

Web Environment in JaKtA

Distributed Systems course project

Federico Pirazzoli

Functional Requirements / Goals

The goals of the project are to extend the capabilities of agents inside the JaKtA framework, by implementing external actions usable by agents, to have a Web-Interactive environment, all of which has been discussed organized with Professor Giovanni Ciatto due to the nature of project itself.

In short these are:

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

Non-Functional Requirements

- Implementations needs to be simple, concise and extendable. This goes for:
 - External Actions;
 - Parsers;
 - Query Systems;
- Actions need to adhere to JaKtA's original structure;
 - As shown in documentation;
 - And demos online.

Scenarios

The main idea is to provide the JaKtA agents with Web-Interactive capabilities to fetch, post or modify online resources.

Some example scenarios include:

- **HTML:** Information scraping on journalist websites, e.g. Scraping information about headlines from the journalistic website "La Gazzeta di Bologna";
- **YAML:** Extracting information about historical locations and monuments to be used, e.g. Information retrieval about the city of "Bologna" from the website Dbpedia;
- **RDF:** 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.

JaKtA

- **What it is:**

JaKtA is an internal DSL (Domain-Specific Language) built in Kotlin that enables developers to define BDI (Belief-Desire-Intention) agents in a clean and expressive way, inspired by the Jason language paradigm.

- **Purpose:**

It provides a structured framework to model and implement intelligent agents using BDI concepts directly within Kotlin, blending object-oriented and functional programming styles.

- **Background:**

JaKtA has been described in academic literature—namely, in a 2023 conference paper and a 2024 journal article that illustrate its design, usage, and integration into Kotlin-based BDI agent systems, of whom main contributors are Giovanni Ciatto and Danilo Pianini.

- **How to use it:**

It's available through Maven Central and integrates seamlessly via Gradle (Kotlin DSL), making it accessible for Kotlin developers to import and experiment with.

Agent Web Actions

One of the main implementation was that of Web Actions for the JaKtA Agents, more specifically they have been implemented using the OkHTTP library.

The following actions were developed:

- **GET**: is responsible for all the **GET** calls that agents can do inside applications, with its main job being fetching only resources and saving them locally to be used again inside the environment, usually in conjunction with the actions related to querying the fetched data;
- **POST**: whose purpose is to append data to online endpoints to be saved and used later;
- **PUT**: whose purpose is to modify the data of online endpoints, similarly to the **PUT** action;
- **DELETE**: finally, the action to delete the information and data of online endpoints.

During the development of the project, mainly the **GET** action has been used for demos and to demonstrate the other features, which are the parsers and query systems.

Parsers and Query Systems

The other main part of the project was the development of the Parsers and Query Systems for the retrieved data.

The formats to process that were decided together with Professor. Ciatto, were:

- **HTML:**
 - Where the focus is to parse the **DOM** elements properly and query for the elements to be extracted;
 - Querying is done through a **CSS**-like selector;
- **YAML:**
 - The focus is to parse the key elements of the files;
 - Query them through **JSONPath**;
- **RDF:**
 - Information is parsed through a **JSON**-like structure;
 - Querying is done via **SPARQL** actions.

HTML

For the Parsing:

- The DOM is parsed through the **JSoup** library;
- Data is extracted and made into a readable state;
- Then a Document object is created.

For the Querying:

- Again, **JSoup** is used to perform queries on the **Document** object;
- Querying is done through **CSS**-like selectors

YAML

For the Parsing:

- The **YAML** text is parsed into a valid **JSON** format;
- The library used is the called **FasterXML**;
- Can be used to parse other formats;

For the Querying:

- To query the **JSON** objects, the **Kotlinx Serialization** library is used;
- Queries are done through a **JSONPath**-like structure.

RDF

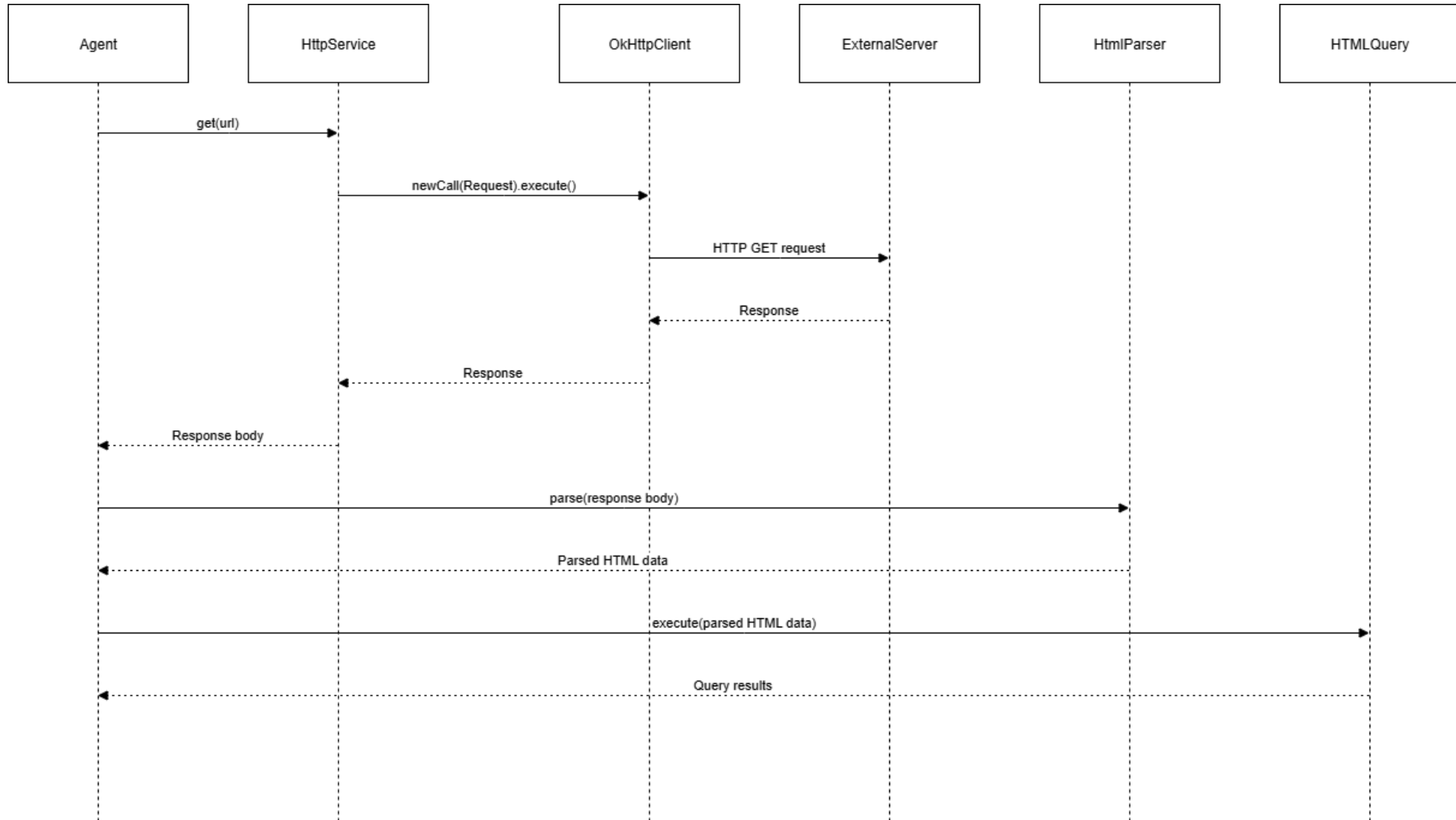
For the Parsing:

- The **RDF** information once again is parsed into a valid JSON format;
- The library used is the called **Apache Jena**;
- Used to manage and query **RDF** data and graphs;

For the Querying:

- To perform the queries, the JSON data parsed is queried with the same library;
- The queries are done using the **SPARQL** format.

General Usage Example – HTML



Future Works

Other formats can be implemented, alongside their parser and system to query them.

Some examples could be:

- **JSON**;
- **XML**;
- **CSV**;
- **TOML** and others;

Further modularity of the backend actions that the agents can perform.

Thanks for your attention!