MA LAB5

JSON-LD (JSON for Linked Data) is a format used for encoding Linked Data using JSON. It provides a way to structure data in a machine-readable format that connects the data with a clear context and semantics, making it easier for applications to understand and use.

### What JSON-LD Contains:

1. @id: A unique identifier for the resource (e.g., the URL of an API endpoint or a resource in a semantic environment).
2. @type: Defines the type or category of the resource (e.g., ttt:Game for a Tic Tac Toe game).
3. @context: Describes the vocabulary and ontologies used in the data, providing semantic meaning for the terms. It often includes URLs linking to external schemas or definitions.
4. **Links actions and Form actions**:
   1. links: Specifies related resources and the methods to access them (e.g., a GET method for retrieving a list of players).
   2. forms: Provides instructions for actions like submitting data (e.g., a POST method to register a player), including details about required properties and content types.

### Example

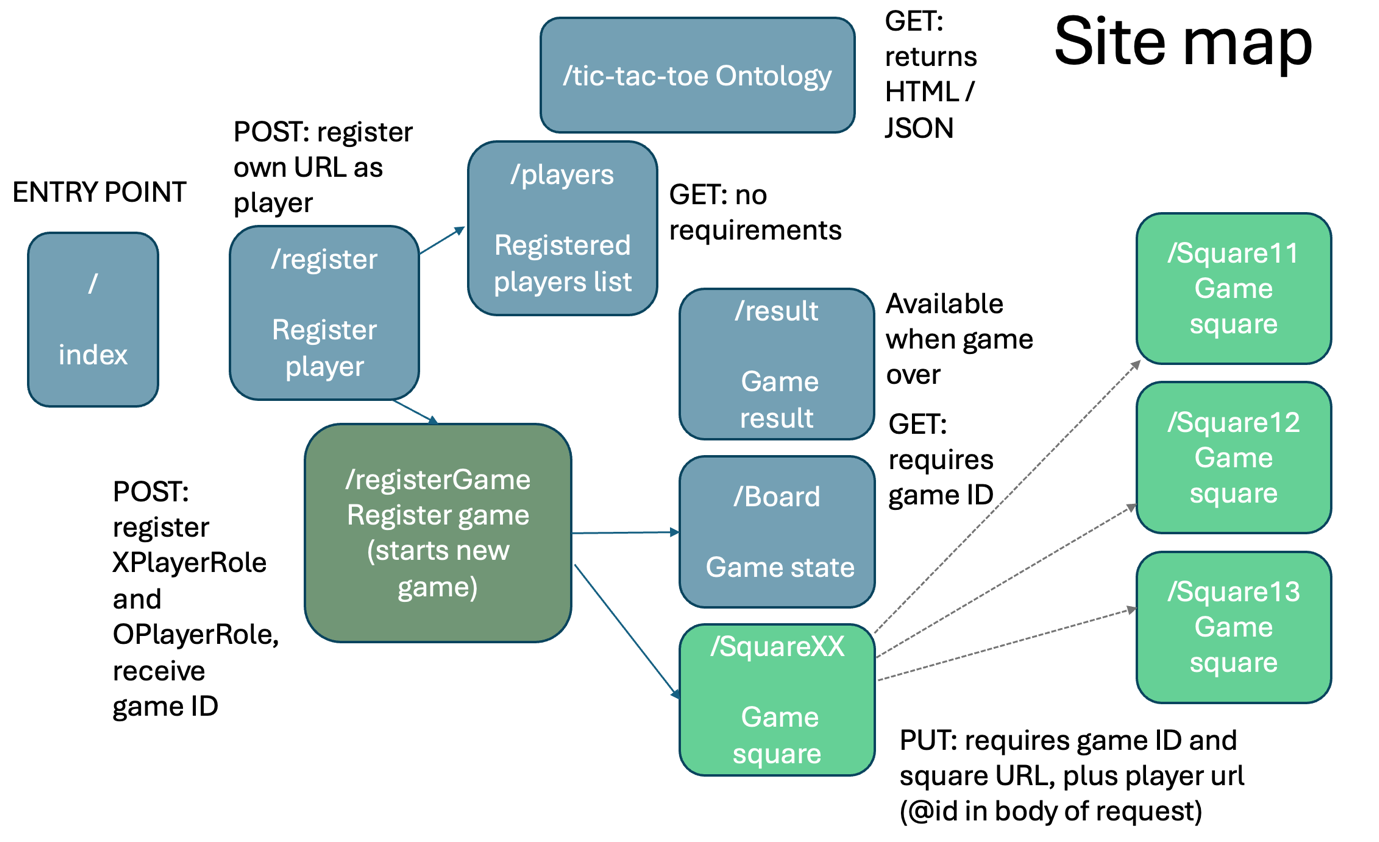
The JSON-LD response includes:

* An entry point (e.g., http://localhost:8083/).
* Information about available actions:
  + A GET request to /players to retrieve player information.
  + A POST request to /register to register a new player with specified properties.

Example:

JSON-LD return data from GET request to entry point





This diagram provides a **site map** for the API used in the Semantic Tic Tac Toe environment as described in the attached PDF. It outlines the main endpoints and actions that agents (players) can perform while interacting with the game.

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.

### Key Components:

**Entry Point (**/index**)**:

* 1. This is the starting point for interacting with the API.
  2. When accessed (e.g., via a GET request), it returns the available actions and links in a JSON-LD format.

**Player Registration (**/register**)**:

* 1. Endpoint for registering a player.
  2. Requires a POST request where the agent provides its unique URL (using @id in the request body).
  3. Registers the agent as a potential player in the game.

**Registered Players List (**/players**)**:

* 1. Provides a list of players currently registered.
  2. Accessed via a GET request without additional parameters.
  3. Useful for identifying available players for a game.

**Game Registration (**/registerGame**)**:

* 1. Starts a new game and assigns roles (XPlayerRole and OPlayerRole) to two players.
  2. Requires a POST request with the player URLs (provided as @id properties in the request body).
  3. Returns a **Game ID**, which is essential for tracking and interacting with the game.

**Game State (**/Board**)**:

* 1. Provides the current state of the game board.
  2. Accessed via a GET request with the Game ID.
  3. Useful for checking the game progress.

**Game Result (**/result**)**:

* 1. Provides the final result of the game.
  2. This endpoint becomes available only after the game concludes.
  3. Accessed via a GET request with the Game ID.

**Game Squares (**/SquareXX**)**:

* 1. Represents individual squares on the Tic Tac Toe board (e.g., Square11, Square12).
  2. Requires a PUT request to make a move, with the Game ID and Square URL provided in the request body, along with the player's URL (@id).

### Workflow:

**Initialization**:

* 1. An agent starts by accessing the /index entry point to discover available actions.
  2. The agent registers itself via /register.

**Game Setup**:

* 1. A game is initiated by posting player details to /registerGame.
  2. The response provides a Game ID.

**Gameplay**:

* 1. Players interact with /Board to check the game state.
  2. Moves are made by sending PUT requests to specific /SquareXX endpoints with the relevant Game ID and player details.

**Result Retrieval**:

* 1. Once the game concludes, the /result endpoint can be queried to determine the winner.

### Ontology Context:

The API uses the Tic Tac Toe ontology to represent the game's structure and logic. This ensures that all actions, roles, and relationships are semantically defined, enabling agents to interact intelligently with the environment.

### What Ontologies Represent in This Context:

**Entities and Their Classes**:

* 1. The ontology includes concepts such as:
     1. Game: Represents the Tic Tac Toe game as an entity.
     2. Square: Represents individual squares on the board (e.g., Square11, Square12).
     3. Move: Represents a player's action of marking a square.
     4. PlayerRole: Defines the roles of players, such as XPlayerRole and OPlayerRole.

**Properties and Relationships**:

* 1. Ontologies describe the relationships between entities. For example:
     1. A Move links a Square to a player (moveTakenBy).
     2. A Game has a set of Squares (e.g., inSquare relationships).
     3. A Game has two roles (XPlayerRole and OPlayerRole) assigned to players.

**Machine-Understandable Semantics**:

* 1. By using ontologies, the API expresses data in a way that software agents can interpret consistently. For example:
     1. @context in JSON-LD maps terms to the ontology's definitions.
     2. Terms like ttt:Game or ttt:Move link directly to the ontology, ensuring the agent knows their meaning and relationships.

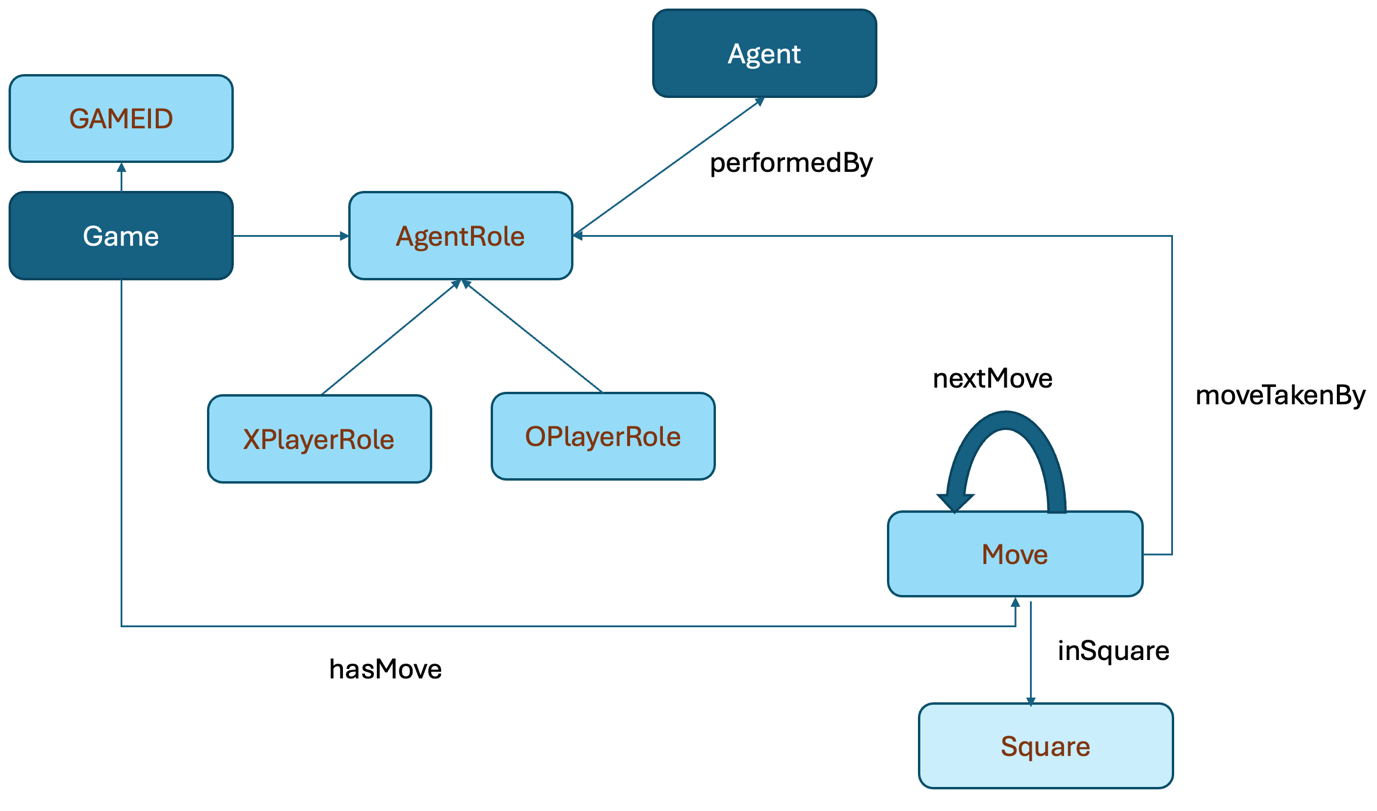
**Dynamic Instantiation**:

* 1. When a game starts, the ontology's abstract concepts (e.g., Square) are instantiated into specific entities (e.g., Square11/1234 for square 11 in game 1234). These instances represent the current state of the game.

### Example from the PDF:

* The ontology includes concepts such as:
  + Square for places on the board.
  + Move for game actions.
  + Relationships like inSquare and moveTakenBy.

These are used to create an RDF graph of the game's state, allowing agents to navigate and interact with the game seamlessly while adhering to the rules defined in the ontology.



Relationships:

* moveTakenBy: Links the Move to the player (via their AgentRole) who made the move.
* inSquare: Links the Move to the specific square on the board where it occurred.
* nextMove: Indicates the sequence of moves within the game.

### ****RDF Basics****

RDF is a framework for representing information about resources in a structured and interconnected way. It uses a graph model made up of:

* **Nodes**: Represent entities or concepts.
* **Edges**: Represent relationships between nodes.
* **Triples**: The basic unit of RDF, consisting of:
  + **Subject**: The resource being described.
  + **Predicate**: The property or relationship.
  + **Object**: The value or another resource.

For example:

* "Move1234" (subject) **inSquare** (predicate) "Square11" (object).
* "Move1234" (subject) **moveTakenBy** (predicate) "Player1" (object).

### ****2. Ontology in this Context****

The ontology provides the **blueprint or vocabulary** for the game, defining:

* **Classes**: Abstract concepts such as Game, Square, Move, and PlayerRole.
* **Properties**: Relationships like inSquare, moveTakenBy, and hasGameID.

### ****Example in the Tic-Tac-Toe Game****

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 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,

string target = http://localhost:8083/players;

if (hypermedia.hasTarget(string innerNodeId, target)) {

…

}

Would also work.