Towards a Modeling Method for Managing Node.js Projects and Dependencies

Bianca Lixandru
Babeş-Bolyai University
Cluj-Napoca, Romania *biancastefanialixandru@yahoo.com*

Robert Andrei Buchmann
Babeş-Bolyai University
Cluj-Napoca, Romania *robert.buchmann@ubbcluj.ro*

Ana-Maria Ghiran
Babeş-Bolyai University
Cluj-Napoca, Romania *ana.ghiran@ubbcluj.ro*

Abstract

This paper proposes a domain-specific and technology-specific modeling method for managing Node.js projects. It addresses the challenge of managing dependencies in the NPM and REST ecosystems, while also providing a specialized workflow model type as a process-centric view on a software project. With the continuous growth of the Node.js environment, managing complex projects that use this technology can be chaotic, especially when it comes to planning dependencies and module integration. The deprecation of a module can lead to serious crisis regarding the projects where that module was used; consequently, traceability of deprecation propagation becomes a key requirements in Node.js project management. The modeling method introduced in this paper provides a diagrammatic solution to managing module and API dependencies in a Node.js project. It is deployed as a modeling tool that can also generate REST API documentation and Node.js project configuration files that can be executed to install the graphically designed dependencies.

Keywords: Modeling Method, Domain-Specific Modeling Languages, Node.js projects, NPM ecosystem, REST ecosystem, Dependency Management

1. Introduction

In our local IT outsourcing industry, Node.js gained significant popularity – this paper's authors are directly involved or collaborate with such companies and have observed a challenge in managing Node.js dependencies, considering the complexity of the NPM or REST API ecosystems and the volatility of respective dependencies. NPM modules as well as REST APIs continuously evolve, or become deprecated – see the recent case of the deprecation for the HTTP requests module that have been used by countless dependent modules in the NPM ecosystem [23]. Auditing NPM dependencies across a rich portfolio of projects handled by a company becomes a significant challenge - this also being one of the reasons for the recent self-criticism of Node.js's creators, which led to the introduction of Deno [8] as an alternative ecosystem offering built-in dependency auditing tools.

Node.js projects are confronted with design-related issues, which result in a high rate of improvements and updates patched after the initial implementation of a functionality. In some cases, it even leads to problems at the projects' configuration level. Furthermore, even if the project structure and all configuration files are written by software developers, sometimes input from project owners, clients, managers, and other non-technical personnel is needed. In these situations, the team needs an easily understandable platform that enables everyone to visualize and discuss project structure and development processes.

This paper proposes a technology-specific modeling method that can help manage the complexity of the NPM and REST ecosystems for Node projects. The resulting functional artifact consists of a modeling language, which was built using Design Science [26] and Agile Modeling Method Engineering (AMME) [14] principles. The method in question provides a diagrammatic representation of concepts and processes surrounding Node.js projects' development, and helps developers, and non-technical stakeholders, to understand project structures and audit their dependencies, while also being able to generate project configuration files ready to install those dependencies.

The remainder of the paper is structured as follows: first, in Section 2 we describe the problem context and the motivation for the proposed method; then in Section 3 we provide a high level overview of the proposed solution and the requirements it addresses; Section 4 details design decisions and describes running examples; Section 5 discusses evaluation based on application in real cases, Section 6 references related works, before ending the paper with a brief concluding section.

2. Background Information

This section provides a motivation regarding the adoption of model-driven engineering paradigm as a solution for managing the complexity of Node.js projects.

2.1. Node.js and its Ecosystems

Node.js is an asynchronous JavaScript runtime environment that runs on the server and can be used as a back-end language for Web applications instead of more traditional approaches like PHP. Node.js encourages the use of a single programming language, JavaScript, to unify development for Web applications, both in servers' and clients' source code. Comparisons between the efficiency of Node.js and PHP or Python show that Node.js can handle more requests in a certain time making it "best choice for I/O intensive websites among the three, being significantly more lightweight and efficient" [15].

Projects using Node.js contain modules and packages, to allow an easier integration with available components and libraries. All packages are managed and organized by the Node Package Manager (NPM), which oversees the downloading and installing of Node.js packages from an NPM registry, handling all the third-party Node.js modules available publicly and privately. Usually, the packages are installed within the Node.js project using the NPM command line, directly by the developers. After the installation process is done, the package becomes a dependency in the project and is added to a dedicated configuration file named package.json. All the modules are stored and categorized within the specific folder node_modules [18].

Every package comes with its own set of dependencies, forming a dependency tree of modules. Because packages can depend on multiple other packages, the NPM can work with a range of versions required for each of them and can even compute the specific version needed to satisfy all requirements. This can lead to complex dependency paths that must keep up with updates and make organizing projects difficult, especially for the larger ones. Many times, unnecessary packages are left in the configuration of the projects, and forgotten about, or outdated modules are impossible to detect and update. All these make the NPM ecosystem hard to work with, unless other processes are set in place to manage its complexity. This is also one reason for which the Node.js creator introduced Deno [8], after rethinking several features pertaining to the management of Node.js projects, including how their dependencies are audited and managed – however numerous legacy projects are still running on Node.js and are not willing to migrate in the near future to Deno, which requires a major overhaul.

In addition, another technology-independent ingredient also raises the challenge of dependency traceability – REST APIs have been in vogue for some time and nowadays tend to be key ingredients in the back-ends of Web applications, including those developed in Node.js, with the help of dedicated frameworks such as Express.js [10]. In complex Web applications that are architected according to a certain (micro)service granularity, REST

APIs may also be part of chains of dependencies.

2.2. Model-Driven Engineering

Model-driven engineering (MDE) [20] offered a promising approach to design and optimize software development while raising the abstraction level and alleviating complexity for those involved in software projects. As previously discussed, Node.js projects and the NPM ecosystem can get quite complex. An efficient management process for Node.js projects can be achieved by applying the MDE principles. A team can work on designing the modules integration needed in the project by using diagrammatic means and a limited set of abstract constructs, which is the proposal of this paper. Project members or architects are empowered to create the map of the application and to discover dependency issues before the actual implementation and can ultimately use all the models, diagrams, and views of the project as *active documentation* for new stakeholders or even clients. The MDE approach can also be used for generating the documentation for the APIs used in the project and for generating NPM configuration files specific for Node.js projects. However, traditional MDE did not address the Node platform, nor the NPM ecosystem – so we needed to adopt domain-specific metamodeling to tailor a modeling language, tool and code generation from it (with "code" being limited here to configuration files and machine-readable documentation of dependencies).

In domain-specific modeling, real-world or planned objects and properties are translated into diagrammatic models. Models are, therefore, abstractions having only their most relevant properties and characteristics encapsulated within [13]. As stated in [21], conceptual models "are mediators between the application world and the implementation or system world" and are designed with a specific purpose. According to [16], models are used for varied reasons, from perception support, explanation or demonstration to simulation or construction. Multiple models of the same system can be different, governed by different layers of specificity dictated either by purpose, application domain or technological environment [4]. Even though only the most relevant properties of objects and systems are captured within a conceptual model, such models have some common properties [21] – e.g. the mapping property (every model is a mapping of a real-world concept), the truncation property (some irrelevant properties of the origin are omitted in the model), the pragmatic property (models are used only by a target audience and for a specific period of time), the amplification property (the model can contain properties that are missing in the origin and are envisioned as an improvement), the distortion property (being built to improve the origin, the model is a transformed version of it), the idealization property (models have idealized properties of their respective origins), the purpose property (each model has a purpose), the carrier property (all models use languages, and because of this, every model is considered as expressive as the language it uses) and the added value property (models are created to bring an added value or utility to the modeler).

In the domain-specific paradigm, specificity is needed in order to achieve a level of detail and specialization determined by purpose. A simple definition for it is given in [24] "A Domain-Specific Language is simply a language that is optimized for a given class of problems, called a domain". DSLs sacrifice reusability and generality for the sake of productivity and expressivity relative to the targeted problem class; they also evolve faster, in response to evolving modeling requirements [4].

Because of the complex nature of Node projects and their dependency-driven ecosystems, to achieve the best results in properly modeling Node projects, we have developed a modeling DSL whose first-class citizens are NPM modules, REST APIs, dependencies, explicit development tasks and appropriate relations between them.

3. The Proposed Modeling Tool

Like in any software development project, Node.js developers must work together with project owners, managers, analysts, and other non-technical stakeholders to assure that the implementation of the project is within the scope of the requirements. This can meet impediments when a technical subject is not properly explained or understood by all parts

involved. General modeling approaches can be used to outline at large the architecture's lines and the development processes, but when it comes to the particularities of Node.js and microservice architecture, project dependencies create an intricate system of sub-dependent modules and libraries, with open-source components being somewhat volatile.

The NPM network does not provide an accurate overview of all the connections between the modules of a project, so it is difficult for development teams to trace and catalogue all the dependencies that cascade with a simple installation of a module. Furthermore, developers need to manually install and update all projects' dependencies, and they need to check and replace by hand deprecated packages. We have gained awareness by direct contact with Node.js projects in the local outsourcing industry focusing on legacy applications maintenance, therefore a Design Science approach was hereby taken to introduce a diagrammatic modeling method that keeps track of dependencies (both for NPM and REST microservices) in a portfolio of Node.js projects.

3.1. Solution Requirements

The main purpose of the proposed method and language is to support all stakeholders, regardless of their technical knowledge, providing a diagrammatic view on the development and management of Node.js projects. Specifically, it enables the description of the development process, project composition, API and NPM module integration; it was also required to automatically generate the detailed API documentation and the specific configuration files executed by the NPM environment in order to deploy a project.

3.2. Methodological Aspects

According to [26], the object of Design Science is to provide an artifact that brings an improvement for the targeted situation, and then to assess the performance of the specific artifact in a determined context. In the presented case, the artifact proposed is a modeling method for managing Node.js projects. Even if, theoretically, it can be applied in any development process in any company that works with Node.js, it was built based on the procedures of the Company Y where the main author works as a developer, the company being specialized in outsourcing software development. During the first stage of the project, the main problem identified was the chaotic organization of Node.js projects, especially of the NPM and REST service dependencies. As an outsourcing company, Company Y adopts Node.Js in most of its projects but the above-mentioned issues raise technical management obstacles. Therefore, a modeling method was proposed to integrate a dependency view on Node.js projects and a process view of their development.

Due to the nature of the proposed artifact, and the aim of developing a DSL for a particular flavour of Model-Driven Engineering, the AMME framework [14] was adopted - Agile Modeling Method Engineering (AMME) applies agile principles to the practice of modeling method engineering [3] resulting in evolving modeling tools that can be tailored to various problem classes. One key ingredient of AMME is the ADOxx meta modeling platform [1] and its native scripting language for processing model contents – in the case of this project, for generating machine-readable Node.js project configuration files and REST service documentations out of the diagrammatic designs.

3.3. Solution Overview

The proposed modeling method consists of three types of models: Development Workflow Model, Project Model and API Model.

- *The Workflow Model* (type) provides a process view, similar to BPMN but tailored for the taxonomy of tasks involved in any Node.js project;
- *The Project Model* (type) provides a structural view and a map of project NPM dependencies;
- *The API Model* (type) keeps track of the REST endpoint tree created for a project

With the help of ADOScript [2], technology-specific functionality was implemented to

generate both REST documentation and NPM configuration files for Node.js projects. Firstly, based on the API Model, modelers can generate detailed documentation for each endpoint used within a microservice-based project. Moreover, and most importantly, modelers can generate the entire configuration file, formatted exactly as used by the NPM environment to install and integrate dependencies. A bird's view architecture of the tool is provided in Figure 1.

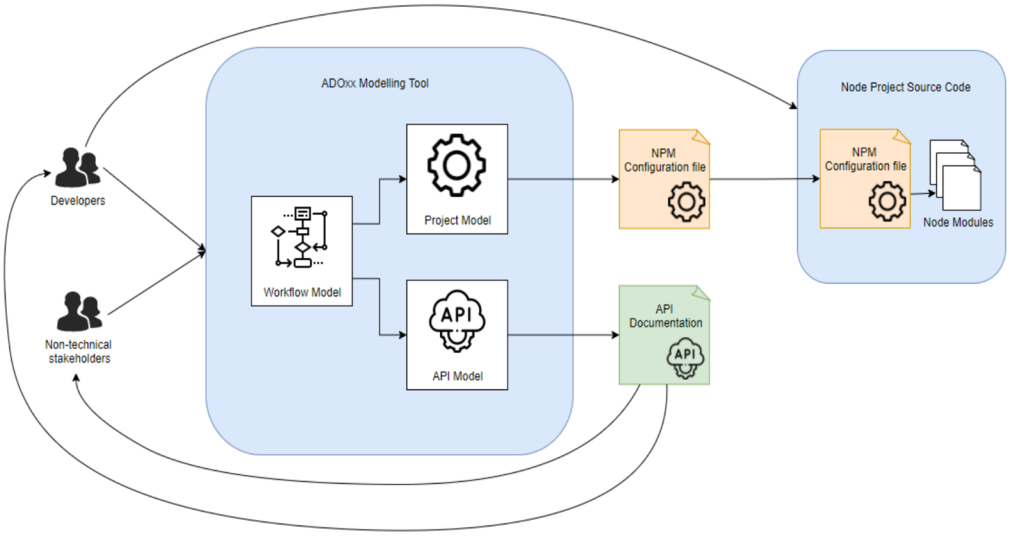

Fig. 1 Architectural view on the proposed artifact

4. Design Decisions

The modeling language is the main component of the Node project management modeling method. Its three types of models are governed by the metamodel shown in Figure 3, with the custom notation depicted in Figure 2. Relationships are depicted in Figure 3 as boxes (classes) with "from" and "to" attributes representing their domains and ranges, respectively. Relationships crossing between the three types of models are implemented as hyperlinks between diagrams of the different types. The concepts prefixed by __D__ are built-in constructs of the metamodeling platform that allow us to reuse some built-in properties (e.g. the "container" behavior, that generates a machine-readable relation between a container and contained elements).

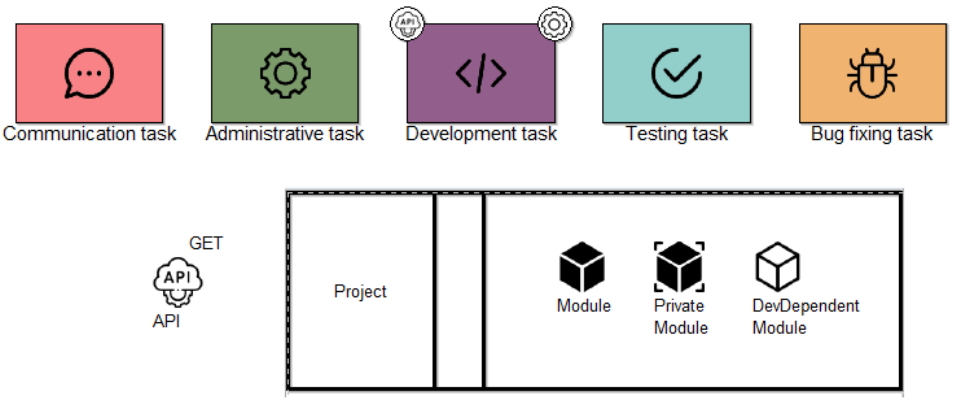

Fig. 2 Concept notations tailored for the modeling language

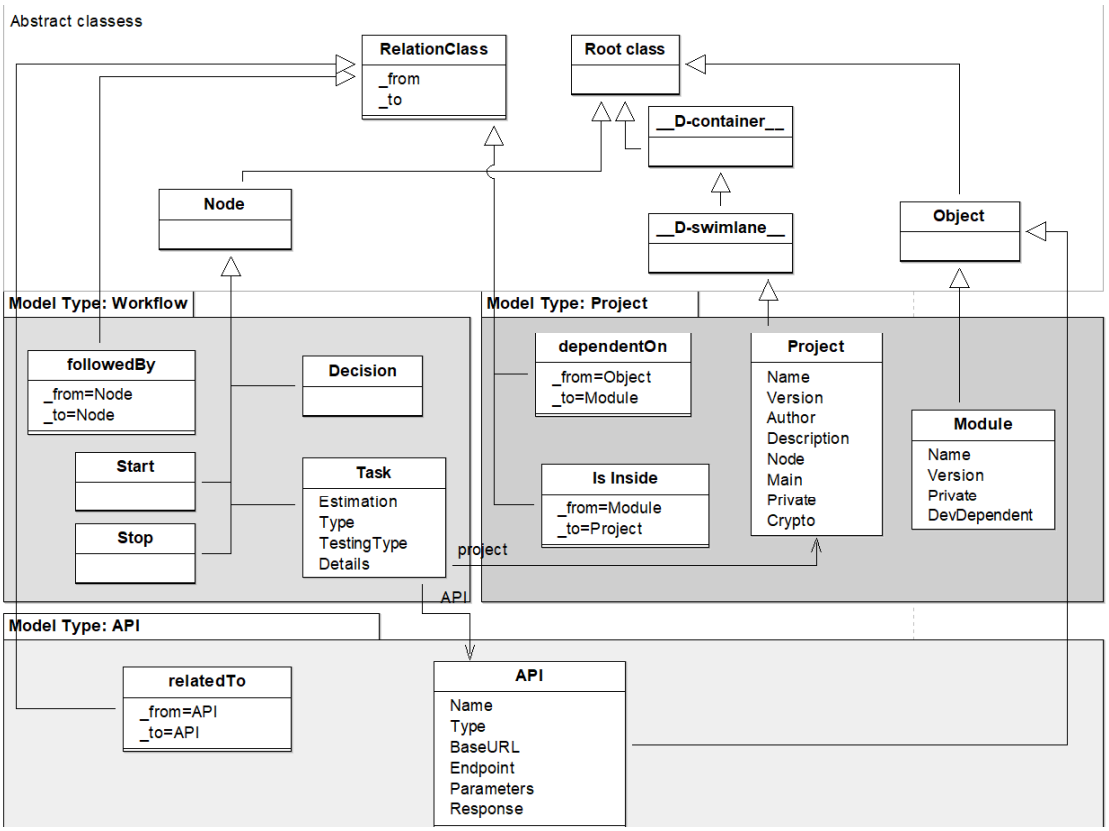

Fig. 3 The metamodel

The Workflow Model Type is used to diagrammatically represent the flow of decisions and tasks of a development process, in a simplified flowchart inspired by BPMN but adopting a taxonomy of tasks specific to the targeted field of activity. Additionally, domain-specific attributes can annotate and later serve to report generation based on model queries:

- *Estimation*, represents the predicted effort needed to fulfil the task;
- *Type*, with predefined values that categorize the types of tasks: Communication, Administrative, Development, Testing and Bug fixing. This attribute dictates the graphical variation among task types and also determines some personalized attributes for specific task types;
- *TestingType*, as its name suggests, is an enumeration that allows the selection of a more precise type for the Testing Tasks. The modeler can further specialize the Testing Tasks into Manual, Unit, Automation testing;
- *Details*, to attach more information for a particular task.
- *Project*, a hyperlink that connects the Workflow Model to a Project Model depicting dependencies.
- *API*, also a hyperlink, but references an API concept from the API Model; it is only active for Development Tasks.

The Project Model Type uses two main concepts: the Module class and the Project class. The Project acts as a visual container for a map of module dependencies, therefore generating a machine-readable containment that allows to distinguish between different projects described on the same canvas. Relevant attributes for a Project are:

- Basic metadata: *Name, Version*, *Author* and *Description*
- *Node*, stores the Node.js version
- *Main*, the project's main access file
- *Private*, to set the privacy of the project if desired.
- *Crypto*, a field to gather the browser encryption preferences.

The Module has similar metadata, and the possibility to indicate if it is a development dependency, i.e. ignored at runtime (it dynamically influences the notation).

The API Model Type is just a map of API endpoint dependencies, which must be managed in a (micro)service-based project. The API concept has all essential properties of APIs:

- *Name*
- *Type*, the HTTP method
- The URL components: *BaseURL, Endpoint, Parameters*
- *Response*, describes the data to be expected as a response.

The functionality for generating API documentation and configuration files for the installation of NPM dependencies was implemented in the native scripting language of ADOxx, ADOScript, see a sample in Figure 4 that parses annotated modules into the file that can be interpreted by NPM. Examples created for these types of models can be seen later on in the Evaluation section.

```
CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"Module"
CC "Core" GET_CLASS_ID classname:"Module"
CC "Core" GET_ALL_NB_ATTRS classid:(classid)
SET dependencies:("  \"dependencies\": {\n")
SET devDependencies:("  \"devDependencies\": {\n")
SET first:"true"
SET firstd:"true"
SET version:""

FOR i in:(objids)
{
    CC "Core" GET_OBJ_NAME objid:(VAL i)
    FOR j in:(attrids)
    {
        CC "Core" GET_ATTR_NAME attrid:(VAL j)
        CC "Core" GET_ATTR_VAL objid:(VAL i) attrid:(VAL j)
        IF (attrname="Version")
        { SET version:((val)) }
        IF (attrname="DevDependent")
        {
            IF ((val)="true")
            {
                IF (firstd="false")
                { SET devDependencies:(devDependencies+",\n") }
                ELSE
                { SET firstd:"false" }
                SET devDependencies:(devDependencies+"    \""+objname+"\": \""+version+"\"")
            }
            ELSE
            {
                IF (first="false")
                { SET dependencies:(dependencies+",\n") }
                ELSE
                { SET first:"false" }
                SET dependencies:(dependencies+"    \""+objname+"\": \""+version+"\"")
            }
        }
    }
}
SET dependencies:(dependencies+"\n  },\n")
SET devDependencies:(devDependencies+"\n  }\n")

SET message:(message+dependencies+devDependencies+"}\n")

CC "AdoScript" FWRITE file:"C:\\scripts\\package.json" text:(message) append:no
CC "AdoScript" INFOBOX "Project config file created in C:\\scripts\\package.json"
```

Fig. 4 Scripting sample for parsing NPM module map into the installable dependency file package.json

5. Evaluation

The context where problem awareness was raised is an outsourcing company specialized in developing and maintaining Node.js projects. The first approach to validating the proposed artifact was to apply its ADOxx-based proof of concept in real-world project development scenarios within the company. Its applicability was internally discussed,

concluding that in future iterations the configuration generation functionality should also be extended to other package management systems, e.g. Python's pip or PHP's composer.

The first project case was an extension to the Web version of Microsoft's email platform, Outlook. The requirements consisted of adding a new warning screen for displaying a user's phone number. This implied preparing the requirements, refining them, retrieving the data from the back-end application, developing the front-end part, and testing the functionality. Figures 5-8 depict concrete examples from this real-world project (e.g. extending the web version of Outlook, previously mentioned) where the modeling method was applied, creating all three types of models.

The development process for the above functionality started with the creation of the Workflow Model - a process view analogous to BPMN; however the team lacked BPMN experience and the new model is better tailored to the task taxonomy involved in software development projects. With the help of built-in effort estimation features inherited from the underlying ADOxx metamodeling platform, the project effort was better estimated and requirements changes were easier to track.

The process described in Figure 5 presents a sequence of tasks that are performed in the project. The visual representation of a task is dynamically changing based on the values in the Type attribute. In addition, as the development tasks can be linked to elements from the other two model types (i.e. Projects models or API models) they show visual cues/hyperlinks in the top corners - for example, the "Get phone number" task is linked to the API "Get phone number" from the API model and to the eslint-plugin-html from the "Outlook add-in Project" (see Figure 6 with the attributes of the task concept).

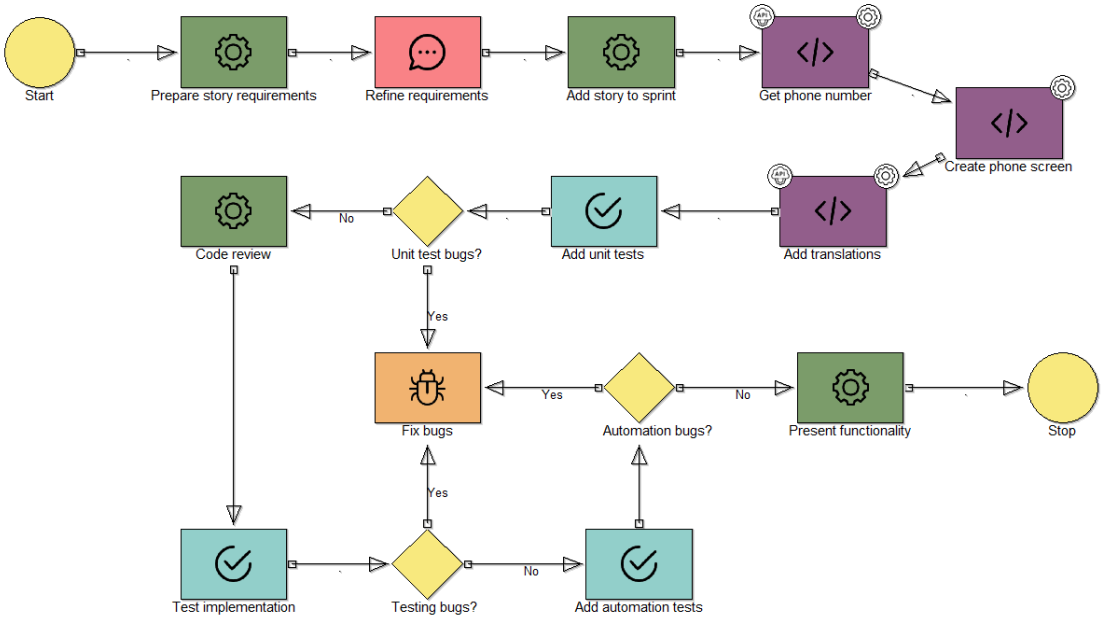

Fig. 5 Workflow of domain-specific tasks

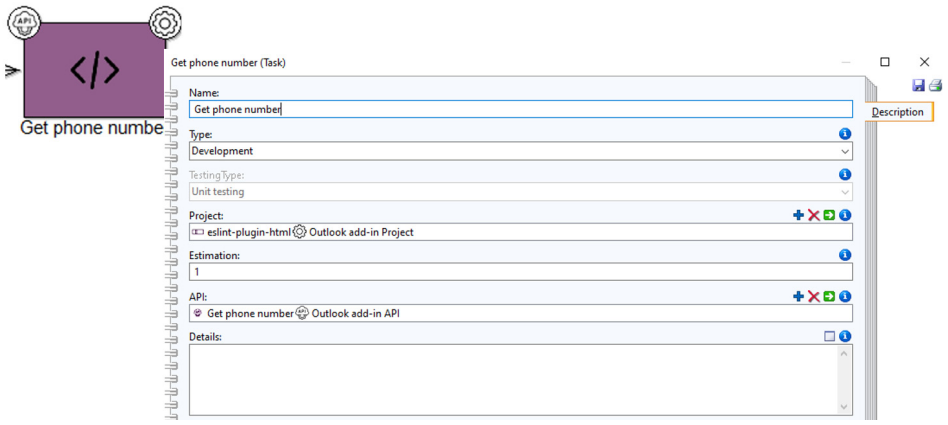

Fig. 6 Attributes for "Get phone number" task

The team also represented the network of APIs used in this project (Figure 7), with the help of the API model, and the generated documentation became part of the project's official documentation (the right side of the picture).

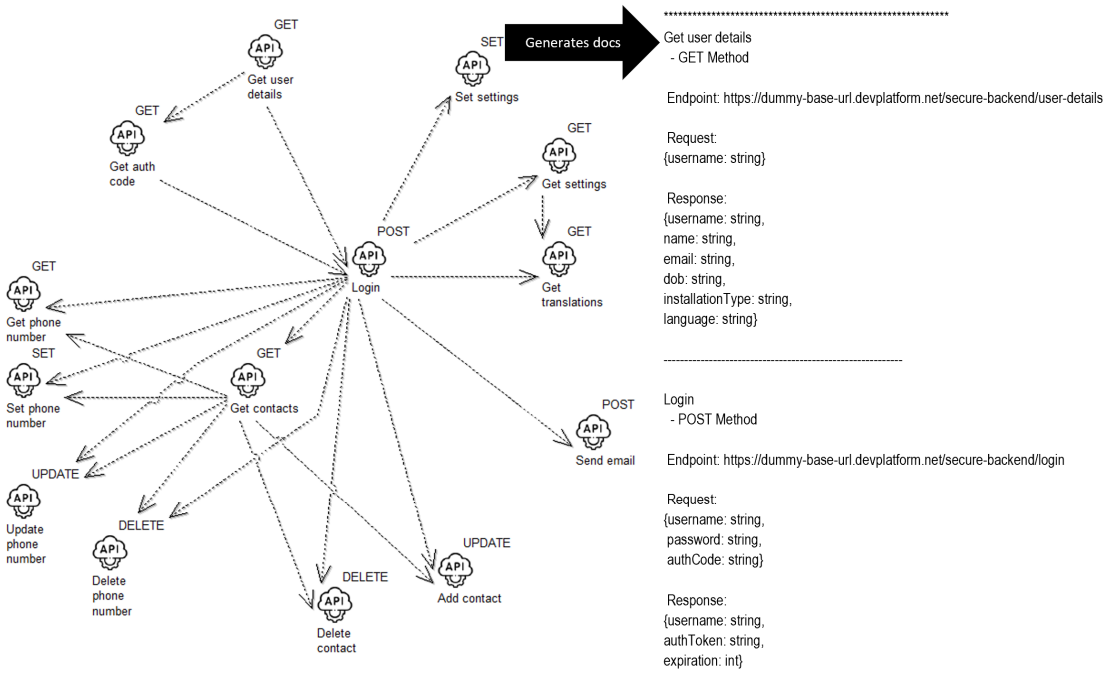

Fig. 7 The API ecosystem and generated documentation

The most well received model type was the one for Node projects (Figure 8), and their system of NPM modules – it makes it easy even for beginners to understand what a module is, where is it necessary and how it impacts other modules. It is also not limited to a design-time visualization, since it also generates the package.json configuration files (the right side of the picture).

Other projects where the proposed modeling tool was applied only involved the Workflow and Project Model since they were maintenance projects for legacy systems with no REST microservice view needed.

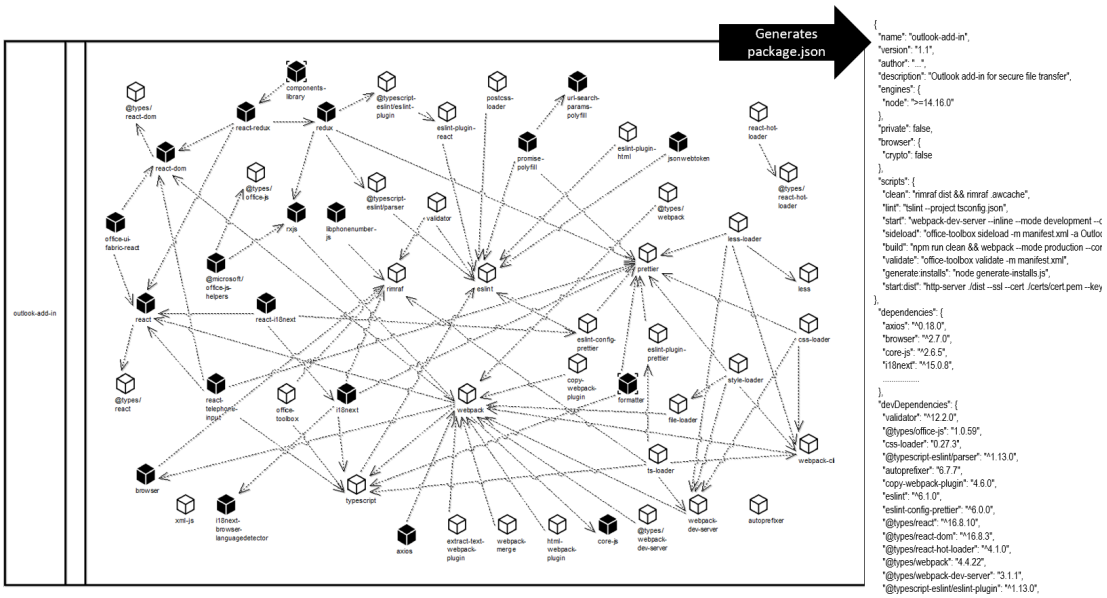

Fig. 8 The NPM dependencies and generated configuration file

During the development of the model-driven scripts, the prototypes for the generated configuration files were always compared to the real package.json files, to assure that the sections, content, and formatting were accurate. It was noted that the following differences and limitations can appear:

- The generation works only one-way (model-to-JSON, model-to-documentation), the roundtrip approach being left for future iterations of the DSR cycle;
- When installed manually, modules are stored alphabetically in the package.json file, for simplicity. The generated file will contain the modules listed in the order of their creation. This misalignment in the order of the modules is insignificant;
- Different development teams have different formatting standards when it comes to indentation, empty lines, spaces before or after operators and punctuation marks. The mechanisms of the proposed modeling method generate the configuration file based on the preferences set within one selected development team in the company;
- Diagrammatic labels are not validated against the NPM ecosystem therefore any spelling errors can lead to a different package being installed.

6. Related Work

Reference [7] provided an early theorizing on a particular flavor of domain-specific modeling, labelled as "technology-specific modeling" – where the concepts of a modeling language are not specific to an application domain, but to a technological environment. Code generation has been a traditional goal of UML-based modeling, but technology-specific modeling can be tailored with the help of metamodeling for any kind of interfacing and technological context. The authors of [21] demonstrate this with a language for the design of ETL processes that are tailored for the in-house ETL engine of a company, having no reusability outside that company but bringing a certain productivity increase due to its tight coupling with the ETL engine.

A similar mindset motivated the current work, however it aims for wider reusability since the Node.js ecosystem is not company-specific but reasonably popular in Web development – the issue of managing dependencies in this ecosystem has been recognized for a long time in the community, with dependency auditing being discussed as a key challenge. We take the path of technology-specific modeling to also achieve a visual layer of a project portfolio for a Node.js-focused company, with a "code generation" approach that does not aim to produce running code, but instead machine-readable configuration files that define the dependency map for any Node project.

Also employing the AMME framework and ADOxx as an implementation platform, [6] presented a modeling tool for visually building Resource Description Framework (RDF) data graphs – it shares with the current work tooling and methodology but it aims for another kind of technology-specific alignment.

The work presented in [17] showcases an intelligent software development platform that enables the design of AI applications with sub-systems for “knowledge-based reasoning, spoken dialogue, image sensing, motion management, and machine learning”, all combined into a workflow editor that enables users to modify in real-time the behavior for multiple robots and sensors, thus bringing the "technology-specific" aspect to the area of cyber-physical systems.

Reference [5] introduced a novel modeling method as an extension to the BPMN standard, facilitating the execution of HTTP requests from diagrammatic business process models, while also ensuring the business process-centric view a business analyst would expect from a diagrammatic model tool. Also, authors in [9] present a model-based approach for generation of RESTful APIs that enables collaboration among the development team members as it is designed as a web application. In our work, the process-centric view is tailored for the task types encountered and gleaned from personal experience in the company that inspired the hereby proposal. Further into the API management field, the work done in [12] introduces RESTalk, a tool that enables a more accurate representation of the client-server interactions. Even if, unlike the previous referenced work, this paper only extends the choreography diagram of the BPMN standard and does employ the AMME methodology, it also facilitates a new layer of abstraction for defining RESTful APIs, something that has been pursued for a long time in service-driven software architects, see also the emergence of the Swagger framework [22].

7. Conclusions and Future Works

To the best of our knowledge, this is the first paper proposing a modeling language specific for Node.js projects and for the NPM ecosystem. The proposed artifact can be used by non-technical stakeholders of a Node.js based project to better organize the development procedures and tasks, and by specialized developers to integrate project details into development processes, to manage dependencies and even generate auxiliary files needed along project execution in different stages. Future plans aim to cover more diverse technologies - the pool of technologies can include most modular development environments, the authors also being familiar with at least Python's and PHP's package ecosystems.

Acknowledgements

Further technical and theoretical aspects of the work are detailed in the master dissertation of B. S. Lixandru titled *A modeling method for managing Node projects and the complex ecosystem of NPM*, defended at University Babeș-Bolyai.

References

1. BOC GmbH (2022) The ADOxx metamodeling platform Available at: https://www.adoxx.org. Accessed March 21, 2022
2. BOC GmbH The AdoScript Programming Language. (2022) Available at: https://www.adoxx.org/live/adoscript-language-constructs Accessed March 17, 2022
3. Buchmann, R. A., Karagiannis, D.: Agile modeling method engineering: lessons learned in the ComVantage research project. In IFIP Working Conference on The Practice of Enterprise Modeling 2015, pp. 356-373, Springer (2015)
4. Buchmann, R.A.: The Purpose-Specificity Framework for Domain-Specific Conceptual Modeling, in Domain-specific conceptual modeling: Concepts, Methods and ADOxx Tools, pp. 67-92, Springer (2022)
5. Chiș, A.: A Modeling Method for Model-Driven API Management, Complex Systems

Informatics and Modeling Quarterly, 25, 1-18 (2021)
6. Chiș-Rațiu, A., Buchmann, R.: "Design and Implementation of a Diagrammatic Tool for Creating RDF graphs", 2nd PrOse Workshop co-located with PoEM 2018, CEUR-WS, vol. 2238, pp. 37–48 (2018)
7. Deme, A., Buchmann, R.A.: A Technology-Specific Modeling Method for Data ETL Processes. In: Proceedings of AMCIS 2021, paper 1128, Association for Information Systems (2021)
8. Deno official website, https://deno.land, Accessed on: March 16, 2022
9. Ed-Douibi, H., Izquierdo, J.L.C., Gómez, A., Tisi, M. Cabot, J.: EMF-REST: generation of RESTful APIs from models. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1446-1453, ACM (2016)
10. Express.js official website, https://expressjs.com/, Accessed March 16, 2022
11. Hota, A. K., Prabhu, D. M.: NODE.JS: Lightweight, Event driven I/O web development. Informatics, https://informatics.nic.in/article/287 (2014)
12. Ivanchikj, A., Pautasso, C., Schreier, S.: Visual modeling of RESTful conversations with RESTalk, Software and System Modeling, 17, 1031–1051 (2018)
13. Karagiannis, D., Buchmann, R.A., Burzynski, P., Reimer, U. and Walch, M.: Fundamental conceptual modeling languages in OMiLAB. In Domain-specific conceptual modeling, pp. 3-30, Springer (2016)
14. Karagiannis, D.: Agile modeling method engineering. In Proceedings of the 19th Panhellenic Conference on Informatics, pp. 5-10, ACM (2015)
15. Lei, K., Ma, Y., Tan, Z.: Performance comparison and evaluation of web development technologies in php, python, and node.js. In 2014 IEEE 17th international conference on computational science and engineering, pp. 661-668, IEEE CS (2014)
16. Liddle, S. W.: Model-driven software development. In Handbook of Conceptual Modeling, pp. 17-54, Springer, (2011)
17. Morita, T., Nakamura, K., Komatsushiro, H.: PRINTEPS: An Integrated Intelligent Application Development Platform based on Stream Reasoning and ROS, The Review of Socionetwork Strategies, 12, 71–96 (2018)
18. NPM Doc: Node Package Manager (npm) Documentation. (2022) Available at: https://docs.npmjs.com/about-npm. Accessed March 22, 2022
19. Peffers, K., Tuunanen, T., Rothenberger, M.A. and Chatterjee, S.: A design science research methodology for information systems research. Journal of management information systems, 24(3), 45-77 (2007)
20. Schmidt, D. C.: Model-driven engineering. Computer 39 (2), 25-31 (2006)
21. Thalheim, B.: Syntax, semantics and pragmatics of conceptual modeling. In International Conference on Application of Natural Language to Information Systems, pp. 1-10, Springer (2012)
22. The SwaggerHub API Development Tool, https://swagger.io/tools/swaggerhub/, Accessed on: March 16, 2022
23. Thung, S.: Request has been deprecated, https://betterprogramming.pub/request-has-been-deprecated-a76415b4910b, Accessed March 16, 2022
24. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Wachsmuth, G. H.: DSL engineering-designing, implementing and using domain-specific languages, CreateSpace (2013)
25. W3Schools Website, (2022) "Node.Js Tutorial: Intro". Available at: https://www.w3schools.com/nodejs/nodejs_intro.asp. Accessed March 16, 2022
26. Wieringa, R. J.: Design science methodology for information systems and software engineering, Springer (2014)