
  }, [isFocused]);
```

src/App.tsx

We made the returned `ref` type safe, and typed it as read-only because we only execute the `focus` method on it (read). React takes over for us there, setting the DOM element to the `current` property.

Lastly, we will apply type safety checks for the InputWithLabel component's props. Note the `children` prop with its React specific type and the **optional types** signaled with a question mark:

```
type InputWithLabelProps = {
  id: string;
  value: string;
  type?: string;

  onInputChange: (event: React.ChangeEvent<HTMLInputElement>) => void;
  isFocused?: boolean;
  children: React.ReactNode;
};
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,

}: InputWithLabelProps) => {

  ...
};
```

src/App.tsx

Both the `type` and `isFocused` properties are optional. Using TypeScript, you can tell the compiler these don't need to be passed to the component as props. The `children` prop has a lot of TypeScript type definitions that could be applicable to this concept, the most universal of which is `React.ReactNode` from the React library.

Our entire React application is finally typed by TypeScript, making it easy to spot type errors on compile time. When adding TypeScript to your React application, start by adding type definitions to your function's arguments. These functions can be vanilla JavaScript functions, custom React hooks, or React function components. Only when using React is it important to know specific types for form elements, events, and JSX.

```
 □ □ □□  □   ã□  F   □□  □   □  )□      □   9□  5□  @@  □    °□   n□   □PNG
□
  IHDR   □   □□□   (-□S   äPLTE""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
  IHDR   □   □□□   ×©ÍÊ  □ePLTE"""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5ù  ïçd
□
  IHDR       □□   D¤□Æ  □APLTE   """"""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}X
□
  IHDR   @   @□□    □·□ì   □:PLTE   """""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
©¬¢Z\Ts©R*□(□  ¯©□J□□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&i¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□ô□I_
-ê>Û□º«¢XÕ¢î}ß¨ëÜÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□½+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'/Å]_□
```

Exercises:

- Confirm the changes from the last section.

- Dig into the React + TypeScript Cheatsheet, because most common use cases we faced in this section are covered there as well. There is no need to know everything from the top off your head.
- While you continue with the learning experience in the following sections, remove or keep your types with TypeScript. If you do the latter, add new types whenever you get a compile error.

Test yourself!

Q

What is the advantage of integrating typescript with react application?

Retake Quiz