Implementation of document state management in pluggable code generation platform

Zdravka Strahinova[1], Nikola Valchanov [2]

[1] *Plovdiv University, Faculty of mathematics & Informatics, 236, Bulgaria Blvd, Plovdiv, Bulgaria*
[2] *Plovdiv University, Faculty of mathematics & Informatics, 236, Bulgaria Blvd, Plovdiv, Bulgaria*

Abstract

This article describes a use case showing the need and usage of a pluggable code generation platform. It describes the steps in which a code generation plugin can be integrated into an already existing system and describes the benefits of its usage.

Keywords

Business logic, code generation, plugin, architectural model

1. Introduction

Many of the problems that we can encounter during the development of an application can be solved trivially. The solution often includes repetitive structures and logic which we call boilerplate code. Writing the code itself and providing the solution takes time and requires maintenance. Most of the time the generated output code is very similar for the different use cases with little to no differences (class names, object names, etc.).

With the improvement of the existing technologies, the developers found a way to automate the process of writing boilerplate code by implementing code generators. These generators allow the developers to implement trivial solutions instantly and to cover almost all of the use cases with just basic configurations. The usage of these tools also guarantees the same file and application structure which enforces best practices all over the project.

The generation process itself always starts with the smallest, most important part which is the data model [1]. Based on the generated model, modern code generators can also generate everything from Data-Access Layers to client-facing pages. Depending on the tool that is used, the code can be optimized using generic classes and methods, inheritance, abstraction, etc. However, we still have a problem because the generated code provides implementation only for the basic acceptance criteria. Because of that, if it's necessary to implement some custom logic, we have to take additional measures to solve the problem.

To provide an effective solution, we can take advantage of the suggested architecture for a model of injecting business logic into auto-generated code [3]. We can review the following use case of a document management system that will demonstrate how the architecture can be implemented.

Education and Research in the Information Society, October 13–14, 2022, Plovdiv, Bulgaria
EMAIL: zdravkastrahinova1022@gmail.com (A. 1); nvalchanov@uni-plovdiv.bg (A. 2)
ORCID: XXXX-XXXX-XXXX-XXXX (A. 1); XXXX-XXXX-XXXX-XXXX (A. 2); XXXX-XXXX-XXXX-XXXX (A. 3)


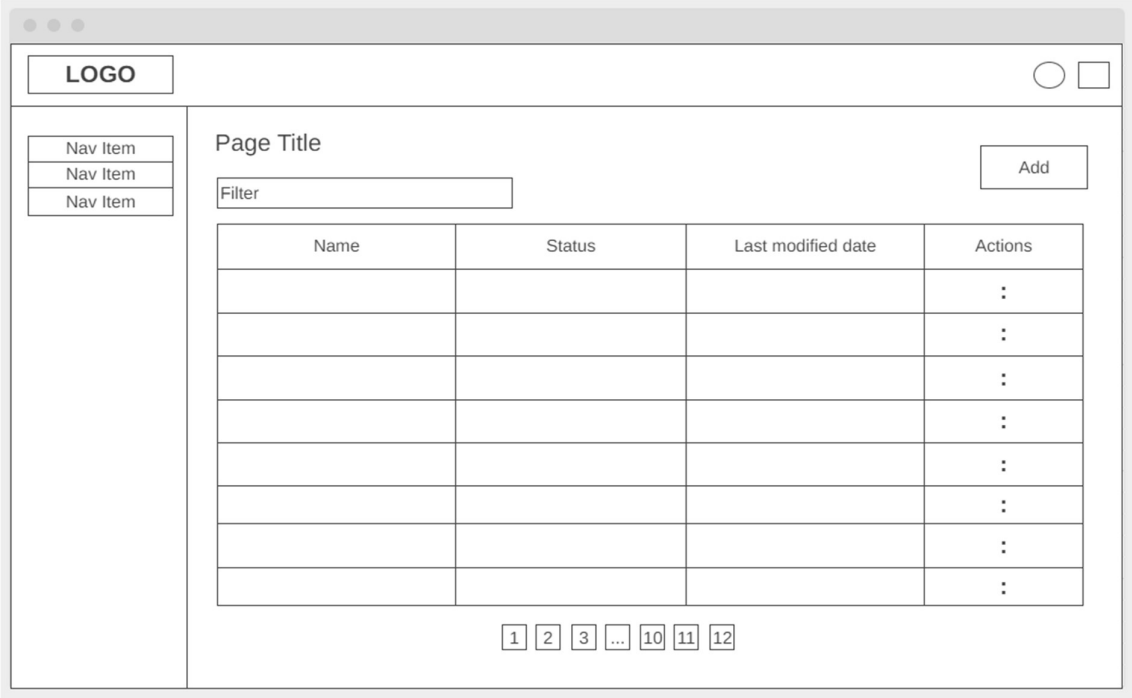

Figure 1: Wireframe that presents the user interface of the document-state management system

By reviewing the wireframe presented in Figure 1, we can see that the main functionality of the software is focused on document state management. The general responsibility of it is to manage documents depending on whether they have been signed or not and additional data will be kept when an operation of that document occurs. Until the document has been signed it can be modified or replaced in any way. However, after the signing operation the file is locked and the functionalities over it are limited to read-only.

Any user in the system can create a document. Every time someone creates a draft, the system will keep additional information regarding the author and the date that the document has been created. The document itself will have a status and that status will determine whether or not the document can be modified any further.

Depending on the different states a variety of actions can be executed over the document. If it is in a state different from signed, then the user can see or edit his documents. On every action that modifies the document, the system will keep history over the changes and will show who and when has edited the document. However, if the document has been signed, the only action that can be performed over it is to delete it (or archive it) and it can only be performed by its author.

From a developer's point of view, the development of the use case above will require a lot of code duplication and unnecessary checks when managing the documents. In that manner, we'll have to think of a way to optimize that application and implement additional services or middleware which can be time-consuming when doing them manually. That will also increase the complexity of the codebase. To solve this problem, we can take advantage of a plugin implementation that will auto-generate this additional business logic by doing runtime calculations and decorating existing entities with additional properties.

2. Architectural model of plugin implementation

2.1. Data Structure

To start the process of code generation for the plugin, we will need to provide an input object in a JSON [4][5][6] format. The plugin will parse the input and will determine property types based on their values. For complex objects or data structures, it will generate the needed classes and their relations.

In our use case, the input JSON will be as shown in Figure 2.

```
{
    "document": {
        "status": "draft | signed",
        "content": "Document text content"
    }
}
```

Figure 2: Example of JSON input data

By examining this input, we can see that we have a complex property called “document” and a description of its properties. The code generator will parse the input object and will generate a Document based on the “document” property. During the generation process, the plugin will automatically extend an already predefined BaseModel class which will be responsible for attaching a unique identifier to the base document class. Additionally, the status property will be parsed to an enumerator thanks to the special pipe syntax provided in the input itself. This gives us a great advantage because the plugin does not pose any restrictions about the count or the type of statuses that will be generated.

When the generation process completes, as an output result two more classes will be generated.

2.2. Model Architecture

To get a better understanding of what the model [2] will be, let’s review the following class diagrams presented in Figure 3 and Figure 4:

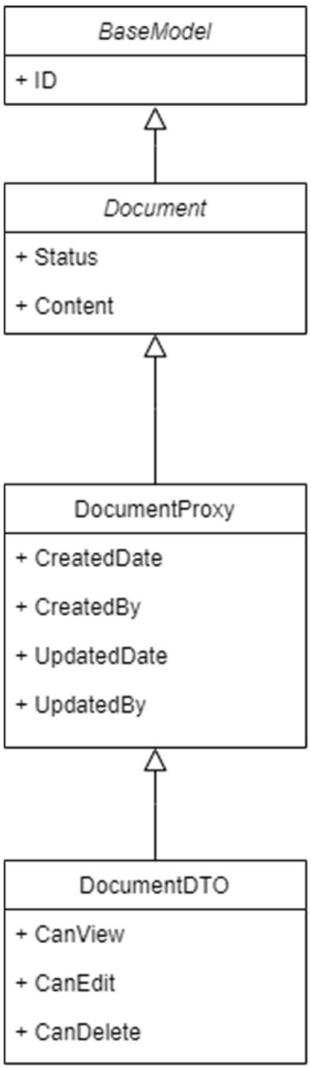

Figure 3: Models class diagram

The diagram presented in Figure 3 describes the structure of the document. It is separated into three parts. We start with the Document itself, which is described by a unique identifier, status, and document content.

The DocumentProxy object describes the object model that will be saved in the database. In it, we have the additional meta properties of the document which will give us information about when it has been created or modified and by whom.

The last part of the diagram shows the DocumentDTO (Data-transfer object [7]). It represents the ACP (Access-Control-Properties) which are evaluated during runtime. They are not saved in the database and do not exist in the original entity. The purpose of the ACP is to be used on the client-side to enforce restrictions over the actions that can be executed.

The main benefit of separating the object as described above in Figure 3 is the possibility to inject code or business logic into different layers of the model (document) architecture. The entity inherits the original document and decorates it with the listed properties during design time. The DTO on the other hand applies business logic to the already decorated document during runtime.

All of the logic described above is implemented by the services. Their architecture is presented on Figure 4. We have separated the logic into two parts - a generic service layer (Service in Figure 4) and a domain-specific layer (DocumentService in Figure 4).

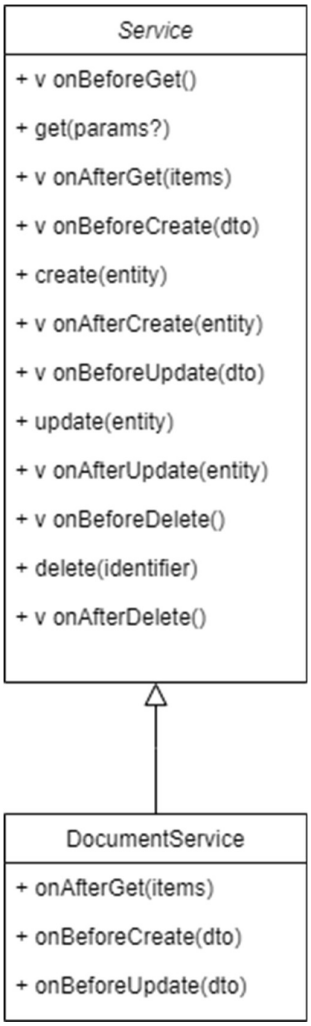

Figure 4: Services class diagram

The Service itself is a class provided by the plugin that contains methods with generic implementation of all CRUD (Create-Read-Update-Delete) operations that will be performed over the entity. In addition to these methods, the service contains a pair of virtual methods for every CRUD operation. These virtual methods follow a standardized naming convention so they can be associated with a corresponding operation. They are a finite amount that depends on the number of CRUD operations. The main idea is that these virtual methods will add hooks that will allow us to execute additional code before and after the specific operation. They are marked as virtual so we can allow the domain-specific layer to choose whether to implement them and to take advantage of the hooks in a specific way.

The DocumentService can inherit [8] the Service class provided by the plugin and with that inheritance, it can use the already defined CRUD operations from the parent. In addition to that, it can implement any of the virtual methods in order to easily customize the business logic. In our use-case, the DocumentService will implement:

- OnAfterGet - this will allow the service to evaluate the ACP during runtime. The input data of that method is the raw value returned from the DB and the result is formed by a mapper function that will transform the raw data into a DTO and will evaluate the ACP properties.
- OnBeforeCreate - this method is responsible for the initial set of the CreatedDate, CreatedBy, UpdatedDate, and UpdatedBy properties. The input data of that method will be a DTO object containing the DocumentDTO properties, and the result of the method will be a DocumentProxy object that will be saved in the database.

- OnBeforeUpdate – it is responsible for the management of the UpdatedDate and UpdatedBy properties. The input parameter of the method is again a DTO, and the result is again a DocumentProxy that will be saved in the database.

2.3. Action Model Architecture

The action model architecture [9] represents a high-level architecture that aims to demonstrate how the Domain-specific service, which implements the Plugin Service, will integrate with the other parts of the system. That architecture is described by the sequence diagram presented in Figure 5:

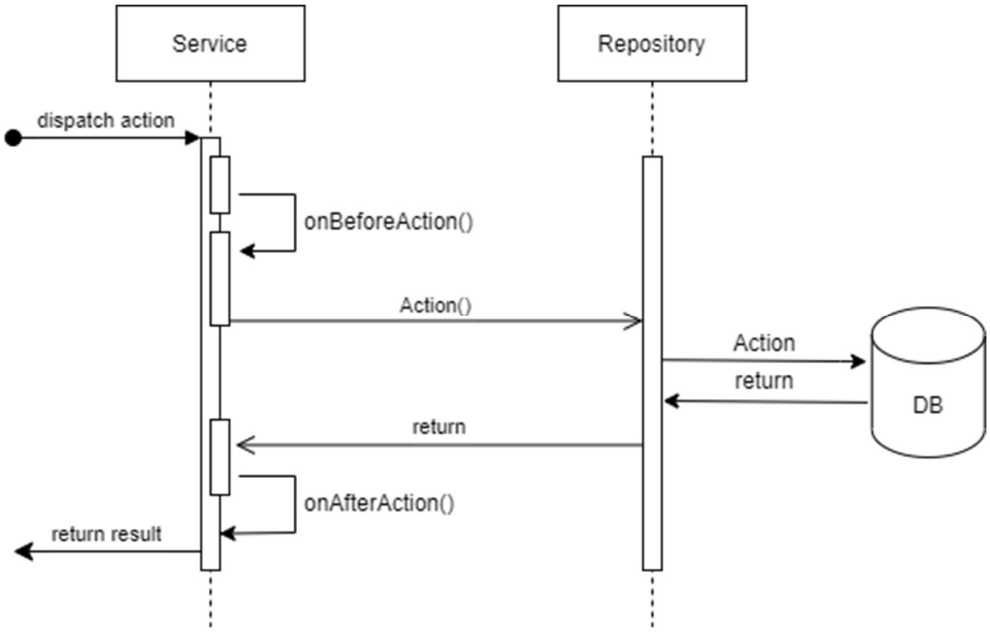

Figure 5: Generic sequence diagram of an action model

The sequence diagram in Figure 5 presents three components of the system architecture. The main purpose of that diagram is to explain how the communication between the different parts of the Data-Access-Layer (Services and Repositories) will be performed and to show how the plugin itself can be integrated as a part of a real architectural solution. These components are as follows:

- Service - the main responsibility of the service is to execute any kind of business logic. This will happen any time an action has been dispatched to it.
- Repository - this component is responsible for the execution of operations over the database and raw data management. It is not overwhelmed by any kind of business logic.
- Database - the data storage. Can be any kind of storage.

By reviewing the sequence diagram in Figure 5 we can understand that this is a generic architectural model of action. Once the action has been dispatched to the service, it'll try to execute a specific CRUD operation. This operation is accompanied by the pair of virtual methods which we reviewed in [Figure 2]. These methods will always be executed but they'll modify the data additionally only if they have any custom implementation.

The sequence of one action begins with an OnBeforeAction method, which is encapsulated only in the service. The base implementation comes from the generic service class, and it is not responsible for any data transferring. Its main responsibility is to modify and transform the DTO into an entity model. Whether or not it'll execute any additional logic depends on the implementation of the DSS (domain-specific service).

Once the OnBeforeAction has been completed, the process continues with the actual action. The action is responsible for transferring the data to the Repository layer or vice-versa, where it will be handled. Here, the action always works with entities.

After the action is completed, the service is going to execute its last callback called OnAfterAction. This method is responsible for the last modifications over the entity where it can apply the needed transformations so the entity can be converted to a DTO. This marks the end of the action, and the result is returned to the dispatcher.

3. Conclusion

Nowadays, we rely on code generation almost daily. All the boilerplate code that is required in any application is provided by them with little to no configurations. This reduces the time needed for some functionalities but introduces a few small issues that need to be handled.

In this article, we presented an architectural model which demonstrates a way to inject business logic on different levels of the Data-Access-Layer by integrating a third-party plugin that provides customizable middlewares. This allows us to use a generic service for CRUD operations and still have control over the data flow so we can customize it for our use-cases. This is achieved by providing virtual methods that are executed before and after every action and by overriding them, we can still manage the workflow.

4. References

[1] S. Kelly, JP. Tolvanen, Domain-specific modeling: Enabling full code generation, Wiley, 2008.

[2] S. J. Mellor, K. Scott, A. Uhl, D. Weise, MDA distilled: principles of model-driven architecture, Addison-Wesley Professional, 2004

[3] F. Zhang, Z. M. Ma, J. Cheng, Enhanced entity-relationship modeling with description logic, Knowledge-Based Systems 93 (2016), 12-32. doi: 10.1016/j.knosys.2015.10.029

[4] P. Bourhis, J. L. Reutter, D. Vrgoč, JSON: Data model and query languages, Information Systems 89 (2020), 101478. doi: 10.1016/j.is.2019.101478

[5] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of JSON and XML data interchange formats: a case study, Caine 9 (2009), 157-162

[6] Z. H. Liu, B. Hammerschmidt, D. McMahon, JSON Data Management: Supporting Schema-Less Development in RDBMS, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 1247–1258. doi: 10.1145/2588555.2595628

[7] P. B. Monday, Implementing the Data Transfer Object Pattern, in: Web Services Patterns: Java™ Platform Edition, Apress, Berkeley, CA, 2003, pp. 279-295. doi: 10.1007/978-1-4302-0776-4_16

[8] D. Shiffman, Advanced Object-Oriented Programming, in: Learning Processing, 2nd. ed, Morgan Kaufmann, Boston, 2015, 487-502. doi: 10.1016/B978-0-12-394443-6.50022-6

[9] S. Smirnov, M. Weidlich, J. Mendling, M. Weske, Action patterns in business process model repositories, Computers in Industry 63 (2012), 98-111. doi: 10.1016/j.compind.2011.11.001