plotly

Star 23,446

Dash Python > ***React for Python Developers***

React for Python Developers: a primer

Introduction

If you're a Dash developer, at some point or another you probably have thought about writing your own set of components for Dash. You might have even taken a peek at some of our source code, or taken the `dash-component-boilerplate` for a spin.

However, if you've never programmed in JavaScript and/or used React before, you might feel slightly confused. By the end of this guide, you should feel comfortable creating your own Dash component in React and JavaScript, even if you have never programmed in those languages before.

> If you're interested in creating components using Python, instead of React, see the **All-in-One components chapter.**

> Dash 2.4 and later supports components created with React and TypeScript. See the **Dash Components with TypeScript and React** section below for detail on creating a Dash TypeScript component.

JavaScript

JavaScript is the language of the web - all modern browsers can run it, and most modern web pages use it to make their pages interactive. It is the de-facto standard of front end development, and has come a long way since its inception. Today, modern JavaScript has a rich set of features, designed to create a development experience perfectly suited for the web.

React

React is JavaScript library for building user interfaces, written and maintained by Facebook. It has been very popular over the last few years because it brings the power of reactive, declarative programming to the world of front end development.

React has made it easier to think about user interface code, and its programming model encourages code that's modular and reusable. It also has a huge, vibrant open-source community that has published all sorts of reusable UI components, from sliders to data tables, dropdowns to buttons.

It is important to realise that React is just JavaScript. React is not a language on its own, nor is it a domain-specific framework that takes years to master. It has a relatively small API, with just a few functions and paradigms to learn before you, too, can use it to write applications for the web. That being said, all unfamiliar technology will have a learning curve, but with practice and patience you will master it!

Dash uses React under the hood to render the user interface you see when you load a web page created with Dash. Because React allows you to write your user interface in encapsulated components that manage their own state, it is easy to split up parts of code for Dash too. At the end of this tutorial, you will see that Dash components and React components map one to one!

For now, the important thing to know is that Dash components are mostly simple wrappers around existing React components. This means the entire React ecosystem is potentially usable in a Dash application!

Installing everything you need

Let's start by setting up our JavaScript development environment. We will use Node.js, NPM, and our `dash-component-boilerplate` to write our first React application. Node.js is a JavaScript runtime, which allows you to run JavaScript code outside of the browser. Just like you would run `python my-code.py` to run Python code in a terminal, you'd run `node my-code.js` to run JavaScript code in a terminal.

Node comes in very handy when developing, even when you intend to run the code in the browser.

NPM

NPM is the "Node Package Manager" and it is used to install packages and run scripts. Besides being a package manager (like `pip` for Python), `npm` also allows you to run scripts and perform tasks, such as creating a project for you (`npm init`), starting up a project (`npm start`), or firing custom scripts (`npm run custom-script`). These scripts are defined in a `package.json` file, which every project that uses `npm` has.

The `package.json` file holds your `requirements` and `devRequirements`, which can be installed using `npm install`, the same way `pip` has a `requirements.txt` option you can use in `pip install -r requirements.txt`.

`package.json` also holds a `scripts` section where custom scripts can be defined. It is usually a good idea to check out a new project's `package.json` file to see which scripts the project uses.

If you go to the **dash-component-boilerplate repo**, you'll find instructions for setting up some React boilerplate code. This code will help you quickly set up a React development environment, complete with the necessary scripts for building our React component for Dash.

These scripts will use a variety of technologies (e.g. Babel, Webpack, and more) to compile our code into a web-ready package.

- To install Node.js, go to **the Node.js website** to download the latest version. We recommend installing the LTS version.
- Node.js will automatically install the Node Package Manager `npm` on your machine
- Verify that node is installed by running: `node -v`
- Verify that npm is installed by running: `npm -v`

Python

You will need python to generate your components classes to work with Dash.

Download python on the official website or through your os distribution package manager.

- **https://www.python.org/**
- `apt-get install python`/`yum install python`

Virtual environments

It is best to use virtual environments when working on projects, we recommend creating a fresh virtual environment for each project you have so they can have specific requirements and remain isolated from your main python environment.

In Python 2 you have to use `virtualenv`:

`pip install virtualenv` Then you create a venv with the command `virtualenv venv`, it will create a folder `venv` in the current directory with your new environment.

In Python 3 you can use the builtin module `venv`:

`python -m venv venv`

Cookiecutter boilerplate

Now that we have Node.js and Python up and running, we can generate a dash component project using the **cookiecutter dash-component-boilerplate**.

The boilerplate is built using **cookiecutter**, a project template renderer made with jinja2. This allows users to create a project with custom values formatted for the project.

Install cookiecutter:

```
pip install cookiecutter
```

Generate a new dash component project

- Run the cookiecutter: `cookiecutter https://github.com/plotly/dash-component-boilerplate.git`
- Answer the questions about the project:
    - `project_name`: A display name for the project, can contain spaces and uppercase letters, for example `Example Component`.
    - `project_shortname`: A variable derived from project_name without spaces and all lowercase letters.
    - `component_name`: Derived from project without spaces and `-`, it will be the default component class name and as such should be PascalCase for naming.
    - `author_name`/`author_email`: Your name/email to be included in `package.json` and `setup.py`.
    - `description`: A short description for the project.
    - `license`: Choose a license from the list.
    - `publish_on_npm`: Set to false if you don't want to publish on npm, your component will always be loaded locally.
    - `install_dependencies`: Install the npm packages and `requirements.txt` and build the initial component so it's ready for a spin.

Project structure

```
- project_shortname           # Root of the project
  - project_shortname         # The python package, output folder for the bundles/classes.
  - src                       # The javascript source directory for the components.
      - lib
          - components        # Where to put the react component classes.
      - demo
          - App.js            # A sample react demo, only use for quick tests.
          - index.js          # A reactDOM entry point for the demo App.
      - index.js              # The index for the components exported by the bundle.
  - tests                     #
      - requirements.txt      # python requirements for testing.
      - test_usage.py         # Runs `usage.py` as a pytest integration test.
  - package.json              # npm package info and build commands.
  - setup.py                  # Python package info
  - requirements.txt          # Python requirements for building the components and running usa
  - usage.py                  # Sample Python dash app to run the custom component.
  - webpack.config.js         # The webpack configs used to generate the bundles.
  - webpack.serve.config.js   # webpack configs to run the demo.
  - MANIFEST.in               # Contains a list of files to include in the Python package.
  - LICENSE                   # License info
```

Build the project

- `npm run build:js` generate the JavaScript bundle `project_shortname.min.js`
- `npm run build:backends` generate the Python, R, and Julia class files for the components.
- `npm run build` generate everything: the JavaScript bundles and the Python, R, and Julia class files.

Release the project

If you choose `publish_on_npm`, you will have to publish on npm first. Publishing your component on npm will rebuild the bundles, do not rebuild between that and publishing on pypi as the bundle will be different when serving locally and externally.

Publish on npm

`npm publish` If you have 2 factor enabled, you will need to enter the otp argument.

Publish on pypi

`python setup.py sdist` will build the python tarball package locally in the dist folder.

You can then upload the package using twine (`pip install twine`):

`twine upload dist/*`

Quick intro to React

Now, let's go ahead and see what the code for our new React application looks like. In your favorite code editor, open the `src/lib/components/ExampleComponent.react.js` file (supposing you named your project `example_component`). This is our first React component!

In React, user interfaces are made of "components," and by convention there will usually be one main component per file. This project imports the `ExampleComponent` in `src/demo/App.js`, the demo application.

The demo application `src/demo/App.js` is what you see after you run `npm run start`. To see how `src/demo/App.js` and `ExampleComponent.react.js` work, try adding `<h1>Hello, Dash!</h1>` inside the `render()` function of either of the files.

When you save the files, your browser should automatically refresh and you should see your change.

JSX

The `<h1>` and `<div>` tags you see look exactly like HTML tags, however, they are slightly different. These tags are what is called JSX - a syntax extension to JavaScript. JSX was developed by the React team to make easy, inline, HTML-like markup in JavaScript components.

There are a few key differences between JSX tags and HTML tags:

- The `class` keyword is renamed `className` (similarly as in Dash)
- In HTML, we write inline styles with strings like `<h1 style="color: hotpink; font-size: 12px">Hello Dash</h1>`. In JSX (as in Dash), we use objects with camelCased properties: `<h1 style={"color": "hotpink", "fontSize": "12px"}>Hello Dash<h1>`
- In JSX, we can embed variables in our markup. To embed a variable in the markup, wrap the variable in `{}`. For example:

```
render() {
    var myText = 'Hello Dash!';
    return (
          <h1>{myText}</h1>
    );
}
```

- In addition to the HTML tags like `<h1>` and `<div>`, we can also reference other React classes. For example, in our `src/demo/App.js`, we render the `ExampleComponent` component by referencing it as `<ExampleComponent>`.

A quick primer on the JavaScript language

Variable declaration

In JavaScript, we have to declare our variables with `let` or `const`. `const` is used when the variable shouldn't change, `let` is used elsewhere:

```
const color = 'blue';
let someText = 'Hello World';
let myText;
myText = 'Hello Dash';
```

Comments

Single line comments are prefixed with `//`. Multi-line comments are wrapped in `/* */`

```
/*
* This is a multi-line comment
* By convention, we use a `*` on each line, but it's
* not strictly necessary.
*/
const color = 'blue'; // This is a single line comment
```

Strings

Strings are defined the same way in JavaScript: single or double quotes:

```
const someString = 'Hello Dash';
const anotherString = "Hello Dash";
```

Instead of Python's `format`, JavaScript allows you to embed variables directly into strings by wrapping the variable in `${}` and wrapping the string in backticks:

```
const name = 'Dash';
const someString = `Hello ${name}`;
```

Dictionaries

In Python, we use dictionaries for key-value pairs. In JavaScript, we use "objects" and they are instantiated and accessed very similarly:

```
const myObject = {"color": "blue", "size": 20};
myObject['color']; // is blue
myObject.color;  // another way to access the color variable
```

In Python, the keys of a dictionary can be any type. But in JavaScript, the keys can only be strings. JavaScript allows you to omit the quotes around the strings and we frequently do:

```
const myObject = {color: "blue"}; // notice how there are no strings around color
```

So if you want to set a dynamic key in an object, you have to wrap it in square brackets:

```
const styleProperty = "color";
const myObject = {[styleProperty]: "blue"};
myObject.color;
```

Lists

In JavaScript, lists are called "arrays" and they're instantiated and accessed the same way:

```
const myList = ["Hello", "Dash", "!"];
myList[0];  // Hello
myList[1];  // Dash
myList[myList.length - 1]; // -1 references aren't allowed in JavaScript
```

Semicolons

In JavaScript, the convention is to end each line in a semicolon. It's not strictly necessary anymore, but it's still the convention.

Print Statements, Errors, and the Console

In JavaScript, we use `console.log` to print statements into the "console":

```
console.log("Hello Dash");
```

Since JavaScript runs in the web browser, we won't see these statements in our terminal like we would in Python. Instead, we'll see these statements in the browser's "dev tools console".

To access the console:

1. Right click on the web page
2. Click "Inspect Element"
3. Click on the "Console" tab.

To see for yourself, add a `console.log` statement inside the `render` method of `ExampleComponent`, refresh the page, and then inspect your browser's console.

Like Python, error messages and exceptions will also appear inside this console.

If, For, While

`if`

```
if (color === 'red') {
  console.log("the color is red");
} else if (color === 'blue') {
  console.log("the color is blue");
} else {
  console.log("the color is something else");
}
```

`for`

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

`while`

```
let i = 0;
while (i < 10) {
  i += 1;
}
```

Functions

In JavaScript, you'll see functions defined in two ways:

The new style way:

```
const add = (a, b) => {
  // The inside of the function
  const c = a + b;
  return c;
}

console.log(add(4, 6)); // 10
```

The traditional way:

```
function add(a, b) {
  // The inside of the function
  const c = a + b;
  return c;
}

console.log(add(4, 6)); // 10
```

Classes

> Heads up! Classes, among other features, are new language features in JavaScript. Technically, they're part of a new version of JavaScript called ES6. When we build our JavaScript code, a tool called Babel will convert these new language features into simpler JavaScript that older browsers like IE11 can understand.

JavaScript classes are very similar to Python classes. For example, this Python class:

```
class MyComponent(Component):
    def __init__(self, a):
        super().__init__()
        self.a = a;

    def render(self):
        return self.a;
```

would be written in JavaScript as:

```
class MyComponent extends Component {
  init(a) {
      super();
      this.a = a;
  }

  render() {
      return this.a;
  }
}
```

Importing and Exporting

In Python, we can import any variable from any file. In JavaScript, we have to explicitly specify which variables we want to make "importable" by "exporting" the variables.

If we only want to export a single variable, we'll write `export default`:

`some_file.js`

```
const text = 'hello world';
export default text;
```

`another_file.js`

```
import text from './some_file.js';
```

If we want to export multiple variables, we'll just write `export`:

`some_file.js`

```
const text = 'hello world';
const color = 'blue';
const size = '12px';

export text;
export color;
```

`another_file.js`

```
import {text, color} from './some_file.js';
/*
* note that we can't import size
* because we didn't export it
*/
```

The Standard Library and Ramda

Unlike Python, JavaScript's "standard library" is pretty small. At Plotly, we use the 3rd party library `ramda` for many of our common data manipulations.

Virtual DOM

If we look at the `App` component again, we see that it is `export`ed at the bottom. If you open up the `src/demo/index.js` file, you can see that it's imported there so that it can be used in a call to `ReactDOM.render()`.

ReactDOM.render() is what actually renders our React code into HTML on the web page. This ReactDOM.render() method is only called here, and only called once.

Classes

We see here in our `App` component that it is defined as a `class` which `extends` from the `Component` class of React. This provides some methods to us, for example the `render()` method we're using here. `render()` is the method that is called by the *component that is rendering it*. In our case, `render()` is called by the `ReactDOM.render()` call in `index.js`.

Notice how the `<App />` is called in the `ReactDOM.render()` method: our `App` component is used as a JSX tag!

Other Methods on React.Component

Other methods provided by React are mostly related to component state management. Lifecycle hooks like `shouldComponentUpdate` and `componentDidMount` allow you to better specify when and how a component should update.

For these methods, please refer to the **React/State and lifecycle**.

Our very own React component

Creating a boilerplate component

Now, let's create our very own component. Create a file named `TextInput.react.js` inside the `src/lib/components/` folder. In `TextInput.react.js` write:

```
import React, { Component } from 'react';

class TextInput extends Component {
  // here we'll define everything we need our TextInput component to have
}
```

Next, we'll write a `constructor` method on our component. A class constructor in Python is usually defined as `def __init__()` on a class, but in JavaScript we use the `constructor()` syntax.

In the constructor, we call the `super()` method on our component's props (more on props later), and set some `state`. It will look like this:

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
}
```

`props` are a component's properties. They are passed down from a component's parent, and are available as the `props` attribute. Calling `super()` on `props` in the constructor makes our props available in the component as `this.props`. The `this` keyword in JavaScript is Python's `self`.

We'll show you how to pass down `props` a bit later on.

Defining the `render` method

Next, let's define our `render()` method for our new component. In React, we are declaring UI components, and React calls the `render()` method when it wants to render those components.

A component's `render` method can return a basic string, for example `return "Hello, World!"`. When this component is used somewhere, it's `render()` method is called and "Hello, World!" will be displayed on the page.

Likewise, you can return a React element (specified using JSX) and React will render that element.

Exporting and importing a component

We'll also go ahead and `export` our component as the `default`. This means whenever we're trying to `import` something from this file, and we don't specify a name, we'll get the `default` export.

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
  render() {
    return <input />
  }
}

export default TextInput;
```

Now let's `import` that component and use it in our `App` component!

Add the `import TextInput from './TextInput';` line to the top of `App.js`, and somewhere in the return of our `render()` method, use our newly created `<TextInput />` component.

Tada!

We've got a text input.

In order to use our component from Python, you also need to import and export the component in `src/lib/index.js` (see how it is done for ExampleComponent).

Updating state with the `setState()` method

However, this input doesn't really do much - it's not connected to anything, nor does it save what you type in. Let's change our `render()` method of `TextInput` to set the HTML `value` attribute on our `<input />` tag, so it looks like this: `<input value='dash' />`. Save it, and we should now see that the value of our `<input>` tag is set to 'dash'!

We can also change our value to be that which is defined in our `state` object, so `<input value={this.state.value} />`. The `{}` syntax in JSX means that we want to write inline JavaScript in our JSX, so our `this.state.value` statement can be computed.

Great! Now our input says 'default'! Unfortunately, our input is still not very useful, because we can't change our input's value, try as we might.

It may seem odd to you that we can't type anything into the `<input/>` box. However, this is consistent with the React model: in our render method, we are telling React to render an input with a particular value set. React will faithfully render this input with that value no matter what, even if we try typing in it.

In order to have the input update when we type, we have to make sure that the `value` variable is updated with whatever we're typing in the input. To do that, we can listen for changes to the input and update our state accordingly:

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
  handleInputChange = (e) => {
    // get the value from the DOM node
    const newValue = e.target.value;
    // update the state!
    this.setState({
      value: newValue
    })
  }
  render() {
    return <input value={this.state.value} onChange={this.handleInputChange}/>
  }
}

export default TextInput;
```

Here, we wrote a method which we set on our input's `onChange` attribute, which will fire every time we type into the input. This method has a parameter (named `e` for event) on which certain attributes are set: `target.value` is what we need. This is how the HTML DOM works - for more information check out **these docs**.

Next, we use a method called `setState()` that's provided by `React.Component`. This method will handle updates to our `state` object. This method is really special. It'll do two things:

1. It'll merge the object that you provide with whatever was currently in `this.state`.
2. Then, it'll rerender the component. That is, it'll tell React to call the components render method again with the new data set in `this.state`.

See how this now allows you to type in our input component? We can also display our state by writing our `render()` method something like:

```
render() {
  return <div>
    <input value={this.state.value} onChange={this.handleInputChange} />
    <p>{this.state.value}</p> // here we're displaying our state
  </div>
}
```

Notice that we're not allowed to return multiple elements from `render()`, but an element with children is totally fine.

Component props

We can also pass along properties to our components, via the aforementioned `props`. This works the same as assigning attributes on a component, as we'll demonstrate by adding a `label` prop to our `TextInput` component!

Let's edit our call to `<TextInput />` in `App.js` to say `<TextInput label='dash-input' />`. This means we now have a `prop` called `label` available on our `TextInput` component. In `TextInput`, we can reference this via `this.props`.

Let's extend our `render()` method further so it renders our `label` prop:

```
render() {
  return <div>
    <label>{this.props.label}</label>
    <input value={this.state.value} onChange={this.handleInputChange} />
    <p>{this.state.value}</p>
  </div>
}
```

Props always flow down, but you can set a method as a prop too, so that a child can call a method of a parent. For more information, please refer to the **React/Components and props**.

These are just the basics of React, if you want to know more, the **official documentation site** is a great place to start!

Using your React components in Dash

We can use most, if not all, React components in Dash! Dash uses React under the hood, specifically in the `dash-renderer`. The `dash-renderer` is basically just a React app that renders the layout defined in your Dash app as `app.layout`. It is also responsible for assigning the callbacks you write in Dash to the proper components, and keeping everything up-to-date.

Control the state in the parent

Let's modify our previous example to control `state` in the parent `App` component instead of in the `TextInput` component. Let's start by moving the `value` in `state` up to the parent component.

In `src/demo/App.js`, add state and pass the value into the component:

```
class App extends Component {
  constructor() {
    super(props)

    this.state = {
        value: 'dash'
    };
  }

  render() {
    return <TextInput label={'Dash'} value={this.state.value}/>
  }
}
```

In `src/lib/components/TextInput.react.js`, use the `value` prop instead of the state:

```
class TextInput extends Component {
  constructor() {
    super(props)
  }

  render() {
    return (
```

```
          <div>
            <label>{this.props.label}</label>
            <input value={this.props.value}/>
            <p>{this.props.value}</p>
          </div>
        )
    }
}
```

Now, as before, the `<input/>` won't actually update when you type into it. We need to update the component's `value` property as we type. To do this, we'll define a function in our parent component that will update the parent's component state, and we'll pass that function down into our component. We'll call this function `setProps`:

```
class App extends Component {
  constructor() {
    super(props)

    this.state = {
        value: 'dash'
    };
  }

  setProps(newProps) {
    this.setState(newProps);
  }

  render() {
    return (
      <TextInput
        label={'Dash'}
        value={this.state.value}
        setProps={this.setProps}
      />
    )
  }
}
```

and in `TextInput`, we'll call this function when the `value` of our `input` changes. That is, when we type into the input:

```
class TextInput extends Component {
  constructor() {
    super(props)

  }

  handleInputChange = (e) => {
    const newValue = e.target.value;
    this.props.setProps({value: newValue});
  }


  render() {
    return (
        <div>
          <label>{this.props.label}</label>
          <input value={this.props.value} onChange={this.handleInputChange}/>
          <p>{this.props.value}</p>
        </div>
      )
  }
}
```

To review, this is what happens when we type into our `<input>`:

1. The `handleInputChange` is called with whatever value we typed into the `<input/>`
2. `this.props.setProps` is called, which in turn calls the `setState` property of the `App` component.
3. `this.setState` in `App` is called. This updates the `this.state` of `App` *and* implicitly calls the `render` method of `App`.

4. When `App.render` is called, it calls `TextInput.render` with the new properties, rerendering the `<input/>` with the new `value`.

In Dash apps, the `dash-renderer` project is very similar to `App.js`. It contains all of the "state" of the application and it passes those properties into the individual components. When a component's properties change through user interaction (e.g. typing into an `<input/>` or hovering on a graph), the component needs to call `setProps` with the new values of the property. Dash's frontend (`dash-renderer`) will then rerender the component with the new property *and* make the necessary API calls to Dash's Python server callbacks.

Handling the case when `setProps` isn't defined

Note: This section is present for legacy purposes. As of v0.40.0, setProps is always defined.

In Dash, `setProps` *is only defined if the particular component is referenced in an `@callback`.* If the component isn't referenced in a callback, then Dash's frontend will not pass in the `setProps` property and it will be undefined.

> As an aside, why does Dash do that? In some cases, it could be computationally expensive to determine the new properties. In these cases, Dash allows component authors to skip doing these computations if the Dash app author doesn't actually need the properties. That is, if the component isn't in any `@callback`, then it doesn't need to go through the "effort" to compute its new properties and inform Dash.

In most cases, this is a non-issue. After all, why would you render an `Input` on the page if you didn't want to use it as an `@callback`? However, sometimes we still want to be able to interact with the component, even if it isn't connected to Dash's backend. In this case, we'll manage our state locally *or* through the parent. That is:

1. If `setProps` is defined, then the component will call this function when its properties change and Dash will faithfully rerender the component with the new properties that it passed up.
2. If `setProps` isn't defined, then the component isn't "connected" to Dash's backend through a callback and it will manage its state locally.

Here's an example with our `TextInput` component:

```
class TextInput extends Component {
  constructor() {
    super(props)
    this.state = props;
  }

  handleInputChange = (e) => {
    const newValue = e.target.value;
    this.props.setProps({value: newValue});
  }

  render() {
    let value;
    if (this.props.setProps) {
      value = this.props.value;
    } else {
      value = this.state.value;
    }

    return (
        <div>
          <label>{this.props.label}</label>
          <input value={value} onChange={this.handleInputChange}/>
          <p>{value}</p>
        </div>
      )
  }
}
```

Annotate your function with `propTypes`

The final step in authoring your Dash component is to describe which properties are available.

This is done through React's `propTypes`. At the end of your file, write:

```
TextInput.propTypes = {
  ...
}
```

(You should fill that in with the actual ones.)

You must include `propTypes` because they describe the input properties of the component, their types, and whether or not they are required. Dash's React-to-Python toolchain looks for these `propTypes` in order to automatically generate the Dash component Python classes.

A few notes:

- The comments above each property are translated directly into the Python component's docstrings. For example, compare the output of `>>> help(dcc.Dropdown)` with **that component's propTypes**.
- The `id` property is required in all Dash components.
- The list of available types are available **here**.
- In the future, we will use these `propTypes` to provide validation in Python. That is, if you specify that a property is a `PropTypes.string`, then Dash's Python code will throw an error if you supply something else. Track our progress in this issue: **264**.

React as a peer dependency

React and reactDOM are included as peer dependencies, your editor may warn you that react is not installed. You can safely ignore those warnings as they are served by the `dash-renderer` and they don't need to be bundled with your components.

Build your component in Python

Now that you have your React component, you can build it and import it into your Dash program. View instructions on how to build the component in **the boilerplate repo**.

In this tutorial, we rebuilt the `ExampleComponent` that was provided in **the boilerplate**. So, the Python component code in `usage.py` should look familiar - the properties and behaviour of `ExampleComponent` are exactly the same as our `TextInput`.

Dash Components with TypeScript and React

> Dash 2.4 and later supports components created with React and TypeScript.

A **cookiecutter boilerplate for TypeScript components** is available to make it easy to get started with building a TypeScript/React Dash component.

Generating the Boilerplate

To generate a new Dash TypeScript component boilerplate:

1. Create a Python virtual environment and activate it.
2. Install cookiecutter with `pip install cookiecutter`
3. Run the Dash TypeScript component template: `cookiecutter https://github.com/plotly/dash-typescript-component-template.git`
4. Answer the questions about the component to create it.
    - **project_name:** The "human-readable" name of your project. For example, "Dash Core Components".
    - **project_shortname**: Derived from the project name, it is the name of the "Python library" for your project.
    - **component_name**: This is the name of the initial component that is generated.

- **jl_prefix**: Optional prefix for Julia components. For example, dash_core_components uses "dcc" so the Python dcc.Input becomes dccInput in Julia, and dash_table uses "dash" to make dashDataTable.
- **author_name**: For package.json and DESCRIPTION (for R) metadata.
- **author_email**: For package.json and DESCRIPTION (for R) metadata.
- **github_org**: If you plan to push this to GitHub, enter the organization or username that will own it.
- **description**: The project description, included in package.json.
- **publish_on_npm**: Set to false to only serve locally from the package data.

Writing Your Component

Go to your created component directory and you'll see a structure like this:

```
├── LICENSE
├── MANIFEST.in
├── README.md
├── __init__.py
├── <project-shortname>
│   ├── __init__.py
│   └── _imports_.py
├── justfile
├── package.json
├── requirements.txt
├── setup.py
├── src
│   └── ts
│       ├── components
│       │   └── <component-name>.tsx
│       ├── index.ts
│       └── props.ts
├── tsconfig.json
├── usage.py
└── webpack.config.js
```

Installing Requirements and Building Your Component

Install the Python requirements with `pip install -r requirements.txt` and run `npm install` and `npm run build` to build your initial component.

Adding Props

The boilerplate creates a `props.ts` file in `src/ts`. In this file you'll find the default props that Dash gives to all components, where they are exported. In order to use types defined in other files, the types must be imported. For example, to use the DashComponentProps of `src/ts/props.ts`, import the type and add it to the props of the component:

```
import {DashComponentProps} from '../props';

type MyProps = {
    my_value: string;
} & DashComponentProps;  // Add the props after the new one.
```

> Note: We recommend using type instead of interface for prop ordering.

In the following example, we define props for an Audio component using **TypeScript builtin utility types** and React HTML Element types, to keep only the props that are supported, and add more custom props. We define these props in `props.ts`, but you can define props in any file as long as you import them in the component file.

`./src/props.ts`

```
import React from 'react'

export type DashComponentProps = {
    /**
     * Unique ID to identify this component in Dash callbacks.
     */
    id?: string;
    /**
     * Update props to trigger callbacks.
     */
    setProps: (props: Record<string, any>) => void;
}

type InvalidHtmlProps =
    | 'dangerouslySetInnerHTML'
    | 'defaultChecked'
    | 'defaultValue'
    | 'suppressContentEditableWarning'
    | 'suppressHydrationWarning'
    | 'ref'
    | 'key'
    | 'async';

type EventsProps = keyof Omit<React.DOMAttributes<any>, 'children'>;
type AriaProps = keyof React.AriaAttributes;
export type HtmlOmittedProps = InvalidHtmlProps | EventsProps | AriaProps;

type SourceProps = Omit<React.SourceHTMLAttributes<any>, HtmlOmittedProps>;

export type AudioProps = Omit<React.AudioHTMLAttributes<any>, HtmlOmittedProps> & Partial<{
    sources: JSX.Element[] | SourceProps[];
    /**
     * Set to true to play the audio source.
     */
    play: boolean;
    status: 'playing' | 'paused' | 'stopped';
    /**
     * READONLY: true when the audio can be played.
     */
    can_play: boolean;
    /**
     * READONLY: true when the audio has been download completely.
     */
    can_play_through: boolean;
}> & DashComponentProps;
```

`./src/components/Audio.tsx`

```
import React from 'react'
import { AudioProps } from '../props';


const Audio = (props: AudioProps) => {
    // Remove the custom props & keep the rest to give to the html component.
    const { sources, play, status, can_play, can_play_through, setProps, ...audioProps } = prop
    return (
        <audio {...audioProps}>
            // ...
        </audio>
    )
}

export default Audio;
```

Functional vs class component props

The example `./src/components/Audio.tsx` file above shows how to set props for a functional component. For a class component, this will be different:

For a class component without state:

```
export default class MyComponent extends React.PureComponent<Props>
```

For a class component with state:

```
export default class MyComponent extends React.Component<Props, State>
```

tsconfig.json

The root of the project contains a configurable `tsconfig.json` file. For more details on changing from the defaults see **What is a tsconfig.json**.

Adding TypeScript Support to an Existing Component Library

To add TypeScript support to an existing component library:

1. Install TypeScript dependencies with `npm i -D typescript ts-loader`
2. Add a **tsconfig.json** file. The following `tsconfig.json` will work with most configurations. You'll just need to update the `baseUrl`.

```
{
"compilerOptions": {
 "jsx": "react",
 "baseUrl": <your-base-directory>
 "inlineSources": true,
 "sourceMap": true,
 "esModuleInterop": true
},
"exclude": [
 "node_modules"
]
}
```

3. In `webpack.config.js`, add TypeScript resolve extensions and add `ts-loader` in module rules like this:

```
resolve: {
    extensions: ['.ts', '.tsx', '.js'],
},
module: {
    rules: [
        {
            test: /\.tsx?$/,
            use: 'ts-loader',
            exclude: /node_modules/,
        },
        {
            test: /\.jsx?$/,
            exclude: /node_modules/,
            use: 'babel-loader'
        }
    ]
}
```

Here are some helpful links:

- **The React docs**
- **dash-core-components in the Dash GitHub repo**
- **dash-component-boilerplate**

Dash Python > ***React for Python Developers***