FRC Web Components

Release 0.1

Amory Galili

Aug 28, 2022

CONTENTS

Welcome to FRC Web Components!

CHAPTER

ONE

CONTENTS

1.1 Plugins

Plugins are used to extend the functionality of the FRC Web Components dashboard. The main components that can be extended are:

Elements
: These are UI components like buttons, line charts and robot visualizations that are used to build dashboards. Users have the option of adding their own custom elements in addition to the ones that ship with the dashboard. Read more about elements *here*.

Source Providers
: Source providers are what provide data to your elements, such as NetworkTables and Camera Streams. You can create your own Source providers to provide additional sources of data to your elements. Read more about Source providers *here*.

1.1.1 Elements

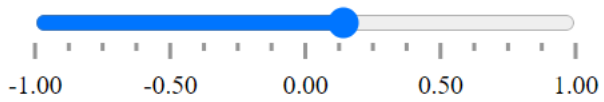
Stuff about components

1.1.2 Source Providers

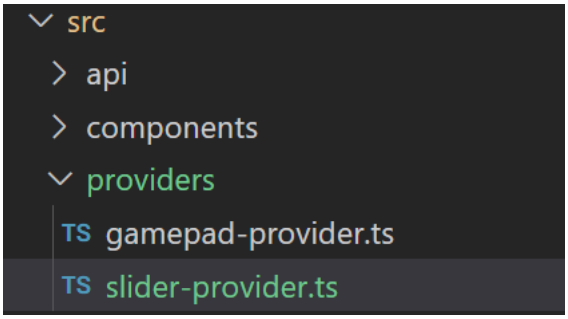
Source Providers are used to provide data to your elements. For example the NetworkTables source provider can be used to control dashboard elements through NetworkTables:

Above the FRC Accelerometer element's source provider is "NetworkTables" with the key "/accel". This will map entries in the "/accel" subtable to the accelerometer element's properties.

Let's create a source provider which can be used to control a number slider:



First, create a file called *slider-provider.ts* in the providers folder:



Paste the following code into *slider-provider.ts*:

```
import { SourceProvider } from '@webbitjs/store';

export default class SliderProvider extends SourceProvider {
   constructor() {
      super();
      this.updateSource('/slider/value', 3);
      this.updateSource('/slider/min', 0);
      this.updateSource('/slider/max', 10);
   }
}
```

Now add the following lines to *index.ts*:

```
// This is an import that goes at the top of the file
import SliderProvider from './providers/slider-provider';
```

```
// This is how the plugin adds the source provider to the dashboard
dashboard.addSourceProvider('SliderProvider', new SliderProvider());
```

Your *index.ts* file should now look something like this:

```
4   import { FrcDashboard } from '@frc-web-components/dashboard';
5   import GamepadProvider from './providers/gamepad-provider';
6   import SliderProvider from './providers/slider-provider';
7
8   export default function addPlugin(dashboard: FrcDashboard) {
9     dashboard.addSourceProvider('SliderProvider', new SliderProvider());
10    dashboard.addSourceProvider('Gamepad', new GamepadProvider());
```

You can now use the provider on your dashboard's elements:

Amazing! This would be a lot more interesting if the data came from an external source, however. Let's update SliderProvider so that the sources can be updated in the browser's URL:

```
import { SourceProvider } from '@webbitjs/store';

function getHash(): Record<string, string> {
   const keyValuePairs: Record<string, string> = {};
   const hash = new URL(document.URL).hash.substring(1);
```

(continues on next page)

(continued from previous page)

```
   hash.split('&').forEach(keyValue => {
      const [key, value] = keyValue.split('=');
      keyValuePairs[key] = value;
   });
   return keyValuePairs;
}

export default class SliderProvider extends SourceProvider {
   constructor() {
      super();
      this.updateSliderValues();
      window.addEventListener('hashchange', () => {
         this.updateSliderValues();
      });
   }

   private updateSliderValues() {
      const hash = getHash();
      ['value', 'min', 'max'].forEach(key => {
         const value = parseFloat(hash[key]);
         if (!isNaN(value)) {
         this.updateSource(`/slider/${key}`, value);
         }
      });
   }
}
```

Now the slider can be controlled through the browser's URL:

It would be great if moving the slider also updated the URL as well. Let's update SliderProvider to make it do that by adding the following code to *slider-provider.ts*:

```
function updateHash(key: string, value: string) {
   const newHash = {
      ...getHash(),
      [key]: value
   };
   const hashString = Object.entries(newHash)
      .map(([key, value]) => `${key}=${value}`)
      .join('&');
   document.location.hash = `#${hashString}`;
}
```

Next we will also be overriding the source provider's *userUpdate* method. The *userUpdate* method is called automatically when a change to an element's properties is detected. The *key* is the source key currently bound to the element's property. The *userUpdate* method has a default implementation which simply calls *updateSource* with the key value pair passed into *userUpdate*:

```
userUpdate(key: string, value: unknown) {
   this.updateSource(key, value);
}
```

Instead of setting the source directly, we will instead be updating the URL hash string by calling the *updateHash* function:

```
 userUpdate(key: string, value: unknown) {
    const numberValue = parseFloat(value as any);
    const property = ['value', 'min', 'max'].find(prop => key.endsWith(prop));
    if (!isNaN(numberValue) && property) {
       updateHash(property, numberValue.toString());
    }
}
```

The *slider-provider.ts* file should now have the following code:

```
import { SourceProvider } from '@webbitjs/store';

function getHash(): Record<string, string> {
   const keyValuePairs: Record<string, string> = {};
   const hash = new URL(document.URL).hash.substring(1);
   hash.split('&').forEach(keyValue => {
      const [key, value] = keyValue.split('=');
      keyValuePairs[key] = value;
   });
   return keyValuePairs;
}

function updateHash(key: string, value: string) {
   const newHash = {
      ...getHash(),
      [key]: value
   };
   const hashString = Object.entries(newHash)
      .map(([key, value]) => `${key}=${value}`)
      .join('&');
   document.location.hash = `#${hashString}`;
}

export default class SliderProvider extends SourceProvider {
   constructor() {
      super();
      this.updateSliderValues();
      window.addEventListener('hashchange', () => {
         this.updateSliderValues();
      });
   }

   userUpdate(key: string, value: unknown) {
      const numberValue = parseFloat(value as any);
      const property = ['value', 'min', 'max'].find(prop => key.endsWith(prop));
      if (!isNaN(numberValue) && property) {
         updateHash(property, numberValue.toString());
      }
   }

   private updateSliderValues() {
```

(continues on next page)

(continued from previous page)

```
      const hash = getHash();
      ['value', 'min', 'max'].forEach(key => {
         const value = parseFloat(hash[key]);
         if (!isNaN(value)) {
            this.updateSource(`/slider/${key}`, value);
         }
      });
   }
}
```

The URL should now be updated when you move the slider:

1.2 API