AnnMarieW Update part2_architecture_overview.md 583cdf4 · 2 years ago

Part 2 Architecture Overview

Dash Component Boilerplate

All the component libraries in Dash, such as `dash-core-components`, `dash-html-components` and `dash-daq` use the same toolkit and it's made available to you in the Dash Component Boilerplate library.

In this section, we'll create a project following the instructions in the README in the `dash-component-boilerplate` library. We'll give an overview of the file structure created and show a workflow you can use to develop new components. Note that the component boilerplate is updated periodically, so use this tutorial as a guideline only and be sure to check Github for the latest information.

We start by installing the `cookiecutter` which is a library for initializing projects, and then use the command to get the template from Github to create the project.

```
$ pip install cookiecutter
$ cookiecutter gh:plotly/dash-component-boilerplate
```

> Note for Windows users: Install the cookiecutter and this project in a directory that has no spaces in the pathname. For example this is not valid: `C:\Users\my name\...`

You will be prompted to enter some values, then it will create the package in the working directory based on those values. See the dash-component-boilerplate README for a description of each question. For this demo, the `project_name` is set to Custom Components, all other options contain default values with the exception of the `author_name` and `author_email`; excluding these values will result in an error. The `component_name` is set to `MyComponent` to make it more clear in the following sections -- you can leave it as the default if you'd like.

```
> project_name [my dash component]: Custom Components
> project_shortname [custom_components]:
> component_name [CustomComponents]: MyComponent
> jl_prefix []:
> r_prefix []:
> author_name [Enter your first and last name (For package.json)]: mark
> author_email [Enter your email (For package.json)]: mark@mbele.io
> github_org []:
> description [Project Description]:
>Select use_async:
1 - False
2 - True
Choose from 1, 2 [1]:
> Select license:
1 - MIT License
```

```
2 - BSD License
3 - ISC License
4 - Apache Software License 2.0
5 - GNU General Public License v3
6 - Not open source
Choose from 1, 2, 3, 4, 5, 6 [1]: 1
> publish_on_npm [True]: False
> install_dependencies [True]:
```

You will see a message:

```
use_async
False
use_async is set to False, your component will not be lazy loaded and fragments will not be
created.
```

This message is expected if you use the default setting for `use_async`, which is False. If you select `True`, then it will create a project structure that splits the code into smaller pieces and allows for "lazy loading". This can improve app performance by loading just the things that are currently needed by the user. This can be added later if necessary, but for now it's not needed. See section x for more information.

The cookiecutter process installs the Python and JavaScript dependencies (which can take a while), and creates a folder called `custom_components` which corresponds to the `project_name`. Within the `custom_components` folder there are several files and folders that are created. However, we'll only highlight the folders that you need to focus on to create custom dash components:

```
./custom_components/custom_components
./custom_components/package.json
./custom_components/setup.py
./custom_components/src
./custom_components/tests
./custom_components/venv
..... (other files and folders)
```

Virtual Environment

The first folder of interest is `venv/`. This contains a Python virtual environment that was set up by cookiecutter. To proceed with building a component, we activate this environment:

```
$ cd custom_componets
$ source venv/bin/activate
```

or on some Linux and Mac environments use "." instead of "source"

```
$ . venv/bin/activate
```

On Windows:

```
venv\Scripts\activate
```

It is important to note that this is different from the virtual environment that was originally used to create the dash project and install the `cookiecutter` package. The purpose of the `custom_components` virtual environment is for developing the custom components.

JavaScript files

Now let's go to the `.custom_comonents/src/lib/components/` folder. This folder will contain the JavaScript files for the custom components. Currently it contains the file for the example component created with the `cookiecutter -- MyComponent.react.js`. This is the name we provided to the `component_name` prompt when we started the `cookiecutter` process.

Let's take a closer look at `MyComponent.react.js`:

```
import React, {Component} from 'react';
import PropTypes from 'prop-types';

/**
 * ExampleComponent is an example component.
 * It takes a property, `label`, and
 * displays it.
 * It renders an input with the property `value`
 * which is editable by the user.
 */
export default class MyComponent extends Component {
    render() {
        const {id, label, setProps, value} = this.props;

        return (
            <div id={id}>
                ExampleComponent: {label} 
                <input
                    value={value}
                    onChange={
                        /*
                         * Send the new value to the parent component.
                         * setProps is a prop that is automatically supplied
                         * by dash's front-end ("dash-renderer").
                         * In a Dash app, this will update the component's
                         * props and send the data back to the Python Dash
                         * app server if a callback uses the modified prop as
                         * Input or State.
                         */
                        e => setProps({ value: e.target.value })
                    }
                />
            </div>
        );
    }
}

MyComponent.defaultProps = {};

MyComponent.propTypes = {
    /**
     * The ID used to identify this component in Dash callbacks.
     */
    id: PropTypes.string,

    /**
     * A label that will be printed when this component is rendered.
     */
    label: PropTypes.string.isRequired,

    /**
     * The value displayed in the input.
     */
    value: PropTypes.string,

    /**
     * Dash-assigned callback that should be called to report property changes
```

```
     * to Dash, to make them available for callbacks.
     */
    setProps: PropTypes.func
};
```

`PropTypes` and `Docstrings`

First, take note of the comments near the top of the file and in the PropTypes. The Dash toolchain uses these JavaScript comments to automatically create the docstring in the Python files.

You may have noticed that the Dash documentation uses these docstrings too – the reference section for the components are automatically generated from them.

To see an example, let's look at our `MyComponent` docstring.

In a Python console:

```
> import custom_components
> help(custom_components.MyComponent)
```

```
class MyComponent(dash.development.base_component.Component)
|  MyComponent(id=undefined, label=required, value=undefined, **kwargs)
|
|  A MyComponent component.
|  ExampleComponent is an example component.
|  It takes a property, `label`, and
|  displays it.
|  It renders an input with the property `value`
|  which is editable by the user.
|
|  Keyword arguments:
|
|  - id (string; optional):
|      The ID used to identify this component in Dash callbacks.
|
|  - label (string; required):
|      A label that will be printed when this component is rendered.
|
|  - value (string; optional):
|      The value displayed in the input.
```

Compare this Python docstring with the `MyCompont.react.js` file and you will see that the descriptions match the JavaScript comments. Also, the keyword arguments are the props from the component's PropTypes.

The PropTypes also provide validation to ensure that the correct prop names and data types are used in the Dash app. For example, you will see helpful error messages if you try to use a prop name that is not defined or is the wrong type (eg : `options[0].label is not a string`). Here you can also set default values and indicate which props are required. Note that the `label` prop is required for `MyComponent`

While the PropTypes aren't strictly necessary in React, they are required to for the Dash toolchain to work. (Learn more about PropTypes here)

`setProps`

Usually in a React app you would use the `setState` or (`useState` for functional components) to manage the state of the component. However, how does Dash get access to the internal state? This is where `setProps` is used, and as the comment states:

```
    /**
     * Dash-assigned callback that should be called to report property changes
     * to Dash, to make them available for callbacks.
     */
```

To learn more about how updates are traditionally made with React, see the Lifting State Up chapter in the React tutorial. Once you understand this section, then you'll have a better idea of how `setProps` works: it's a function that's used to "lift state up" to the Dash app front-end, enabling updates to get passed down to callbacks, and if necessary, rerendering the component.

In the `MyComponent` example, `setProps` is used to update the input value when it changes:

```
                    onChange={
                        e => setProps({ value: e.target.value })
                    }
```

index.js

When the project was initialized, our component was automatically added to the `index.js` file in `.custom_comonents/src/lib/` folder.

```
import MyComponent from './components/MyComponent.react';

export {
    MyComponent
};
```

If you change the name of `MyComponent`, you will need to update it here. Also, if you add a new component to this project, it needs to be added to `index.js` as well.

Python files

You might be wondering how the JavaScript files are transformed into Python packages to be imported into your Dash application. These files are automatically generated by the Dash toolchain during the build process. You can find the Python file in the `.custom_components/custom_components/` folder. You will find the file `MyComponent.py` which corresponds to `MyComponent.react.js`. On viewing the python file, you will see that the `__init__` method maps to the `PropTypes` defined in the React counterpart.

Within the folder, there will also be a file named `custom_components.min.js`. This file contains the transpiled and minified javascript code for the dash custom components as well as all its dependencies.

Using the Component in a Dash app

Once the custom component's Python and JavaScript files have been created, the next question is; How do you use this in a Dash application? Conveniently, the cookiecutter created a sample Dash app in the root directory called `usage.py`:

```
import custom_components
import dash
from dash.dependencies import Input, Output
import dash_html_components as html

app = dash.Dash(__name__)

app.layout = html.Div([
    custom_components.MyComponent(
```

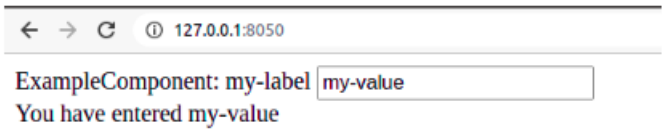
```
        id='input',
        value='my-value',
        label='my-label'
    ),
    html.Div(id='output')
])


@app.callback(Output('output', 'children'), [Input('input', 'value')])
def display_output(value):
    return 'You have entered {}'.format(value)


if __name__ == '__main__':
    app.run_server(debug=True)
```

Notice that it imports our new library with `import custom_components` and adds `custom_components.MyComponent(…)` to the layout.

Before making any changes to any of the files, run this `usage.py` file. Open the browser as usual with any Dash app, and if you see the following you will know that everything is set up correctly and you have created your first custom Dash component.



Tests

The `cookiecutter` created a sample integration test file `.custom_components/tests/.test_usage.py` . To run the test:

```
$ pip install -r tests/requirements.txt
$ pytest tests
```

To learn more about testing in Dash see the dash documentation https://dash.plotly.com/testing

README.md

The cookiecutter also generated a `README.md` file for the project that gives step-by-step instructions on how to use the sample files that were created. You will also see an explanation of how to create a production build and publish. More on these topics later in this tutorial.

Component Development Workflow

The great thing about creating Dash components is that you only need to write the component in React and then the Dash toolchain will create all the other files for you automatically.

This is done by running a script with `npm` .

```
$ npm run build
```

To see what commands are run, look for the `build` script in the `package.json` file. This `build` script will compile the React files and create the `custom_components.min.js` file that will be served by the dash-renderer, plus it will create the Python, Julia and R files as well.

To summarize, a basic workflow for component development is:

- Modify or add new components written in React to the `src/lib/components/` folder.
- Add new components to the `index.js` file in `src/lib/` folder.
- Compile and rebuild the files with `$ npm run build`
- Use the component in Dash -- for example by modifying/running the `usage.py` sample Dash app.

Try this process by making some small changes to `MyComponent.react.js`, such as changing the text displayed with the `label`, or using different `jsx` tags. After rebuilding, when you run `usage.py` you should see your changes in the browser.

Now that you understand the structure of a Dash component project, when you go to component libraries on Github, such as `dash-core-components` it should look familiar to you. You can use these libraries to see more examples of how to make components and how to create tests.

Up Next Part 3 Conventions and Common Patterns