CCL1127-K23 - Lab: Build the ServiceNow photobooth with custom component and UI Builder (UIB)

ServiceNow

K23-CCL1127 Build the K23 Photobooth App Part 1 - The Custom Component

Goal

The goal of this lab is to help you learn the basics of creating a simple custom UI Component for use within UIB, using the Photobooth app from the K23 floor as an example.

ServiceNow needed a photobooth for fun photoboothy experiences at the Knowledge23 and CreatorCon23 showroom floors. It needed to be flexible, work on a variety of client devices such as iPads and desktop browsers, and integrate nicely with the third party tools used to manage the show. Sounds like a great application for a UI Builder app, but unfortunately there is no camera component out of the box. Today you will learn how to fix that!

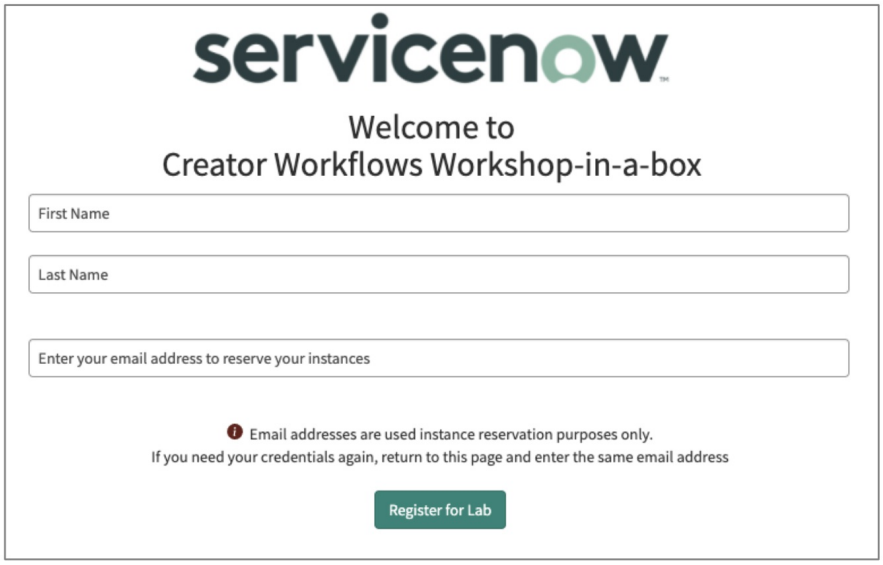
> Note: If completing this lab in your own environment jump down to Appendix 1 for instructions for installing the Now CLI and the UI Component extension. This lab guide is geared to those using the provided Cloud Labs Windows instances, so if you are working on your own you may need to tweak the steps slightly.

Getting Started

Register your Lab Instances and Connect via RDC

If you are using ServiceNow instances as part of a lab, you will need to get a ServiceNow instance and a Windows (MID) Server provisioned in the lab before you get started.

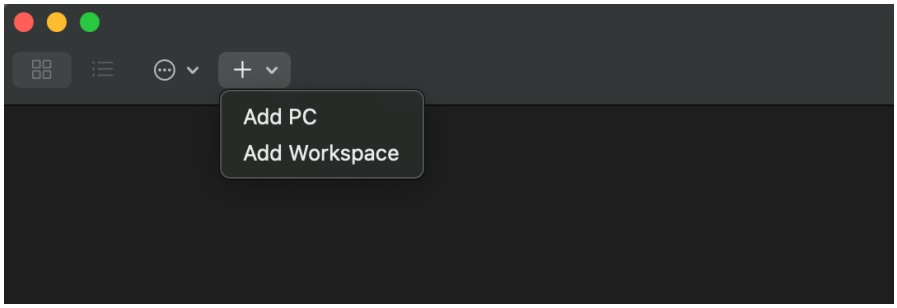
1. Open the link to the lab registration.
2. Type in your First Name, Last Name, and Email address and click ‘Register for Lab.’



Register for Lab Page

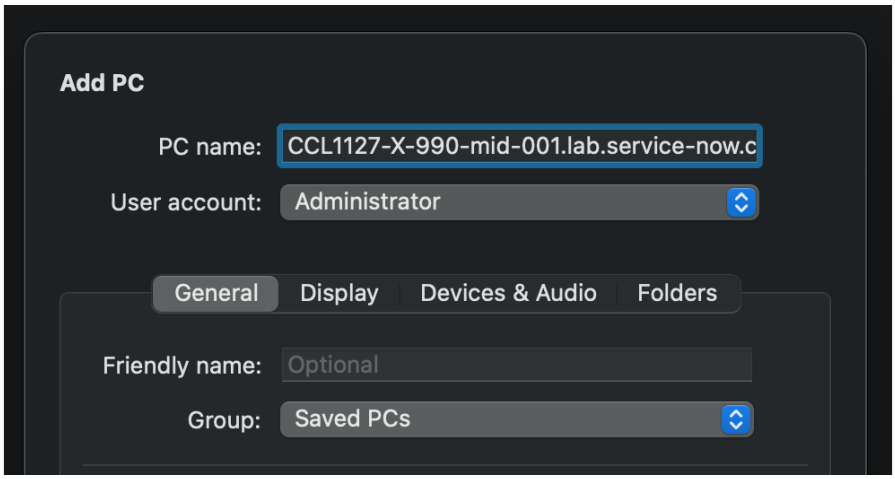
3. Type in the reservation code: XXXXX
4. You will be provided a ServiceNow instance with admin credentials–*make a note of the admin password*–as well as a Windows MID server upon which we will be doing our development work. If you have not already done so, install the Microsoft Remote Desktop client.
5. Copy the address to the MID server (without the "https://" prefix).

   Of the two VMs it will be the one with the word "mid" in it, e.g. "CCL1127-X-990-mid-001.lab.service-now.com".
6. Open Remote Desktop client and choose "Add PC".



Add PC

7. Paste the MID server's address into *PC name* and select "Administrator" as the *User account*.



Add PC Dialog

8. Double click the icon to connect.

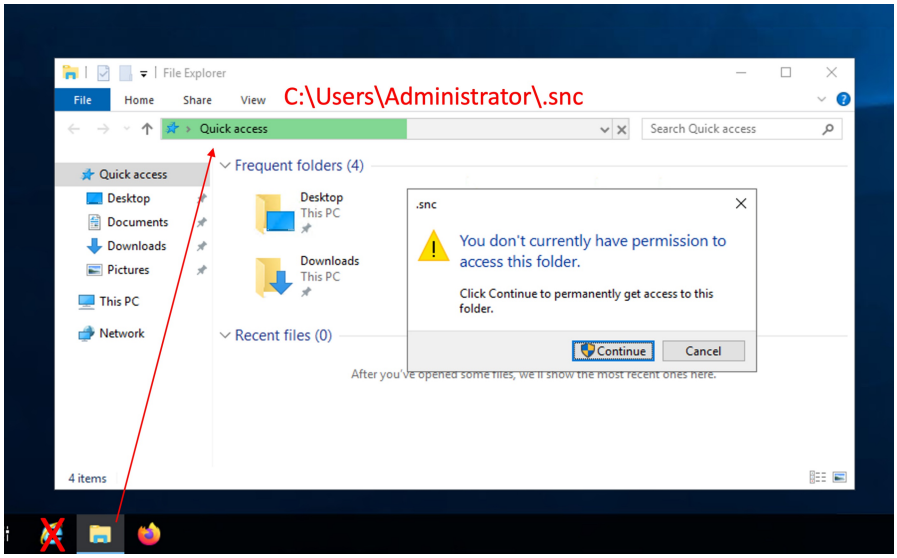
Grant Access to the ".snc" Folder

A quirk of the installation in the cloud labs VM is acess to the CLI's configuration folder is not owned by the Administrator account, which we can fix manually. (This will not be an issue for your own dev environment).

1. Open Windows Explorer and type the following path into the navigator bar:

```
C:\Users\Administrator\.snc
```

2. You will be prompted to grant permission, click "Continue".

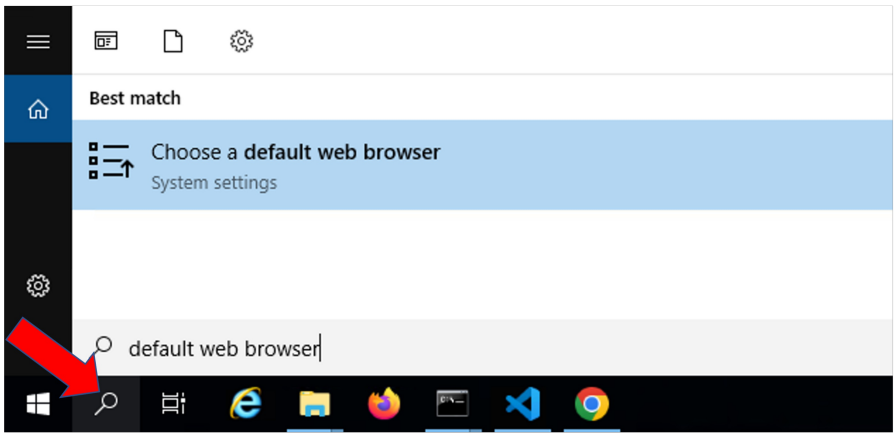


Grant Access to .snc

Make Chrome your Default Browser

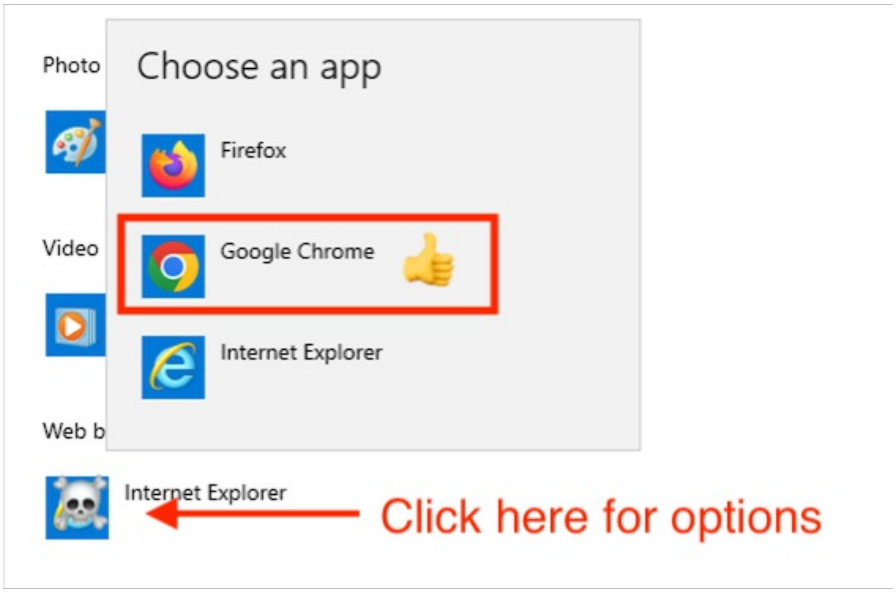
For the cloud labs instance Chrome will be used as the default browser and a special command line used to start the browser with a mock video feed. You may be disappointed that we are changing the default web browser from Internet Explorer.

1. Click “Search” (magnifying glass) and search for “Default Web Browser”.



Search for Default Web Browser system setting

2. Scroll down to “Web browser”, click the Internet Explorer icon and select **Google Chrome** from the choices.



Select Chrome as Default Browser

Create new Dev Directory and Open VSCode

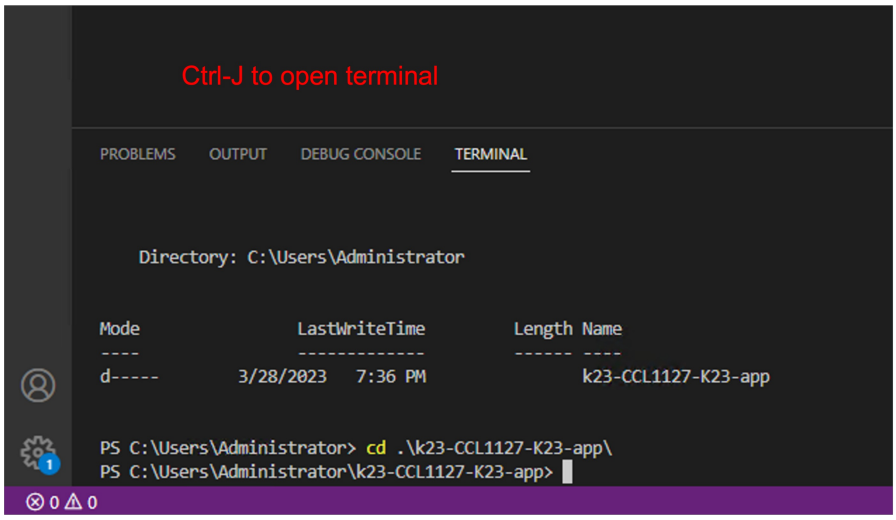
1. Click the Windows start menu and type **CMD** and hit enter.

2. Make a new directory called *k23-CCL1127-K23-app* for the component, change into it and start Visual Studio Code from this directory.

```
mkdir k23-CCL1127-K23-app
cd k23-CCL1127-K23-app
code .
```

3. Type “Ctrl-J” to bring up a Powershell terminal window.

   NOTE: All commands from this point forward will be run from within VSCode, which uses Powershell.



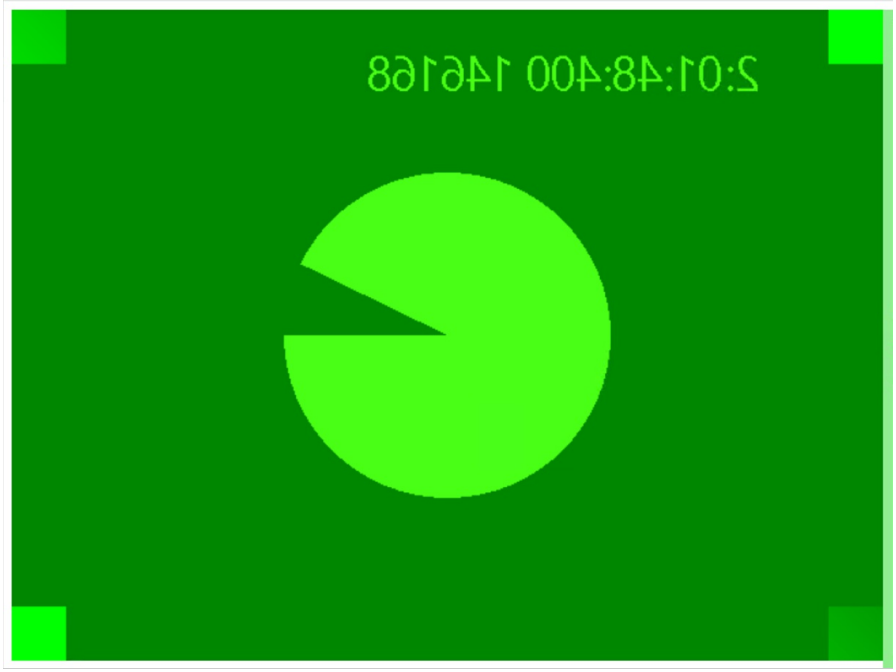
Ctrl-J to Open Terminal

Note: To paste commands into the VSCode terminal in the remote desktop **Right Click** within the terminal.

4. Open Chrome with a mock video feed.

```
./start-chrome.bat
```

Note: While running this in a virtual machine as part of the lab you will not have access to your camera feed. Using the start-chrome.bat file causes Chrome to be loaded to display a test pattern. To verify your camera you may check https://webcamtests.com/. When running this on your own machine the camera will work as expected.


Test Pattern

Create a text file on your VM and copy links and passwords locally

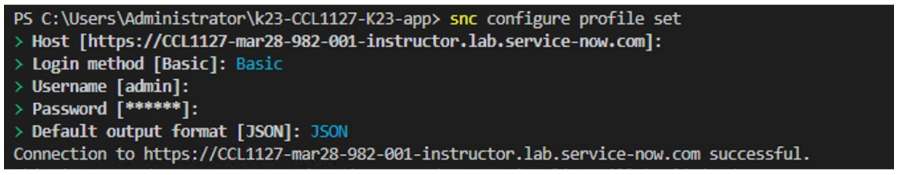
Pro Tip: copy and paste important links and passwords to a new text file on your VM. Save the link to the lab guide book, your instance, and the password in a text file.

Open the guide book in a browser window in the VM. This will make copy and pasting large text blocks safer and more efficient.

Authenticate

1. Set your default profile pointing to the ServiceNow Instance that was provisioned when you registered your lab instances (something like “https://CCL1127-X-990-001.lab.service-now.com”) using the admin credentials provided during registration.

```
snc configure profile set
```



Configure Profile

2. Authenticate a proxy for your app to your dev ServiceNow instance using the following command, but providing your instance’s URL, username and password.

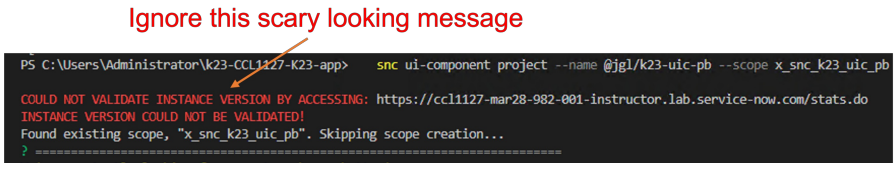
```
snc ui-component login {instance URL} basic {username} {password}
```

Link to Git Repo and Scaffold Project

1. Use the SNC Command Line tool to scaffold the project, using a scope name from the instance. (Note that *name* uses hyphens and *scope* uses underscores and that for Cloud Labs instances this scope has already been created).

```
snc ui-component project --name @jgl/k23-uic-pb --scope x_snc_k23_uic_pb
```

Ignore the scary looking red message.



Ignore the Red Messages

2. Link to the Git repo for this lab, getting the source code and switching to the branch exercise_0 which contains the initial state before installing npm dependencies to scaffold the project.

   NOTE: The following steps are more complex due to the fact that we are scaffolidng then pulling from Git. For your future projects you will scaffold then push.

   Type each of the following commands one at a time.

```
git init
git remote add origin https://github.com/ServiceNowEvents/k23-CCL1127-K23-app.git
git fetch
git checkout exercise_0 -f
```

3. The last step is to install the dependencies from NPM (note that this may take a while).

```
npm install
```

4. Run the following command and verify that the component loads in your browser with the “hello world” message.

   NOTE: This is the one cheat I made to the source, adding the text “hello world”. When you create your own components it will start out blank.

```
snc ui-component develop --open
```

   Note: To stop the running web server use the “trashcan” icon on the terminal to kill the terminal session, then you may use Ctrl-J to open a new one.

5. Leave this command running–it is your web server and will recompile and reload your browser as we make changes to the code.

6. Close the Chrome browser now. We will re-open it with special parameters to mock the video feed in the next exercise.

How to Catch Up If You Get Behind

Time will fly so you may find yourself getting behind. If that happens use git to catch up to the completion of each exercise with the following command. In this example checking out the lab where it was when Excercise 1 was completed.

```
git checkout exercise_1 -f
```

Important Note: When switching branches be sure to **close your open code windows first** to avoid potential conflicts with changes in your editor and the files being checked out.

How to Catch Up

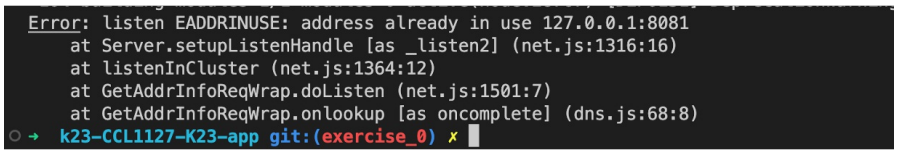
Gotchas

- Page doesn’t load in browser after “develop” command

Sometimes the page will compile and the server will start but the browser doesn’t automatically open the page. If that happens, just enter the URL into a tab:

```
https://localhost:8081
```

- Address Already in Use Error



Address Already In Use

You may find yourself with a rogue web server or the standard port of 8081 is taken while trying to run develop. In that case use the “–port {X}” parameter when opening develop

mode, e.g.:

```
snc ui-component develop --port 8082 --open
```

- Powershell Acting Weird

  If Powershell starts acting strange, e.g. only accepting every other key stroke or responding slowly to typing, or *if anything goes wrong at all really*, use the trash can to kill the terminal and Ctrl-J to bring up a new one.

Kill Terminal

Resources

The completed code, including challenges, is available in the main branch of the Git repo https://github.com/ServiceNowEvents/k23-CCL1127-K23-app.

The complete code for the Photobooth used on the Knowledge 23 show floor is available at https://github.com/ServiceNowNextExperience/photobooth.

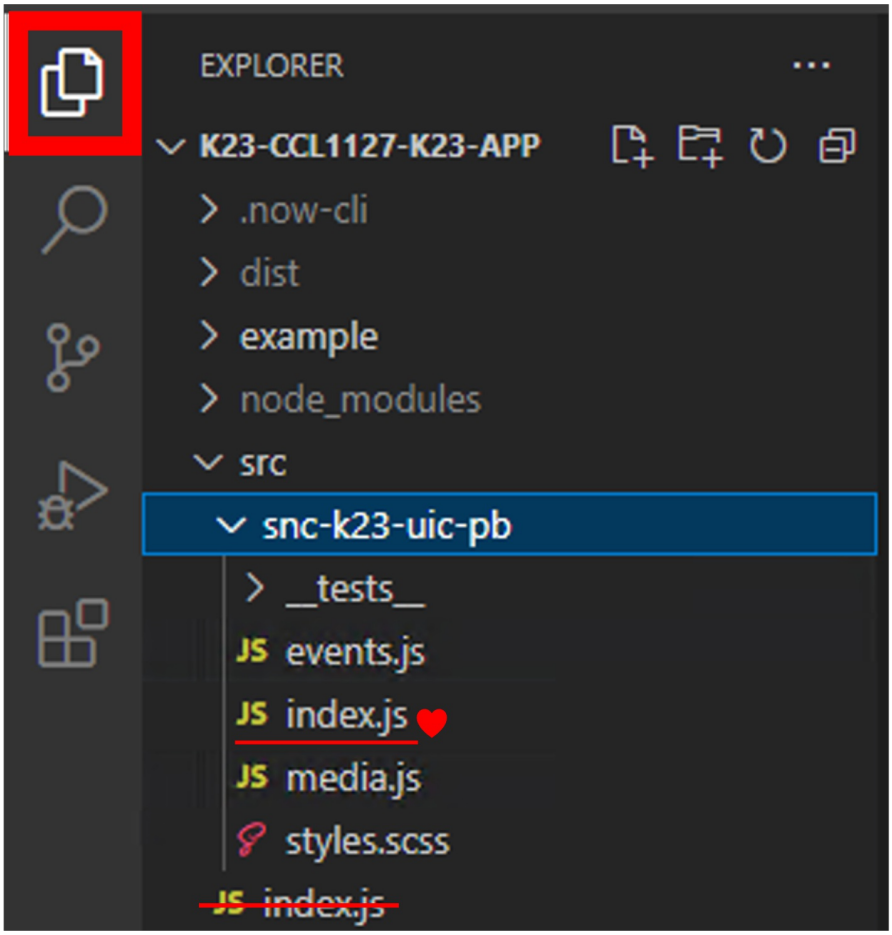
Exercise 1: Camera Initialization

Goal

Excellent! We are scaffolded and ready to start implementing the camera. The basic HTML and events necessary to render the camera will be implemented and the component tested locally with a mock camera by the end of this exercise.

HTML and Properties

1. Return to VSCode and inspect the directory structure.



Code Structure

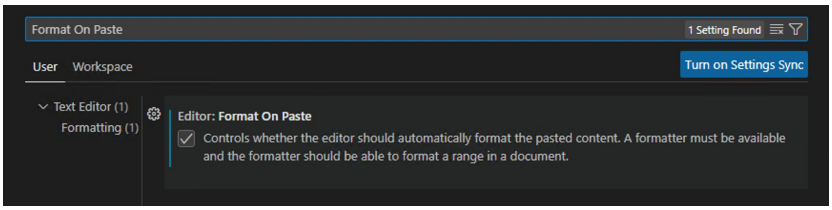
2. The main code file is located at **src/snc-k23-uic-pb/index.js**. This is where most of the work will be done for component development. In this file are *imports* at the top, which will be used to access dependencies and libraries, a const for the *view*which contains our HTML and the call to the *createCustomElement* method which bootstraps our component.

3. Turn on auto-formatting so that when you copy and paste you do not lose code formatting.

    1. In VSCode type **Ctrl-,** (Ctrl and comma).

    2. Search “Format on Paste”

    3. Check the box.



Enable Format on Paste

4. Edit the “const view” to remove “Hello World!” and instead render a “<video>” element. Also initialize blank “{}” constants for each of **initialState, actionHandlers, dispatches** and **properties**, then add references to those objects to the “createCustomElement” method call. (The details for what these do will be covered later.)

Yes, there is a good bit of copy and paste to get us started, but from this point we will cover how to use this framework to continue building our component and discover how these things work by seeing them in action.

File: *src/snc-k23-uic-pb/index.js*

```
import { createCustomElement } from "@servicenow/ui-core";
import snabbdom from "@servicenow/ui-renderer-snabbdom";
import styles from "./styles.scss";
import { actionTypes } from "@servicenow/ui-core";
import { PHOTOBOOTH_CAMERA_SNAPPED,
   PHOTOBOOTH_AVAILABLE_CAMERAS_UPDATED } from "./events";


const { COMPONENT_CONNECTED, COMPONENT_PROPERTY_CHANGED,
   COMPONENT_DOM_READY } = actionTypes;

const view = (state, { updateState }) => {
   return (
      <div>
         <video id="video" autoplay="" style={{ width: "800px" }}></video>
      </div>
   );
};

const initialState = {}; // Pre-set "state" variables

// Events that will be handled by this component
const actionHandlers = {};

const dispatches = {}; // Events that will be dispatched by this component

const properties = {
   /**
    * Camera is enabled
    * @type {boolean}
    */
   enabled: {
      schema: { type: "boolean" },
      default: true,
   },
   /**
    * Triggers a snapshot
    * Required: No
    */
   snapRequested: {
      default: "",
      schema: { type: "string" },
   },
};

createCustomElement("snc-k23-uic-pb", {
   renderer: { type: snabbdom },
   view,
   styles,
   initialState,
   actionHandlers,
   dispatches,
   properties,
});
```

5. These properties will be used more in the next exercise, but they ultimately will be set from UIB. They are accessible from your code via the “properties” object on “state”.

Initialize Media

1. Up near the top, after the “imports”, add an import for the method **selectMediaDevice** from

the *media.js* library. (Include **snap** and **toggleTracks** for future reference.)

File: *src/snc-k23-uic-pb/index.js*

```
import { selectMediaDevice, toggleTracks, snap } from "./media";
```

2. Add a new method called **intializeMedia** just after the "imports" at the top which will create the necessary HTML elements to stream video and compose the results on the DOM and use the media library to initialize the camera.

   File: *src/snc-k23-uic-pb/index.js*

```
const initializeMedia = ({
   host,
   updateState,
   properties: { enabled },
}) => {
   console.log("Initialize Media");

   // This is where the camera will be rendered
   // Note that "host" is how you get access to the DOM
   const video = host.shadowRoot.getElementById("video");
   // This is how the snapshot is composed
   const canvas = host.shadowRoot.ownerDocument.createElement("canvas");
   const context = canvas.getContext("2d");

   // Get access to the camera!
   selectMediaDevice({ enabled, video });

   // We will need these later when taking snapshots
   updateState({
      video,
      context,
   });
};
```

DOM Ready Action Handler

1. Events have passed-in dependencies. In this case *host* will be used to interact with the DOM, *state* to retrieve state variable, *updateState* to modify state variables, *dispatch* to emit events and *properties* to grab properties set by UIB.

2. To initialize our component and to monitor changes to properties we are interested in the events *COMPONENT_PROPERTY_CHANGED* and *COMPONENT_DOM_READY*. These were imported from **actionTypes** near the top of the file in the previous step.

3. Implement the handler for **COMPONENT_DOM_READY** to call the **initializeMedia** method. Go back near the bottom of index.js and modify the existing **actionHandlers** declaration.

   File: *src/snc-k23-uic-pb/index.js*

```
const actionHandlers = {
   [COMPONENT_DOM_READY]: ({
      host,
      state: { properties },
      updateState,
      dispatch,
   }) => {
      initializeMedia({ host, properties, updateState });
   },
};
```

4. Save your code and return to the browser and see the video stream in action.

5. Though not available in the lab, there will be an issue in the real world–when you move left the image moves right. That feels weird and is not what we expect after years of using video conferencing software. Add some CSS to flip the video stream horizontally.

    1. Styles are stored in the **styles.scss** file.
    2. Add this snippet near the top after the @import line to flip the video with a clever use of scaling.

    File: *src/snc-k23-uic-pb/styles.scss*

```
video {
  -webkit-transform: scaleX(-1);
  transform: scaleX(-1);
}
```

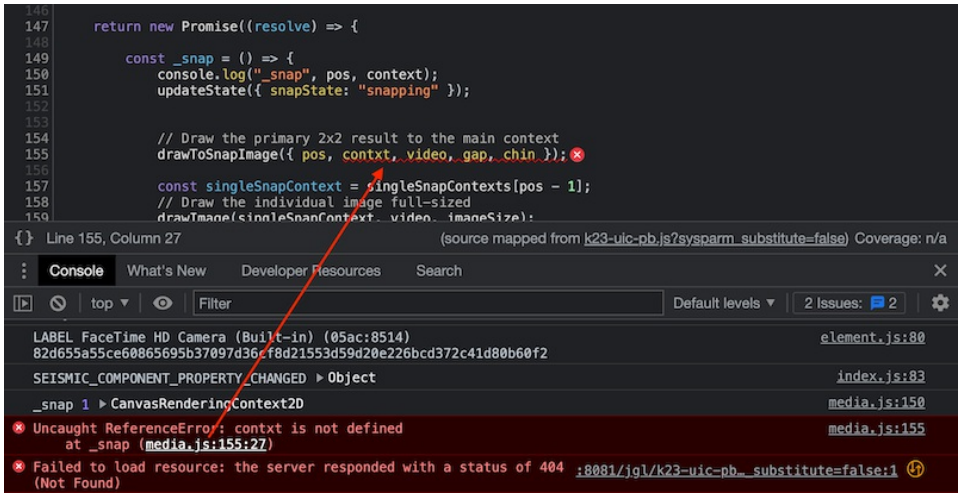
    1. Return to the browser and see the change!

Great job!

Debugging

If it is not working start by inspecting the console for errors (right-click on a blank section of the page and choose **Inspect**).

In this example I made a mistake and typoed a variable name in a library script. The console message explains that and clicking the link will open the source code for further analysis.



Example of Error in Console

Catch Up

If you cannot get it running, no worries! Use the following command to catch up:

```
git checkout exercise_1 -f
```

Reference

- index.js
- styles.scss

Exercise 2: Properties and Test Page

Goal

In this exercise additional properties will be created and the test page *element.js* used to modify those properties.

Properties allow the component to be configured by and interacted with from the application. Properties set values going into the component and events (aka “actions”) communicate data coming out.

> Fun Fact: This is called a “precipitation model”!

1. The property definitions in the **properties** object are based on the JSON Schema.
2. At the basic level it is just a schema type and a default value, but this can also used to define more complex properties that you’ll see in UIB (e.g. enumerations of choices or accepting complex objects and arrays).
3. Add three more properties using the existing properties as a template.

Name	Type	Default
countdownDurationSeconds	number	0
pauseDurationSeconds	number	1
fillStyle	string	lightgreen

4. The syntax for properties is follows:

```
export const properties = {
 /**
  * Camera is enabled
  * @type {boolean}
  */
 enabled: {
  schema: { type: "boolean" },
  default: true,
 },

 /**
  * Triggers a snapshot
  * Required: No
  */
 snapRequested: {
  default: "",
  schema: { type: "string" },
 },

 /**
  * How long to wait after requesting a snap and beginning the shots.
  */
 countdownDurationSeconds: {
  default: 0,
  schema: { type: "number" },
 },

 /**
  * Number of seconds to pause between each snap.
  */
 pauseDurationSeconds: {
  default: 1,
  schema: { type: "number" },
 },

 /**
  * The html fillStyle property for the canvas, e.g. "green"
  * Required: No
  */
 fillStyle: {
  default: "lightgreen",
  schema: { type: "string" },
 },
};
```

Testing Locally

1. To test locally the file **/example/element.js** is used. Open the file and see that it is quite rudimentary. To make it easier to exercise the component it will be useful to have it operate similarly to the component, i.e. it will be turned into it's own *custom element*.

2. Copy and paste this boilerplate into *element.js*.

   > Note: this can be re-used as a template for your own future custom component dev.

   File: */example/element.js*

```
import { createCustomElement } from "@servicenow/ui-core";
import snabbdom from "@servicenow/ui-renderer-snabbdom";
import { PHOTOBOOTH_CAMERA_SNAPPED } from "../src/snc-k23-uic-pb/events";
import '../src/snc-k23-uic-pb';

const initialState = {
   // TODO: Add the starting values to initialize the component
   enabled: true,
   countdownDurationSeconds: 0
};

const view = (state, { updateState }) => {
   console.log("ELEMENT VIEW", state);
   const {
      // TODO: Add state variable reference here
      snapRequested,
      countdownDurationSeconds,
      enabled,
      imageData
   } = state;

   const requestSnap = (countdownDurationSeconds = 0) => {
      console.log("REQUEST SNAP");
      updateState({
         countdownDurationSeconds,
         snapRequested: Date.now() + "",
      });
   };

   return (
      <div id="element">
         <div style={{ display: "flex" }}>
            <div id="component" style={{ flex: 1 }}>
               <snc-k23-uic-pb
                  snapRequested={snapRequested}
                  countdownDurationSeconds={countdownDurationSeconds}
                  enabled={enabled}
               ></snc-k23-uic-pb>
            </div>
            <div id="outputs" style={{ flex: 1 }}>
            </div>
         </div>
         <div id="controls">
            {/* Place buttons or other controls here */}
            <button on-click={() => requestSnap(countdownDurationSeconds)}>
               Snap!
            </button>
            <button on-click={() => updateState({ enabled: !enabled })}>
               Toggle Enabled
            </button>
         </div>
      </div>
   );
};

const actionHandlers = {};

createCustomElement("example-element", {
   initialState,
   renderer: { type: snabbdom },
   view,
   actionHandlers
});

const el = document.createElement("main");
document.body.appendChild(el);

el.innerHTML = "<example-element/>";
```

3. Save the file, return to the browser and notice the addition of the “Snap” and “Toggle Enabled” buttons.

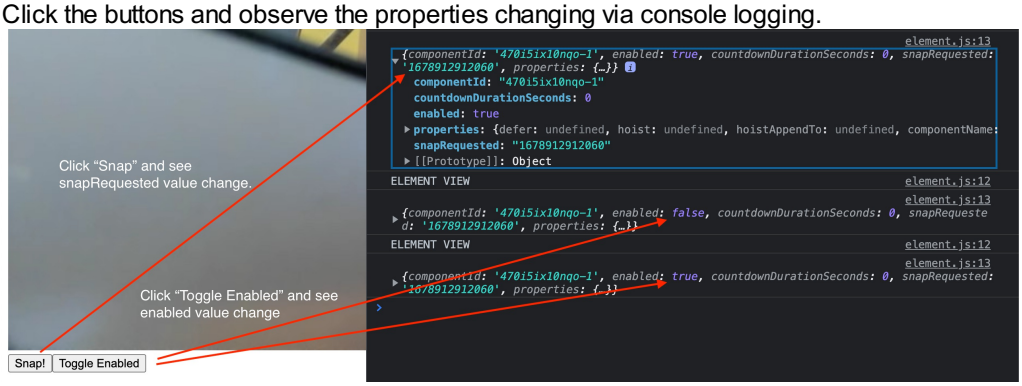
Properties Overview

Properties pass values *into* the components and events return values *out of* the component. Code will be added to monitor properties for changes and to respond when activities complete.

Inspect Property Changes

The browser inspector can be used to observe properties changing.

1. Open the component in the browser and right-click on a blank part of the page and choose “Inspect”.
2. Click the buttons and observe the properties changing via console logging.



React to Property Changes

Code will be added to monitor for changes to properties within the component, and the **enabled** property wired up to toggle blanking out the camera.

1. Open *index.js* and add a new Action Handler for **COMPONENT_PROPERTY_CHANGED** after the existing “COMPONENT_DOM_READY” handler, just before the closing brace of the *actionHandlers* object.

File: *src/snc-k23-uic-pb/index.js*

```
[COMPONENT_PROPERTY_CHANGED]: ({
   state,
   action: {
      payload: { name, value, previousValue },
   },
   dispatch,
   updateState,
}) => {
   console.log(COMPONENT_PROPERTY_CHANGED, { name, value });
   const {
      snapState,
      video,
   } = state;

   const propertyHandlers = {
      enabled: () => {
         toggleTracks({ video, enabled: value });
      }
   };

   if (propertyHandlers[name]) {
      propertyHandlers[name]();
   }
}
```

2. This action handler is called whenever *any* property changes.

3. The *propertyHandlers* object is a collection of methods to respond to property changes by name. So far an implementation for **enabled** has been created to call *toggleTracks* from the *media.js* library.

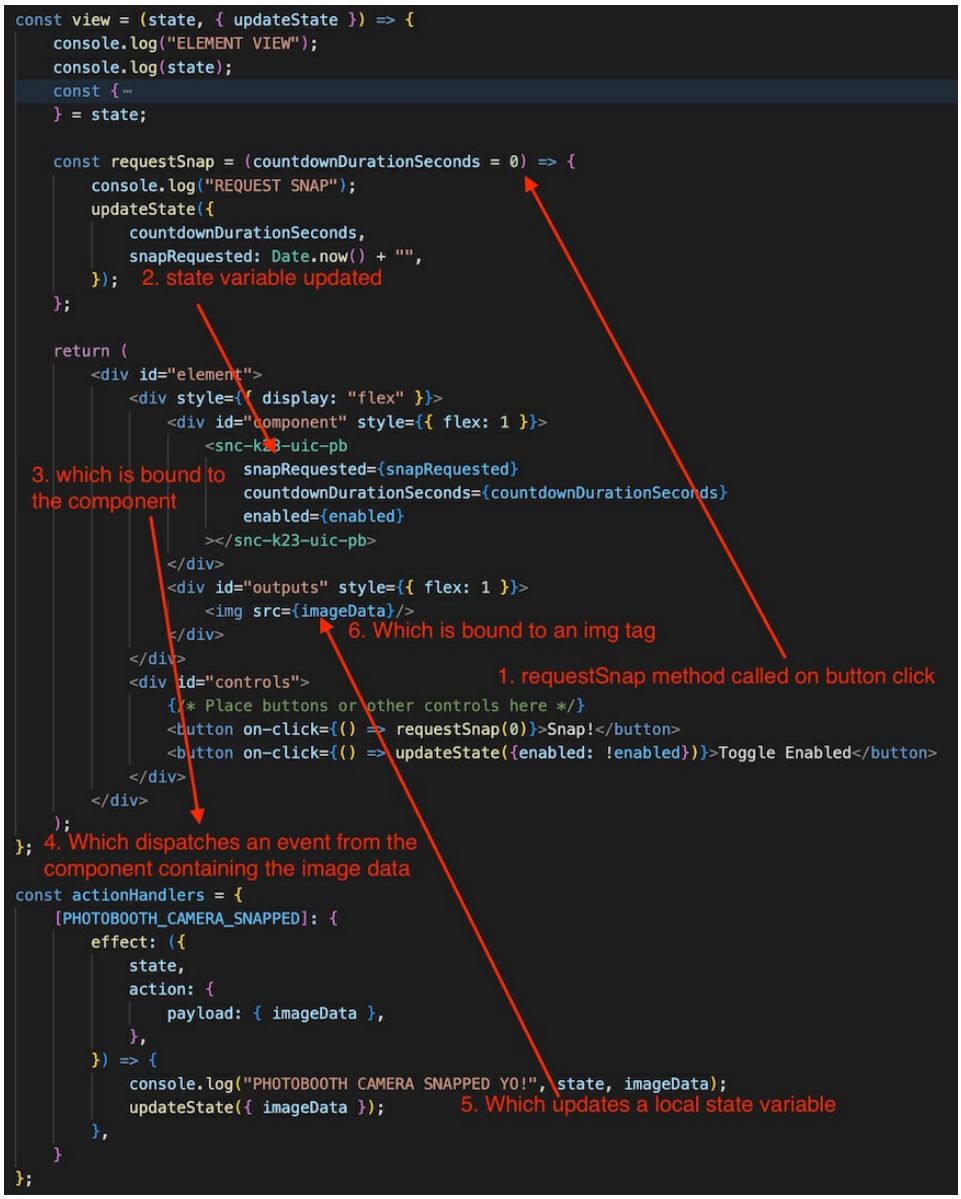
4. Start develop mode if it's not still running and verify that clicking "Toggle Enabled" blanks out the camera. Great–we can finally stop the stressful spinning green pacman!

```
snc ui-component develop --open
```

Event Lifecycle

In order to understand the event lifecycle, look at the process that happens starting with the "on-click" event of a button in our element.js test page.

1. on clicking the "Snap!" button the *requestSnap* method is called, (2) which updates the state variable called *snapRequested*, (3) which updates the property within the component which performs the snapshot and then (4) dispatches an event that is captured by the test page which then (5) updates the local *imageData* variable to (6) render the image.



Event Lifecycle

Optional Challenge

This lab is depending on default values to configure itself, but in the real world more properties will be useful and they will all need to be settable from outside the component, i.e. from UIB. Add the following properties and wire them up to be used within *index.js* from the properties object and to be settable from *element.js* for testing.

> Note: These values will be automatically passed to the *snap* method in media.js via the state properties–you need only wire them up in the **properties** object and **now-ui.json**.

1. Add Additional properties:

Name	Type	Default	Description

Name	Type	Default	Description
imageSize	object	{ width: 800, height: 600 }	Size of the output image snaps
gap	number	10	Gap in pixels between each image in the output image
chin	number	0	Height of “chin” at bottom of output snap to be used for a watermark

2. Go to *element.js* and add a text box that accepts a number for the countdown duration seconds. Bind that number to a state variable using the *on-blur* event on the input field and use this instead of hard-coding “0” in the call to the **requestSnap** method.

Reminder: The completed code with challenges is available in the main branch of the Git repo.

Catch Up

If you cannot get it running, no worries! Use the following command to catch up:

```
git checkout exercise_2 -f
```

Reference

- element.js completed
- index.js completed

Exercise 3: Taking a Snapshot

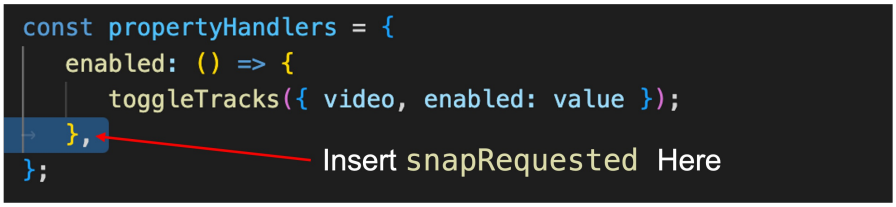
Goal

In this exercise the photobooth will finally start working like a photobooth. An event will be implemented to take a snapshot and compose it into a 4x4 format and return the content to the test page to render the image.

Take a Snap by adding a new Property Handler

1. Add a method to **propertyHandler** for **snapRequested** which calls the *snap* method passing in *state* and *updateState* just after the existing *enabled* method in *propertyHandlers*.

   File: *src/snc-k23-uic-pb/index.js*



Insert new snapRequested in propertyHandlers

```
snapRequested: () => {
   snap({ state, updateState }).then(({ context }) => {
      console.log("SNAP COMPLETED", context);
      const imageData = context.canvas.toDataURL("image/jpeg");
      dispatch(PHOTOBOOTH_CAMERA_SNAPPED, { imageData });
   });
}
```

2. In this method the canvas context is converted to an imageData string and dispatched with the PHOTOBOOTH_CAMERA_SNAPPED event to be handled by the calling application.

Render the Snap in the test page

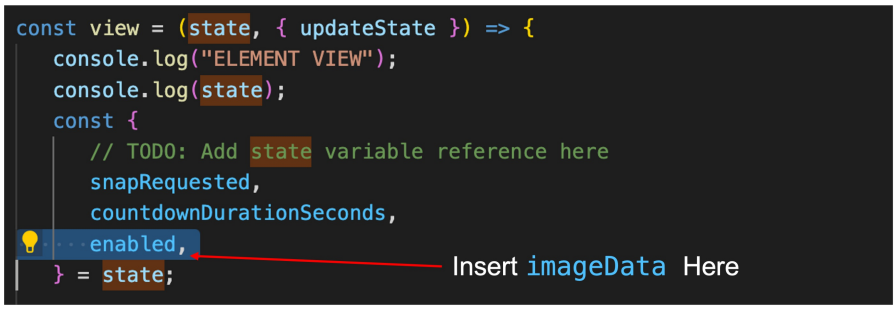
Next an Action Handler will be implemented in the test page to render the image.

1. Open *element.js* and add an implementation for **PHOTOBOOTH_CAMERA_SNAPPED** to the *actionHandlers* object that calls *updateState* and stores the *imageData* value.

   File: *example/element.js*

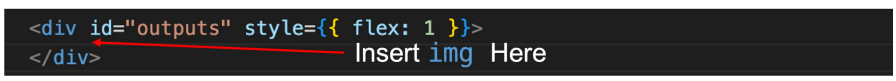
```
const actionHandlers = {
   [PHOTOBOOTH_CAMERA_SNAPPED]: {
      effect: ({
            state,
            updateState,
            action: {
               payload: { imageData },
            },
      }) => {
            console.log("PHOTOBOOTH CAMERA SNAPPED YO!", state, imageData);
            updateState({ imageData });
      },
   }
};
```

2. Add reference to **imageData** in *view*declaration. In order to use a state variable within a script block it must be declared first. In this case add it just affter the existing *enabled* variable in the section with the comment “Add state variable reference here”.



Insert imageData reference in view

3. Add an IMG tag and bind it to the *imageData* as its “src” to render the picture within the div “outputs”.



Insert new IMG tag in outputs div

```
<img src={imageData} />
```

4. Run the component, use “Snap!” and watch it take your first snapshot!

```
snc ui-component develop --open
```

Add a nice “Flashing” effect

Pretty good so far, but it’s not clear for the user when the snaps are being taken. Next a “flashing” effect will be added to blank out the screen briefly during each snap.

1. Some CSS has been provided in the file *styles.scss*. This includes an animation called “flasher” and some CSS targeting a DIV with the ID of “flash” when it has the class “snapping” applied. To control the behavior this the class will be added in the HTML to trigger the rendering of the flashing DIV by complexifying the HTML a bit.

2. Find the **view** within *index.js* and update it with a DIV with the id of **container** housing a new DIV called **flash** alongside the original video DIV. Don’t forget to include a reference to the **animationDuration** property with an “s” suffix, e.g “1s”. Also reference the variable **snapState** and bind it the attribute *className* on the *container* div.

   File: *src/snc-k23-uic-pb/index.js*

```
const view = ({
   snapState,
   properties: {
      pauseDurationSeconds,
      animationDuration = pauseDurationSeconds + "s",
   },
}) => {
   return (
      <div id="container" className={snapState}>
         <div
            id="flash"
            style={{
               "animation-iteration-count": 4,
               "animation-duration": animationDuration,
            }}
         ></div>
         <video id="video" autoplay="" style={{ width: "800px" }}></video>
      </div>
   );
};
```

3. In order to make the *snapState* default to “idle” find the **intialState** declaration and add the variable and specify the default value.

   File: *src/snc-k23-uic-pb/index.js*

```
const initialState = { snapState : "idle" };
```

4. Run the component locally and verify the flashing effect.

```
snc ui-component develop --open
```

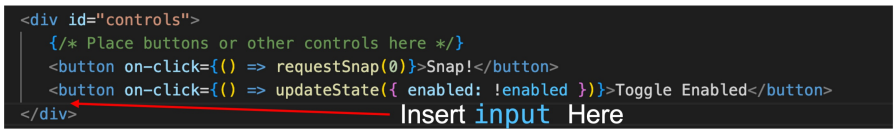
Bind a Text Field to Set Countdown Duration

Next we will bind a text field on the *element.js* page to allow specifying the countdown duration in seconds.

> Note: the *countdownDurationSeconds* property was added to the component in a previous exercise.

1. Go to *element.js* and add a new INPUT text field immediately after the buttons in the “controls” div.

   File: *example/element.js*



Insert new INPUT tag in controls div

Note that none of the values need double-quotes unless specified in this table.

<input> field attributes:

Attribute	Value
type	“number”
style	{{ width: “2rem” }}
value	{countdownDurationSeconds}

2. Add another attribute for the **on-blur** event that accepts an object called “target” with a field called “value” and pass that to the local variable state called **countdownDurationSeconds** (store the value there for use when the “snap” button is clicked).

   Your new html element should look something like this:

```
<input
   type="number"
   style={{ width: "2rem" }}
   value={countdownDurationSeconds}
   on-blur={({ target: { value } }) =>
      updateState({ countdownDurationSeconds: value })
   }
/>
```

3. Save it and run it and verify a pause matching the duration of the value you entered in the input field before snapshots begin.

```
snc ui-component develop --open
```

Optional Challenge

Add multi-camera support so that the user can switch between web cams (e.g. front or back camera on an iPad).

To accomplish this a list of available cameras is returned from the component, the list rendered in the *element.js* page, and when a selection is made the new camera device ID is updated within the component and the active camera is switched.

1. Add a property to hold the selected camera device ID.

Name	Type	Default
cameraDeviceId	string	“”

2. Use the method **getConnectedDevices** from *media.js* during intialization to retrieve a list of available cameras then dispatch the event *PHOTOBOOTH_AVAILABLE_CAMERAS_UPDATED* with that list.

3. Bind that event to a list or selector in *element.js* and when the user selects a camera update **cameraDeviceId** in the component.

4. When the **cameraDeviceId** property changes inside the component call **selectMediaDevice** again to switch to the newly selected camera.

   Reminder: The completed code with challenges is available in the [main branch of the Git repo](#).

Catchup

If you cannot get it running, no worries! Use the following command to catch up:

```
git checkout exercise_3 -f
```

Reference

- [element.js completed](#)
- [index.js completed](#)

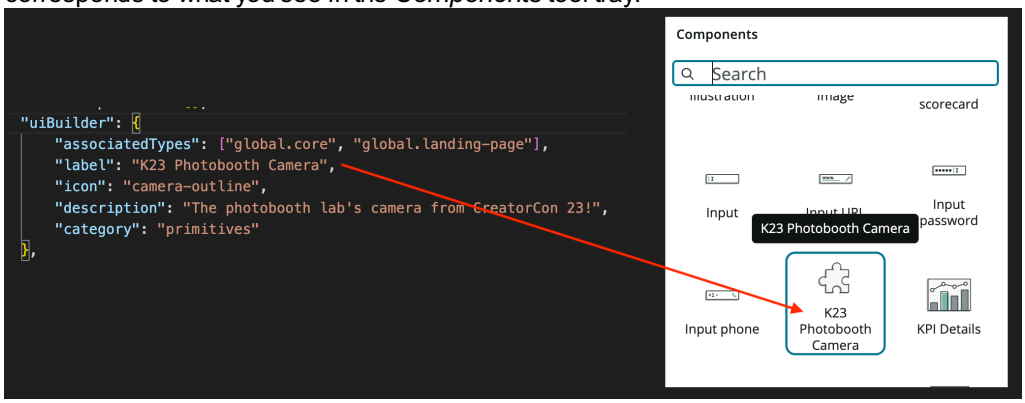
Exercise 4: Deploying to an instance

The ultimate goal when creating a custom component is to deploy it to an instance, and to do that requires a little bit of busy work.

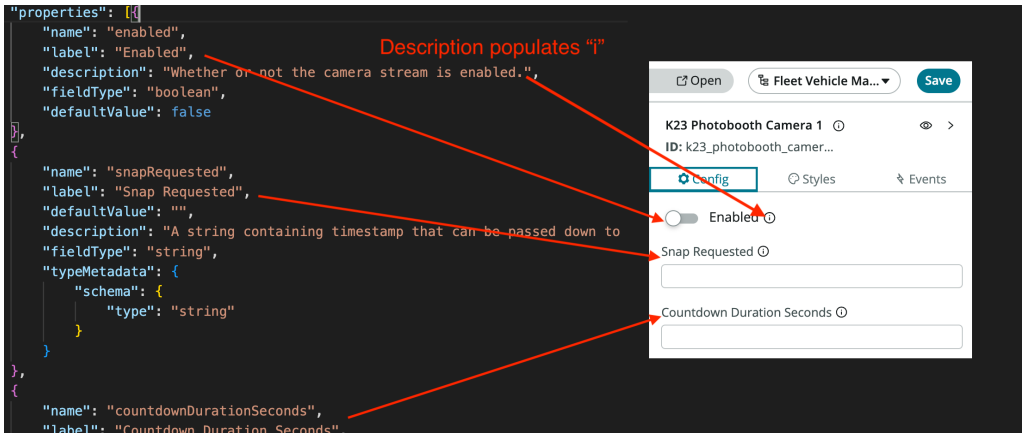
now-ui.json

This file is used to allow UIB to work with your component. There are three primary sections, but you get a simple copy automatically created in the root of your project.

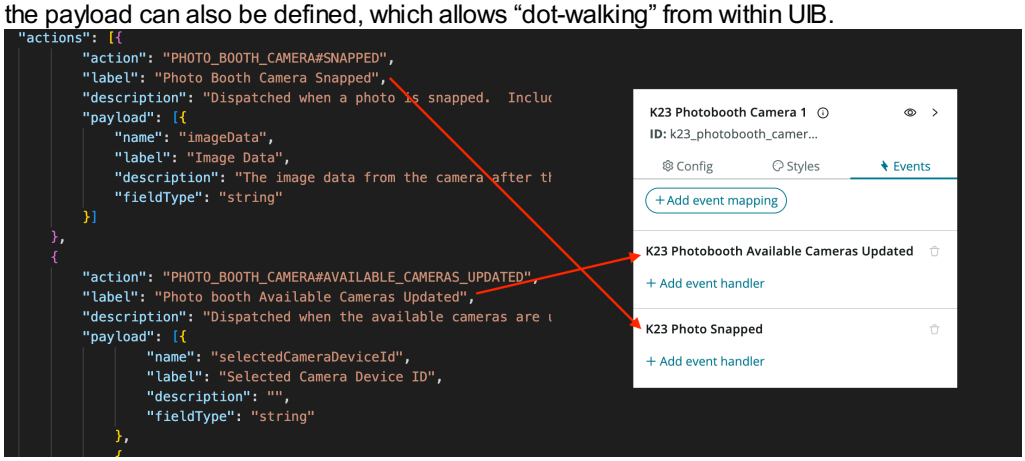
1. **uiBuilder** is used to specify the label, icon and description of your component. This corresponds to what you see in the *Components* tool tray.



2. **properties** is an optional array that comes just after uiBuilder. This data is partially redundant to the properties object which you defined in your code (i.e., every property should have an entry in the code and the JSON file), but is necessary for UIB to offer the ability to configure your component on a page and bind data to it.



3. **actions** follows immediately after that and is an optional array of actions which the component may emit. This corresponds to the "Events" tab in UIB. Details on the schema of the payload can also be defined, which allows "dot-walking" from within UIB.



4. Copy and paste the following JSON into your *now-ui.json* file.

   Note: Here is a link to the file in source control to make copy and paste easier.

   File: */now-ui.json*

```
{
  "components": {
    "snc-k23-uic-pb": {
      "innerComponents": [],
      "uiBuilder": {
        "associatedTypes": ["global.core", "global.landing-page"],
        "label": "K23 Photobooth Camera",
        "icon": "camera-outline",
        "description": "The photobooth lab's camera from CreatorCon 23!",
        "category": "primitives"
      },
      "properties": [{
        "name": "enabled",
        "label": "Enabled",
        "description": "Whether or not the camera stream is enabled.",
        "fieldType": "boolean",
        "defaultValue": false
      },
      {
        "name": "snapRequested",
        "label": "Snap Requested",
        "defaultValue": "",
        "description": "A string containing timestamp that can be passed down to trigger a snap. Use script to set this val
```

```
      "fieldType": "string",
      "typeMetadata": {
        "schema": {
          "type": "string"
        }
      }
    },
    {
      "name": "countdownDurationSeconds",
      "label": "Countdown Duration Seconds",
      "defaultValue": 0,
      "description": "Number of seconds to wait after Snap Requested to take the shot",
      "fieldType": "integer",
      "typeMetadata" : {
        "schema" : {
          "type" : "number"
        }
      }
    },
    {
      "name": "pauseDurationSeconds",
      "label": "Pause Duration Seconds",
      "defaultValue": 1,
      "description": "Number of seconds to wait between each shot",
      "fieldType": "integer",
      "typeMetadata" : {
        "schema" : {
          "type" : "number"
        }
      }
    },
    {
      "name": "fillStyle",
      "label": "Fill Style",
      "defaultValue": "",
      "description": "The html fillStyle property for the canvas, e.g. 'green'",
      "fieldType": "string",
      "typeMetadata": {
        "schema": {
          "type": "string"
        }
      }
    }],
    "actions": [{
          "action": "PHOTO_BOOTH_CAMERA#SNAPPED",
          "label": "Photo Booth Camera Snapped",
          "description": "Dispatched when a photo is snapped.  Includes the imageData in the payload.",
          "payload": [{
            "name": "imageData",
            "label": "Image Data",
            "description": "The image data from the camera after the snap is completed",
            "fieldType": "string"
          }]
        },
        {
          "action": "PHOTO_BOOTH_CAMERA#AVAILABLE_CAMERAS_UPDATED",
          "label": "Photo Booth Available Cameras Updated",
          "description": "Dispatched when the available cameras are updated.",
          "payload": [{
                "name": "selectedCameraDeviceId",
                "label": "Selected Camera Device ID",
                "description": "",
                "fieldType": "string"
            },
            {
                "name": "cameras",
                "label": "Cameras",
                "description": "List of camera objects with the label, deviceId, id (same as deviceId), groupId and kind f
```

```
                    "fieldType": "json",
                    "typeMetadata": {
                       "schema": {
                          "type": "array",
                          "items": [{
                             "type": "object",
                             "properties": {
                                "deviceId": {
                                   "type": "string"
                                },
                                "kind": {
                                   "type": "string"
                                },
                                "label": {
                                   "type": "string"
                                },
                                "groupId": {
                                   "type": "string"
                                }
                             }
                          }]
                       }
                    }
                 },
                 {
                    "name": "selectedDeviceIdFound",
                    "label": "Selected Device ID was Found",
                    "description": "Whether or not the Selected Camera Device ID was found and is in use. If 'false' check Act

                    "fieldType": "boolean"
                 },
                 {
                    "name": "boundCameraDeviceId",
                    "label": "Actual Camera Device ID",
                    "description": "The Device ID of the camera that is actually being used. (May be different from the Select

                    "fieldType": "string"
                 }
              ]
           }
        ]
     }
  },
  "scopeName": "x_snc_k23_uic_pb"
}
```


Note: If running the lab on your own machine change **scopeName** to match the actual scope name.

Deploying

Now is the time to deploy it to the instance. It will deploy to the instance that you configured in Exercise 1.

Go to the terminal and execute the following command:

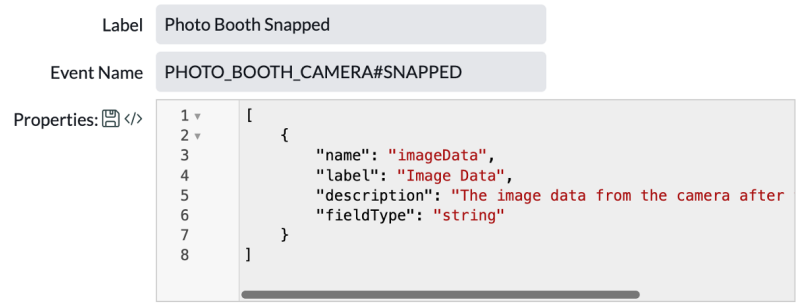
```
snc ui-component deploy --force
```

Note: “–force” is not necessary on the first deployment, but will be necessary on each subsequent deployment since the application will already exist on the instance.

Important Quirk 1 After completing deployment the tool will warn that ‘“actions” at position 0 does not match any of the allowed types’. Ignore it and proceed.

1. The tool does not do it automatically so you must create the event records in your instance.

1. Login to your instance.
2. Switch to the scope **K23 UIC Photobooth**
3. Go to the table **UX Events** at *NowExperience Events* > *UX Events* and create new records for each of the actions defined in *now-ui.json*, using the values *label*, *action* and *payload* from the “action” definition to set *Label*, *Event Name*, and *Properties* on the form.



Creating new UX Event record

1. Label: Photo Booth Camera Snapped
   - Event Name : PHOTO_BOOTH_CAMERA#SNAPPED
   - Properties

```
[{
   "name": "imageData",
   "label": "Image Data",
   "description": "The image data from the camera",
   "fieldType": "string"
}]
```

2. Label: Photo Booth Available Cameras Updated
   - Event Name: PHOTO_BOOTH_CAMERA#AVAILABLE_CAMERAS_UPDATED
   - Properties

```
[{
      "name": "selectedCameraDeviceId",
      "label": "Selected Camera Device ID",
      "description": "",
      "fieldType": "string"
   },
   {
      "name": "cameras",
      "label": "Cameras",
      "description": "List of cameras with the label, deviceId",
      "fieldType": "json",
      "typeMetadata": {
         "schema": {
            "type": "array",
            "items": [{
               "type": "object",
               "properties": {
                  "deviceId": {
                     "type": "string"
                  },
                  "kind": {
                     "type": "string"
                  },
                  "label": {
                     "type": "string"
                  },
                  "groupId": {
                     "type": "string"
                  }
               }
            }]
         }
      }
   },
   {
      "name": "selectedDeviceIdFound",
      "label": "Selected Device ID was Found",
      "description": "If the Selected Camera Device ID was found",
      "fieldType": "boolean"
   },
   {
      "name": "boundCameraDeviceId",
      "label": "Actual Camera Device ID",
      "description": "The Device ID of the camera in use.",
      "fieldType": "string"
   }
]
```


2. Assign these events to your component.

Assign events to component

1. Open the **sys_ux_macroponent** table (type **sys_ux_macroponent.list** in the navigator).
2. Find the component named **K23 Photobooth Camera** and open the record.
3. Near the bottom of the form find the **Dispatched Events** field and unlock it.
4. Find the two events created previously named **Photo Booth Camera Snapped** and **Photo Booth Available Cameras Updated** and select them and save the record.

Important Quirk 2 Whenever you redeploy the component all of the assets are deleted and recreated. However, *the reference to events are not preserved* and the macroponent must be updated each time using the procedure from the previous step.

See Appendix 3: UIC Event Fixer for a solution which you can add to your dev environment to preserve these values between deployments.

Using it from UIB

1. Open **UI Builder**.
2. Create a new page and click "+" to add a new component.
3. Find "K23 Photobooth Camera" and add it to the canvas.

Catchup

If you cannot get it running, no worries! Use the following command to catch up:

```
git checkout exercise_4 -f
```

Reference

- element.js completed
- index.js completed
- now-ui.json completed

Interested in seeing the full source code, including the core tables, workflows and the UIB workspace that was used at Knowledge23 and CreatorCon23?
https://github.com/ServiceNowNextExperience/photobooth-uic-camera

Appendix

Appendix 1: How to run lab in your own environment

Use this section to set up the Now CLI tooling.

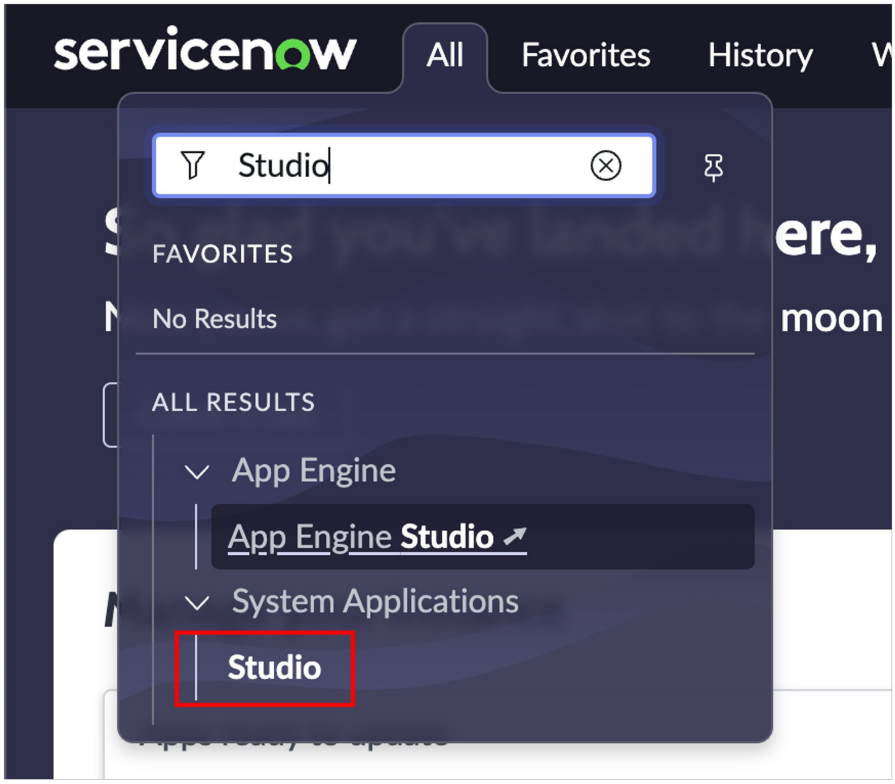
Install Command Line Tool

1. Follow the steps in how To Install the now-cli to get Now CLI running on your environment. Stop after completing step **C – Add the UI Component Extension to the SNC CLI**.
2. Go to your instance and open *System Applications* > *Studio* and create a new app to house your custom component (and *only the custom component*). Set the scope label to **K23 Camera** and make a note of the name of the scope, e.g. *x_snc_k23_uic_pb* for this lab, but it will differ in your instance.

Create Scope in Instance

Before beginning with scaffolding the project locally, login to your instance and create a new scope.

1. Navigate to *System Applications* > *Studio*.



Open Studio

2. Choose the “Create Application” button and “Let’s Get Started”.
3. Enter the App Name of “K23 UIC Photobooth”, a description and tidy up your scope name and make a note of it, e.g. “x_54321_k23_uic_pb”.

   SUPER MEGA NOTE: Wherever you see “x_snc_k23_uic_pb” referenced in this lab replace it with your actual scope name.

4. Return to **Getting Started** and continue the lab.

   NOTE: If running on your own machine with a camera attached you may skip the “Make Chrome your Default Browser” step.

Appendix 2: Tips and Tricks

1. What objects and methods are available? Once you add an event handler or method you may wonder how to tell what information is available from the method, e.g. “state” and “updateState”. The following snippet will allow you see whatever is passed in and then use the console in your browser to drill into the objects to find out what they do.

In this example four abitrary variables are mapped and the destructuring of state values is moved to a separate line to maintain functionality while testing.

```
// In this example we want to see what obects are passed to the view method
// Remove any existing parameters or destructuring and replace it with variables
const view = (a, b, c, d) => {
   console.log({ a, b, c, d });
   const {
      snapState,
      properties: {
         pauseDurationSeconds,
         animationDuration = pauseDurationSeconds + "s",
      },
   } = a;
```

1. How to use the debugger When debugging locally you can use “debugger;” commands in your code, but some people find those messy. They also do not work once deployed. Another alternative to identify locations in your code and jump to them during runtime is to add a “console.log” call and then use the browser inspector to jump into that block of code by clicking the file name.

   ```
   console.log("REQUEST SNAP");
   ```

Appendix 3: UIC Event Fixer

Due to a bug in the command line tool it does not preserve the dispatched and handle events values that were previously selected on the macroponent record during re-deployment. To resolve this issue until it is fixed install the UIC Event Fixer in your dev environment (if using a provided lab instance it is already installed).

Appendix 4: Mocking Camera Input

Not everyone will have a camera attached to their devices at all times. Those using the cloud labs instances, for instance, do not have a camera. In that case it can be mocked by starting Google Chrome from the terminal with a flag that will replace the camera with a test pattern.

–use-fake-device-for-media-stream

Mac

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --use-fake-device-for-media-stream
```

Windows

```
C:\Program^ Files\Google\Chrome\Application\chrome.exe --use-fake-device-for-media-stream
```