# RESTmantics
## -Scholarly HTML-

Authors:
1. Bogdan Volosincu
Mail: bogdan.volosincu@gmail.com
2. Andrei Avram
Mail: avram1andrei@gmail.com
3. Teodora Hoamea
Mail: teodora.hoamea@gmail.com

## Project description:

The developed web application (RESTmantics) is a framework that implements a tool that interacts by text with a REST API. The following concepts will be used in the implementation: modeling vocabulary, taxonomies, thesaurus with RDF schema and SKOS and a database based on models.

## Requirements:

- Business requirements:
  - RESTmantics should be able to recognize words in common language
  - RESTmantics should be able to send a request to the API and search in the client databases
  - RESTmantics should be able to return all the related results to the user search
- User requirements:
  - The user should be able to enter a word in different languages (romanian/english) and receive results for his search
  - The user should be able to click on a result and then he/she should be able to access the URL behind that result
- Software requirements:
  - Performance:
    - The page should be able to send a feedback to the user in less than 0.4 sec
  - Usability:
    - This application should be able to be accessed from any browser
    - The operating systems should not affect the functionality of the framework
  - Other requirements:
    - Frontend: HTML, Css, JavaScript, REACT
    - Backend: NodeJs, Sails
    - Testing: java, Selenium, Postman, K6
    - OpenAPIs: Swagger

## Preliminary Considerations:

- Internal data structures/models:
  The internal data is organized in models based on Sails v1.x. They represent a set of structured data, called records, and they correspond to a collection in the database.

  Examples of models in this application:
  - Actor has the following attributes: id {string}, name {string}, surname {string}, birthday {string}, movies {json}, plays {json}, foto {json}, prizes {json}.
  - Movie has the following attributes: id {string}, title {string}, cast {json}, isReleased {boolean}, isPlaying {boolean}, isSeries {boolean}, releasedTimestamp {string}
  - Review has the following attributes: id {string}, starts{number}, comment {string}, movieId {string}
  - User has the following attributes: emailAddress{string} (this attribute is required and unique), emailStatus{string} (default value = confirmed), emailChangeCandidate {string}, password {string} (this attribute is also required), fullName {string}(required), isSuperAdmin {boolean} (this attribute indicates if the user has extra permissions), passwordResetToken {string} (A unique token used to verify the user identity), passwordResetTokenExpiresAt {number}, emailProofToken {string}, emailProofTokenExpiresAt {number}, stripeCustomerId{string} (The id of the customer entry in Stripe associated with this user), hasBillingCard {boolean}, billingCardBrand {string}, billingCardLast4 {string}, billingCardExpMonth {string}, billingCardExpYear {string}, lastSeenAt {number}

  The models are accessed from within the controller actions, helpers or tests, in this way the call model methods communicate with the database.

- External data sources:
  Because this application is an OpenAPI schema for movies, an external data source is Internet Movie Database, IMDb which is a massive online website directory housing tons of information related to films, TV programs, video games, internet streams and more.

  The list of movie database APIs such as the Open Movie Database and Internet Movie Database API can be accessed at the link:
  [https://rapidapi.com/collection/omdb-imdb-apis](https://rapidapi.com/collection/omdb-imdb-apis)

## Application flow:

The Proof of concept that we build exposes an OpenAPI schema for movies, similar to IMDb.com.
From the graphic interface the user is able to query the data using natural language.

The application will read the input from the user through an input text field.
In the backend, the input will be sent first to a syntactic and semantic module that will analyze the text and with the results from the parser the application will create a query to map user's words with the OpenApi schema as best as it can.

Example of scenarios:
User enters inputs like:

1. Movie with Angelina Jolie. => extract Subject, Object and map to movies titles, movies cast or actor name.

2. Plays of Shakespeare. => map play to themes from OpenAPI schema like drama, movies, tragedy, theatre (query synsets from WordNet )

3. Best movies from last year. => map adjectives to Reviews, to prizes won by movie, actors

To map the words from user input to OpenApi schema we also create a dictionary of semantic terms based on OpenAPI endpoints, parameters and data models.

## Application architecture

Because this application requires a significant storage space and execution time as short as possible, the architecture is one based on microservices and is using APIs to connect the services. In this way, the implementation is distributed and loosely coupled which leads to dynamic scalability and fault tolerance.
In this case, the actor enters the input into the UI Service which plays the role of the client. Then the user search is processed using a Controlled Vocabulary modeled by a RDF Schema and it is created a request to a REST API gateway which redirects the request to multiple REST APIs parallel services. Each of them is connected to a database, over which the request will be made. In the opposite way, one of the databases is sending a feedback to its REST API service then to the gateway. In the final stage, the user gets the response via UI Service.

## Linked Data Principles:

In this section it will be developed how this web application incorporates the Linked Data Principles. This principles refers to URIs:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information.
4. Include links to other URIs so that they can discover more things.

These four rules help keeping data interconnected and they represent the unexpected re-use of information which is the value added by the web.

In this case, the response that the user is going to receive will consist of one or more URIs, based on how many results will be found in the database. Those URIs will contain the URL address for a significant page for that keyword.
The first two rules are respected by providing the user HTTP URIs responses that respect semantic web technology.
The third rule is applied by formatting every URI using RDF and the fourth rule is used if a user searches a keyword, for example an actor, he/she will receive all the links related to that actor:

When the user sends “john” a RDF file will be created<[http://RAT.org/john](http://RAT.org/john)> then it will use the local identifiers within the file #smith, #Alan etc.

This is how the RDF file will look like:

```
<rdf:Description about="#john">
<fam:child rdf:Resource="#smith">
 <fam:child rdf:Resource="#Alan">
</rdf:Description>
```

The WWW architecture is going to return a global identifier "http://RAT.org/john#Alan" to Alan.

## RDF - Schema

Resource Description Framework is an open standard of the W3C to describe digital resources with semantic meanings. It describes digital resources by defining and using classes and properties. The RDF Schema will help in converting the user input into a language that the REST API can process and initialize the process. It facilitates the dialogue between the client and application.
One of the formats that defines a statement in RDF is Turtle (.ttl) that is a syntax that has abbreviations, prefixes and is easier to read for humans. The next exemple from W3 describes the relationship between Green Goblin and Spiderman:

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ;    # in the context of the Marvel universe
    foaf:name "Green Goblin" .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
```

```
    a foaf:Person ;
        foaf:name "Spiderman", "Человек-паук"@ru .
```

**rdf: domain** and **rdf: range** are used to semantically describe and derive relationships between objects. The first one declares that a property belongs to one or more classes and the second one is used to deduce that a value of an instance belongs to one or more classes.

The resources in RDF are linked globally because they are identified by IRI (International Resource Identifier) which is a generalization of URL. This will help convert the response from the database in a language that the user can easily understand. The IRI can be built using a prefix. For this, we have this example from W3:

To write `http://www.perceive.net/schemas/relationship/enemyOf` using the original Turtle syntax for prefixed declaration:

```
@prefix somePrefix: <http://www.perceive.net/schemas/relationship/> .
<http://example.org/#green-goblin> somePrefix:enemyOf <http://example.org/#spiderman>
```

## Project API Architecture

At this moment, the user is able to enter a keyword and get relevant responses from the application. This is possible due to the APIs calls that RESTmantics is performing in 3 levels:

Level 1:

The user is sending a keyword to the RAT System which is a core system for interacting with other API using natural language, then the Software System is sanding calls to Swagger to generate the openAPI schema, Google Cloud Natural Language to process the natural language to determine entities and Google Translate API, which is a System used for term translation when processing tweet texts.

Level 2 - rest API interactive tool:

This is starting with App Controller which is a Web API. First, It is making API calls to the NLP Module to deal with natural language processing (tokenization, remove stop words, extract entities, translate text). NLP will call via API Google Cloud Natural Language and Google Translate API. Then, the Web API is making a call to Request Builder to create a request for openAPI based on information received from the NLP module. This request is sent via API call to OpenAPI schema to generate schema with hook module. The received data will be stored in the Local Disk DataBase.

Level 3 - NLP Module

At this level it will be detailed how the NLP module is working. After the App Controller is making a call to the NLP module this is arriving in the NLP controller which is binding all components. It uses Text Translator which makes an API call to Google Translate API to translate text from any language to english. After that, the NLP Controller uses Text Tokenizer to extract tokens and remove stop words and Named Entity Recognizer, which is making an API call to Google Cloud Natural Language to extract entities from text, such as syntactic functions: noun verbs and type: location, name. After this step is done, the result is sent to the Open API schema.

## Frontend development

The frontend part was developed so that it is user friendly and easy to use without any additional knowledge about the technologies behind RESTmatrics. It was developed following the single responsibility principle and it is low coupled and has a high level of cohesion.
The frontend was built using react and javaScript for the functional part and using css and HTML for the visual part.
It has more important elements such as:

1. index component that is just calling the App page

2. App page is calling the interface pages and is routing them using the BrowserRouter, Switch and Route from “react-router-dom”

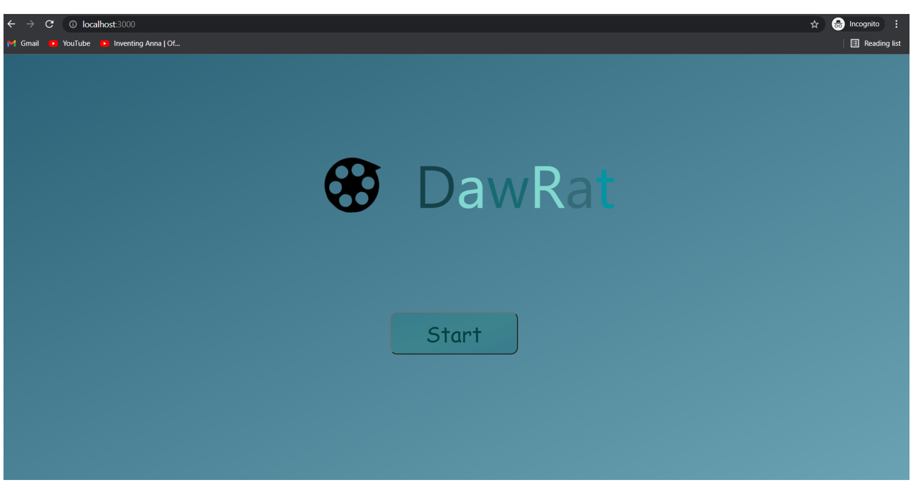
3. The Home page is the first page that the user is getting in touch with. It contains the logo, the title of the application and the start button. After the button is pressed the SearchForm is called

4. The SearchForm component contains the input field and the search button. After the input is entered in the field, the search button calls the SearchPage that is displaying the results.

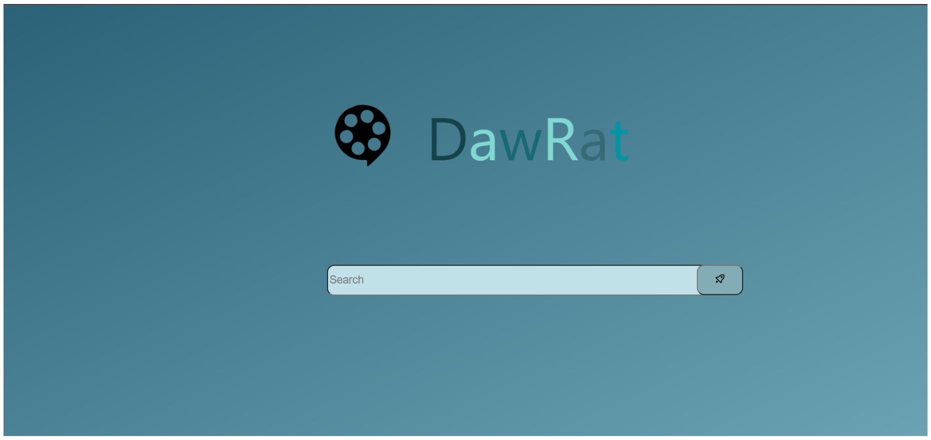
5. The state of the input component is tracked by reducer,js which verifies the state of the input and creates an actionType to set the search term to be the user input.

6. The state is available because of the StateProvider which creates the context and applies the Provider, imported from react, to the reducer.

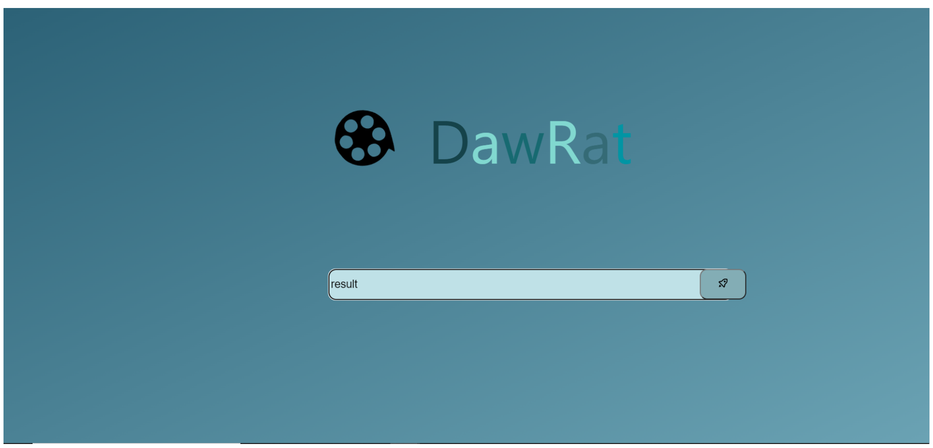
First version of frontend:
Home:



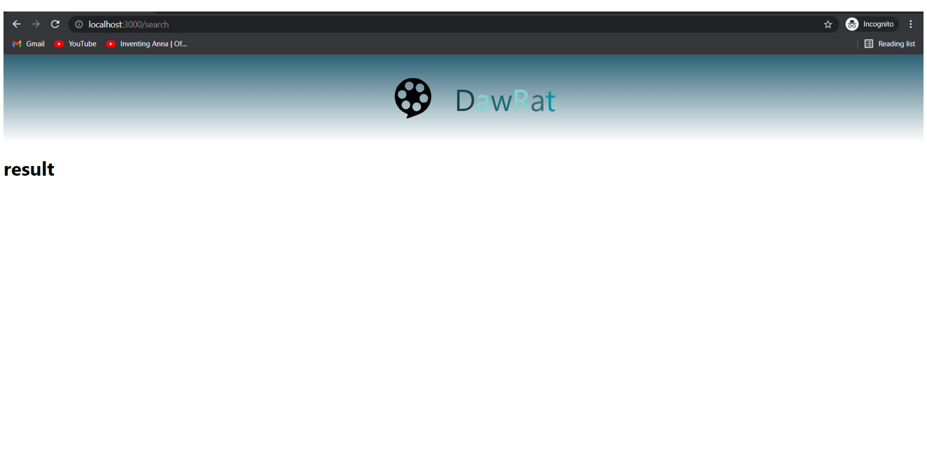
After click “Start” button:




After the input:




After click on the search button:



For the simplification of  the solution, the frontend became less complex.

## Backend development

We developed RESTmantics application to build REST API queries using natural language processing (NLP) that enhance the REST API openApi schemas with the means and documentation of RDF resources and standards like SKOS and OWL.

To reach the goal of interoperability across today’s expanding networks of web API's is an immense and infinitely extensible endeavor. Our implementation is a solution for assemble and extend most of the information in this semantic web schemas in a way to make it feasible in web querying, resource discovering, data mining, web scraping, and other general purposes.

Encoding information in RDF compliant documents allows it to be passed between computer applications in an coherent way but the verbosity can be a challenge in an agile environment where the technology upgrades, business challenges and change requests, distributed teams are present.

OpenApi schemas are an important resource and data format for web developers and web applications, for declaring the API structure, models and methods. Used mainly and strongly for documentation purposes the OpenApi schema being a part of RDF standard offers more possibilities to be employed in web API query generation.

The guarantee that between all these standards is always a good fit is not given and this can lead to many definitions remaining outdated or out of reach and they remain in a dormant state. This is the reason we have built RESTmantics to take advantage of RDF definitions that contain collections of properly defined and correlated concepts.

Concepts by themselves could be really strict and can raise a variation problem in the sense that are limiting, being restricted only for a certain domain. On the other side an ontology for example gives the advantage that has clear boundaries and well defined relations.

Since standards like OpenApi, Simple Knowledge Organization System (SKOS) and (OWL) are close neighbors in the Semantic Web this offers a possibility for reuse, despite the differences, there is good calibration, making it possible to develop a set of correspondences between components of the data models.

We thought that if starting from a more strict domain like an ontology we can enhance it with NLP algorithms to emulate a new domain that is flexible and coherent to build REST API queries.

The solution takes this steps :

A. Take a input from a user (search, tweet, request, comment, review)

B. Scan and build a vocabulary based on API OPEN API:

1. model name and attributes
2. resource (endpoints names, paths, parameters)
3. tags
4. descriptions

C. Enhance the user input with a semantic field based an a set of RDF schemas and ontologies available online

1. Query WordNet

D. analyze it using nltk and build a syntactic tree

1. with the semantic field based on ontologies we pass it to the NLP module
2. based on syntactic tree we decide on the semantic analysis of words (verbs-actions, names, objects, etc)
3. determine a polarity and intensity of the semantic field

E. Build the query based on the polarity and intensity

F. Essentially advises on the selection and fitting together of concepts, terms and relationships to make a good thesaurus. SKOS addresses the next step, with recommendations on porting the resultant thesauri (or other 'simple Knowledge Organization Systems') to the Web.

G. Between the recommendations of these complementary standards, the teams responsible for them have maintained contact throughout. The respective data models are not identical

Despite the differences, however, there is good alignment, making it possible to develop a set of correspondences between components of the data models.

## Project Progress

At this moment, RESTmatrics is able to read the input from the user, process it and return relevant data about the topic searched. It is based on a movie collection, but as future work it is planned to make the context larger.

For the future, the user interface will be more complex and the application will give the user the possibility to download the result that one searched for.

Another next step for the development of RESTsemantic is that the results to be more specific for improving service quality.
Another aspect which we are working on is performance of the application to decrease the response time and the level of stress that the page is currently supporting.
In the future we are also aiming to return a more user friendly response that will be in the form of an URL that the user will access and get a wikipedia page about that topic.

## Conclusion

The application will convert the user input using RDF Schema and will send it to a REST API gateway and therefore to multiple parallel REST APIs that will search in databases in order to achieve an answer. The response for the user request will be an IRI/URL that will provide to him the information that he/she needs.
RESTsemantic is an user friendly application that gets as input a sentence about an actor, movies etc and it returns all the relevant data about that topic.

Bibliography:


1. Turtle: https://www.w3.org/TR/turtle/
2. RDF Schema: https://www.clearbyte.org/?p=5895&lang=en
3. Microservices: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
4. Microservices Architecture: https://www.devteam.space/blog/microservice-architecture-examples-and-diagram/
5. Swagger: https://github.com/swagger-api/swagger-editor