User Manual for Ontology-based Knowledge Graph Application:

1. Overview of the Ontology Schema

This ontology models the relationships between Items, Parts, Procedures, Tools, Steps, and Images. Below are the key components and their relationships:

Classes:

- **Item**: Represents a complete product (e.g., a phone).
- **Part**: Represents components or pieces of an item (e.g., a battery or screen).
- **Procedure**: Refers to a set of steps that outline how to repair or replace parts (e.g., "Battery Replacement").
- **Sub-procedure**: A procedure related to the same item or a part of that item.
- **Tool**: Represents tools required for a procedure (e.g., a screwdriver or spudger).
- **Step**: Each procedure consists of multiple steps describing detailed actions.
- **Image**: Visual aids (like photos) associated with steps for clearer instructions.

Relationships Between Classes:

- **Item ↔ Part**: An Item has Parts (e.g., a phone has a battery) using `has_part` / `is_part_of`.
- **Item/Part ↔ Procedure**: An Item or Part may have a Procedure associated with it (e.g., repair instructions) via `has_procedure` / `is_procedure_for`.
- **Procedure ↔ Tool**: A Procedure requires certain Tools through `uses_tool` / `is_used_by`.
- **Procedure ↔ Step**: Each Procedure contains multiple Steps with `has_step` / `is_step_of`.
- **Step ↔ Image**: Steps may be associated with an Image (e.g., a picture for better instruction) using `has_image` / `is_image_of`.
- **Procedure ↔ Sub-Procedure**: Procedures can have nested or related procedures through `has_sub_procedure` / `is_sub_procedure_of`.

Schema Summary:

- **Classes**: Item, Part, Procedure, Tool, Step, Image.
- **Object Properties**:
  - `has_part`: Relates an Item to its Parts.
  - `has_procedure`: Links Items or Parts to their associated Procedures.
  - `uses_tool`: Specifies the tools required for a Procedure.
  - `has_step`: Specifies the steps within a Procedure.
  - `has_image`: Links a Step to an Image for visual guidance.
  - `has_sub_procedure`: Indicates related or nested Procedures.
- **Data Properties**:
  - `step_text`: Contains text instructions for a step.
  - `mentioned_tools`: Lists tools mentioned in a step's description.

Example Use Case:

- For a phone with parts such as a battery and screen, the ontology links repair instructions (Procedures) to both the phone (Item) and the individual parts (Parts). Each procedure consists of several steps, each of which may include images and mention tools required to complete the repair.
- This structure allows querying for repair procedures, tools used, steps involved, and associated parts of an item.

2. Example Queries

Here are some example SPARQL queries to help you interact with the knowledge graph:

1. Retrieve all parts of an item (e.g., a phone):

```
SELECT DISTINCT ?part
WHERE {
  ?item a <http://example.org/phone_knowledge_graph.owl#Item> .
  ?item <http://example.org/phone_knowledge_graph.owl#has_part> ?part .
}
```

Interpretation: This query lists all the parts associated with items, such as the battery or screen of a phone.

2. List procedures related to a specific part (e.g., battery):

```
SELECT DISTINCT ?procedure
WHERE {
  ?part a <http://example.org/phone_knowledge_graph.owl#Part> .
  ?part <http://example.org/phone_knowledge_graph.owl#has_procedure> ?
procedure .
  FILTER CONTAINS(STR(?part), "Battery")
}
```

Interpretation: This query finds procedures like “Battery Replacement” for the given part.

3. Find tools used in a specific procedure (e.g., Screen Replacement):

```
SELECT ?tool
WHERE {
  ?procedure a <http://example.org/phone_knowledge_graph.owl#Procedure>
.
  ?procedure <http://example.org/phone_knowledge_graph.owl#uses_tool> ?
tool .
  FILTER(CONTAINS(STR(?procedure), "Screen_Replacement"))
}
```

Interpretation: The result shows tools needed, such as a screwdriver.

4. Retrieve all steps and corresponding images for a procedure:

```
SELECT ?stepText ?image
WHERE {
  ?procedure a <http://example.org/phone_knowledge_graph.owl#Procedure>
.
  ?procedure <http://example.org/phone_knowledge_graph.owl#has_step> ?
step .
  ?step <http://example.org/phone_knowledge_graph.owl#step_text> ?
stepText .
  OPTIONAL { ?step
<http://example.org/phone_knowledge_graph.owl#has_image> ?image . }
  FILTER(CONTAINS(STR(?procedure), "Screen_Replacement"))
}
```

Interpretation: Lists steps for a procedure with associated images for better visualization.

3. Setting Up the Environment

Before creating the knowledge graph and running the web application, you need to set up a Python virtual environment and install the required dependencies.

Setting Up a Virtual Environment

- A virtual environment (`venv`) isolates the project's dependencies from the system's Python packages, making it easier to manage and avoid conflicts.
- Steps to create and activate a virtual environment:

1. Create the virtual environment:

Run the following command from the root of the project directory `CITS3005-iFixit-Project` to create a virtual environment called `venv`:

```
python -m venv venv
```

This will create a folder named venv containing the isolated Python environment.

2. Activate the virtual environment:

- On Windows, run:

```
venv\Scripts\activate
```

- On macOS/Linux, run:

```
source venv/bin/activate
```

Once activated, your terminal prompt should change to reflect the virtual environment, e.g., `(venv)`.

3. Deactivate the virtual environment (when done):

To deactivate the virtual environment, simply run:

```
deactivate
```

Installing the Required Dependencies

Once the virtual environment is activated, you can install the project dependencies.

Install dependencies: Run the following command to install all required Python libraries from the `requirements.txt` file:

```
pip install -r requirements.txt
```

This will install libraries such as `Flask` (for the web application), `Owlready2` (for managing the ontology), and `RDFlib` (for SPARQL queries).

4. Creating the Knowledge Graph

Once the environment is set up and the dependencies are installed, you can proceed to create the knowledge graph using the following scripts:

1. Defining the Ontology (`define_ontology.py`)

- This script defines the structure of the ontology, specifying classes like `Item`, `Part`, `Procedure`, `Tool`, `Step`, and `Image`.
- It also defines object properties (e.g., `has_part`, `has_procedure`, `uses_tool`) and datatype properties (e.g., `step_text`, `mentioned_tools`).

- The ontology is saved to the file `ontology/phone_ontology.owl`.

Run the following command to define the ontology:

```
python scripts/define_ontology.py
```

2. Populating the Knowledge Graph (`populate_ontology.py`)

- This script populates the ontology with actual data from the `data/Phone.json file`.
- It creates individuals (instances) for each `Item`, `Part`, `Procedure`, `Tool`, `Step`, and `Image` and links them based on the ontology structure.
- The output is saved in `ontology/phone_knowledge_graph.owl`.

To populate the graph with data, run:

```
python scripts/populate_ontology.py
```

5. Using the Web Application

Once the knowledge graph is set up, you can interact with it through the web application.

1. Running the Flask Application

With the virtual environment activated and the knowledge graph created, run the Flask app by executing the following command in your terminal:

```
python run.py
```

This will start a local server where you can interact with the knowledge graph through a web interface.

2. Navigating the Application

- `Browse`: Also the `home` route, view the items, procedures, and parts displayed on the browse page, allowing you to navigate the relationships between them by visiting the specific procedure.
- `Validate Data`: Identify inconsistencies or missing links in the data, such as tools used in a procedure but not mentioned in any steps.
- `Search`: Enter SPARQL queries or select predefined queries to explore the knowledge graph. If both a custom query and a predefined query are provided, the predefined query will take precedence.

6. Managing Data in the Knowledge Graph

Adding New Data

To add new data to the knowledge graph, follow these steps:

1. Add new data to the JSON Data File (`Phone.json`):

- All new data must be added to the `Phone.json` file in the `data/` directory.
- Ensure that the new data includes details for Items, Parts, Procedures, Steps, and Tools.
- Each entry should follow the same structure as the existing records in the JSON file. For example:

```
{
  "Title": "iPhone 12 Screen Replacement",
  "Category": "iPhone 12",
  "Toolbox": [
      {"Name": "screwdriver", "Url": "https://example.com/screwdriver"},
      {"Name": "spudger", "Url": "https://example.com/spudger"}
  ],
  "Steps": [
      {"Order": 1, "Text_raw": "Remove the screws with a screwdriver."},
      {"Order": 2, "Text_raw": "Use a spudger to pry open the case."}
  ],
  "Subject": "Screen"
}
```

2. Rerun the Scripts to Populate the Graph:

Once you've added data to the Phone.json file, rerun the population script to update the knowledge graph:

```
python scripts/populate_ontology.py
```

3. Check for Errors:

Use the Validate Data option in the web application to ensure no inconsistencies exist in the populated data (e.g., tools not mentioned in steps).

Modifying Existing Data

Modifying data in the knowledge graph follows a similar process to adding new data. If you need to update an existing record:

1. Modify the JSON Data File:

- Find the existing record you wish to modify in the `Phone.json` file.
- For example, if you need to update the steps of a procedure, modify the `Steps` field as needed:

```
{
  "Title": "iPhone 12 Screen Replacement",
  "Category": "iPhone 12",
  "Toolbox": [
      {"Name": "screwdriver", "Url": "https://example.com/screwdriver"},
      {"Name": "spudger", "Url": "https://example.com/spudger"}
  ],
  "Steps": [
      {"Order": 1, "Text_raw": "Remove the screws with a screwdriver."},
      {"Order": 2, "Text_raw": "Use a spudger to pry open the case."},
      {"Order": 3, "Text_raw": "Remove the screen using a suction tool."
  ],
  "Subject": "Screen"
}
```

2. Rerun the Scripts to Populate the Graph:

After making changes to the JSON file, rerun the population script:

```
python scripts/populate_ontology.py
```

3. Validate the Changes:

Use the Validate Data option to check that the modifications have been applied correctly and there are no inconsistencies in the updated data.

Deleting Data

To delete data from the knowledge graph, follow these steps:

1. Remove the Entry from the JSON Data File:

- Locate the entry (item, part, procedure) that you want to delete in the `Phone.json` file and remove it completely from the file
- Example before deletion:

```
{
  "Title": "iPhone 12 Screen Replacement",
  "Category": "iPhone 12",
  "Toolbox": [
      {"Name": "screwdriver", "Url": "https://example.com/screwdriver"},
      {"Name": "spudger", "Url": "https://example.com/spudger"}
  ],
  "Steps": [
      {"Order": 1, "Text_raw": "Remove the screws with a screwdriver."},
      {"Order": 2, "Text_raw": "Use a spudger to pry open the case."}
  ],
  "Subject": "Screen"
}
```

- After deletion: Simply delete the entire entry for `iPhone 12 Screen Replacement`.

2. Rerun the Population Script:

After removing the entry from the JSON file, rerun the population script to update the graph and reflect the deletion:

```
python scripts/populate_ontology.py
```

3. Check for Inconsistencies:

Use the Validate Data option to ensure there are no leftover references to the deleted data in the knowledge graph (e.g., a procedure that referenced a deleted part).

4. Alternative: Deleting Data Programmatically

- If you want to delete data programmatically (instead of manually removing it from the JSON file), modify the RDFLib graph by running custom Python scripts.
- Example Python code for deleting an individual from the knowledge graph:

```
from rdflib import Graph, URIRef

# Load the knowledge graph
g = Graph()
g.parse("ontology/phone_knowledge_graph.owl")

# Define the URI of the individual to delete
individual_uri = URIRef("http://example.org/phone_knowledge_graph.owl#iP

# Remove all triples associated with the individual
g.remove((individual_uri, None, None))

# Save the updated graph
g.serialize(destination="ontology/phone_knowledge_graph.owl", format="xm
```

- After running this script, rerun the web application or scripts to reflect the changes.

7. Adding or Modifying Ontology Rules

Adding New Rules

To extend the ontology, you can add new rules using OWL. For example:

Enforce that every part must have at least one procedure:

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_procedure"/>
  <owl:minCardinality>1</owl:minCardinality>
</owl:Restriction>
```

This rule ensures that each `Part` in the ontology has at least one associated procedure.

Updating Ontology Rules

1. Modify the RDF/OWL XML File:

- To update existing rules, directly modify the `phone_ontology.owl` file in the ontology/ directory.
- Ensure that your updates remain consistent with the existing schema to avoid conflicts or inconsistencies.

2. Re-upload the Updated OWL File:

- After making changes to the ontology, re-upload the updated OWL file into the system.
- You can reload the ontology by running the `define_ontology.py` script:

```
python scripts/define_ontology.py
```

3. Verify the changes

Use the web app's `Search` or `Validate Data` options to ensure the new rules are applied and working correctly.

8. Troubleshooting and FAQ

`Q: Why is my query not returning results?`

- Possible Cause: The entities or relationships you are querying may not exist in the graph.
- Solution: Verify that the data has been correctly populated and that the ontology includes the expected relationships. Adjust the query filters if necessary, and use simpler queries to test the graph before applying more complex filters.

`Q: How do I restore the original ontology?`

- Solution: To restore the original ontology, re-load the original `phone_ontology.owl` file. You can do this by either running the `define_ontology.py` script again or by manually re-uploading the OWL/XML file into the system if you're using a custom interface.

```
python scripts/define_ontology.py
```

`Q: Can I extend the ontology with new classes?`

- Solution: Yes, you can extend the ontology by adding new classes and properties to the OWL/XML file. Be sure to follow the existing structure and avoid introducing conflicts. Once changes are made, reload the ontology by running the `define_ontology.py` script to ensure the new classes and properties are applied.

9. Best Practices:

- `Use Descriptive Labels`: Ensure that new items, procedures, and parts have clear and meaningful names. This will make querying and understanding the knowledge graph much easier.
- `Maintain Ontology Consistency`: Always follow the schema rules when adding or modifying data. This ensures that the relationships between classes (e.g., Item, Procedure, Part) remain valid and prevent inconsistencies that could lead to incorrect results or errors.

- `Validate After Each Update`: After adding new data or modifying the ontology, use the web application's Validate Data functionality to check for any errors or inconsistencies. This ensures the ontology remains valid and usable.
- `Keep the OWL File Updated`: If you make significant updates to the ontology schema (such as adding new classes or properties), ensure these changes are reflected in the `phone_ontology.owl` file. This keeps the ontology current and avoids issues when running queries.