**📚 Resources**

👉 Check the official [IDS technical documentation](https://github.com/buildingSMART/IDS/tree/development/Documentation)

**🗝 Key Concepts**

**⚙️ Automated File Generation**

The generation of Revit-compatible files automates the integration of specific data from IDS specifications, significantly reducing manual configuration and potential errors.

**↔️ Data Type Conversion**

Converting IFC data types to Revit-compatible types ensures seamless transfer of property information during IFC exports, bridging the gap between openBIM standards and proprietary software.

**🆔 IDS-Driven Parametrization**

Leveraging IDS files to define shared parameters and user property sets enforces data consistency and adherence to project requirements within Revit models.

**🧩 Custom Property Configuration**

Configuring custom property sets for IFC export from Revit allows for the inclusion of specific project data, ensuring that all relevant information is captured in IFC files.

**🚦 Conditional Data Processing**

Employing conditional logic filters data based on specific criteria, such as property types and applicability, ensuring that only relevant information is processed and included in the generated files.

**🗂 Mapping Schemas for Interoperability**

Creating mapping tables between IFC and Revit elements and property types enables the accurate translation of BIM data, facilitating seamless integration and data exchange between different software platforms.

**🔄 Revit Integration Workflow**

Importing shared parameters, assigning them to project elements, and configuring IFC export settings are essential steps to ensure the data defined in IDS files is correctly integrated into Revit models and exported to IFC files.

**🤓 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**

This implementation is a NodeJS script. Just create a new folder and add three files to it:

* index.js
* mappings.js
* package.json
* The IDS file of your choice or the one provided here in the guide down in Get The Code section.

Make sure your package.json file contains the following:

{

"type": "module",

"dependencies": {

"@thatopen/components": "2.4.4"

}

}

Then, just execute npm install in the terminal of your project.

**✍ Creating the Implementation**

The index.js file will include all the logic need to create the Revit configuration files. Inside the JavaScript file, write the following:

import \* as fs from "fs"; // Import the file system module

import \* as OBC from "@thatopen/components"; // Import the That Open Company components package

import { ifcRevitTypeMapping, ifcClassMapping, ifcRevitExportMapping } from "./mappings.js"; // Import mapping objects

const components = new OBC.Components(); // Create a new instance of the Components class

const ids = components.get(OBC.IDSSpecifications); // Get an instance of the IDSSpecifications component

const convertIdsParameters = () => {

// Initialize arrays to store shared parameter and property set information. These will hold the data extracted from the IDS file and transformed into Revit-compatible formats.

const parameters = [];

const sets = [];

// Iterate through all specifications in the IDS component. Each specification represents a set of requirements defined in the IDS file.

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

// Iterate through each requirement to extract property information.

for (const req of spec.requirements) {

// Skip requirements that are not properties. We are only concerned with requirements that define properties.

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

// Extract data type and base name from the requirement.

const { dataType, baseName } = req;

// Map the IFC data type to a Revit data type.

const revitType = ifcRevitTypeMapping[dataType];

// Skip if the base name type is not simple, or there is no Revit type mapping. A simple base name allows direct extraction of the property name, and a Revit type mapping is necessary for compatibility.

if (baseName.type !== "simple" || !revitType) continue;

// Extract the property name.

const { parameter: propName } = baseName;

// Skip if a property with the same name already exists. This prevents duplicate properties in the shared parameters file.

const existingProp = parameters.find(({ name }) => name === propName);

if (existingProp) continue;

// Create a shared parameter object. This object represents a parameter definition for the Revit shared parameters file.

const sharedParameter = {

param: "PARAM",

guid: crypto.randomUUID(),

name: propName,

dataType: revitType,

dataCategory: "",

group: 1,

visible: 1,

description: "",

userModifiable: 1,

hideWhenNoValue: 0,

}

parameters.push(sharedParameter); // Add the shared parameter to the list.

}

// Initialize an array to store the IFC classes for each specification.

const elements = [];

// Iterate through the applicability section to determine which IFC classes the properties apply to.

for (const app of spec.applicability) {

// Skip applicability entries that are not entities. We are only interested in entities, since those are the IFC classes.

if (app.facetType !== "Entity") continue;

// Extract the name (entity data) from the applicability entry.

const { name: entityData } = app;

// Skip if the entity name type is not simple. A simple entity name allows direct extraction of the IFC class name.

if (entityData.type !== "simple") continue;

// Map the IFC class name to a Revit class name.

const entity = ifcClassMapping[entityData.parameter];

// Skip if there is no mapping. If we don't know how to map it to a revit class, we simply skip it.

if (!entity) continue;

elements.push(entity); // Add the Revit class name to the list of elements.

}

// Skip the specification if there are no elements. There is no point in creating configurations for properties that are not assigned to any element.

if (elements.length === 0) continue;

// Iterate through the requirements again to create the IFC export configuration.

for (const req of spec.requirements) {

// Check if the requirement is of type property.

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

// Extract data type, property set, and base name.

const { dataType, propertySet, baseName } = req;

// Map the IFC data type to a Revit export type.

const exportType = ifcRevitExportMapping[dataType];

// Ensure the propertySet and baseName types are simple and a corresponding export type is found, else skip this iteration. We want direct name extraction and a corresponding type.

if (propertySet.type !== "simple" || baseName.type !== "simple" || !exportType) continue;

// Extract the property set name and property name.

const { parameter: psetName } = propertySet;

const { parameter: propName } = baseName;

// Find an existing property set with the same name.

let set = sets.find(({ name }) => name === psetName);

// If the set doesn't exist, create one.

if (!set) {

set = { name: psetName, elements, props: [] }; //Create a set containing its name, the element which it applies to and an empty list to store properties

sets.push(set); // Push the set to the set list

}

// Create a property.

const prop = { // Create a property using the relevant names and types.

nameInIfc: propName,

type: exportType,

revitName: propName,

}

set.props.push(prop); // Push the property to the set of properties

}

}

// Initialize an array to store lines for the shared parameters file.

const sharedParamsLines = [];

// Iterate through parameters to create lines for the shared parameters file.

for (const param of parameters) {

const values = Object.values(param); // Get the values from the parameter object.

sharedParamsLines.push(values.join("\\t")); // Join the values with a tab character and add the line to the sharedParamsLines array.

}

// Define the text content for the shared parameters file.

const sharedParamsText = `# This is a Revit shared parameter file.

# Do not edit manually.

\*META VERSION MINVERSION

META 2 1

\*GROUP ID NAME

GROUP 1 IFC Parameters

\*PARAM GUID NAME DATATYPE DATACATEGORY GROUP VISIBLE DESCRIPTION USERMODIFIABLE HIDEWHENNOVALUE

${sharedParamsLines.join("\\n")}`;

// Check if the shared parameters file already exists and delete it if it does.

const sharedParamsFileExists = fs.existsSync("./SharedParameters.txt");

if (sharedParamsFileExists) fs.rmSync("./SharedParameters.txt");

// Write the shared parameters text to the file.

fs.writeFileSync("./SharedParameters.txt", sharedParamsText);

// Initialize an array to store lines for the property set file.

const psetLines = [];

// Iterate through sets to create lines for the property set file.

for (const set of sets) {

const { name, elements, props } = set; // Extract the name, elements, and properties from the set object.

const psetLine = `PropertySet:\\t${name}\\tI\\t${elements.join(",")}`; // Create the property set line with the name and elements.

const propLines = []; // Initialize an array to store the lines of text for the properties in the current set.

// Iterate through the properties in the current set to create the lines of text for the properties.

for (const prop of props) {

const { revitName, type, nameInIfc } = prop; // Extract the Revit name, type, and IFC name from the property object.

let propLine = `\\t${nameInIfc}\\t${type}`; // Create the property line with the IFC name and type.

if (revitName) propLine += `\\t${revitName}`; // If a Revit name is specified, add it to the property line.

propLines.push(propLine); // Add the property line to the propLines array.

}

const line = `${psetLine}\\n${propLines.join("\\n")}`; // Combine the property set line and the property lines.

psetLines.push(line); // Add the line to the psetLines array.

}

// Join the property set lines with a newline character.

const userDefinedPsetsText = psetLines.join("\\n");

// Check if the property sets file already exists and delete it if it does.

const psetsFileExists = fs.existsSync("./UserDefinedPsets.txt");

if (psetsFileExists) fs.rmSync("./UserDefinedPsets.txt");

// Write the user defined property sets text to the file.

fs.writeFileSync("./UserDefinedPsets.txt", userDefinedPsetsText);

}

// Read the contents of the IDS file.

const requirements = fs.readFileSync("./requirements.ids", "utf8");

// Load the IDS content into the IDS component.

ids.load(requirements);

// Call the function to convert the IDS parameters.

convertIdsParameters()

As you saw, we used some mappings between IFC and Revit data. In the mappings.js file (if you don’t have the file just create it) write the following:

// \*Common Revit Exporting Types

// Text

// Integer

// Real

// Length

// Volume

// Boolean

export const ifcRevitExportMapping = {

IFCDURATION: "Real",

IFCBOOLEAN: "Boolean",

IFCLABEL: "Text",

}

export const ifcClassMapping = {

IFCWALL: "IfcWall"

}

// \*Common Revit Data Types

// TEXT

// INTEGER

// ANGLE

// AREA

// DISTANCE

// LENGTH

// MASS\_DENSITY

// NUMBER

// SLOPE

// VOLUME

// URL

// YESNO

export const ifcRevitTypeMapping = {

IFCDURATION: "NUMBER",

IFCBOOLEAN: "YESNO",

IFCLABEL: "TEXT"

}

Enjoy your implementation!

**⚔ Quest (code: 7m98g)**

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: 👇

Improve the implementation so it can take multiple IDS files and add all their data to a single SharedParameters file and a single UserDefinedPsets file. With that enhancement, you will process multiple IDS files in no time.

**🦶 Development Steps**

For this quest, replace the following code:

const requirements = fs.readFileSync("./requirements.ids", "utf8");

ids.load(requirements);

With the steps below:

1. Create a new variable that holds a path given as the second argument in the process. Let’s call it folderPath. TIP: You can take the second process argument with process.argv[2].
2. Make sure there is actually a folderPath. In case is not, simply log to the console that there was not path provided and exit the process. You can exit the process with process.exit(1).
3. Read the contents of the path you got from the second argument in the process. Use the file system API to do it.
4. Filter the result from the last step so it only takes into consideration files that ends with .ids
5. Loop through all IDS files found. Inside the loop:
   1. Use the path module (import it first) to join the folderPath with the file name.
   2. Read the IDS file the same as done in the lesson.
   3. Load the IDS file in the component the same as done in the lesson.