Enabling Content Management Systems as an Information Source in Model-driven Projects

Joan Giner-Miguelez[1][0000−0003−2335−6977], Abel Gómez[1][0000−0003−1344−8472], and Jordi Cabot[2][0000−0003−2418−2489]

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC), Barcelona, Spain
{jginermi, agomezlla}@uoc.edu,
[2] ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Content Management Systems (CMSs) are the most popular tool when it comes to create and publish content across the web. Recently, CMSs have evolved, becoming *headless*. Content served by a *headless CMS* aims to be consumed by other applications and services through REST APIs rather than by human users through a web browser. This evolution has enabled CMSs to become a notorious source of content to be used in a variety of contexts beyond pure web navigation. As such, CMS have become an important component of many information systems. Unfortunately, we still lack the tools to properly discover and manage the information stored in a CMS, often highly customized to the needs of a specific domain. Currently, this is mostly a time-consuming and error-prone manual process.

In this paper, we propose a model-based framework to facilitate the integration of headless CMSs in software development processes. Our framework is able to discover and explicitly represent the information schema behind the CMS. This facilitates designing the interaction between the CMS model and other components consuming that information. These interactions are then generated as part of a middleware library that offers platform-agnostic access to the CMS to all the client applications. The complete framework is open-source and available online.

Keywords: Content Management System · Headless · Model-Driven Engineering · Reverse Engineering · REST API

1 Introduction

Content Management Systems (CMSs) are one of the most popular options to create content across the web. These systems, such as WordPress, Drupal, or Joomla, represent nearly 61,3% in terms of published websites [3, 28]. One of the main reasons for this popularity is the great user experience provided by CMSs while creating content that empowers non-technical users to be part of the content creation chain [4]. Besides, CMSs have evolved in the last years by implementing APIs to allow external apps to discover and interact with the

content of these systems. This evolution, known as *headless CMSs*, has shifted the main focus of these systems, traditionally on desktop solutions, to other kinds of applications such as mobile apps and other front-end apps.

As an example, in the media industry, CMSs are widely used to build digital solutions. Sony Pictures[3], Le Figaro[4], and Syfy Channel[5] are some of the media companies powering their digital solutions with open-source CMSs. The content created inside these solutions, such as news, podcast, or video, represents the key asset of these companies. Therefore, any new software development project in these companies will likely depend on, and require, the content in the CMSs.

This integration is becoming more difficult as CMSs solutions are becoming increasingly complex as so they do the APIs they expose. Moreover, typically, large companies must deal with various deployed CMSs, some of them legacy versions, that need to be combined to satisfy the application informational needs. There is a clear need for new methods that help in managing these complex integration processes as part of new software development projects. In parallel, there is increasing adoption of the Model-Driven Engineering (MDE) practices [11, 14] as MDE has been proven useful to tame the development complexity. Unfortunately, while we have several solutions to facilitate the integration of SQL [10,16] and No-SQL [1,7] backends in MDE-based processes, there is a lack of solutions to support applications that rely on headless CMSs as a content source. Therefore, the integration of headless CMSs with other apps has been done by manual solutions, being these solutions time-consuming and error-prone.

In this work, we propose a framework to enable the integration of headless CMSs in MDE-based software development processes. The framework is composed of *(i)* a core model of CMSs, *(ii)* a reverse-engineering process to extract the model from an existing deployed CMS in a UML representation, and finally, *(iii)* a code-generation process that generates a middleware library to bridge the gap between the front-end consumer app and the CMSs. The framework focuses on existing CMSs rather than on creating new ones from a UML model because we have detected that most of the CMSs are already created, and there is a need to use them as a content source.

With our framework, companies can quickly discover the information schema behind each CMS and represent it explicitly as a UML model. This model can then be integrated with other software models in the project. Finally, our framework also assists in the generation phase, simplifying the writing of all the required *glue* code between the different components. As an outcome, we have built our proposal as an Eclipse plug-in and can be found in an open repository[6].

The paper is structured as follows. Section 2 presents the general overview of the framework. Section 3 presents our *Core CMS model* and, Section 4 presents the reverse engineering process. Section 5 shows an example of integration and, Section 6 presents the generation process. Finally, Section 7 presents the devel-

---

[3] https://www.sonypictures.com/tv
[4] https://www.lefigaro.fr
[5] https://www.syfy.com
[6] https://hdl.handle.net/20.500.12004/1/C/RCIS/2022/001

oped tool, Section 8 summarizes the related work, and Section 9 wraps up the conclusions and the future work.

2 Framework overview

In this section, we provide an overview of the proposed framework. The main goals of the framework are *(i)* to extract and explicitly represent the specific model of a deployed CMS to enable the integration of CMSs as information sources within a global model-based development process, and *(ii)* to generate the *glue code* that will transform any model-level interaction between the CMS model and other models (e.g. GUI models) in proper calls to the Headless CMS API.

Following Fig. 1, the first step is the *Reverse engineering* process. This process takes as input the target CMS (in particular, its URL and user credentials), and imports the predefined *Core CMS model* with basic CMS concepts. Then, creates the target CMS model as an extension to the core CMS one. The discovery process is implemented as a set of calls to the CMS API, adapted to the API offered by each CMS platform. We use UML to represent these models.

As this obtained model is a purely standard model, designers can refine it and combine it with other models. For instance, we can easily model how a User Interface model queries the CMS model to retrieve the information it needs to show to the user in a platform-independent way.

The final step is the *Code generation* process. This process takes as input the *CMS Driver* that corresponds to the CMS platform and uses it to generate a middleware library allowing consumer apps to get the information they need from the CMS. Note that the API offered by the middleware only depends on the CMS content schema, not on the underlying CMS platform. The API is consistent across any CMS technology which facilitates any future CMS migration. As part of the API we have all the usual filtering, ordering and pagination methods that may be needed to iterate on the CMS content.

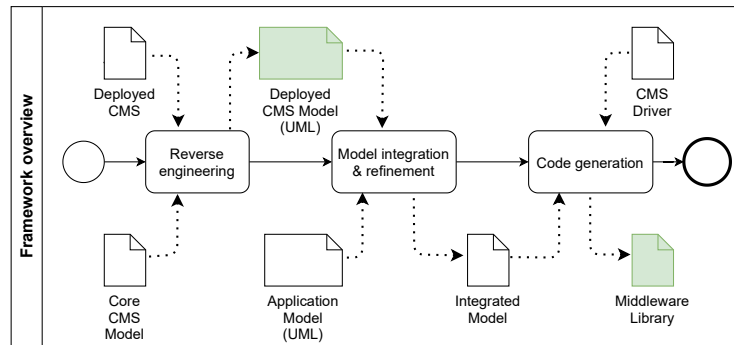In next sections we present in detail each of these steps.

Fig. 1. Framework overview

3

3 The *Core CMS Model*

This section presents our core model for CMS, representing the common concepts shared by all major CMS vendors. The goal of this model is to facilitate the representation of the information schema behind specific CMSs by providing basic types that can be extended. This avoids having to start from scratch every time. Moreover the core elements also facilitate the management of CMS models when there is no need to access its detailed structure. This core model will be used by the reverse engineering process, see Section 4, to create the complete model-based representation of an input CMS,

We have come up with the elements in the core model after analyzing the three major CMS platforms (Drupal, WordPress and Joomla; as they together have over 70% of the market share in this domain [3, 28]) and inferring the commonalities among them. Figure 2 shows a partial view of this common model. We focus on the access control and the content information parts. A more complete model, which also supports features such as revisions or translations of the content, is available in the online repository.

According to Figure 2, we can see that many classes inherit from *ContentEntity*. This class provides the last update time of the entity and a unique identifier for it. This identifier will be used to query the CMS API and get the entity information and its linked resources.

Immediately below, we have *ContentType*, that represents any content stored in the CMS. Typically, most CMSs offer at least pages and posts as content types, but any non-trivial CMS will come with several custom types to better represent the information domain the CMS is serving. Each custom type will be a subclass of this root *ContentType* element. We can always associate a *Comment* to a content type. Comments have a hierarchical relationship, allowing different threads of comments to coexist in parallel and can be associated with a user if the comment is done by a registered one. Content types can also be classified in taxonomies. Two typical taxonomies are *Tags* and *Categories*. As we have found these taxonomies in all CMSs we have directly included them in the core model but a specific CMS could also have custom taxonomies. Taxonomies also have a hierarchical relationship.

Moreover, CMSs also have *Media* files that can be attached to the content, such as images, videos, or audio files. They are represented independently of the context where they are embedded, since media items can be reused across different content pieces. This also means that CMSs could also be used as a media repository. Finally, *Blocks* represent complementary pieces of content of a *ContentType*, for example, the ads attached to a particular content piece.

Beyond content management, CMS pay special attention to the access control and allow defining who can access the content following a standard Role-Based Access Control approach [15], where every user has a set of *Roles* attached, and every *Role* has a set of permissions attached. *Permission* represents a specific action that a user can do inside the system. There are two subtypes of permissions, the *GeneralPermission* which is not attached to a particular domain entity, for
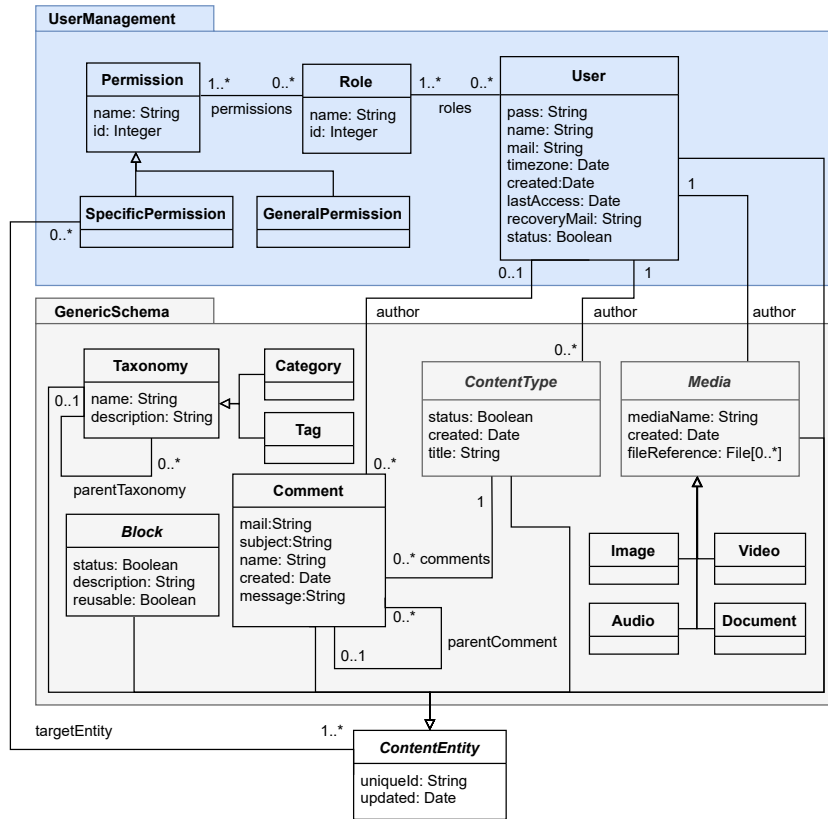
4

Fig. 2. Schema and User Management of the Core CMS model

instance, "access to the admin dashboard" and, the *SpecificPermission* that is related to a specific content type, for example, "edit video article".

4 The reverse engineering process

In this section, we explain the *reverse engineering process*. This process is the first step in the integration framework and aims to extract the model of a deployed CMS. On one side, it starts by getting the URL and the user credentials of the target CMS to discover its API. On the other side, the process receives the *Core CMS Model* presented above to perform a set of extensions over it to extract the model of the deployed CMS. As a result, we get the model in UML facilitating the integration with other parts of the system as shown in Section 5

In this section, we present a comparison of the API architecture between technologies, the discovery process followed to extract insights from the API, and the extension strategy followed over the *Core CMS Model* to fit particular data scenarios.

5

4.1 The CMSs API

Concerning the API, the different CMSs implementations use a REST API to expose their content[7]. These REST APIs follow different standards depending on the implementation. As an example, Drupal and Joomla follow JSON:API [12] as a way of structuring the answer and providing links to navigate between resources, while WordPress uses HAL [23] as its standard way to provide linking between resources.

In contrast, all the APIs expose the schema of the CMS. As a schema, we mean the entities and attributes of the CMS and the custom ones created over the specific installation. Besides, if a change is done in the CMS schema, the API is updated automatically, meaning that we can consider this API as the current state of the CMS schema. Finally, all the analyzed technologies implement Open API [17] as a standard for API description in conjunction with custom discovery mechanisms specific to each technology. The *Reverse Engineering* process uses the Open API descriptions and these custom methods to perform the discovery process.

4.2 Discovery process

To perform the extraction, we analyze the API by using the discovery methods of each CMS implementation. First, we systematically fetch all the API resources to detect all the entities exposed by the deployed installation. Then, we analyze the answer to infer from which type is every detected entity and which attributes has. Once we have spotted all the entities and their attributes, we analyze, again, the answers to detect their relationships.

Listing 1. Example excerpt of API answer

```
1   "info": {
2       "title": "Journal CMS",
3   },
4   "host": "example.com",
5   "basePath": "/api",
6   "definitions": {
7       "node--videoarticle": {
8           "title": "node:videoArticle Schmea",
9           "description": "news main content",
10          "properties": {
11              "attributes": {
12                  "id": "Integer",
13                  "title": "String",
14                  "...": "..."
15              },
16              "relationships": {
17                  "0": {
18                      "type": "taxonomy--category",
19                      "link": "..."
20                  }
21              }
22          }
23  }
```

---

[7] Other API architectures as GraphQL are also implemented but not included in the standard installation. More information about the discussion can be found at https://dri.es/headless-cms-rest-vs-jsonapi-vs-graphql.

In Listing 1 we can see an example excerpt of Drupal's API answer. In this example we can see a *node--videoarticle* definition which is an entity that inherits from *ContentType*. We guess the inheritance from the term "node" which refers to *ContentType* in Drupal's terminology. Under *properties* we can see a set of attributes as *id*, *title* and others (lines from 11 to 15). Finally, we can see that our entity has a relationship with a *Taxonomy* of type *Category* and the same answer provides us a link to fetch and discover this relationship (lines from 16 to 20).

If we detect entities that are not in the proposed *core CMS* model, we can know from which types these entities inherit. For instance, in Listing 1, *VideoArticle* is a specific type of *ContentType*, and therefore, we will create a new class extending our *ContentType* class of the core model. This new class also will retain some specific annotations, such as the URL of the endpoint. It could be noted that this process is tied to every technology implementation as every technology builds the answer in different ways. Therefore, a specific extractor needs to be developed to support every CMS implementation.

4.3 Extraction example

In Figure 3 we have an example of an extracted model of a journal site. This example shows how the discovery process and the extensions over the *Core CMS model* can describe complex data scenarios. This example describes a journal with different types of content such as video articles, written news, and podcasts with relationships between them.

In the figure, *VideoArticle* represents the news edited as video. This class has a relationship with a *Video* and some attributes, like *likes*, which represent the number of likes made by users. In terms of Taxonomies, *VideoArticle* have a relationship with *AgeRating* which classifies the video by age recommendation. In addition, it also has a set of *Tags* to facilitate the search of this content. Finally, this class has a relationship with a set of *NewsArticle*. If a user wants further information about the news, this represents the related written news of the *VideoArticle*.

NewsArticle class, which extend from *ContentType*, represents the written news. As previous, this class has attributes, like *likes*, which represent the number of likes made by users. Also, the *NewsArticle* has a relationship with *BannerAds*, which extends from *Block*, that represents the advertisements attached to a specific news. This relationship allows configuring the advertisements policy inside the CMS, and reproduce it also in the consumer apps. In addition, *NewsArticle* can have a set of related *VideoArticle* attached to it. In the journal's example, this represents written news that have their version as video news to be shown on a streaming platform or TV. Moreover, *NewsArticle* also have media *Images* as the images of news.

Finally, another main content of the site is *Podcast*, which has a relationship with media *Audio*, which represents content edited by audio. This content audio format has been gaining popularity in the last year and is a clear example of how CMS manages different types of content.
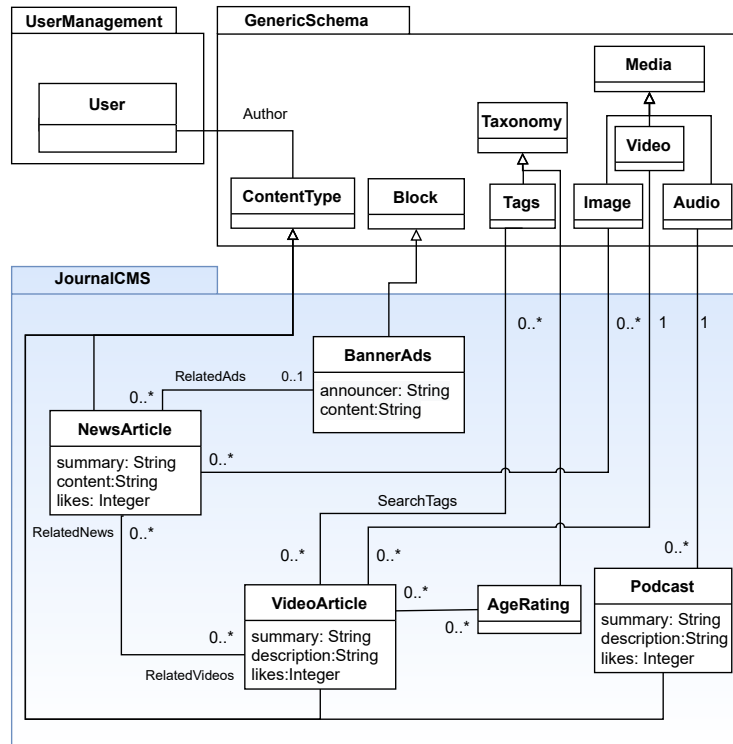
Fig. 3. Example extracted model of a journal's CMS

In conclusion, after the *Reverse Engineering* process, we obtain the model of our deployed CMS in UML. This model can work with complex data scenarios, as we can see in Figure 3, and is ready to be integrated with other parts of the systems, as we see in the next section.

5 Model integration and refinement

The result of the reverse engineering process is a fully compliant UML class diagram. As such, we can use, combine, and refine that diagram as we would do with any other model of our information system. One of the most obvious integrations can be specifying how the user interface of different consumer components, such as web applications or mobile apps, query the extracted CMS model to retrieve the information they need to show to their users.

We will use this scenario to continue with our running example. Let's assume our media company is interested in building a mobile app to show a video news feed to its users. More specifically, the app will show the video feed, let users *like* some specific videos (and these *likes* will be stored the CMS for future rankings
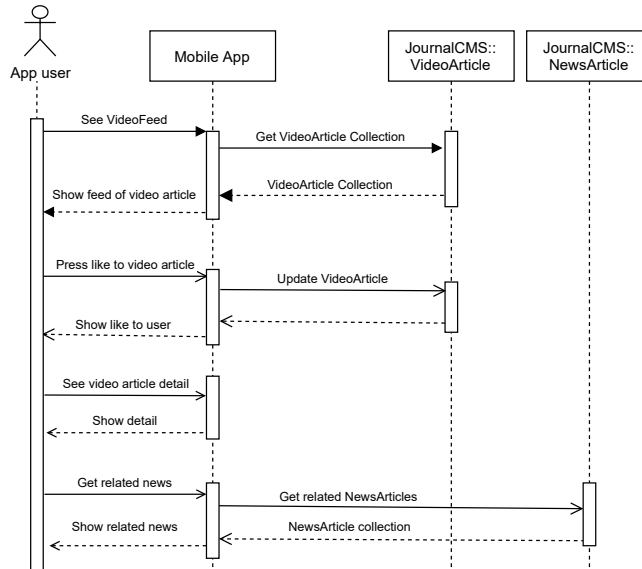
Fig. 4. Integration example sequence

and optimizations), access the details of a particular video and get related news to the ones she is watching.

All this information is available in the Journal CMS. And since we now have it explicitly represented as a model, we just need to define the interaction between the UI app model and the CMS model. A possible sequence diagram is shown in Figure 4. In the diagram, when a user wants to see the video feed, a call to the *VideoArticle* class is performed. When a user likes a specific video, this is translated to an update call to the journal's *VideoArticle* class. Finally, when a user sees a video detail, the user can access the related content, querying, in that case, the journal's *NewsArticle* class.

The methods used in the sequence diagram are self-explanatory and, as we will see in the next section, will be made available in the next section via the code generation process. For more advanced scenario, we could also consider using a language like the Object Constraint Language [19] to more formally specify the exact queries to perform to the CMS.

Similarly, we could model other types of interactions or static diagrams with our CMS model. The key point is to realize that these interactions are modeled as any other type of interaction between two models and that when doing so the designer does not need to worry about the underlying CMS platform and version. The designer can stay at a platform-independent model. In fact, it would be possible to define a client consumer application that reads from two different CMS models without the designer not even being aware that the queried data elements belong to separate CMSs.

6 Code generation infrastructure

The final step of the framework is the *code generation* process. This process aims to generate a middleware library to facilitate the communication between the deployed CMS and consumer applications.

Regarding the presented integration example in Figure 4, the middleware would be in charge of the interaction between the *Mobile app* and the content of the Journal's CMS such as *VideoArticle* and *NewsArticle*. This middleware could work with other components (e.g., generating the mobile app) by providing the glue code required to ensure the communication between the app and the content source.

In this section, following Figure 5, we explain how this middleware is structured to be easily extensible for other technologies and how the consistent API is designed to abstract practitioners from the underlying technology. Finally, we provide a usage example of the middleware library following the integration example presented in Section 5.

6.1 Middleware structure

The middleware structure is composed of a set of generated classes from the extracted model, and a *CMS Driver* composed by a set of static classes that allows us to move back and forward the data of the generated classes between the consumer app and the source CMS.

The generated classes are the dynamic part of the middleware. For each UML class of the extracted model, a java class is generated for it. This class contains the UML class attributes and relationships, and the possible sorter and filters to interact with the API. For instance, in Figure 5, the class *VideoArticle* corresponds to the class *VideoArticle* of the example extracted model in Section 4. In addition to the previous ones, a *JournalSiteManager* is generated. This class is built using the singleton pattern and represents the whole CMS site, and brings access to the generated classes with specific methods for each one.

The *CMS Driver* classes are the static part of the middleware. The *Driver* class is in charge of the communication with the source CMS and is the only one tied to a particular technology. This approach means that the driver is the only class we need to reimplement to extend support for new CMS technologies. This driver has a single relationship with the *JournalSiteManager* class. In Figure 5, we can see a *DrupalDriver* class as our journal's site is powered by Drupal.

Also, inside the *CMS Driver*, and to ensure the separation between the agnostic part and the specific part, we propose the *GenericResource* class as an interface between the generated classes and the driver. In addition, we suggest a *SearchQueryBuilder* class as a way to provide a consistent API to practitioners in terms of fetching the source CMS. At last, *SearchQuery* is the result of the query builder being an immutable class and allowing the *DrupalDriver* class to build the particular final query to the CMS API.
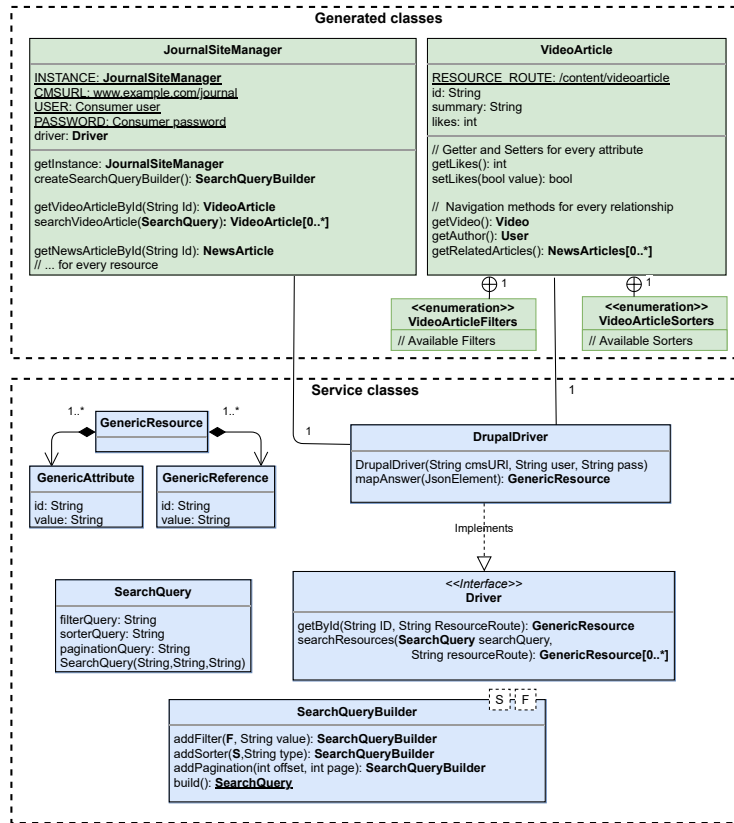
10

Fig. 5. Middleware's implementation diagram: *VideoArticle* example

In summary, the middleware is composed of a set of dynamic classes generated from the extracted model and a set of already developed static classes representing the *CMS Driver*.

6.2 The middleware API

One of the key points is to facilitate the usage of this middleware without regarding the underlying technology. To do so, we have proposed a consistent API as a standard method to interact with them. We have inspired the design of the consistent API in the identified shared concepts of the different open-source CMS APIs.

The main methods to interact with the content are present in the *JournalSiteManager* of our example. This class has a *getById* and a *search* methods for every generated class from the extracted model. The first one needs a content id to return the result, and the second one needs a *SearchQuery* to do so.

The *SearchQueryBuilder* is a query builder that wraps up the concepts of filtering, ordering, and pagination mentioned before. It provides a way of building queries benefiting from the main API features without regarding its underlying implementation. Therefore, we can add filters, sorters, and pagination using the provided methods. Once the search query is built, it returns a *SearchQuery* as an immutable class.

Finally, one of the key points of the analyzed REST API is hypermedia. Hypermedia is the ability to navigate through resources to discover the API. For example, if we get a *VideoArticle* and we want to get a specific related *NewsArticle*, we know from the answer itself how to retrieve this related content. Instead of automatically getting the related content, we have decided to implement this navigation method to avoid recursive calls. In Figure 5 we can see an example of this in the *VideoArticle* class that has navigation methods. From a *VideoArticle* we can get the related *NewsArticle* or the *User* corresponding to the author.

6.3 Middleware usage example

In Listing 2, we can see an example of the *Mobile App* class of Figure 4 using the generated middleware. In this example, we can see how practitioners could easily integrate the Journal CMS as a content source without regarding the underlying technology. This example uses the java code generated by the tool presented in the next section following the integration example of Section 5.

In the main method, we perform the sequence represented in Figure 4. Therefore, in line 5, we get the video feed by calling the *getVideoFeed()* method. In this method, we get an instance of *JournalSiteManager* class as it provides methods to interact with all the generated classes. Then, to get the list of videos we call the method *searchVideoArticle()* form the *JournalSiteManager* in line 18. This method requires a *SearchQuery* as an entry parameter. In our case, we get a

Listing 2. Example MobileApp class demonstrating the use of the middleware library

```
1   class MobileApp() {
2
3     public void main(){
4       // 1 - Get the video feed
5       List<VideoArticle> feed = getVideoFeed();
6       // 2 - User press like to the first video
7       VideoArticle firstVideo = feed.get(0);
8       firstVideo.setLike(true);
9       // 3 - User wants to see the related news of the video
10      List<NewsArticle> relatedNews = firstVideo.getRelatedArticles();
11    }
12    public List<VideoArticle> getVideoFeed() {
13      // The site manager class
14      JournalSiteManager site = JournalSiteManager.getInstance();
15      // We get a search query builder
16      SearchQueryBuilder queryBuilder = site.getSearchQueryBuilder();
17      // We call the search method with an empty query
18      return site.searchVideoArticle(queryBuilder.build());
19    }
20  }
```

searchQueryBuiler, but we do not need to set any filter or sorter configuration. So we build it directly in line 18. The method returns a list of *VideoArticles*.

Then, a user presses the *like* button of the first video of our video feed. Therefore we get the pressed video in line 7, and we call the *setlike()* method of it in the next line. Finally, in line 10, to retrieve the related content, we use the navigation methods of the *VideoArticle* class calling the *getRelatedNews()* method. This methods returns the list of *NewsArticles* related to the particular *ViodeArticle*.

7   Tool support

We have developed a prototype implementation of our framework, available under the Eclipse Public License[8]. The source code of the project and the Eclipse update site are available in the public repository.

Our implementation is built in Java and relies on the Eclipse Modeling Framework [24] for the core modeling support. In particular, we have used Xtext and Xtend [25] to implement the code-generation process. WordPress and Drupal are the two CMSs supported by our tool, but others could be easily integrated. Further instructions about the usage of this tool are provided on the public repository.

8   Related Work

Several research works have investigated the intersection between CMS and MDE. A first group of works have focused on modeling specific CMS components or extensions/plugins. Priefer [18] proposes a model-driven approach to generate Joomla extensions. Martínez [15] presents a metamodel for the access-control rules in a CMS to be used for verification and validation purposes. Our work targets the complete CMS definition and not just specific components.

Another group of works focus on the generation of a new CMS implementation. Trías [26] propose a CMS common metamodel to capture the key concerns required to model CMS-based web applications and, Qafmolla [29] propose the generation of CMS instances from it. Similarly, Souer [22] aims to generate a CMS from an automatically generated configuration description created by business users. Modeling languages to define new CMS is also the goal of Saraiva [21]. Our work is different from these approaches in that: 1) Our model is not generic but extensible to precisely represent the information schema of the CMS (and not just global concepts as *post* or *page*) and 2) our target is the integration of the CMS with the rest of the system and therefore our framework comprises the interactions between the data consumer components and the CMS and 3) we start from an exiting CMS, which, in our opinion, is a more realistic scenario as most companies already have a CMS in place and are not looking to start from scratch but evolve it.

---

[8] https://www.eclipse.org/legal/epl-2.0

It is also worth mentioning works that have similar goals to ours, but that target different information sources. SQL databases (e.g. Egea [10], and Nguyen [16]), NoSQL databases (e.g. [8], [1], [5]), Data Warehouses (e.g. [27]) and even spreadsheets [2] have been integrated in MDE processes.

Finally, given that headless CMSs expose their data through a REST API, we comment on model-based proposals targeting REST APIs. Some relevant examples are [9], presenting an MDE-based tool to create, visualize, manage, and generate OpenAPI definitions, [6], proposing an MDE process to discover and composite JSON documents or [20], focusing on building REST APIs directly from user requirements. In our case, we do not aim to generate new APIs but to provide a specific reverse engineering process adapted to the concrete REST API endpoints and patterns offered by popular CMSs. In this sense, some concepts of these works have been integrated in our approach as building blocks on top of which we have built the specific connectors we required for the CMSs.

9 Conclusions and future work

In this work, we have proposed a model-based framework to integrate headless CMSs in a development process. Thanks to our framework, the conceptual schema behind the CMS can be easily exposed and reused as part of the global development process. The framework comprises a core CMS model, a reverse engineering process that extends it to precisely represent the data structure of an input CMS, and a middleware that enables modeling front-ends that need to consume that data in a platform-agnostic way.

This facilitates the reusability and evolution of the software system, also enabling potential migrations among different CMS versions and platforms, e.g., to benefit for more advanced features or scalability improvements, while minimizing the impact of such changes on all the deployed clients (e.g. mobile apps).

As further work, we will study how the reverse engineering process could take into account presentation suggestions that are now also being embedded in the CMS headless API to achieve a more homogeneous look&feel across different client front-ends. There are some important trade-offs here that deserve to be analyzed. We would also like to explore the benefits of exposing CMS data in other domains, like Machine Learning, where easy access to CMS data could be exploited for training ML models. In this context, deploying our framework as a plugin for model-based data science tools like KNIME [13] could be interesting. Finally, at the tool level, we plan to extend our support to other CMSs and to validate the tool in industrial settings.

Acknowledgements

References

1. Abdelhedi, F., Ait Brahim, A., Atigui, F., Zurfluh, G.: MDA-Based approach for NoSQL databases modelling. In: Bellatreche, L., Chakravarthy, S. (eds.) Big Data Analytics and Knowledge Discovery. pp. 88–102. Springer International Publishing (2017)
2. Belo, O., Cunha, J., Femandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: QuerySheet: A bidirectional query environment for model-driven spreadsheets. In: IEEE Symposium on Visual Languages and Human Centric Computing - VL/HCC. pp. 199–200 (2013)
3. BuildWith.com: CMS Usage Distribution in the Top 1 Million Sites (2021), https://trends.builtwith.com/cms, accessed Jan 2022
4. Cabot, J.: Wordpress: A content management system to democratize publishing. IEEE Software **35**(3), 89–92 (2018)
5. Comyn-Wattiau, I., Akoka, J.: Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j. In: International Conference on Big Data (Big Data). pp. 453–458. IEEE (2017)
6. Cánovas Izquierdo, J.L., Cabot, J.: JSONDiscoverer: Visualizing the schema lurking behind JSON documents. Knowledge-Based Systems **103**, 52–55 (2016)
7. Daniel, G., Gómez, A., Cabot, J.: UMLto[No]SQL: Mapping conceptual schemas to heterogeneous datastores. In: 2019 13th International Conference on Research Challenges in Information Science (RCIS). pp. 1–13 (2019)
8. Daniel, G., Sunyé, G., Cabot, J.: UMLtoGraphDB: mapping conceptual schemas to graph databases. In: International Conference on Conceptual Modeling. pp. 430–444. Springer (2016)
9. Ed-douibi, H., Cánovas Izquierdo, J.L., Bordeleau, F., Cabot, J.: WAPIml: Towards a Modeling Infrastructure for Web APIs. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 748–752 (2019)
10. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
11. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering. p. 471–480. ICSE '11, Association for Computing Machinery (2011)

12. JSON API Initiative: JSON:API (2020), https://jsonapi.org, accessed Jan 2022
13. KNIME AG: KNIME's home page (2022), https://www.knime.com, accessed Jan 2022
14. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice. Softw. Syst. Model. **17**(1), 91–113 (Feb 2018)
15. Martínez, S., Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Cabot, J.: Towards an access-control metamodel for web content management systems. In: Sheng, Q.Z., Kjeldskov, J. (eds.) Current Trends in Web Engineering. pp. 148–155. Springer International Publishing (2013)
16. Nguyen Phuoc Bao, H., Clavel, M.: Ocl2psql: An OCL-to-SQL Code-Generator for Model-Driven Engineering. In: Dang, T.K., Küng, J., Takizawa, M., Bui, S.H. (eds.) Future Data and Security Engineering. pp. 185–203. Springer International Publishing (2019)
17. OpenApi initiative: Specification (2020), https://swagger.io/specification, accessed Jan 2022
18. Priefer, D., Rost, W., Strüber, D., Taentzer, G., Kneisel, P.: Applying MDD in the content management system domain. Software and Systems Modeling **20**(6), 1919–1943 (Dec 2021)
19. Richters, M., Gogolla, M.: OCL: Syntax, semantics, and tools. In: Object Modeling with the OCL, pp. 42–68. Springer (2002)
20. Rivero, J.M., Heil, S., Grigera, J., Robles Luna, E., Gaedke, M.: An extensible, model-driven and end-user centric approach for API building. In: Casteleyn, S., Rossi, G., Winckler, M. (eds.) Web Engineering. pp. 494–497. Springer International Publishing (2014)
21. Saraiva, J.d.S., Silva, A.R.d.: Development of CMS-based web-applications using a model-driven approach. In: 2009 Fourth International Conference on Software Engineering Advances. pp. 500–505 (2009)
22. Souer, J., Kupers, T., Helms, R., Brinkkemper, S.: Model-driven web engineering for the automated configuration of web content management systems. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) Web Engineering. pp. 121–135. Springer (2009)
23. Stateless group: HAL - Hypertext Application Language (2018), https://stateless.group/hal_specification, accessed Jan 2022
24. The Eclipse Foundation: Eclipse Modeling Project - Eclipse Modeling Framework - Home (2021), www.eclipse.org/emf/, accessed Sep 2021
25. The Eclipse Foundation: Xtend (2021), www.eclipse.org/xtend, accessed Sep 2021
26. Trias, F., de Castro, V., Lopez-Sanz, M., Marcos, E.: Migrating traditional web applications to CMS-based web applications. Electronic Notes in Theoretical Computer Science **314**, 23–44 (2015)
27. Trujillo, J., Luján-Mora, S.: A uml based approach for modeling etl processes in data warehouses. In: International Conference on Conceptual Modeling. pp. 307–320. Springer (2003)
28. W3C: Usage statistics of content management systems (2021), https://w3techs.com/technologies/overview/content_management, accessed Jan 2022
29. Xhevi Qafmolla, Viet Cuong Nguyen, K.R.: Metamodel-based generation of web content management systems. International Journal on Information Technologies & Security **6**(4), 17–30 (2014)