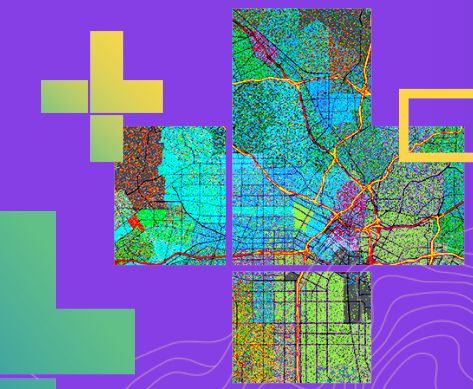
ArcGIS Experience Builder: Building Web Solutions, Advanced Topics

Yiwei Ma

David Martinez

2020 ESRI DEVELOPER SUMMIT | Palm Springs, CA



Agenda

- Short Intro
- Widget implementation
- Data source
- Messaging actions
- Debugging
- Deployment
- Widget UI Development
- Theme Customization

What is

ArcGIS Experience Builder

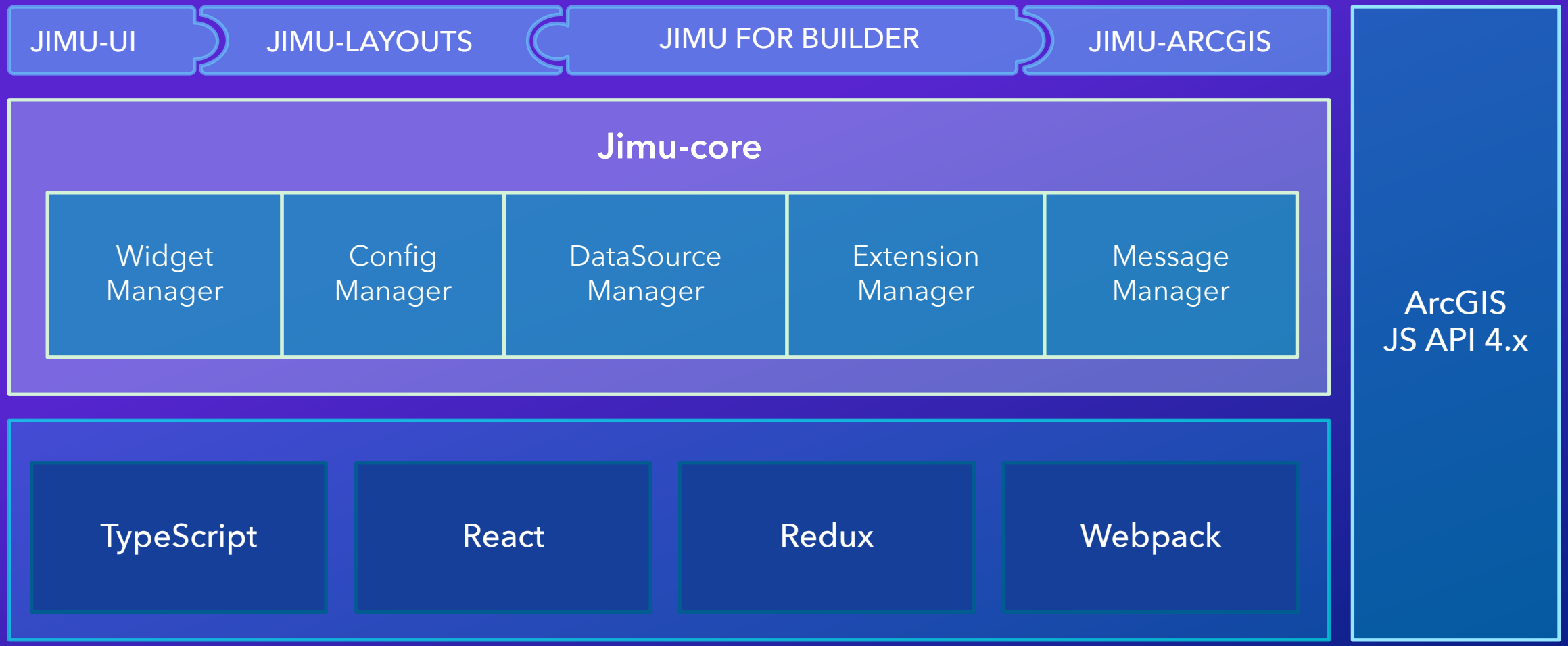
ArcGIS Experience Builder empowers you to quickly transform your data into compelling web apps.

Benefits

- Complete control over the layout and design elements
- Can easily get started with provided templates and widgets,
- Extensible to build your own

Our goal is to enable you to build compelling web experiences for your audiences.

Jimu Libs


JIMU-UI
JIMU-LAYOUTS
JIMU FOR BUILDER
JIMU-ARCGIS
Jimu-core
Widget Manager
Config Manager
DataSource Manager
Extension Manager
Message Manager
TypeScript
React
Redux
Webpack
ArcGIS JS API 4.x

Widget implementation

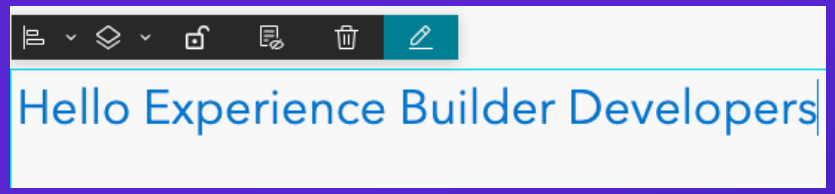
Modules in the ArcGIS API for JavaScript (JSAPI)

- Two options to use JSAPI modules:
  - Widgets dependent on the JSAPI
    - Declare jimu-arcgis dependency in the widget's manifest.json
    - Import the module you need in widget.tsx
  - Widgets dependent on JSAPI conditionally
    - use import {loadArcGISJSAPIModules} and loadArcGISJSAPIModules ([ ]) to dynamically load modules.

```
loadArcGISJSAPIModules([
  "esri/layers/FeatureLayer",
  "esri/tasks/support/Query",
  "esri/geometry/SpatialReference"
]).then(modules => {
  [this.FeatureLayer, this.Query, this.SpatialReference] = modules;
    });
```

Support inline editing

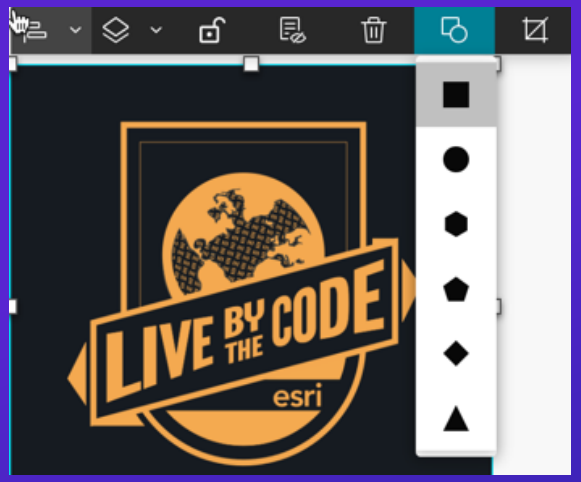
- Declare supportInlineEditing in the widget's manifest.json under the properties object.



- Declare hasEmbeddedLayout in widget's manifest.json and use layout component.
  - jimu-layouts/layout-builder
  - jimu-layouts/layout-runtime
  - Use this.props.builderSupportModules.LayoutClass

Support inline editing

- Declare CONTEXT_TOOL extensions in the widget's manifest.json



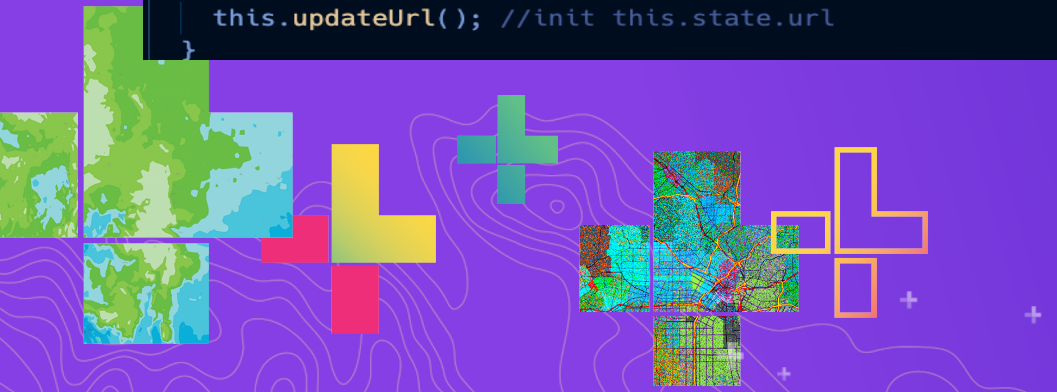
- Place modules in builder-support.tsx
  - Located in the same folder of widget.tsx
  - Available via this.props.builderSupportModules.widgetModules

```
export class Widget extends BaseWidget<AllWidgetProps<IMConfig>, State>{
  extentWatch: __esri.WatchHandle;
  btnRef: React.RefObject<any>;

  constructor(props) {
    super(props);

    this.state = {
      url: '',
      longUrl: '',
      shortUrl: '',
      uiMode: UiMode.Popup,
      isPopupOpen: !!props.isPopupOpen,
      showItem: null,
      popperRef: null
    }

    this.btnRef = React.createRef();
  }
  componentDidMount() {
    this.updateUrl(); //init this.state.url
  }
```

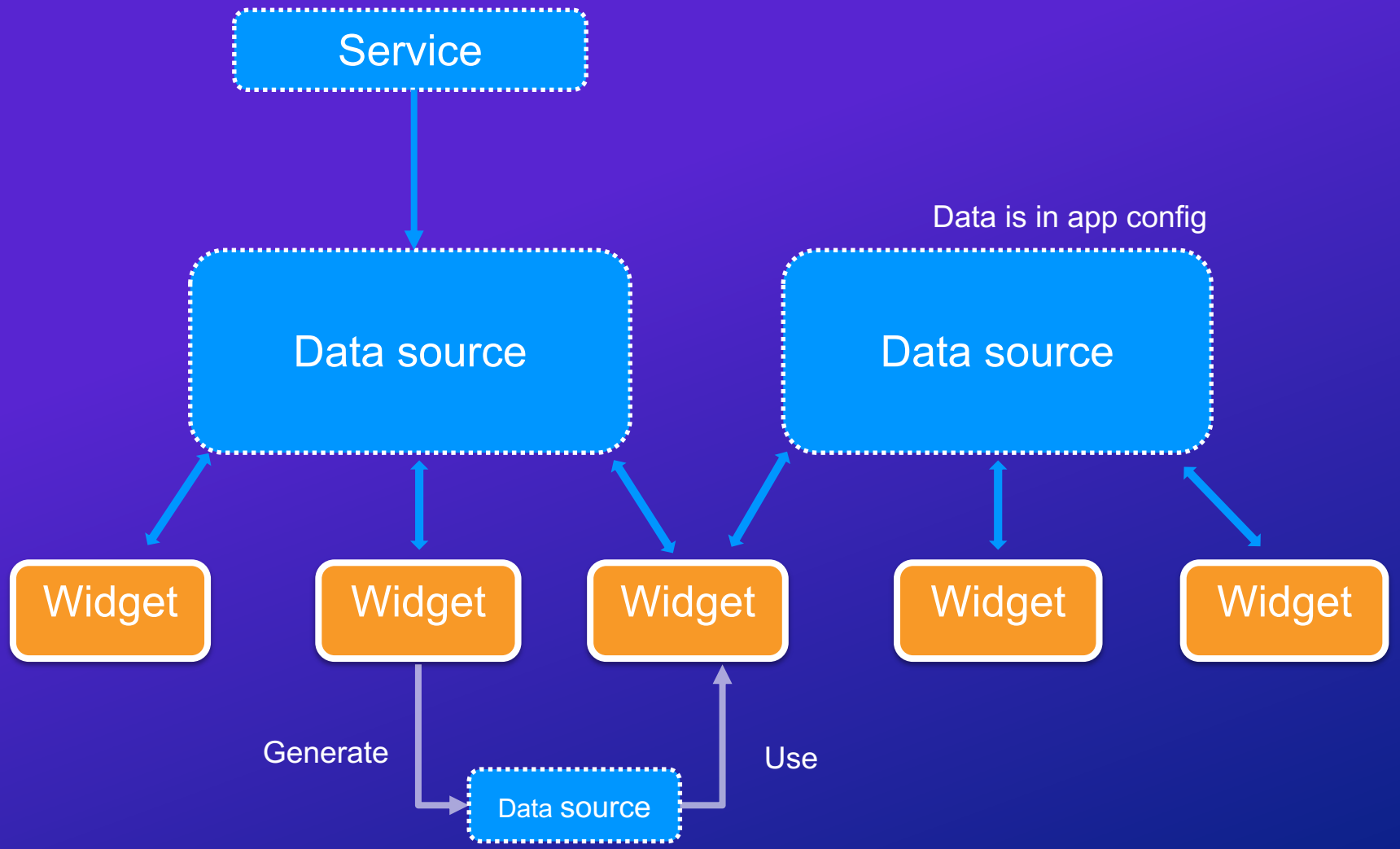


Creating widgets

Presenter(s)

Data source

Experience Builder – data source centric app



Data source

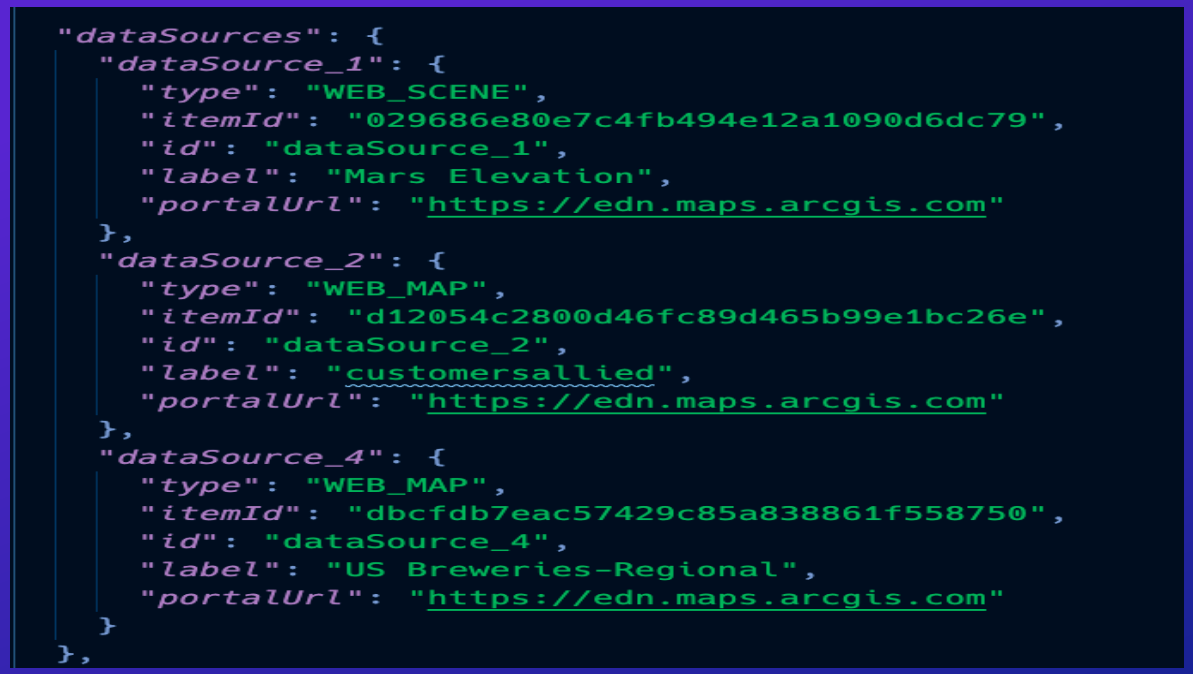
- Data source defines how your widget accesses data.
- jimu-core has a Data Source interface with common methods and properties:
  - id: the data source id
  - type: widget use this property to check type of data source
  - fetchSchema(): every data source must implement this method
  - getSchema(): method to access data source schema
  - getRecords(): method to access data records
  - getStatus: property to get the status such as loading, loaded etc.
- Custom data source can be created through DataSource interface or by extending the AbstractDataSource class
- Data source is managed by DataSourceManager

Data source main features

- Share data between widget
- Pipeline data from one widget to another
- Put the selected data in URL to create durable URL
- Change data source but does not break the app

Using data source

- Datasources is saved in dataSources property in app config
  - Use of datasource is declared with useDataSources property
- Config.json is located in server/src/public/apps folder



```
"dataSources": {
  "dataSource_1": {
    "type": "WEB_SCENE",
    "itemId": "029686e80e7c4fb494e12a1090d6dc79",
    "id": "dataSource_1",
    "label": "Mars Elevation",
    "portalUrl": "https://edn.maps.arcgis.com"
  },
  "dataSource_2": {
    "type": "WEB_MAP",
    "itemId": "d12054c2800d46fc89d465b99e1bc26e",
    "id": "dataSource_2",
    "label": "customersallied",
    "portalUrl": "https://edn.maps.arcgis.com"
  },
  "dataSource_4": {
    "type": "WEB_MAP",
    "itemId": "dbcfdb7eac57429c85a838861f558750",
    "id": "dataSource_4",
    "label": "US Breweries-Regional",
    "portalUrl": "https://edn.maps.arcgis.com"
  }
},
```

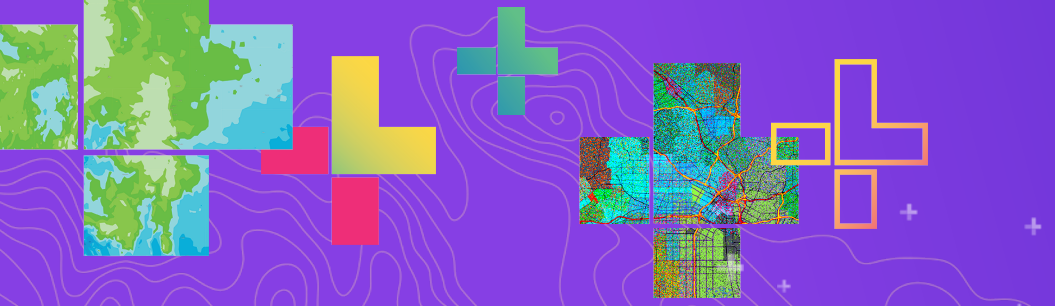
```ts
interface IStates {
  sourceData: FeatureLayerDataSource | FeatureQueryDataSource;
}

const queryTask = (ds: DataSource): any =>{
  let q = {
    where: '1=1',
    outFields: ['OBJECTID'],
    returnGeometry: false
  } as any;

  return q;
}

const queryRecords = (props: AllWidgetProps<IMConfig>, ds: DataSource): Promise<DataRecord[]> => {
  if(!ds)return Promise.resolve([]);
  if(ds.type === ArcGISDataSourceTypes.FeatureLayer){
    return (ds as FeatureLayerDataSource).queryCount(queryTask).then(() => ds.getRecords());
  }else if(ds.type === DataSourceTypes.FeatureQuery){
    return (ds as FeatureQueryDataSource).queryCount(queryTask).then(() => ds.getRecords());
  }

}
```



Data source

Presenter(s)

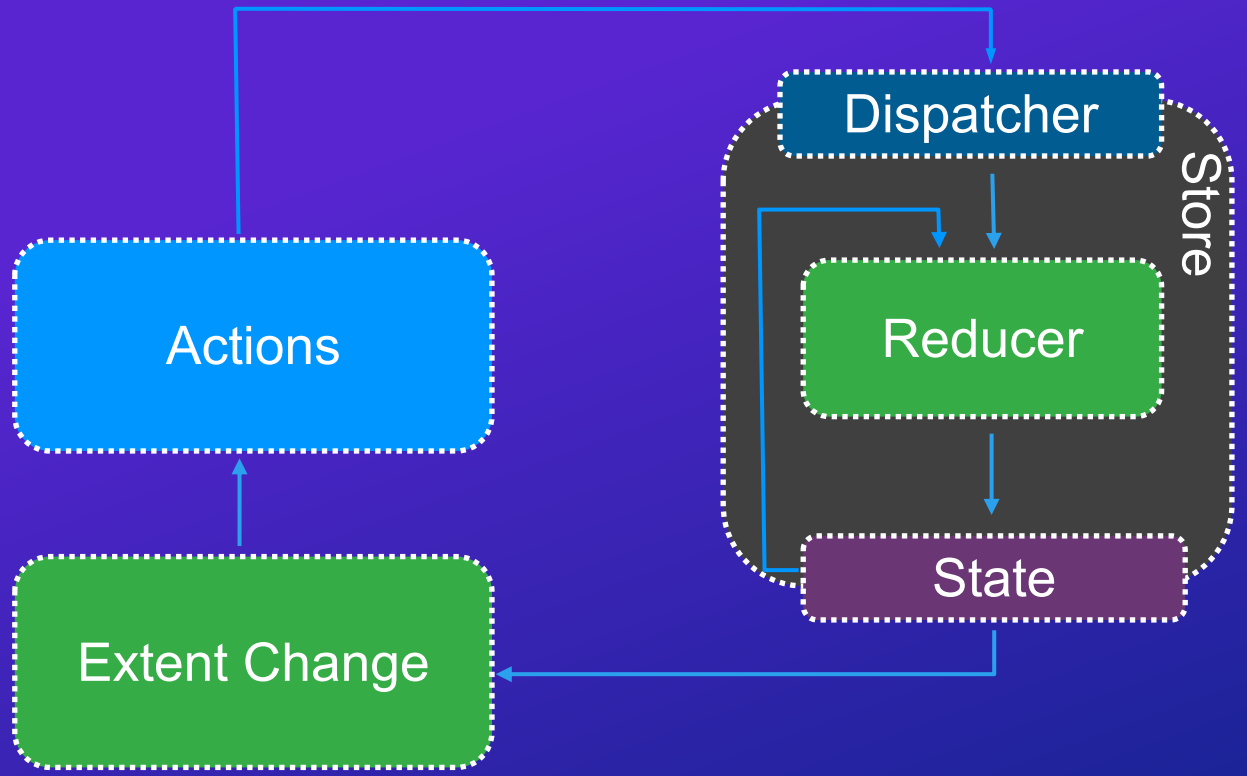
Message actions

What is a message action

Message/action is a way to support communication between widget to widget, widget to framework, and framework to widget

Redux

- A simple way to centrally manage the state of your application



Publishing a message

- A message is identified by *MessageType*, which is defined by the jimu framework.
- Use *MessageManager.getInstance().publishMessage(message)*
- Example of DataRecordsSelectionChange and ExtentChange
- Supported MessageTypes:
  - *StringSelectionChange*
  - *ExtentChange*
  - *DataRecordSelectionChange*
  - *DataRecordSetCreate*
  - *DataRecordSetUpdate*

Update manifest

```
"publishMessages": [
  "DATA_RECORDS_SELECTION_CHANGE"
]
```

Creating a message action

- Extend the *AbstractMessageAction* class

```
export default class QueryAction extends AbstractMessageAction{
  filterMessageType(messageType: MessageType, messageWidgetId?: string): boolean{
    return [MessageType.StringSelectionChange, MessageType.DataRecordsSelectionChange].indexOf(messageType) > -1;
  }
```

- *filterMessage* method is used to filter messages in message manager

```
filterMessage(message: Message): boolean{return true; }
```

- getSettingComponentUri component

```
getSettingComponentUri(messageType: MessageType, messageWidgetId?: string): string {
  return 'actions/query-action';
}
```

Creating a message action

- *OnExecute* method handles the logic for your message type

```
onExecute(message: Message, actionConfig?: any): Promise<boolean> | boolean{
  let q = `${(actionConfig as ConfigForStringChangeMessage).fieldName} = '${message}'`
  switch(message.type){
    case MessageType.StringSelectionChange:
      q = `${(actionConfig as ConfigForStringChangeMessage).fieldName} = '${(message as StringSelectionChangeMessage).str}'`
      break;
    case MessageType.DataRecordsSelectionChange:
      q = `${(actionConfig as ConfigForFeatureChangeMessage).queryFieldName} =
      '${(message as DataRecordsSelectionChangeMessage).records[0].getData()[(actionConfig as ConfigForFeatureChangeMessage).featureFieldName]}'`
      break;
  }

  getAppStore().dispatch(appActions.widgetStatePropChange(this.widgetId, 'queryString', q));
  return true;
 }
}
```

```
onExecute(message: Message, actionConfig?: any): Promise<boolean> | boolean{
  const dataRecordSetMessage: DataRecordSetCreateMessage | DataRecordSetUpdateMessage =
    message as DataRecordSetCreateMessage | DataRecordSetUpdateMessage;

  if (dataRecordSetMessage.dataRecordSet && dataRecordSetMessage.dataRecordSet.records) {
    const datasource: DataSource = dataRecordSetMessage.dataRecordSet.records[0]
      && dataRecordSetMessage.dataRecordSet.records[0].dataSource;
    if (datasource.type === DataSourceTypes.FeatureQuery) {
      return loadArcGISJSAPIModules(['esri/renderers/support/jsonUtils', 'esri/layers/support/Field', 'esri/Graphic']).then(modules => {
        let jsonUtils: __esri.jsonUtils = null;
        let Field: __esri.FieldConstructor = null;
        let Graphic: __esri.GraphicConstructor = null;
        [jsonUtils, Field, Graphic] = modules;
        const layer = {
          title: datasource.label,
          renderer: jsonUtils.fromJSON((datasource as FeatureQueryDataSource).getLayerDefinition().drawingInfo.renderer),
          fields: (datasource as FeatureQueryDataSource).getLayerDefinition().fields.map((field) => {return Field.fromJSON(field)})
        }
        return this.handleMessage(message, layer, Graphic);
      });
    } else {
      return this.handleMessage(message);
    }
  } else {
    return true;
  }
}
```

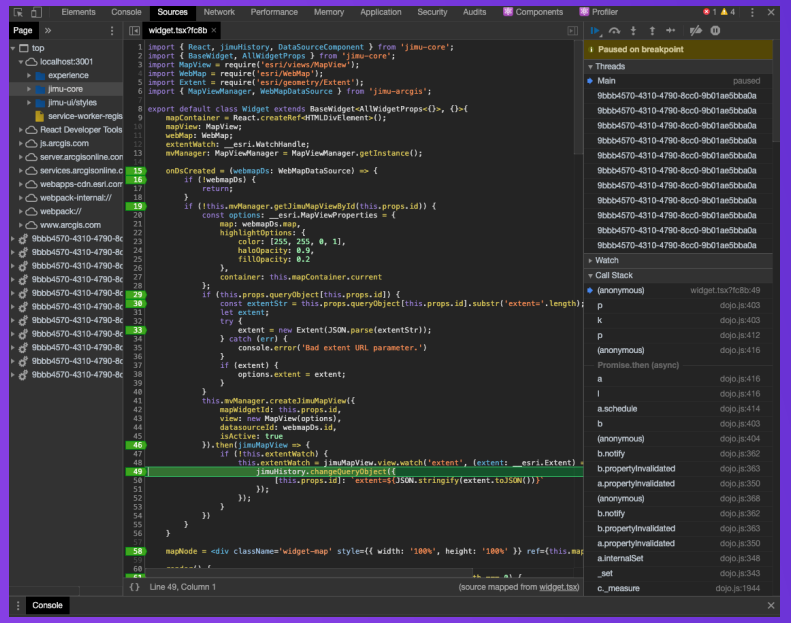
Message actions

Presenter(s)

Debugging Widgets

Setting up a development environment

1. Select any IDE or text editor: Visual Studio Code (VS code) is a good option because it has TypeScript awareness built-in by default.
2. Add the client folder to your IDE workspace
3. Recommended extensions from VS Code:
   - IntelliSense for CSS class names in HTML - CSS class name completion
   - vscode-styled-components - Syntax highlighting for styled-components



Debugging

Presenter(s)

Deployment

Widget UI Development

Three steps to define your Widget UI

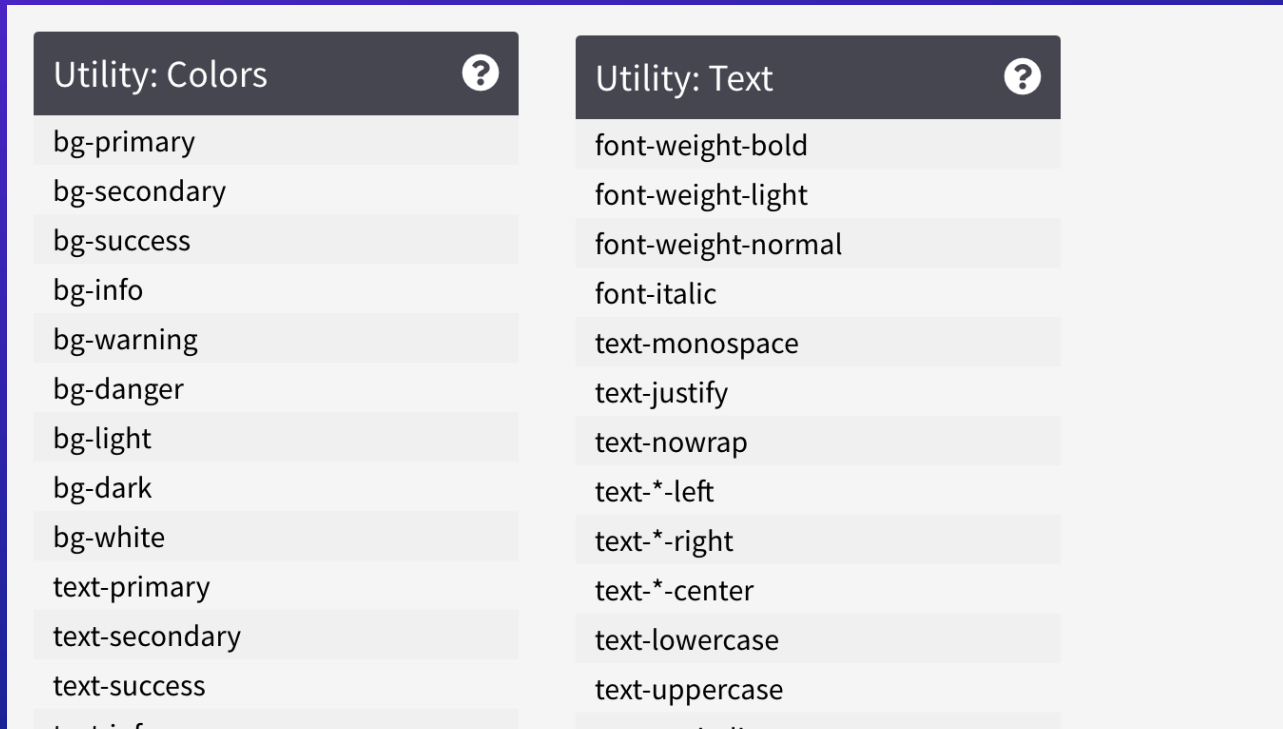
1. Create Widget HTML Template
2. Add class names
3. Add styles

1. Create Widget HTML Template:

- HTML structure is written in JSX
- The code goes into the ***render( )*** function
- Utilize Jimu UI components
  - Instead of designing your own
  - To have a consistent look-and-feel with other UI elements

2. Add CSS Classes:

- CSS class names can be added to DOM elements through the ***className*** attribute
- Utilize the CSS class utilities from Jimu UI to quickly apply a style from the theme
  - Same class names from Bootstrap can be used, such as:
    - Background color
    - Text color
    - Border
    - Padding and margin
    - Shadow
    - Etc.



3. Add CSS Styles:

Three approaches:

1. Create a separate CSS or Sass file and import it in the widget
2. Use the default ***style*** attribute from JSX
   - Supports object styles

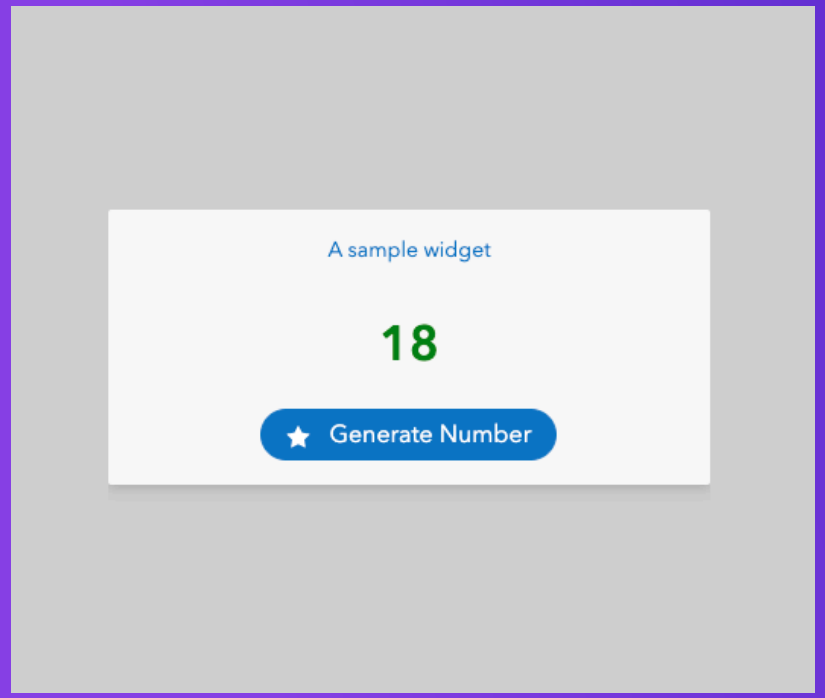
3. Use the ***css*** prop from Emotion (CSS-in-JS)
   - Supports object styles as well as string styles

   Recommended

3. Add CSS Styles: Using Theme Variables

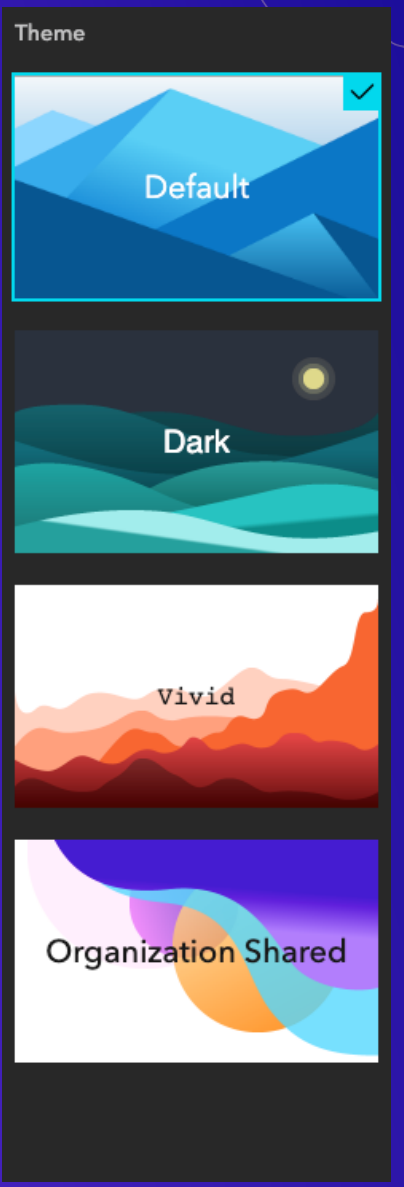
- Theme variables are available as an object in widget's props: ***props.theme***
- Reference theme variables using template literals

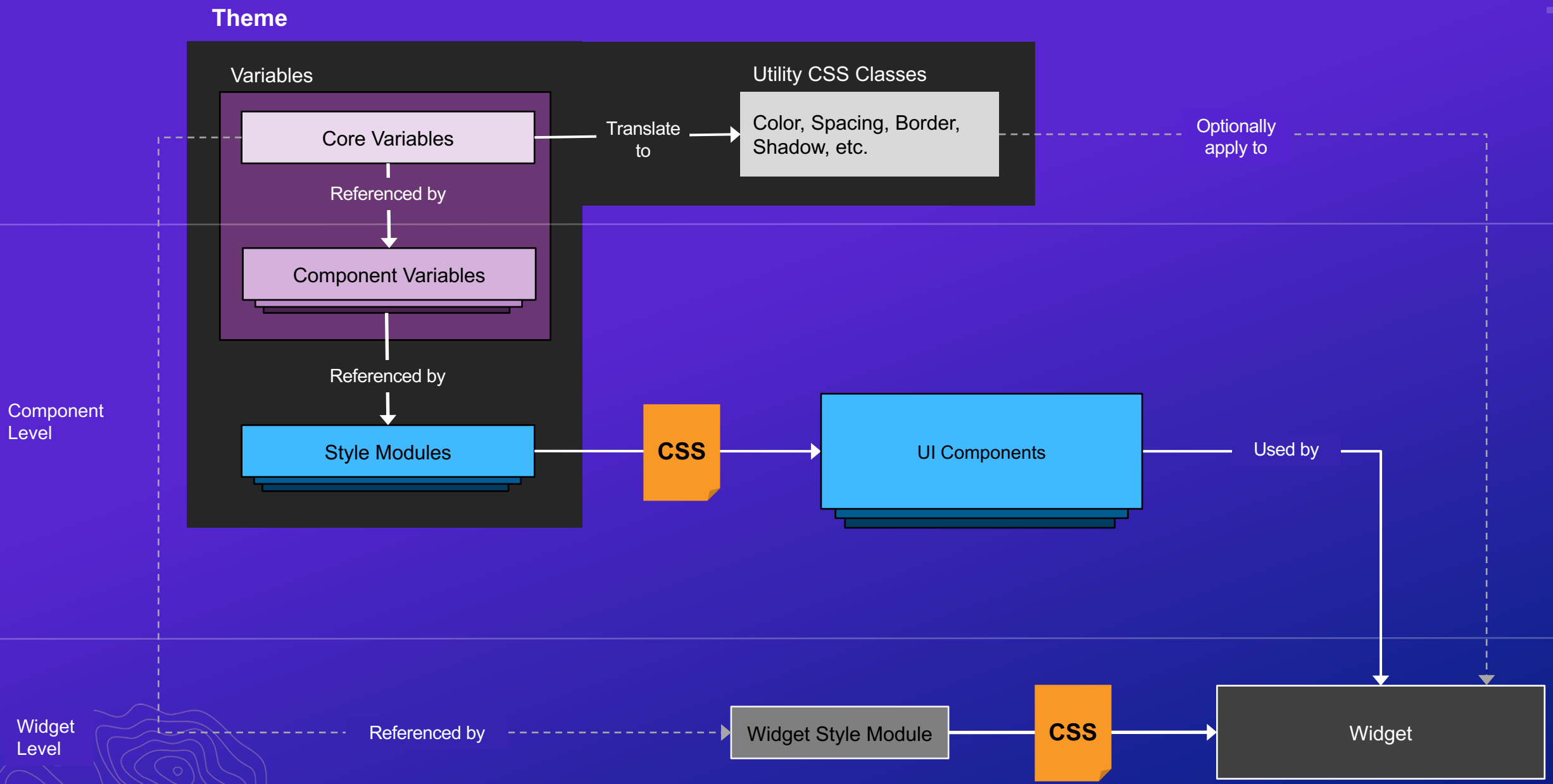
```
getStyles = function() {
  const theme = this.props.theme;
  return css`
    background: ${theme?.colors?.light};
    color: ${theme?.colors?.primary};
    padding: ${theme?.sizes?.[3]};
    box-shadow: ${theme?.boxShadows?.default};
    text-align: center;
    border-radius: ${theme?.borderRadiuses?.default};
  `
}
```



Demo: Widget UI

Theme Customization




Theme
Variables
Core Variables
Referenced by
Component Variables
Referenced by
Style Modules
Translate to
Utility CSS Classes
Color, Spacing, Border, Shadow, etc.
Optionally apply to
Component Level
CSS
UI Components
Used by
Widget Level
Referenced by
Widget Style Module
CSS
Widget

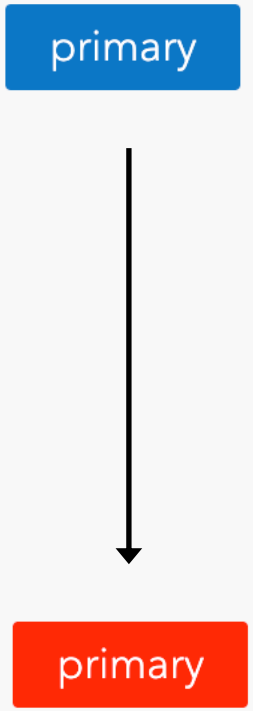
There are Two Approaches to Create A New Theme:

- Override the default theme variables
- Provide additional style modules or stylesheets

Override Theme Variables

- The ***VARIABLES*** of the default theme are defined in a JSON format
- Provide a separate JSON file with custom variables to customize the default th

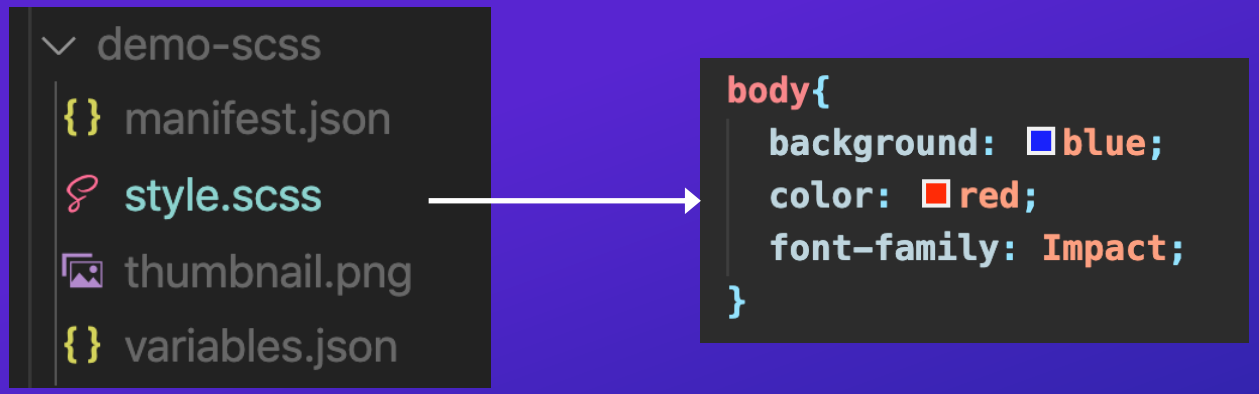
Default colors in default theme's JSON structure



Provide a Custom CSS (SCSS)

EXB supports writing styles in CSS (via SASS preprocessor):

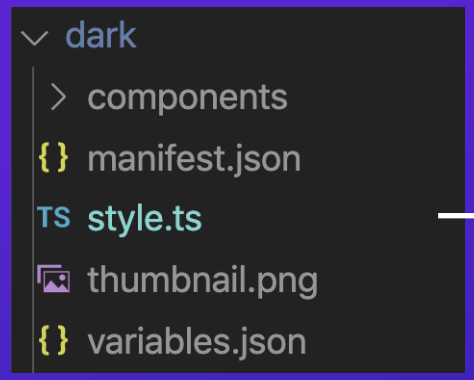
- Write your CSS in a *style.scss* file:



Provide a Custom Style Module

EXB also supports writing custom CSS module in JS:

- This will allow you to access any theme variables
- Write your CSS in a *style.ts* file:

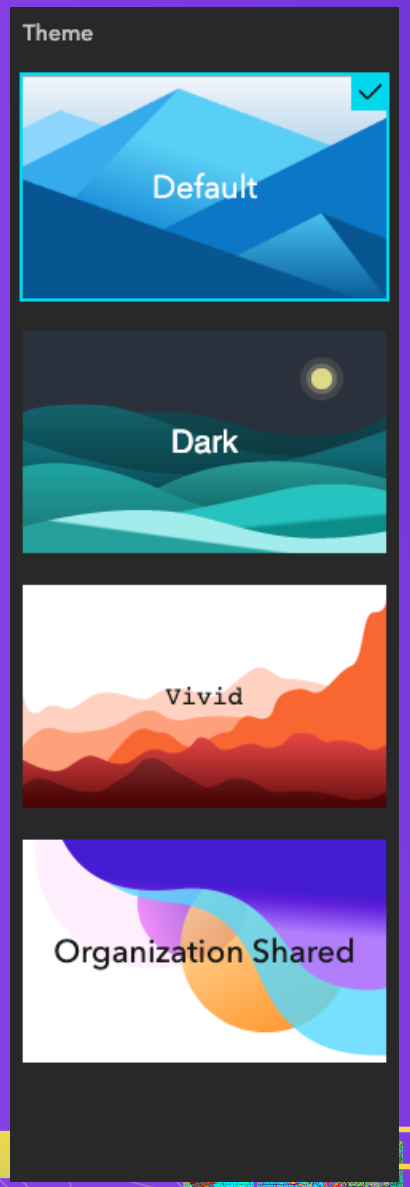



```
import { css } from 'jimu-core';

const customGlobalStyles = (props) =>
  const theme = props.theme;
  const fontFamily = 'Impact';
  return css`
    body {
      background: ${theme.colors.prima
      color: ${theme.colors.danger};
      font-family: ${fontFamily};
    }
  `
};
```

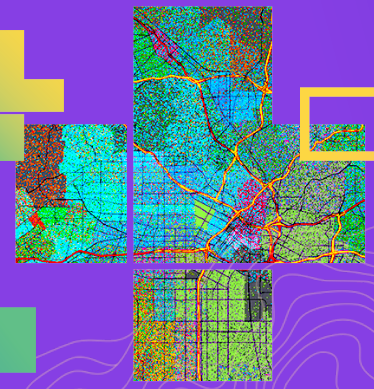
Theme A:

Theme B:



Demo:
Custom Theme

Resources

- *Experience Builder developer website*
- *Samples repo for Experience Builder*
- *React*
- *Redux*
- *Geonet community for Experience Builder*

esri | THE SCIENCE OF WHERE