Information Technology of Untyped Information Processing

Maksym Konovaliuk[1] and Oleg Dmytrenko[1,2]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
[2] Institute for Information Recording of National Academy of Sciences of Ukraine, Kyiv, Ukraine

Abstract

The article presents a toolkit that contains the functionality of universal CRUD operations. The presented solution can be applied to most cases that require the basic functionality of CRUD operations. This solution can be applied to save the time of back-end developers when performing routine work of the same type. Front-end developers can independently configure the necessary data model and work with the developed universal CRUD. Current investigative results in the field of universal APIs and CRUD operations were considered.

Keywords

Create Read Update Delete (CRUD), Mongo, MongoTemplate, Spring

1. Introduction

The advancement of information technologies has largely been fueled by the need to transition document workflows from paper-based to electronic formats. Popular programming languages were initially developed with the goal of seamlessly transferring information between different forms, primarily from presentation to storage formats and vice versa. Object-oriented programming languages represent a program as a collection of interacting class instances. Some classes are designed to encapsulate information, referred to as information classes, while others are tailored for processing instances of these information classes. In the process of software development, developers are frequently tasked with processing information stored in databases. This involves implementing operations for saving, updating, and retrieving information from the database, commonly known as Create Read Update Delete (CRUD) operations. The technologies used to implement CRUD operations are continuously evolving through abstraction to higher levels and automation of technical details. Modern libraries automate a significant portion of technical complexities, enabling developers to concentrate on implementing the application's business logic. In most cases, implementing CRUD operations does not require substantial effort from the software developer and primarily involves routine technical tasks. The functionality being implemented is typically not unique, with the main difference lying in the data model being operated on. Nonetheless, developing such functionality still demands developers' time. Hence, automating developers' work with the data model is a logical step forward.

2. Context

Consider current investigation results in the field of universal APIs and CRUD operations. Richardson and Amundsen's book presents the fundamental concepts of creating and applying RESTful APIs [1]. The book delves into the principles and optimal methods for creating RESTful APIs, emphasizing the significance of adhering to RESTful design principles. Richardson and

ITS-2023: Information Technologies and Security, November 30, 2023, Kyiv, Ukraine
konovalyuk@gmail.com (M. Konovaliuk); dmytrenko.o@gmail.com (O. Dmytrenko)
0000-0003-4601-3790 (M. Konovaliuk); 0000-0001-8501-5313 (O. Dmytrenko)

Amundsen introduce readers to various HTTP methods and functionalities, exploring how to leverage them to enhance API performance and security. Additionally, the book examines hypermedia and HATEOAS, showcasing how hypermedia links can enrich API discoverability, decouple clients from servers, and facilitate the evolution of APIs. Covering the entire API lifecycle from design and implementation to versioning and documentation, the book provides practical guidance on versioning strategies, maintaining backward compatibility, and crafting effective API documentation. Throughout the text, Richardson and Amundsen present thematic case studies and real-world examples drawn from popular web APIs, illustrating best practices, common pitfalls, and insights gleaned from developing and managing large-scale RESTful API interfaces.

Martin Kleppmann's book [2] serves as a thorough guide that delves into the fundamental principles of constructing reliable, scalable, and maintainable systems for handling large volumes of data. Kleppmann explores various data storage and processing technologies, including both relational and non-relational databases, assessing their strengths and weaknesses while offering real-world examples of their implementation. The book covers topics such as data modeling, storage, and management in multi-threaded environments, alongside distributed transactions, stream processing, and batch data processing. Throughout the book, Kleppmann provides numerous examples, best practices, and common pitfalls encountered when designing systems that interact with big data. Robert Cecil Martin's book [3] is a guide to writing clean, maintainable, and high-quality code. It covers the fundamental principles of clean code, simplicity, and readability, with examples provided throughout. The book also discusses various practices and methods for improving code quality, including error handling, testing, and refactoring.

The articles by Yazdani Niloofar and Sanaz Malekizadeh [4], and Gjorgjioski Valentin [5] present academic research in the realm of automatically generating RESTful APIs. The authors discuss the dynamic creation of API interfaces. Thuraisingham Bhavani and Ashish Gupta's publication [6] delves into the flexible management of REST APIs utilizing resource models. Tsirulnikov and Smith [7], in their paper, examine the automated generation of RESTful APIs based on relational database schemas. The study presented by Yu, Ziyu [8] delves into the creation of a universal RESTful API server based on hypermedia principles. The author explores the design and implementation aspects of a RESTful API server aligned with hypermedia principles. They discuss the significance of hypermedia in enhancing the flexibility, scalability, and adaptability of API server architecture. Additionally, practical scenarios and challenges encountered during the design and implementation of the universal RESTful API server are examined. The publication highlights the potential benefits of adopting hypermedia-driven approaches in developing RESTful APIs, including improved discoverability, reduced coupling between clients and servers, and simplified client integration with evolving API endpoints. Articles [9-15] describe the design and implementation of universal and extensible RESTful APIs. The paper [16] authored by Liu Yang introduces a fresh perspective on dynamically generating APIs for heterogeneous data sources using meta-modeling. Their approach offers a unique method enabling the automated creation of CRUD (Create, Read, Update, Delete) APIs catering to diverse data sources such as relational databases, NoSQL storage, file systems. A pivotal aspect of their proposal lies in leveraging meta-modeling to delineate the structure and interrelations of data across varied sources. They advocate for a methodological framework that describes data structures through meta-models, thereby abstracting away specific intricacies of individual data sources and facilitating a uniform access interface. Building upon this meta-modeling paradigm, the authors devise an algorithm for automatically generating APIs, streamlining the coding process for executing CRUD operations across disparate data sources. The paper substantiates its claims with practical examples, demonstrating the application of their method in crafting APIs for a spectrum of data types and sources. The investigation results of the universal CRUD are presented in academic articles [17-19]. A significant role is played by the technical capabilities of the tools whose descriptions can be found in the documentation [20-22].

There are different approaches and techniques in programming to reduce the amount of code. Class parametrization (generic classes) is one of them. A class can perform actions on variables whose type is specified as a parameter. In other words, the business logic is the same but the type of data on which these actions are carried out is different. The parametrization of classes mostly applies to the business logic classes. Classes designed to encapsulate information (information classes) are mostly used as the parameter. Such technique allows to reduce the amount of code responsible for business logic, but this often does not affect to the number of model classes (information classes).

When considering the use of parametrization (generics) in the context of project architecture, the utilization hinges on the chosen architectural approach. Consider two prevalent methodologies: microservices and monolithic architectures. Both paradigms typically encompass similar internal structures, predominantly consisting of layers (levels) – namely the persistence layer, the business layer, and the presentation layer. Parametrization commonly manifests during the implementation of the persistence layer, although its application is not exclusive to this layer alone. For instance, in the implementation of the Data Access Object (DAO) pattern, often employed in realizing the persistence layer, we observe linear dependencies from entity classes to corresponding DAO interfaces. Typically, each entity class describing the data model corresponds to one implementation class responsible for basic CRUD operations.

CRUD operations are the basic methods for data manipulation across most applications. Within the DAO pattern, these operations execute the basic tasks between the data model, depicted by entity classes, and the database. Generalization CRUD operations facilitates applying identical CRUD operations to various data models, rather than implementing separate sets of operations for each specific data model [Figure 1].

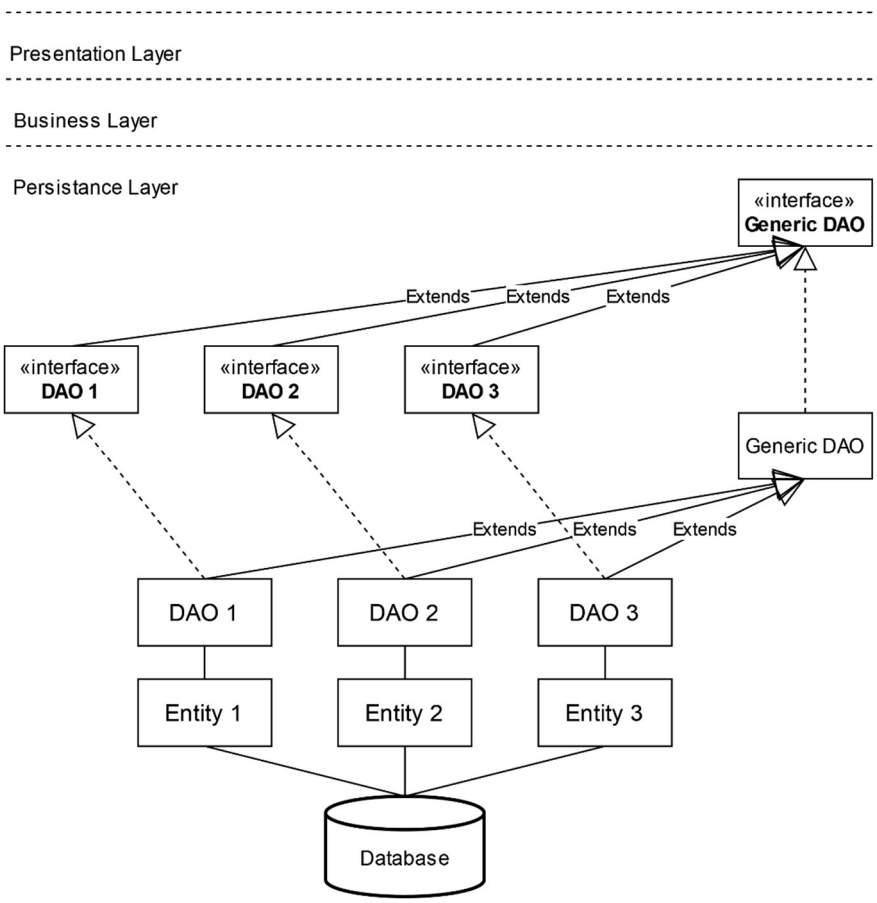

Figure 1: Generic Data Access Object schema on persistence layer

Considered scenario still remains essential to explicitly indicate the existing data models (entity classes), each of which should possess a clearly defined structure. A portion of developers' time is dedicated to creating and adjusting these data models. Essentially, all data models represent varying forms of information representation contingent on the domain, yet they do not influence the core actions conducted on this information.

There's a possibility to try lifting restrictions on parametrization. For instance, by generalization with the Object class (in Java), to allow executing basic generalized CRUD operations with any type (data model). However, in such a scenario, additional checks on the input type will be needed, and we'll encounter challenges in persisting information in the database when using ORM (object relational mapping).

Parametrization can help reduce overall code volume and accelerate implementation. However, an alternative approach is proposed where basic operations can be conducted without specifying any type, which should further abstract the implementation and eliminate the need for developers to create information classes (data models) for each specific case.

3. Purpose of the study

The aim of this research is to devise a solution that enables abstraction from the data model and develops a universal toolkit for carrying out CRUD operations at the API level. The proposed solution should be independent of the data being handled and capable of managing diverse information. In essence, the goal is to implement a universal CRUD that can be utilized across various data models, thus saving back-end developers from repetitive tasks.

4. Design

The development of most projects typically begins at from the data storage level. The majority of modern projects store the data they operate on in a database.

Let's delve into the realm of data storage, specifically databases. Generally, databases can be divided by the nature of information storage into two categories: relational and non-relational databases. Relational databases, unlike non-relational ones, store information in the form of tables. Storing data in a tabular format implies a clear structure. Each table consists of fixed columns, with names and types.

In contrast to relational databases, non-relational databases offer a more flexible approach to storing information. A lot depends on the technological solution of the non-relational database. Let's consider the widely popular solution MongoDB. MongoDB stores information in the form of a document, containing fields and values, and can be represented as a JSON document. In other words, MongoDB stores information in the form of a key-value data structure, where the field name is the key.

The way Mongo structures its data typically relies on the data model, which is defined by entity classes at the code level that interact with MongoDB. Let's explore how Mongo integrates with Spring. Utilizing the spring-data-mongodb library is one way to interact with MongoDB. An established method for Java's interaction with MongoDB involves utilizing "repository" interfaces and their corresponding entity classes, which rigorously define the data model. These repository interfaces conduct CRUD operations with the specified data model through parametrization. However, the spring-data-mongodb library offers a more adaptable approach to interacting with MongoDB through the methods of the MongoTemplate class. This class executes CRUD operations on an instance of any type passed as a parameter to the MongoTemplate methods — usually these are entity classes. Additionally, the spring-data-mongodb library encompasses a Document class, which can also be employed for methods in the MongoTemplate class. Differing from entity classes, this class can accommodate any fields and values as it inherits from the Map data structure. Consequently, there

exists a technical feasibility for record and store any information in the database in the form of key-value pairs using an instance of the Document class. Why not extend a similar capability to the API endpoint level? It's important to note that storing information with different structures in the same collection might present challenges for API users when handling this information. Moreover, varying structures and primary key types within the same collection could pose complications. Therefore, it's essential for information of uniform type to be stored within a single collection while also adhering to an arbitrary data model. The proposed approach involves leveraging the Document class's capability to save any information, creating a reference data model using auxiliary CRUD, and subsequently comparing input information with the reference model (Figure 2).

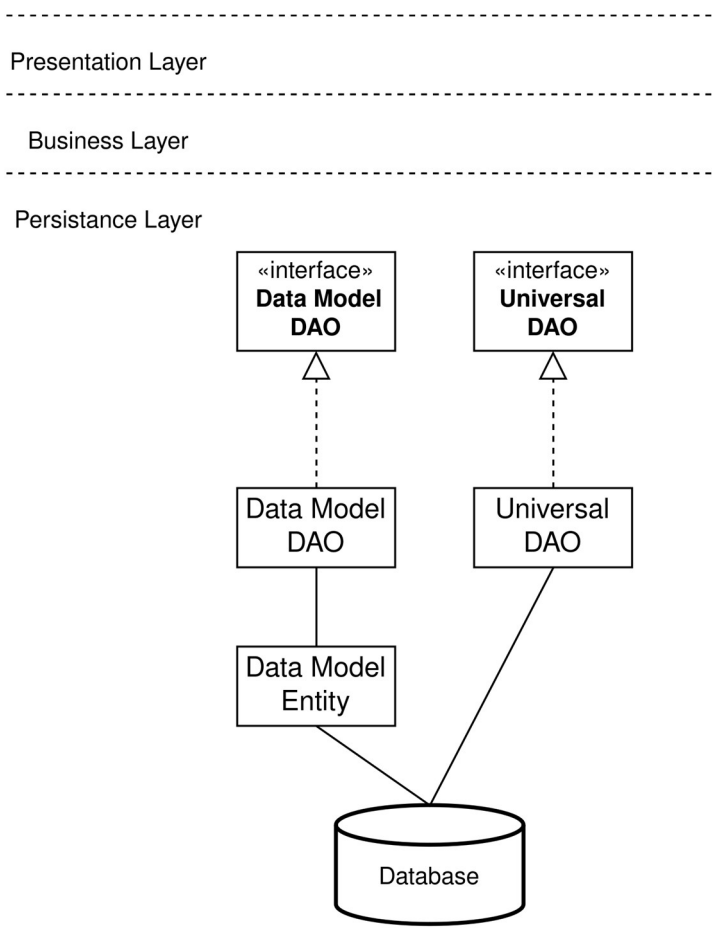

Figure 2: Proposal schema on persistence layer

5. Implementation

The implementation is built upon the following technologies: Spring Boot, MongoDB, and associated libraries, notably spring-boot-starter-data-mongodb. Let's delve into the implementation of the reference data model.

To streamline operations, a string field called "*model*" has been introduced. This field determines the configuration's name and must hold a unique, non-repeating value to streamline operations across the service. The reference data model must align with a specific collection in MongoDB. The "*collection*" field in the reference model specifies the collection on which actions will be executed. Collections in MongoDB are created and deleted alongside the creation and deletion of their corresponding data models. It's important to refrain from using an existing MongoDB collection when creating a reference model. The "*document*" field is an object and can assume any structure, such as numerous key-string value pairs or key-object combinations. This field aims to define the structure

of the primary CRUD operations. It's worth mentioning that in the previously discussed development process, the Document class, which extends the Map interface, was highlighted. MongoTemplate has the flexibility to accept variables of any data type as parameters. To minimize dependency on the MongoDB library implementation, employing a Map instead of a Document seems logical. Therefore, the data model for a reference model comprises a model name (*model*), a data collection (*collection*), and the reference model itself (*document*) in the form of a Map type field.

During the creation of the reference model, the following validations are conducted:

- Ensuring the availability of a reference model for the specified model name in the request.
- Ensuring the availability of a reference model for the specified collection name in the request.
- Verifying the existence in the database of a collection whose name matches the specified collection name in the request.

If any of these validations fail, the creation process for the reference model is halted.

Universal CRUD functionality has been developed using methods that accept the model's name and an instance of the Map interface, allowing the transmission of arbitrary information. The model parameter is essential to load the previously created current reference model by its name and verify the conformity between the incoming information and the reference model in terms of field presence and absence, and the correspondence between types of fields. Additionally, the loaded reference model, obtained through the model parameter, contains details about the collection's name for which CRUD operations must be performed. It's worth noting that the incoming data model may or may not include the *id* field. If the main universal CRUD request contains an *id*, then a check is performed to verify its existence. If such an *id* already exists, then an appropriate error message will be generated. The request to load a reference model for field comparison adds complexity to the processing of the main universal CRUD and increases execution time due to the additional database query. However, it is essential to maintain consistency between models within single collection. A basic implementation of data caching has been incorporated to expedite queries for retrieving information using universal CRUD. During data persistence using universal CRUD, besides saving to the database, data is also added to the cache. During read operation the information firstly searched in the cache. If the requested data is not found in the cache then it retrieved from the database and, if such data are available they returned as a response to the request and stored in the cache. The cache is capable of handling various data models, that is why all information stored within it is grouped according to the model.

Cache initialization with data from the database has been implemented during the service loading and initialization stage to ensure that initial requests are executed with the current cache.

It's important to take a closer look at the process of verifying the alignment between incoming data through the main universal CRUD and the reference data model. This validation process varies depending on whether it's a creation, update, or partial update operation. Since partial updates don't necessarily include all the fields present in the reference model, the approach needs to be different. For creation or full update operations, there's a check to ensure that all fields specified in the reference model are included in the request to the universal CRUD. If the number of fields is less or greater, the request execution will be halted, and an error message will be generated. If all field names match those in the reference model, the next step is to verify that the value types correspond to the types specified in the reference model. It's worth noting that the structure of the values of incoming information fields can vary. The value may not necessarily be a single-line value of one of the standard types, such as integer types, string types, date types, or boolean types. Field values can be nested instances, meaning that a field value can be an object consisting of multiple fields and their corresponding values. In such cases, a recursive check is performed to ensure that all nested fields and values match the reference model throughout the entire depth.

6. Application of the proposed solution

The practical application of the proposed approach can find utility across a broad spectrum of domain areas. It can be seamlessly integrated into virtually any domain for data model descriptions. Developers devote a substantial amount of time to tweaking data models. Requests for these modifications keep as the project evolves and is maintained. The suggested approach is particularly

pertinent in cases where frequent alterations to the data model are necessary, and these changes do not impact the application's core business logic. For instance, consider the intricacies of managing localization, specifically handling text translations displayed to users on various portal pages. There exist several methodologies for implementing localization functionalities in applications. Among them, the most prevalent and efficient entails the use of properties files. Translations are stored in these files using a key-value format. While this approach is reliable and effective, modifying translation texts within properties files demands a certain level of proficiency from users and may necessitate restarting the associated service. On occasions, developers are tasked with enabling translation edits for administrators, allowing them to tweak translation texts without interrupting specific services. One potential solution involves storing translations within a database. It's important to note that working with databases typically entails an understanding of the data model. However, in the scenario at hand, translations will be consistently appended. Storing translations as an array of values would entail iterating through said values on the Back-End, thereby increasing algorithmic complexity and potentially slowing down application performance. In the scenario under consideration, the proposed universal CRUD could prove advantageous. It enables storing translations in a key-value format, akin to how they're stored in properties files, without necessitating additional transformations during processing. Ultimately, this facilitates storing translations in the Mongo database as Json documents, allowing system administrators to modify the model without developer intervention. At the same time, due to the reference model, all translation documents will maintain an identical field list required for translations across different languages. Let's take a look into this scenario further through an illustrative example.

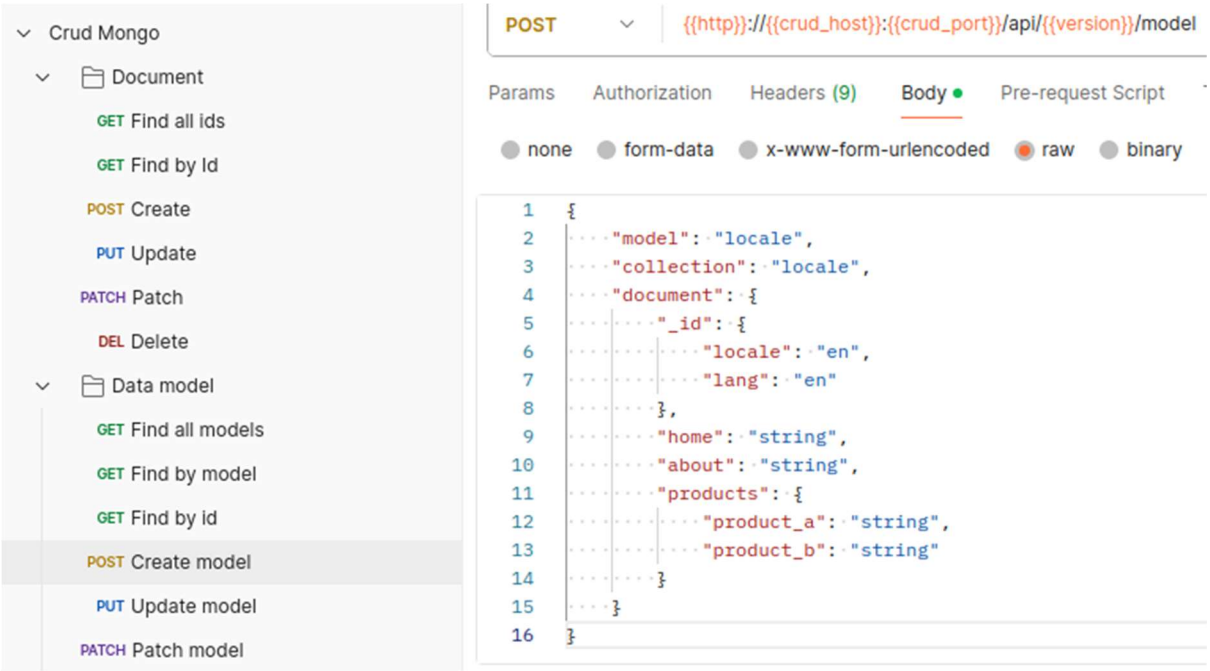

Figure 3: Screenshot of a data model in Postman

In the initial stage, we establish a reference model [Figure 3]. This process results in the creation of a collection within the Mongo database, where the reference model is stored under the "*model*" collection. Validation of the created model's outcome can be conducted using endpoints, employing either the generated ID or the specified model name within the request. Furthermore, there is the option to view a comprehensive list of all reference models. If any modifications to the established model will be necessary, one of the two available endpoints can be utilized. One endpoint facilitates complete updates to the reference model, while the other permits partial modifications. Should the need arise, the reference model can be deleted, consequently removing the associated data collection.

Following the establishment of the reference model, we can be directed towards utilizing the primary universal CRUD [Figure 4]. Throughout the operation of the primary universal CRUD, any data model corresponding to the reference model can be specified. Additional specification of the model parameter is required when engaging with the universal CRUD. The set of implemented operations for the universal CRUD adheres to industry standards, encompassing requests such as full and partial updates to recorded information.

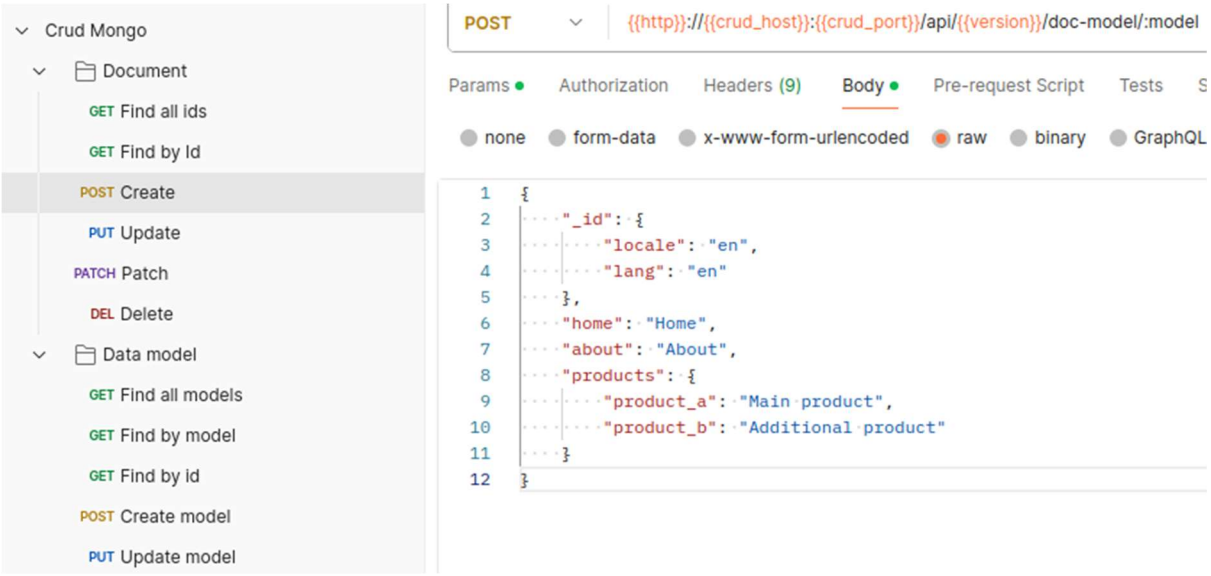

Figure 4: Screenshot of a universal model in Postman

7. Conclusions

This article presents the developed tool that incorporate the functionality of universal CRUD operations. The implemented solution lacks specific business logic, so it doesn't cover all possible scenarios, but it can be applied to most cases requiring basic functionality. In this presented solution, the amount of code isn't significantly larger than that of a standard CRUD implementation, yet the proposed universal CRUD can significantly reduce human resources in the future when implementing such widely-used functionality. This solution can be applied to save time for Back-End developers in performing routine repetitive tasks. Front-end developers can independently configure the required data model and work with the developed universal CRUD.

8. Future steps

The solution presented in this work is based on utilizing the Mongo database. In future it would be prudent to explore the feasibility of crafting a universal CRUD solution applicable to other databases. As demonstrated earlier, the reliance on the Document class from the spring-data-mongo library can be replaced with the utilization of instances of the Map interface. Nevertheless, the MongoTemplate class continues to hold significance in the proposed approach. Future endeavors should aim to eradicate dependencies on databases specific libraries. Exploring the development of a generalized solution that can seamlessly integrate with various databases or file storage systems to abstract away from the underlying database infrastructure would be beneficial.

Further research directions could encompass enhancing the functionality of the universal CRUD to encompass additional operations such as search, sort, and filter. Improving the performance of the universal CRUD and enhancing its user-friendliness are also viable avenues for future exploration.

Acknowledgements

I would like to express my gratitude to the Programme Committee of the XXIII International Scientific and Practical Conference "Information Technologies and Security" (ITS-2023) for the invitation to prepare a full-text version of the paper presented at the ITS-2023 conference that was held in November 30, 2023, Kyiv, Ukraine.

References

[1] Richardson, Leonard; Amundsen, Mike; Ruby, Sam (2013). RESTful Web APIs (ed. First edition). O'Reilly. ISBN 978-1-4493-5806-8.

[2] Martin Kleppmann. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems / Martin Kleppmann. – 41 E University Ave, Champaign, IL 61820,: O'Reilly Media; 1 edition. – 616 c.

[3] Robert Cecil Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall. ISBN 978-0132350884. 2009.

[4] Yazdani, Niloofar, and Sanaz Malekizadeh. "Towards Automatic Generation of RESTful Web APIs." In Proceedings of the International Conference on Web Engineering, pp. 147-160. Springer, Cham, 2019.

[5] Gjorgjioski, Valentin, et al. "Dynamic Generation of RESTful APIs from Heterogeneous Data Sources." In Proceedings of the International Conference on Web Engineering, pp. 309-322. Springer, Cham, 2018.

[6] Thuraisingham, Bhavani, and Ashish Gupta. "Flexible Management of REST APIs with Resource Models." In Proceedings of the International Conference on Web Services, pp. 325-338. Springer, Cham, 2017.

[7] Tsirulnikov Daniil and Hillary Smith. "Automated Generation of REST APIs from Relational Databases." In Proceedings of the International Conference on Database Systems for Advanced Applications, pp. 209-224. Springer, Cham, 2019.

[8] Yu, Ziyu, et al. "Design and Implementation of a Universal RESTful API Server Based on Hypermedia." In 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 472-477. IEEE, 2018.

[9] Kim, Jinoh, et al. "Design and Implementation of a Unified API for Heterogeneous Data Sources." In Proceedings of the International Conference on Big Data and Smart Computing, pp. 77-84. ACM, 2019.

[10] Zhou, Yifan, et al. "Generic and Extensible RESTful API for Data Analytics." In Proceedings of the IEEE International Conference on Big Data, pp. 2810-2815. IEEE, 2019.

[11] Qin, Jiwei, et al. "A Universal API Framework for Integration of Heterogeneous Data Sources in Big Data Applications." In Proceedings of the International Conference on Web Services, pp. 83-96. Springer, Cham, 2018.

[12] Li, Hao, et al. "A Unified API Gateway for Data Services in IoT Applications." In Proceedings of the IEEE International Conference on Web Services, pp. 287-294. IEEE, 2018.

[13] Liu, Xin, et al. "Dynamic Generation of API for Heterogeneous Data Sources Based on Meta-Modeling." In Proceedings of the International Conference on Web Services, pp. 79-92. Springer, Cham, 2019.

[14] Wang, Yue, et al. "Towards a Unified API for Heterogeneous IoT Devices." In Proceedings of the International Conference on Cloud Computing and Big Data Analysis, pp. 95-106. Springer, Cham, 2020.

[15] Zhang, Yucheng, et al. "Design and Implementation of a Universal API Gateway for Cloud-Based Data Integration." In Proceedings of the IEEE International Conference on Cloud Computing, pp. 209-216. IEEE, 2019.

[16] Liu, Yang, et al. "Dynamic Generation of CRUD API for Heterogeneous Data Sources Based on Meta-Modeling." In Proceedings of the International Conference on Web Services, pp. 137-150. Springer, Cham, 2018.
[17] Zhang, Weihua, et al. "Unified Data API for Heterogeneous Data Sources Based on CRUD Operations." In Proceedings of the International Conference on Web Services, pp. 103-116. Springer, Cham, 2017.
[18] Wang, Hao, et al. "A Unified CRUD API for Federated Data Services in IoT Applications." In Proceedings of the IEEE International Conference on Web Services, pp. 267-274. IEEE, 2019.
[19] Chen, Zhen, et al. "Design and Implementation of a Universal CRUD API Gateway for Cloud-Based Data Integration." In Proceedings of the IEEE International Conference on Cloud Computing, pp. 149-156. IEEE, 2020.
[20] MongoDB Documentation. URL: https://www.mongodb.com/docs/manual/core/document/
[21] Spring Data MongoDB API. URL: https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/MongoTemplate.html
[22] Spring Documentation: URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/