Catavolt React SDK User’s Guide

Scope of this document

This guide covers the Catavolt React 'Core' components. These are base-level components that provide declarative access to many of the Catavolt Javascript SDK functions, making the construction of custom Catavolt Apps much easier. These components provide no styling and allow the client to fully control the document structure (i.e. html and css)

The Catavolt Extended components (catreact-html) are not covered here. However, they are largely built upon these core components and add primarily, HTML renderers.

Renderers and Callback Objects

Renderers

The Cv components make heavy use of delegate 'renderer' functions, to allow for custom rendering. Almost every component supports a 'renderer' function as a parameter (some support specialized renderers with additional arguments, i.e. 'queryRenderer') that allows for complete control of what's rendered. Generally renderers are passed the component's relevant SDK object, as the **'scopeObj'** property of the **'scopeCtx'** object.

```
<CvWorkbench workbenchId={workbenchId} appWinDef={appWinDef} renderer={(cvContext:CvContext)=>{
       Const workbench:Workbench = cvContext.scopeCtx.scopeObj;
       //return the rendered workbench
       return <div>`A simple workbench named ${workbench.name}`</div>
}}/>



<CvListPane {...listPaneProps} queryRenderer={(cvContext:CvContext, callback:CvQueryPaneCallback)=>{
   const listContext:ListContext = cvContext.scopeCtx.scopeObj;
   //return the rendered list
   return ...
}}/>
```

Most renderers expose a **CvContext** object that allows access to the object in scope (**scopeCtx**), the catavolt **AppContext** object, and the **CvEventRegistry** object.

```
export interface CvContext {


   /**
    *  The Catavolt SDK entry point, an instance of the sdk {AppContext} which is always available to
    *  components.
    *  A singleton instance is used by the underlying components to interact with the sdk.
    */
   catavolt?:AppContext;

   /**
    *  The {CvEventRegistry} handles decoupled communication between components.  Any component may
    *  subscribe to and publish {CvEvent}s.  See also {@link CvListener} and {@link CvEventType}
    */
   eventRegistry?:CvEventRegistry;

   /**
    * Allows for exposing a 'scope' object to child components
    * This is usually the corresponding SDK object
    */
   scopeCtx?:CvScopeContext;
}
```

Callback Objects

In addition to the CvContext object, components often provided callback objects as an additional parameter to renderer functions. The callback objects allow the client to perform additional operations relevant to the particular component. CvAction for example, allows the Action to be ‘fired’ or performed via the callback.

```
<CvAction {...actionProps}
   renderer={(cvContext:CvContext, callback?:CvActionCallback)=>{
         return <a onClick={callback.fireAction}>Fire Action!</a>
  }}/>
```

Events and Results

The framework makes frequent use of the CvEvent<T> object to publish and return the results of operations. **CvEvent** is usually parameterized with a type of 'Result' object representing a specific result and it's relevant fields. For example, **CvActions**'s navigation listeners are notified with **CvEvent**<CvNavigationResult> and action listeners are notified with **CvEvent**<CvActionFiredResult>.

```
<CvAction actionId={'#search'}
  navigationListeners={[(e:CvEvent<CvNavigationResult>)=>{
      this.showSearchPanel(e.eventObj);
   }]}
   actionListeners={[(e:CvEvent<CvActionFiredResult>)=>{
       const actionFiredResult = e.eventObj as CvActionFiredResult;
       if(actionFiredResult.type === CvActionFiredResultType.ACTION_STARTED) {
         this.showLoadingIndicator();
       } else if(actionFiredResult.type === CvActionFiredResultType.ACTION_COMPLETED) {
         this.hideLoadingIndicator();
       }
   }]}
   renderer={(cvContext:CvContext, callback?:CvActionCallback)=>{
         return <a onClick={callback.fireAction}>Open Search</a>
  }}/>
```

In addition to providing listeners to specific components, **CvEvent**s may be subscribed to globally, and handled in isolation if desired. The CvEventRegistry is a global event registry that allows for subscribers to register (and deregister) for specific event types by providing CvListener functions. See 'Error Handling' and 'Base Component' for more information on this mechanism.

ValueProviders and ValueListeners

CvValueProviders and CvValueListeners are an implementation of the pub/sub or observer/observable pattern, and are used throughout the component framework. They provide a generalized mechanism for 'connecting' components, allowing them share information when necessary. **CvValueProvider** and **CvValueListener** are both interfaces that define the mechanism. A **CvValueListener** can simply subscribe to a **CvValueProvider** to receive notification of some result (T). They are defined as:

```
export interface CvValueProvider<T> {
   value:T;
   subscribe(updateListener:CvValueListener<T>):void;
}

export interface CvValueListener<T> {
   (value:T):void;
}
```

CvValueAdapter

The CvValueAdapter is an implementation of a CvValueProvider that maintains a list of subscribers and handles notification.

```
export class CvValueAdapter<T> implements CvValueProvider<T> {

   private _value:T;
   private _subscriberArray:Array<CvValueListener<T>> = [];

   ...

   createValueListener():CvValueListener<T> {
       return ((value:T) => {
           this.value = value;
           this._subscriberArray.forEach((updateListener:CvValueListener<T>)=> {
               updateListener(value)
           });
       });
   }

   subscribe(updateListener:CvValueListener<T>):void {
       if (this._subscriberArray.indexOf(updateListener) < 0) {
           this._subscriberArray.push(updateListener);
       }
   }

}
```

CvValueAdapter is convenient when two or more components need to be ‘connected’ to share some information. The client can create an instance of **CvValueAdapter** which can be provided to another component as a **CvValueProvider**. That component can then subscribe (internally) to the **CvValueProvider** to receive data updates.   Upon creation of the **CvValueAdapter**, the client can also obtain a **CvValueListener** by using the ‘createValueListener’ method. The client can ‘ask’ the **CvValueAdapter** to notify all subscribers by providing the result value to the **CvValueListener**.

For example, the pattern is used to allow the **CvListPanel** to ‘notify’ the **CvDropdownMenu** when a list item is selected:

```
const selectionAdapter:CvValueAdapter<Array<string>> = new CvValueAdapter<Array<string>>();

const menu = <CvDropdownMenu {...menuProps} selectionProvider={selectionAdapter}/>

const panel = <CvListPanel {...listProps} selectionListener={selectionAdapter.createValueListener()}/>
```

Catvolt Pane

The CatavoltPane is the required root component in a Catavolt React application. It initializes the context hierarchy and the Catavolt Javascript SDK instance. It should always be present at the root of the rendering tree.

```
<CatavoltPane enableResourceCaching={true}>
   {this.props.children}
</CatavoltPane>
```

Logging In

CvLogin provides a callback for logging in via the user provided renderer function. Supplied 'loginListeners' will receive a **CvLoginResult** containing the session window ID that can be used with a **CvAppWindow** component.

```
<CvLogin loginListeners={[this.loginListener]}
  renderer={(cvContext:CvContext, callback:CvLoginCallback)=>{

   if(!callback.isLoggedIn()) {
      return(
      <form onSubmit={this.submitForm.bind(this, callback)}>
        <input id="userId" type="text" value={this.state.userId}
               onChange={this.setFieldValue.bind(this, 'userId')}/>
        <input id="password" type="password" value={this.state.password}
               onChange={this.setFieldValue.bind(this, 'password')}/>

      </form>
      );
   } else {
     return null;
   }
}}/>

handleSubmit: function (callback:CvLoginCallback, e:any) {
       e.preventDefault();
       callback.login(gatewayUrl, tenantId, clientType, this.state.userId, this.state.password);
}

loginListener: function(event:CvEvent<CvLoginResult>){
       const windowId = event.resourceId;  //get the session (window) from the LoginEvent
       this.context.router.replace('/workbench/' + windowId);
}
```

Logging Out

CvLogout provides callback for logging out via the user provided renderer function. Logout listeners will be invoked after logout occurs and the session has been invalidated. In addition, in the event the session has become invalid due to expiration, the logoutListeners provided to the top-level **CvAppWindow** will be invoked so that the invalid session can be handled.

```
const logoutListener = ()=>{ this.context.router.replace('/'); //return to the login page }

<CvLogout
   renderer={(cvContext:CvContext, callback:CvLogoutCallback)=>{
       return <a onClick={callback.logout}>Logout</a>
   }}
   logoutListeners={[logoutListener]}
/>
```

```
<CvAppWindow windowId={windowId} logoutListeners={[logoutListener]}>
   ...
</CvAppWindow>
```

Application Window

CvAppWindow is the container for a 'logged in' or active session. It is initialized with the windowId (or resourceId) obtained from the **CvLoginResult** upon login. The renderer is supplied with the **AppWinDef** SDK object via the **CvContext**.

```
<CvAppWindow windowId={windowId} logoutListeners={[logoutListener]}
    renderer={(cvContext:CvContext)=>{
      const appWinDef:AppWinDef = cvContext.scopeCtx.scopeObj as AppWinDef;
      return(
        <div className="cv-window">
          <div className="cv-logo pull-left"/>
          {this.props.children}
        </div>
      );
}}/>
```

Workbenches and Launchers

Workbenches and their composite Launchers can be accessed and rendered with the CvWorkbench and CvLauncher components respectively. The CvLauncher's renderer callback may be used to fire the Launch Action. Following a successful launch, the registered **'launchListener'** functions will receive a CvNavigationResult object that may be displayed with a CvNavigation component.

```
<CvWorkbench workbenchId={workbenchId} appWinDef={appWinDef} renderer={(cvContext:CvContext)=>{

    const workbench:Workbench = cvContext.scopeCtx.scopeObj as Workbench;
    const renderedWorkbench =
       workbench.workbenchLaunchActions.map((launchAction:WorkbenchLaunchAction) => {

         return(
           <CvLauncher launchAction={launchAction} key={launchAction.actionId}
             launchListeners={[this.launchListener]} actionListeners={[this.actionListener]}
             renderer={(cvContext:CvContext, callback:CvLaunchActionCallback)=>{

               const launcher = cvContext.scopeCtx.scopeObj as WorkbenchLaunchAction;
```

```
                return (
                  <div onClick={callback.fireLaunchAction} className="launcher-container">
                    <img className="launcher-icon" src={launcher.iconBase}/>
                    <h4 className=”launcher-text">{launcher.name}</h4>
                  </div>
                );

              }}/>
           );

        });

     return <div>{renderedWorkbench}</div>

}}/>

actionListener: function(actionEvent:CvEvent<CvActionFiredResult>) {
   this._showWaitState(actionEvent);
}
launchListener: function(launchEvent:CvEvent<CvNavigationResult>)=>{
   const navigationId = launchEvent.resourceId;
   this.context.router.push('/navigator/' + windowId + '/' + navigationId);
}
```

Navigations and Forms

CvNavigation allows for the rendering of the result of a **‘Navigation’**, which is most often a Catavolt ‘Form’, composed of one or more subsections of List panes and Details panes or other specialized versions of these (i.e. Graphs, Maps, Calendars, etc.) A CvNavigation should contain a child CvForm component, and any corresponding subcomponent to be targeted by **‘paneRef’** for rendering. The **paneRef** parameter is a 0-based index that allows for targeting specific subcomponents that have been previously defined within a Form.

```
<CvNavigation navigationId={navigationId}>
   <CvForm>
       <div className="form-container" renderer={(cvContext:CvContext)=>{
           const formContext:FormContext = cvContext.scopeCtx.scopeObj as FormContext;
           return <div>
               <div className="panel-heading">{formContext.paneTitle}</div>
               <div className=”list-container”>
                 <CvListPane paneRef={0}>
                   <CvRecordList wrapperElemName={'span'} rowRenderer={(cvContext, record)=>{
                    //render the list
                    ...
                   }}/>
                 </CvListPane>
               </div>
           </div>
       </div>
}}/>
```

Actions

The CvAction component corresponds to ‘Actions’ that may be performed on the server-side.

‘actionListeners’ can expect to receive notification of the action starting and ending.

‘navigationListeners’ can expect receive a CvNavigationResult if the action completes successfully.

```
<CvAction actionId={'alias_createMessage'}
  navigationListeners={[(e:CvEvent<CvNavigationResult>)=>{
       this.navigateToMessageCreated(e.eventObj);
   }]}
   actionListeners={[(e:CvEvent<CvActionFiredResult>)=>{
       const actionFiredResult = e.eventObj as CvActionFiredResult;
       if(actionFiredResult.type === CvActionFiredResultType.ACTION_STARTED) {
         this.showLoadingIndicator();
       } else if(actionFiredResult.type === CvActionFiredResultType.ACTION_COMPLETED) {
         this.hideLoadingIndicator();
       }
   }]}
   renderer={(cvContext:CvContext, callback?:CvActionCallback)=>{
         return <a onClick={callback.fireAction}>Fire Action!</a>
  }}/>
```

Automatically Firing Actions

CvActions can also be ‘automatically’ fired upon mounting, by providing a callback function as the **‘fireOnLoad’** parameter.

```
<CvAction actionId={'#search'}
   paneContext={listContext}
   fireOnLoad={(success, error)=>{
     if(error){
       //report error
     }
   }}
   navigationListeners={[(event:CvEvent<CvNavigationResult>)=>{
       this.setState({searchNavResult: event.eventObj});
   }]}/>
```

Searching

The CvSearchPane is a specialized CvDetailsPane (both are built on CvEditorBase mixin) that can be targeted with the returned **CvNavigationResult** after performing a ‘#search’ action. In addition to behaving like a standard **CvDetailsPane**, **CvSearchPane** provides a CvSearchPaneCallback to the supplied detailsRenderer. **CvSearchPaneCallback** provides methods for getting/setting search parameters and enabling search operations in addition to also providing all of the methods available on the **CvDetailsPaneCallback** object.

```
<CvSearchPane paneRef={0}
   detailsRenderer={(cvContext:CvContext, entityRec:EntityRec, callback:CvSearchPaneCallback)=>{

     //use callback to set set search params, etc

}}/>
```

Error Handling

Errors can usually be handled in one of two ways. They may be handled directly by providing a callback to a specific operation, or the handling logic can be ‘decoupled’ from the invocation of specific operations by using an ‘event listener’ to respond to errors and messages.

Handling Errors Directly

Most callback objects' methods allow you to supply a 'CvResultCallback' function as a parameter. For example, CvActionCallback's **fireAction** method is defined as:

```
fireAction(resultCallback?:CvResultCallback<CvNavigationResult>):void;
```

The **CvResultCallback** type is just a function, with the following signature:

```
(success:A, error?:any):void
```

In the fireAction example, the provided callback function will either receive a **CvNavigationResult** (if the action was successful) or an error object.

Generalized Error Handling

There are 2 ways to register for 'CvMessage' events, including errors:

1) Use a CvMessagePane with a render method to automatically catch these and call your render method (you could filter on type error here if that's all you're interested in):

```
return <CvMessagePane messageRenderer={(cvContext:CvContext, message:CvMessage,
callback:CvMessagePaneCallback)=>{

    if(message.type === CvMessageType.ERROR) {
        if(message.messageObj && message.messageObj instanceof DialogException) {
            const dialogException:DialogException = message.messageObj as DialogException;
            const serverMessage = dialogException.message || dialogException.name;
            //do something with the serverMessage
         }
    } else {
        return null;
    }
}}/>
```

2) Alternatively, register a listener directly with the **CvEventRegistry** instance to catch and handle them yourself (**CvMessagePane** does this behind the scenes):

```
(cvContext.eventRegistry as CvEventRegistry).subscribe<CvMessage>(this._messageListener,
CvEventType.MESSAGE);

_messageListener: function(event:CvEvent<CvMessage>) {
       this.setState({message: event.eventObj});  //do something with the message
}
```

More Detail on Message Events:

Any type of message, (CvMessageType: **ERROR**, **WARN**, or **INFO**) is published as an event by the event registry. For example, the event that gets published in the failed CvLauncher case looks something like this:

```
const event:CvEvent<CvMessage> =
{
  type:CvEventType.MESSAGE,
  eventObj:{
                   message:'Launch of ' + this.launchAction().name + ' failed',
                   messageObj:launchTry.failure,
                   type:CvMessageType.ERROR
                }
}
```

Base Component

CvBaseMixin

CvBaseMixin is included as a ‘mixin’ in every ‘Cv’ component. This makes the following operations available to all components:

catavolt():AppContext
The ‘catavolt’ object (of type **AppContext**) is the entry point into the Catavolt Javascript SDK

eventRegistry():CvEventRegistry
The global, shared instance of the **CvEventRegistry** that allows for subscribing to and publishing **CvEvent**s

scopeCtx():CvScopeContext

Each component instance has a corresponding **CvScopeContext** that provides access to a relevant ‘scopeObj’, usually from the Catavolt Javascript SDK (i.e. a **CvListPane** has a **ListContext** as its scopeObj) and also access to the ‘parentScopeObj’ if available.

Accessing From Outside of a Component

In most cases, an application using the Cv components should not need to access the objects provided by the **CvBaseMixin** directly. However, in cases where it is necessary, it is best achieved by using React’s ‘ref’ mechanism, which returns a component instance after it mounts. Any component instance can be accessed this way:

```
<CvListPane ref={(theListInstance)=>{
             const eventRegistry:CvEventRegistry = theListInstance.eventRegistry();
             //do something with the eventRegsitry
 }}>
```