**📚 Resources**

👉 Check the [full list of data types](https://github.com/buildingSMART/IDS/blob/development/Documentation%2FImplementersDocumentation%2FDataTypes.md) supported by BuildingSmart.

👉 You can fin more IDS files to work with in the official [BuildingSmart repo](https://github.com/buildingSMART/IDS/tree/development/Documentation%2FImplementersDocumentation%2FTestCases%2Fproperty).

👉 Learn how to make the most of the Properties Manager with this [tutorial](https://docs.thatopen.com/Tutorials/Components/Core/IfcPropertiesManager).

**🗝 Key Concepts**

**🔗 IDS File Integration for BIM Models**

Integrating IDS (Information Delivery Specification) files into BIM models enables seamless property addition. This ensures that stakeholder requirements are incorporated directly into the model elements for enhanced compliance.

**🖋 Dynamic Property Management**

Creating and assigning properties dynamically based on IDS specifications enhances model flexibility. Users can adapt their BIM data to reflect evolving project requirements efficiently.

**🔄 Interactive Selection and Restriction**

Implementing interactive selection and restriction mechanisms ensures that only elements matching specific IDS criteria can be selected, reducing errors and maintaining data integrity.

**🎨 Visual Feedback Through Highlighting**

Highlighting elements that meet IDS criteria provides immediate visual feedback, streamlining workflows and improving user understanding of property assignments within the BIM model.

**🛠 Custom Dialogs for User Input**

Using custom dialogs to confirm and adjust property assignments allows users to refine values before adding them to elements, enhancing accuracy and control.

**🧩 Component-Based Model Updates**

The modular approach to updating model data ensures consistency across elements while leveraging relationships like "IsDefinedBy" to align with the IFC schema.

**🌐 Extending BIM Capabilities with Highlighters**

Highlighter tools enhance BIM applications by enabling visual emphasis on selectable elements and managing restrictions, offering a richer user interaction layer.

**🤓 Exercise Development**

The following is the step-by-step guide on how to accomplish the exercise. Use this as complementary information from the video. 📽

**🏗️ Scaffold the Project**

Start by initializing the BIM App

npm create bim-app@latest

Select Vanilla and install dependencies

npm i

Create the files with the following structure

src/

|-- bim-components/

| |-- AppManager/

| |-- IdsParameters/

| | |-- src/

| | | |-- user-interface.ts

| | | |-- confirm.ts

| | | |-- index.ts

| | |-- index.ts

| |-- index.ts

**🎨 User Interface**

import \* as BUI from "@thatopen/ui";

import \* as OBC from "@thatopen/components";

import \* as OBF from "@thatopen/components-front";

import \* as FRAGS from "@thatopen/fragments";

// The goal with this UI is to have a toolbar section with a dropdown list

// with the available properties and a button to update the selected element.

// The user should select a property from the dropdown, then an element from

// the allowed elements and then click on update

// to review the data to add to the element.

// These imports will be explained in the next sections

import { IdsParameters, Property } from "..";

import { confirmPropsModal } from "./confirm";

export function IdsUI(components: OBC.Components) {

const parameters = components.get(IdsParameters);

// The highlighter allows us to get the selected elements.

const highlighter = components.get(OBF.Highlighter);

// A dropdown to hold the available properties

const propsDrop = document.createElement("bim-dropdown");

propsDrop.label = "Property Name";

propsDrop.addEventListener("change", () => {

BUI.ContextMenu.removeMenus();

// This method will be explained in the component section.

parameters.restrictSelection(propsDrop.value[0]);

});

// When the IDS file is loaded, we will iterate the data to add the options

// To the dropdown

if (parameters.idsData) {

for (const property of parameters.idsData) {

const option = document.createElement("bim-option");

option.label = property.name;

option.value = property;

propsDrop.append(option);

}

}

// When we want to confirm the addition of the new property

// We will show a dialog with the data, more on that in the confirm section below.

const onUpdateClick = (

components: OBC.Components,

property: Property,

selection: FRAGS.FragmentIdMap,

) => {

// We have to make sure there is a property and elements selected.

if (Object.keys(selection).length === 0 || !property) return;

const [confirm] = confirmPropsModal({

components,

property,

selection,

onSubmit: () => confirm.close(),

});

confirm.showModal();

};

return BUI.html`

<bim-toolbar-section label="IDS Selection" icon="catppuccin:properties">

${propsDrop}

<bim-button label="Update" @click=${() => onUpdateClick(components, propsDrop.value[0], highlighter.selection.select)}></bim-button>

</bim-toolbar-section>`;

}

**☑️ Confirm UI**

import \* as BUI from "@thatopen/ui";

import \* as OBC from "@thatopen/components";

import \* as FRAGS from "@thatopen/fragments";

import { IdsParameters, Property } from "..";

// Interface for the stateful component

interface AssignPropsModalState {

components: OBC.Components;

property: Property;

selection: FRAGS.FragmentIdMap;

onSubmit: () => void;

}

const template: BUI.StatefullComponent<AssignPropsModalState> = (state) => {

const { components, property, selection, onSubmit } = state;

const parameters = components.get(IdsParameters);

// Give the dialog a unique id.

const panelSectionID = `form-${BUI.Manager.newRandomId()}`;

// This block will populate the form so when the user

// sees the dialog, will know the details of the property to be added.

const pset = document.createElement("bim-text-input");

pset.label = "Property Set";

pset.placeholder = property.pSet;

const name = document.createElement("bim-text-input");

name.label = "Property Name";

name.placeholder = property.name;

const type = document.createElement("bim-text-input");

type.label = "Data Type";

type.placeholder = property.type;

const value = document.createElement("bim-text-input");

value.label = "Value";

value.placeholder = property.value as string;

// The add function will get the respective dialog

const onAdd = () => {

const panelSection = document.getElementById(

panelSectionID,

) as BUI.PanelSection;

if (!panelSection) return;

// If a value was given in the form, the data

// from the property will be changed.

// Else it will not be modified and the displayed data

// will remain

if (pset.value) property.pSet = pset.value;

if (name.value) property.name = name.value;

if (type.value) property.type = type.value;

if (value.value) property.value = value.value;

parameters.updateModel(property, selection);

onSubmit();

};

return BUI.html`

<dialog>

<bim-panel>

<bim-panel-section id=${panelSectionID} label="Confirm New Property" fixed>

${pset}

${name}

${type}

${value}

<bim-button label="Add" @click=${onAdd}></bim-button>

</bim-panel-section>

</bim-panel>

</dialog>

`;

};

export const confirmPropsModal = (state: AssignPropsModalState) => {

const component = BUI.Component.create<

HTMLDialogElement,

AssignPropsModalState

>(template, state);

document.body.append(component[0]);

return component;

};

**✍️ Main Component Logic**

import \* as OBC from "@thatopen/components";

import \* as FRAGS from "@thatopen/fragments";

import \* as WEBIFC from "web-ifc";

import \* as OBF from "@thatopen/components-front";

import \* as THREE from "three";

// This is the structure we want to have for each property.

export interface Property {

specId: string; // Needed in case multiple specifications are loaded

pSet: string; // Property set name

name: string; // Property name

type: string; // Data type of the property

value: string | boolean | number; // Initially the threshold value from the IDS

}

export class IdsParameters extends OBC.Component {

enabled = false;

static readonly uuid = "fb6821d5-a9e2-4dd0-a46c-dcea964b875e";

// This will hold the properties to be displayed in the UI

idsData: Property[] = [];

highlighter: OBF.Highlighter;

constructor(components: OBC.Components) {

super(components);

components.add(IdsParameters.uuid, this);

// We use the highlighter to highlight restricted elements

// with the yellow color

this.highlighter = this.components.get(OBF.Highlighter);

this.highlighter.add("selectable", new THREE.Color("rgb(248, 236, 60)"));

}

// The overall purpose of the load method is to take an IDS file, traverse its content

// and parse the information into the interface given above to store the data

// into the idsData, then the array will be read in the UI.

async load() {

// Load the ids and read the data from the file. Each read adds the

// data into the component incrementally.

const ids = this.components.get(OBC.IDSSpecifications);

const file = await fetch('your\_ids\_file')

const fileData = await file.text()

const specs = ids.load(fileData);

// For simplicity, we will take the first specification's requirements.

const requirements = Array.from(specs[0].requirements);

const specId = specs[0].identifier;

for (const requirement of requirements) {

// There are many facet, as we are working with properties

// We need to confirm the requirement is actually a property

if (requirement.facetType !== "Property") continue;

// The particular structure of the property is {type, parameter}

const \_requirement = requirement as OBC.IDSProperty;

if (

!(

// As seen in the interface, we must confirm the types

// The dataType does not change so basically just confirm its existence

typeof \_requirement.propertySet.parameter === "string" &&

typeof \_requirement.baseName.parameter === "string" &&

\_requirement.dataType &&

\_requirement.value?.type === "simple"

)

)

continue;

const pSet = \_requirement.propertySet.parameter;

const name = \_requirement.baseName.parameter;

const type = \_requirement.dataType;

const value = \_requirement.value?.parameter;

this.idsData.push({ specId, pSet, name, type, value });

}

}

// This method will highlight the elements to which the property can be applied.

// Also restricts the selection of elements to those that were highlighted while the

// property is selected in the dropdown.

async restrictSelection(property: Property) {

// When a property is deselected, this would remove the restriction

// clear any selections made and clear the highlighted elements

delete this.highlighter.selectable.select;

this.highlighter.clear("selectable");

this.highlighter.clear("select");

// Also, there is no need to execute the rest of the code when a property

// is deselected.

if (!property) return;

// The ids will contain the loaded specifications

const ids = this.components.get(OBC.IDSSpecifications);

// The fragment manager allows us to traverse the model

const fragsManager = this.components.get(OBC.FragmentsManager);

// For each model, check each id and data specification

for (const [\_, model] of fragsManager.groups.entries()) {

for (const [id, data] of ids.list) {

if (id !== property.specId) continue; // Here the ids should match.

// Applicability contains the entities to which the specification can be

// applied to.

for (const entity of data.applicability) {

// This will return the expressIds related to the entities.

const entities = await entity.getEntities(model, {});

// To then get the fragment map

const fragsMap = model.getFragmentMap(entities);

// With that fragment map, we can then restrict the selection

// The key of the object must match the standard highlighter name

// which is different from the custom one we created.

this.highlighter.selectable = { select: fragsMap };

// And then highlight with our custom highlighter.

await this.highlighter.highlightByID("selectable", fragsMap, false);

}

}

}

}

// As the name suggest, after confirming the addition, this method will

// be executed to modify the model and add the new property.

// There are three cases:

// 1. The property exists and so the property set does too. Nothing needs to be done

// 2. The property doesn't exists but the PSet does. Create and add the property.

// 3. Nothing exists. Create both the PSet and the property and add them.

async updateModel(property: Property, selection: FRAGS.FragmentIdMap) {

// Just another check that elements and a property were selected.

if (!(property || selection)) return;

// The prop manager allows us to add the properties and property sets.

// The indexer to find relations between elements of the IFC schema.

// Fragments, again, to traverse the model.

const propsManager = this.components.get(OBC.IfcPropertiesManager);

const indexer = this.components.get(OBC.IfcRelationsIndexer);

const fragments = this.components.get(OBC.FragmentsManager);

const modelIdMap = fragments.getModelIdMap(selection);

// For each model and its associated express ids of the selection...

for (const [modelID, expressIDs] of Object.entries(modelIdMap)) {

const model = fragments.groups.get(modelID);

if (!model) continue;

// Go through each expressId...

for (const expressID of expressIDs) {

// It must have a expressId related that "defines it"...

const definitions = indexer.getEntityRelations(

model,

expressID,

"IsDefinedBy",

);

// Placeholder for when the PSet exists

let pset: Record<string, any> | null = null;

// For each related expressId

for (const defID of definitions) {

const defAttrs = await model.getProperties(defID);

if (!defAttrs) continue;

// Check whether the PSet exists or not.

if (defAttrs.Name?.name === property.pSet) {

// Extract the properties to check if the one the be added exists.

const properties = defAttrs.HasProperties;

if (!properties) continue;

for (const prop of properties) {

const propValue = await model.getProperties(prop.value);

if (!propValue) continue;

// This check case number 1, so nothing is done.

if (propValue.NominalValue?.name === property.name.toUpperCase())

return;

}

// If the loop finishes, that means the property was not found.

// But the PSet exists, case number 2.

pset = defAttrs;

}

}

// If pset was not updated, the PSet was not found, case number 3.

if (!pset) {

// Get model history.

const { handle: ownerHistoryHandle } =

await propsManager.getOwnerHistory(model);

// Create a new PSet with WEBIFC.

// THe IFC4 matches the version of my IFC file, you should check yours.

pset = new WEBIFC.IFC4.IfcPropertySet(

new WEBIFC.IFC4.IfcGloballyUniqueId(OBC.UUID.create()),

ownerHistoryHandle,

new WEBIFC.IFC4.IfcLabel(property.pSet),

null,

[],

);

// The PSet has been created but it must be added to the model.

await propsManager.setData(model, pset);

}

// In the end, create the new property with the value from the selected one.

// This block can raise an error, ignore it with the next comment.

// @ts-ignore

const newProp = await propsManager.newSingleProperty(

model,

property.type,

property.name,

property.value,

);

// Add it once more to the model.

await propsManager.setData(model, newProp);

// And push it to the PSet, either created or existent.

pset.HasProperties.push(new WEBIFC.Handle(newProp.expressID));

// And since IFC works with relations, assign the relationship between

// the expressIds.

indexer.addEntitiesRelation(

model,

expressID,

{

type: WEBIFC.IFCRELDEFINESBYPROPERTIES,

inv: "IsDefinedBy",

},

pset.expressID,

);

}

}

}

}

export \* from "./src";

Remember to export the component:

// In src/bim-components/index.ts

export \* from "./IdsParameters";

// In src/bim-components/src/index.ts

export \* from "./user-interface.ts";

Import everything in the main.ts:

/\*import help from "./components/Panels/Help";

import camera from "./components/Toolbars/Sections/Camera";

import measurement from "./components/Toolbars/Sections/Measurement";

import selection from "./components/Toolbars/Sections/Selection";\*/

import { AppManager, IdsParameters, IdsUI } from "./bim-components";

// Load the component

/\*const projectInformationPanel = projectInformation(components);

const elementDataPanel = elementData(components);\*/

const parameters = components.get(IdsParameters);

await parameters.load()

// And add the UI

/\*const toolbar = BUI.Component.create(() => {

return BUI.html`

<bim-tabs floating style="justify-self: center; border-radius: 0.5rem;">

<bim-tab label="Import">

<bim-toolbar>

${load(components)}

</bim-toolbar>

</bim-tab>

<bim-tab label="Selection">

<bim-toolbar>

${camera(world)}

${selection(components, world)}

</bim-toolbar>

</bim-tab>

<bim-tab label="Measurement">

<bim-toolbar>

${measurement(world, components)}

</bim-toolbar>

</bim-tab>\*/

<bim-tab label="Properties">

<bim-toolbar>

${IdsUI(components)}

</bim-toolbar>

</bim-tab>

</bim-tabs>

`;

});

**⚔ Quest**

The quest is your chance to grow. It will let you increase your level to earn different badges and get access to benefits that only people with certain levels can achieve, like getting Bounties 💰 (cash rewards) for contributing to That Open Company's open-source libraries, or special gifts. To complete this lesson’s quest, do the following: 👇

Allow the user to select the IDS files to load and properly display all the available properties in the dropdown.

**🦶 Development Steps**

1. Remove the call to the load method in the main file.
2. Add an argument called fileData of type string in the load method in the component.
3. Remove the fetch and the file.text() from the load method.
4. In the user-interface.ts file create an input element from the document.
5. Define an arrow function called askForFile that receives a string argument called extension.
6. return a Promise that works with a File or null types.
7. The callback function takes the resolve argument.
8. Inside the brackets set the input attributes like this: Type to file, accept to the given extension, multiple to false.
9. Set the onchange event to an arrow function that takes no arguments.
10. Inside the brackets get the file list from input.
11. In case there is no list or no file was selected, resolve to null and return.
12. If a file was selected, get the first element from the array and resolve to that element.
13. Outside the onchange event, force the click on the input.
14. Inside the IdsUI create an async arrow function called loadIds. No arguments needed.
15. Call the askForFile function with “.ids” parameter.
16. Return if nothing was returned.
17. Get the text from the returned data and call the load method from parameters.
18. Move the parameters.idsData check to be inside of the loadIds function.
19. Remove the for loop but not the logic inside of it.
20. Do a slice to get the last element of the idsData and assign it to a const called property of type Property.