Semantic Tic Tac Toe

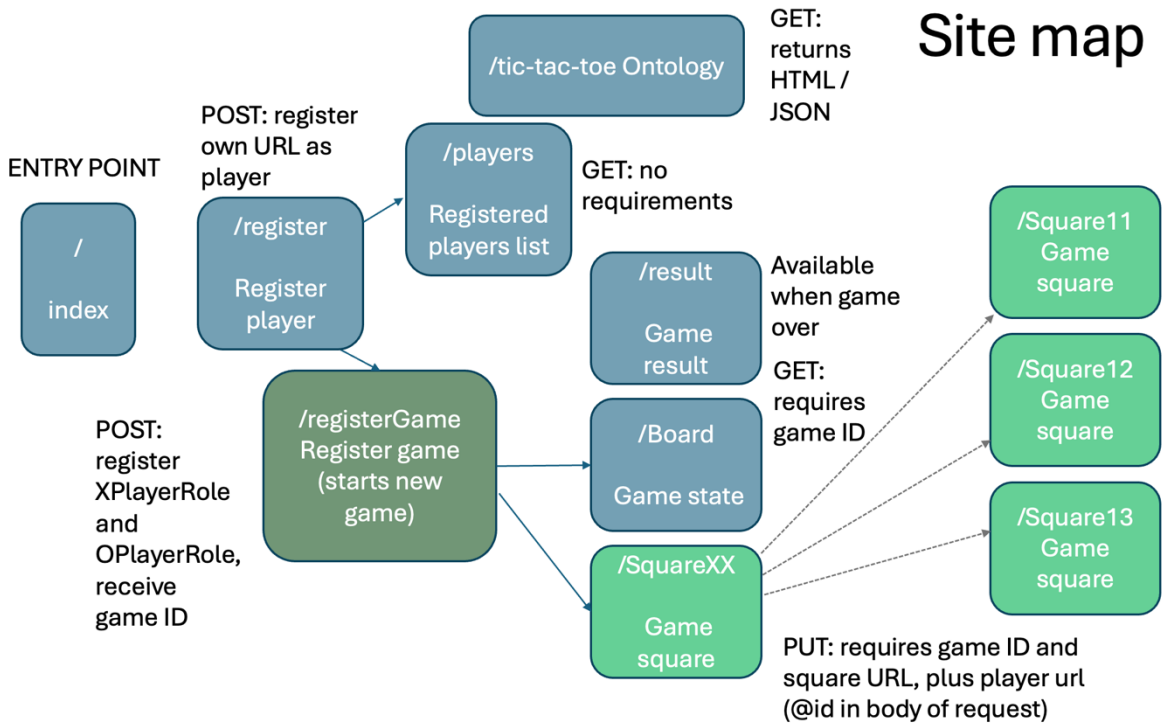
Requirements: **Java 11**, Docker or Python
Code: https://gitlab.com/mams-ucd/examples/semantic-environments/tic-tac-toe

Complete all of Part1: E
Complete all of Part2: D
Complete all of Part3: C
Complete all of Part4: B
Complete all of Part5: A

The Semantic Web is a concept, the concept of machine-understandable data and information that software agents can manipulate. Key technologies, such as ontologies aim to provide interoperable standards for knowledge exchange; mechanisms for agents to interact with and understand different, distributed Web resources.

This lab provides a different challenge: from modelling your agent's experience of the environment as beliefs and performing logical reasoning, your agents now need to interact with a different type of knowledge representation, merging the semantic with the symbolic...

The Semantic Tic Tac Toe environment is an API which can be deployed on the Web, or (breathe a sigh of relief), locally on your machine. It represents knowledge about the game board as a series of triples. The entry point is the index page. If deployed locally, this will be http://localhost:8083/



Sending a GET request to the entry point returns the available actions, in JSON-LD (JSON Linked Data).

The response will look something like this:

```
{
 "@id": "http://localhost:8083/",
 "@type": "ttt:Game",
 "@context": {
  ...
 },
"links": [
  {
   "href": "http://localhost:8083/players",
   "htv:methodName": "GET"
  }
 ]
 "forms": [
  {
   "href": "http://localhost:8083/register",
   "contentType": "application/json",
   "htv:methodName": "POST",
   "wot:op": "writeproperty",
   "properties": [
    {
     "name": "@id",
     "readOnly": false,
     "required": true
    }
   ]
  }
 ]
}
```

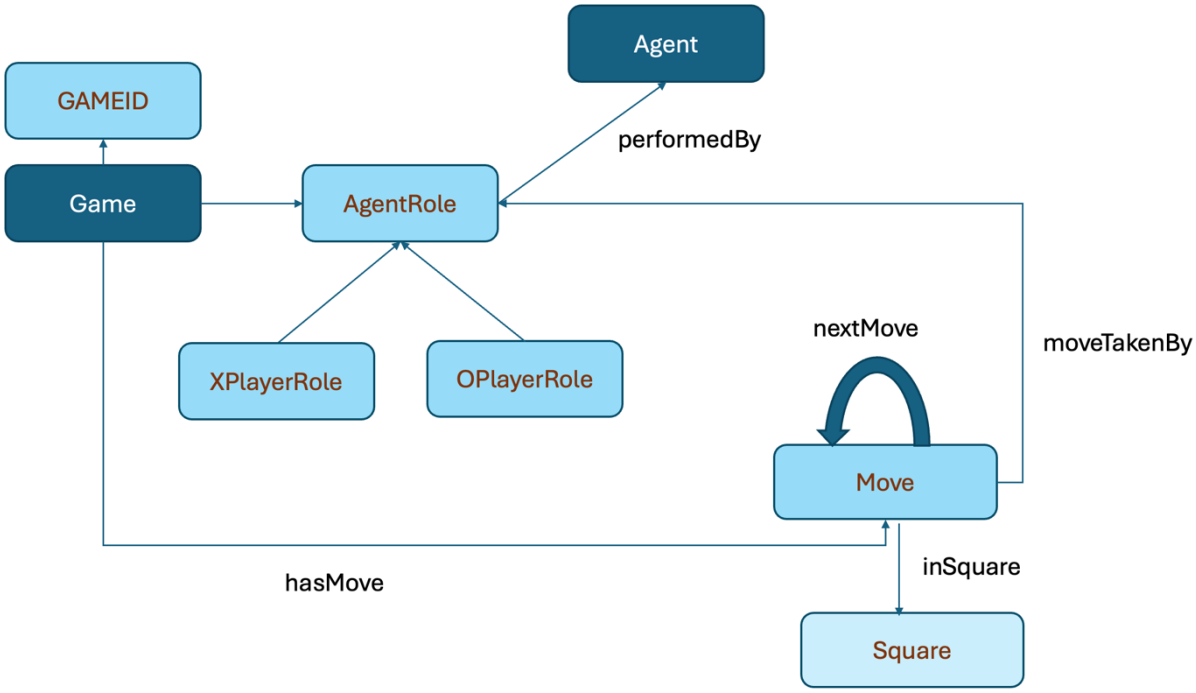
This indicates that there are two actions that can be taken. One is to navigate to the /players endpoint, using a GET request. The other is to submit a form via POST request to the /register endpoint, with the property @id. In the agent code later, these will be referred to as **link actions** and **form actions**.

The principle is that from a single entry point, a client, or agent, can navigate the API in the same way that you might navigate a website.

The API includes an *ontology* for the game play, which acts like a blueprint or vocabulary for the game: it includes concepts like “Square” for the places on the Tic-Tac-Toe board, “Move” for a game move, etc. When new agents register to play, a new RDF graph of the instantiated ontology classes is created: for each game, the set of possible squares is created. When moves are made, an instance of the Move class is created along with relationships which link the Move to a Square, *inSquare*, and with a player role, *moveTakenBy*.

The diagram below gives you a rough idea of the concepts in the ontology and how they are connected to each other: a Game has a GameID, and provides two AgentRoles:

XPlayerRole and OPlayerRole. The Game has a series of moves, which are taken by the agent role.



Each of the squares in the Tic Tac Toe game is represented by a number: 11 for the top left, through to 33 for the bottom right square. When a game is in play, these Square classes are instantiated. E.g. *Square11*, *Square12*... acts like a class, and when a game starts (say with some game ID, 1234), the Square instances *Square11/1234*, *Square12/1234* etc are created.

Square11	Square12	Square13
Square21	Square22	Square23
Square31	Square32	Square33

How does this agent access this?

The agent can access this knowledge using a Knowledge Store module. This converts the JSON-LD response into a set of triples. For example, for the JSON-LD response below:

```
{
  "@id": "http://localhost:8083/",
  "@type": "ttt:Game",
  "@context": {
    "@vocab": "https://www.w3.org/2019/wot/hypermedia#",
    "ttt": "http://localhost:8083/tic-tac-toe#",
    "htv": "http://www.w3.org/2011/http#",
    "wot": "https://w3c.github.io/wot-thing-description/#",
    "sch": "https://schema.org/",
    "links": {
      "@id": "Link"
    },
    "forms": {
      "@id": "Form"
    },
    "href": {
      "@id": "hasTarget"
    },
    "rel": {
      "@id": "hasRelationType",
      "@type": "@vocab"
    }
  },
  "links": [
    {
      "href": "http://localhost:8083/players",
      "htv:methodName": "GET"
    }
  ]
  "forms": [
    {
      "href": "http://localhost:8083/register",
      "contentType": "application/json",
      "htv:methodName": "POST",
      "wot:op": "writeproperty",
      "properties": [
        {
          "name": "@id",
          "readOnly": false,
          "required": true
        }
      ]
    }
  ]
}
```

The KnowledgeStore creates triples that look like this:

6d094fa6a9d02c6370e39458caf40241 https://www.w3.org/2019/wot/hypermedia#required "true^^http://www.w3.org/2001/XMLSchema#boolean" .
6d094fa6a9d02c6370e39458caf40241 https://www.w3.org/2019/wot/hypermedia#readOnly "false^^http://www.w3.org/2001/XMLSchema#boolean"
6d094fa6a9d02c6370e39458caf40241 https://www.w3.org/2019/wot/hypermedia#name "@id" .
217a6260a4323fe8af1d2253a728dae4 https://www.w3.org/2019/wot/hypermedia#properties 6d094fa6a9d02c6370e39458caf40241 .
217a6260a4323fe8af1d2253a728dae4 https://www.w3.org/2019/wot/hypermedia#hasTarget "http://localhost:8083/register" .
217a6260a4323fe8af1d2253a728dae4 https://www.w3.org/2019/wot/hypermedia#contentType "application/json" .
217a6260a4323fe8af1d2253a728dae4 https://w3c.github.io/wot-thing-description/#op "writeproperty" .
217a6260a4323fe8af1d2253a728dae4 http://www.w3.org/2011/http#methodName "POST" .
http://localhost:8083/ https://www.w3.org/2019/wot/hypermedia#Link d46ebc63c772d624faf40522b56c0da9 .
http://localhost:8083/ https://www.w3.org/2019/wot/hypermedia#Form 217a6260a4323fe8af1d2253a728dae4 .
http://localhost:8083/ http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://localhost:8083/tic-tac-toe#Game .
d46ebc63c772d624faf40522b56c0da9 https://www.w3.org/2019/wot/hypermedia#hasTarget "http://localhost:8083/players" .
d46ebc63c772d624faf40522b56c0da9 http://www.w3.org/2011/http#methodName "GET" .

The API endpoint, has a link denoted by an ID `d46ebc63c772d624faf40522b56c0da9`
The last two lines show that connected to this ID (an inner node ID[1]), there are two fields:
`hypermedia#hasTarget /players` and `http#methodName "GET"`

These correspond to the triples

Subject	Predicate	Object
innerNodeID `d46ebc63c772d624faf40522b56c0da9`	`hypermedia#hasTarget`	http://localhost:8083/players
innerNodeID `d46ebc63c772d624faf40522b56c0da9`	`http#methodName`	"GET"

The RDFSchema modules allow you to access what is in the Knowledge Store by reference to these ontologies. For example, if the schemas are imported like so:

```
module RDFSchema("http://www.w3.org/2011/http#") http;
module RDFSchema("https://www.w3.org/2019/wot/hypermedia#") hypermedia;
```

The triples can be accessed like this in ASTRA:

```
string url = http://localhost:8083;

foreach(hypermedia.Link(url, string link_inner_node_id)){
   if (hypermedia.hasTarget(link_inner_node_id, string
target)) {
   …
   }
}
```

With "http://localhost:8083/players" returned as the value of the inline variable target.

You either have to know the subject and predicate, or predicate and object to access this. For example,

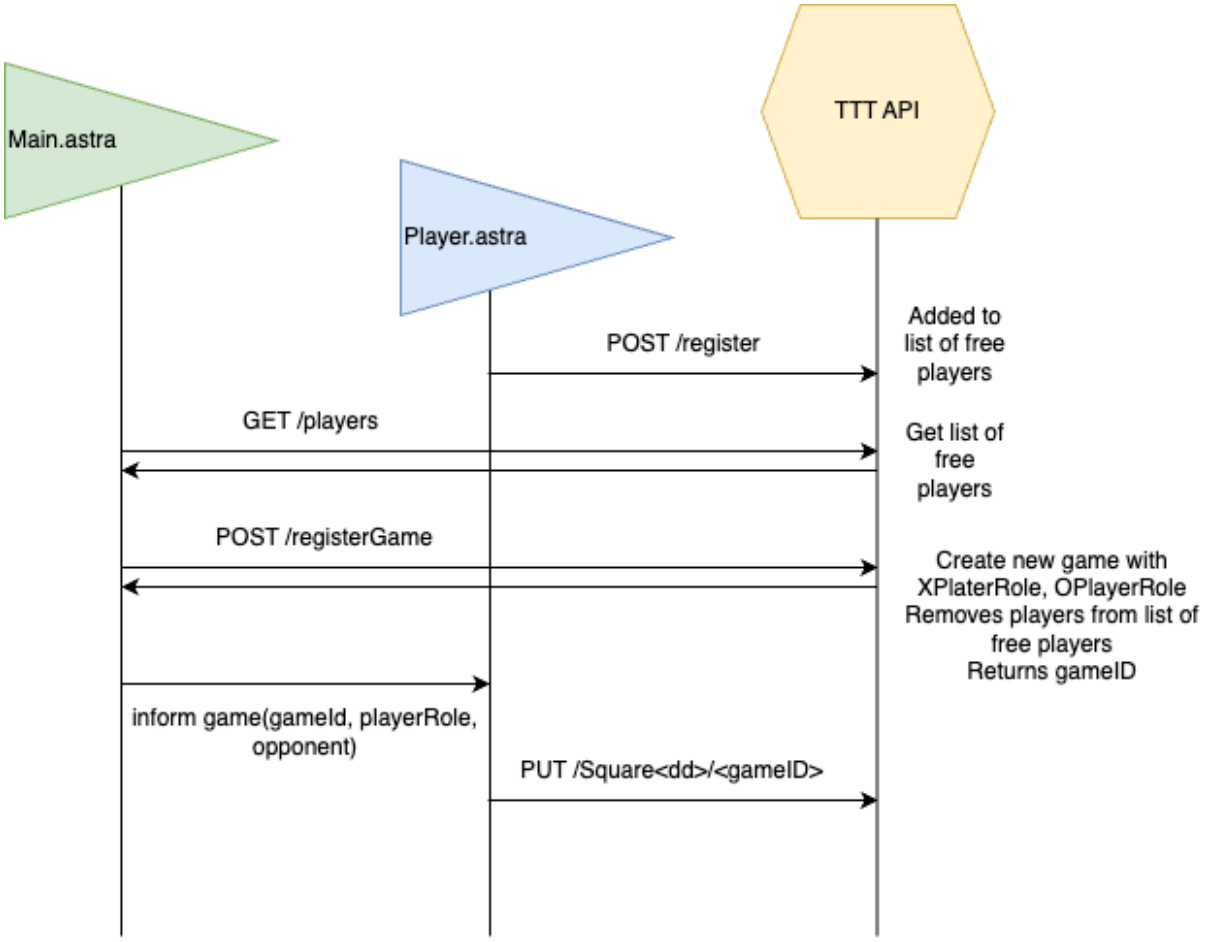
[1] The underlying Apache Jena module (which the Knowledge Store uses) generates inner node IDs for these structures.

```
string target = http://localhost:8083/players;
if (hypermedia.hasTarget(string innerNodeId, target)) {
…
}
```

Would also work.

Game play

You are going to modify sample code so that the Main agent manages Player agents, and their interactions. The Player is going to register itself as a player with the ttt-api. It gets added to the list of free players. The Main agent can query this list, then register a game between two players. Once the players are informed of the relevant details (game ID, their role, the URL of their opponent) they can play away…



Part 1: Download the sample code, run the API and have Player.astra register with the API

a) Download the code from https://gitlab.com/astra-language/examples/semantic-environments/tic-tac-toe

b) Look at the ttt-api. Carefully read and download the requirements as per the README.md file: you either need Python or Docker to be installed.
Look through the README.md to familiarise yourself with the game flow and the form requests and responses.

Either use Docker (see README, you can build the image yourself or download it from the container registry https://gitlab.com/mams-ucd/examples/semantic-environments/tic-tac-toe/container_registry)

Or (with Python installed locally) run the API using the command:
`flask run --host=localhost --port=8083`

Test different endpoints by either downloading a REST API extension for your web browser, or using CURL (https://curl.se/). When your system is set up

c) Download sample-agents and go into Part1. Run `mvn`, you should get this output that looks like this:

```
[INFO] --- astra-maven-plugin:1.4.4:deploy (default-cli) @ ttt-agents ---
[Scheduler] Processors: 8
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/Users/katharinefitzpatrick/.m2/repository/org/apache/jena/jena-jdbc-driver-bundle/4.10.0/jena-jdbc-driver-bundle-4.10.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/Users/katharinefitzpatrick/.m2/repository/org/apache/logging/log4j/log4j-slf4j-impl/2.21.0/log4j-slf4j-impl-2.21.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
WARNING: Runtime environment or build system does not support multi-release JARs. This will impact location-based features.
Web Server started at 9000
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by mams.web.HttpClient (file:/Users/katharinefitzpatrick/.m2/repository/mams-ucd/mams-cartago-core/1.0.11/mams-cartago-core-1.0.11.jar) to field java.lang.reflect.Field.modifiers
WARNING: Please consider reporting this to the maintainers of mams.web.HttpClient
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
[main]creating: main-base
Creating base artifact: main
Exposing Agent @ Base Uri: http://127.0.0.1:9000/main
[AgentZero]creating: AgentZero-base
Creating base artifact: AgentZero
Exposing Agent @ Base Uri: http://127.0.0.1:9000/AgentZero
```

Look at Player.astra and the main goal. Here the agent is set up as a MAMS agent. The IntegratedInbox allows it to use the inbuilt messaging functionality to communicate with other MAMS agents.
The agent adds stores information about itself in the beliefs `agent_attributes`, then it adopts the subgoal `!get_index_page()`.

You need to complete this subgoal by getting the agent to perform a GET request on the game API index. Go to the subgoal (search `TODO Part 1`) and add the line:

```
knowledgeStoreGame.getKnowledge(game_url, "JSON-LD");
```

This performs an action in the Knowledge Store module. It performs a GET request on the URL, and expects the response to be in the form of JSON-LD. It then processes the response, and converts it into triples based on the specified ontologies (in the @context part of the response).

Run the code, and you should see this additional logging on the console:

```
[AgentZero]Processed information from http://localhost:8083/
```

This logging is output on line 75 when the Knowledge Store module fires a "read" event. Below this logging, add the line:

```
knowledgeStoreGame.displayModel();
```

And run the agent again. This should display the triples which have been processed by the Knowledge Store.

d) In Shared.astra, look at the subgoals `!link_actions(string url)` and `!form_actions(string url)`. Using the ontologies loaded in (the RDFSchema module declarations near the top of the class), the agent can interact with the triples processed by the Knowledge Store a bit like beliefs. In the `!form_actions(…)` goal it iterates through each of the Form items processed by the Knowledge Store and extracts the target (the URL to POST/PUT the form response to), the method (i.e. POST or PUT), and any required properties (i.e. the data that needs to be included in the body of the request for it to be successful).

Adhering to the principles of HATEOAS means that each endpoint of the ttt-api should contain all of the information the agent needs to navigate the API in that moment, just as when you navigate a webpage (like shopping on Amazon), each journey through the site (ideally) has a logical flow. So in comparison to the earlier Tic Tac Toe lab, the agent doesn't need to infer what a "free" move is, or keep track of the state of the game board (unless in Parts 4 and 5, you find this useful when devising a strategy).

However because of this in later parts of this assignment, in between calls to the API (whether GET, PUT or POST), and processing by the Knowledge Store, you will need to clear down form and link beliefs and clear the Knowledge Store by calling the subgoal `!clear_knowledge()`. This has been written for you. If

you do not, the agent may have redundant information about the possible actions it can take.

e) Now, when the agent processes information and a `form_actions` belief is added with the target (URL to POST/PUT to) to "register", we want the agent to perform this action with the API. Look for the deliberation rule in the Player.astra code for the addition of a form belief (around line 88, or search "TODO Part 1"). The context condition means that this will only fire if there is no active game (i.e. no game belief), the url ends with the string "register", and it also accesses a belief `http_fail_count`.

   You need to perform the form action by performing a PUT or POST request to the url, with certain information in the body of the request.

   For the body of the request, you will need to create a JsonNode object using:

   ```
   JsonNode bodyJson = builder.createObject();
   ```

   Now the `agent_attributes` belief should come in useful. In the main goal, we added certain information to it. The form action to register requires an "@id" field, which we have in the `agent_attributes` belief.
   Extract this information and add it to the JSON object like so:

   ```
   builder.addProperty(bodyJson, "@id", "www.myurl.com");
   ```

   Replacing www.myurl.com with the relevant variable.

   Convert the body to a string with this code:

   ```
   string bodyStr = builder.toJsonString(bodyJson);
   ```

   You will then need to check the HTTP method the form expects (i.e. PUT or POST) and perform the request either with this code:

   ```
   MAMSAgent::!post(form_url, bodyStr, HttpResponse response);
   ```

   Or this:

   ```
   MAMSAgent::!put(form_url, bodyStr, HttpResponse response);
   ```

   The response is available to you as an inline variable. You can print this to the console to see what it looks like:

   ```
   console.println(httpUtils.bodyAsString(response));
   ```

   After this, invoke the clear knowledge subgoal:

   ```
   !clear_knowledge();
   ```

Note that when you start to run your agents, you may need to restart the API in between testing your code to register agents (otherwise they will already be registered!).

f) As for try and catch statements in Java, in ASTRA you can surround code with `try {} recover {}`

For example:

```
try {
   !someSubgoal();
recover {
   console.println("Something bad happened…");
}
```

Surround the code you wrote to register with `try {} recover {}` statements in order to provide some sort of fault tolerance when registering with the API. The API will return an error if the agent is already registered. In the recover statement, update the belief `http_fail_count`, and re-add the form belief which triggered the process, as a way to recursively perform the action to register.

Check that your code compiles and runs, and manually check the url: http://localhost:8083/players to see if you agent has registered (using curl[2], or a browser extension that lets you view RDF[3].

[2] https://curl.se/

[3] Such as https://addons.mozilla.org/en-US/firefox/addon/rdf-browser/ for Firefox, or use Safari if you want a wall of text

Part 2: Multi-agent management: have the Main agent query the API and assign roles

a) Copy Part1 and rename it Part2. In the main goal of the agent Main.astra, create an additional player called AgentOne (in exactly the same manner as for AgentZero). Run this, and check that they both register by checking http://localhost:8083/players.
At this point, you can comment out the logging added in Part1 to display the knowledge store model, and log the response after registering the agent.

b) In the Main agent, the subgoal `!game()` is ready for you to fill in: you want the Main agent to query the /players endpoint (i.e. now you want it to check http://localhost:8083/players for you). Refer back to the Player agent querying the index page to see how to go about this.

c) Below the game subgoal, there is a goal to handle the Knowledge Store read event for the /players endpoint. Add the following code to it:

```
//Get the possible players
foreach(ttt.Agent(url, string agent_url)){
     console.println("Available player: " + agent_url);
}
```

Modify this code so that the agent decides which agent is going to play the XPlayerRole (in this game, X always goes first) and which agent is going to play the OPlayerRole, by adding the belief `player(XPlayerRole, agent_url)` for one agent, and `player(OPlayerRole, agent_url)` for another agent. How you do this is up to you.

You can also add the line:
`knowledgeStore.displayModel();`

to see the information the API is returning. If you add this line, you'll see that the form, /registerGame, is now available to the agent.

The Main agent also extends the Shared agent, so it can adopt the subgoal
`!form_actions(url);`

to process available form actions returned by the /players endpoint. Invoke this subgoal below the code you added the Knowledge Store read event.

d) As for Part 1, the `form_actions` belief should tell the Main agent exactly what information to put in the request. You need to write code in the to handle the `form_actions` belief being added for the /registerGame endpoint, this is in the next 'TODO Part 2' in rule `+form_actions(….)`.

You want the Main agent to register the agents. You can the `player` beliefs. You need to check whether the form requires a POST or PUT request, and build the

JSON body for the request. You can loop through the list of required actions to build the body, e.g.

```
forall(string item : required_actions) {
      if (strings.equal(item, "ttt:OPlayerRole")
            & player(OPlayerRole, string playerURL)) {

           builder.addProperty(bodyJson, item, playerURL);
      }
      //Now you need to do the XPlayerRole…
}
```

Once you have the form response, clear the knowledge store, and then process the response with the line:

```
knowledgeStore.getKnowledgeFromString(httpUtils.bodyAsString(response
), form_url);
```

This can be used when we already have the response from the API endpoint, i.e. the request has already been made, and we only require the Knowledge Store to process it, not to make a GET request.

e) Now the Main agent should have registered two agents to play a game, and the response from this includes the crucial game ID which agents need to play the game. You'll need to extract this in the Knowledge Store read event with the condition that the url ends with "registerGame" (the next 'TODO Part 2'). In this rule, add the following code:

```
if (ttt.hasID(url, string gameID)) {
    console.println("Game ID is " + gameID);
}
```

Now you have all of the information you need to send the Player agents a message that there is a new game: you can do this with the internal MAMS messaging, which will send the requests over HTTP to the agent inbox endpoints. You need to write some code here to get the information you need (i.e. look at the player beliefs…) then send a message that looks something like this to each agent:

```
send(inform, <agent_url>, game(<gameID>, <agent_role>,
<opponent_url>));
```

f) Finally, in the Player.astra code, amend the rule that handles the incoming message (look for 'TODO Part 2'), and have it log out the game details to the console with the line:

```
   console.println("I've been given the game " + id + " with role " +
role + " and opponent " + opponent);
```

and add the `game(…)` belief to the agent with that information. If the agent has the X player role, also add the belief that it is it's turn.

Part 3: Two agents play a game (without any particular strategy)

a) Copy Part2, and rename it Part3. Find the (empty) deliberation rule in Player.astra for the addition of a new turn belief (look for TODO Part 3). Here you need to add the code[4]:

```
knowledgeStoreGame.getKnowledge(url + "Board/" + id,"JSON-LD");
```

This allows the agent to get the state of the board. Look at this endpoint and the information it gives you (go back to the ttt-api README if you need).

That's all you need in that rule, but there are two more KnowledgeStore rules you need to code in order to process the endpoint (look for TODO Part 3). Start with the second one, with the condition that there is a game, but not that the URL contains the word "result".

b) Here, you need to add code to do the following:
   - Process form and link actions
   - Sleep for 1 second (give it time to add the beliefs)
   - Check to see if there is a link to the /result endpoint. if so, use the Knowledge Store to perform a GET request on the endpoint. This will then trigger the Knowledge Store read event above, which you haven't coded yet
   - If not, pick a form action and perform the request as indicated by the `form_actions` belief *(there is no requirement in this Part for any strategy or sophistication when picking an action)*.
   - Handle it being the end of this turn (what do you need to do? Check the deliberation rule for a turn belief being removed)

c) In the other Knowledge Store read event, you need to access the result (hint: look at how you got the game ID), print the winner to the console, and handle it being the end of the game (what do you need to do? Which beliefs need updating?).

An important note: when you get the player role from the API, this is going to be the fully qualified ontology term (e.g. http://localhost:8083/tic-tac-toe#XPlayerRole) whereas you might have used a shortened version in your beliefs (e.g. when adding the player belief to the Main agent, the string will be "XPlayerRole").

d) Finally, you need to handle the incoming message that informs the agent that it is it's turn.

[4] Unfortunately this is the last bit of code to help you. But you've got this.

e) Two agents should now play a game to completion. When the game is complete, you can see the ttl[5] file of the game result is written to the results folder in the ttt-api.

f) Once the game is over, inform the Main agent of the result (there are no placeholder rules or beliefs to help you here).

[5] https://www.w3.org/TR/turtle/

Part 4: Strategy

Copy Part3, and rename it Part4. Create two new agents, Opponent.astra and Better.astra.

Better.astra should always beat Opponent.astra no matter which agent starts first. Both should have some sort of strategy (look back at your solution to the previous Tic Tac Toe lab).

They should be able to play multiple consecutive games, i.e. you should demonstrate them playing each other, with different agents taking the XPlayerRole, without having to comment or uncomment code, and re-run the program.

For a B+ your solution should as much as possible adhere to practical reasoning[6], the console logging should be concise, the code well commented and well-presented.

Part 5: Plan to Fail and Tournaments

Copy Part4, and rename it Part5.

Have the Main agent set up a tournament between three agents, where each agent plays the other two agents once. The Main agent should keep track of the score and print the results of the tournament to the console.

Agents should be able to handle other agents not being available and the API not being available, but in no more complex ways than incrementing a belief and shutting down when a threshold is reached.

For an A+ make this work for any number of agents, and the console logging interesting.

[6] Hint: there may be long rules in Player.astra from Part 3 that you could refactor