SD21092

Visual Reporting with Connected Design Data

Philippe Leefsma
Forge Partner Development

Learning Objectives

- Access and manipulate the data which is available out of the box with your Forge Viewer models
- Explore various way of overlaying and embedding custom graphics into the Viewer
- Learn how to create powerful, highly customized visual applications integrating elements from multiple data sources
- Leverage the Forge Model Derivative API to access and manipulate design data in the cloud

Description

Viewing 3D models on a webpage or in a mobile App is great, but visualizing design data isn't just about meshes and textures

In this class I will show you how to use JavaScript to turn the Forge Viewer into a powerful visual reporting tool, giving you access to valuable data associated with each model extracted from the original design file and connect your viewing application to external databases &Web APIs to add rich visualization experience

Basic Web development experience is preferable - HTML5 JavaScript, CSS, Web Services, REST API's, …

Your AU Expert

Philippe has been working for the last 10 years at the Autodesk Developer Network, nowadays renamed Forge Partner Development. He writes JavaScript for the World Wide Web around the Autodesk Forge Web Services, helping partners and customers who migrate or evolve in the Cloud. He is sharing his experiences through devblog articles and on Twitter: @F3lipek

Technology Stack

The Forge Viewer allows you to create custom web applications that enable your customers to visualize and interact with 2D and 3D design data in a web browser or on a mobile device, anytime, anywhere. No downloads or plug-ins required. Forge Viewer support over 60 file formats to bring you stunning, detailed, high-fidelity visualizations of models - right within your browser experience. Our REST and JavaScript APIs make it easy for you to quickly develop web applications.

Here are the main features

- REST and JavaScript API make it easier to create applications.
- Upload your model via our REST API, and our web service creates the viewing stream for you.
- Our JavaScript API provides access to model and model components data in your web app - no need for desktop CAD software installations.
- 2D and 3D models come to life right within your browser - view detailed textures, vivid visualizations, smooth navigation, and in-depth design data.
- Browser compatibility with Internet Explorer, Edge, Chrome, Safari, Firefox and Opera.
- Visualize and interact with more than 60 design file formats in your web app; no plug-ins or downloads required.

File management REST API

The REST file management API's allows you to upload your CAD models to the Autodesk Cloud. It requires OAuth 2.0 client authentication and returns JSON formatted responses.

For a step by step description of the exact workflow required to upload and manage your files, please refer to our developer documentation at https://developer.autodesk.com

Client Side Technology

The client side viewer is a zero-client browser-based JavaScript API that allows you to load and embed your models in any webpage. It requires a WebGL capable browser and is implemented on top of WebGL and three.js API.

I - Accessing Forge Design Data

Accessing the viewer API

Here is a basic pseudo code sample that illustrates how easy it is to instantiate a viewer, all you need is an empty container div and a model Id or URN, then the API will take care about initializing the WebGL canvas and render the model:

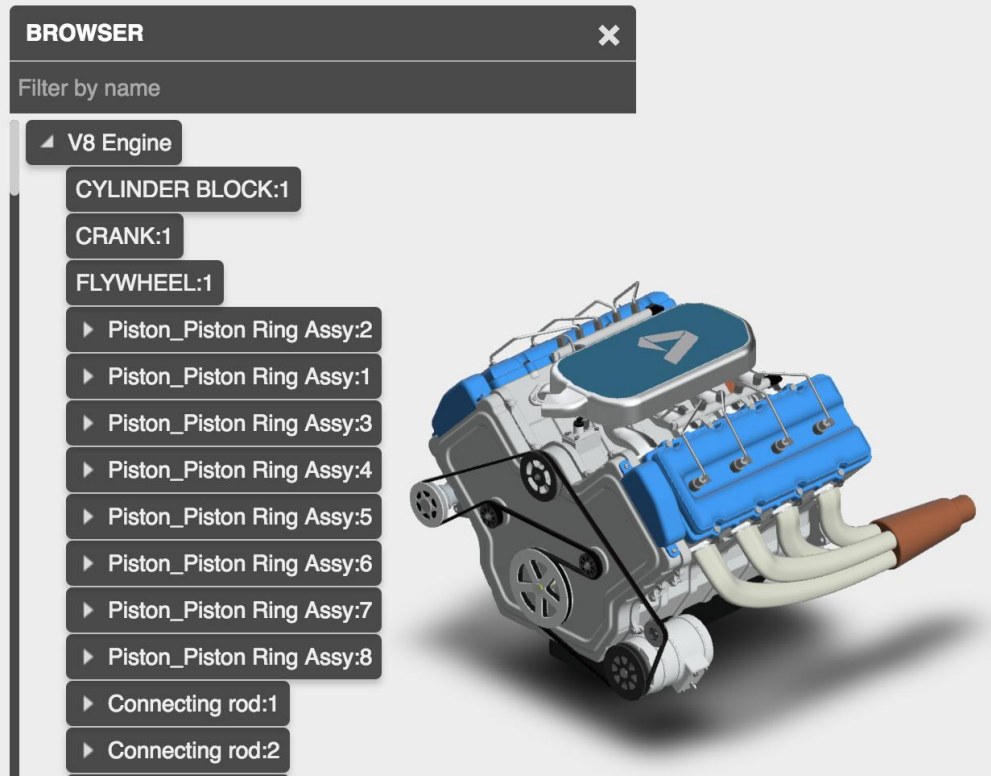
```
html:
<div id='viewer'> </div>

js:

  loadDocument(urn, ..., function(doc) {

    var viewerContainer = document.getElementById('viewer')

    var viewer = Autodesk.Viewing.Viewer3D(viewerContainer)

    viewer.loadModel(getViewablePath(doc))

    // call API's

    viewer.setBackgroundColor( ... )

    viewer.loadExtension('MyExtId')

    // ...
  })
```

Detailed explanations about uploading a design and translate it for the viewer can be found on the API portal documentation:
https://developer.autodesk.com/en/docs/viewer/v2/tutorials/basic-viewer

Viewer Component Model

Here is a simple component model that we can use to represent each selectable component in a translated design:

```
{
  name: 'chassis'          //display name of the component
  dbId: 53                 //unique id for the component in the model
  fragIds: [38, 39]        //reference the three.js meshes
  parent: 37               //dbId of the parent node
  children: [65, 113, 146] //array of children nodes ids
}
```



Components representing the design hierarchy in ModelStructure Panel

Model Structure

The two following snippets illustrate how to recursively iterate the instance tree for the root node and generate the hierarchy for the whole model:

```
1  function buildModelTree (model) {
2
3    var instanceTree = model.getData().instanceTree
4
5    var rootId = instanceTree.getRootId()
6
7    var rootNode = {
8      dbId: rootId,
9      name: instanceTree.getNodeName(rootId)
10   }
11
12   buildModelTreeRec(rootNode)
13
14   return rootNode
15 }
```

```
function buildModelTreeRec (node) {

    instanceTree.enumNodeChildren (node.dbId, function (childId) {

        node.children = node.children || []

        var childNode = {
            dbId: childId,
            name: instanceTree.getNodeName(childId)
        }

        node.children.push(childNode)

        buildModelTreeRec(childNode)
    })
}
```

Complete code sample can be found on my extensions repository: [ModelStructure Extension]

Node Fragments

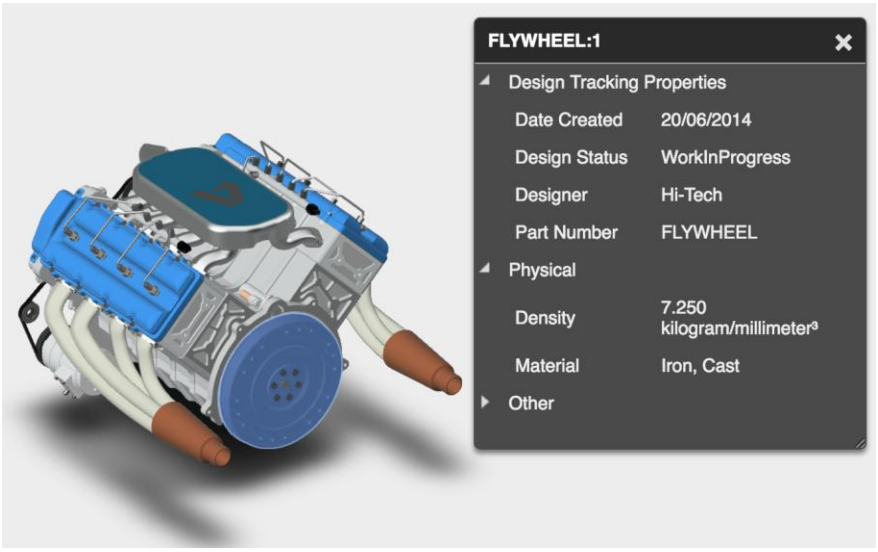
Fragments represent each mesh created in the model, one node can be made of one or several fragments, the snippet below illustrates how to iterate fragments for a specific nodeId:

```
var instanceTree = model.getData().instanceTree

var fragIds = []

instanceTree.enumNodeFragments(dbId, function (fragId) {

    fragIds.push(fragId)
})
```

Component Properties

Each component comes along with specific properties attached to it, they differ depending from which native file format was generated the design. The following snippet illustrates how to access properties for a specific nodeId:

```
model.getProperties(dbId, function(result) {

    if (result.properties){

        result.properties.forEach( function (prop) {

            console.log(prop)
        })
    }
}
```



Properties are accessible in the viewer through the property panel

You can check a complete sample extension form there: [PropertyDump Extension]

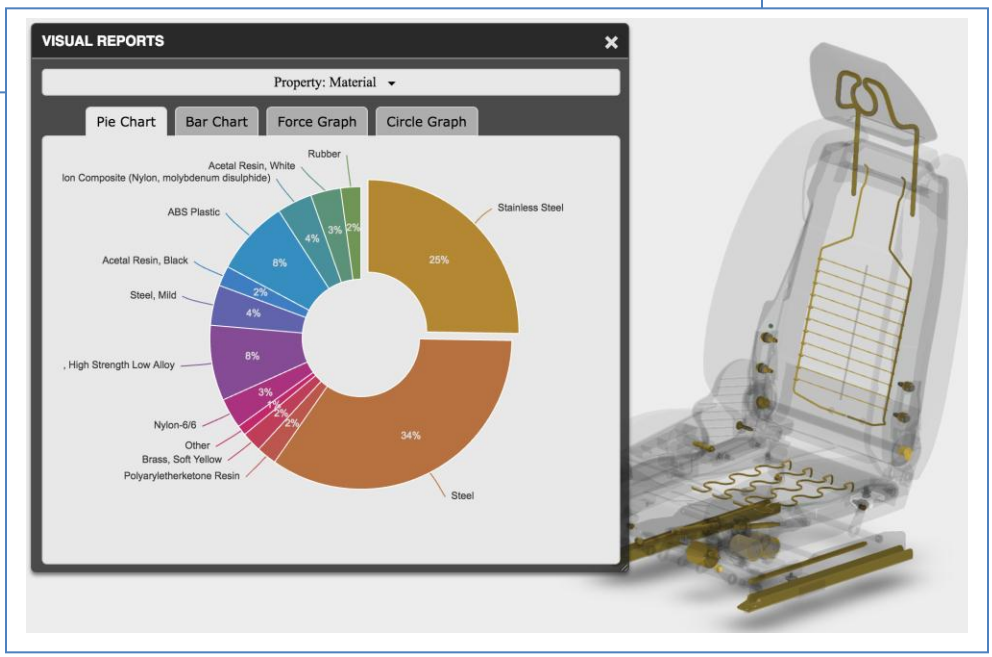
II - UI Customization

2D Overlays

The viewer is simply a WebGL 3D ***canvas*** element contained inside a ***div***, so it means that with html and CSS you can overlay any custom 2D element you want on top of it, then control them or hook them up to viewer events using JavaScript. This makes UI customization pretty flexible and powerful, letting you the capabilities to overlay any custom data on top of the models.

Custom Panel

```
1  class CustomPanel extends Autodesk.Viewing.UI.DockingPanel {
2
3   constructor(container, title, options = {}) {
4
5    super(container, panelId, title, options)
6
7    this.container.appendChild(...)
8   }
9
10 }
```



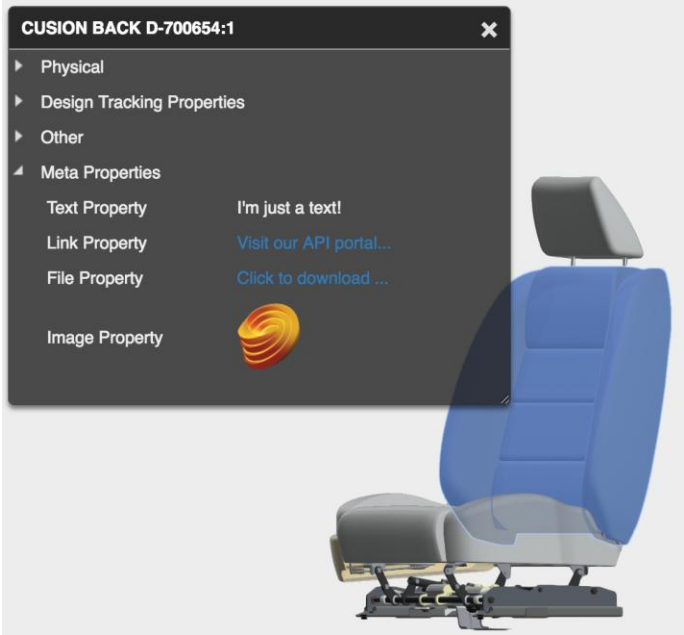
The ***DockingPanel*** is a component provided by the viewer API, it is the base class used to display information inside the viewer and interact with the settings. You can reuse it and extend it in your own applications to provide seamless integration experience to the user:

DockingPanel Extension

Custom PropertyPanel

The ***ViewerPropertyPanel*** extends the previous base ***DockingPanel***. It is used by the viewer to display selected component properties. You can extend it to inject or replace original properties by custom properties.

```
class CustomPropertyPanel extends Autodesk.Viewing.UI.ViewerPropertyPanel {

  constructor(viewer) {

    super(viewer)
  }

  setProperties (properties) {

    properties.push({ ... custom property ... })

  super.setProperties(properties)
  }
}
```



MetaProperties Extension panel

Following samples illustrate how to
Extend and customize the property panel:

PropertyPanel Extension

MetaProperties Extension

Fragment Overrides

We introduced before how to access fragmentId's associated with each node. Here is what they are useful for: they can be used to access the actual Three.js meshes in order to apply different material to them or transform the components:

```
//current model
var model = viewer.model

//create custom material
var material = new THREE.MeshPhongMaterial({
    color: '#F43BC1'
  //... other properties
})

//set material on specific fragId
model.getFragmentList().setMaterial(
    fragId, material)

//force viewer to update scene
viewer.impl.invalidate(true)
```

```
//access render proxy
var renderProxy = viewer.impl.getRenderProxy(
    model, fragId)

//clone geometry
var meshProxy = new THREE.Mesh(
    renderProxy.geometry)

meshProxy.matrix.copy(
    renderProxy.matrixWorld)

//create 3d overlay
viewer.impl.addOverlay(
    materialName, meshProxy)

// force update
viewer.impl.invalidate(true)
```

```
// access fragment proxy i.e. THREE.Mesh
var fragProxy = viewer.impl.getFragmentProxy(
    model, fragId)

fragProxy.getAnimTransform()

fragProxy.position = new THREE.Vector3(x, y, z)

//Not a standard three.js quaternion
fragProxy.quaternion._x = qx;
fragProxy.quaternion._y = qy;
fragProxy.quaternion._z = qz;
fragProxy.quaternion._w = qw;

fragProxy.updateAnimTransform()

viewer.impl.invalidate(true)
```

The following links contain samples that illustrate how to map component materials based on specific selected property. They also make use of the d3.js visualization library

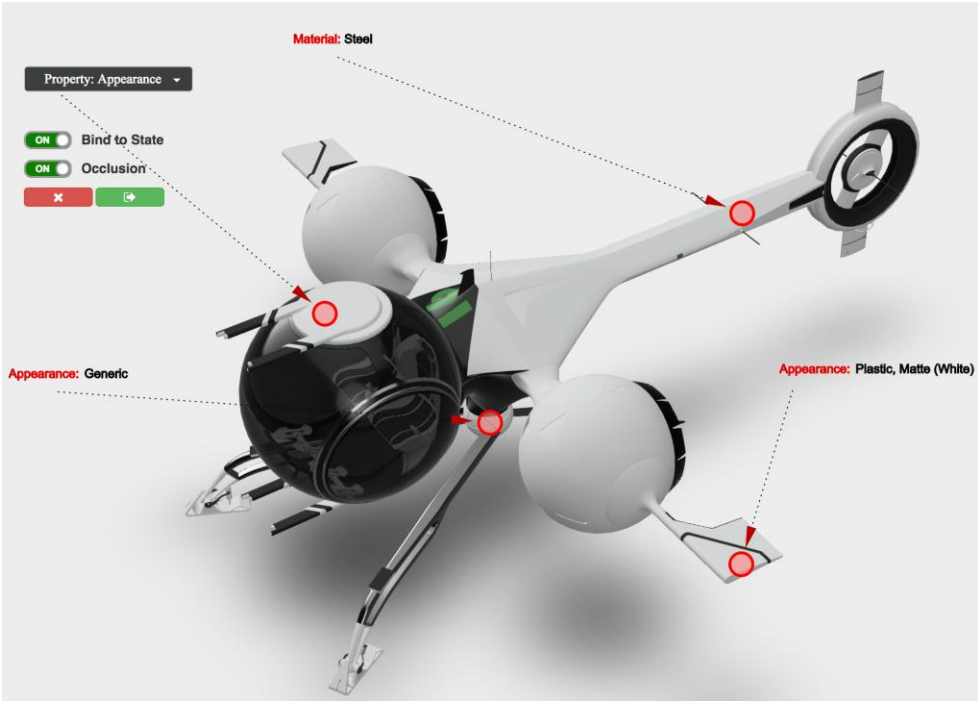
Visual Reporting

VisualReport Extension

SVG

Scalable Vector Graphics (SVG) is an XML-based markup language for describing two-dimensional vector graphics. SVG is essentially to graphics what HTML is to text.
(source: Mozilla Developer Network)

SVG can provide pretty powerful customization capabilities to overlay custom data on top of the viewer and hook them up to events.



Makup3D Extension

See those links for complete code samples and live demo:

Markup3D

Markup3D Extension

Coordinates Conversion

The previous Markup3D demo is using coordinate conversion to keep 2D markups in sync with the actual 3D point location on the model.

Here is how it can be achieved

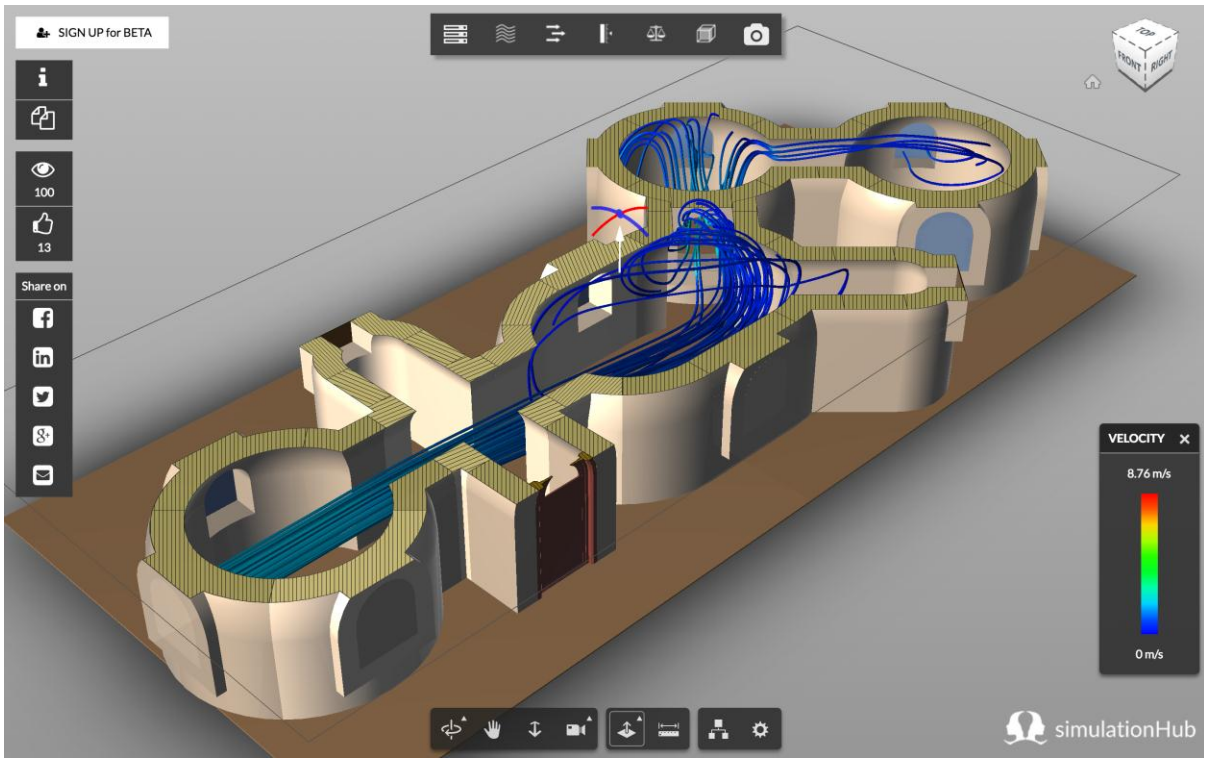
```
function screenToWorld (screenPoint) {

    var viewport = viewer.navigation.getScreenViewport()

    var n = {
        x: (screenPoint.x - viewport.left) / viewport.width,
        y: (screenPoint.y - viewport.top) / viewport.height
    }

    var worldPoint = viewer.utilities.getHitPoint(n.x, n.y)

    return worldPoint
}
```

```
function worldToScreen(worldPoint) {

  var p = new THREE.Vector4()

  p.x = worldPoint.x
  p.y = worldPoint.y
  p.z = worldPoint.z
  p.w = 1

  var camera = viewer.navigation.getCamera()

  p.applyMatrix4(camera.matrixWorldInverse)
  p.applyMatrix4(camera.projectionMatrix)

  var screenPoint = viewer.impl.viewportToClient(p.x, p.y)

  //snap pixel centre
  screenPoint.x = Math.floor(screenPoint.x) + 0.5
  screenPoint.y = Math.floor(screenPoint.y) + 0.5

  return screenPoint
}
```

3D Overlays & Custom meshes

Another technique that you may use to insert additional visual data into the viewer is to add your own 3D meshes to the scene:

```
var geometry = new THREE.SphereGeometry(size, 4, 4)

var mesh = new THREE.Mesh(
  geometry,
  material)

mesh.position.set(x, y, z)

viewer.impl.scene.add(mesh)
```



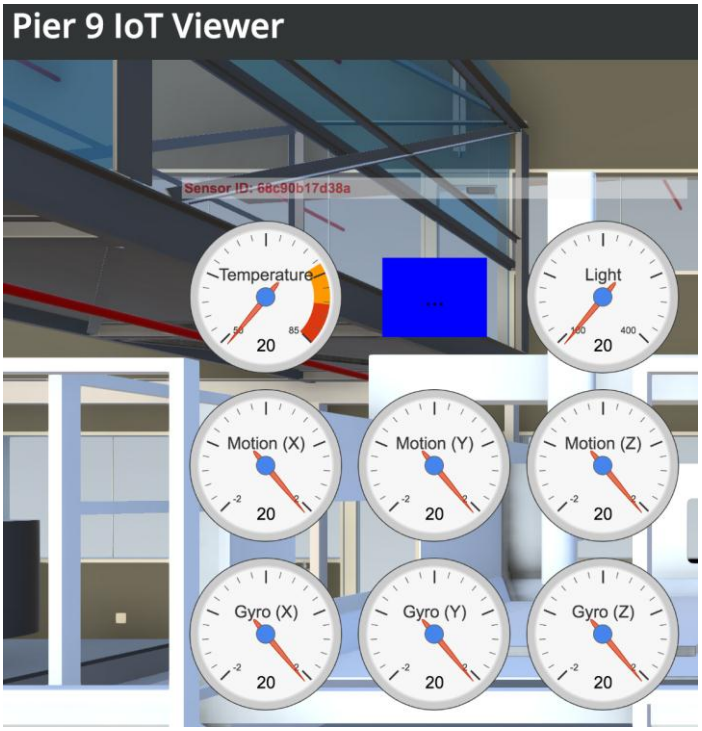
A very example of custom 3D meshes inserted in the viewer bringing useful information is the [Simulation Hub]

CSS3DRenderer & CSS3DObject

The CSS3DRenderer and CSS3DObject let you insert 2D content inside the 3D scene, providing some interesting customization capabilities

```
var cssRenderer = new THREE.CSS3DRenderer()

viewer.container.appendChild(
  cssRenderer.domElement)

var glScene = new THREE.Scene()

var iFrame = document.createElement('iframe')

var cssObj = new THREE.CSS3DObject(iFrame)

cssObj.position.set(x, y, z)

cssObj.scale.set(sx, sy, sz)

glScene.add(cssObj)
```

CSS3D demo: embedding 2D web page in the viewer



CSS3D demo: embedding 2d graphics in the viewer

III – Connecting your Application to the Cloud

RESTful API with Node.js & Express

Setting up REST routes in a Node.js/Express application on your server that can be called by your client application in order to exchange data is very easy

```
var router = express.Router()

router.get('/items/:id', function (req, res) {

  var item = getItemFromDatabase(id)

  res.json(item)
})

var app = express()

app.use('/api', router)

app.listen(process.env.PORT)
```

On the client side, you can make use of the new fetch API to easily request data from your server:

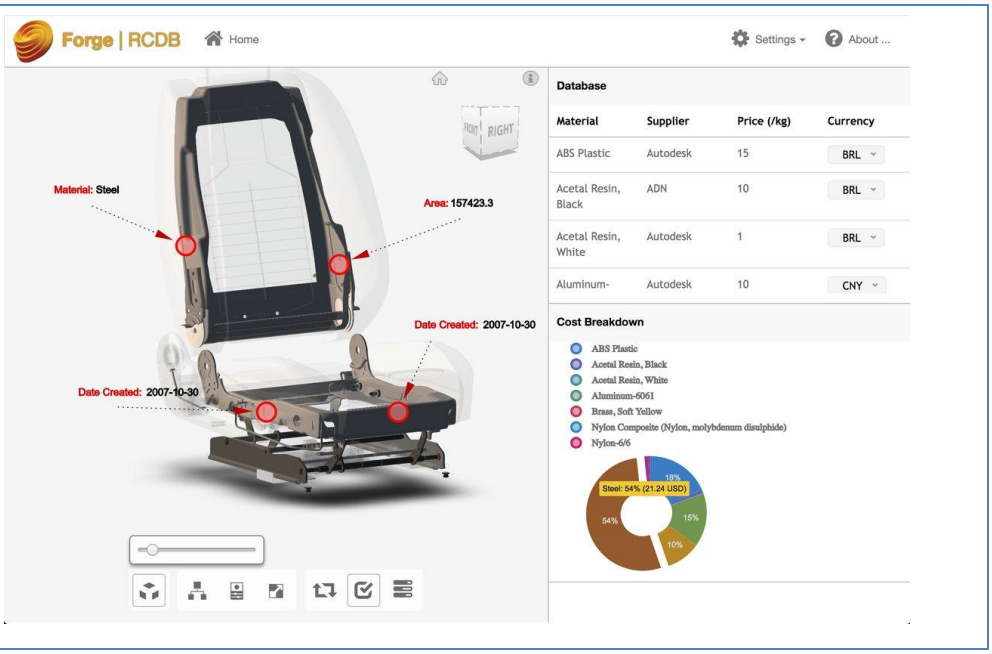
```
function getItem(id, onSuccess, onError) {

  fetch('/api/items/' + id).then(function (response) {

    response.json(function (item) {

    onSuccess(item)
    })

  }, function (error) {

    onError(error)
  })
}
```

The latest async/await JavaScript feature lets you write asynchronous code in a pretty neat way

```
async function getItem (id) {

  var response = await fetch('/api/items/' + id)

  var item = await response.json()

  return item
}

async function taskOnItems (itemIds) {

  const itemTasks = itemIds.map((id) => {

    return getItem(id)
  })

  const items = await Promise.all(itemTasks)

  //All items retrieved
  items.forEach((item) => {

      console.log('Item: ' + item.name)
  })
}
```

For a comprehensive sample using all those concepts, take a look at the following project:

https://github.com/Autodesk-Forge/forge-rcdb.nodejs



Resources

- Autodesk Forge Developer Portal
- This presentation Online
- Github repository for this presentation
- Github repository with Autodesk Forge samples (1)
- Github repository with Autodesk Forge samples (2)
- Library of viewer extension demos
- Forge RCDB repository on Github
- D3 Library