Deep Dive on How ArcGIS API for JavaScript Widgets Were Built

Matt Driscoll – **@driskull**

JC Franco – **@arfncode**

Agenda

- Prerequisites
- How we got here
- Our development lifecycle
- Widget development tips
- Tools we use
- Resources
- Q & A

Prereqs: Accessor

- **Accessor SDK** *esri/core/Accessor*
- **Building Classes Using Accessor and the ArcGIS API for JavaScript**

Prereqs: TypeScript

- Leverage ES6 (syntactic sugar)
- Interfaces
- Typing
- `const` and `let` vs `var`
- `() => {}` vs `bind` or this-binding utility
- **TypeScript Setup**
- **Using TypeScript with ArcGIS API for Javascript**

How we got here

- 3.x
  - Dojo Dijit
  - Dijit Themes
  - Logic tied to UI
- 4.x
  - Abstracted & framework independent
  - ViewModels

Why?

- Framework independent
- Easily customizable themes
- Responsive design
- Redesigned Widget API
  - Consistent with core API

Our development lifecycle

How do we go about developing widgets?

Development lifecycle

- API design
- Kickoff UI/UX design
- Develop ViewModel
- Develop View
- Write tests
- Pull request
- API merge!!!

Development lifecycle: API Design

- Widget developer writes objective for widget
  - **Sample**
- Widget dev defines API in markdown
  - View & ViewModel
    - Properties
    - Methods
    - Events
  - Sample code snippets
  - Demos
  - Q & A
- API reviewed and tweaked
- JS doc written and approved

Development lifecycle: ViewModel

- Friendly, consistent naming
- Public methods
  - Return types
  - Arguments
- Public properties
- Make sure no view/UI logic

Development lifecycle: View

- Research DOM structure needed for widget
  - Layout containers needed
  - Using proper semantic tags for nodes
- CSS lookup object used in `render()`
- Accessible, Aria roles present if necessary
- Properties, events, methods aliased as necessary
- Make sure no API logic

Development lifecycle: Styles

- Classes needed
- BEM naming of classes
- 4x Widgets can use flexbox for layout

Development lifecycle: UI/UX Design

- Meeting with our creative lab
- Discuss needs, API
- Collaborate on design and tweak markup as necessary
- Receive mockup/wireframes/assets/Sass
- Implement design

Development lifecycle: Tests

- Make sure we have tests that hit all the API
- Unit, integration, functional, screenshot tests
- Methods are tested with all options and return types
- Assert properties behave as expected when modified
- Screenshot tests
- Test early

Development lifecycle: Pull Request

- All the code changes done in a git branch
- PR is opened with all changes and tests included
- PR is reviewed and tested
- API build is successful
- Merge!

Development tips

Development tips: API Design

How things can be done differently in 4 compared to 3

- Leverage
  - Collection
  - Accessor
- View properties instead of events
  - Read-only properties
- Promises for async operations
- Support modules
  - Offloading logic where appropriate. More modular.

Development tips: *render()*

- Hiding nodes

```
render() {
  const childNode = this.childVisible?

  return (
    <div>{childNode}</div>
  );
}
```

Development tips: *render()*

- Reusing classes

```
const CSS = {
  root: "example",
  part: "example__part",
  disabled: "example--disabled"
};
```

Development tips: *render()*

- Toggling classes

```
render() {
  const dynamicClasses = {
    [CSS.disabled]: this.isDisabled
  };

  return (
    <div classes={dynamicClasses}>...<
  );
}
```

Development tips: `render()`

- `class` cannot change between renders

```
render() {
  const rootClass = someCondition ? CS

  // throws error - cannot change clas
  return (
    <div class={rootClass}>...</div>
  );
}
```

Development tips: *render()*

- Use *join* to apply multiple classes

```
render() {
  return (
    <div class={join(CSS.root, CSS.but
  );
}
```

Development tips: *render()*

- Toggle styles (similar to *classes*)

```
render() {
  const dynamicStyles = {
    "x": getX(),
    "y": getY()
  };

  return (
    <div styles={dynamicStyles}>...</d
  );
}
```

Development tips: *render()*

- Attributes are not removed between render calls

```
render() {
  const tabIndex = this.focusable ? 0

  // `tabIndex` attribute will get rer
  return (
    <div tabIndex={tabIndex}>...</div>
  );
}
```

Development tips: *render()*

- String templates!

```
render() {
  return (
    <div class={CSS.root}>`Hello, ${th
  );
}
```

Development tips: *render()*

- Distinguishable children

```
render() {

  // children are NOT dynamically adde
  return (
    <div>
      <div>foo</div>
      <div>bar</div>
    </div>
  );
}
```

Development tips: *render()*

- Distinguishable children

```
render() {
  const foo = this.showFoo? <div key='
  const bar = this.showBar? <div key='

  // children are dynamically added/re
  return (
    <div>
      {foo}
      {bar}
    </div>
  );
}
```

Note: *key* can be a string, number or object

Development tips: *render()*

- Storing data on attributes

```
render() {
  return (
    <div onclick={this._handleClick}
         data-lucky-numbers={luckyNumb
  );
}

private _handleClick(event: MouseEvent
  const node = event.currentTarget as
  const luckyNums = node.getAttribute(
  console.log(`Today's lucky numbers a
}
```

Development tips: *render()*

- Binding

```
render() {
  return (
    <div class={CSS.base}>
      <div onclick={this._logThis}>thi

      <div bind={this}
           onclick={this._logThis}>thi
    </div>
  );
}
```

Development tips: *render()*

- DOM events

```
render() {
  return (
    <div class={CSS.base}>
      <img onclick={this._handleClickE
    </div>
  );
}

private _handleClickEvent(event: Mouse
  // do something with event
}
```

Development tips: *render()*

- Real nodes
  - Markup in *render()* is virtual
  - Need to store reference to actual node with *afterCreate* or *afterUpdate*

```
private _realNode: Element = null;

render() {
  return (
    <div afterCreate={this._storeThisN
  )
}

private _storeThisNode(node: Element):
  this._realNode = node;
}
```

Development tips: *render()*

- accessibleHandler

```
render() {
  return (
    <div onclick={this._doSomething}
         onkeydown={this._doSomething}
  );
}

@accessibleHandler()
private _doSomething(): void {
  // ...
}
```

Development tips: *render()*

- Close childless node tags for conciseness

```
render() {
  return (
    <div>
      <div class={CSS.childless} />
    </div>
  );
}
```

Development tips: *render()*

- Keep *render* manageable by extracting pieces as it grows

```
render() {
  return <div>{this._renderContent()}<
}

private _renderContent(): any {
  return (
    <div>
      <h1>{this.title}</h1>
      {this._renderItems()}
    </div>
  );
}
```

Development tips: ViewModels

- Rethinking APIs
  - More collections
  - More Accessors
  - View properties instead of events
  - Read-only properties
- Support modules
  - Offloading logic where appropriate. More modular
- Autocasting

Development tips: Styling

- CSS to Sass
  - Variables
  - Theming
  - Mixins
- **SDK Guide: Styles**
- **Sass**
- **BEM**
- **Icon font** / SVG

Widget Theming: Sass

- CSS preprocessor
- Variables
- `@mixin` (group statements)
- `@include` - (use mixins)
- `@import` - (split up files)
- `@extend` - (inheritance)
- More power!

Sass Install

- **Installing grunt-sass**

```
$ npm install --save-dev grunt-sass
```

Widget BEM

- **BEM**: Block Element Modifier
- Methodology to create reusable components
- Uses delimiters to separate block, element, modifiers
- Provides semantics (albeit verbose)
- Keeps specificity low
- Scopes styles to blocks

```
/* block */
.example-widget {}

/* block__element */
.example-widget__input {}
.example-widget__submit {}

/* block--modifier */
.example-widget--loading {}

/* block__element--modifier */
.example-widget__submit--disabled {}
```

Development tips: Styling within View

CSS lookup object

```
const CSS = {
  base: "my-widget",
  title: "my-widget__title"
};
```

Lookup referenced in JSX

```
<div class={CSS.base}/>
  <h1 class={CSS.title}>Hello world</h
</div>
```

Our Tools

- IDEs
- Tasks
- Testing
- Other

Tools: IDEs

- Multiple flavors
  - Visual Studio Code
  - WebStorm
  - **and more...**
- Plugins

Tools: Tasks

- Node/npm
- Installing Grunt
- Compile TS/Sass

Testing

- **Intern**
- **Jasmine**
- **QUnit**
- **Karma (test runner)**

Other tools

Besides an IDE and browser dev tools…

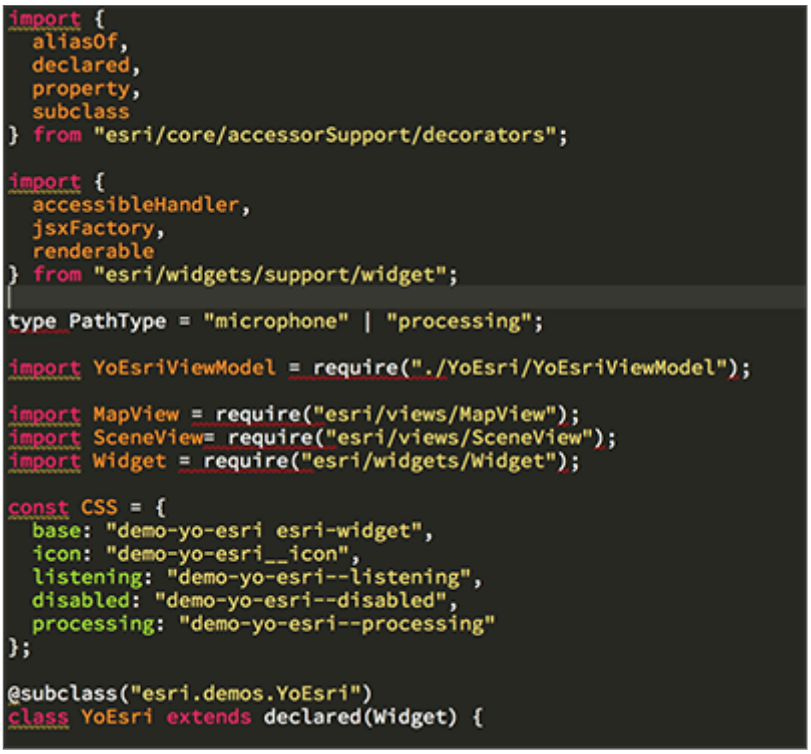
- SourceTree
- Terminal
- GitHub Enterprise
- Slack :D
- A handful of browsers
- Coffee

Additional Resources

- **4x widget snippets**
- **JavaScript Sessions at DevSummit**
- **Documentation - 4.3**

Find this on GitHub

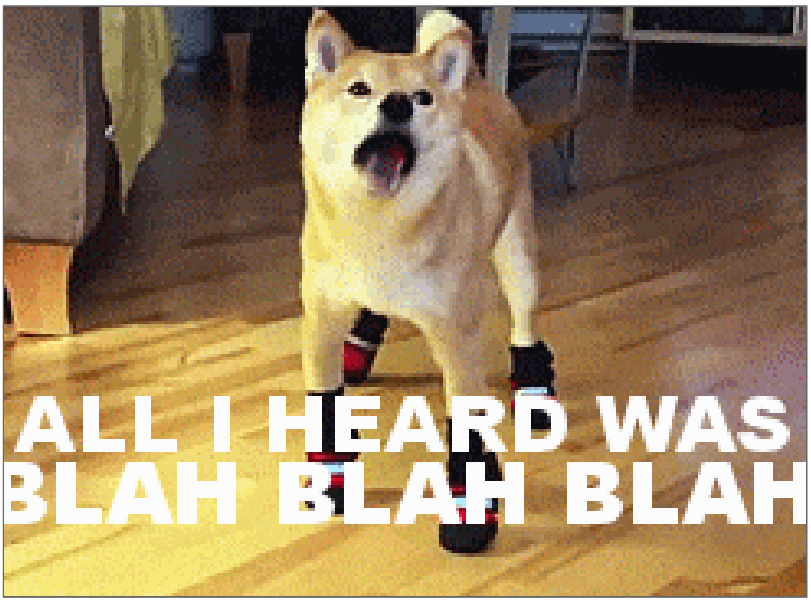
GitHub Code



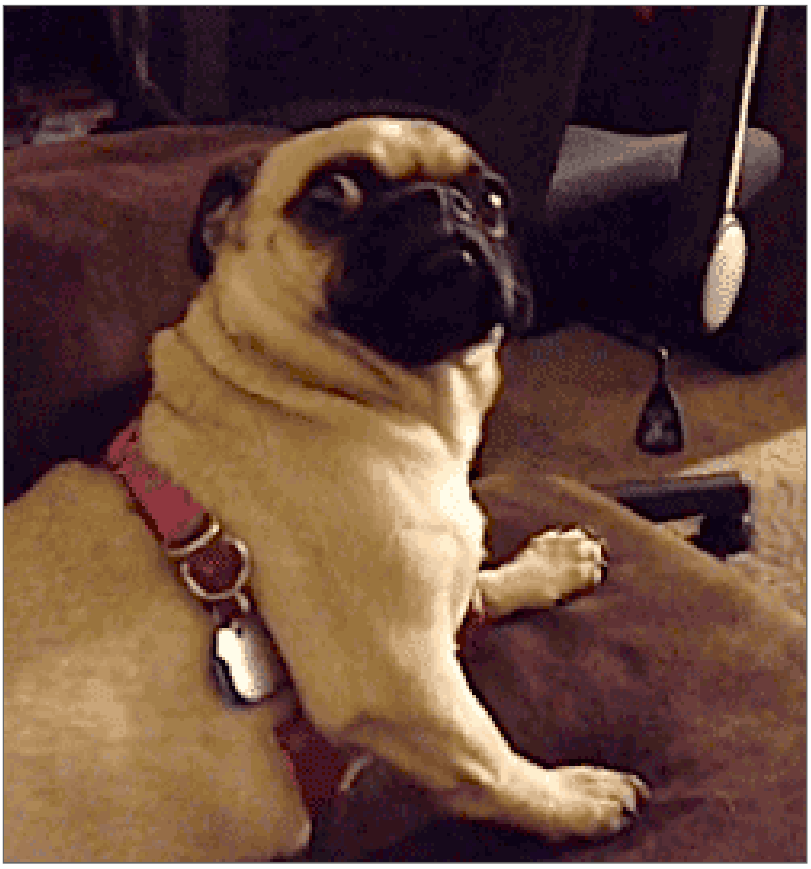
```
import {
  aliasOf,
  declared,
  property,
  subclass
} from "esri/core/accessorSupport/decorators";

import {
  accessibleHandler,
  jsxFactory,
  renderable
} from "esri/widgets/support/widget";

type PathType = "microphone" | "processing";

import YoEsriViewModel = require("./YoEsri/YoEsriViewModel");

import MapView = require("esri/views/MapView");
import SceneView= require("esri/views/SceneView");
import Widget = require("esri/widgets/Widget");

const CSS = {
  base: "demo-yo-esri esri-widget",
  icon: "demo-yo-esri__icon",
  listening: "demo-yo-esri--listening",
  disabled: "demo-yo-esri--disabled",
  processing: "demo-yo-esri--processing"
};

@subclass("esri.demos.YoEsri")
class YoEsri extends declared(Widget) {
```

Please Take Our Survey!

1. Download the Esri Events app and go to DevSummit
2. Select the session you attended
3. Scroll down to the "Feedback" section
4. Complete Answers, add a Comment, and Select "Submit"



Questions?




THANK YOU