Dynamic generation of web service APIs and interfaces through compilation of JSON

Marisa Yeung, Nayan Deep Vemula and Sandy Siththanandan

Team 4

COEN 259 - Compilers

Professor Ming-Hwa Wang

Santa Clara University

Santa Clara, CA
Abstract

The goal is to dynamically create APIs for the entry of new and modified data object types on the fly. This will be achieved through the compilation and analysis of incoming JSON objects, implemented using the ANTLR framework in Java. By recognizing when the key-value pairs have been augmented with new elements, the required schema and documentation will be generated that will enable automatic adaptation of the interfaces to the database as well as the API for front end use in web development and other application end points.
Introduction

The vision of a Semantic Web of the future requires development of technologies which can ease and automate the digestion and annotation of meaningful structural properties in the mass loads of data being produced in our information systems without requiring up front specification and freezing of concept ontologies. The systems need to be not only easy for people to implement and understand but also 'understandable' to the machines so we can offload higher level needs to them like pinpointing answers to subtle and complex queries.

Full stack application development still involves intense amounts of effort on the part of large teams of highly technical software engineering staff. Many share the vision of enabling application development by laymen to the software field -- that domain experts might be able to easily specify their needs and create systems that can organize their information and processes. If these mechanisms were more easily harnessed, with lower barriers to entry, the information systems of the future could better grow, contain and manage the nuanced needs of the international community that all use the World Wide Web.

Our objective is to see how we might contribute to this effort, by developing an attempt to intelligently solve a particular aspect of difficulty in application development. Specifically, during agile application development, it may happen that data schemas need to evolve from their initial specifications. This causes difficulties on many fronts, including things like needing to reconsider and re-implement how the data will be input to the database, as well as communicating the new requirements for consumers of the data objects. We aim to investigate and prototype a framework which will recognize the evolution of data objects as they are first encountered, and propagate those changes to the components that require modifications to handle the changes.

We will approach this problem through parsing JSON objects as they are input into the system plus augmented actions via compilation techniques learned in this course. We hope to generate code that will enable an application to adapt to these data schema changes, accepting the new data forms without effort on the part of the consumer or producer of the application, and presenting front end interfaces which communicate the new fields in the complex data types.

We are aware of systems that exist such as CASE tools or object modeling initialization in frameworks which are supposed to enable ease of development, but as far as we know, these require up-front "describe once" design. We would like to advance the capability to adapt the data system as the user becomes aware of new requirements. Also we would like this tool to be simple and lightweight, so that it might be useful and interesting to non-technical people, for all kinds of as-yet to be imagined use cases. Since JSON syntax is quite human-readable, compared to other approaches which have involved XML or newly specified languages for
declaration of the data model, we think this may be a boon in the understandability of this approach.

For now, we limit our scope to prototyping with JSON objects, input into the Neo4J graph database, and attempting one context of client generation, with the possibility of different kinds of front end generation depending on what we can achieve in our time frame. Beyond this proof of concept, there is also the possibility of parsing other data interchange formats so that alternative systems can be compiled. With further enhancements, we would aim to be able to recognize redundancies and inconsistencies in data modelling by different users, to try to merge models within organizations, or even across organizations to recognize isomorphic complex data types and ontologies in spite of variations in terminology.

Theoretical bases and literature review

The history of computer engineering shows a slow but steady climb upwards from the base metal of binary bits to assembly code to higher level languages, all for the purpose of enabling humans to direct the operations of the machines to solve our problems. We have not yet reached the day when we can describe our needs in human languages, but many attempts are made to delineate new communication protocols for the implementation of complex computations. With the Internet growing more massive with every second and more human needs touched by technological requirements every day, we definitely need to develop ever more sophisticated ways to manage this data.

Existing research that attempts to ease specification of information systems include several examples that approach the problem by delineating new XML based modeling languages, and in more recent work, RDF and OWL languages. In [Ortiz-Comejo, et. al] they describe their use of WSML (Web Site Modeling Language) and WAML (Web Application Markup Language) which declare properties of the various components and tasks required to implement a web information system, from which code is generated for a functional system.

Investigations into ways to enable the Semantic Web, which envisions that promise of a web where the machines can do more work for us, have mostly covered the ground of pre-defining ontologies of terminologies and types, in the hope that all information and search needs can be anticipated and foundations and templates laid down ahead of time. As it often turns out, peoples' needs shift and they are not fully aware of what they need until they are in the midst of a project. Also the process of crafting ontologies is an intensive, expensive and exclusive operation with gatekeepers basically controlling the establishment of types. Thus users at large do not have enough of a voice in defining what is important to be included in our conceptual systems.
Some approaches examine how ideas like tagging might aid in annotating data types and improve search. In [Andreas Heß, Christian Maaß and Francis Dierick] they discuss the need for automating and aiding the bottom up approach to data annotation using machine learning techniques so that consistent tagging systems are established. Others have attempted to create resources such as meta-metadata wrapper types which are coded by developers as tools to enable other users to take in and mash up myriad external data sources. Once wrapped, they present predictable and unified interfaces. These techniques serve as a band aid to shore up the creation of patchwork systems that are somewhat searchable and manageable, but they do not look ahead enough to how we might really ensure semantic meaning beyond one off tag hits or accepting the existing metadata provided by information sources.

We would like to create our system so that structural information is an inherent and continually tracking part of data as it is input into the system. Properties can be attached to many different aspects of the graph database to aid in future analysis of the relationships between types and track how they evolve and settle into 'canonical types' over time rather than trying to guarantee understanding of them up front. Even if such settling may be discovered, our system should continue to be flexible and adaptable to variations in peoples' needs and use cases. With more meta-structural information carried along, we may someday be able to recognize and grade probabilistic matches in data types to give a little more, albeit fuzzy, information than is explicitly declared.

Goals

Our goal is to design something that can adapt to whatever data the user provides. The existing systems requires the users to specify what kind of data they want to store in a database. Based on that, the database schema is designed. Whenever a user wants to add additional fields to the data, the schema of the database has to be updated to store the new format manually.

The advantage of our approach is that it lets users not specify what type of data they want to store. They just enter the data and ignore the other details. This is possible by making changes to the database schema on the fly every time a new type of data is encountered automatically.

The disadvantage is that there will not be any strict ordering of data. In traditional relational database systems, tables are used to describe the schema and data is stored in the tables created for that data. Since we make changes to the schema automatically, the data in our approach may not be organized. For example, if a user wants to store a list of movies she wants to watch. She enters the movie details and stores the data. The next time she enters
some other movie details and forgets to add a particular field or adds an extra field. Now, the
schema may be updated to store this new kind of data or it is not stored in the place where
the previous movies were stored.

The other hypothesis is to take the best of both worlds i.e, traditional RDBS and the approach
we are taking. Since the core problem we are solving is updating the schema automatically,
we can create some rules to identify the type of data to be stored. For example, if the user
stores a list of movies, based on the key name like actors etc., the system identifies it as a
movie. The same thing applies for books and music by identifying the keys like author,
composer, tracklist etc.

Built into our tool will be rules for organizing the data, parsing the incoming data, and
identifying the type of data. It will then store the data in the correct place just like traditional
relational database tables. This approach helps also in searching the data very easily.

Methodology

The flexibility to change the database at any time without manual work specific to the required
change is achieved by a) using a graph database instead of a relational database and b)
generating code to make the appropriate modifications. Modifications to the graph include
adding or removing nodes, relationships, properties or labels.

Front End Interface
In keeping with the goal of developing tools for laypeople, we will be developing a user
interface for the user to manage their database and submit new data.

The creation and modification of the database will be done by example, no formal schema or
abstracted description is required. The user will also be able to view a representation of the
current state of their database.

The records created by the user will be stored in JSON format and handed off to the back end
for processing.

Back End Logic
Because JSON allows types and nesting, we need a parser to generate the right code for a
given type, and a given situation.

The back end will have a JSON scanner and parser which process the incoming data and
builds a parse tree. The scanning, parsing and tree building will be done using the compiler
generator ANTLR. ANTLR uses a parsing technology developed by Terence Parr (the creator of ANTLR) called Adaptive LL(*). It is a recursive descent parser with the added ability to parse non-left recursive grammar (Parr 2014).

The parse tree contains the data submitted by the user, and will be used to translate the data into code that updates the database. The code generation will be implemented by extending the ANTLR generated tree walking mechanism called a listener, which is implemented in Java. Visiting nodes of the tree triggers the actions we specified in the listener. For example, upon encountering a certain key value pair, we may want to generate a node in the database, and so we determine the appropriate query given the data.

The actions we want to take to modify the database require knowledge of the state and structure of the database. For example, if a user submits a record in which all the corresponding node types already exist in the graph, except for one key value pair, the code generated should create that node and do nothing for the rest.

The structure of the data dictates the structure of the resulting graph, therefore we can create graphs by example instead of by a formal schema. Neo4j graphs have nodes which can have labels and properties. Graphs also have relationships which organize the nodes. JSON may consist of an object with string value pairs, and/or arrays which are lists of values. The value can be a string, number, object, array, true, false or null. Our application will provide the appropriate mapping, as well as the appropriate actions to take given the particular data submitted.
Implementation

The project requires the following packages installed on a linux machine:
- openjdk or Oracle JDK
- ANTLR
- Neo4j server
- PHP

Setting the classpath for ANTLR and generating parser for JSON files:

$ export CLASSPATH=".:./antlr-4.5-complete.jar"
$ java -jar ./antlr-4.5-complete.jar JSON.g4

ANTLR generated the Parser and Lexer for JSON grammar declared in JSON.g4.

$ javac *.java

The above command compiles all the files created by ANTLR.

The Source code is organized as follows. It is written in Java

Source Folder:
- coen259/antlr - Contains Grammar generation code
- coen259/antlr/processingIncomingJson.sh - Generates new grammar by parsing the JSON file given by the user.
- coen259/neo4j - Contains the neo4j server jar files
- coen259/www - Contains the php code and HTML files for the users to enter data.

Grammar Generation Implementation:

The grammar generation for new json objects is implemented in Grammargen.java

The java code to insert data into the database is present in QueryGen.java

The following are the steps involved:

- The json file is given as an argument to the shell script file. It generates a new grammar for it and uses ANTLR to generate the Lexer and Parser for the file.
- If a new type of data comes, it generates the grammar for it and compares with the existing ones to check if it is a data type already stored in the database or a new kind of data.
- It returns valid if a grammar exists for the data, otherwise it stores the new grammar generated and keeps track of it from the next time.
Flow chart:

1. Input
2. Grammar gen
3. Grammar exists?
   - Yes: Updates fields in DB
   - No: Store data
Data analysis and discussion

Output generation:

Store data
By using **QueryGen**, the code to store the data is generated. It takes the json file we need to compile as the first argument and the name of the java file to save the generated code to store the data.

$ java QueryGen movies.json MovieClass.java

Generate grammar for new data objects
The following commands need to be run only for the first time.

$ chmod a+x processIncomingJson.sh
$ java -jar ./antlr-4.5-complete.jar ALTJSON.g4
$ javac ALTJSON*.java
$ javac NewJsonTool.java
$ javac GrammarGen.java

The shell script **processIncomingJson** generates a new grammar for the incoming data objects using ANTLR. It takes a new json file as an argument.

$ ./processIncomingJson movies.json
If the json has never been seen before, it will create a new dir called book-gen

You can create new json files that have a different filename but same value for the first pair (ie book).

```
$ ./processIncomingJson BadMovie.json
```

If it doesn't have the same fields etc as the first movie, it will be found to be invalid. If it does, it will be found valid.

```
$ ./processIncomingJson GoodMovie.json
```

We can open movies.g4 to check the grammar generated by the script. This grammar can be used to compare if the data object already exists or it is something new. If it is something new, the data type is stored and recognized from the next time.

The compiling of JSON files helps users not specify what type of data to be stored. This is possible by generating grammar for various kinds of data objects and comparing with the existing ones to see if the data is a mere update to existing data or a new kind of data. For example, if the user stores a list of movies, based on the key name like actors etc., the system identifies it as a movie. The same thing applies for books and music by identifying the keys like author, composer, tracklist etc.

In the abnormal cases, this approach may take a lot of time to store data into the database when compared to the current approaches. This is because, we need to check what kind of data to be inserted by generating the grammar and then comparing it with the existing ones. Even for updating the existing data, it may take a lot of time which is not acceptable for common end users. Also, it becomes very difficult to track if the data is overwritten by new data because the new data is not organized and if there are any typos made by the end users, the data is stored as new data. The grammar generated for various data types must be stored in an external server and backups have to be taken very frequently. Otherwise, if the grammar cannot be fetched, the programs treat it as a new data object and stores it in the database. This creates a lot of issues like validity of the grammar i.e which one is the correct type of grammar and which one is incorrect.

**Conclusion**

We conclude that dynamically creating APIs for new and modified data objects by compiling JSON fastens application development process. This approach also eliminates the redundancy of writing JSON parsing code in various languages especially when the application is being developed for various platforms.
We have seen that it works only when we use Neo4j database to store the data. We can check if we can somehow try other traditional RDBMS databases like MySQL etc., in the future. However, our approach can be used to write new data to any kind of database with little modification.

Bibliography


Appendices

The Grammar generation flow chart
The source code with documentation is hosted on a Bitbucket repository [https://bitbucket.org/nayandeepv/coen259/src/1cb1de81fb8c414f2d1760dfa7400ab1fc241a51/antlr/?at=master](https://bitbucket.org/nayandeepv/coen259/src/1cb1de81fb8c414f2d1760dfa7400ab1fc241a51/antlr/?at=master)