Video Instructions

https://www.loom.com/share/a4fb98b1140b4c0c939e495b4575852a?sid=e7e5a4da-36b4-4f7d-8a09-58f6696cb6a3

Convert TSV to CSV

https://products.groupdocs.app/conversion/tsv-to-csv

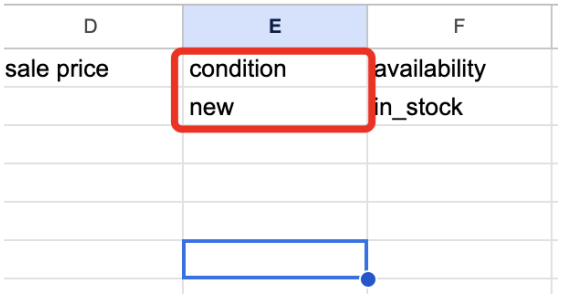
Upload Data To Google Sheets

Create Supplemental Feed Sheet

Re-Format Attributes (Maybe Script Could Do This)

Change in stock to in_stock

Add condition attribute. - *I forget to mention this in the video, but it's important to find the condition column in your google sheet and add the value for this.*



Delete columns:

- Channel
- Language
- All Clicks
- Feed label
- Remove GTIN

```
JavaScript
/**
```

```
 * Configuration for Product Availability Monitor
 * Update these values before running the script
 */
const CONFIG = {
  // The URL of your Google Sheet containing the supplemental feed
  // Example: https://docs.google.com/spreadsheets/d/YOUR_SHEET_ID/edit
  SHEET_URL:
    "[ENTER YOUR SHEET URL]",

  // The email address to receive notifications
  NOTIFICATION_EMAIL: "[ENTER YOUR NOTIFICATION EMAIL]",

  // Logging configuration
  LOGGING: {
    // Set to true to enable detailed logging
    ENABLED: true,
    // Maximum number of log entries to keep
    MAX_LOGS: 1000,
    // Log sheet name (will be created if it doesn't exist)
    SHEET_NAME: "Execution_Logs",
  },

  // Feed processing configuration
  FEED: {
    // The name of the sheet containing the supplemental feed data
    SHEET_NAME: "Sheet1",
    // The suffix used to identify duplicate products
    DUPLICATE_SUFFIX: "dup",
  },

  // Merchant Center configuration
  MERCHANT: {
    // Your Merchant Center account ID
    MERCHANT_ID: "[ENTER YOUR MERCHANT CENTER ID]",

    // Maximum number of products to check in a single batch
    BATCH_SIZE: 20,
  },
};

/**
 * Validates the configuration settings
 * @returns {boolean} True if configuration is valid
 * @throws {Error} If configuration is invalid
```

```
 */
function validateConfig() {
  if (!CONFIG.SHEET_URL || CONFIG.SHEET_URL === "YOUR_SHEET_URL") {
    throw new Error("Please update the SHEET_URL in the CONFIG object");
  }

  if (
    !CONFIG.NOTIFICATION_EMAIL ||
    CONFIG.NOTIFICATION_EMAIL === "your.email@example.com"
  ) {
    throw new Error(
      "Please update the NOTIFICATION_EMAIL in the CONFIG object"
    );
  }

  try {
    const spreadsheet = SpreadsheetApp.openByUrl(CONFIG.SHEET_URL);
    if (!spreadsheet) {
      throw new Error("Unable to access the specified Google Sheet");
    }
  } catch (e) {
    throw new Error(
      "Invalid SHEET_URL or insufficient permissions: " + e.message
    );
  }

  return true;
}

/**
 * Logger class for handling script execution logs
 */
class Logger {
  constructor() {
    this.logs = [];
  }

  /**
   * Adds a log entry
   * @param {string} message - The log message
   * @param {string} level - The log level (INFO, ERROR, WARNING)
   */
  log(message, level = "INFO") {
    if (!CONFIG.LOGGING.ENABLED) return;
```

```
  const timestamp = new Date().toISOString();
  const logEntry = {
    timestamp,
    level,
    message,
  };

  this.logs.push(logEntry);
  console.log(`[${timestamp}] [${level}] ${message}`);

  // Write to log sheet if logs exceed threshold
  if (this.logs.length >= CONFIG.LOGGING.MAX_LOGS) {
    this.flushLogs();
  }
}

/**
 * Writes logs to the logging sheet
 */
flushLogs() {
  if (this.logs.length === 0) return;

  try {
    const spreadsheet = SpreadsheetApp.openByUrl(CONFIG.SHEET_URL);
    let logSheet = spreadsheet.getSheetByName(CONFIG.LOGGING.SHEET_NAME);

    // Create log sheet if it doesn't exist
    if (!logSheet) {
      logSheet = spreadsheet.insertSheet(CONFIG.LOGGING.SHEET_NAME);
      logSheet.appendRow(["Timestamp", "Level", "Message"]);
    }

    // Convert logs to 2D array for batch writing
    const logRows = this.logs.map((log) => [
      log.timestamp,
      log.level,
      log.message,
    ]);

    // Append logs to sheet
    logSheet
      .getRange(logSheet.getLastRow() + 1, 1, logRows.length, 3)
      .setValues(logRows);
```

```
      this.logs = [];
    } catch (e) {
      console.error("Failed to write logs to sheet:", e);
    }
  }
}

// Create global logger instance
const logger = new Logger();

/**
 * Class to handle supplemental feed operations
 */
class SupplementalFeed {
  constructor() {
    this.spreadsheet = SpreadsheetApp.openByUrl(CONFIG.SHEET_URL);
    this.sheet = this.spreadsheet.getSheetByName(CONFIG.FEED.SHEET_NAME);
    if (!this.sheet) {
      throw new Error(
        `Sheet "${CONFIG.FEED.SHEET_NAME}" not found in the spreadsheet`
      );
    }
  }

  /**
   * Gets the column index for a given header name
   * @param {string} headerName - The name of the header to find
   * @returns {number} The column index (1-based)
   * @throws {Error} If header is not found
   */
  getColumnIndex(headerName) {
    const headers = this.sheet
      .getRange(1, 1, 1, this.sheet.getLastColumn())
      .getValues()[0];
    const columnIndex = headers.indexOf(headerName) + 1;

    if (columnIndex === 0) {
      throw new Error(`Header "${headerName}" not found in the sheet`);
    }

    return columnIndex;
  }
```

```
/**
 * Reads product data from the supplemental feed
 * @returns {Array<Object>} Array of product objects with their data
 */
readProductData() {
  logger.log("Reading product data from supplemental feed");

  try {
    // Get all data from the sheet
    const data = this.sheet.getDataRange().getValues();
    const headers = data[0];

    // Find required column indices
    const idColumnIndex = headers.indexOf("id");
    const availabilityColumnIndex = headers.indexOf("availability");

    if (idColumnIndex === -1 || availabilityColumnIndex === -1) {
      throw new Error(
        "Required columns 'id' and 'availability' not found in sheet"
      );
    }

    // Process each row (excluding header)
    const products = data
      .slice(1)
      .filter((row) => row[idColumnIndex]) // Filter out empty rows
      .map((row) => ({
        id: row[idColumnIndex].toString(),
        availability: row[availabilityColumnIndex],
        rowIndex: data.indexOf(row) + 1,
      }));

    logger.log(`Found ${products.length} products in supplemental feed`);
    return products;
  } catch (e) {
    logger.log(`Error reading product data: ${e.message}`, "ERROR");
    throw e;
  }
}

/**
 * Maps duplicate products to their original product IDs
 * @param {Array<Object>} products - Array of products from the feed
 * @returns {Object} Mapping of duplicate product IDs to original product IDs
```

```
   */
  mapDuplicateProducts(products) {
    logger.log("Mapping duplicate products to originals");

    const productMapping = {};
    const duplicateProducts = products.filter((product) =>
      product.id.endsWith(CONFIG.FEED.DUPLICATE_SUFFIX)
    );

    duplicateProducts.forEach((product) => {
      const originalId = product.id.slice(
        0,
        -CONFIG.FEED.DUPLICATE_SUFFIX.length
      );
      productMapping[product.id] = {
        originalId,
        rowIndex: product.rowIndex,
      };
    });

    logger.log(
      `Found ${Object.keys(productMapping).length} duplicate products`
    );
    return productMapping;
  }

  /**
   * Updates availability for specified products
   * @param {Array<Object>} updates - Array of updates with product IDs and new
availability
   * @returns {boolean} Success status
   */
  updateAvailability(updates) {
    logger.log(`Updating availability for ${updates.length} products`);

    try {
      const availabilityColumn = this.getColumnIndex("availability");

      updates.forEach((update) => {
        this.sheet
          .getRange(update.rowIndex, availabilityColumn)
          .setValue(update.availability);
      });
```

```
      logger.log("Successfully updated product availability");
      return true;
    } catch (e) {
      logger.log(`Error updating availability: ${e.message}`, "ERROR");
      return false;
    }
  }
}

/**
 * Class to handle Merchant Center operations
 */
class MerchantCenter {
  constructor() {
    try {
      this.merchantId = CONFIG.MERCHANT.MERCHANT_ID;
    } catch (e) {
      logger.log(`Merchant Center initialization error: ${e.message}`,
"ERROR");
      throw e;
    }
  }

  /**
   * Checks availability status for products in Merchant Center
   * @param {Array<string>} productIds - Array of product IDs to check
   * @returns {Object} Map of product IDs to their availability status
   */
  checkProductsAvailability(productIds) {
    logger.log(
      `Checking availability for ${productIds.length} products in Merchant
Center`
    );
    logger.log(`Product IDs to check: ${productIds.join(", ")}`);

    const availabilityMap = {};

    try {
      // Format product IDs with merchant prefix
      const formattedIds = productIds.map((id) => `online:en:GB:${id}`);

      // Build query to get product availability
      const queryString = `SELECT product_view.id, product_view.availability
        FROM ProductView
```

```
        WHERE product_view.id IN ('${formattedIds.join("','")}')`;

      logger.log(`Executing query: ${queryString}`);

      const resource = {
        query: queryString,
        pageSize: CONFIG.MERCHANT.BATCH_SIZE,
      };

      // Execute the query
      const productReports = ShoppingContent.Reports.search(
        resource,
        this.merchantId
      );

      if (productReports.results && productReports.results.length > 0) {
        for (const productReport of productReports.results) {
          // Extract the original product ID from the full ID
          const productId = productReport.productView.id.split(":").pop();
          const availability = productReport.productView.availability;
          availabilityMap[productId] = availability;
          logger.log(`Product ${productId} availability: ${availability}`);
        }
      } else {
        logger.log("No products found in Merchant Center", "WARNING");
      }
    } catch (e) {
      logger.log(`Error checking product availability: ${e.message}`, "ERROR");
      logger.log(`Error details: ${JSON.stringify(e)}`, "ERROR");
    }

    const foundProducts = Object.keys(availabilityMap).length;
    logger.log(
      `Found availability status for ${foundProducts} out of
${productIds.length} products`
    );

    return availabilityMap;
  }
}

/**
 * Main function to be triggered by time-based trigger
 */
```

```
function main() {
  try {
    validateConfig();
    logger.log("Script execution started");

    // Process supplemental feed
    const feed = new SupplementalFeed();
    const products = feed.readProductData();
    const productMapping = feed.mapDuplicateProducts(products);
    logger.log(
      `Found ${Object.keys(productMapping).length} products to monitor`
    );

    // Check original products in Merchant Center
    const merchantCenter = new MerchantCenter();
    const originalProductIds = [
      ...new Set(
        Object.values(productMapping).map((product) => product.originalId)
      ),
    ];

    const availabilityStatus =
      merchantCenter.checkProductsAvailability(originalProductIds);
    logger.log(`Retrieved availability status from Merchant Center`);

    // Process availability changes and prepare updates
    const updates = [];
    for (const [dupProductId, mapping] of Object.entries(productMapping)) {
      const originalId = mapping.originalId;
      const merchantAvailability = availabilityStatus[originalId];

      // Get current availability from supplemental feed
      const currentProduct = products.find((p) => p.id === dupProductId);
      const currentAvailability = currentProduct
        ? currentProduct.availability
        : "";

      logger.log(
        `Checking product ${dupProductId}: Current availability:
'${currentAvailability}', Merchant availability: '${merchantAvailability}'`
      );

      // Normalize merchant availability to lowercase for comparison
      const normalizedMerchantAvailability = merchantAvailability
```

```
        ? merchantAvailability.toLowerCase()
        : "";

      // Update if current availability is blank or different from merchant
availability
      if (currentAvailability !== normalizedMerchantAvailability) {
        logger.log(
          `Updating ${dupProductId} availability from '${currentAvailability}'
to '${normalizedMerchantAvailability}'`
        );
        updates.push({
          rowIndex: mapping.rowIndex,
          availability: normalizedMerchantAvailability,
        });
      } else {
        logger.log(
          `No update needed for ${dupProductId} (current:
'${currentAvailability}')`
        );
      }
    }

    // Update supplemental feed if there are any changes
    if (updates.length > 0) {
      logger.log(
        `Updating availability for ${updates.length} products in supplemental
feed`
      );
      const updateSuccess = feed.updateAvailability(updates);

      if (updateSuccess) {
        logger.log(
          "Successfully updated product availability in supplemental feed"
        );
      } else {
        logger.log(
          "Failed to update product availability in supplemental feed",
          "ERROR"
        );
      }
    } else {
      logger.log("No availability updates needed");
    }
  } catch (e) {
```

```
    logger.log(e.message, "ERROR");
    throw e;
  }
}

/**
 * Test function to verify availability update logic
 */
function testAvailabilityUpdate() {
  try {
    logger.log("Starting availability update test");

    // Test with sample data
    const feed = new SupplementalFeed();
    const products = feed.readProductData();
    const productMapping = feed.mapDuplicateProducts(products);

    // Get actual availability from Merchant Center
    const merchantCenter = new MerchantCenter();
    const originalProductIds = [
      ...new Set(
        Object.values(productMapping).map((product) => product.originalId)
      ),
    ];

    const availabilityStatus =
      merchantCenter.checkProductsAvailability(originalProductIds);

    // Log the results
    logger.log("Product availability status:");
    for (const [dupProductId, mapping] of Object.entries(productMapping)) {
      const originalId = mapping.originalId;
      const availability = availabilityStatus[originalId];
      logger.log(
        `Original product ${originalId} (${availability}) -> Duplicate
${dupProductId}`
      );
    }

    logger.log("Availability update test completed");
  } catch (e) {
    logger.log(`Test failed: ${e.message}`, "ERROR");
    throw e;
  }
```

```
}
```