FDC322692

Forge Configurator for Construction Manufacturing Assemblies

Mark Mendez
Faith Technologies

Learning Objectives

- Create a basic assembly configurator with adjustable configuration
- Design a model translation workflow to update the configurator model
- Develop an extension that allows for custom user interaction in the Forge Viewer
- Understand current challenges in flexing configurator models and options available to enhance configuration behavior

Description

With Revit and Forge, we successfully developed and deployed a Configurator that enables clients to configure and order assemblies from our manufacturing facilities to be delivered straight to the job site for installation. This allows project managers to know exactly what will be delivered and gives them an option to order assemblies straight from their mobile devices—safe in the knowledge that what is delivered will be exactly what they ordered. In this class, we'll take a look at the Configurator that we built and talk through various aspects that were important to consider when developing it; including the difficulty in flexing geometry by using isolation techniques, showing packs of items with different quantities using THREE.js to duplicate the geometry and data, setting axes of rotation to allow for better interaction, and showing custom dimensions by default that user feedback requested.

Speaker(s)

Mark Mendez has extensive expertise and knowledge in computational BIM integration. He has a Bachelor's of Science in Engineering Physics from the Colorado School of Mines, and while spending a few years as an AEC consultant, he's detailed solutions that have been delivered in every aspect of BIM. From Architectural, Structural, and MEP design, to Facilities Management and overall building life cycle across many different platforms and software. Now, he currently works on a research and development team at Faith Technologies, where they design and implement cutting-edge BIM-oriented applications to grow the organization's Manufacturing abilities and enable efficiencies in Construction projects.

Creating a Basic Assembly Configurator

Why should we make a configurator?

There're a few different business processes that a configurator can facilitate. One business process that can be seen is allowing a client to be a part of the design decisions, while another business process is communicating to a contractor the specifications of a manufactured product. Up until now, we've mainly solved these business process with sit-down meetings, going through a few different options in person, or we would hand documents back and forth and communicating through engineering or submittal drawing sets. A configurator for Construction Manufacturing, aims to solve the business process of communicating engineering specifications. It also has the side effect of being able to fluidly display your entire product catalog.

With a configurator powered by the Forge Platform, we have a simple user experience to allow prospective clients to navigate our product catalog. When they navigate the catalog, they can understand dimensions, select hardware and investigate properties, and view the product from many different angles. One value that this adds is that the client can make a more informed decision. On the manufacturing side, we can display configurations in a way that can direct a client toward configurations that are more common. This allows us the ability to integrate recommendations, have greater elasticity of production by keeping certain products in stock, and we can further provide more competitive pricing on products.

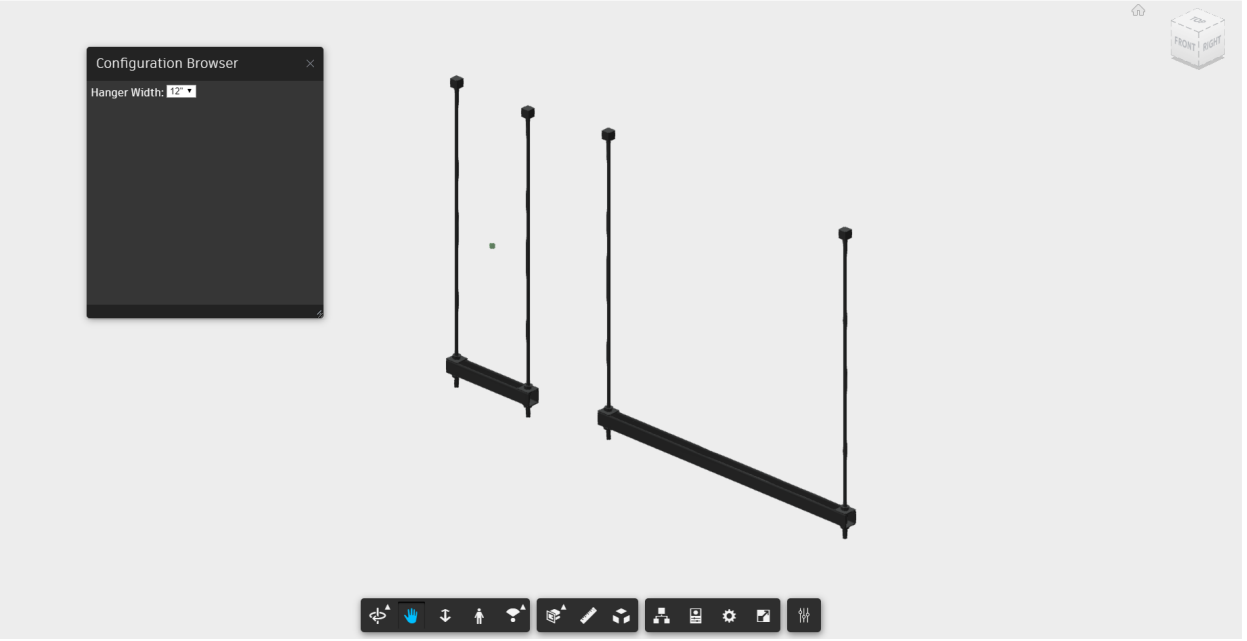
How do we define a Configurator?

The core functionality of a configurator is to interface different options and enable a user to make selections on any subset of those options which results in a unique specification of a product.

Here, the configurator we develop will be a Forge Viewer Extension. The extension has an input of a group of properties and their options available, how the option selections map to the specific geometry ID in the configurator's model, and a process to change which configuration is being shown.

Creating a Configurator

To begin, we'll create a basic configurator that switches between two different options based on the value of a control. We'll start with a translated model that has two objects that represent two different configurations of a product.



Configuration Extension

The two primary functions of the configurator are the controls, and the action to configure a model when a control is changed. We'll jump right in by first creating a new extension, and then we'll add these two primary functions to the extension to create a configurator.

We'll start by creating a new extension by extending the Autodesk.Viewing.Extension class provided by the Forge Viewer API:

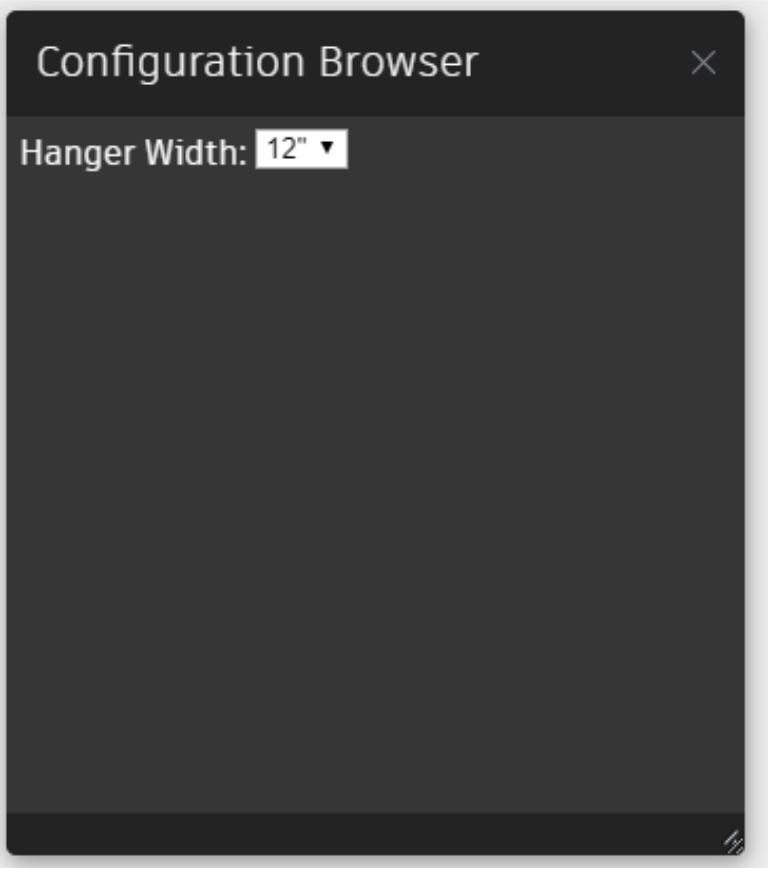
```
class Configurator extends Autodesk.Viewing.Extension {
  constructor(viewer, options) {
    super(viewer, options);
    this.viewer = viewer;
    this.createToolbarButton;
    this.panel = null;

    // Custom Event Names implemented
    this.CONFIGURATOR_DATA_UPDATE = 'configuratorDataUpdate'
    this.CONFIGURATOR_DATA_CHANGED = 'configuratorDataChanged'
    this.CONFIGURATION_CHANGED_EVENT = 'configurationChanged'
  }

  load() {
    if (this.viewer.toolbar) {
      // Toolbar is already available, create the UI
      this.createUI();
    } else {
      // Toolbar hasn't been created yet, wait until we get notification of its c
reation
      this.onToolbarCreatedBinded = this.onToolbarCreated.bind(this);
      this.viewer.addEventListener(this.viewer.TOOLBAR_CREATED_EVENT, this.onTool
barCreatedBinded);
    }
    console.log('Configurator Loaded!')
    return true;
  }

  unload() {
    this.viewer.toolbar.removeControl(this.subToolbar);
    return true;
  }

  onToolbarCreated() {
    this.viewer.removeEventListener(this.viewer.TOOLBAR_CREATED_EVENT, this.onToo
lbarCreatedBinded);
    this.onToolbarCreatedBinded = null;
    this.createUI();
  }
```

```
  // Here we create the Configurator Tool button clicked
  // by the user to activate the Configurator and open the Configurator Panel
  createUI() {
    const buttonName = 'configurator-configuration-button';
    const button1 = new Autodesk.Viewing.UI.Button(buttonName);

    // We'll show the docking panel when the configurator button is clicked
    button1.onClick = () => {
      this.showDockingPanel();
    };
    button1.addClass('configurator-configuration-button');
    button1.setToolTip('Configure');

    const toolbarName = 'configurator-toolbar';
    this.subToolbar = new Autodesk.Viewing.UI.ControlGroup(toolbarName);
    this.subToolbar.addControl(button1);

    this.viewer.toolbar.addControl(this.subToolbar);
  }


  // Every extension must define a method to get its ID
  static get ExtensionId() {
    return 'Configurator.Extension.Configurator';
  }
}

// Here we register the extension, and we will activate the
// extension when instantiating the viewer
Autodesk.Viewing.theExtensionManager.registerExtension(
    Configurator.ExtensionId,
    Configurator
);
```

We can see in the constructor method; we defined a few custom event names. These events will be used to load new data and to listen for changes and communicating those changes to the UI panel.

We defined and call a method ‘createUI’, where we add a button to the Forge Viewer Toolbar. This is how users will open the configurator options browser.

Next, we'll set up a new DockingPanel. This panel will display the Configurator controls. It will look like this:

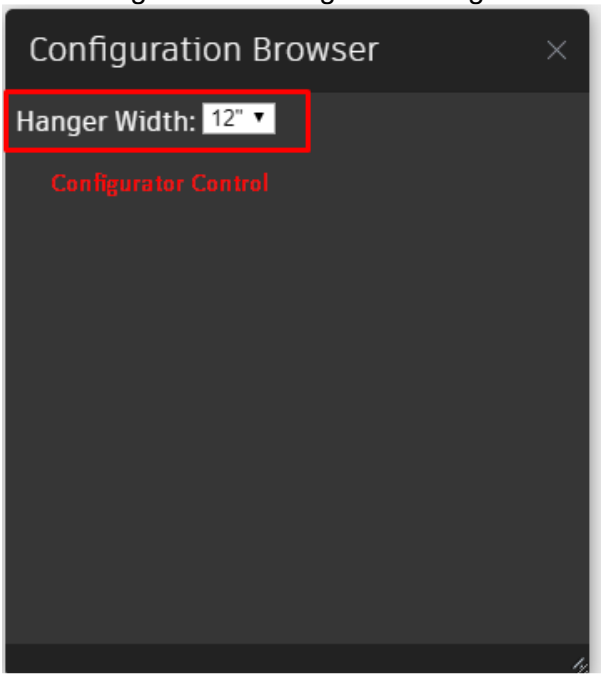


We do this by extending the Autodesk.Viewing.UI.DockingPanel class provided by the Forge Viewer API:

```
class ConfiguratorConfigurationPanel extends Autodesk.Viewing.UI.DockingPanel {
  constructor(viewerContainer, container, id, title, options, viewer, configurato
r) {
    super(viewerContainer, container, id, title, options);
    this.configurator = configurator;
    this.viewer = viewer;
    this.create();
  }

  // when the panel is opened
  create() {
    // container here refers to the docking panel
    // the below attributes define the initial state of the panel
```

```
    this.container.classList.add('docking-panel-container-solid-color-a');
    this.container.style.top = '10px';
    this.container.style.left = '10px';
    this.container.style.width = '350';
    this.container.style.height = '400';
    this.container.style.resize = 'auto';

    this.createConfiguratorControlsArea();
    this.updateControls();


    // Lastly we define an event listener to update the controls when
    // the core extension sends the panel new control data
    window.addEventListener(this.configurator.CONFIGURATOR_DATA_CHANGED, () => {
      this.updateControls();
    });
  }

  // Here we define the DOM elements that
  // represent the controls within the panel
  createConfiguratorControlsArea() {
    const $controlsTable = $('<table>', {
      id: 'configurator-configuration-area',
      class: 'docking-panel-scroll',
    });
    const $controlsList = $('<tbody>', {
      id: 'overall-controls-body',
    });

    $controlsList.appendTo($controlsTable);

    // Add controls area to the Forge UI Panel (container)
    this.container.appendChild($controlsTable[0]);
  }

}
```

So far, we have the Configurator Extension and a Configurator Panel defined. Now, we need to add the logic for receiving new configurator control data and creating new controls.



First, we receive data and we'll store it in the browser database for future retrieval by the panel. Then, we trigger the event that the UI listens for to create its controls:

```
class Configurator extends Autodesk.Viewing.Extension {

 ...

   window.addEventListener(this.CONFIGURATOR_DATA_UPDATE, (event) => {
      this.setConfiguratorDB(event);
   });

 ...

   setConfiguratorDB(event) {
     localStorage.setItem('configuratorData', JSON.stringify(event.detail));
     window.dispatchEvent(new Event(this.CONFIGURATOR_DATA_CHANGED));
   }
}

class ConfiguratorConfigurationPanel extends Autodesk.Viewing.UI.DockingPanel {

...
```

```
    window.addEventListener(this.configurator.CONFIGURATOR_DATA_CHANGED, () => {
      this.updateControls();
    });

...


  updateControls() {
    const configuratorDataDB = localStorage.getItem('configuratorData');
    if (!configuratorDataDB) {
      return;
    } else {
      const configuratorData = JSON.parse(configuratorDataDB);
      const controlsData = configuratorData.controls;
      const controls = this.createControls(controlsData);
      this.setPanelControls(controls);
      this.setControlChangedEvents();
      this.controlsChangedEvent();
    }
  }

...

  // We watch each control for a change. When the change occurs,
  // we aggregate the new selections, and we send that in an event
  // that is caught by the Core Extension which will update the viewer
  controlsChangedEvent() {
    const configurationCodeComponents = [];
    $('.configurator-control').each((index, control) => {
      configurationCodeComponents.push($(control).val());
    });
    const configurationCode = configurationCodeComponents.join('-');
    window.dispatchEvent(new CustomEvent(this.configurator.CONFIGURATION_CHANGED_
EVENT, { 'detail': configurationCode }));
  }
}
```

In the above code we implemented the logic to switch between configurations. Each configuration is defined an aggregate of each selected control value. This aggregate selection we call an ‘Assembly Code’. We use this code to lookup relevant information related to the specific configuration.

We aggregate the selected configurator values to create an Assembly Code, and then we use the assembly code to look up the model element the assembly code corresponds to. We'll study this relationship more in depth when we show the model translation process, but essentially the assembly code maps to a specific UniqueId of an element in the translated Revit model. This mapping is created when the Revit model is created, and then we pass the mapping between assembly codes and UniqueIds to our configurator.

When we retrieve the UniqueId (Forge labels this an ExternalId), we then can use the Forge Viewer mapping between ExternalIds and dbIds to isolate the model element that corresponds to the configuration selected.

```
class Configurator extends Autodesk.Viewing.Extension {
 ...
    window.addEventListener(this.CONFIGURATION_CHANGED_EVENT, (event) => {
      this.handleConfigurationChange(event);
    });
...

// To handle a configuration change, we take the new aggregated assembly code,
// we lookup the externalId(s) for that object (from the Revit model), and we
// then lookup the dbId(s) that the externalId(s) are mapped to.
// Then we use the new dbId to display the configuration in the forge model
handleConfigurationChange(event) {
    const configurationCode = event.detail;
    let configuratorData = localStorage.getItem('configuratorData');
    if (!configuratorData) {
      return;
    } else {
      configuratorData = JSON.parse(configuratorData);
      const configuratorMapping = configuratorData.configurationMapping;
      const uniqueId = configuratorMapping[configurationCode];
      if (uniqueId) {
        this.configureElementByUniqueId(uniqueId);
      }
    }
  }

  configureElementByUniqueId(uniqueId) {
    // Every Forge Viewer model has an 'ExternalId Mapping'
    // this mapping is an object that has as keys the
    this.viewer.model.getExternalIdMapping((mapping) => {
      this.configureElementByUniqueIdAndMapping(uniqueId, mapping);
    });
```

```
  }

  configureElementByUniqueIdAndMapping(uniqueId, mapping) {
    const elementDbId = mapping[uniqueId];
    if (elementDbId) {
      this.viewer.isolate(elementDbId);
      this.viewer.fitToView(elementDbId);
    }
  }
}
```

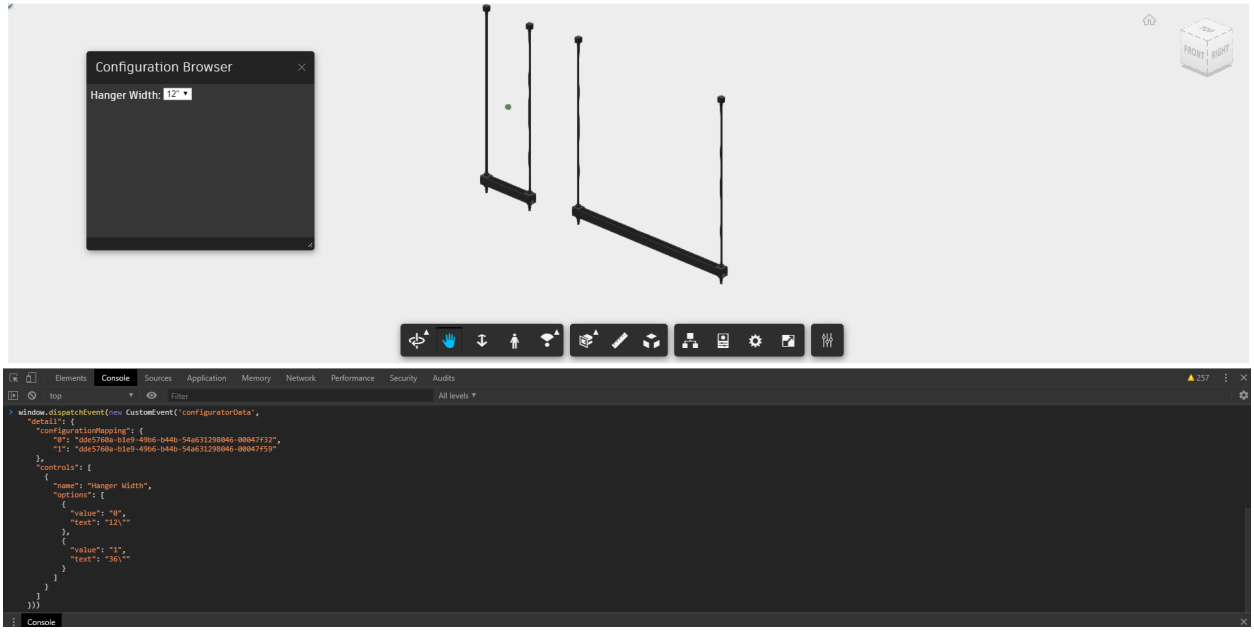
Now, we have a basic configurator implemented. We can feed it data. As an example, we can have the data look like this:

Note how we have controls defined, and we also have the mapping between assembly codes and UniqueIds, which we created when we created the Revit model to be used in the Viewer.

```
{
  'detail': {
    // The '0' key below is the assembly code. Given the below controls
    // this assembly code would be selected if 'Hanger Width' was changed
    // to 12"
    'configurationMapping': {
      '0': 'dde5760a-b1e9-49b6-b44b-54a631298046-00047f32',
      '1': 'dde5760a-b1e9-49b6-b44b-54a631298046-00047f59'
    },
    // Each control has a name, and an array of options
    // each option has a value, which is the assembly code component
    // and a text, which is the displayed value that the user will choose
    'controls': [
      {
        'name': 'Hanger Width',
        'options': [
          {
            'value': '0',
            'text': '12"',
          },
          {
            'value': '1',
            'text': '36"'
          }
        ]
      }
```

```
    ]
  }
}
```

We can load this data into the configurator by using the console within our browser dev tools:



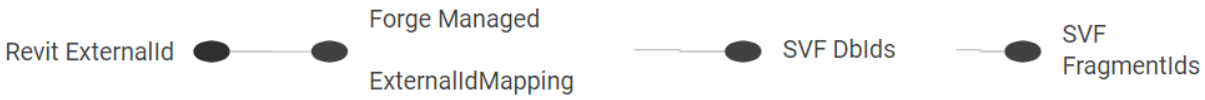
When this event is dispatched, the Configurator Extension will receive the data the event provided, it will create the controls, and it will isolate and fitToView our initial configuration.

Switching between 12” and 36” in the above example, will transition between the two hanger elements.

The source code for this example can be found here: https://github.com/mossherder/au2019-configurator/blob/master/app/static/js/src/ConfiguratorExtension.js

Mapping a Configuration to Model Geometry

The structure of the Forge Viewer API as it relates between model UniqueIds or ExternalIds, and the SVF model's dbIds, is shown in the below image.



ExternalIds are the UniqueIds provided in the external (original) model before it is translated.

The Forge Viewer API, provides an ExternalIdMapping object which we use to retrieve the Forge Viewer dbIds (used in looking up element properties, isolating, etc.).

Lastly, the geometry in the Forge Viewer can be retrieved if we know the FragmentIds, and the Viewer API provides a method to retrieve the fragment ids given a dbId.

If we have complex products in our original model which we want to configure, such as a family with nested, shared families, we may have to use the dbId retrieved from the externalMapping object, to then lookup child elements of that dbId during our process. We will see an example where we implement the extension to display a custom dimension on our configured product later in this handout.

Design a Model Translation Process to Update the Configurator Data

As a product catalog and the associated configurator grow, we will have to handle more engineering changes to the products, and one aspect that will become more intensive from a data management standpoint is the organization and management of product changes as they relate to updating the configurator.

Why think about designing this process?

The maintenance of a manufacturing product catalog can involve engineering changes of varying impact as it relates to the functionality of the configurator. The addition of options could not only change the configurable options list, but it might also necessitate an update to the model providing the geometry to the Forge Viewer.

Autodesk has a few different products to manage Product Lifecycle, and since we have our configurator developed on the Forge Platform, we also have the potential to integrate with an ERP system or any existing system an organization uses for PLM.

With all the different options available, we can have a fully automated process, a fully manual process, or anywhere in between.

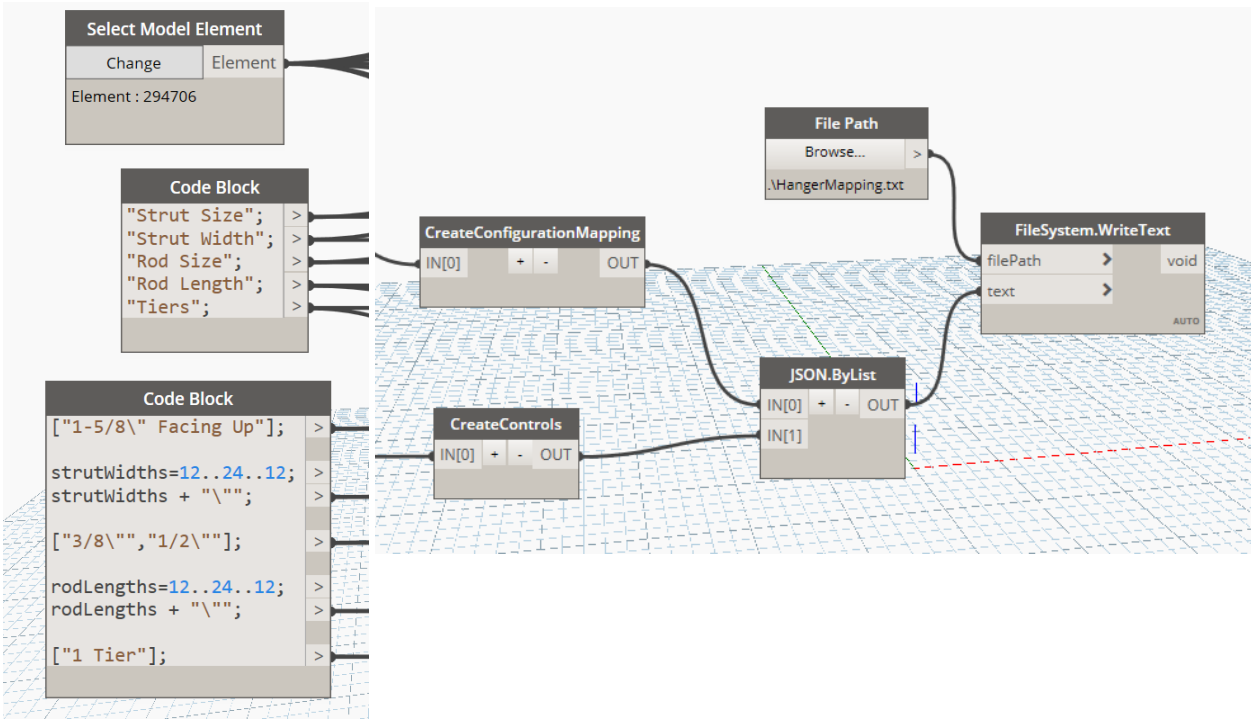
Creating a semi-automatic configurator update process

For our first implementation of an update process, we first look at what we need to update for any given product.

- Configurable Options
- Specific Configuration to model ExternalId mapping
- Translated model (RVT -> SVF)

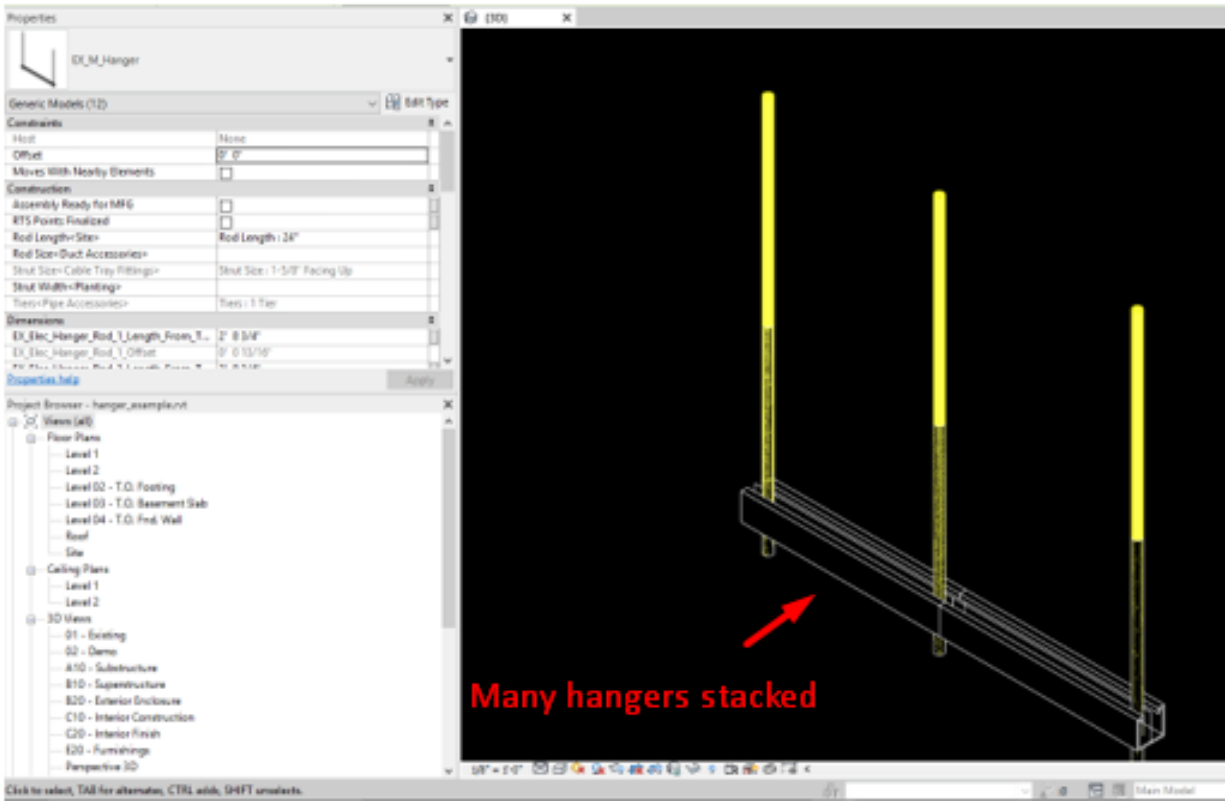
First, we'll look at updating both the configurable options and generating the mapping between a specific configuration and it's ExternalId.

We can use a few different tools for this, and the implementation we have for this class is to show the process using Dynamo for Revit. We set up a way to select which model element we want to configure (in the example above it would be the Hanger), we can choose the various options, and then we decide which values are available to select from.



Running our Dynamo script, we generate an updated model with all of the configurations placed, and we generate a JSON object which defines the configurator data that relates assembly codes to UniqueIds, and which also defines the controls for this product configurator.

Here is an example of what the model looks like after this particular run:



This model has all the configurations stacked on top of each other, but we can also space them out (which is needed when our configurations available gets large).

We now take this model and translate it using the Model Derivative API. The urn from the translation process, and the updated configurator data can now be passed to the configurator, and we'll have more options available in our model.

For all of our Forge applications, we created a basic Python Forge SDK. This SDK is a work in progress with regards to the developer API, but it does wrap most of the Model Derivative API, the BIM 360 API, and some of the Authentication API. The SDK can be found here: https://github.com/mossherder/au2019-configurator/tree/master/app/forge

The files for this model translation process can be found here: https://github.com/mossherder/au2019-configurator/tree/master/resources/modelTranslationProcess

Develop Extensions for Custom User Interaction

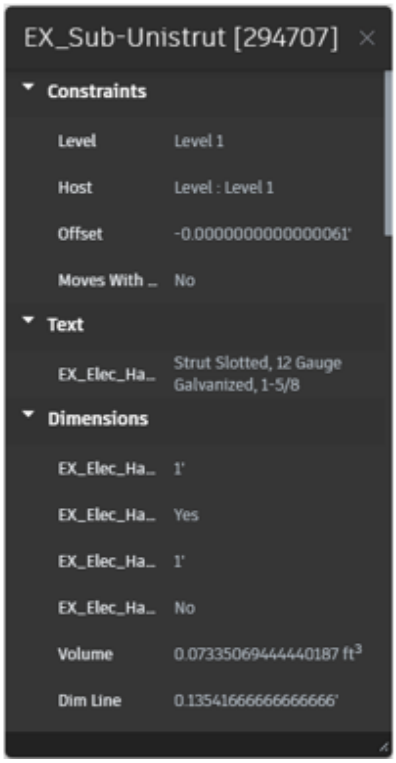
With any Forge Viewer application, we can use extensions to provide better user experiences. When a model is translated using the Forge Model Derivative API, the resulting translated model has properties representing every parameter available in the original model. This is great from a programmatic standpoint, but a downside is in the Forge Viewer properties panel, where we have many, many properties that won't provide value to the user of the application. Another interesting opportunity to use an extension is when the application is used on a mobile device, particularly a smaller device such as a phone. When you zoom by pinching or expanding on the screen, the Forge model can rotate in a weird way, and the user is then stuck trying to rotate it back. We can develop an extension to limit how the model can rotate to allow a simpler user experience. Finally, we'll look at developing an extension that displays a custom dimension that users find important and would like to see when configurating a product.

The list goes on, but we will now develop extensions for our application that make for a better user experience.

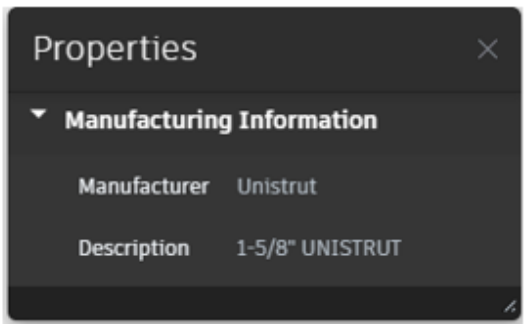
Customizing the Properties Panel

The first user interaction extension will override the properties browser. The default properties browser displays every parameter that was available to the element, and we'll add value by limiting these properties to only the important properties.

Before



After



We'll start by creating a new extension by extending the Autodesk.Viewing.Extension object.

```
class ConfiguratorPropertiesExtension extends Autodesk.Viewing.Extension {
  constructor(viewer, options) {
    super(viewer, options);
    this.viewer = viewer;
    this.options = options;
  }

  load() {
    this.panel = new ConfiguratorPropertiesPanel(this.viewer, this.options);
    this.viewer.setPropertyPanel(this.panel);
    return true;
  }

  unload() {
    this.viewer.setPropertyPanel(null);
    this.panel = null;
    return true;
  }

  static get ExtensionId() {
    return 'Configurator.Extension.Properties';
  }
}

Autodesk.Viewing.theExtensionManager.registerExtension(
    ConfiguratorPropertiesExtension.ExtensionId,
    ConfiguratorPropertiesExtension
);
```

This allows our code to be loaded when the Forge Viewer is initialized.

When we override the Panel Browser, instead of extending a DockingPanel, we will extend the Forge Viewer ViewerPropertyPanel extension.

When the Forge Viewer calls the VIewerPropertyPanel methods, the methods we override after extending the ViewerPropertyPanel will be called instead. For our application, we need to override the setProperties method, and we also override the updateTitle method.

There is logic to filter which properties we want to display, but there is no logic to retrieve properties based on the selection. The reason for this is when the Forge Viewer calls the setProperties method, it passes in the properties of the selected element. Having the Forge Viewer retrieving these properties for us makes our code cleaner, as we then only need to filter the given properties.

```
class ConfiguratorPropertiesPanel extends Autodesk.Viewing.Extensions.ViewerPrope
rtyPanel {
  constructor(viewer, options) {
    super(viewer);
    this.viewer = viewer;
    this.options = options;
  }

  // This method is overriding the setProperties method that is originally
  // implemented by the ViewerPropertyPanel
  // Anytime the ViewerPropertyPanel is updated (selection change, etc),
  // this method will be called
  setProperties(properties, options) {
    this.properties = properties;
    this.options = options;
    this.updateProperties();
    this.updateTitle();
  }

  // We'll give our properties panel a static
  // title, but we can change that here!
  updateTitle() {
    this.setTitle('Properties');
  }

  updateProperties() {
    this.removeAllProperties();
    this.addCustomPropertiesToPanel();
  }
```

```
  addCustomPropertiesToPanel() {
    const customProperties = this.findProperties();
    this.setCustomProperties(customProperties);
  }

  findProperties() {
    const customProperties = [];
    this.properties.forEach((property) => {
      if (this.isCustomProperty(property)) {
        customProperties.push(property);
      }
    });
    return customProperties;
  }

  isCustomProperty(property) {
    return this.customPropertyNames().includes(property.displayName);
  }

  // We'll limit our property panel to these two properties
  customPropertyNames() {
    return [
      'Manufacturer',
      'Description',
    ];
  }

  // Here we can see how to add a property to the panel.
  // we call the ConfiguratorPropertiesPanel.addProperty method.
  setCustomProperties(properties) {
    const additionPropertiesGroupName = 'Manufacturing Information';
    properties.forEach((property) => {
      this.addProperty(
          property.displayName,
          property.displayValue,
          additionPropertiesGroupName
      );
    });
  }
}
```

The source code for this extension can be found here: https://github.com/mossherder/au2019-configurator/blob/master/app/static/js/src/PropertiesOverrideExtension.js

Handling Pinch Zoom Gesture

The next opportunity to customize the user interaction of our application is to handle the pinch zoom when a user is on mobile.

This interaction has been documented by Kean Walmsley on his blog: https://www.keanw.com/2017/04/fixing-pinch-zoom-in-forge-viewer-applications.html

The issue is introduced when we zoom in on a mobile device, the model not only zoomes, but also rotates.

We'll document his solution here, but having rewritten it in the framework of the other extensions that we've written so far.

First, we define the extension by extending Autodesk.Viewing.Extension.

```
class PinchZoomFixExtension extends Autodesk.Viewing.Extension {
  constructor(viewer, options) {
    super(viewer, options);
    this.viewer = viewer;
    this.options = options;
  }

  load() {
    const pinchZoomFixTool = new PinchZoomTool();

    // Our Pinch Zoom Gesture is a Tool from the Forge Viewer
    // So we register and activate it.
    const toolRegistered = this.viewer.toolController.registerTool(pinchZoomFixTo
ol);
    const toolActivated = this.viewer.toolController.activateTool(pinchZoomFixToo
l.getNames()[0]);
    if (toolRegistered && toolActivated) {
      return true;
    } else {
      return false;
    }
  }
```

```
  unload() {
    return true;
  }

  static get ExtensionId() {
    return 'Configurator.Extension.PinchZoomFix';
  }
}

Autodesk.Viewing.theExtensionManager.registerExtension(
    PinchZoomFixExtension.ExtensionId,
    PinchZoomFixExtension
);
```

For this tool, we'll extend from the Autodesk.Viewing.ToolInterface. The ToolInterface is available for us to override different behavior related to user interactions. Actions such as onClick, onMouseMove, onResize.

In order to overcome this issue with rotation, we'll take the amount zoomed in, calculated by 'how much' the user pinched and we'll apply it directly to moving the camera in the viewing direction in the camera.

```
class PinchZoomTool extends Autodesk.Viewing.ToolInterface {
  constructor() {
    super()
    this.names = ['PinchZoomTool'];
  }

  getNames() {
    return this.names;
  }

  handleGesture(event) {
    // We don't care about drag gestures - leave these to another handler
    if (event.type.indexOf('drag') === 0) {
      return false;
    } else if (event.type.indexOf('pinch') === 0 || event.type.indexOf('rotate')
=== 0) {

      // Pinch or rotate gestures should result in a zoom (with no rotation)
      let target = this._viewer.navigation.getTarget();
      if (event.type === 'pinchstart' || event.type === 'rotatestart') {
        // When the gesture starts, we'll store the view direction
```

```
        // and initial distance between the camera and target
        let position = this._viewer.navigation.getPosition();

        this._direction = new THREE.Vector3();

        this._direction.subVectors(position, target);

        this._dist = this._direction.length();
      }
      // Then we normalize the direction vector and multiply it by the
      // scale factor of the gesture, adding this to the target to get
      // the new camera position
      this._direction.normalize();

      this._direction.multiplyScalar(this._dist / event.scale);

      let newPos = target.add(this._direction);

      this._viewer.navigation.setPosition(newPos);
      return true;
    }
    return false;
  }
}
```

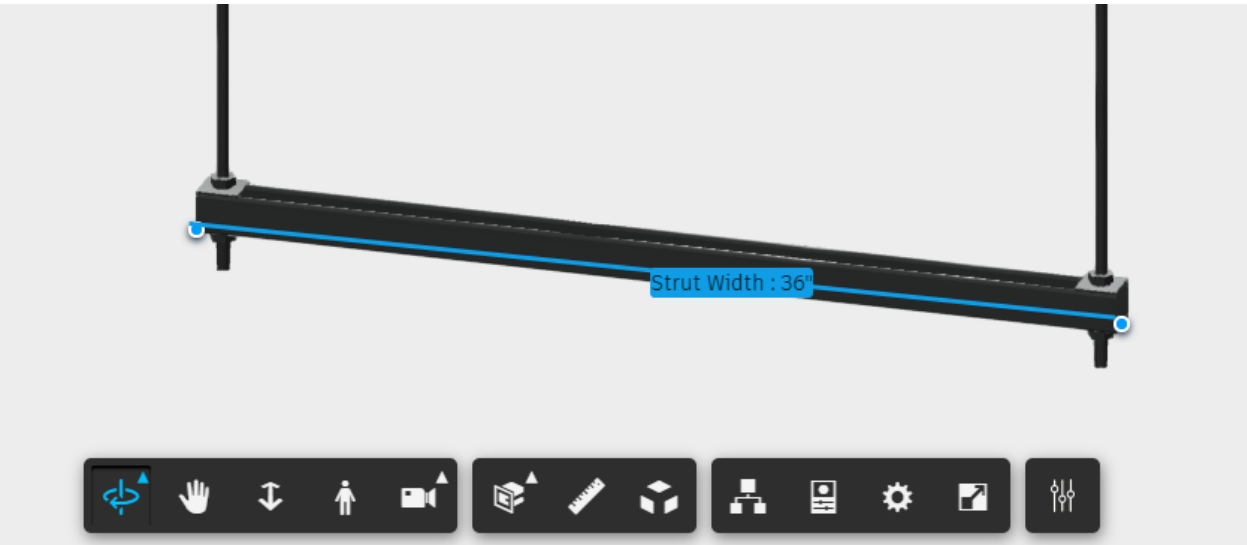
The source code for this extension can be found here: https://github.com/mossherder/au2019-configurator/blob/master/app/static/js/src/PinchZoomFixExtension.js

Default Common Dimension

Certain products are sold by many companies in the industry, and the specification is defined in different ways. A 12” width hanger for one company can be the strut length, but for another company, it could be the distance between the threaded rods.

Every time a user would begin to look at our hangers, this question would come up. So, we developed a common dimension system to show a default common dimension.

Below we can see the resulting functionality:



When we change views, the isolated model is analyzed, and the dimension is updated to show the length of this new configuration component.

We start by creating an extension. We'll want to hook into the ISOLATE_EVENT and the CAMERA_CHANGE_EVENT. Both are critical to show the dimension and then update the dimension as the model is viewed.

```
class DefaultDimensionExtension extends Autodesk.Viewing.Extension {
  constructor(viewer, options) {
    super(viewer, options);
    this.viewer = viewer;
    this.options = options;
    this.currentIsolatedElement = null;
    this.defaultDimensionElementName = 'EX_Sub-Unistrut';
    this.defaultDimensionParameterName = 'Strut Width'
    this.configuratorTagId = 'configurator-tag';
    this.configuratorTagClass = 'configurator-tag-item';
    this.configuratorTagEndClass = 'configurator-tag-end';
    this.configuratorTagEndOneId = 'configurator-tag-end-one';
    this.configuratorTagEndTwoId = 'configurator-tag-end-two';
    this.configuratorTagLineId = 'configurator-tag-line';
    this.forgeViewerId = 'forge-viewer';
  }

  load() {
```

```
      this.viewer.addEventListener(Autodesk.Viewing.ISOLATE_EVENT, (event) => {
        this.handleIsolationChange(event);
      });
      this.viewer.addEventListener(Autodesk.Viewing.CAMERA_CHANGE_EVENT, (event)
=> {
        this.handleCameraChange(event);
      });
      return true;
  }

  unload() {
    return true;
  }

  static get ExtensionId() {
    return 'Configurator.Extension.DefaultDimension';
  }
}

Autodesk.Viewing.theExtensionManager.registerExtension(
    DefaultDimensionExtension.ExtensionId,
    DefaultDimensionExtension
);
```

The dimension is made up of three different elements. The two endpoint elements of the dimension and the dimension line that spans between the two endpoints. The dimension elements are created when the dimension is first created, then they are referenced by looking the elements up in the DOM. To create the dimension HTML elements we use JQuery, and we create them using the below method. In the next sections we'll see how to properly position these dimension elements.

```
  createTags() {
    const configuratorTagEndClasses = [
      this.configuratorTagEndClass,
      this.configuratorTagClass,
      'measure-label-icon'
    ].join(' ');

    const tag = $('<div>', {
      id: this.configuratorTagId,
      class: this.configuratorTagClass
    });
    const tagEnd1 = $('<div>', {
```

```
    class: configuratorTagEndClasses,
    id: this.configuratorTagEndOneId
  });
  const tagEnd2 = $('<div>', {
    class: configuratorTagEndClasses,
    id: this.configuratorTagEndTwoId
  });
  const tagLine = $('<div>', {
    id: this.configuratorTagLineId,
    class: 'measurement-selection-area ' + this.configuratorTagClass
  });
  // Adding tags to the DOM
  tag.appendTo($('#' + this.forgeViewerId).children());
  tagEnd1.appendTo($('#' + this.forgeViewerId).children());
  tagEnd2.appendTo($('#' + this.forgeViewerId).children());
  tagLine.appendTo($('#' + this.forgeViewerId).children());
}
```

One interesting point to note is that we use the ‘measure-label-icon’ class to style our dimension the same way the Forge Viewer measure extension styles dimensions, to get the same look and feel.

A quick note on using HTML elements for annotations: When using HTML elements as annotations in the Viewer, we should be careful to not degrade performance. If we need to display thousands of annotations, then we should use a point cloud or THREE.js geometry-based annotation. For this use case, we’ll only have at most one dimension at a time, so we will have excellent performance.

Now, we’ll take a look at how we position the dimension.

Since we need to analyze the geometry of the isolated element, to know where to draw and redraw the dimension, we’ll need to access the underlying geometry of the Viewer model by using THREE.js, an underlying graphics API library.

First, we can look at the utility method to retrieve vertices from Fragment Ids. The code for this utility is mainly sourced from a Forge Team blog post located here: https://forge.autodesk.com/cloud_and_mobile/2015/07/accessing-mesh-information-with-the-view-data-api.html

The original code was used to draw vertices and edges of the geometry. We are just interested in the vertex locations, so we have this method gather and return only the vertices of the isolated geometry.

```
class DefaultDimensionExtension extends Autodesk.Viewing.Extension {

...

  getVerticesByFragId(fragId) {
    const fragProxy = this.viewer.impl.getFragmentProxy(this.viewer.model, fragId
);
    fragProxy.updateAnimTransform();
    const renderProxy = this.viewer.impl.getRenderProxy(this.viewer.model, fragId
);

    const matrix = new THREE.Matrix4();
    fragProxy.getWorldMatrix(matrix);

    const geometry = renderProxy.geometry;

    const attributes = geometry.attributes;

    const vA = new THREE.Vector3();
    const vB = new THREE.Vector3();
    const vC = new THREE.Vector3();

    const vertices = [];
    if (attributes.index !== undefined) {
      const indices = attributes.index.array || geometry.ib;
      const positions = geometry.vb ? geometry.vb : attributes.position.array;
      const stride = geometry.vb ? geometry.vbstride : 3;

      let offsets = geometry.offsets;
      if (!offsets || offsets.length === 0) {
        offsets = [{start: 0, count: indices.length, index: 0}];
      }

      for (let oi = 0, ol = offsets.length; oi < ol; ++oi) {

        const start = offsets[oi].start;
        const count = offsets[oi].count;
        const index = offsets[oi].index;

        for (let i = start, il = start + count; i < il; i += 3) {

          const a = index + indices[i];
```

```
        const b = index + indices[i + 1];
        const c = index + indices[i + 2];

        vA.fromArray(positions, a * stride);
        vB.fromArray(positions, b * stride);
        vC.fromArray(positions, c * stride);
        vA.applyMatrix4(matrix);
        vB.applyMatrix4(matrix);
        vC.applyMatrix4(matrix);
        vertices.push(vA.clone());
        vertices.push(vB.clone());
        vertices.push(vC.clone());
      }
    }

  }
  else {
    const positions = geometry.vb ? geometry.vb : attributes.position.array;
    const stride = geometry.vb ? geometry.vbstride : 3;

    for (let i = 0, j = 0, il = positions.length; i < il; i += 3, j += 9) {

      const a = i;
      const b = i + 1;
      const c = i + 2;

      vA.fromArray(positions, a * stride);
      vB.fromArray(positions, b * stride);
      vC.fromArray(positions, c * stride);
      vA.applyMatrix4(matrix);
      vB.applyMatrix4(matrix);
      vC.applyMatrix4(matrix);
      vertices.push(vA.clone());
      vertices.push(vB.clone());
      vertices.push(vC.clone());
    }
  }
  return vertices;
}
```

To get the opposite ends of the geometry, the points we want to dimension to, we use another brief utility that uses the utility we defined to retrieved the vertices:

```
  getElementEndsByDbIds(dbIds) {
    const fragIds = this.getFirstFragIdsByDbIds(dbIds);

    // We start by retrieving all the element end face geometry vertices.
    const elementEndVertices = fragIds.map((fragId) => {return this.getVerticesBy
FragId(fragId);})[0];
    let vertexEndOne;
    let vertexEndTwo;
    // Using all the end vertices of the geometry, we'll get two vertices that
    // are furthest away from each other.
    // This guarantees that the two vertices we filter to are on opposite ends
    // of the geometry.
    elementEndVertices.forEach((vertex, index) => {
      if (vertex.equals(new THREE.Vector3())) {
        return;
      } else {
        if (index == 0) {
          vertexEndOne = vertex;
          vertexEndTwo = vertex;
        } else {
          if (vertex.distanceTo(vertexEndOne) > vertexEndTwo.distanceTo(vertexEnd
One)) {
            vertexEndTwo = vertex;
          }
        }
      }
    });
    // Lastly, we set the two end vertices to have the same z value, as that
    // will give us a straight dimension across the strut.
    return [
      vertexEndOne,
      new THREE.Vector3(vertexEndTwo.x, vertexEndTwo.y, vertexEndOne.z)
    ];
  }
```

Great, so now we have the two vertices which define where we are drawing our dimension (opposite ends of the strut of the hanger, in this example).

We have our dimension tags created, we have the locations where the dimension needs to be positioned, so lastly, we need to convert these locations to their respective locations on the web browser viewport to finish positioning our dimension.

```
  updateTagPositions(dbIds) {
    // We retrieve tags that are created once and then stored in the class
    const tagOne = this.currentTagEndOne();
    const tagTwo = this.currentTagEndTwo();
    const tagLabel = this.currentTag();
    const tagLine = this.currentTagLine();

    // Gather the end vertices
    const tagEndLocations = this.getElementEndsByDbIds(dbIds);
    // Get the midpoint of the two vertices as the center of the dimension line
    const locationCenter = this.vectorMidpoint(tagEndLocations[0], tagEndLocation
s[1]);

    const centerCanvasCoordinates = this.viewer.worldToClient(locationCenter);
    // The points in 3D Forge Viewer space are transformed into 2D points in
    // the web browser canvas space
    const locationOneCanvasCoordinates = this.viewer.worldToClient(tagEndLocation
s[0]);
    const locationTwoCanvasCoordinates = this.viewer.worldToClient(tagEndLocation
s[1]);

    // Some fancy math is used to calculate how long the dimension line is
    // in the 2D canvas space, and the rotation of the line based on the
    // rotation of the model
    const lineLength = Math.hypot(locationTwoCanvasCoordinates.y - locationOneCan
vasCoordinates.y, locationTwoCanvasCoordinates.x - locationOneCanvasCoordinates.x
);
    const lineDegrees = Math.atan2(locationTwoCanvasCoordinates.y - locationOneCa
nvasCoordinates.y, locationTwoCanvasCoordinates.x - locationOneCanvasCoordinates.
x) * (180 / Math.PI);

    // All our gathered geometric information is set to the dimension
    // HTML elements using CSS. These calls are available using JQuery
    tagLabel.css('top', centerCanvasCoordinates.y);
    tagLabel.css('left', centerCanvasCoordinates.x);
    tagOne.css('top', locationOneCanvasCoordinates.y);
    tagOne.css('left', locationOneCanvasCoordinates.x);
    tagTwo.css('top', locationTwoCanvasCoordinates.y);
```

```
    tagTwo.css('left', locationTwoCanvasCoordinates.x);
    tagLine.css('top', locationOneCanvasCoordinates.y);
    tagLine.css('left', locationOneCanvasCoordinates.x);
    tagLine.css('transform', 'rotate(' + lineDegrees + 'deg)');
    tagLine.css('transform-origin', '0 7.5px');
    tagLine.css('width', lineLength + 'px');
    tagLine.css('height', '3px');
    return true;
  }
```

And now we'll have the dimension automatically applied when we configure a hanger.

The source code for this extension can be found here: https://github.com/mossherder/au2019-configurator/blob/master/app/static/js/src/DefaultDimensionExtension.js

Understanding Current Challenges in Flexing Models and Options to Enhance Behavior

Currently, there is no parametric engine available with the Forge Viewer. Therefore, we don't have the ability to 'flex' geometry. When our configuration calls for a 1 foot length, and then it is changed to a 2 foot length, we don't have a general and reliable way to apply the geometry change to our Forge Viewer model. To overcome this inability to natively flex the model geometry, we'll show an easy process in the Forge Viewer which mimics the behavior.

We'll also show a powerful technique to enhance the behavior of our configurator. With the Forge Viewer, we have access to the THREE.js library. A lightweight, 3D library, to manipulate (or add to) the geometry of the model loaded in the Viewer. Using this library, we will show how to duplicate geometry to create 'packs' of a product.

Process of 'Flexing Geometry'

One issue when we're making a configurator, is that we can't natively, parametrically flex the geometry in the Viewer.

Due to this issue, the best way we have found so far is to have the different geometries all placed in the model, and then when we configure the model, we just hide, and isolate according to the new configuration.

When creating a model with Revit for configurations with thousands of different choices, we can't place all the family instances in the same location, or we get performance issues in our Revit model. Therefore, we place the family instances in rows and line all of them up.

Now, when we configure this model in the Viewer, we will have an issue if we only implement the extension to the point where we did earlier in this handout. The issue is that we will see the object moving along the screen as it is being fitted to the view. This may be okay depending on the user experience you're looking for, but for our use case we really wanted a configurator where the user experience would feel like they were changing the same object into different specifications. To this end, we add a slightly more complex process when we change a configuration:

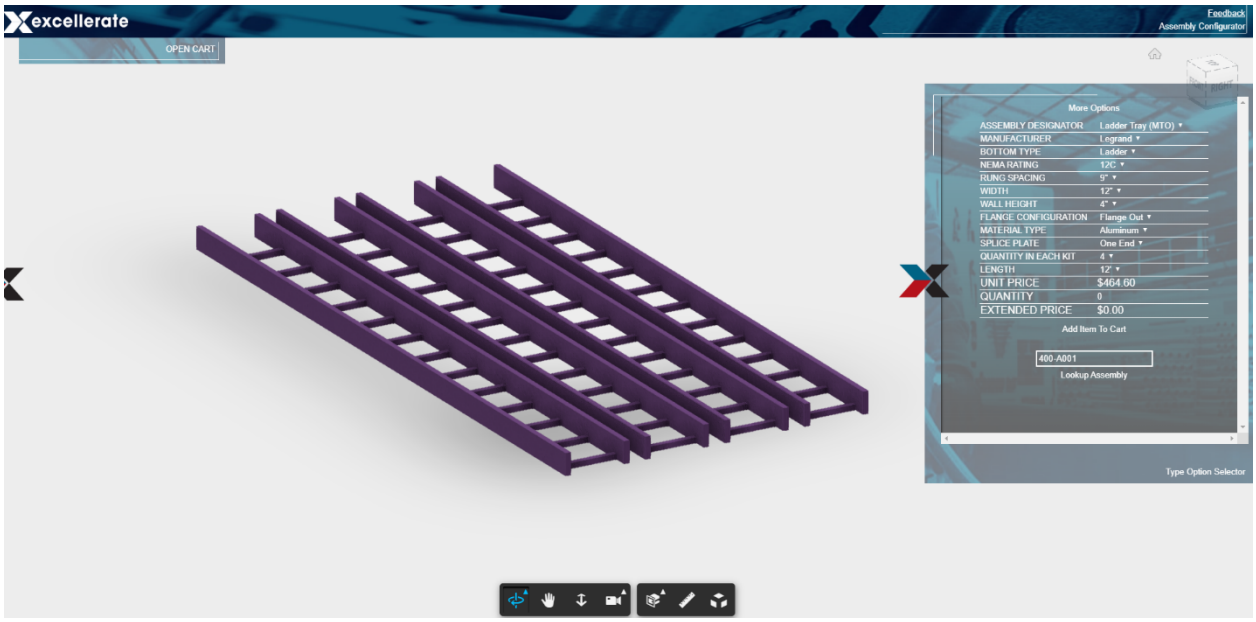
```
    // by isolating a non-existent dbId, we essentially hide all dbIds
    this.viewer.impl.visibilityManager.isolate(
         [99999999],
         this.viewer.model
     );

     // fitToView gets the camera to beginning flying over to where it should
     // be to display the new configuration
     this.viewer.fitToView(dbIds, this.viewer.model);

     // Finally we wait to isolate until the camera is finished transitioning
```

```
    // to the new configuration
    this.viewer.addEventListener(Autodesk.Viewing.CAMERA_TRANSITION_COMPLETED,
     () => {
        this.viewer.impl.visibilityManager.isolate(dbIds, this.viewer.model);
      },
      {'once': true}
    );
```

Until there is a parametric engine in the Forge Viewer environment, we'll have to find workarounds. For now, this process works very well and gives a good feeling to the user that an object is being configured.

Using THREE.js to add Packs of a Product

As a final example, one of our products, Cable Tray, is sold in packs. The user can order a pack of four, five, or six, 10' or 20' length Cable Tray for one of the product offerings. One option is to create a Revit model that has the various packs of Cable Tray all placed, and the ExternalId looked up based on the Assembly Code configured would instead be a list of ExternalIds, which would then translate into multiple dbIds that are shown (with these dbIds each corresponding to a different tray in the pack).



Another option is to duplicate the geometry of the Cable Tray using THREE.js and then dynamically create the number of trays needed to represent the pack.

Dynamically creating the needed number of tray would keep our Revit models smaller, so we chose this option.

```
      const isolatedNodes = this.viewer.getIsolatedNodes();
      isolatedNodes.forEach((dbId) => {
        const fragIds = this.getFragIds(dbId);
        const meshWidth = this.getWidthOfMesh(fragIds);
        fragIds.forEach((fragId) => {
          for (let i = 0; i < qty - 1; i++) {
            const currentMesh = this.viewer.impl.getRenderProxy(this.viewer.model
, fragId);
            let newMesh = currentMesh.clone();
            newMesh.applyMatrix(currentMesh.matrixWorld);
            newMesh.translateY((meshWidth + 2.0/12.0) * (i + 1)); // buffer of 2"
            this.viewer.impl.scene.add(newMesh);
            clones.push(newMesh);
          }
        });
      });
      this.viewer.impl.sceneUpdated();
```

Using the Render Proxy of the Fragment Id of the element in the model (one specific tray), we can access the underlying geometry of the model. This render proxy is a THREE.Mesh object, and one of the methods available to the Mesh object is the ability to clone the mesh.

We clone the mesh, move the clone over, and then add the clone to the Forge Viewer scene. After we finish adding the new meshes (Cable Tray clones) to the scene, we use the sceneUpdated method to refresh the Forge Viewer with the new geometry. The user will now see the whole pack of Cable Tray represented in the configuration.

One thing to note when adding THREE.js geometry to the Viewer is we must track the created geometry in order to delete it when a new option is configured.

The source code for this configurator application can be found here:
https://github.com/mossherder/au2019-configurator/