Resource Description Framework

```
Mix.install([
  {:rdf, "~> 1.1"},
  {:sparql, "~> 0.3"},
  {:kino, "~> 0.10.0"}
])
```

Context

Links:

- RDF module for Elixir
- SparQL module for Elixir

Convenience

```
import RDF.Sigils
```

```
alias RDF.NS.{RDFS}
```

Helper function to display resultsets in a pretty way:

```
result_as_table = fn ro ->
  cols = ro.variables
  results = ro.results

  lines =
    results
    |> List.foldl("", fn result, acc ->
      main =
        cols
        |> Enum.map(fn col -> "#{Map.get(result, col)}" end)
        |> Enum.join(" | ")

      acc <> "| #{main} |\n"
    end)

  Kino.Markdown.new("""
  | #{cols |> Enum.join(" | ")} |
  | #{cols |> Enum.map(fn _ -> "--" end) |> Enum.join(" | ")} |
  #{lines}
  """)
```

```
end

result_as_tree = fn ro ->
  cols = ro.variables
  results = ro.results

  contents =
    List.foldl(results, "graph LR;\n", fn result, acc ->
      src =
        Map.get(result, Enum.at(cols, 0))
        |> RDF.IRI.to_string()
        |> String.split("#")
        |> Enum.at(-1)

      dst =
        Map.get(result, Enum.at(cols, 1))
        |> RDF.IRI.to_string()
        |> String.split("#")
        |> Enum.at(-1)

      acc <> "  #{src}-->#{dst};\n"
    end)

  Kino.Mermaid.new(contents)
end
```

Schema

Vocabulary:

```
defmodule Schema do
  use RDF.Vocabulary.Namespace

  @base "https://github.com/aslakjohansen/livebook-
demos/blob/main/rdf.livemd/2024/schemas/"

  defvocab(Data,
    base_iri: "#{@base}data#",
    terms: ~w[data Data hist live]
  )

  defvocab(Unit,
    base_iri: "#{@base}unit#",
    terms: ~w[unit Unit Unitless DegreesCelcius DegreesFahrenheit Kelvin
Percentage]
  )

  defvocab(Location,
    base_iri: "#{@base}location#",
    terms: ~w[Location Building Floor Room contains area name adjacent]
  )
  defvocab(Modality,
    base_iri: "#{@base}modality#",
    terms: ~w[modality provides controls Modality Occupancy Temperature
RelativeHumidity AbsoluteHumidity TemperatureSetpoint]
  )

  defvocab(Point,
    base_iri: "#{@base}point#",
    terms: ~w[Point Sensor PIR TemperatureSensor HumiditySensor
TemperatureActuator Thermostat]
  )

  def base(), do: @base
end
```

Relationships:

```
schema = [
  # data
  Schema.Data.Data
  |> RDFS.subClassOf(RDFS.Class),

  # unit
  Schema.Unit.Unit
  |> RDFS.subClassOf(RDFS.Class),
  Schema.Unit.Unitless
  |> RDFS.subClassOf(Schema.Unit.Unit),
  Schema.Unit.DegreesCelcius
  |> RDFS.subClassOf(Schema.Unit.Unit),
  Schema.Unit.DegreesFahrenheit
  |> RDFS.subClassOf(Schema.Unit.Unit),
  Schema.Unit.Kelvin
  |> RDFS.subClassOf(Schema.Unit.Unit),
  Schema.Unit.Percentage
  |> RDFS.subClassOf(Schema.Unit.Unit),

  # location
  Schema.Location.Location
  |> RDFS.subClassOf(RDFS.Class),
  Schema.Location.Building
  |> RDFS.subClassOf(Schema.Location.Location),
  Schema.Location.Floor
  |> RDFS.subClassOf(Schema.Location.Location),
  Schema.Location.Room
  |> RDFS.subClassOf(Schema.Location.Location),

  # modality
  Schema.Modality.Modality
  |> RDFS.subClassOf(RDFS.Class),
  Schema.Modality.Occupancy
  |> RDFS.subClassOf(Schema.Modality.Modality),
  Schema.Modality.Temperature
  |> RDFS.subClassOf(Schema.Modality.Modality),
  Schema.Modality.RelativeHumidity
  |> RDFS.subClassOf(Schema.Modality.Modality),
  Schema.Modality.AbsoluteHumidity
  |> RDFS.subClassOf(Schema.Modality.Modality),

  # point
  Schema.Point.Point
  |> RDFS.subClassOf(RDFS.Class),
  Schema.Point.Sensor
  |> RDFS.subClassOf(Schema.Point.Point),
  Schema.Point.PIR
  |> RDFS.subClassOf(Schema.Point.Sensor),
  Schema.Point.TemperatureSensor
  |> RDFS.subClassOf(Schema.Point.Sensor),
  Schema.Point.HumiditySensor
  |> RDFS.subClassOf(Schema.Point.Sensor),

  # Building
  Schema.Location.Building
  |> RDFS.subClassOf(Schema.Location.Location),


  # Quest 5
  Schema.Point.TemperatureActuator
  |> RDFS.subClassOf(Schema.Point.Point),
  Schema.Point.Thermostat
  |> RDFS.subClassOf(Schema.Point.Point),
  Schema.Modality.TemperatureSetpoint
  |> RDFS.subClassOf(Schema.Modality.Modality)
]
```

Model

```
premodel = "https://github.com/aslakjohansen/livebook-
demos/blob/main/rdf.livemd"
model = "#{premodel}#"

# namespaces to bind
ns = [
  rdfs: RDFS,
  data: Schema.Data,
  unit: Schema.Unit,
  location: Schema.Location,
  modality: Schema.Modality,
  point: Schema.Point,
  model: model
]

m = RDF.Graph.new(prefixes: ns)
```

Instances:

Variables for relevant entities:

```
building = ~i"#{model}building1"
floor = ~i"#{model}floor3"
roomA = ~i"#{model}roomA"
roomAocc = ~i"#{model}roomA/occ"
roomApir = ~i"#{model}roomA/pir"
roomApirData = ~i"#{model}roomA/pir/data"
roomAtemp = ~i"#{model}roomA/temp"
roomAtempData = ~i"#{model}roomA/temp/data"
roomAtc = ~i"#{model}roomA/tc"
roomAtcData = ~i"#{model}roomA/tc/data"
roomArhum = ~i"#{model}roomA/rhum"
roomAhum = ~i"#{model}roomA/hum"
roomAhumData = ~i"#{model}roomA/hum/data"
roomB = ~i"#{model}roomB"
roomBocc = ~i"#{model}roomB/occ"
roomBpir = ~i"#{model}roomB/pir"
roomBpirData = ~i"#{model}roomB/pir/data"
roomBtemp = ~i"#{model}roomB/temp"
roomBtempData = ~i"#{model}roomB/temp/data"
roomBtc = ~i"#{model}roomB/tc"
roomBtcData = ~i"#{model}roomB/tc/data"
roomBrhum = ~i"#{model}roomB/rhum"
roomBhum = ~i"#{model}roomB/hum"
roomBhumData = ~i"#{model}roomB/hum/data"
# Quest 5
roomA_actuator = ~i"#{model}roomA/temp_actuator"
roomA_thermostat = ~i"#{model}roomA/thermostat"
roomA_setpoint = ~i"#{model}roomA/setpoint"
roomA_setpoint_data = ~i"#{model}roomA/setpoint/data"

roomB_actuator = ~i"#{model}roomB/temp_actuator"
roomB_thermostat = ~i"#{model}roomB/thermostat"
roomB_setpoint = ~i"#{model}roomB/setpoint"
roomB_setpoint_data = ~i"#{model}roomB/setpoint/data"
```

Add relationships:

```
model =
  schema ++
    [

      # Buildings
      building
      |> RDF.type(Schema.Location.Building)
      |> Schema.Location.name(~L"SDU Main Building")
```

```
         |> Schema.Location.contains(floor),

         # floors
         floor
         |> RDF.type(Schema.Location.Floor)
         |> Schema.Location.name(~L"3rd Floor")
         |> Schema.Location.contains(roomA)
         |> Schema.Location.contains(roomB),


         # rooms
         roomA
         |> RDF.type(Schema.Location.Room)
         |> Schema.Location.area(~L"17")
         |> Schema.Modality.modality(roomAocc)
         |> Schema.Modality.modality(roomAtemp)
         |> Schema.Modality.modality(roomArhum)
         |> Schema.Location.adjacent(roomB),
         roomB
         |> RDF.type(Schema.Location.Room)
         |> Schema.Location.area(~L"23")
         |> Schema.Modality.modality(roomBocc)
         |> Schema.Modality.modality(roomBtemp)
         |> Schema.Modality.modality(roomBrhum)
         |> Schema.Location.adjacent(roomA),

         # modalities
         roomAocc
         |> RDF.type(Schema.Modality.Occupancy),
         roomAtemp
         |> RDF.type(Schema.Modality.Temperature),
         roomArhum
         |> RDF.type(Schema.Modality.RelativeHumidity),
         roomBocc
         |> RDF.type(Schema.Modality.Occupancy),
         roomBtemp
         |> RDF.type(Schema.Modality.Temperature),
         roomBrhum
         |> RDF.type(Schema.Modality.RelativeHumidity),

         # pir sensors
         roomApir
         |> RDF.type(Schema.Point.PIR)
         |> Schema.Unit.unit(Schema.Unit.Unitless)
         |> Schema.Data.data(roomApirData)
         |> Schema.Modality.provides(roomAocc),
         roomBpir
         |> RDF.type(Schema.Point.PIR)
         |> Schema.Unit.unit(Schema.Unit.Unitless)
         |> Schema.Data.data(roomBpirData)
         |> Schema.Modality.provides(roomBocc),

         # temperature sensors
         roomAtemp
```

```
        |> RDF.type(Schema.Point.TemperatureSensor)
        |> Schema.Unit.unit(Schema.Unit.DegreesCelcius)
        |> Schema.Data.data(roomAtempData)
        |> Schema.Modality.provides(roomAtemp),
        roomBtemp
        |> RDF.type(Schema.Point.TemperatureSensor)
        |> Schema.Unit.unit(Schema.Unit.Kelvin)
        |> Schema.Data.data(roomBtempData)
        |> Schema.Modality.provides(roomBtemp),

        # humidity sensors
        roomAhum
        |> RDF.type(Schema.Point.HumiditySensor)
        |> Schema.Unit.unit(Schema.Unit.Percentage)
        |> Schema.Data.data(roomAhumData)
        |> Schema.Modality.provides(roomArhum),
        roomBhum
        |> RDF.type(Schema.Point.HumiditySensor)
        |> Schema.Unit.unit(Schema.Unit.Percentage)
        |> Schema.Data.data(roomBhumData)
        |> Schema.Modality.provides(roomBrhum),

        # data
        roomApirData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"24")
        |> Schema.Data.live(~L"5a3573c3"),
        roomBpirData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"28")
        |> Schema.Data.live(~L"92d17015"),
        roomAtempData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"12")
        |> Schema.Data.live(~L"17c5ae15"),
        roomBtempData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"37")
        |> Schema.Data.live(~L"b0bc97af"),
        roomAhumData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"13")
        |> Schema.Data.live(~L"65d0be73"),
        roomBhumData
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.hist(~L"22")
        |> Schema.Data.live(~L"06ef2490"),

        # Room A actuator and thermostat
        roomA_actuator
        |> RDF.type(Schema.Point.TemperatureActuator)
        |> Schema.Unit.unit(Schema.Unit.DegreesCelcius)
        |> Schema.Modality.controls(roomAtemp),  # controls the temperature
modality
```

```
        roomA_thermostat
        |> RDF.type(Schema.Point.Thermostat)
        |> Schema.Unit.unit(Schema.Unit.DegreesCelcius)
        |> Schema.Data.data(roomA_setpoint_data),

        # Room A setpoint modality
        roomA_setpoint
        |> RDF.type(Schema.Modality.TemperatureSetpoint),

        # Room A setpoint data
        roomA_setpoint_data
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.live(~L"setpoint_a1"),

        # Add setpoint modality to room A
        roomA
        |> Schema.Modality.modality(roomA_setpoint),

        # Connect thermostat to setpoint
        roomA_thermostat
        |> Schema.Modality.provides(roomA_setpoint),

        # Room B actuator and thermostat
        roomB_actuator
        |> RDF.type(Schema.Point.TemperatureActuator)
        |> Schema.Unit.unit(Schema.Unit.Kelvin)
        |> Schema.Modality.controls(roomBtemp),

        roomB_thermostat
        |> RDF.type(Schema.Point.Thermostat)
        |> Schema.Unit.unit(Schema.Unit.Kelvin)
        |> Schema.Data.data(roomB_setpoint_data),

        # Room B setpoint modality
        roomB_setpoint
        |> RDF.type(Schema.Modality.TemperatureSetpoint),

        # Room B setpoint data
        roomB_setpoint_data
        |> RDF.type(Schema.Data.Data)
        |> Schema.Data.live(~L"setpoint_b1"),

        # Add setpoint modality to room B
        roomB
        |> Schema.Modality.modality(roomB_setpoint),

        # Connect thermostat to setpoint
        roomB_thermostat
        |> Schema.Modality.provides(roomB_setpoint)
      ]
```

```
model = model |> RDF.Graph.new() |> RDF.Graph.add_prefixes(ns)
```

Serialize

If we want to store the file to disk:

```
model |> RDF.Turtle.write_string!() |> IO.puts()
```

Queries

Fetch the proper base IRI of the schema location:

```
base = Schema.base()
```

Room sizes:

```
q = "
  PREFIX location: <#{base}location#>

  SELECT ?area
  WHERE {
    ?room a location:Room .
    ?room location:area ?area .
  }
"

SPARQL.execute_query(model, q)
|> result_as_table.()
```

To create a query like the one above, you need a prefix this defines the shorthand uri, this is done to have an easy way to refrence specifics inside the schema. location here referes to the location inside the schema.

The section "select ?area" means you want to retrive data into a sparql variable

Where statemented is the conditions for selecting data

this is the triple pattern in the where statment : ?room a location:Room . the ?room is a variable you want to save, the "a" is a rdf type definition and tells rdf what type the variable room should be, the Room type is grabbed through the namespace and the location name space contains rooms. the "." signals the end of a statement

the statment "?room location:area ?area ." is used on the rooms we grabed from the elaier statsment, and moves us along the edge to get the node area connected to the type room. this how queries are generally executed.

Per-floor combinations of temperature and relative humidity data:

```
q = "
  PREFIX location: <#{base}location#>
  PREFIX modality: <#{base}modality#>

  SELECT ?name ?temp ?rhum
  WHERE {
    ?floor a location:Floor .
    ?room a location:Room .
    ?temp_mod a modality:Temperature .
    ?rhum_mod a modality:RelativeHumidity .

    ?floor location:name ?name .
    ?floor location:contains ?room .

    ?room modality:modality ?temp_mod .
    ?temp modality:provides ?temp_mod .

    ?room modality:modality ?rhum_mod .
    ?rhum modality:provides ?rhum_mod .
  }
"

SPARQL.execute_query(model, q)
|> result_as_table.()
```

Class hierarchy:

```
q = "
  SELECT ?sub ?super
  WHERE {
    ?sub rdfs:subClassOf ?super .
  }
"

SPARQL.execute_query(model, q)
|> result_as_tree.()
```

Quest 3

```
q_current = "
PREFIX location: <#{base}location#>
PREFIX modality: <#{base}modality#>
PREFIX point: <#{base}point#>
PREFIX unit: <#{base}unit#>
PREFIX data: <#{base}data#>
```

```
SELECT ?room ?live_temp_reading
WHERE {
    ?floor a location:Floor .
    ?floor location:contains ?room .
    ?room a location:Room .

    ?room modality:modality ?temp_mod .
    ?temp_mod a modality:Temperature .
    ?temp_sensor modality:provides ?temp_mod .
    ?temp_sensor a point:TemperatureSensor .
    ?temp_sensor unit:unit unit:Kelvin .
    ?temp_sensor data:data ?temp_data .
    ?temp_data data:live ?live_temp_reading .
}
"

SPARQL.execute_query(model, q_current)
|> result_as_table.()
```

Here we see a little different query " ?temp_sensor unit:unit unit:Kelvin ." this is still a tripple first we grab the namespace modality, we then get the type temperature and store it inside the temp_mod variable, temp_sensor is used to grab all modalities with the relationship provides and are of class Temperature meaning we get all the modalities assoicated with a room that has temperature in them.

Nex line we say that all temperature sensors we found are of the class temperatureSensor, meaning any modality that is not that will not be included.

next line is the weird line, here we get all temperetaure sensors that has the property unit and of these only the ones that uses kelvin

Lastly we grab the live temp reading and the select works since we start with the room to grab the modalities and only find 1 one sensor reporting in kelvin, if we had found four all in room b we would get room b listed 4 times

Quest 4

```
q_hierarchy = "
PREFIX location: <#{base}location#>
SELECT ?buildingName ?floorName ?room ?roomArea
WHERE {
    ?building a location:Building .
    ?building location:name ?buildingName .
    ?building location:contains ?floor .
    ?floor a location:Floor .
    ?floor location:name ?floorName .
    ?floor location:contains ?room .
    ?room a location:Room .
    ?room location:area ?roomArea .
}
"
```

```
SPARQL.execute_query(model, q_hierarchy)
|> result_as_table.()
```

Quest 5

```
# Step 5: Verification Query
q_verify_actuators = "
PREFIX location: <#{base}location#>
PREFIX modality: <#{base}modality#>
PREFIX point: <#{base}point#>
PREFIX data: <#{base}data#>

SELECT ?room ?actuator ?thermostat ?temp_sensor
WHERE {
    ?room a location:Room .
    ?room modality:modality ?temp_mod .
    ?temp_mod a modality:Temperature .

    # Temperature sensor
    ?temp_sensor modality:provides ?temp_mod .
    ?temp_sensor a point:TemperatureSensor .

    # Temperature actuator
    ?actuator modality:controls ?temp_mod .
    ?actuator a point:TemperatureActuator .

    # Thermostat
    ?thermostat a point:Thermostat .
}
"



# STEP 5: Test with verification query
SPARQL.execute_query(model, q_verify_actuators)
|> result_as_table.()
```

```
# Step 6: Main Thermostat Query - Extract all necessary IDs
q_thermostat_ids = "
PREFIX location: <#{base}location#>
PREFIX modality: <#{base}modality#>
PREFIX point: <#{base}point#>
PREFIX data: <#{base}data#>

SELECT ?room ?temp_sensor_id ?setpoint_id ?actuator
WHERE {
    ?room a location:Room .
```

```
        # Temperature sensing
        ?room modality:modality ?temp_mod .
        ?temp_mod a modality:Temperature .
        ?temp_sensor modality:provides ?temp_mod .
        ?temp_sensor data:data ?temp_data .
        ?temp_data data:live ?temp_sensor_id .

        # Temperature setpoint
        ?room modality:modality ?setpoint_mod .
        ?setpoint_mod a modality:TemperatureSetpoint .
        ?thermostat modality:provides ?setpoint_mod .
        ?thermostat data:data ?setpoint_data .
        ?setpoint_data data:live ?setpoint_id .

        # Temperature actuation
        ?actuator modality:controls ?temp_mod .
        ?actuator a point:TemperatureActuator .
    }
    "


    # STEP 6: Run the main thermostat query
    SPARQL.execute_query(model, q_thermostat_ids)
    |> result_as_table.()
```

Let's address **Quest 6**, which delves into schema validation for your RDF model. This quest highlights the importance of defining rules to maintain data integrity and prevent inconsistencies.

---

1. What would a model checker look like?

In the context of RDF, a **model checker** would be a system or a set of processes that evaluates your RDF graph (your model) against a defined set of schema rules. Its primary function would be to identify any triples or patterns in the graph that violate these rules, flagging them as errors or inconsistencies.

Conceptually, an RDF model checker would:

- **Parse the RDF Graph**: It would first load and understand the structure of your RDF data.
- **Load the Schema Definitions**: It would then load the schema rules, which could be expressed using RDFS, OWL, or SHACL.
- **Iterate and Validate**: It would iterate through the triples in your data graph and compare them against the constraints defined in your schema.
    - For example, if a rule states that a `location:Room` must have a `location:area`, the checker would find all instances of `location:Room` and verify the presence of the `location:area` property.
    - If a rule states that `location:area` must have a literal value of type integer, it would check the datatype of the `location:area` property's object.
- **Report Violations**: It would output a report detailing any violations found, indicating which triples or resources fail to conform to the schema, and ideally, why.

---

2. Which if any rules are missing?

Based on the provided Schema module and the relationships defined, while you have a good start with `rdfs:subClassOf` for class hierarchy, several types of rules are commonly missing for robust schema validation in RDF models, especially for IoT applications:

- **Cardinality Constraints**: Rules that specify how many times a property can (or must) appear for a given resource.
  - **Example**: A `location:Room` must have exactly one `location:area`.
  - **Example**: A `point:TemperatureSensor` must provide exactly one `modality:Temperature`.
- **Property Domain and Range Restrictions**: While `rdfs:domain` and `rdfs:range` are basic RDFS constructs, explicitly defining them in your schema ensures that properties are used correctly with subjects and objects of specific types.
  - **Example**: `location:contains` has a domain of `location:Building` or `location:Floor` and a range of `location:Floor` or `location:Room`.
  - **Example**: `unit:unit` has a range of `unit:Unit` (or its subclasses).
- **Datatype Restrictions**: Ensuring that literal values conform to expected datatypes (e.g., `xsd:integer`, `xsd:string`).
  - **Example**: `location:area` should be an integer.
  - **Example**: `data:live` and `data:hist` should be strings.
- **Property Shape Constraints**: More complex constraints on the shape of data, not just simple domain/range or `subClassOf`.
  - **Example**: A `point:TemperatureSensor` that provides a `modality:Temperature` must also have a `unit:unit` property.
- **Disjoint Classes**: Stating that certain classes cannot overlap (e.g., an instance cannot be both a `location:Room` and a `location:Building` simultaneously).
- **Mandatory Properties**: Ensuring that certain properties are always present for instances of a specific class.
  - **Example**: Every `location:Room` must have a `location:name`.

---

3. How would you describe them?

Let's describe some of the missing rules more formally:

- **Mandatory Property: Room Name**: Every resource typed as `location:Room` must have exactly one `location:name` property.
  - **Rationale**: Ensures all rooms are identifiable.
- **Datatype Constraint: Area**: The `location:area` property must have a literal value that is an integer.
  - **Rationale**: Ensures numerical consistency for area measurements.
- **Cardinality Constraint: Sensor Modality**: A `point:TemperatureSensor` must provide exactly one `modality:Temperature`.
  - **Rationale**: A sensor should be clearly linked to the single modality it provides.
- **Domain and Range for `location:contains`**:

    - **Domain**: The subject of a `location:contains` triple must be an instance of `location:Building` or `location:Floor`.
    - **Range**: The object of a `location:contains` triple must be an instance of `location:Floor` or `location:Room`.
    - **Rationale**: Prevents illogical relationships like a room containing a building.
- **Co-occurrence Constraint: Sensor Unit**: If a `point:Sensor` provides a `modality:Temperature`, then it must also have a `unit:unit` property.
    - **Rationale**: Temperature readings are meaningless without a unit.

---

4. How would you implement them?

Implementing these rules in an RDF environment involves using schema languages and potentially programmatic checks.

Using RDFS and OWL (Ontology Web Language):

- `rdfs:domain` and `rdfs:range`: You can add these directly to your schema definitions for properties. For example, in your schema list:

  ```
  Schema.Location.contains
  |> RDFS.domain(Schema.Location.Building) # or Schema.Location.Floor
  |> RDFS.range(Schema.Location.Floor) # or Schema.Location.Room
  ```

  (Note: Elixir's RDF library might require specific syntax for multiple domains/ranges or more complex property axioms.)

- `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`: OWL provides properties for expressing cardinality. You would typically define these on classes or properties.

    - **Example for "a Room must have exactly one area"**: This would require defining a class axiom on `Schema.Location.Room`.

- `owl:hasValue`: For ensuring a property has a specific value.

- `owl:disjointWith`: To declare classes as mutually exclusive.

While RDFS and OWL can express these rules, their enforcement is usually done by:

- **Reasoners**: OWL reasoners (e.g., Pellet, HermiT) can infer new facts and check for inconsistencies based on OWL axioms.
- **Programmatic Checks**: You can write Elixir code to query the model and check for adherence to these rules. For instance, to check mandatory properties, you'd write a SPARQL query to find all `location:Room` instances that don't have a `location:name`.

Using SHACL (Shapes Constraint Language):

SHACL is a W3C recommendation specifically designed for validating RDF graphs against a set of structural constraints (shapes). It's more expressive for validation than RDFS/OWL for many common data quality checks.

You would define SHACL "shapes" in your RDF graph. Each shape specifies constraints on a set of nodes. A SHACL engine then checks your data graph against these shapes.

Example SHACL-like rule for Mandatory Room Name (conceptual):

```
:RoomShape a sh:NodeShape ;
  sh:targetClass location:Room ;
  sh:property [
    sh:path location:name ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:message "Every room must have exactly one name." ;
  ] .
```

To implement this, you would need a SHACL engine (or build a simplified one) that can process these shapes and validate your model. In Elixir, this would involve:

- Defining your SHACL shapes as RDF triples within your model or a separate graph.
- Writing Elixir code that iterates through your data graph and applies the logic defined by your SHACL shapes using SPARQL queries or graph traversal.

Programmatic Implementation (Elixir):

Since your environment includes SPARQL and RDF modules, you can write custom Elixir functions to act as a basic model checker.

Example: Checking for mandatory `location:name` for `location:Room` instances:

```
# Inside your Livebook, after your model is defined
base = Schema.base()

# Query to find rooms missing a name
q_missing_name = """
PREFIX location: <#{base}location#>
SELECT ?room
WHERE {
  ?room a location:Room .
  FILTER NOT EXISTS { ?room location:name ?name . }
}
"""

missing_rooms_result = SPARQL.execute_query(m, q_missing_name)
missing_rooms = missing_rooms_result.results

if Enum.empty?(missing_rooms) do
```

```
  IO.puts "All rooms have a name."
else
  IO.puts "WARNING: The following rooms are missing a name:"
  Enum.each(missing_rooms, fn %{room: room_iri} ->
    IO.puts "- #{RDF.IRI.to_string(room_iri)}"
  end)
end
```

Example: Checking for datatype of `location:area`:

```
q_invalid_area_datatype = """
PREFIX location: <#{base}location#>
SELECT ?room ?areaValue
WHERE {
  ?room a location:Room .
  ?room location:area ?areaValue .
  FILTER (!isLiteral(?areaValue) || datatype(?areaValue) != xsd:integer)
}
"""
# Note: You'd need to add xsd prefix if not already defined globally
# PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

invalid_area_result = SPARQL.execute_query(m, q_invalid_area_datatype)
invalid_areas = invalid_area_result.results

if Enum.empty?(invalid_areas) do
  IO.puts "All room areas have valid integer datatypes."
else
  IO.puts "WARNING: The following room areas have invalid datatypes:"
  Enum.each(invalid_areas, fn %{room: room_iri, areaValue: value} ->
    IO.puts "- Room #{RDF.IRI.to_string(room_iri)} has area '#{value}' with
invalid datatype."
  end)
end
```

By combining these methods, you can build a comprehensive model checker. For this exercise, demonstrating a few programmatic checks using SPARQL would be a good starting point to illustrate the concept.