BACKBASE

Widget Architecture 2.0 development

Widget Extensions

Agenda

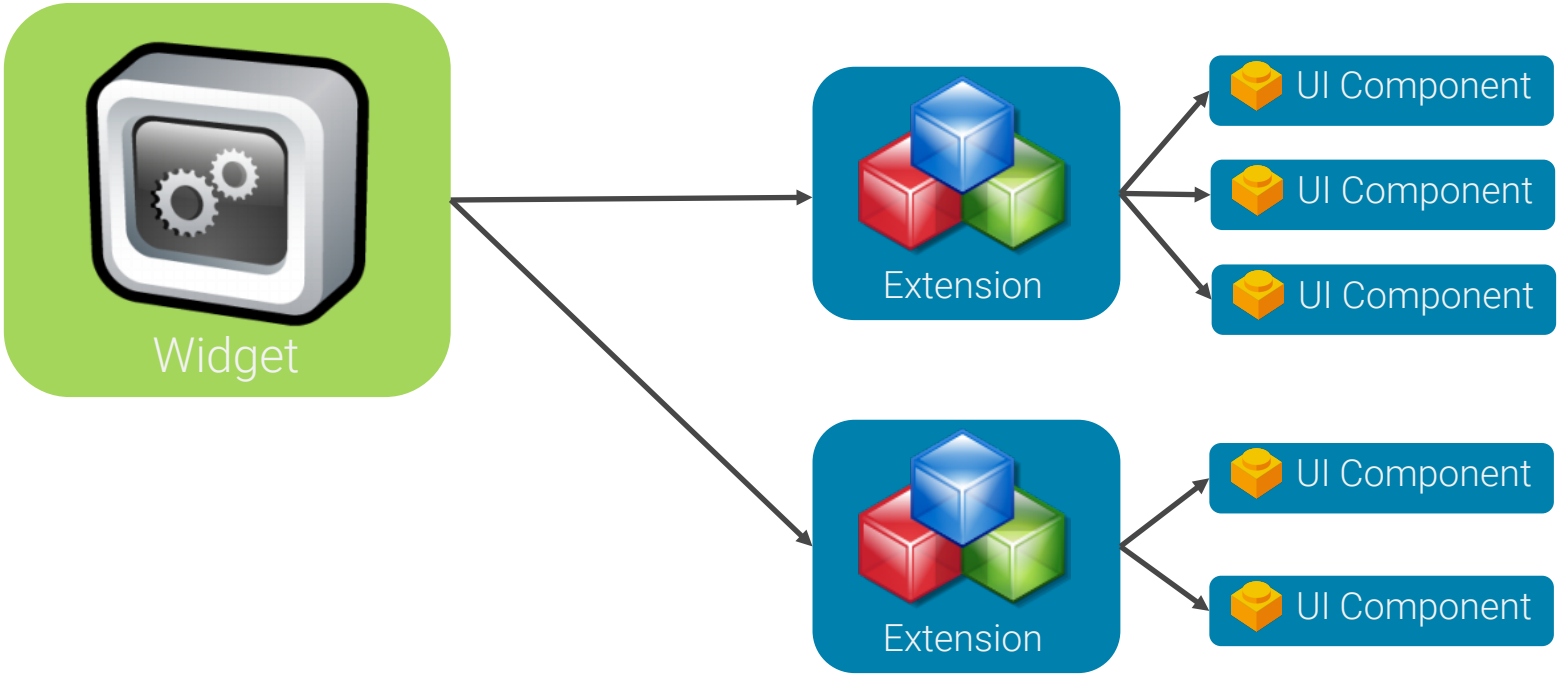
1. Understanding Extensions
2. Extending Widget Functionality

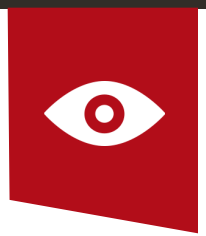
BACKBASE

Understanding Extensions

Extensions Overview

*"An extension allows developers to **customize** a widget's **logic** and **presentation**. The combination of the widget-core and its extension defines the widget"*

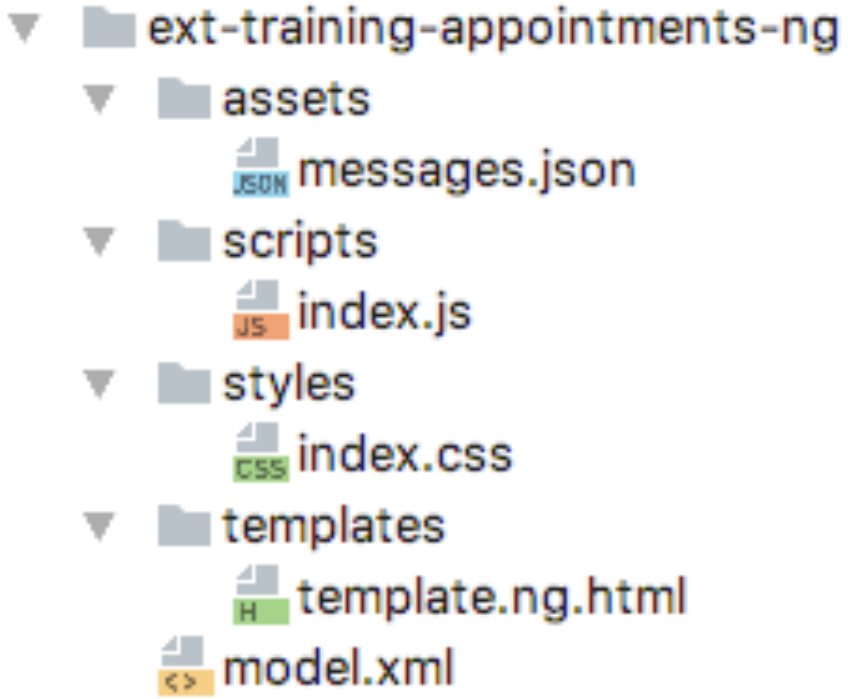

Widget
Extension
UI Component
UI Component
UI Component
Extension
UI Component
UI Component



Let’s look at how flexible widgets are by switching their extension.

Extension scaffolding

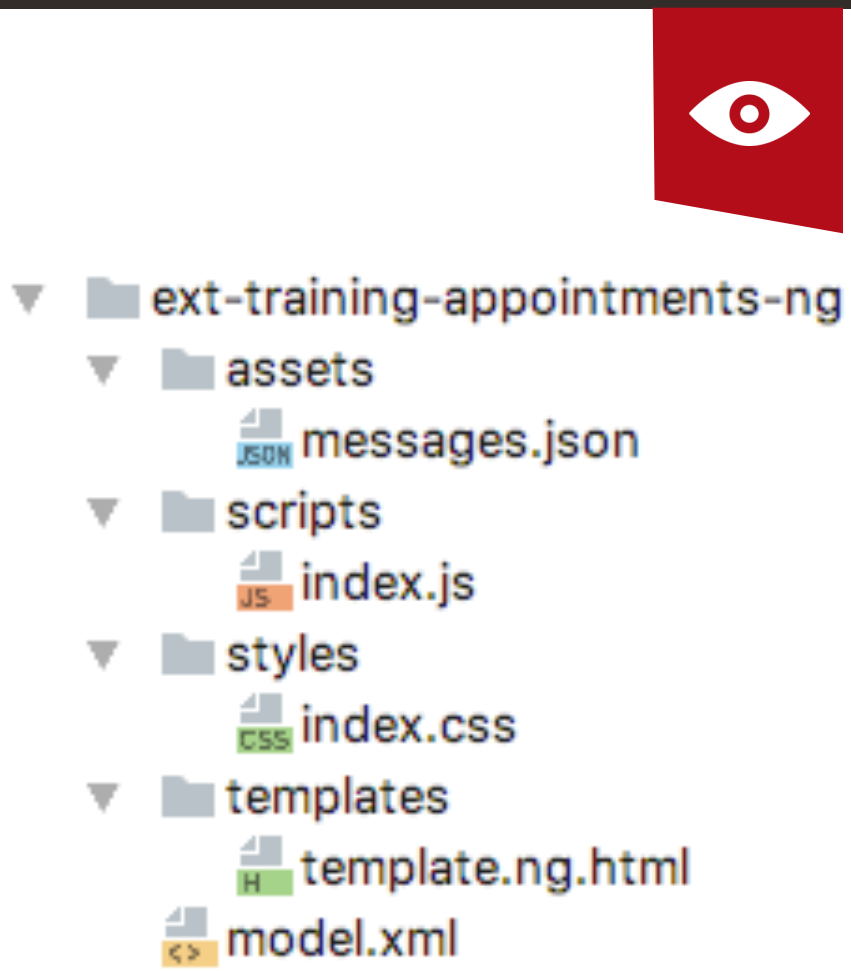
- Presentation
- Styles
- Behavior
- Definition
- Assets



Extension scaffolding

- **Presentation**: AngularJS Template.
- **Styles:** CSS File. Just if it's REALLY needed.
- **Behavior:**
  - Inject dependencies
  - Implement Hooks
  - Implement Helpers
  - Subscribe to events

- **Presentation**
  AngularJS Template
- **Styles**
  CSS
- **Behavior**
  - Inject dependencies
  - Implement Hooks
  - Implement Helpers
  - Subscribe to events



BACKBASE

Extending Widget Functionality

Understanding Extensions

Exercise

Create a tile view for the Contact Manager Widget by generating a custom extension.

1) Use the UI Component **“ui-bb-avatar”** for the tile view

2) Customize Widget Styles

BACKBASE

Hooks

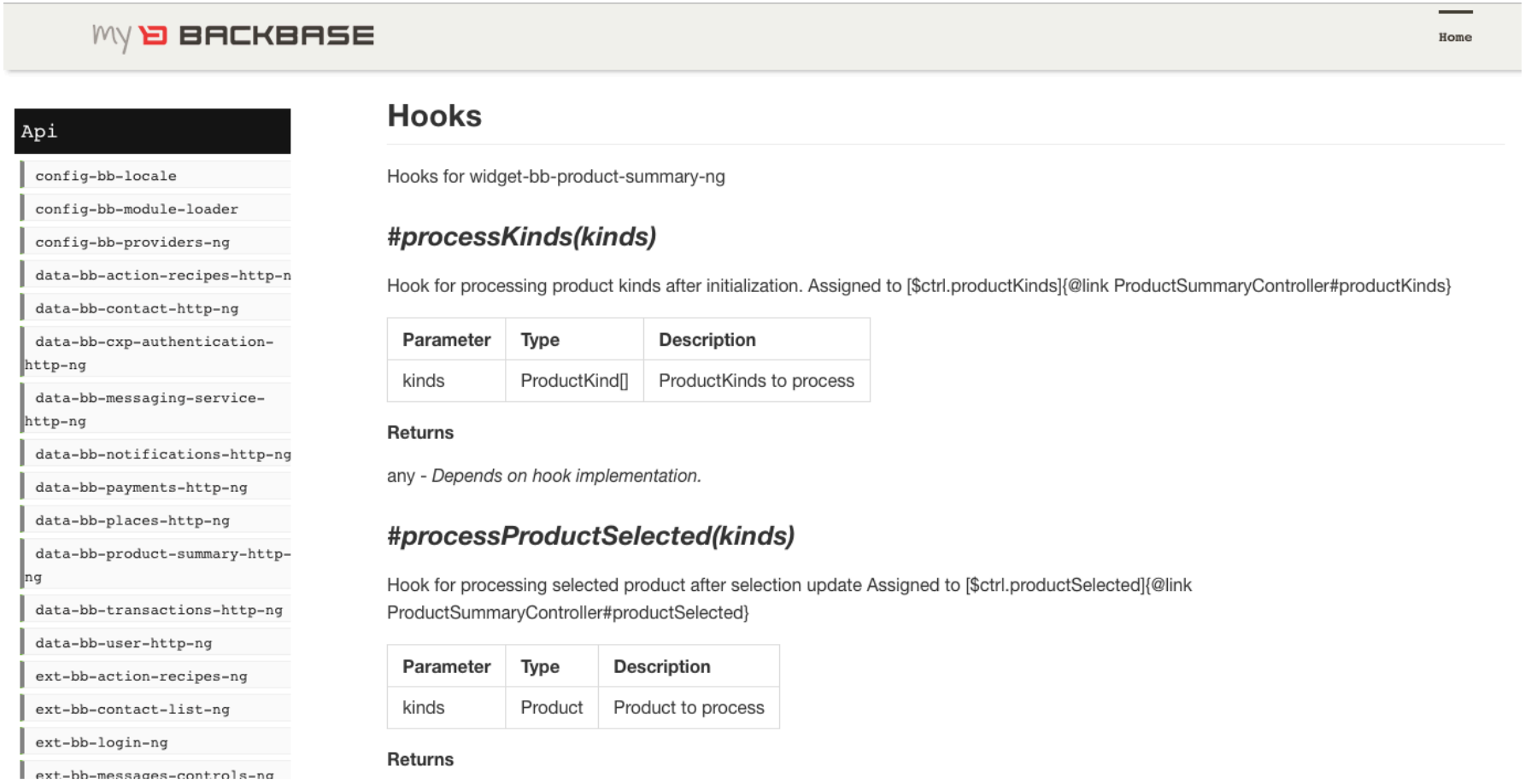
Extending Widget Functionality

*"Hooks are **defined places** within a **widget-core** from which a **developer's code can be called**. **This allows data** being passed to an endpoint and data received from an endpoint **to be changed in someway**."*

Implementing a hook

- Hooks provides the ability to transform received data or data to be sent
- Developers can not add new data fields within a hook's implementation
- A hook will be called only if it is implemented
- Every hook provides a specific set of data to the implementation.
- The available hooks list can be found in the Widget's Documentation

Product Summary Widget hooks



Product Summary Widget hooks

#processKinds(kinds)

Purpose:
Intended to allow the Developer to process the different kind of account types.

Return:
Any. It depends on the implementation

Example:
The implementation will return reversed product kinds names

```
const NAME_PREFIX_PREFERENCE = 'namePrefix';
export const hooks = (context) => {
    const NAME_PREFIX = context.widget.getStringPreference(NAME_PREFIX_PREFERENCE);
    return {
        processKinds: (productKinds) => productKinds.map(kind =>
            Object.assign({}, kind, { name: `${NAME_PREFIX}${kind.name}` })
        )
    };
});
```

Exercise

Implement a hook within the Contact Manager Widget extension that will show a different avatar image.

1) Find the Hook that will allow you to change the avatar image

2) Get the avatar image externally

*"A widget is **black-boxed** and developers cannot add new preferences to the widget. In order to implement **Extension-Specific Preferences**, we need to create them **manually** in **CXP Explorer** and **document** it"*

For documentation purposes

```
// model.xml


<catalog>
  <feature>
    <name>ext-training-contact-tiles-ng</name>
    <contextItemName>[BBHOST]</contextItemName>
    <properties>
      <property label="Version" name="version" viewHint="text-
input,admin,designModeOnly">
          <value type="string">0.1.0-alpha.0</value>
      </property>
      <property name="randomUserApiEndpoint" label="Random Lego Avatar"
viewHint="admin,designModeOnly">
        <value type="string">
            https://randomuser.me/api/portraits/lego
         </value>
      </property>
    </properties>
  </feature>
</catalog>
```

viewHint attribute

```
<property label="Version" name="version" viewHint="designModeOnly,admin,text-input">
     <value type="string">0.1.0-alpha.0</value>
</property>
```

- String composed of the following optional, comma-separated parts

1 `[designmode],[role],[input-field]`

  - `[designmode]`: *Where the property can only be edited within CXP Manager*
  - `[role]`: *The role needed to edit the property via the preference form*
  - `[input-field]`: *What input field is displayed in the preference form*

Exercise

Refactor the Hook to use an Extension-Specific Preference

1) Use the Widget Object in the Hook

2) Create the property via CXP Explorer

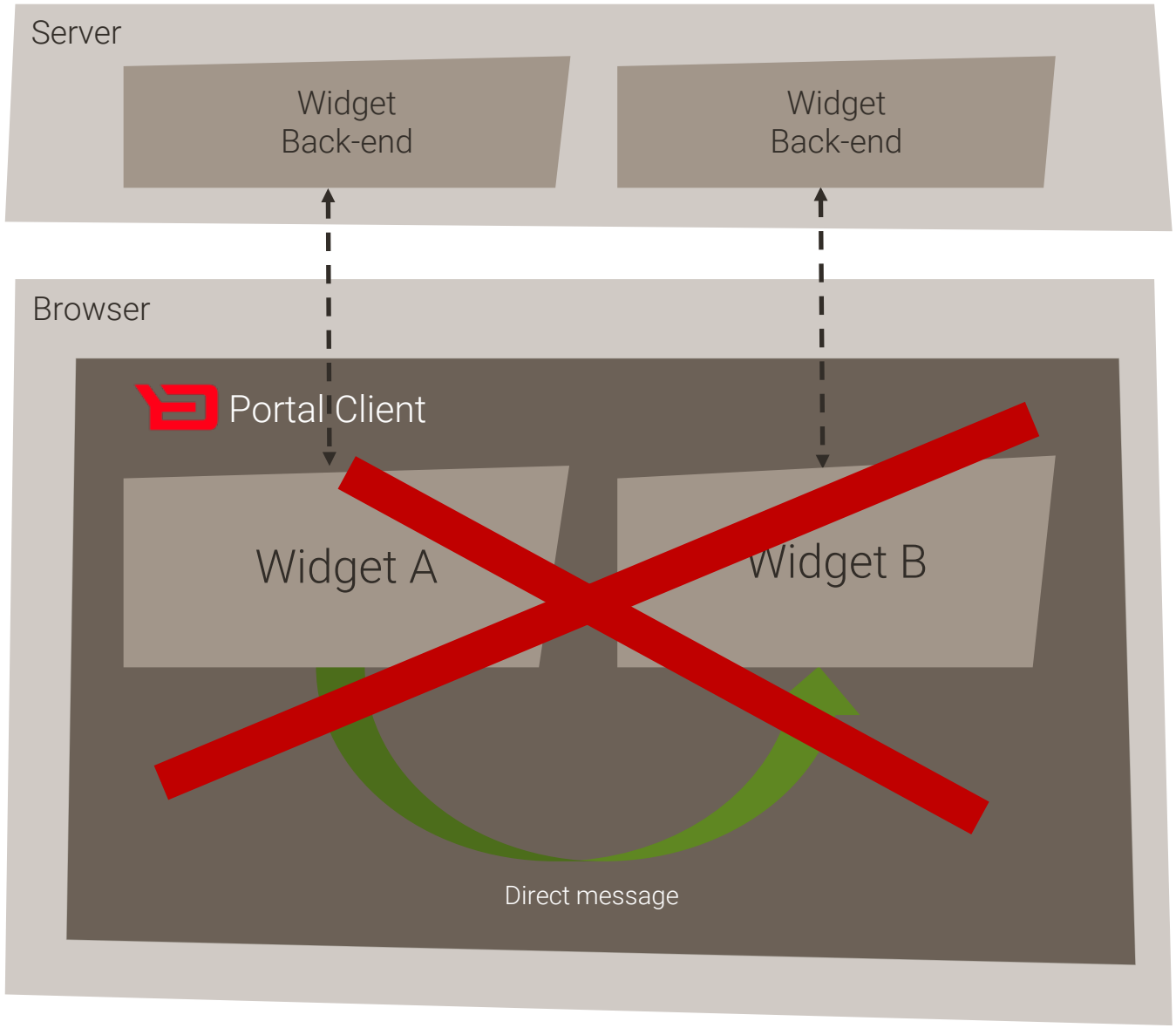
BACKBASE

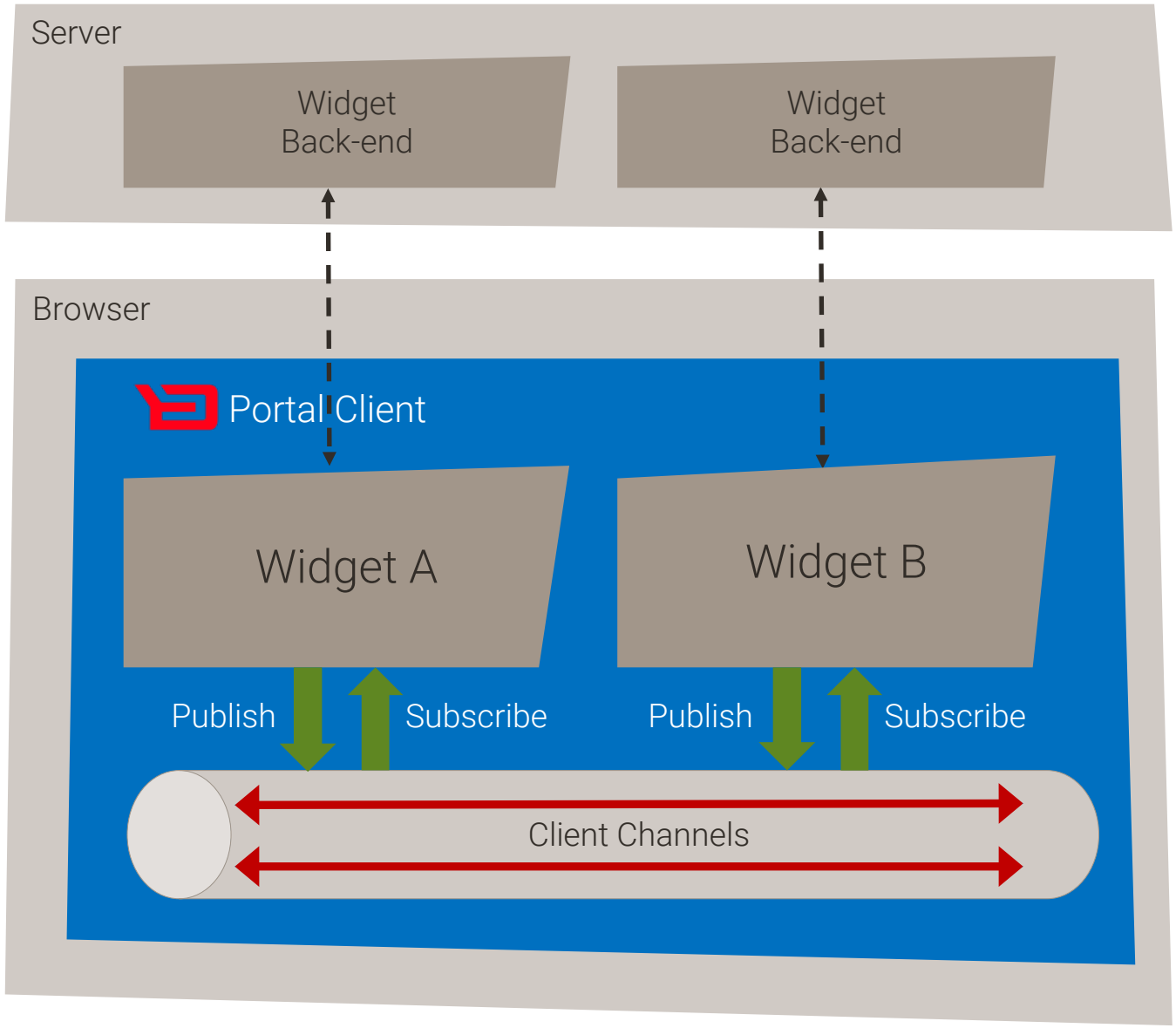
Events

Extending Widget Functionality

*“The Publish and Subscribe pattern, often referred to simply as **PubSub** is used to **pass information between widgets**.”*

Widget Functionality – Pubsub




Server
Widget Back-end
Widget Back-end
Browser
Portal Client
Widget A
Widget B
Publish
Subscribe
Publish
Subscribe
Client Channels

Product Summary Widget events

Events

Pub/sub events allow for interwidget communication.

The following is a list of pub/sub events the widget subscribes to:

Event	Action
`bb.event.product.selected`	Updates the display, for example, to show more details when the user has selected a product.

The following is a list of pub/sub events which the widget publishes:

Event	Action	Arguments
`bb.event.product.selected`	Published when a user has selected a product.	product
`bb.event.product.selected.${product.kind}`	Published when a user has selected a product kind (category).	product

Product Summary Widget events

bb.event.product.selected

Purpose:
Published when a user has selected a product.

Arguments:
Selected product.

Example:
Subscribing to bb.event.product.selected

```
export const events = {
    "bb.event.product.selected": (product) => console.dir(product)
};
```

Context object

- Providing the context object is a common pattern for extension helpers and events when exporting them as functions.
- Context object properties:
  - $filter: The angular $filter object
  - publish: The publish function. (Pub / Sub)
  - widget: The widget object.

```
export const events = ({ $filter, publish, widget }) => {
    "bb.event.product.selected": (product) => console.log(product)
};
```

Demo

Subscribing to events.

BACKBASE

Helpers

Extending Widget Functionality

*“Sometimes you need some **extra logic** in your view that is **not part** of the **controller interface**. For this you can use **helpers** to move view logic out of the template.”*

Implementing an extension helper

Example:

Highlighting products that have balance below zero.

```
export const helpers = ({$filter, publish, widget}) => ({
  isLowBalance:
    (product) => product.availableBalance < widget.getLongPreference('lowBalanceThreshold')
});
```

Tip: Widget API documentation

The helpers are made available on the root scope on **ext.helpers.<helper-name>**

```
<span ng-class="{low-balance: ext.helpers.isLowBalance(product)}"></span>
```

Exercise

Implement an extension helper within the Contact Tiles Extension

- Implement a custom preference to show or hide the IBAN #
- Show / hide the IBAN # depending on the preference value

BACKBASE

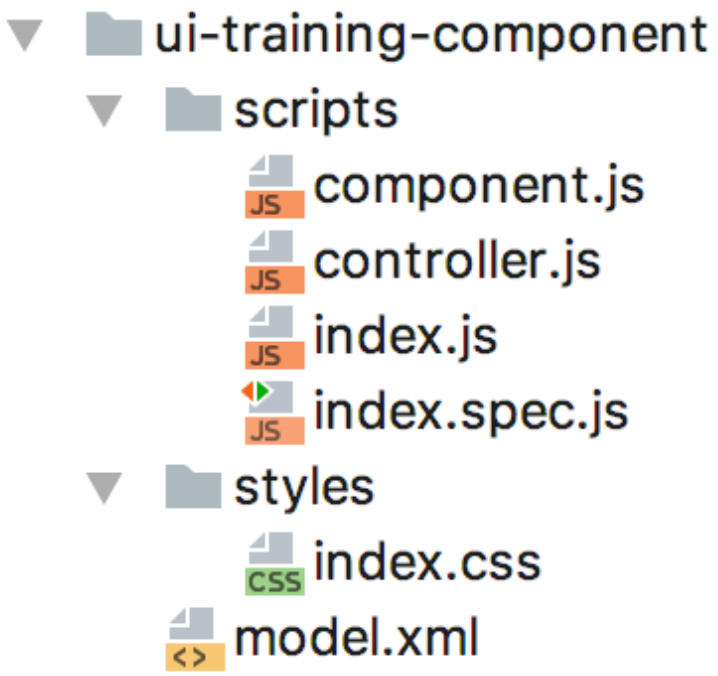
UI Components

Extending Widget Functionality

”UI Components ***maximize the reusability of frontend code****, allowing the developer to build several views with the same frontend blocks”*

UI Component scaffolding

- Component (template)
- Controller
- Styling
- Definition



Demo

Creating a UI Component from the scratch.

BACKBASE

Thank you!

www.backbase.com
sales-eu@backbase.com

New York: +1 646 478 7538
Amsterdam: +31 20 465 8888