Web Environment in JaKtA

Federico Pirazzoli
federico.pirazzoli@studio.unibo.it

July 2025

The project aims to extend the capabilities of the JaKtA framework by introducing the capacity for the agent to have a Web-interactive environment. The primary goal is to provide agents with the ability to make HTTP calls to online resources, interpret responses in various formats, in our case (HTML, YAML, RDF), and specifically query the received data using specific query languages.

The core of the project is the design and development of a library of concrete actions that each agent can register and execute. These actions include, for example, the ability to make a GET, POST, PUT, or DELETE request to a specific URL, execute CSS queries on an HTML page, extract structured values from a YAML document using JSONPath, or query an RDF graph with SPARQL. Each action is encapsulated in a modular structure that conforms to the framework's abstractions, with inputs and outputs.

The envisaged use cases involve scenarios in which an autonomous agent needs to extract information from web pages, APIs, or semantic datasets. For example, consider an agent querying a public RDF endpoint to learn about an author's work, or an agent reading a YAML configuration from a remote resource to adapt its behavior.

The project offers a significant contribution both in terms of semantic interoperability and in terms of cognitive agents capable of actively interacting with the Web. This extension can find applications in domains such as Web Intelligence, Semantic Web, and Intelligent Automation based on dynamic data.

1 Goal/Requirements

On a more detailed note, the goals for this project were to integrate additional external actions, which would be implemented inside of the "environment" section of a JaKtA application, that may be used by agents to perform HTTP calls to online resources, in the form of a URL, and then use, based on preferences, of the parsers and query

languages implemented and available for the agents to obtain specific information on the queried resource. In short, the goals are as follows.

1. The agent must be able to execute HTTP requests: GET, POST, PUT, and DELETE;
2. The Agent must be able to:
   - Save the full contents of the response (raw data).
   - Save parsed results (DOM, RDF model, YAML data structure).
3. Provide query actions:
   - **HTML**: extraction via CSS selector.
   - **RDF**: Query via SPARQL endpoint on the RDF response.
   - **YAML**: YAML structure filtering with a JSONPath syntax.
4. Management and export of complex structures of querying the data (e.g., article list, metadata, RDF triplestore).

1.1 Scenarios

The main scenario/s of the project expect the end users to download the library and use the various actions available to perform data extraction on online resources, based on actions that their Agent implementation needs to do, so, for instance, a couple of hypothetical examples scenarios could be as follows:

- Information scraping on journalist websites, e.g. Scraping information about headlines from the journalistic website "La Gazzeta di Bologna";
- Extracting information about historical locations and monuments to be used, e.g. Information retrieval about the city of "Bologna" from the website DBpedia[1];
- Or obtaining information from settings file to be used in the future, or messaged-passed along to other Agents, e.g. Data extraction from publicly available YAML endpoints.

All of the previously mentioned examples have actually been implemented and will be shown later on, as a sort of "demo" to demonstrate the actions functionality.

1.2 Self-assessment policy

Generally speaking, the emphasis was placed on making each action implementation clear and well structured, that is, to make the action as simple as possible, while performing well, and make it easily extendable to modify/include further behavior that one may want to have. Therefore, the quality of the code was assessed based on its clarity, modularity, and adherence to the project requirements previously described.

2 Requirements Analysis

As requirements, the main factor that was kept in mind while developing the actions, aside from the ones previously mentioned, was to make the implementation adhere to the one described in the original JaKtA implementation, as to not divert from the formality from which these actions are supposed to conform to. For the various parsers and queries systems implementations, already existing library have been used as to make the implementation of the actions easier, and making it dependent on already existing resources which are regularly updated.

To recap the main requirements needed for the actions, what is needed is:

- External actions, and the parsers and query systems, implemented need to be simple, concise and extendable;
- The actions need to adhere to the structure of actions shown in the original JaKtA[2] library, as for example shown in the documentation;

3 Design

The projects actions, implement the design and architecture provided by the documentation to align with the already existing JaKtA documentation, meanwhile, the parsers and query systems, are implemented by deteaching them the from actions themselves, and change the implementation whenever is/will be needed. The query and parsers use external libraries to obtain online resources, and thereafter perform the information filtering that is necessary to obtain the wanted information out of the resource.

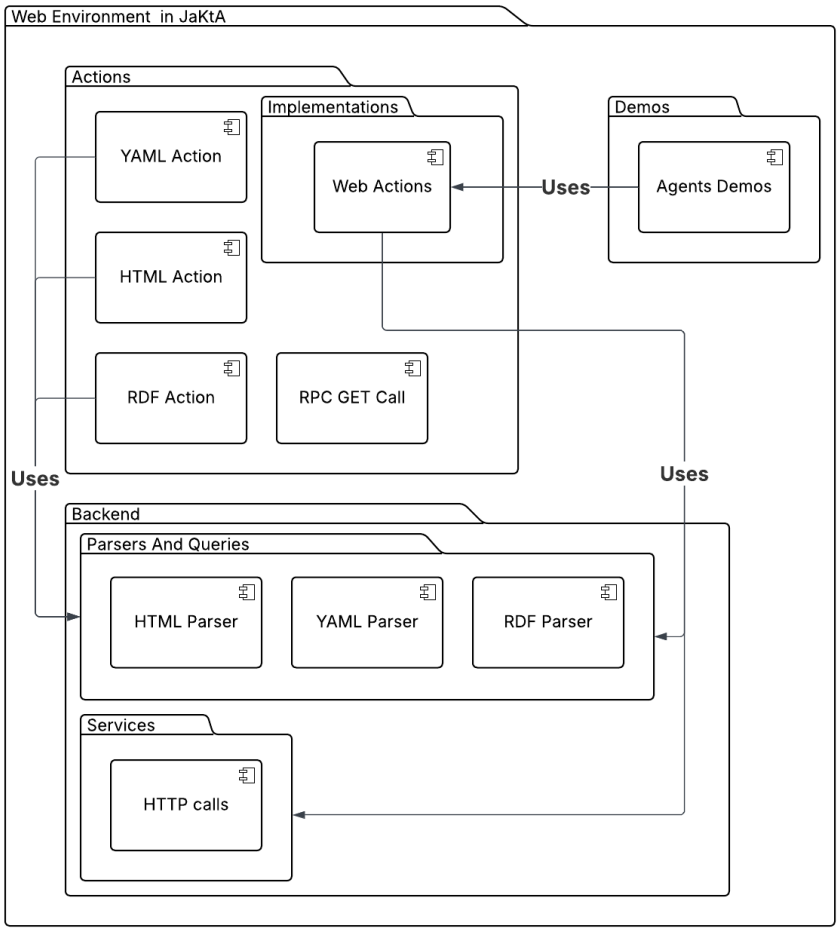

Figure 1: General diagram of the project structure

3.1 Structure

3.1.1 Actions

This package contains everything related to building external actions for JaKtA environments, as to let applications only import the necessary actions to import inside MAS programs.

RPC GET Call is responsible for all the GET calls that agents are able to do inside applications, with its main job being fetching only resources and saving them locally to be used again inside the environment, usually in conjuction with the actions related to querying the fetched data.

HTML, YAML and RDF Actions which are the actions implementations for directly parsing, and querying the data in the desired manner, so of course either by parsing and querying:

- HTML code, also supplying a CSS selector for the data that ought to be extracted;
- YAML syntax, again supplying a JSONPath selector for the wanted data;
- And RDF syntax, supplying once again a SPARQL query as selector in order to properly filter the information to fetch from the querying.

3.1.2 Implementations

This package instead is related to organizing the various system calls executed by the actions.

WebActions is responsible for what is stated above, containing and directing every call made by either a: HTTP, HTML, YAML or RDF external action, for every case, and those are:

- The GET calls to the outside;
- The HTML ones, contain operations to parse and query resources, or even fetch, parse and query directly;
- The YAML calls, which the same operations but for the YAML resources;
- And the RDF ones, which has operations same as the YAML and HTML ones.

3.1.3 Backend

This package meanwhile is responsible for handling all the incoming requests from the implemented actions, so it's mainly made up of the implementations of the parsers, the query systems for the three chosen data formats, and the HTTP calls made.

3.1.4 Parsers and Queries

HTML, YAML and RDF Parsers and Queries constitute the bulk of the implementation of the parsers and query systems, using external libraries for the handling of these, to be more specific:

- For the HTML code, the ***JSoup*** library has been used, to both parse the data into a DOM readable state, and upon this create Document object, CSS queries can be executed to extract data from the content of the DOM;
- For the YAML part, two libraries have been used:
  - the ***FasterXML*** library for parsing the YAML content into a valid JSON format;
  - and the ***Kotlinx Serialization*** library to instead perform queries using a JSONPath system;
- And finally for the RDF content, the ***Apache Jena*** library has been used to both read and query the RDF information into a readable and easily accessible format. The query system takes in input a full SPARQL query.

3.1.5 Services

The Services package and HTTP calls system, contain the backend implementation for the HTTP calls executed by the external actions, where all the RPC calls, so therefore the GET, POST, PUT and DELETE actions have been very simply implemented using the ***OkHTTP*** library.

Figure 2: Class diagram with the dependencies of the system

3.2 Behaviour

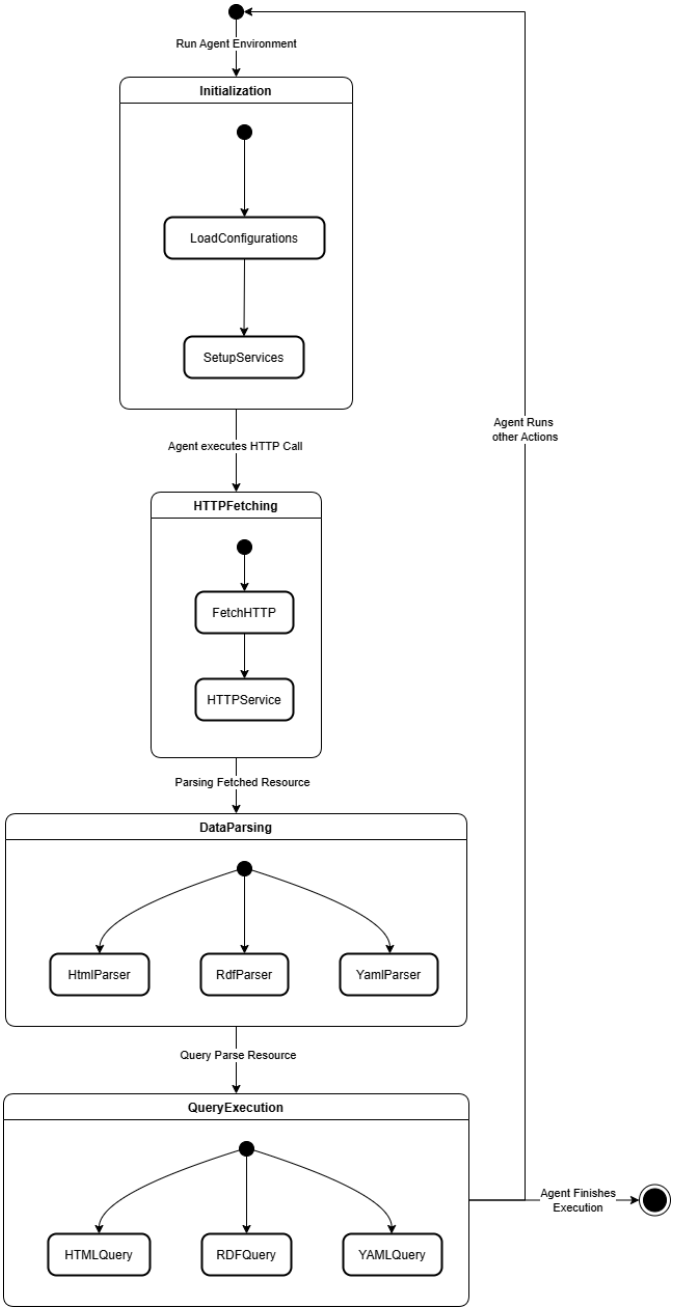

Figure 3: State diagram representing the behavior of an Agent using the external actions implemented

As an example of executing a JaKtA application, the Agent/s will first instantiate the external actions that have been implemented, and once that is done, HTTP calls will be

made with the appropriate parsing and querying of the material that has been delivered, to then be used afterwards.

3.3 Interaction

In this part, the interactions between an instantiated Agent and the provided external actions will be shown via Sequence Diagrams.

3.3.1 HTML Processing

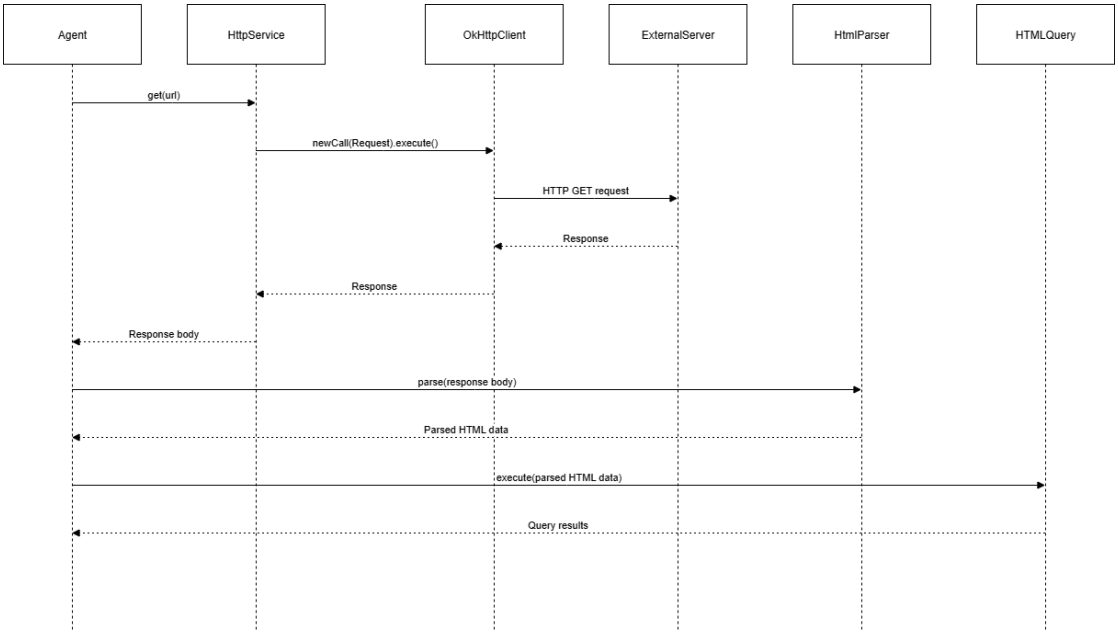

Figure 4: Sequence diagram representing the interactions of an Agent processing a HTML resource

As we can see from the provided diagram, and agent will first execute the GET HTTP call (as an example) to fetch the required online resource from an external server, by instantiating and running the *OkHTTPClient*. Thereafter, parsing on the fetched data will be performed, and once the information has been properly fashioned in the correct format, querying will be made to retrieve the material we are interested in, in this case by querying HTML code.

The following interactions provided are very similar to the HTML one described above, mainly only changing the format of the data that has been fetched.

3.3.2 YAML Processing

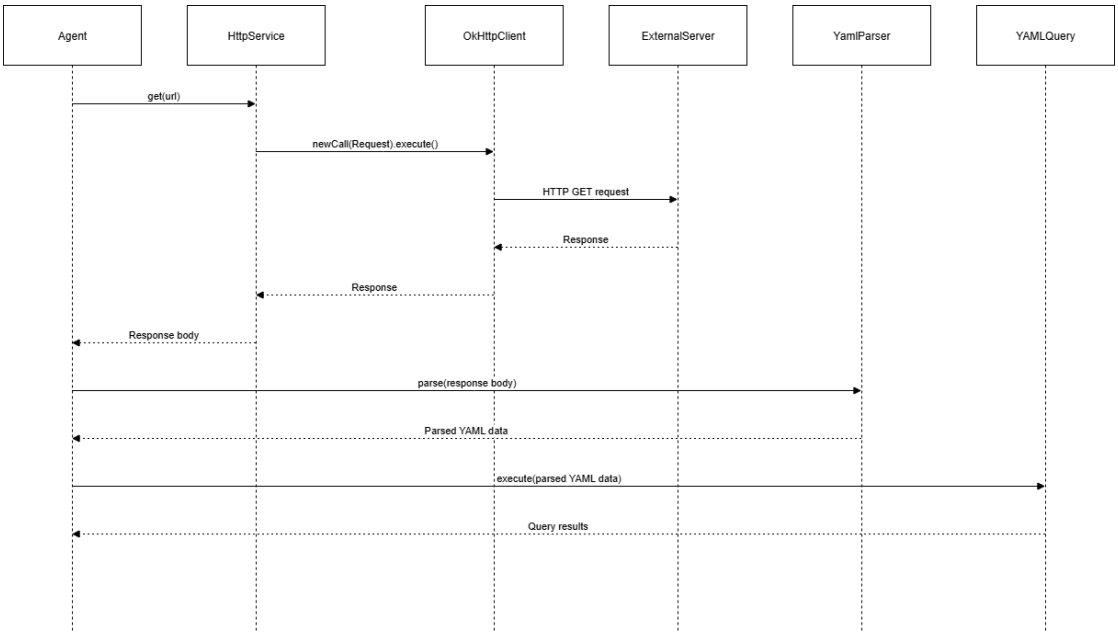

Figure 5: Sequence diagram representing the interactions of an Agent processing a YAML resource

3.3.3 RDF Processing

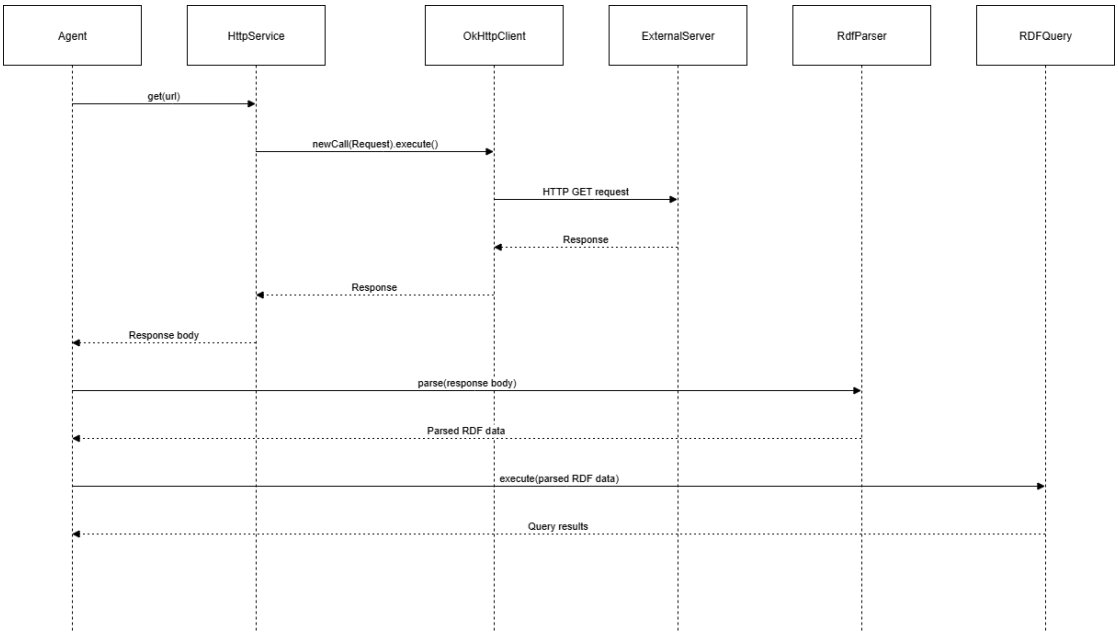

Figure 6: Sequence diagram representing the interactions of an Agent processing a RDF resource

4 Implementation Details

In this section the library used for the implementation of the HTTP calls, parsers and query systems will be briefly explained, with linkage to the respective pages.

4.1 HTML

4.1.1 JSoup

JSoup[3] is a robust, open-source Java library designed for working with "real-world" HTML—whether clean or malformed. It implements the WHATWG HTML5 specification to parse HTML into a DOM model similar to what modern browsers use.

It is one of the most widespread library, for Java and JVM languages, for dealing with HTML content, and considering the constant update that the library goes through, I decided to use to have up-to-date functionalities, even for future works.

4.2 YAML

4.2.1 FasterXML/Jackson

Jackson (FasterXML)[4] is a high-performance, open-source data-processing suite for the Java ecosystem, widely regarded as "the Java JSON library". Initially focused on JSON, it's evolved into a comprehensive toolkit that supports streaming, databinding, and tree-model APIs. Jackson can parse/generate JSON, and through various modules it also handles formats like XML, CSV, YAML, BSON, CBOR, Avro, Protobuf, and more.

The library is structured into core modules:

- **jackson-core**: Implements fast, low-level streaming parser/generator APIs;
- **jackson-databind**: Implements fast, low-level streaming parser/generator APIs;
- **jackson-annotations**: Implements fast, low-level streaming parser/generator APIs;

In our cas, we obviously used it to parse the fetched content into a usable JSON format, that would be further implemented when performing queries afterwards to retrieve content.

4.2.2 Kotlinx Serialization

kotlinx.serialization[5] instead, is a multiplatform, compiler-plugin-based serialization framework for Kotlin, allowing you to serialize/deserialize @Serializable classes across formats like JSON, Protobuf, CBOR, Hocon, and others. It generates efficient, reflection-free serializers at compile time, making it fast and lightweight, especially for cases like ours.

For the project, kotlinx has been used to perform JSONPath-like queries on the previously parsed data through Jackson, and considering the speed of kotlinx, this is done well efficiently.

4.3 RDF

4.3.1 Apache Jena

Finally, **Apache Jena**[6] is an open-source, Java-based Semantic Web framework that offers a comprehensive API for managing RDF graphs, parsing RDF data, and executing SPARQL queries.

Again for this project, Jena has been used to parse and query RDF content; the fascinating thing about Jena is that it's capable of parsing a various amount of RDF formats, like:

- Turtle;
- RDF/XML;
- TriG;

- N-Triples;
- JSON-LD;

while also performing syntax validation, conversion, inference etc. Meanwhile for the querying part, Jena fully supports SPARQL 1.1 standard library, so it's very easy to perform queries on the parsed content.

5 Self-assessment / Validation

The fulfillment of the requirements was ascertained by what was determined optimal while conversing with Prof. Ciatto when appointing the requirements of the project, which was finally in the implementation of the specifics mentioned at the beginning of the document, culminating in a couple of demonstrations, as if used in a normal BDI JaKtA environment, as will be shown in the "Usage Examples" section.

6 Usage Examples

Consider the following as multiple test runs of agents utilizing the external actions implemented to fetch, parse and query data, based on the subsection title, of which all exist as runnable applications inside the project.

6.1 HTTP

Listing 1: Code example of an Agent executing the HTTP GET external action to fetch data

```
import actions.FetchHTTP
import it.unibo.jakta.agents.bdi.dsl.mas

fun main() {
    mas {
        environment {
            actions { action(FetchHTTP) }
        }

        agent("GET Fetcher"){
            goals {
                achieve("GETurl")
            }
            plans {
                +achieve("GETurl") then {
                    execute("fetchHttp"("https://example.com", X))
                    execute("print"("The value is ", X))
```

```
                    }
                }
            }
        }
.start()
}
```

As we can see from the code, The agent fetches content from an always accessible and online endpoint website called "Example.com", which is then used to get the data, and is outputted correctly as follows:

Listing 2: Data fetched from previous shown action

```
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI",
        "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

6.2 HTML Parsing and Querying

Listing 3: Code example of an Agent executing the parsing and querying of HTML code

```
import actions.FetchHTTP
import actions.HTMLParseQuery
import it.unibo.jakta.agents.bdi.dsl.mas
```

```
fun main() {
    mas {
        environment {
            actions {
                action(HTMLParseQuery)
                action(FetchHTTP)
            }
        }

        agent("Webscraper"){
            goals {
                achieve("extractParagraphs")
            }
            plans {
                +achieve("extractParagraphs") then {
                    execute("fetchHttp"("https://gazzettadibologna.it", X))

                    execute("htmlParseQuery"(X, ".entry-title a[title]", Y))
                    execute("print"("Titoli Gazzetta di Bologna:", Y))

                    execute("htmlParseQuery"(X, ".author a[title]", Z))
                    execute("print"("Autori Gazzetta di Bologna:", Z))

                    execute("htmlParseQuery"(X, ".posted-on time", W))
                    execute("print"("Date Gazzetta di Bologna:", W))
                }
            }
        }
    }
.start()
}
```

Here instead, we can see that we fetch data in the form of the main page of the journalistic website "La Gazzetta di Bologna", by which, once the data is fetched, it's parsed and query, through the second argument passed to the *htmlParseQuery* action, using a CSS selector, to obtain, in order:

- The titles of the articles of the main page;
- The authors of the articles of the main page;
- And the dates of the articles of the main page.

6.3 YAML Parsing and Querying

Listing 4: Code example of an Agent executing the parsing and querying of YAML data

```
import actions.FetchHTTP
import actions.YamlParseQuery
import it.unibo.jakta.agents.bdi.dsl.mas

fun main() {
    mas {
        environment {
            actions {
                action(YamlParseQuery)
                action(FetchHTTP)
            }
        }
        agent("YAML Fetcher And Query"){
            goals {
                achieve("parseYaml")
            }
            plans {
                +achieve("parseYaml") then {
                    execute("fetchHttp"("https://filesamples.com/samples/code/yaml/verify_apache.yaml", X))

                    execute("yamlParseQuery"(X, "$[0].hosts", Y))
                    execute("print"("Gli host del file di verifica Apache sono ", Y))

                    execute("yamlParseQuery"(X, "$[0].remote_user", Z))
                    execute("print"("Il Remote User del file di verifica Apache e' ", Z))

                    execute("yamlParseQuery"(X, "$[0].tasks[*].name", W))
                    execute("print"("I task del file di verifica Apache sono ", W))
                }
            }
        }
    }
.start()
}
```

For the YAML content, I used the "FileSamples" website as an example endpoint of YAML data to fetch and parse, which worked very well, and the data is parsed and queried, this time using a JSONPath format to navigate the converted fetched data, as we can see from the second argument of the *yamlParseQuery* action.

6.4 RDF Parsing and Querying

Listing 5: Code example of an Agent executing the parsing and querying of RDF data

```
import actions.FetchHTTP
import actions.RDFParseQuery
import it.unibo.jakta.agents.bdi.dsl.mas

fun main() {
    mas {
        environment {
            actions {
                action(RDFParseQuery)
                action(FetchHTTP)
            }
        }
        agent("RDF Fetcher And Query"){
            goals {
                achieve("parseYaml")
            }
            plans {
                +achieve("parseYaml") then {
                    execute("fetchHttp"("https://dbpedia.org/data/Bologna.ttl", X))
                    execute("rdfParseQuery"(X, """
                        PREFIX dbo: <http://dbpedia.org/ontology/>
                        PREFIX dbr: <http://dbpedia.org/resource/>
                        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                        SELECT ?labelEN ?abstractEN WHERE {
                          dbr:Bologna rdfs:label ?labelEN ;
                                      dbo:abstract ?abstractEN .
                          FILTER (lang(?labelEN) = "en" && lang(?abstractEN) = "en")
                        }
                    """.trimIndent(), Y))
                    execute("print"("L'abstract di Bologna e' ", Y))
                    execute("rdfParseQuery"(X, """
                        PREFIX dbo: <http://dbpedia.org/ontology/>
                        PREFIX dbr: <http://dbpedia.org/resource/>
                        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

                        SELECT ?label ?pop ?area ?sisterCity ?founding WHERE {
                          dbr:Bologna rdfs:label ?label ;
                                      dbo:populationTotal ?pop ;
                                      dbo:areaTotal ?area .
                          OPTIONAL { dbr:Bologna dbo:sisterCity ?sisterCity. }
                          OPTIONAL { dbr:University_of_Bologna dbo:established ?founding. }
                          FILTER(lang(?label)="en")
                        }
                    """.trimIndent(), Z))
                    execute("print"("Alcuni elementi di interesse di Bologna sono ", Z))
                }
            }
        }
    }
.start()
}
```

Finally, for the RDF content, I used DBPedia as an endpoint from where to fetch data for testing, in this case I retrieved data relating to the city of Bologna, where I retrieved the .ttl data, and properly parsed it, and executed two SPARQL queries successfully, retrieving the abstract related to the city, and various points of interest present inside of the file.

7 Conclusions

In conclusion, the project aimed to expand the number of external actions of the JaKtA library, more precisely, by adding multiple actions to execute HTTP calls, and also to use these implemented actions to fetch data in the format of HTML, YAML and RDF, to parse it properly, and query it to retrieve data that might be important.

The project was successful in doing so, presenting actions that, once executed, satisfy all the requirements previously mentioned in this document, all the while maintaining the implementations of said actions clean, modular and flexible, in order to open up the possibility to modify them if the implementation needs it.

7.1 Future Works

For future works, definitely other parsers and query systems for other formats could be implemented, giving a wider range of abilities for the Agents to perform their work. Further modularity of the backend components could be implemented, for example by detaching the detaching the parsing from the querying for the external actions.

7.2 What did we learned

This project was a good experience to further improve my skills in working in BDI environments, especially custom ones like JaKtA, which proved definitely an interesting and inspiring read in studying the library for the project's work. Furthermore, it was very instructional to work with parsers for the HTML, YAML and especially RDF format, and the adopted query systems, like SPARQL for RDF, which I've never used before and proved to be challenging to implement.

References

[1] DBPedia, Global and UnifiedAccess to Knowledge Graphs. `https://www.dbpedia.org/`.

[2] JaKtA, a Kotlin internal DSL adding support for the definition of BDI agents in the spirit of the well-known Jason language. `https://github.com/jakta-bdi/jakta`.

[3] JSoup, Java HTML parser for real-world HTML. `https://jsoup.org/apidocs/`.

[4] Jackson, Suite of data-processing tools for Java. `https://github.com/FasterXML/jackson`.

[5] Kotlinx.Serialization, Provides sets of libraries for various serialization formats. `https://github.com/Kotlin/kotlinx.serialization`.

[6] Apache Jena, A free and open source Java framework for building Semantic Web and Linked Data applications. `https://jena.apache.org/`.