**Introduction to AWS IoT Twin Maker**

Organizations are now accumulating vast amounts of operational data, facilitated by advancements in technologies such as the Internet of Things (IoT), Artificial Intelligence (AI), and cloud computing. Industrial companies, in particular, are keen to leverage this data to create digital representations, or "twins," of their physical equipment and environments.

**Overview of AWS IoT TwinMaker**

AWS IoT TwinMaker is a service designed to streamline the process of creating digital twins, ultimately optimizing operational efficiency. Developers can utilize this service to amalgamate data from diverse sources including IoT sensors, video cameras, and business applications. The goal is to accurately model physical systems and provide an immersive 3D view into operations.

**Features and Capabilities**

With AWS IoT Twin Maker, users can seamlessly connect to various data sources, accurately model physical systems, and gain insights through an immersive 3D visualization. The service enables the creation of digital replicas of physical environments, enhancing monitoring capabilities and facilitating improvements in efficiency, production, and performance.

**Navigation and Visualization**

The 3D scene viewer allows users to navigate digital twins freely, while the hierarchy panel aids in selecting specific elements or areas of equipment. Tags are employed to bind data from multiple sources to equipment or physical systems, facilitating visualization of different data sources within the spatial context of the environment.

**Monitoring and Alarm Notifications**

Alarm notifications and history features are utilized to identify and address malfunctioning equipment promptly, thus minimizing downtime and enhancing operational efficiency. The digital twin serves as a comprehensive monitoring tool, enabling operators to detect and resolve issues effectively.

**Key Steps in Creating a Digital Twin**

The process of creating a digital twin with AWS IoT TwinMaker involves several key steps:

Defining components to connect to various data sources, utilizing built-in or custom data connectors.

Creating entities that represent physical equipment, processes, and spaces, and specifying relationships between them.

Uploading existing 3D models and visual assets to the resource library.

Using the scene composer to add 3D visual assets into a scene and position them to match the physical environment.

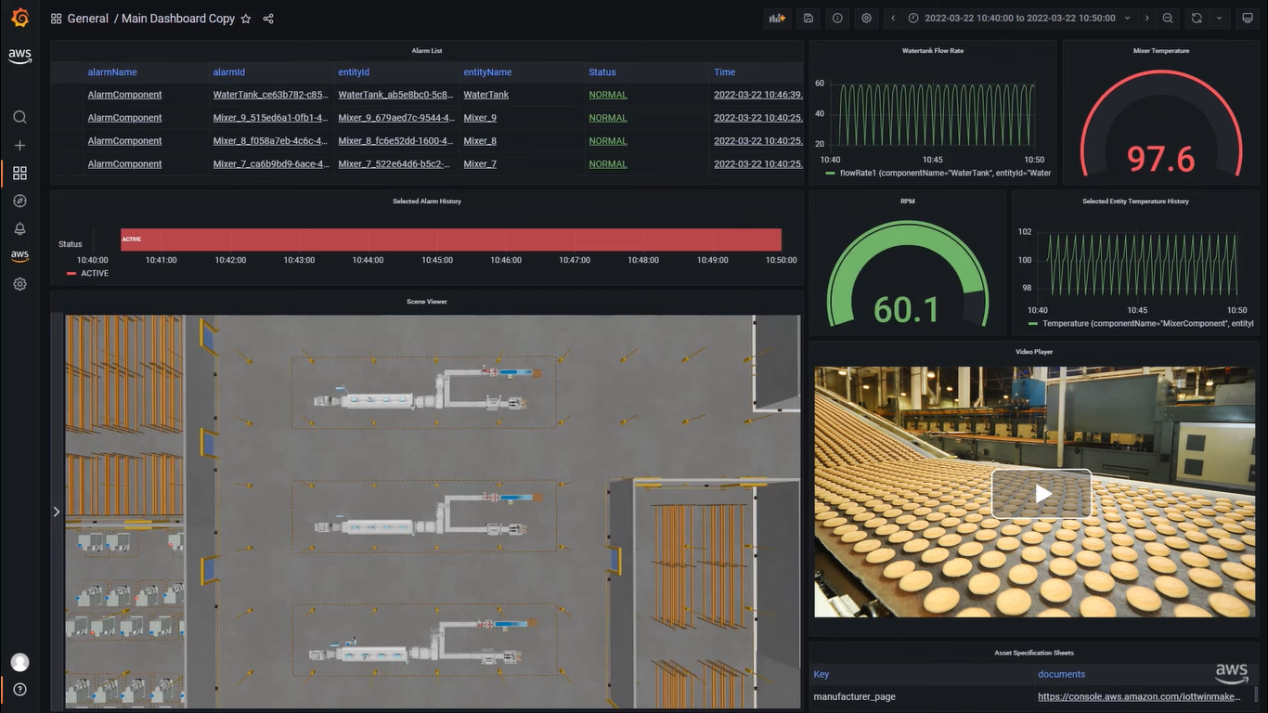
Binding real-world data from the digital twin graph to the visual assets.

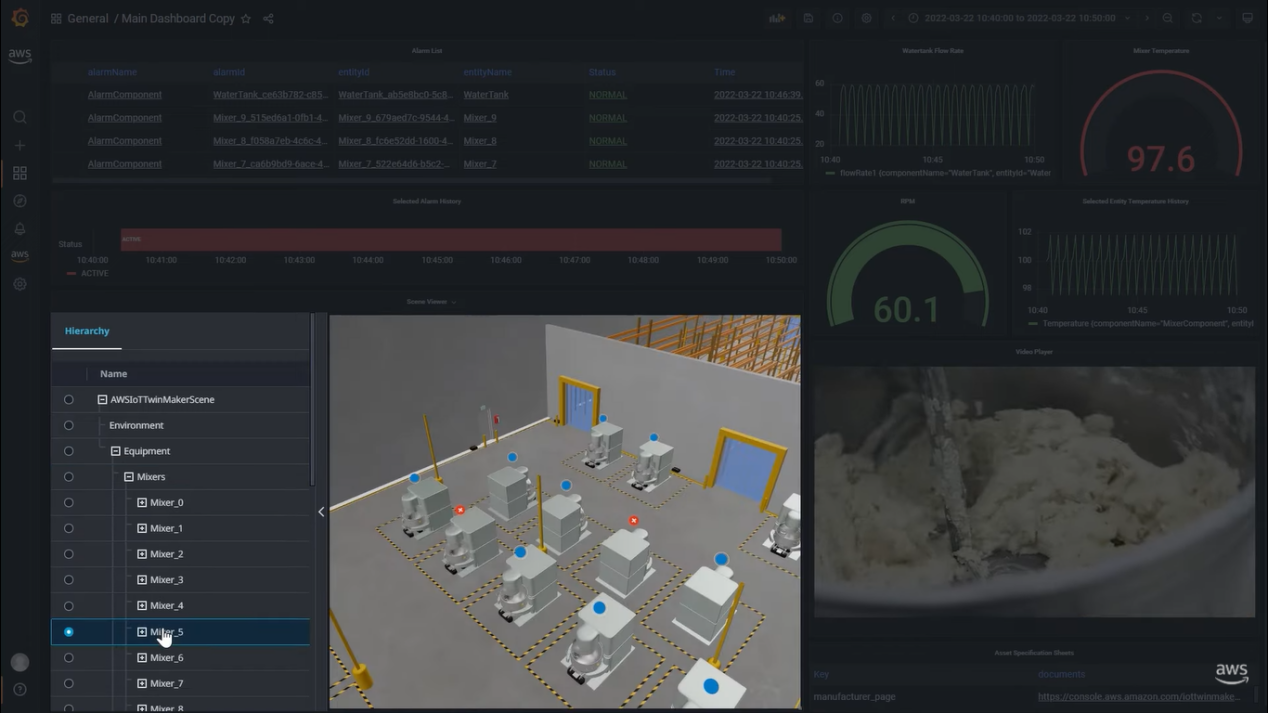
**Integration with Grafana**

AWS IoT TwinMaker seamlessly integrates with Grafana through a plugin, enabling the incorporation of digital twins into web applications. This integration empowers operators and engineers to monitor and optimize operations effectively using the digital twin.

**Conclusion**

AWS IoT TwinMaker offers a comprehensive solution for creating digital twins, facilitating operational optimization and reducing downtime. Organizations are encouraged to leverage this service to enhance efficiency and gain actionable insights into their operations.



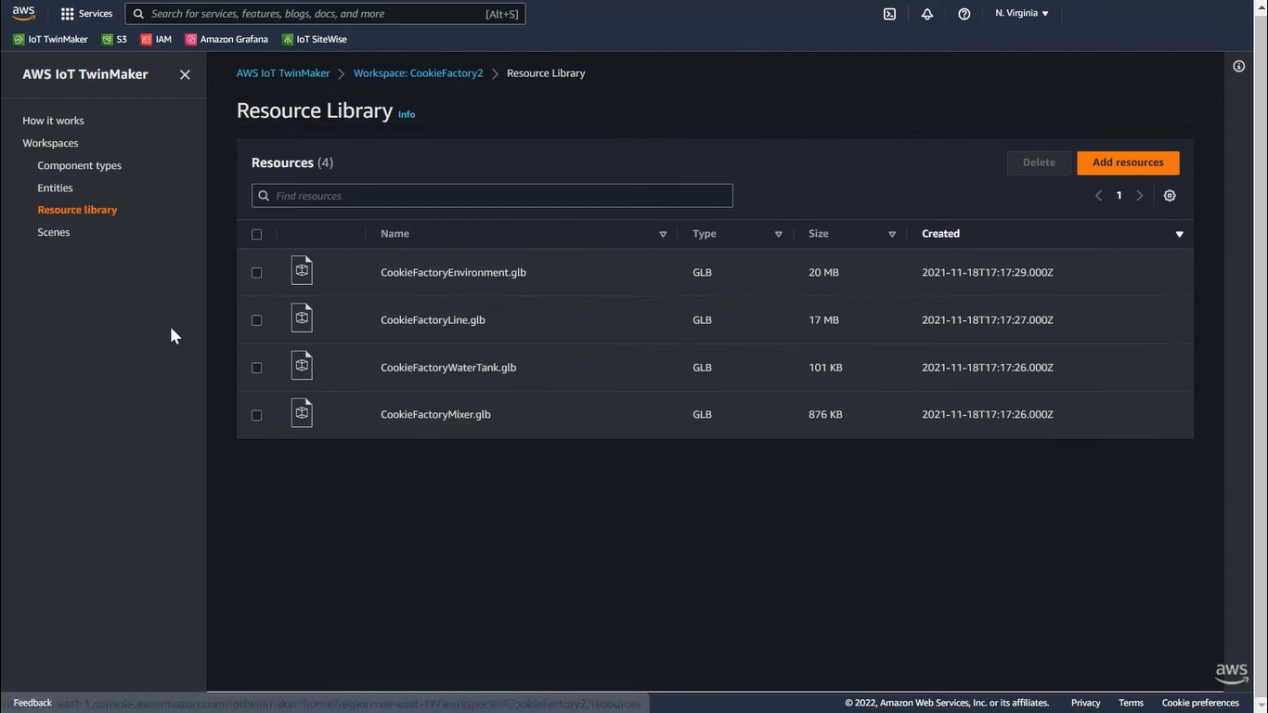
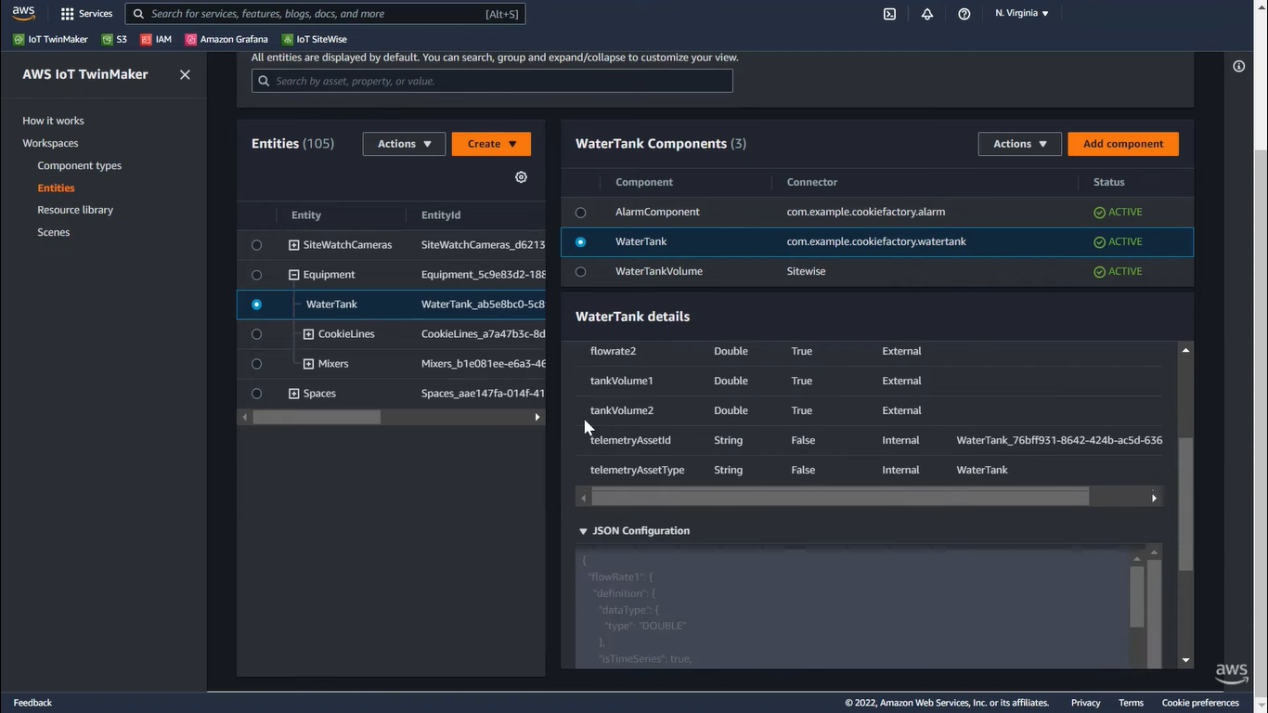
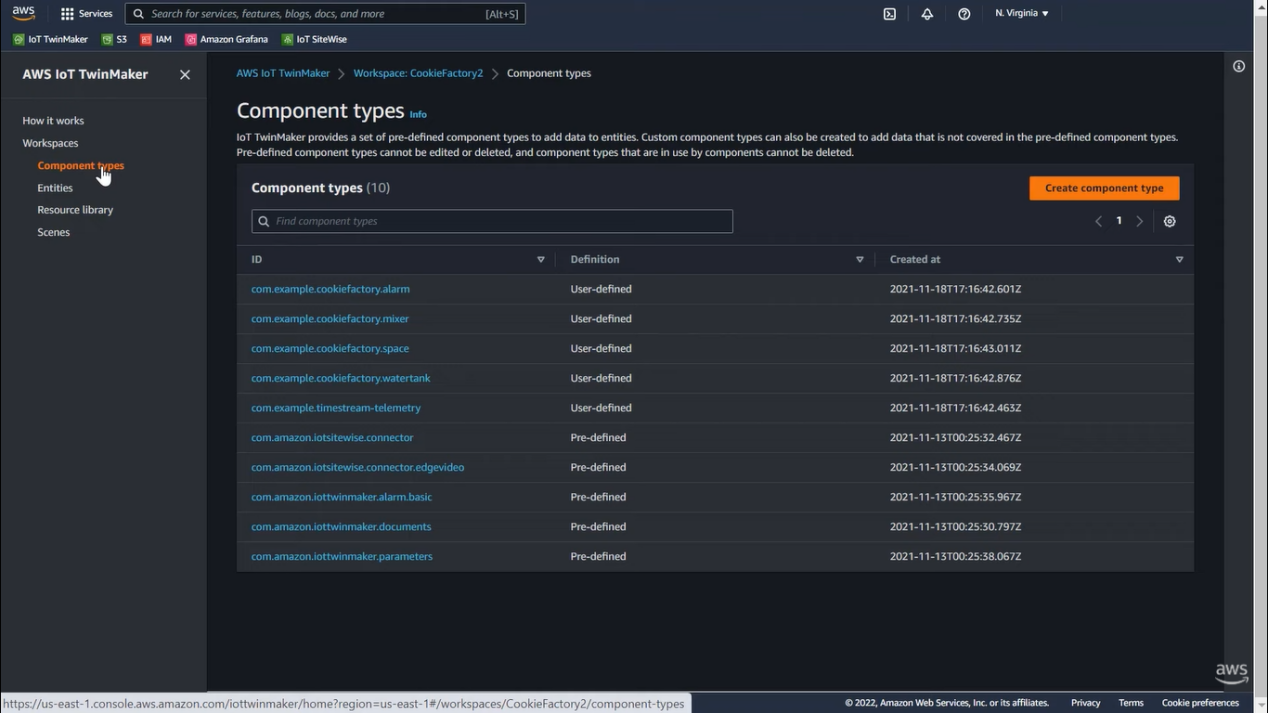


Components: These are elements used to connect to various data sources. They include built-in data connectors for accessing sensor data from AWS IoT SiteWise, video data from Amazon Kinesis video streams, and custom data connectors for other sources like Amazon Timestream.

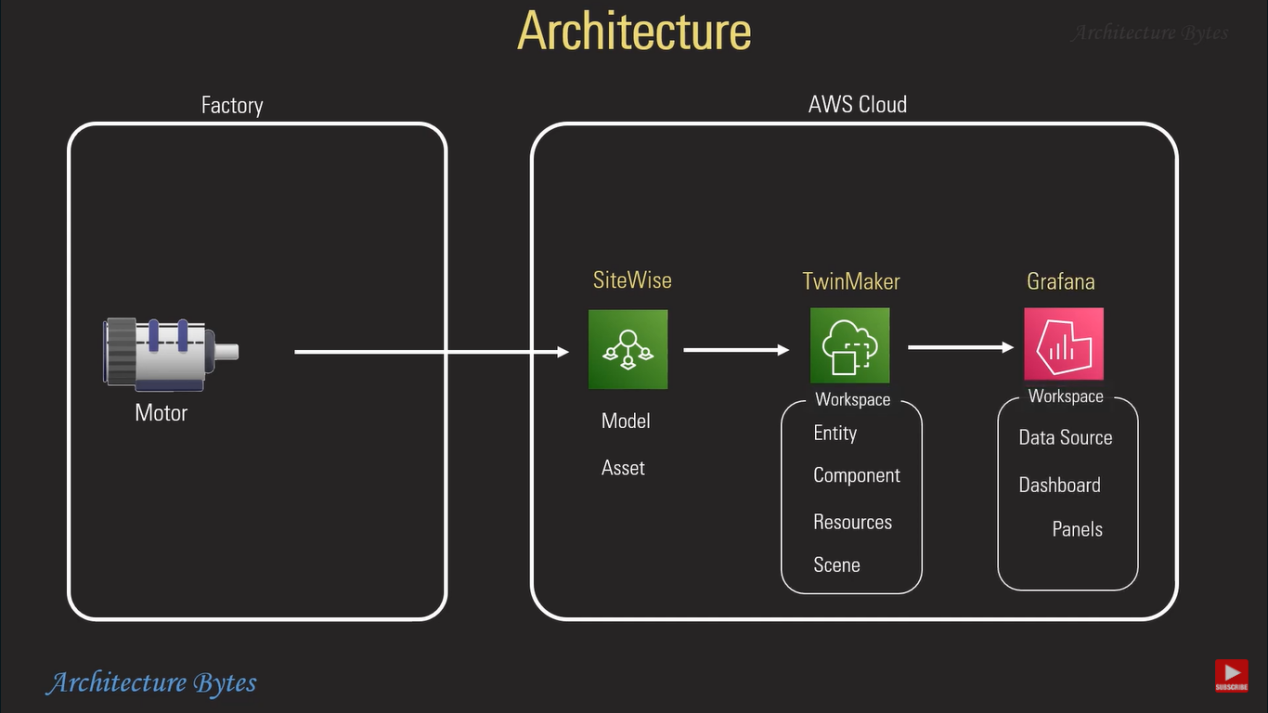
Entities: Entities represent and capture the capabilities of physical equipment, processes, and spaces. They are used to specify relationships between different components and connect them to data sources, forming the basis of the digital twin graph managed by AWS IoT TwinMaker.

Resources: These refer to existing 3D models and visual assets uploaded to the resource library. They are utilized to create 3D visualizations that accurately depict the physical environment.

Scenes: Scenes are compositions of 3D visual assets within the digital twin environment. Using the scene composer, users can add visual assets into a scene and position them to match the physical environment, enhancing the overall visualization experience.



Example usecase:



***3D Scene Studio tool for Azure Digital Twins.***

The discussion revolves around the integration of 3D visualizations with Digital Twins, particularly focusing on the new 3D Scene Studio tool for Azure Digital Twins (ADT). Here's a detailed breakdown of the main points discussed in the conversation:

**Introduction to Digital Twins and 3D Visualization:**

Digital Twins are digital representations of physical assets or systems.

3D visualization provides an intuitive way to understand Digital Twins and related data.

Riley, a PM with Azure Digital Twins, highlights the importance of visualizing complex data for better insights and decision-making.

**Use Cases for 3D Scenes in IoT and Digital Twins:**

Examples include wind farm turbine simulations for maintenance optimization, monitoring building occupancy during emergencies, and visualizing real-time data from the International Space Station (ISS).

These scenarios demonstrate the practical applications of 3D visualization in various industries.

**Challenges in Reconciling 3D Models with Digital Twins:**

Reconciling 3D models with Digital Twins can be challenging due to differences in expertise between industry experts and graphic designers.

The goal is to empower subject matter experts (SMEs) to use 3D technology without extensive graphic design knowledge.

**Introduction to the 3D Scene Studio Tool:**

The 3D Scene Studio simplifies the process of creating 3D visualizations by linking Digital Twins data with 3D models.

It supports standard 3D file formats like GLTF and GLB, making it compatible with existing CAD models.

**Demonstration of Creating a Scene for a Distribution Center:**

Riley demonstrates how to create a scene for a distribution center using the 3D Scene Studio.

The process involves linking Digital Twins data, uploading 3D models, creating elements, defining behaviors, and adding widgets for visualization and data analysis.

**Key Features of the 3D Scene Studio:**

Users can map 3D elements to specific Digital Twins entities and define behaviors based on data attributes.

The tool provides options for visualizing data through color overlays, alerts, and diagnostic widgets.

Scenes can be customized to suit different roles and use cases within an organization.

**Built on Web Technologies:**

The 3D Scene Studio is built on Babylon 5.0, enabling lightweight 3D rendering in web browsers.

This approach makes 3D visualizations more accessible to users compared to traditional desktop applications.

*Viewers are encouraged to explore the 3D Scene Studio tool for Azure Digital Twins.*

*The tool offers a user-friendly interface for creating and visualizing 3D scenes based on real-time IoT data.*

*Overall, the discussion highlights the significance of 3D visualization in enhancing the understanding and utilization of Digital Twins data across various industries, facilitated by the innovative capabilities of the 3D Scene Studio tool.*

<https://youtu.be/OzTagK628FM?si=3oqvPHaGC1SEaKWO>

<https://techcommunity.microsoft.com/t5/internet-of-things-blog/3d-scenes-studio-build-immersive-azure-digital-twins/ba-p/3403244>

Creating a 3D digital twin involves leveraging various tools, technologies, and methodologies from around the world. Below, I'll mention some of the widely used and cutting-edge approaches to create 3D digital twins:

**Computer-Aided Design (CAD) Software:**

Widely used CAD software like AutoCAD, SolidWorks, and CATIA enables engineers and designers to create detailed 3D models of physical objects.

These tools are essential for accurately modeling the geometry, dimensions, and properties of assets to be represented in the digital twin.

**3D Modeling and Visualization Software:**

Advanced 3D modeling and visualization software such as Blender, Autodesk Maya, and SketchUp offer powerful tools for creating and rendering complex 3D scenes.

These tools are invaluable for designing intricate environments, architectural structures, and visualizing assets within the digital twin.

**Game Engines:**

Game engines like Unity and Unreal Engine are widely used for creating interactive and immersive 3D environments.

These engines provide sophisticated rendering, physics simulation, and scripting capabilities, making them ideal for building dynamic and interactive digital twins.

**Simulation Software:**

Simulation software such as Simulink, COMSOL Multiphysics, and ANSYS allows engineers to simulate the behavior of physical systems and components.

Simulation tools are crucial for modeling the dynamics, interactions, and performance characteristics of assets within the digital twin.

**Internet of Things (IoT) Platforms:**

IoT platforms like Microsoft Azure Digital Twins, AWS IoT, and Siemens MindSphere enable the integration of sensor data and IoT devices with digital twins.

These platforms provide APIs, SDKs, and data management tools for connecting and monitoring real-world assets within the digital twin environment.

**Cloud-Based Services**:

Cloud-based services such as Autodesk Forge, Bentley iTwin, and Dassault Systèmes 3DEXPERIENCE platform offer scalable and collaborative environments for building and deploying digital twins.

These platforms provide cloud-based storage, computational resources, and data analytics capabilities for managing and visualizing digital twins on a global scale.

**Augmented Reality (AR) and Virtual Reality (VR) Technologies:**

AR and VR technologies are increasingly being used to enhance the visualization and interaction with digital twins.

Tools like ARKit, ARCore, and Oculus Rift enable users to experience and interact with 3D digital twins in immersive virtual environments.

**Custom Development:**

Organizations may develop custom software solutions using programming languages such as Python, JavaScript, or C++ to create specialized digital twin applications.

Custom development allows for tailored functionality and integration with existing systems and workflows.

**Data Integration and Analytics:**

Advanced data integration and analytics tools, including machine learning and artificial intelligence algorithms, play a crucial role in processing and analyzing data from digital twins.

These tools enable predictive maintenance, anomaly detection, and optimization of asset performance based on real-time data from the digital twin.

Overall, the best and latest way to create a 3D digital twin involves leveraging a combination of these tools, technologies, and methodologies to accurately model, simulate, and monitor physical assets and environments in a digital context.

Here are a few approaches you can consider for integrating 3D model displaying with D3.js:

**Three.js Integration:** Three.js is a popular JavaScript library for creating 3D graphics in the browser. You can use Three.js alongside D3.js to render 3D models and scenes. D3.js can handle the data-driven aspects of your visualization, while Three.js handles the 3D rendering. You can embed Three.js views within D3.js visualizations or vice versa, depending on your requirements.

**WebGL**: D3.js itself doesn't directly support WebGL, the technology underlying most modern web-based 3D rendering. However, you can use WebGL directly or through libraries like regl.js alongside D3.js to create custom 3D visualizations. WebGL provides low-level access to the GPU, enabling high-performance 3D rendering directly in the browser.

**Pre-rendered 3D Graphics:** If your 3D models are static or don't require real-time rendering, you can pre-render them into 2D images or SVG paths and incorporate them into your D3.js visualizations. This approach is suitable for displaying complex 3D objects as part of a larger data visualization.

**External 3D Visualization Libraries:** There are other JavaScript libraries dedicated specifically to 3D visualization, such as Babylon.js or CesiumJS. You can use these libraries alongside or instead of D3.js, depending on your project requirements. They offer advanced features for 3D rendering, interaction, and animation.

**SAP 3D Visual Enterprise Viewer:** SAP provides a 3D viewer component as part of its Visual Enterprise solutions. This component allows you to display and interact with 3D models within SAPUI5 applications. However, it may require additional licensing and setup, as it's part of SAP's enterprise offerings.

**Three.js Integration:** Three.js is a popular JavaScript library for creating 3D graphics in the browser. You can integrate Three.js into your SAPUI5 application to build custom 3D viewers. Three.js provides a wide range of features for loading, rendering, and interacting with 3D models in real-time.

**Babylon.js** Integration: Babylon.js is another powerful JavaScript library for building 3D applications on the web. Similar to Three.js, you can integrate Babylon.js into your SAPUI5 application to create immersive 3D experiences. It offers features such as loading various 3D model formats, shaders, physics simulations, and more.

**WebGL and Web Components**: You can directly leverage WebGL, the underlying technology for real-time 3D rendering in web browsers, to build custom 3D viewers. Additionally, you can encapsulate WebGL functionality into web components and integrate them into your SAPUI5 application for reusability.

**External 3D Viewer Components:** Explore third-party 3D viewer components or plugins that offer integration with SAPUI5. Some commercial or open-source solutions may provide pre-built 3D viewer components that you can easily incorporate into your SAPUI5 application.

When selecting a solution for integrating a 3D viewer into your SAPUI5 application, consider factors such as compatibility, performance, features, licensing, and support. Additionally, ensure that the chosen solution aligns with your project requirements and architectural considerations.

**Blender is a powerful open-source 3D modeling and animation software, but it is not directly usable within SAPUI5 applications. However, Blender can be used to create and export 3D models in various formats that can then be integrated into SAPUI5 applications using appropriate libraries or frameworks.**

Here's how Blender can be leveraged in conjunction with SAPUI5:

**Model Creation:** Blender can be used to create 3D models, animations, and scenes. Designers and artists can use Blender's extensive toolset to create visually appealing 3D assets for use in SAPUI5 applications.

**Export to Supported Formats:** Blender supports exporting 3D models in various formats such as OBJ, FBX, STL, Collada (DAE), and glTF. These formats are widely supported by 3D rendering libraries and frameworks used in web development.

Integration with Three.js or Babylon.js: Once you've exported your 3D models from Blender, you can integrate them into your SAPUI5 application using libraries like Three.js or Babylon.js. These libraries provide APIs for loading 3D models, applying materials and textures, and rendering them within web browsers.

Customization and Interaction: With Three.js or Babylon.js, you can customize the appearance and behavior of your 3D models within SAPUI5 applications. You can add interactivity, animations, lighting effects, and more to enhance the user experience.

Deployment: Ensure that the exported 3D models and associated assets (textures, materials, animations) are properly deployed along with your SAPUI5 application. This may involve hosting the assets on a web server or including them directly in your application's resources.

By combining Blender with 3D rendering libraries like Three.js or Babylon.js, you can create immersive 3D experiences within SAPUI5 applications. This approach allows you to leverage Blender's capabilities for 3D modeling and animation while utilizing web-based technologies for rendering and interactivity.

To render 3D models uploaded by users within a SAPUI5 application, you can use libraries specifically designed for 3D model visualization. While SAPUI5 itself doesn't provide native support for 3D model rendering, you can integrate third-party libraries that offer this functionality. Here are some options:

Babylon.js Viewer: Babylon.js offers a viewer component specifically designed for rendering 3D models in web applications. The Babylon.js viewer supports various 3D model formats such as glTF, OBJ, and STL. You can integrate the Babylon.js viewer into your SAPUI5 application to display user-uploaded 3D models.

Three.js: Three.js is another popular JavaScript library for 3D rendering, which supports loading and displaying 3D models in web applications. While Three.js doesn't provide a pre-built viewer component like Babylon.js, you can create a custom viewer using Three.js to render user-uploaded 3D models within your SAPUI5 application.

Sketchfab Viewer API: Sketchfab offers a viewer API that allows you to embed Sketchfab's 3D model viewer into web applications. Users can upload 3D models to Sketchfab, and you can use the Sketchfab Viewer API to embed the uploaded models into your SAPUI5 application. This approach leverages Sketchfab's viewer capabilities without the need to handle 3D model rendering directly.

Custom WebGL Integration: If you prefer more control over the rendering process, you can use WebGL directly or through a lower-level library like regl.js to render 3D models in your SAPUI5 application. This approach requires more effort but provides flexibility for customizing the rendering pipeline to suit your specific requirements.

Before choosing a library or approach, consider factors such as compatibility with SAPUI5, ease of integration, performance, and features required for your application. Once you've selected a library, you can follow its documentation and examples to integrate 3D model rendering into your SAPUI5 application.

**Working code for 3d model:**

**Step 1:Controller code**

sap.ui.define([

    "sap/ui/core/mvc/Controller",

    "sap/ui/model/json/JSONModel",

    "sap/ui/vk/ContentResource",

    "sap/m/MessageToast"

], function(Controller, JSONModel, ContentResource, MessageToast) {

    "use strict";

    var handleEmptyUrl = function(view) {

        var oBundle = view.getModel("i18n").getResourceBundle();

        var msg = oBundle.getText("missingUrl");

        MessageToast.show(msg);

    };

    var loadModelIntoViewer = function(viewer, remoteUrl, sourceType, localFile) {

        // what is currently loaded in the view is destroyed

        viewer.destroyContentResources();

        var source = remoteUrl || localFile;

        if (source) {

            // content of viewer is replaced with new data

            var contentResource = new ContentResource({

                source: source,

                sourceType: sourceType,

                sourceId: "abc"

            });

            // content: chosen path. content added to the view

            viewer.addContentResource(contentResource);

        }

    };

    return Controller.extend("igitaltwin.controller.View1", {

        onInit: function() {

            var sourceData = {

                localFile: undefined,

                remoteUrl: undefined

            };

            var model = new JSONModel();

            model.setData(sourceData);

            this.getView().setModel(model, "source");

        },

        onPressLoadRemoteModel: function(event) {

            var view = this.getView();

            var sourceData = view.getModel("source").oData;

            var viewer = view.byId("viewer");

            if (sourceData.remoteUrl) {

                loadModelIntoViewer(viewer, sourceData.remoteUrl, "vds4");

            } else {

                handleEmptyUrl(view);

            }

        },

        onPressLoadRemoteImage: function(event) {

            var view = this.getView();

            var sourceData = view.getModel("source").oData;

            var viewer = view.byId("viewer");

            if (sourceData.remoteUrl) {

                loadModelIntoViewer(viewer, sourceData.remoteUrl, "jpg");

            } else {

                handleEmptyUrl(view);

            }

        },

        onPressLoadSampleFile: function(event) {

            var view = this.getView();

            var viewer = view.byId("viewer");

            loadModelIntoViewer(viewer, "model/cooper.vds", "vds4");

        },

        onChangeFileUploader: function(event) {

            var view = this.getView();

            var viewer = view.byId("viewer");

            var localFile = event.getParameter("files")[0];

            // if user selects a local file

            if (localFile) {

                var fileName = localFile.name;

                var index = fileName.lastIndexOf(".");

                if (index >= 0 && index < fileName.length - 1) {

                    var sourceType = fileName.substr(index + 1).toLowerCase();

                    if (sourceType === "vds") {

                        // Source type 'vds' is using old DVL rendering engine which is now deprecated.

                        // Use 'vds4' for new Threejs based rendering engine

                        sourceType = "vds4"

                    }

                    loadModelIntoViewer(viewer, null, sourceType, localFile);

                }

            }

        }

    });

});

**View code:**

<!-- <mvc:View controllerName="igitaltwin.controller.View1"

xmlns:mvc="sap.ui.core.mvc" displayBlock="true"

xmlns="sap.m">

<Page id="page" title="{i18n>title}">

    <content />

</Page>

</mvc:View> -->

<mvc:View controllerName="igitaltwin.controller.View1"

xmlns="sap.m"

xmlns:mvc="sap.ui.core.mvc"

xmlns:vk="sap.ui.vk"

xmlns:l="sap.ui.layout"

xmlns:f="sap.ui.layout.form"

xmlns:u="sap.ui.unified"

displayBlock="true">

<Shell>

    <App id="singleFile">

        <Page title="{i18n>pageTitle}">

            <VBox height="100%">

                <items>

                    <HBox>

                        <layoutData>

                            <FlexItemData growFactor="1" />

                        </layoutData>

                        <items>

                            <vk:Viewer id="viewer" toolbarTitle="{i18n>viewerToolbarTitle}">

                                <vk:layoutData>

                                    <FlexItemData growFactor="1" />

                                </vk:layoutData>

                            </vk:Viewer>

                        </items>

                    </HBox>

                    <f:Form editable="true">

                        <f:layout>

                            <f:GridLayout/>

                        </f:layout>

                        <f:formContainers>

                            <f:FormContainer>

                                <f:formElements>

                                    <f:FormElement>

                                        <f:fields>

                                            <Input value="{source>/remoteUrl}" valueLiveUpdate="true" placeholder="{i18n>formRemoteURL}">

                                                <layoutData>

                                                    <f:GridElementData hCells="auto"/>

                                                </layoutData>

                                            </Input>

                                            <Button text="{i18n>buttonLoadModel}" press="onPressLoadRemoteModel">

                                                <layoutData>

                                                    <f:GridElementData hCells="2"/>

                                                </layoutData>

                                            </Button>

                                            <Button text="{i18n>buttonLoadImage}" press="onPressLoadRemoteImage">

                                                <layoutData>

                                                    <f:GridElementData hCells="3"/>

                                                </layoutData>

                                            </Button>

                                        </f:fields>

                                    </f:FormElement>

                                    <f:FormElement>

                                        <f:fields>

                                            <u:FileUploader fileType="vds,png,jpg,jpeg,gif" placeholder="{i18n>formLocalFileName}" width="100%" change="onChangeFileUploader">

                                            </u:FileUploader>

                                            <Button text="{i18n>buttonLoadSample}" press="onPressLoadSampleFile">

                                                <layoutData>

                                                    <f:GridElementData hCells="3"/>

                                                </layoutData>

                                            </Button>

                                        </f:fields>

                                    </f:FormElement>

                                </f:formElements>

                            </f:FormContainer>

                        </f:formContainers>

                    </f:Form>

                </items>

            </VBox>

        </Page>

    </App>

</Shell>

</mvc:View>

**I18 code:**

# App Descriptor

appTitle=App title

appDescription=This is a description coming from the i18n as specified in manifest.json

# Viewer Descriptor

viewerToolbarTitle=Single file upload

# Page Descriptor

pageTitle=Single File

# Form Descriptor

formRemoteURL=Remote Model URL

buttonLoadModel=Load

buttonLoadImage=Load Image

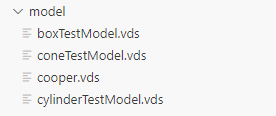
buttonLoadSample=Load Sample File

formLocalFileName=Local File Name

# Message Toast

missingUrl=Please specify a URL

**Model should be having files of 3d models/user can upload :**



Follow the similar thing for to make it work from demo kit

# Step 1: 3D Viewer With Single File Loading

In this step, you will be creating a 3D Viewer application that allows a user to load a single 2D image or 3D model that is stored locally or remotely.

**App.view.xml**

The App.view.xml file specifies how the page in the application will be laid out. We have one form container containing two form elements (formElement). The first formElement element contains the fields for loading 2D or 3D resources located remotely. We have specifed one input text field, and two buttons: one button for loading 2D images, and the other button for loading 3D models.

In the second formElement, we are using the FileUploader control to generate an input text field and a button to load 2D or 3D resources stored locally. We have specified the following file types as valid file types for loading using this formElement.

The labels for each of the fields are specified in the text attributes.

**App.controller.js**

This file contains the logic for loading files into the Viewer application.

The handleEmptyUrl function specifies what should occur if a user clicks on any of the buttons for loading, without having specified a URL to a resource first.

The loadModelIntoViewer function specifies how the resource will be loaded into the Viewer application for view

The following event functions specify how the form elements should behave during certain events. The following list outlines what each of the functions do:

onInit - declares an empty structure when the controller is initialized. The empty structure is set as the model for the URLs.

onPressLoadRemoteModel - the logic for the button that loads 3D resources stored remotely.

onPressLoadRemoteImage - the logic for the button that loads 2D resources stored remotely.

onChangeFileUploader - the logic for the fields that load 2D or 3D resources stored locally.

# Step 2: 3D Viewer With Multiple File Loading

In this step, you will be creating a Viewer application that allows a user to load multiple 3D resources stored locally.

Since we now have a different layout for the Viewer application compared to the sample Viewer application in [Step 1: 3D Viewer With Single File Loading](https://help.sap.com/docs/UI_ADD-ON_FOR_SAP_NETWEAVER_20/b4b7cba328bc480d9b373c7da9335537/0e219128fe564855b91ae50d7a29424b.html?version=2.07" \o "In this step, you will be creating a 3D Viewer application that allows a user to load a single 2D image or 3D model that is stored locally or remotely.), we will need to change the logic for the application to accommodate for multiple file loading.

Add the checkIfAllInputsEmpty function to check whether the user has entered text into any of the input fields in the application. The checkIfAllInputsEmpty function returns the value true if the user hasn't entered any input at all, and the existing handleEmptyUrl function is called to display a message on the screen.

Update the onInit function so that we are specifying an empty data structure with three properties (url1, url2, and url3).

Replace the loadModelIntoViewer and onPressLoadRemoteModel functions with the following functions:

* loadModelsIntoViewer - loads the models into Viewer
* onPressLoadRemoteModels - handles the click event on the **Load** button

Remove the following functions:

* onPressLoadRemoteImage (since we are only loading 3D resources)
* onChangeFileUploader (since we are not using the FileUploader control for this application)

# Step 3: 3D Viewer Using the Viewport Control

In this step, you will be creating a 3D Viewer application using the sap.ui.vk.Viewport control.

In previous steps, we utilized the sap.ui.vk.Viewer composite control to create a Viewer application capable of loading a 2D or 3D resource. Now, we will create a Viewer application with a pre-loaded resource without using the composite sap.ui.vk.Viewer control. Instead, we will use the following controls and library in sap.ui.vk, which are what you'll need at a minimum to display a 3D model in your application.

* Viewport control
* ContentResource control
* GraphicsCore library

We will build on this sample application in later steps of the 3D Viewer tutorial by introducing the other non-composite sap.ui.vk controls to create more complex Viewer applications.

The logic in this App.controller.js file can be summarized as follows:

1. Create a Content Resource that stores a pre-specified model
2. Initiate a scene in our application in the following order:
   1. Create a Graphics Core instance
   2. Create a Viewport that is bound to the Graphics Core instance
   3. Load the Content Resource to the Graphics Core for rendering on the Viewport

 sap.ui.define([

    "sap/ui/core/mvc/Controller",

    "sap/ui/vk/ContentResource",

    "sap/ui/vk/ContentConnector",

     "sap/ui/vk/dvl/ViewStateManager"

], function(Controller, ContentResource, ContentConnector, ViewStateManager) {

    "use strict";

    return Controller.extend("standaloneViewport.controller.App", {

      onInit: function() {

            var view = this.getView();

            var oViewport = view.byId("viewport");

          // Constructor for a new content resource. procides an object that owns content resouces, tracks changes, loads and destroys

            // content built from the content resource.

            var contentResource = new ContentResource({

                // specifying the resource to load

                source: "models/boxTestModel.vds",

                sourceType: "vds",

                sourceId: "abc123"

            });

            // Constructor for a new content connector

            var contentConnector = new ContentConnector("abcd");

      // Manages the visibility and the selection states of nodes in the scene.

            var viewStateManager = new ViewStateManager("vsmA", {

                contentConnector: contentConnector

            });

    // set content connector and viewStateManager for viewport

            oViewport.setContentConnector(contentConnector);

            oViewport.setViewStateManager(viewStateManager);

      view.addDependent(contentConnector).addDependent(viewStateManager);

            // Add resource to load to content connector

            contentConnector.addContentResource(contentResource);

        }

    });

});

xml

<mvc:View controllerName="standaloneViewport.controller.App"

xmlns="sap.m"

xmlns:mvc="sap.ui.core.mvc"

xmlns:vk="sap.ui.vk"

displayBlock="true">

<Shell>

    <App id="standaloneViewport">

        <Page title="{i18n>pageTitle}">

            <vk:Viewport id="viewport"/>

        </Page>

    </App>

</Shell>

</mvc:View>

The onInit function in this code snippet is a part of an SAPUI5 controller, which is typically executed when the corresponding view is initialized. Let's break down the code within the onInit function:

var view = this.getView();: This line retrieves the view instance associated with the current controller. The this keyword refers to the controller instance, and getView() fetches the view instance.

var oViewport = view.byId("viewport");: This line retrieves a reference to a UI element with the ID "viewport" from the view. It's assumed that there's an element in the view with the ID "viewport", which is typically a UI element used for displaying graphical content.

var contentResource = new ContentResource({ ... });: Here, a new instance of ContentResource is created. This ContentResource is responsible for managing a resource that contains content to be loaded and displayed in the viewport. It's configured with the following properties:

source: Specifies the path to the resource to be loaded. In this case, it's "models/boxTestModel.vds".

sourceType: Specifies the type of resource being loaded. In this case, it's "vds".

sourceId: Specifies an ID for the resource. Here, it's "abc123".

var contentConnector = new ContentConnector("abcd");: This line creates a new instance of ContentConnector, which is used for managing the loading and unloading of content resources. It's given an ID of "abcd".

var viewStateManager = new ViewStateManager("vsmA", { ... });: This line creates a new instance of ViewStateManager. The ViewStateManager is responsible for managing the visibility and selection states of nodes in the scene. It's configured with a content connector (contentConnector) to handle changes in content.

oViewport.setContentConnector(contentConnector);: This line sets the ContentConnector instance (contentConnector) for the viewport. This indicates that the viewport should use this ContentConnector for managing content.

oViewport.setViewStateManager(viewStateManager);: This line sets the ViewStateManager instance (viewStateManager) for the viewport. This indicates that the viewport should use this ViewStateManager for managing the visibility and selection states of nodes.

view.addDependent(contentConnector).addDependent(viewStateManager);: This line adds the contentConnector and viewStateManager as dependents to the view. This ensures proper lifecycle management of these objects with respect to the view.

contentConnector.addContentResource(contentResource);: This line adds the contentResource to the ContentConnector. This means that the ContentConnector will handle the loading and management of this resource.

Overall, this code sets up the necessary components for loading and managing content in a viewport, including the content resource, content connector, and view state manager.

**reference:**

**Sample: 3D Viewer - Step 1 - 3D Viewer With Single File Loading**

https://sapui5.hana.ondemand.com/#/entity/sap.ui.vk.tutorial.VIT/sample/sap.ui.vk.tutorial.VIT.01/code

**Sample: 3D Viewer - Step 2 - 3D Viewer With Multiple File Loading**

https://sapui5.hana.ondemand.com/sdk/#/entity/sap.ui.vk.tutorial.VIT/sample/sap.ui.vk.tutorial.VIT.02/code

Complete explaination of codes:

https://help.sap.com/docs/UI\_ADD-ON\_FOR\_SAP\_NETWEAVER\_20/b4b7cba328bc480d9b373c7da9335537/80b0b57480e54a50a08daf468453aa33.html?version=2.07