APACHE SLING & FRIENDS TECH MEETUP
10-12 SEPTEMBER 2018

Migrating a large AEM project to Touch UI

António Ribeiro & Gregor Zurowski (Mercedes-Benz.io)

Introduction

adaptTo()

About Us

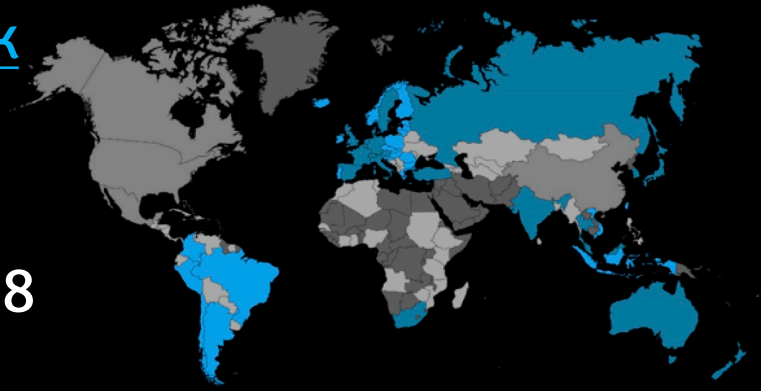
GREGOR ZUROWSKI
Germany

.io

ANTÓNIO RIBEIRO
Portugal

About Our Project

- Global CMS for all Mercedes-Benz product sites
  - https://www.mercedes-benz.co.uk
  - https://www.mercedes-benz.de
  - https://www.mercedes-benz.pt
  - + 53 other countries by end of 2018
- Project inception in 2014
  - Started on AEM 6.0
  - Currently running on AEM 6.3 and upgrading to 6.4 shortly



adaptTo()

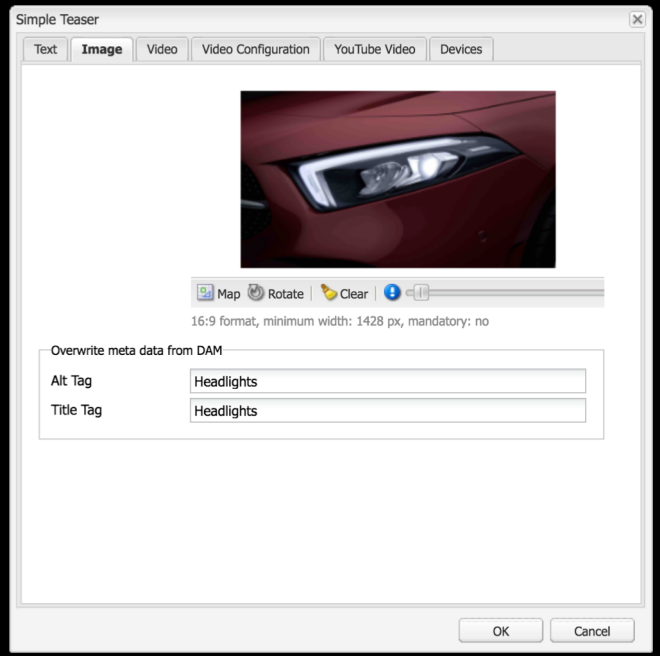
Touch UI Migration

Touch UI Migration Scope

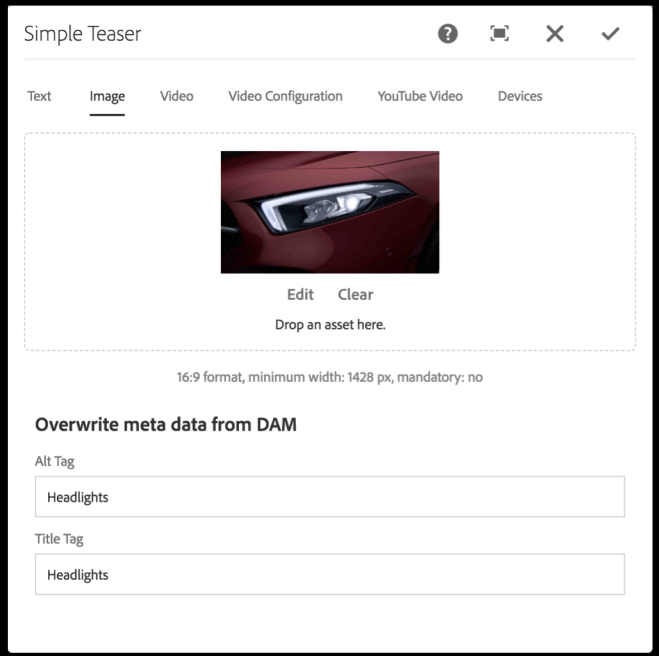
- Started migrating to Touch UI in early 2018
- Project scope
  - 80+ page components
  - 140+ components
  - 50+ AEM customizations
    - Custom widgets
    - Overlays
    - SiteAdmin/DAMAdmin/Sidekick extensions
- Presentation scope: dialogs

How is Touch UI different? (1/2)

Classic UI



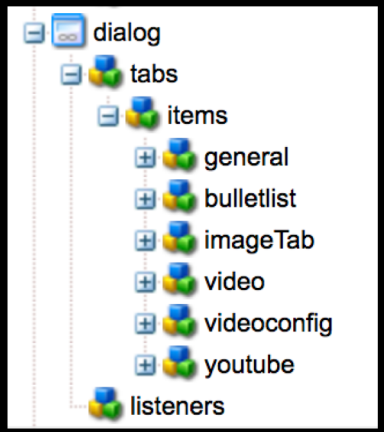
Touch UI



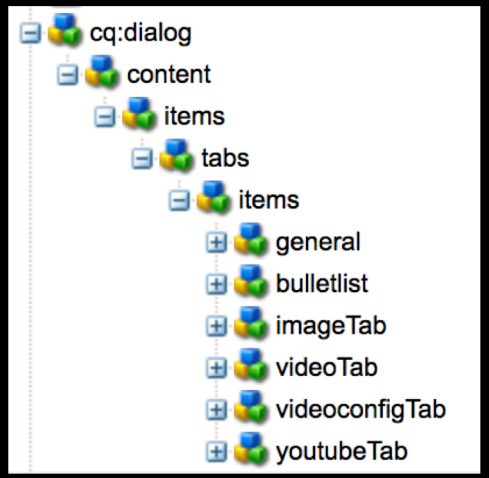
How is Touch UI different? (2/2)

- UI interfaces in AEM, including components and dialogs, are declaratively described by nodes and properties
- Similar structure with different node types and different wrapper nodes

Classic UI



Touch UI

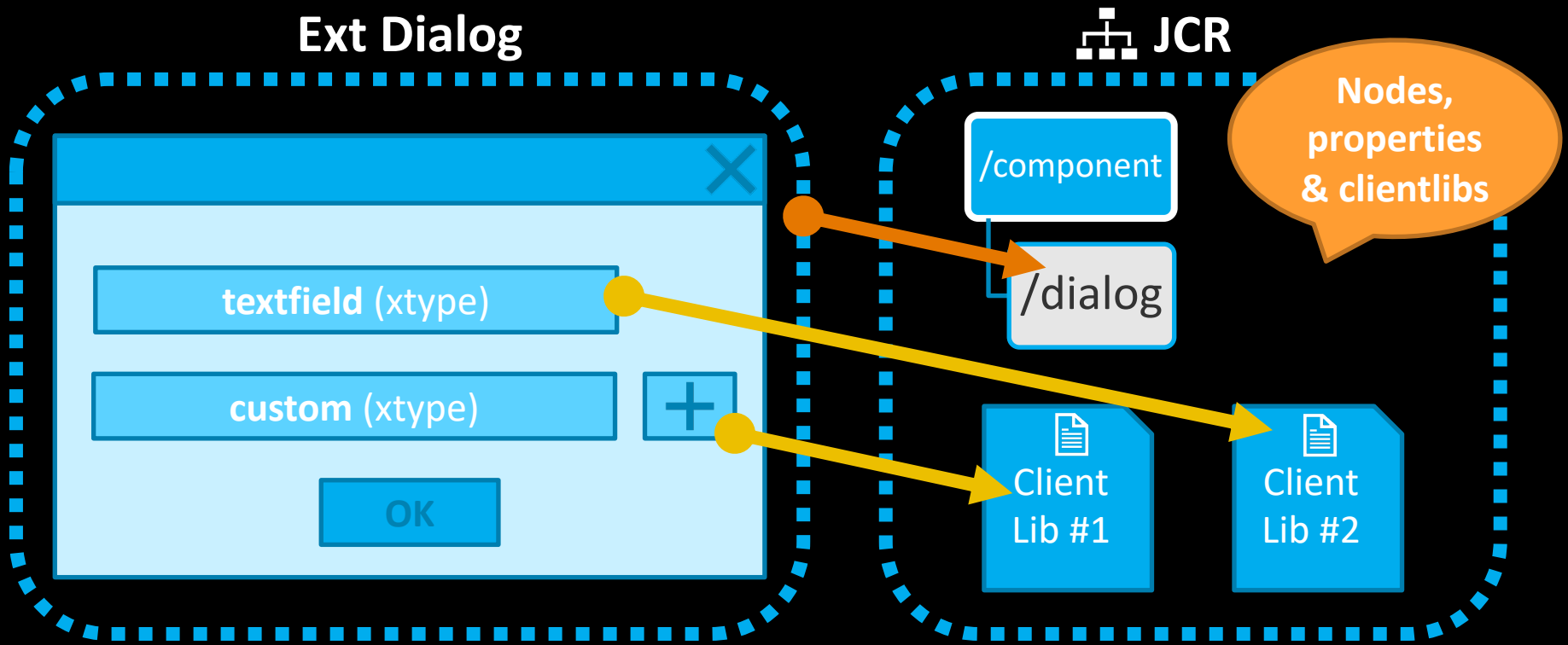


adaptTo()

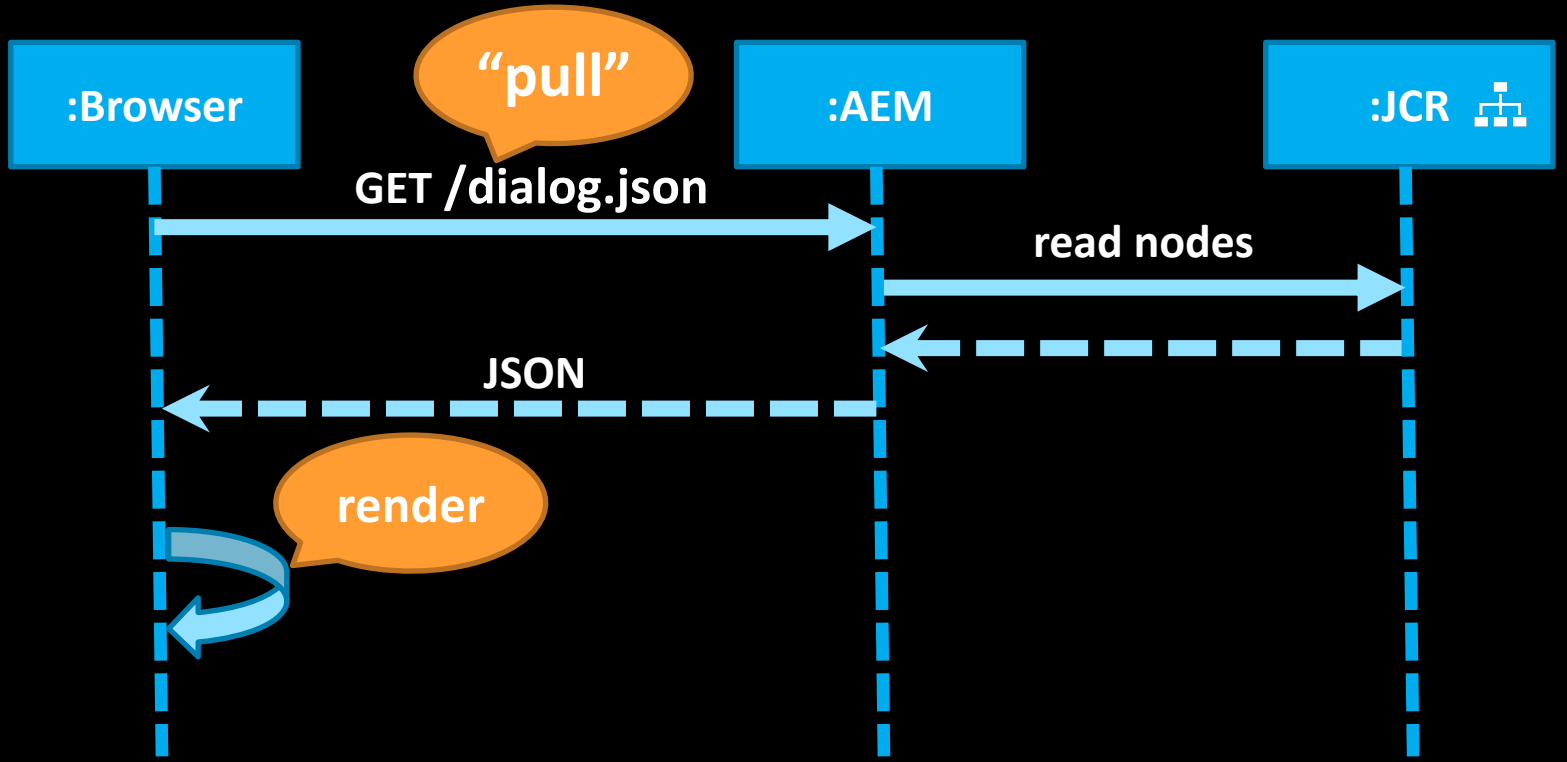
Classic UI Architecture (1/3)

- UI components are only partly defined in the repository
  - *xtypes* are defined in JavaScript sources
- Client requests component definition as JSON from server (“pull”)
- Client is responsible for dynamically creating UI components

Classic UI Architecture (2/3)



Classic UI Architecture (3/3)

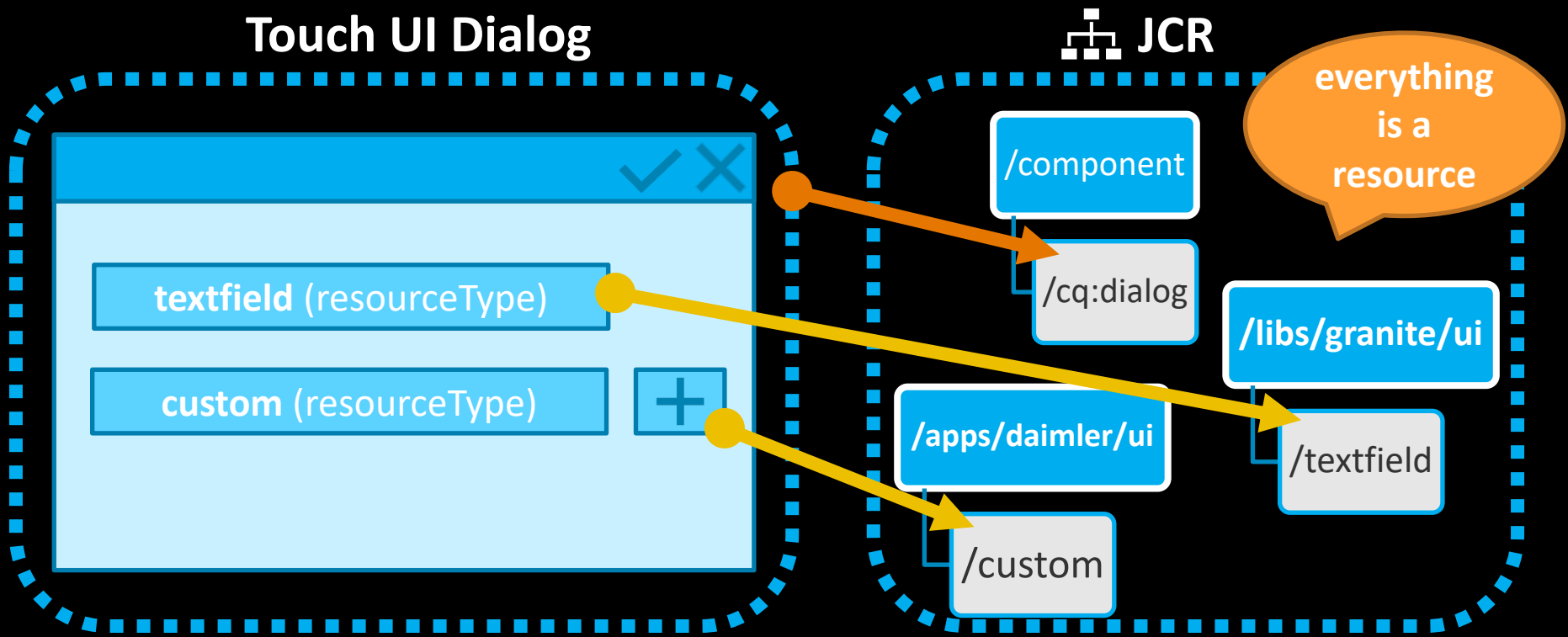


adaptTo()

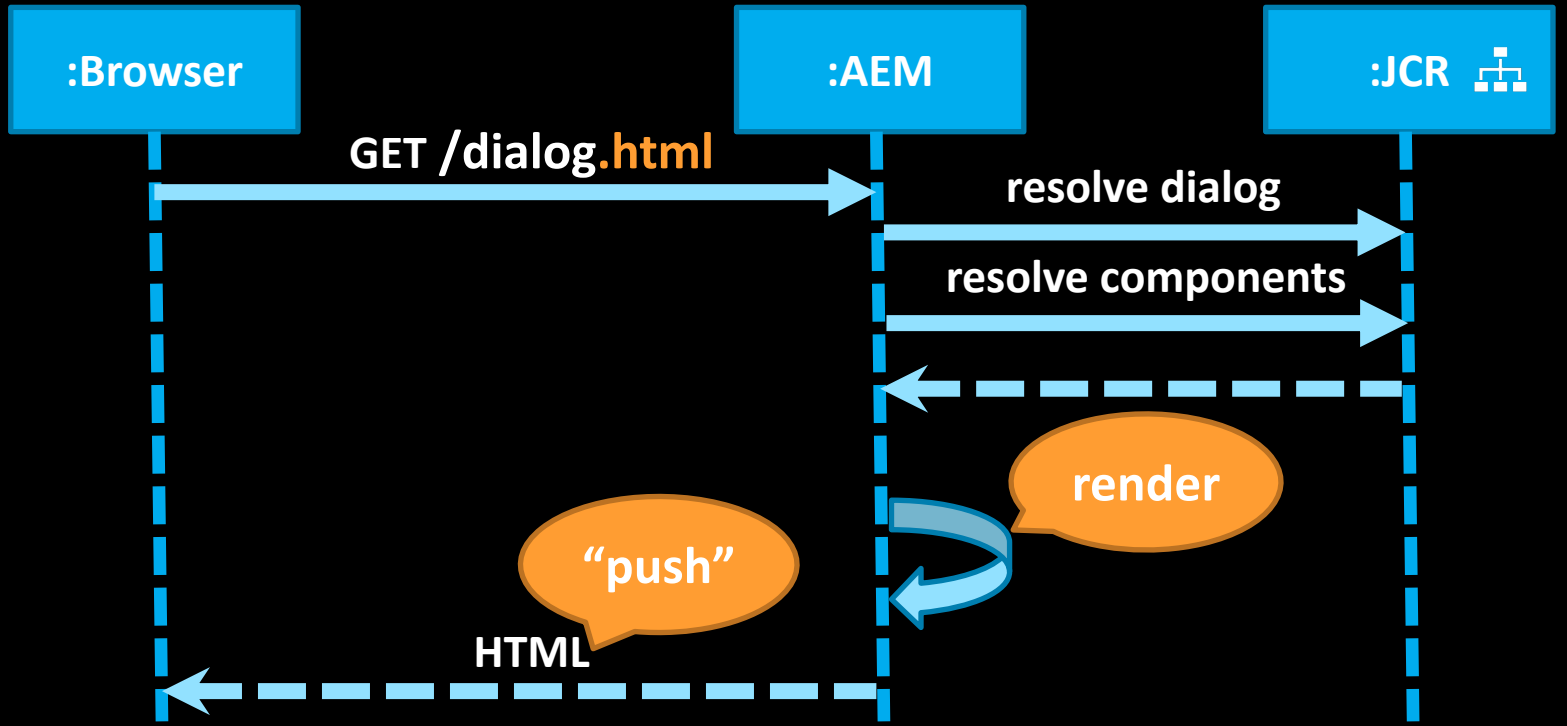
Touch UI Architecture (1/3)

- All Touch UI components defined in the repository
- Every component is a Sling resource
- Client requests page with UI
- Server sends UI as HTML documents using Coral UI HTML5 web components (“push”)
- “Dumb” client with fewer responsibilities

Touch UI Architecture (2/3)



Touch UI Architecture (3/3)

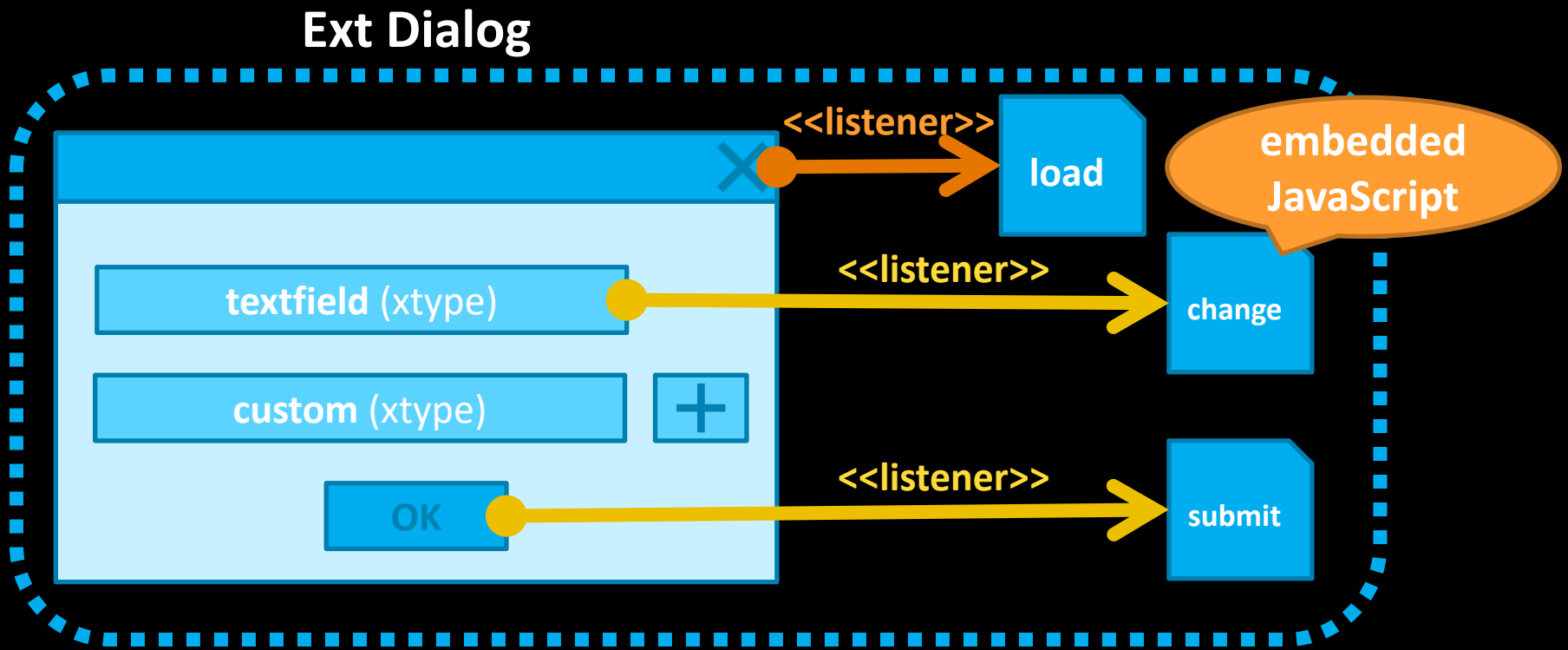


adaptTo()

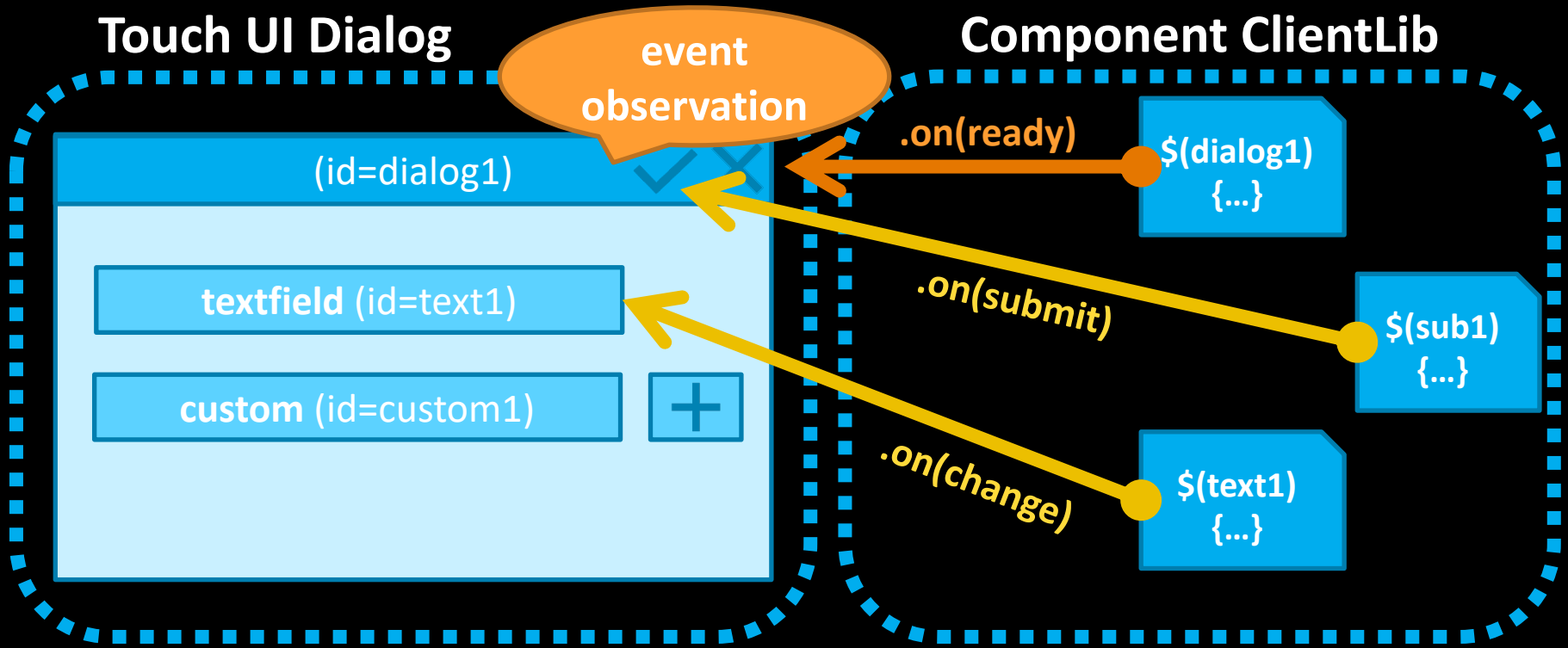
Summary of Differences

	Classic UI	Touch UI
Dialog nodes	▪ name: *dialog* ▪ jcr:primaryType*: cq:Dialog*	▪ name: *cq:dialog* ▪ jcr:primaryType: *nt:unstructured*
Sling resources	*sling:resourceType* not used	*sling:resourceType* cq/gui/components/authoring/dialog
JavaScript location	Imperative parts are directly embedded using *listeners* or managed in clientlibs.	Imperative parts cannot be embedded in dialog definition (separation of responsibilities)
Event handling	Dialog widgets directly reference JS code	JS observes dialog events

Classic UI Event Handling



Touch UI Event Handling



adaptTo()

Touch UI Event Handling Implementation (1/2)

- Need to create hook-in points for observation
- Uniquely identify components with *granite:id* that translate into HTML ID attributes
- Apply common behavior to a set of components with *granite:class* that translate into HTML class attributes

Touch UI Event Handling Implementation (2/2)

HTML

granite:id

```
<coral-fileupload
  id="touchui-dialog-imagetab-image"
  class="coral-Form-field cq-FileUpload …"
  name="./image/file" […]>
  […]
</coral-fileupload>
```

granite:class

adaptTo()

Validation (1/3)

- Validation in Classic UI and Touch UI is purely done on the client side
- Classic UI *vtypes* can no longer be used
- Touch UI provides form validation with Granite UI foundation-validation

Validation (2/3)

Dialog

```
<component1 … validation="uppercase">
```

JavaScript

```
var registry = $(window).adaptTo('foundation-registry');

registry.register('foundation.validation.validator', {
    selector: '[data-foundation-validation*=uppercase]',
    validate: function (element) {
        return checkUpperCase($(element));
    }
});
```

Validation (3/3)

- Possibility to simply combine multiple validations per component in Touch UI

```
<textfield1 …
    validation="uppercase,maxlength">
```

- In Classic UI only one vtype per widget

```
<textfield2 …
    vtype=”mycomponent-composite-validation">
```

- Simple mandatory field validation using *required* attribute, but issues with some components

```
<textfield3 …
    required="{Boolean}true">
```

Custom Component Properties (1/2)

- Components need additional properties that are not available in the default configuration
- Use *granite:data* properties that get translated into HTML data attributes
- Use case example:
  - Maximum number of allowed elements within a multifield component

Custom Component Properties (2/2)

Dialog

```
<multifield1 …
  <granite:data
    jcr:primaryType="nt:unstructured"
    max-items="{Long}4"/>
</multifield1>
```

HTML

```
<coral-multifield … data-max-items="4">
```

JavaScript

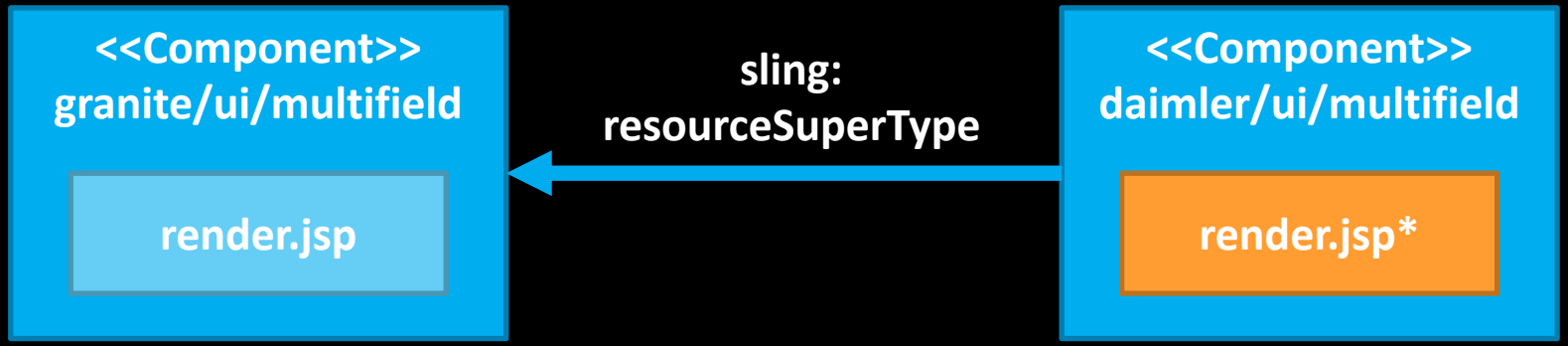
```
var maxItems = $multifield1.data('max-items');
```

Customizations

- Two approaches for customizing UI components
- Overlaying (“replacing”/ “hiding”)
  - Works in Classic UI and Touch UI
  - Place overlays in corresponding /apps component path taking precedence over /libs
  - Be aware of “sustainable upgrades” introduced with AEM 6.4
- Overriding (“inheriting”)
  - Classic UI: Widget inheritance using *CQ.Ext.extend()*
  - Touch UI: Sling Resource Merger using *sling:resourceSuperType*
  - Parent component remains as-is
- Need to sync changes coming from newer versions and patches

Customization Example Using Overrides

Use case: Create multifield component using an alternative persistence format based on the existing Granite *multifield* component.



Touch UI Migration Approach

adaptTo()

Migration Approach – User Interfaces

- Both Classic UI and Touch UI interfaces coexist
  - Introduce changes in stages
  - Test migration before final switch over
  - Extending testing period without affecting users
  - No parallel integration branches
    - No painful branch tracking/merging
  - Allows multi-staged user training in our international context

Migration Approach – Data Structures

- Existing content data structure remains untouched
  - No “big bang” data migration
  - Needs some extra tweaks to accommodate this requirement
    - Example: multifield that supports “JSON” structure

Migration Approach – JavaScript Code

- Touch UI and Classic UI JavaScript code co-exist
  - Manage each in separate clientlibs
  - Prevent colliding behavior
    - Classic UI code must not assume it’s running in Classic exclusively
    - Potentially runs in Touch UI Classic dialog fallback mode
    - Add safe-guards in existing code to prevent from running in wrong mode

adaptTo()

Recommendations & Best Practices

Recommendations & Best Practices (1/2)

- Start with conversion of simple component dialogs
- Gradually convert more complex components
  - Event handling
  - Validations
  - Custom component-specific clientlibs
- Convert dialogs with the AEM Dialog Conversion Tool
  - has limited functionality, but good to start with

adaptTo()

Recommendations & Best Practices (2/2)

- Avoid Coral 2 components (soon deprecated)
  - Coral 2: granite/ui/components/foundation/form/textfield
  - Coral 3: granite/ui/components/**coral**/foundation/form/textfield
- Avoid deprecated Coral 3 components
  - Example: *pathbrowser* deprecated with AEM 6.3
- Behavior of components can break with product updates
  - Example: customized *assetdetails* stopped working after AEM 6.3 SP2
  - Check release notes

Pain Points and Room for Improvement

- Richtext (RTE) component with numerous problems
  - Validation not working as expected
  - Various rendering issues (fix in future releases)
- Use of *sling:hideResource* causing inheritance locks to disappear
- Dialog layout problems
  - Example: Tooltips overflow dialog frames
  - unable to use workaround due to AEM “sustainable upgrades”

Thank You.

References

- Adobe Granite UI documentation: https://adobe.ly/2QetbPf
- Granite foundation-validation documentation: https://adobe.ly/2Nrn5MR
- Coral 3 component documentation: https://adobe.ly/2McKlcI
- Dialog Conversion Tool: https://adobe.ly/2O0M9HF
- Sustainable upgrades: https://adobe.ly/2MZVFhO

Appendix 1: Touch UI Component Example (1/2)

render.jsp

```
<%
AttrBuilder attrs = tag.getAttrs();
cmp.populateCommonAttrs(attrs);
[...]
%>
<coral-fileupload <%= attrs.build() %>
    [...]
</coral-fileupload>
```

Appendix 1: Touch UI Component Example (2/2)

HTML

```
<coral-fileupload
  class="coral-Form-field cq-FileUpload …"
  id="touchui-dialogimagetab-image"
  name="./image/file"
  data-foundation-validation="dd-image"
  accept="image/jpeg,image/png"
  action="/content/…/teaserelement_9c21” […]>
  […]
</coral-fileupload>
```

Appendix 2: Naming Conventions

Create naming conventions for Component IDs

- Readable naming structure
- Easily control JavaScript observation
- Components must have unique IDs for component-specific behavior
- IDs should contain common “parts” for applying shared behavior (as an alternative to using *granite:class*).

touchui-<component-type>-<component-name>[-<…>]

- Example: touchui-dialog-calltoaction-button
  - Apply notifications on touchui-dialog-*
  - Apply click event on touchui-*-calltoaction-button

Appendix 2: Naming Conventions Example