Seamless Integrated Consistent Personalised

Enactor Training Course
Web Connector and React POS

Web Connector and React POS

Introduction to the Web Connector

- Architecture
- How to use the Web Connector

Enactor React POS

- Architecture
- Customising the React POS

Integration with External Web Pages

What is the Enactor Web Connector?

A JavaScript library that allows you to interact with an Enactor Application Process Runtime

Allows reuse of application level logic that may have been developed for a Point-of-Sale in other channels:

- Take advantage of a Java back-end in your web-application
- Doesn’t make any assumption on the technology in the UI layer
- Doesn’t require the development of an additional REST service layer

Helps promote a clean separation between View logic (in the Browser) and Controller logic (in the Application Process)

Enactor Web Connector

The web page may be hosted on an external web site, or could be “bundled” with the Enactor POS

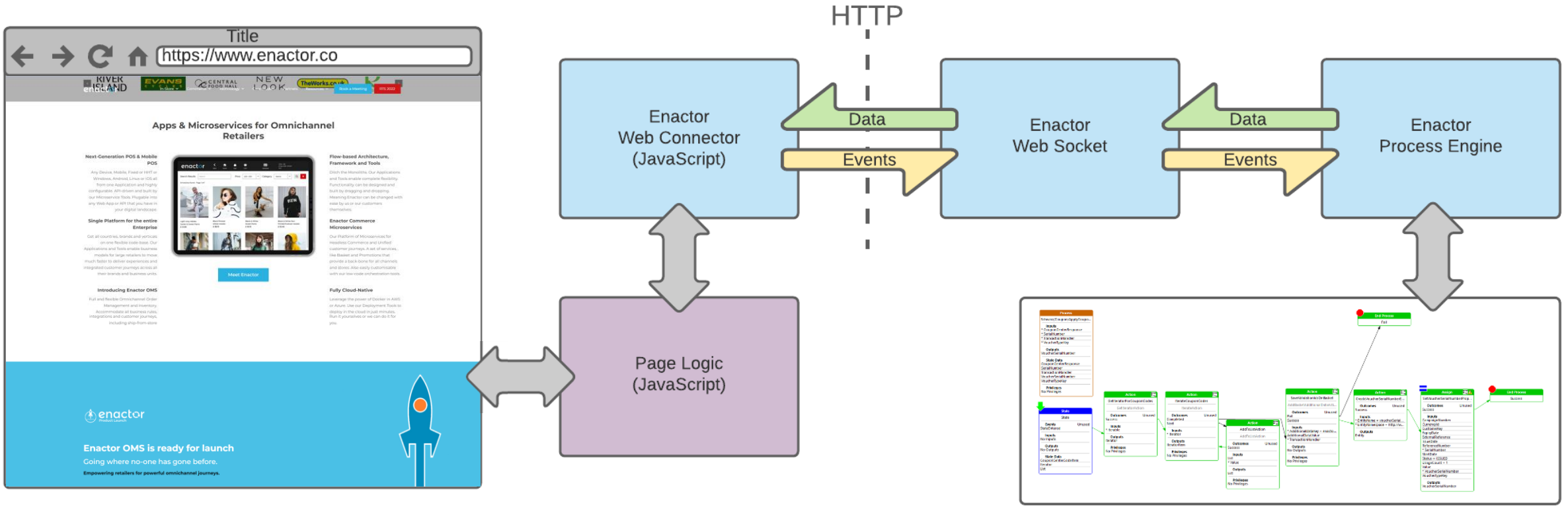
Provides the full power of the Enactor POS

- Peripheral integration, including card readers and printers
- Native support for the Enactor Basket and Promotion Engines
- All Enactor Application Process are available for use

Allows you to move Rest service invocation to the Application layer, out of the View layer if desired

- May be suitable for sensitive APIs that you do not want to make available publicly on the internet

Architecture of the Enactor Web Connector



Enactor Web Connector API

The API for the Enactor Web Connector is separated into three areas:

- Web Connector Configuration API
  This is used to configure how the Web Connector should talk to the server-side web socket

- Web Connector Manager API
  This is use to manage Web Connector instances

- Web Connector API
  The API for interacting with the connected Application Process

Configuration API

This offers a “fluent” or “builder” style pattern for preparing the configuration needed to connect to the server-side web socket:

- withBridgeHostEndpoint(hostURL)
  The URL hosting the web socket

- withMessageResourceId(messageBase, isDefault)
  A default message resource that the web connector should use (other message resources can be resolved as needed)

- withBridgeEventHandler(eventName, callback)
  Register a callback handler for a Web Connector Event

Manager API

This API allows the application to create a new Web Connector:

- createBridge(config, callback, autoConnect)
  Call to establish a new bridge to the Web Connector, supplying a configuration object prepared by the Web Connector Configuration API

  The callback will be invoked once the bridge has been established

  The autoConnect flag controls if the connection to the server should be immediately established, or if only the necessary object structures should be prepared

- getBridge()
  Locate and return any existing instance of the Web Connector

Connector API

This is the main API for the Web Connector:

- getPromptData(name, callback)
  Asks the server-side to supply some data held by the current Prompt

- sendEvent(event, data, eventHandlers)
  Send an event to the server-side, potentially with some data
  The event handlers will be called if any matched events are raised while the event is being handled

- getResource(resourceType, resourceURI, resourceResolvedCallback)
  Ask the Web Connector to return a resource that is available in the server-side application

Connector API (cont.)

- testPrivilege(privilegeId, callback)
  Check if the “user” has a particular privilege assigned to them. The callback will be supplied the result of the authorisation check

- resolveExpressionForPage(pageId, expression, callback)
  Asks the server to resolve an expression. This is useful where there is some dynamic data available on the server that is hard to pass through as prompt data

- registerForViewEvent(eventName, callback)
  Allows the UI to be notified when the Application Process receives an Event

- getViewData(dataName, callback)
  Asks the server-side to supply some “view data” that it is holding

Connector API (cont.)

The Web Connector also provides a number of properties that describe the current state of the server-side Application Process:

- page
  An object structure that holds details about the current prompt being “displayed” on the server-side

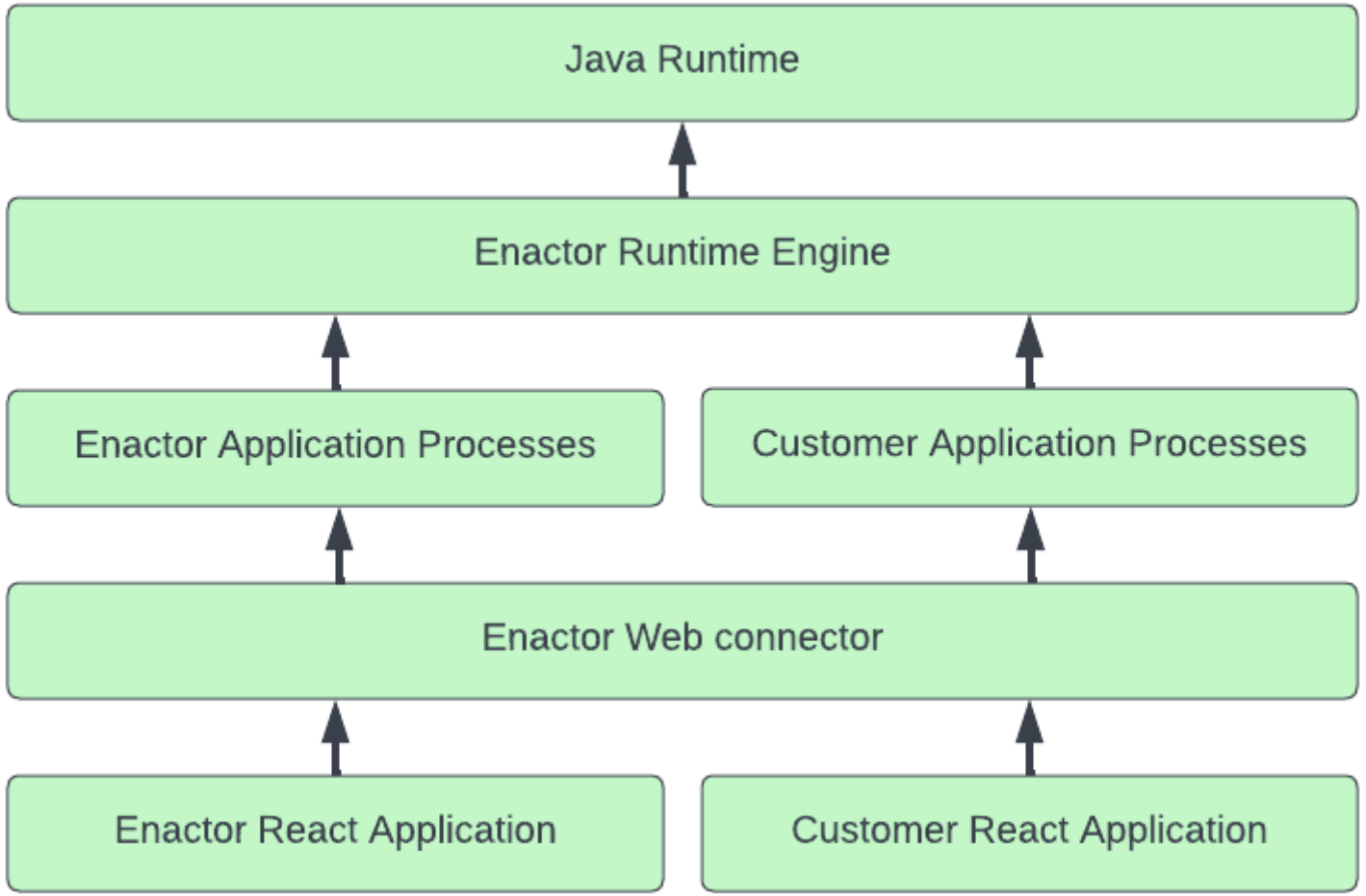
- bridgeType
  A string that can allow the HTML to change its behaviour based on the type of environment it is running in (iOS / Android / Swing)

- theme
  A string that describes what “Theme” has been configured in the POS Terminal for this device

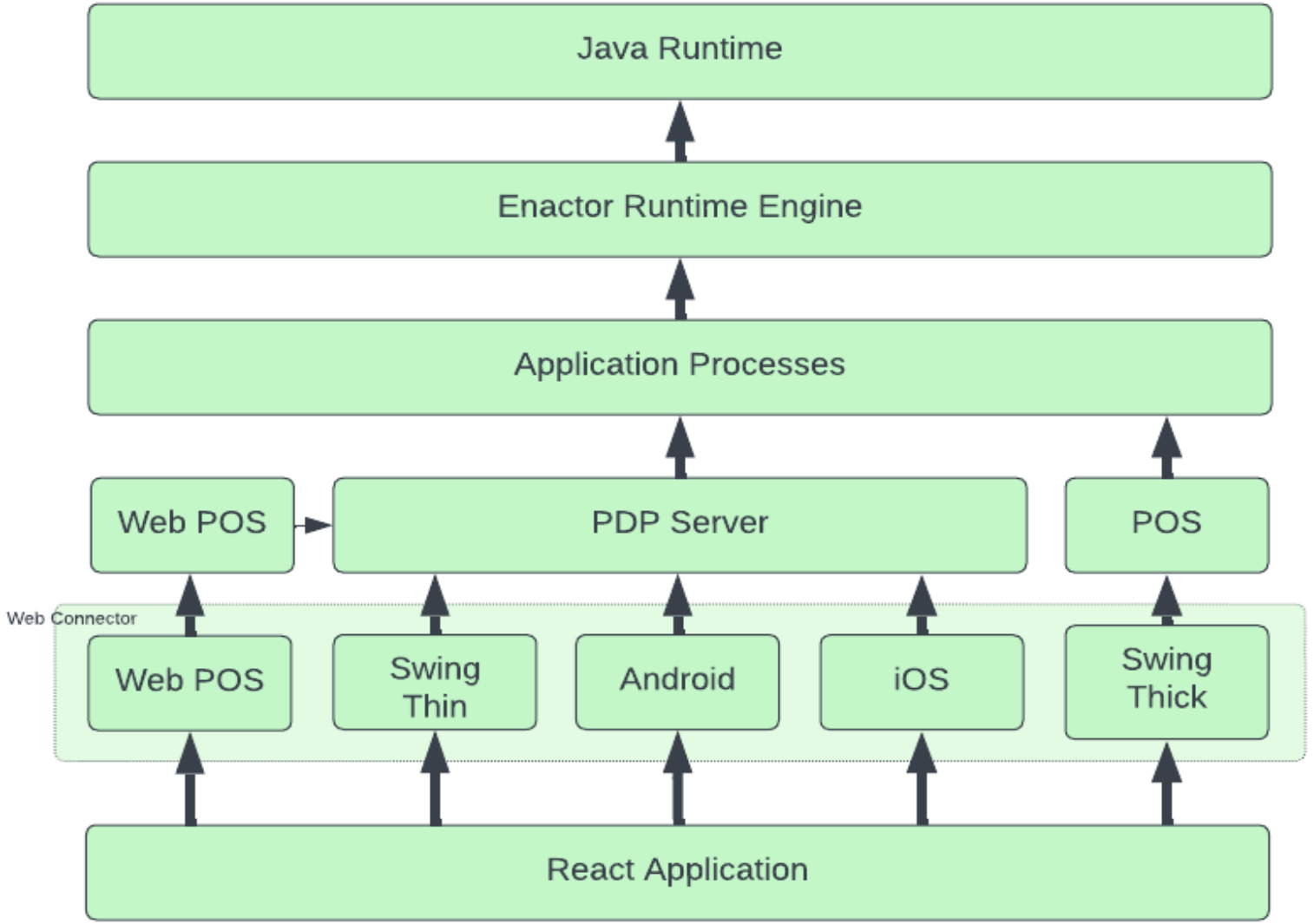
- locale
  An ISO locale code that describes the locale of the signed-on user

Enactor React POS

React POS Architecture



Architecture



React POS – Swing Thick

Full Screen Browser

Enactor Application Processes

Prompts

Prompt Renderer

Data Processor

JSON

React App

React POS – Android / iOS

React App

JSON

Thin Client

XML wrapped JSON

Enactor Thin Server

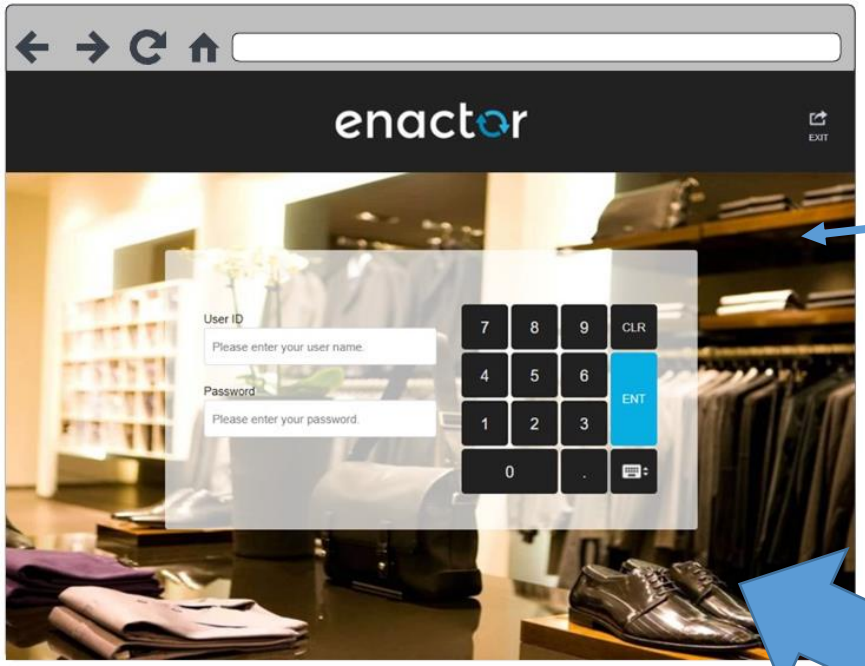
Prompts

Data Processor

Prompt Renderer

Thin Server Socket

React POS – Web POS



Web Server

index.js

Web Socket

JSON

Enactor Thin Server

Prompts

Prompt Renderer

Data Processor

Thin Server Socket

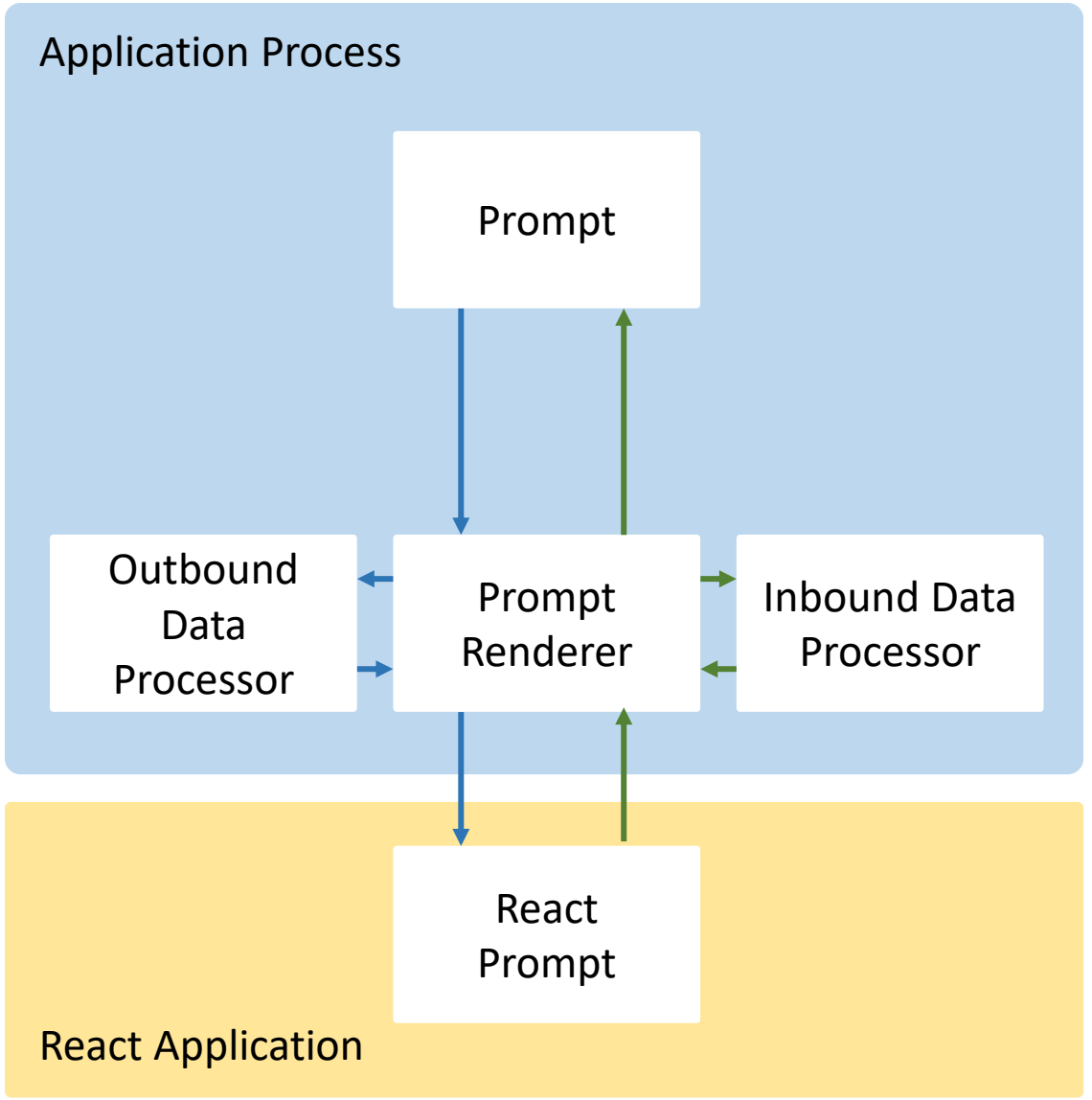
XML wrapped JSON

Data Processors

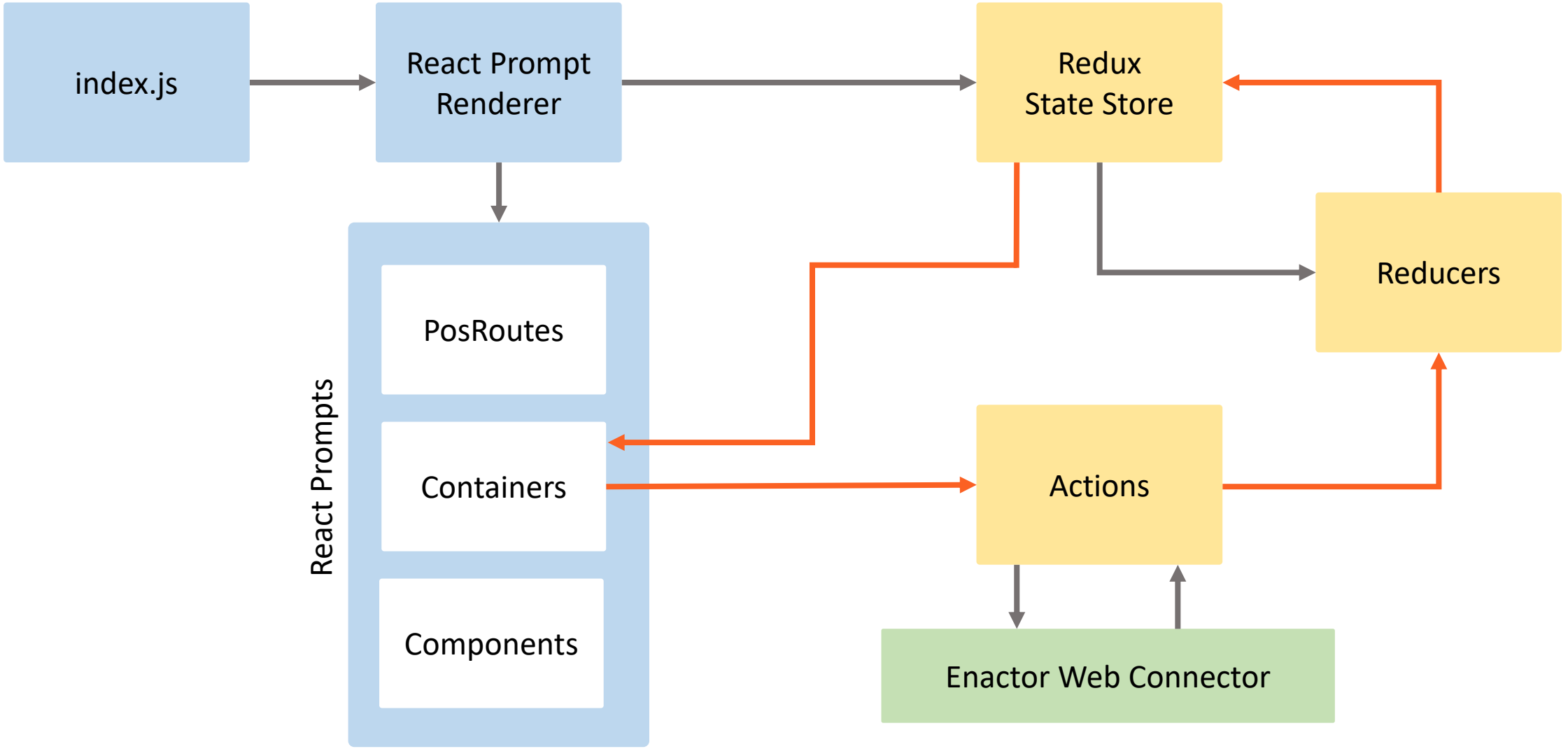
Data Processors are used for preparing the data needed by the React Application

They can be used to augment the data available in an Application Process

They also support converting data from the user interface into a format usable by the application



React POS Architecture



Routing

In a typical React app, a react-router is used for routing.

We use our own route resolving mechanism; we maintain the route mappings in our components in the PosRoutes root module.

The root component that handles this is the PromptRenderer.

```
const PosRoutes = {
    "Pos/Sale/Sale": {
        promptUrl: "Pos/Sale/Sale",
        component: PosContainer
    },
    "Pos/Modifier/ModifyItemOptionsPrompt": {
        promptUrl: "Pos/Modifier/ModifyItemOptionsPrompt",
        component: ModifyItemOptionsPrompt
    },
    "Pos/Modifier/ModifyTenderOptionsPrompt": {
        promptUrl: "Pos/Modifier/ModifyTenderOptionsPrompt",
        component: ModifyTenderOptionsPrompt
    },
    "Pos/Modifier/ModifyBundleItemOptionsPrompt": {
        promptUrl: "Pos/Modifier/ModifyBundleItemOptionsPrompt",
        component: ModifyBundleItemOptionsPrompt
    },
    "Pos/General/MessagePromptCancelOnly": {
        promptUrl: "Pos/General/MessagePromptCancelOnly",
        component: PosContainer
    },
    "Pos/General/MessageOKCancelPrompt": {
        promptUrl: "Pos/General/MessageOKCancelPrompt",
        component: MessageOKCancelPrompt
    },
    "Pos/General/MessagePrompt": {
        promptUrl: "Pos/General/MessagePrompt",
        component: MessagePrompt
    },
    "Pos/General/NoPrompt": {
        promptUrl: "Pos/General/NoPrompt",
```

Customising the React POS

Enactor React POS applications can be customised using the following:

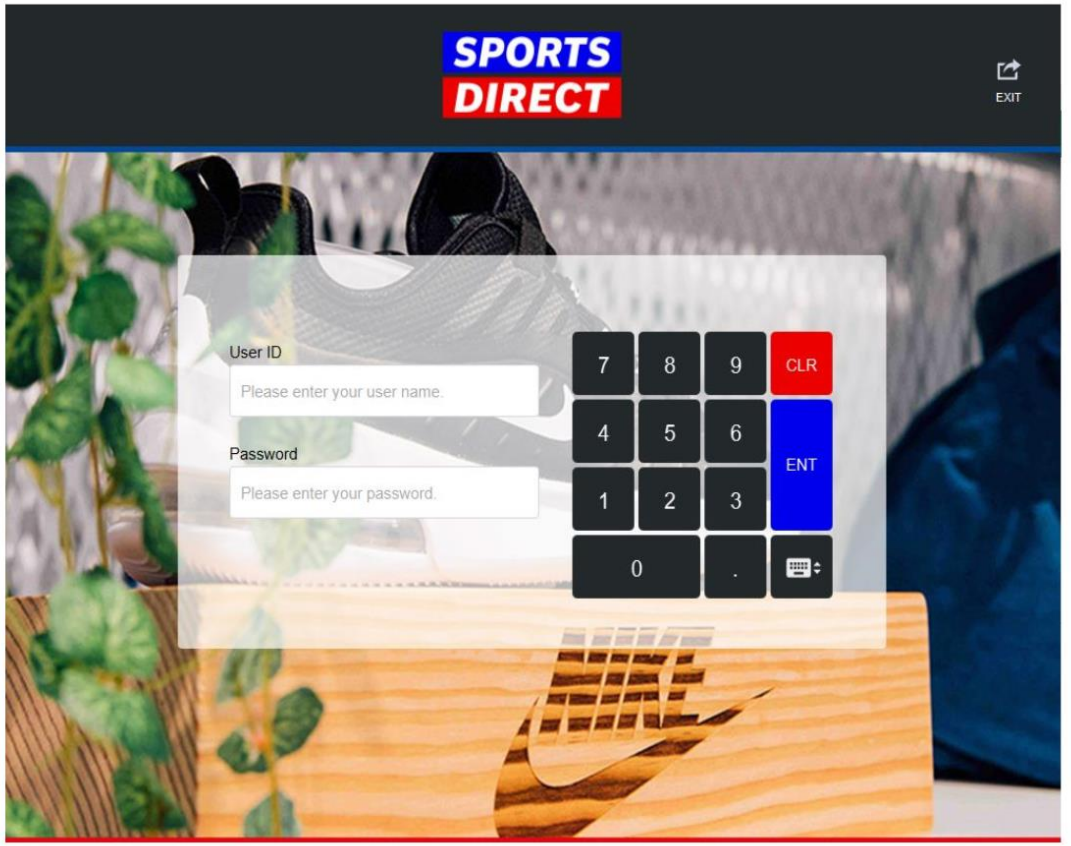
- Stylesheets
  All elements are styled using a stylesheet. You can load your own stylesheet to override the standard Enactor styling

- Base Component
  A simple React component independent of the POS. These are building blocks that other React components can be built from

- Page
  A page is a component that manages the overall layout for a prompt from the POS

- Widget
  A React component that renders a part of a prompt

Stylesheet Theming



Customising Base Components

The default Base Components are exported using the EnactorDefaultBaseComponentsMap, this should be used as a basis for your own component maps when you are customising the Enactor React POS:

```
import { EnactorDefaultBaseComponentsMap } from
   "@enactor/react-pos";
import MenuButton from
   "../components/Overrides/BaseComponents/MenuButton"

const DemoBaseComponentsMap = {
   "MenuButton": MenuButton
};

export default Object.assign({},
   EnactorDefaultBaseComponentsMap, DemoBaseComponentsMap);
```

React Standard Widgets

Enactor React POS uses a number of widgets to build the user interface:

- Connected Input (Text, Drop Down, Checkbox, Calendar)
- Buttons
- Onscreen Keyboard
- Table View
- Image
- Resolvable Message
- Formatted Amount

Customising Widgets

Widgets can be customised by supplying a custom componentsMap when the EnactorProvider is created, for example:

```
const MyCustomComponentsMap = {
        "Header": MyCustomerHeader
};

const EnactorProvider = ({
        verbose = false,
        componentsMap = MyCustomComponentsMap,
        children,
        appContext
}) => {
        ...
}
```

You only need to define the components you want to override; any components you do not provide an override for will use the default widget

Development Tools

The React POS runs using the Chromium rendering engine.

As a result, you can use the standard Chrome developer tools to investigate the structure of the React POS from within the POS itself

You can also remotely connect a full browser to the React POS at the URL:

http://localhost:9222/

Integration with External Web Pages

Integration with External Web Pages

Using the Web Connector, you can take full advantage of the running POS application from your Web Site

You can detect if you are running in a POS, and if so change the web site behaviour

For example, if a product is added to the “basket”, when you are running in the POS you can add the item to the POS basket itself, rather than the web basket.

Using the Web Connector in a Web Page

Enactor provides examples that demonstrate how to use the Web Connector in your own Web Pages in our Confluence pages

This can allow you to embed pages from your standard e-commerce website in the POS and then integrate them so that you can update the POS basket

Notes on how to use the external bridge

Importing:

```
import { EnactorExternalBridge } from "@enactor/javascript-bridge";
```

Creating the external bridge

```
var externalBridge = new EnactorExternalBridge();
externalBridge.init() // this connects to the bridge
```

Grabbing the current page response and prompt data

```
var pageResponse = externalBridge.getPageResponse();
var promptData = pageResponse.promptData;
```

Sending an event to the bridge

```
externalBridge.sendEvent("<eventName", { <data: optional> } )
externalBridge.sendEvent("OKPressed", { reasonId: "COD004" } )
```

Selling a product

```
var postData = {
            "enactor.mfc.ProductCode" : productId,
            "enactor.mfc.ProductQuantity" : quantity
        }
externalBridge.sendEvent("QuantitySellProduct", postData);
```

Using the Web Connector in a Web Page

To allow a web page to easily detect if it is being run on a POS device, Enactor will add a string into the User-Agent header:

- JavaFX or Enactor-Chromium
  Indicates the Swing POS is running the page
- Enactor-Android
  Indicates the Android Thin Client is running the page
- Enactor-iOS
  Indicates the iOS Thin Client is running the page

enactor®

retail systems for a digital world

Q & A