External Representations of Semantically Rich Content in Complex Systems Using Loose Coupling

Hans-Werner Sehring
Nordakademie
Elmshorn, Germany
e-mail: sehring@nordakademie.de

***Abstract*—Content-based software systems like websites and online shops are based on multiple components that collaborate in different ways while serving users. In recent years, the typical architecture of solutions centered around CMSs changed from monolithic to loosely coupled systems. Current approaches are called "composable architecture" or "composable commerce" because they focus on substitutability of components that provide a specific service. Data exchange between system components takes place in an external format that conforms to a system-wide agreed schema. Content Management Systems (CMSs) are one central component of a content-based system. CMSs manage and publish meaningful content. Such content is represented by data, but it is not processed under fixed semantics. However, collaborating systems require a consistent interpretation of data as content on both ends in order to preserve meaning. We argue that such a consistent interpretation requires mappings between the content models underlying CMSs and the data models that are used for communication, and that these mappings, therefore, must be shared by all components of the system. In order to justify this claim, we compare the expressiveness of plain data formats with that of content modeling languages, and we study mappings between them. In this paper, we use JSON and JSON Schema as typical examples of external data representations. We discuss content models using the example of the Minimalistic Meta Modeling Language (M³L). Our initial research shows that schemas for data exchange should be tightly linked to content models in order to not only represent content as data, but also to allow for consistent interpretations of content.**

Keywords*—*content model; data schema; schema mapping

I. INTRODUCTION

Content Management Systems (*CMSs*) are an established tool for (in particular online) content publication. They are software systems that incorporate various functions for content creation, editing, management, (automated) document creation based on layouts, and document delivery. Over time, many CMS products started integrating additional functionality to keep up with emerging requirements. At the same time, such products became increasingly complex because many of them incorporate new functions in a monolithic way.

Since they often provide a comprehensive software infrastructure comparable to an application server, many content management solutions are built using a CMS as a platform. Custom code is integrated into the CMS, making the overall solution an even larger monolith. This approach is often suitable for purely content-based functionality.

In recent years, an opposite trend has taken hold under the name *headless CMS*. Such CMSs basically focus on basic content creation, editing and management functions. Content is published via a delivery service that makes "pure" content accessible in the form of *Application Programming Interfaces* (*APIs*). All additional services are provided by separate software components. This includes document preparation and delivery that is implemented outside of a headless CMS. Components are combined using a *composable architecture*, also called *composable commerce*.

Though the idea of using simple interfaces based on a data exchange format is appealing, it constitutes an "impedance mismatch" with rich content structures as employed by capable CMSs. Ideally, a CMS provides various means of structuring content. Many allow defining a *schema* or *content model*. Such a schema is, on the one hand, used to provide type safety to functions handling content, and on the other it constitutes the basis to capture the meaning of content. To make use of structure and meaning assigned to content, content structure and semantics defined by content models need to be preserved in external representations, and they are used as a basis to map content to an external form.

In this article, we argue that expressive content models are required to globally describe the meaning of content so that content is correctly represented as data and data is interpreted as content. The discussion in this article extends the presentation in [1].

With the landscape of digital communication solutions becoming more complex, there is an increasing number of services that integrate data and services for other entities than structured content – media files, customer data, product data, etc. The services need to interface with CMS solutions. Systems typically require application-specific integrations (see, for example, [2]). These integrations make systems that rely on data exchange with a centralized monolithic platform overly complex since they have to deal with a variety of data exchange formats and different entity lifecycles.

We discuss complex content-based systems and the various perspectives on content in such systems in Section II.

Systems that incorporate CMSs using APIs typically are built following microservice architectures. These consist of multiple services that provide one functionality each, with the CMS providing content as one of those services. System properties are established by service orchestration in the overall architecture.

APIs for access to content consist of service signatures and of structured content representations that are used as input and output parameters. Content representations typically focus on

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

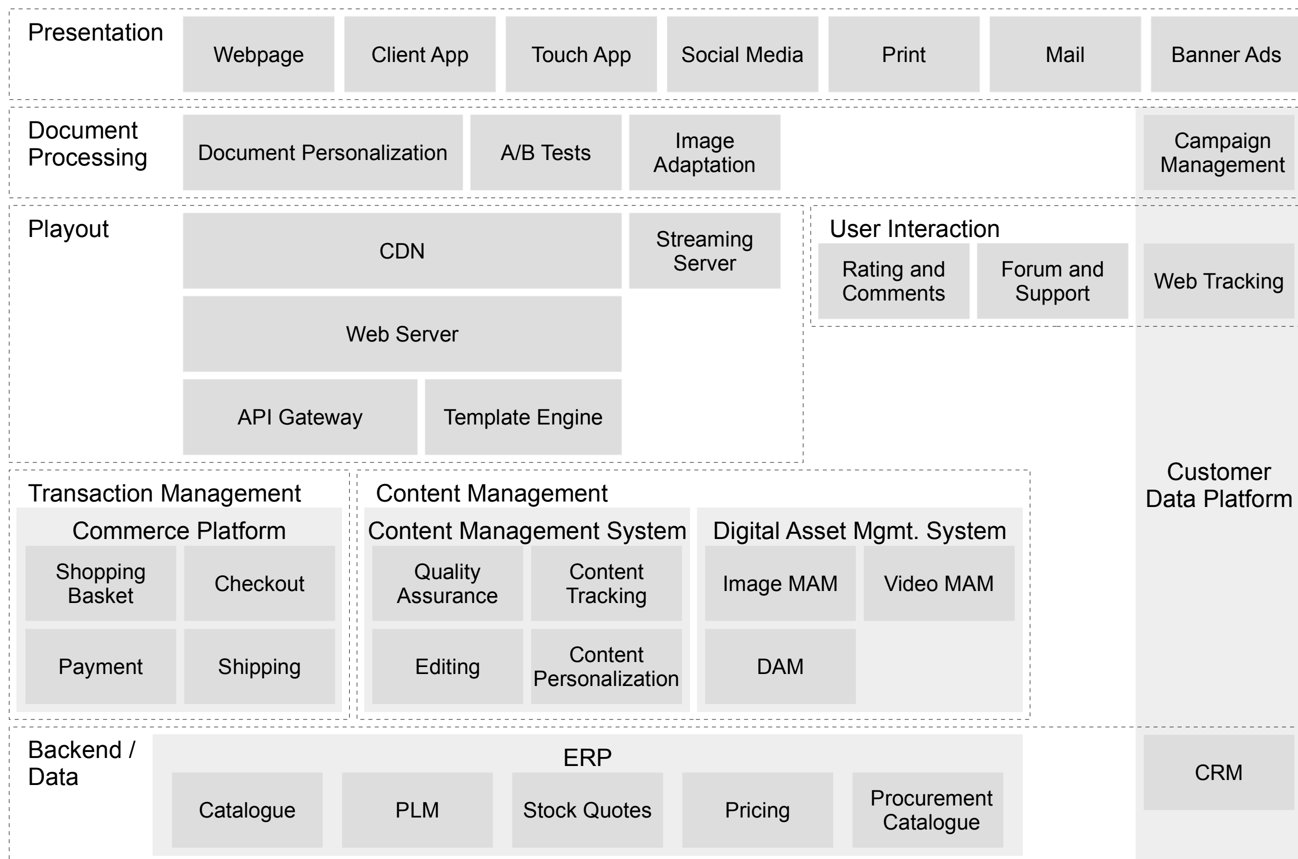

Figure 1. High-level Architecture [3].

structured content – mainly textual content and descriptions of unstructured content. Unstructured content, be it provided by a CMS or a Digital Asset Management system, is typically transferred in some binary format.

RESTful APIs are a current de-facto standard for communication between distributed CMS components. The *JavaScript Object Notation* (*JSON*) is the usual language chosen to represent (structured) content. Section III names typical aspects of APIs defined this way.

In Section IV, we introduce the Minimalistic Meta Modeling Language (M³L) as an example of a rather powerful content modeling language.

We use the M³L's capabilities for binding to external representations to study some aspects of interfaces for content access and interchange. In particular, we demonstrate different cases of JSON generation and parsing in Section V and discuss general differences of custom generated JSON and such generated by content models formulated in the M³L in Section VI.

We conclude the paper in Section VII.

II. Interfaces in Content-Based Systems

Content in the content management sense is found in a class of applications that enables digital communication. In a commercial setting, it primarily addresses the communication between companies and their customers. These interact using marketing websites, online shops, customer support systems, mobile apps, and similar mass communication facilities.

Content used to be of central interest because it provides the language of companies used in mass communication. This includes pragmatics, tonality, etc. With a tighter integration of communication means on digital communication platforms, other entities received equal interest, first and foremost customers (or, more precisely: the relationship of customers to companies).

With growing digital communication platforms, the need to exchange content between system components becomes ever more immanent.

A. *Content-Based Information Systems Architecture*

Different communications channels that are centered around customers and content are often integrated in growing content-based information systems. The increasing number of system components that contribute to such platforms and the additional need of integrating content and content-centric processes pose a challenge that companies have to master today.

Single communication requirements are met by specific systems, and new approaches and systems continuously emerge. To guide the creation of information systems from such base systems, reference architectures have been formulated.

Figure 1 outlines such an architecture for a website infrastructure that is typical for online shops, for example.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The architecture introduces layers in which customer-facing systems are placed on the top of the figure and internal, more technical components are located at the bottom.

The dotted lines in the figure are used to show functional clusters of components that in concert provide some service.

The *presentation* area on the top of the figure lists some (digital) communication channels. A company's website is often at the core of a communication strategy. Other channels may co-exist, but at best the channels work together in an omni-channel strategy. For example, a newsletter sent via mail may hint users to new content on the website, or banner ads redirect users to the website.

The basic *content management* functionality is located at the center of Figure 1. Besides CMSs there are digital asset management (DAM) and multimedia asset management (MAM) systems located in this functional area.

Content is delivered on the communication channels by creating and distributing documents with that content during *playout*. Typically, documents are generated from templates into which content is filled in. Webservers distribute the documents via a caching layer.

Even though the preparation of content for document creation is a content management task, documents are post-processed in some cases. *Document processing* functions often are Cloud services that leverage the core systems from optimization tasks.

Besides content management, a *transaction management* area drives processes like sales and logistics. These are different in nature from content-based processes, but the two closely interact.

Other means of *user interaction* allow users to provide own content, not only to consume the existing one. User-generated content calls for specific handling that differs from content management. User interaction also is a very valuable source of information on the users.

At the core of a content-based system, some data-driven *backend* services provide the backbone for all business processes.

Inside the dotted areas, there are components that provide single functions each. In component architectures, the components are typically realized by system products. For example, the area of the content management system is implemented by one CMS. Often, it will also incorporate functionality of the *playout* and the *document processing* area. In (micro) services architectures, it is more likely that single functions will be realized by distinct services. Then the larger functional clusters are implemented by orchestrated services.

B. Content Semantics

Content is a term for which there is no uniform definition. In this article, we will not provide a formal definition either. But we assume a certain notion of the term content as a basis, and we distinguish content from data.

For the purpose of the discussion in this article, data is meant to be any formal, digital representation used for storage, transmission, and processing.

Content may be represented by data, but it is something valuable and purposeful, where the purpose is some aspect of digital communication between system users.

Content is also distinguished from documents which are presentations of content. Documents are also determined by a layout and other visual properties that define how content is represented. Depending on the application, the dividing line may vary. For example, the layout of the text of a news article may be a presentation issue, where the position of text in a figure carries semantics.

With these notions, content conveys something people want to communicate about. It is about domain entities that are the subject of some communication. The information on and views of the domain entities may vary and may be subjective. Still, content is meant to capture domain semantics attached to it.

Content follows the semantics of the application domain, data a formal semantics. Both are required in communication. For example, products and services are of central importance on commercial websites. They are described by both content that represents people's views and by data that provides objective facts and figures. While content needs to be interpreted by people with similar views, data can formally be interpreted based on the semantics of some formal domain like mathematics. Product descriptions on commercial websites exhibit this twofold description feature. Marketing texts, legal texts, etc. are provided as content. Product data, like price, physical properties, shipping information, etc. is described by data.

Similar to a data model that conforms to a data schema, a content model is an abstraction of a content set. A content model serves two purposes. Like a data schema, it describes how content is formed. But it also captures some domain semantics, for example, by describing how certain domain entities are meant to be represented in content.

C. Content Management Interfaces

Websites and mobile apps are an important user interface to information sources, and they constitute a means for companies to get in touch with their customers. These applications are based on content that is presented to users in a suitable form.

Contemporary implementations of CMSs, online shops, campaign management solutions, and other content-based systems deal with content in various places: databases, application code, user interfaces, remote calls, URL formatting, HTTP request handling with content lookup and caching, tracking, targeting, campaign attribution, and many more. Figure 2 gives a rough overview.

Technically, content is stored, processed, and transferred as data. The multitude of content applications is reflected in diverse interfaces between the different components that together form a content-based system. Different kinds of interfaces are in place.

Editors collaborate on centrally stored content. To support this collaboration, instances of the content editor tool use a synchronous interface to the central CMS. It is characterized

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

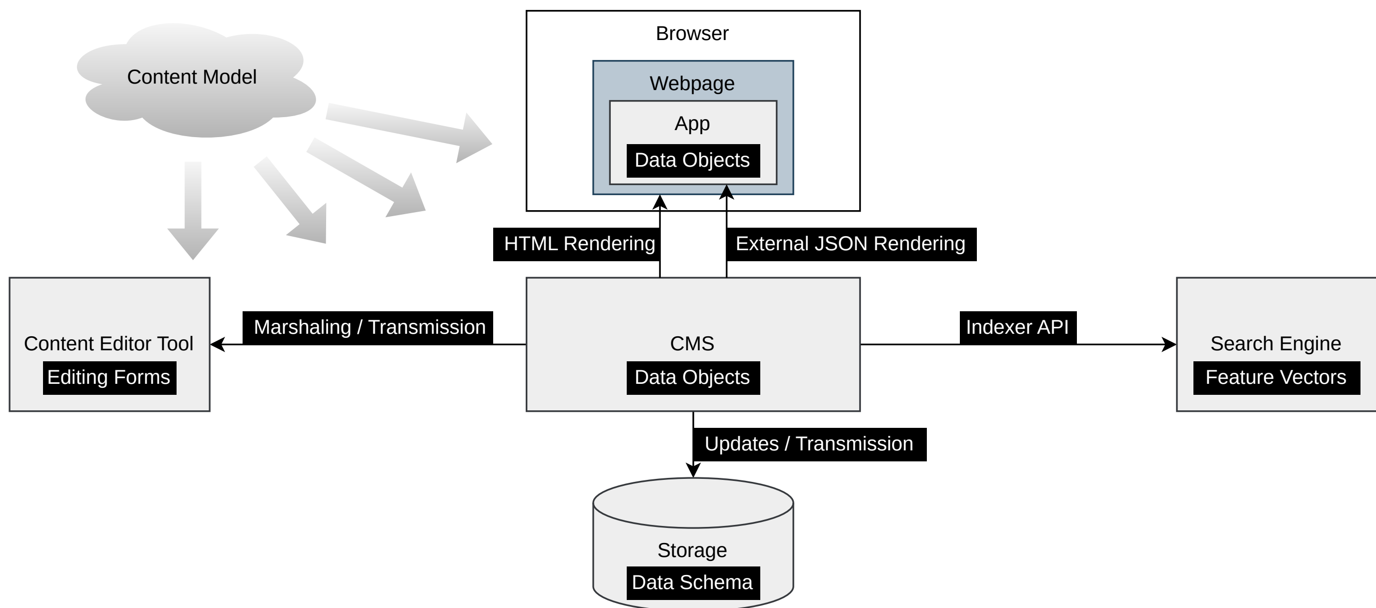

Figure 2. Content model references in a typical content management system.

by loose coupling with message passing, so that editors are notified about changes, and concurrent work is enabled by locking strategies.

Tightly coupled systems like the search engine, on the other hand, exchange copies of content. A search index is built up from content that is analyzed according to its structure.

For publication, documents are created from content. For example, web pages are created to publish content on the web. In particular in this scenario, document generation is called content rendering. It differs depending on the kind of generation used. Server-side rendering, where documents are created by the service provider, works in a synchronous fashion, because documents have to be completely generated before they can be shipped. Client-side rendering, on the other hand, may improve user experience by dynamically loading and rendering content in an asynchronous fashion. Hybrid approaches are usually in place today, and these need both kinds of interfaces to the central CMS.

Each of the indicated functionalities is related to the underlying content model, and all share a common notion of both this model and all content constellations. Or, viewed the other way round, a content model defines multiple interfaces that are consumed by different audiences, for example in a CMS:

- editors that are guided by the editing tool when entering content into forms, and that are supported during quality assurance of webpages,
- an editor-in-chief sees content before publication during quality assurance,
- compliance officers also receive pre-final presentations plus differences from previous versions to conduct legal and compliance reviews,
- application programmers that customize the CMS (services, editor, search engine),
- application programmers that develop client-side apps (JavaScript apps, mobile apps),
- template programmers that implement the rendering of content into documents,
- search engine optimization (SEO) managers who look after content descriptions, and
- last but not least users who think in content categories (product, contact, etc.) while browsing and reading, and who may use a refined search function.

In fact, each of the roles uses more than one interface, and all need to agree on the conceptual content model in order to use and serve the interfaces correctly. In particular, content needs to be encoded as data, and data need to be interpreted as content. Figure 2 names some of the technical interfaces where this applies.

A common content model is not only required for technical reasons of using APIs. Also, there needs to be a globally agreed domain semantics of content as argued in the previous section. For example, an editor maintains the content that is included in documents. For the user of the system to perceive what the editor had in mind, the editor's interface, the schema for content storage, and the APIs used by template programmers all have to be in line with a common understanding of content semantics.

Therefore, for coherence in content-based applications, there is a need for central models that are consistently implemented, or schemas and code for such systems are generated from such models. The upper left of Figure 2 symbolizes this.

III. Standards: RESTful APIs, JSON, and GraphQL

Approaches for (remote) APIs and their implementations are of general interest since the advent of distributed systems. After a series of technological approaches, a current de-facto

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

standard for online interfaces of CMSs has emerged from REST, JSON, and GraphQL.

A. RESTful APIs

Representational State Transfer (REST) was proposed by Fielding as the principle of communication in the Internet [4]. It calls for stateless servers and clients that handle state between requests. In conjunction with URLs that represent services calls, the definition of so-called RESTful APIs allows defining simple APIs for Web-based services.

Such APIs consist of service call signatures composed of an HTTP method and a URL that specify the service to be used and the input parameters. The response to a service call is a regular HTTP response. A typical response format for structured data is JSON as discussed in the subsequent section.

RESTful APIs can be implemented with existing Web technologies, for example, typical software libraries available for all relevant programming languages and existing software components to build a service infrastructure.

B. Content Interchange with JSON

JSON is an object language for JavaScript, allowing to formulate JavaScript object instances, where *instance* refers to data contained in object properties, not any internal object state. It can be used for data storage, transmission, and aggregation.

JSON is typically used as a response format of RESTful interfaces. It provides a simple means of structuring data with typical collection types, and encoding of data as character strings.

Most API-based CMSs use JSON to distribute content. They typically do so by representing content in a straight-forward manner using the structuring means and primitive data types of JSON.

Internally, CMSs allow the definition of content models that describe content. Such models are the basis for describing content, for content editing, and also for JSON generation.

Depending on the kind of CMS, such content models are more expressive than data models in JSON. In structured CMSs, content models are used to capture domain semantics attached to content. In contrast, *Digital Experience Platforms* (*DXPs*) focus on associating content with visual layouts. Content models describe visual building blocks in this case.

Document rendering presents content in visible form for consumption, for example, in the form of HTML files. In API-based CMS solutions, rendering is performed by external rendering engines (on client-side or on server-side). The rendering process is driven by *templates* that define how to layout content. Template code makes use of knowledge about the meaning of content to be represented, either the domain semantics (CMS) or the kind of visual building block (DXP).

The external form of content in a JSON representation is rather generic, though. JSON can be generated in an application-specific form, but basically contains structured data. The representation of content in JSON and interpretation from that format rely on consistent code on both producer's and consumer's side. Such interpretation cannot rely on JSON representations of content alone.

C. JSON Schema Languages

JSON as a format is appealing because of its simplicity combined with reasonable expressiveness. It was defined merely for the description of single records of data. Many applications call for a schema, though, that describes how classes of data are structured.

Several schema languages have been defined for JSON, most prominently *JSON Schema* [5]. Another proposal for a JSON schema language is JSound [6]. Other approaches are Joi [7] for JavaScript applications and Mongoose [8] for configurations of the database system MongoDB.

In this paper, we use JSON Schema for the discussion of schema properties.

D. GraphQL

GraphQL is a query language for structured data. Queries select data from a database, and they describe a JSON format in which the response to a query is expected.

JSON-based APIs often face the problems of "underfetching" (there is no call that delivers all data required in a situation so that a sequence of related remote calls is required) and "overfetching" (a service delivers too much data in a situation, so that too much data is transmitted and a client has the task of selecting the required data).

Using GraphQL as a service interface language helps avoiding these problems. Clients can select the exact dataset they need in a specific situation. On the other hand, GraphQL is dynamic by nature leading to more computation and less options for caching of results.

IV. A Short Introduction into the Minimalistic Meta Modeling Language (M³L)

For the discussion in this paper, we use the M³L since it proved to be a suitable language for the modeling of various aspects of content management. To this end, we briefly introduce the M³L, and we present some exemplary base models for content management and for content interchange based on RESTful APIs.

A. A Short Introduction into the M³L

In this section, we briefly introduce the M³L by highlighting those features that are central to the underlying experiments.

The basic M³L statements are:

- `A`: the declaration of or reference to a *concept* named *A*
- `A is a B`: refinement of a concept *B* to a concept *A*. *A* is a *specialization* of *B*, *B* is a *generalization* of *A*.
- `A is a B { C }`: containment of concepts. *C* belongs to the *content* of *A*, *A* is the *context* of *C*.
- `A |= D`: the *semantic rule* of a concept. Whenever *A* is referenced, actually *D* is bound. If *D* does no exist, it is created in the same context as *A*.
- `A |- E F G.`: the *syntactic rule* of a concept that defines how a string is produced from a concept, respectively how a concept is recognized from a string.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

When the representation of *A* is requested, it is produced by a concatenation of the strings produced out of *E*, *F*, and *G*. When no syntactic rule is defined, a concept is represented by its name. Vice versa, an input that constitutes the name of a concept without a syntactic rule leads to that concept being recognized.

If a concept that is referenced by one of the statements exists or if an equivalent concepts exists, then this one is bound. Otherwise, the concept is created as defined by the statement.

Existing concepts can be redefined. For example, with the definitions above, a statement

```
A is an H { C is the I }
```

redefines *A* to have another generalization *H* and *C* (in the context of *A*) to have *I* as its only generalization.

Every context constitutes a *scope*. A redefinition of a concept in a context is only applied in that context. When a redefinition of a concept takes place in another context as the original definition, we call that redefinition a *derivation*.

The concepts that are defined by such statements are evaluated when used. Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, or to its *narrowing*. A concept *B* is a narrowing of a concept *A* if

- *A* evaluates to *B* through specializations or semantic rules, and
- the whole content of *A* narrows down to content of *B*.

To evaluate a concept, syntactic rules and narrowing are applied repeatedly.

Given the sample M³L statements:

```
Person { Name is a String }
PersonMary is a Person { Mary is the Name}
PersonPeter is a Person {Peter is the Name
                         42    is the Age
}
```

the result of an additional statement

```
Person { Peter is the Name 42 is the Age}
```

is *PersonPeter* since *PersonPeter* is specialization of *Person* and its whole content matches. The statement

```
Person { Mary is the Name 42 is the Age }
```

is not evaluated further. It does not match *PersonPeter* since *Name* has a different specialization, and it does not match *PersonMary* since that has no matching content concept called *Age* or *42*.

B. Basic Content Management and Document Rendering

The M³L is universal and has many applications. Among other modeling tasks, it has proven useful to describe content as lined out in, for example, [9]. This applies both to content models as well as content items since the M³L does not distinguish model layers, such as type and instance.

For example, with a content model like:

```
Article is a Content {
  Title is a String
  Text  is a FormattedString }
```

according content can be created:

```
NewsArticle123 is an Article {
  "Breaking News"            is the Title
  "This is a report on ..."  is the Text }
```

For textual formats, like HTML and JSON, documents can be rendered from content through syntactic rules of content as introduced in the previous subsection. On the level of the content model, syntactic rules describe document templates, on the content item level they render single document instances.

For the sample content definitions above, a JSON template for a piece of content of type *Article* may look like:

```
Article |- "{\"title\":\"" Title
           "\",\"text\":\"" Text "\"}"
```

This syntactic rule produces JSON output for the concept *NewsArticle123* from above:

```
{"title":"Breaking News","text":"This is a
report on ..."}
```

The syntactic rule defines a JSON structure into which the concepts from the content are integrated. These may themselves evaluate to content strings of embedded JSON structures.

Please note that, for example, *{\"title\":\"* is a valid concept name, as is *\"}*. Since new concepts are declared the first time they are referenced, and because they syntactically evaluate to their name by default, they can be used like string literals. The concept name *\"* is an escape sequence for the quote character (not a quote sign for identifiers).

V. Producing JSON using the M³L

As outlined in the preceding section, the M³L can serve as an example of a an expressive content modeling language. For API-driven content distribution, structured content needs to be represented in an external form. In state-of-the-art services, this external form is JSON.

The same holds for JSON generation. JSON Schema allows defining valid forms of JSON structures so that content can be transferred in a reliable manner. It is not expressive enough by itself, however, to recover equivalent content on the receiver's side. Custom code is required to generate JSON out of rich content structures. Appropriate code that shares the same conception of content is required to interpret JSON data.

Schema design for JSON requires careful consideration. Even finding sample instances for a given schema is a non-trivial task since semantics is scattered over a set of definitions and constraints [10].

JSON Schema provides various ways of defining and relating schemas. There are multiple ways of expressing equivalent schemas and equivalence cannot generally be proven [11].

One way of sharing content concepts between sender and receiver is to have a common content model and mappings to and from external representations. We exemplify this by utilizing the capabilities of the M³L for some sample constructs.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A. Defining Lexical Rules for JSON

M³L's lexical rules can produce JSON code out of concepts as sketched in Section IV-B.

The M³L does not distinguish between types and instances of content, let alone other incarnations (materializations, metatypes, etc.). This distinction is, however, required in classical approaches as JSON and JSON Schema.

In addition to the above sample rules that generate JSON, the lexical rules of other concepts may produce JSON Schema. See the following simple rules for the content example. A M³L definition:

```
Article |-
"{\"type\":\"object\",\"properties\":{\""
   Title "\":{\"type\":\"string\"},"
   Text "\":{\"type\":\"string\"}}}"
```

results in the production of the following JSON Schema definition:

```
{ "type": "object",
  "properties": {
    "Title": { "type": "string" },
    "Text":  { "type": "string" }}}
```

Lexical rules for both JSON and JSON Schema require to distinguish between schema and instances. Contextual definitions allow defining both layers for a concept. The decision between schema and instance has to be made explicitly, for example, by providing separate sets of syntactic rules in different contexts:

```
SchemaRules { Article |- ... }
InstanceRules { Article |- ... }
```

The distinction between schema and instance level is atypical for M³L applications. Usually, concepts my play both roles. Therefore, it is possible that the same concept will sometimes be represented by JSON (as an instance) and sometimes as JSON Schema (when it contributes to the structure of another concept)

In any case, a fair amount of extra code is required to state the obvious lexical rules per concept. It is approximately the same effort like providing custom mappings in software.

The effort of mapping an internal content model to its external forms is beneficial, though, to be able to recover the semantics of content. This way, schema definitions contribute to the exchange of meaningful content. In the subsequent subsections, we compare the modeling capabilities of JSON Schema and the M³L for the generation of JSON representations of content.

B. Basic Model Mapping from M³L to JSON

Simple M³L expressions that represent content instances can be expressed in a straight-forward manner as outlined by the content example. Some information is lost in the JSON representation, though. In the example above, the concept name *Article* is not communicated.

Such concept information may be reflected in dedicated properties. But more information on the content is lost if we add content types and descriptions, for example in M³L:

```
Person {
  FirstName is a String
  LastName  is a String
  Address }
Address {
  Street is a String
  City   is a String }
JohnSmith is a Person {
  John                is the FirstName
  Smith               is the LastName
  JohnSmithsAddress is the Address {
    "Main Street" is the Street
    Lincolnshire  is the City } }
```

Syntactic rules may produce the following JSON:

```
{ "FirstName": "John",
  "LastName": "Smith",
  "Address": { "Street": "Main Street",
               "City": "Lincolnshire" } }
```

The intended data structure can be defined by means of JSON schema that is also generated from the content concepts, for example, as follows:

```
{"title": "Person",
 "type": "object",
 "properties": {
  "FirstName": {"type": "string"},
  "LastName": {"type": "string"},
  "Address": {"$ref":"#/$defs/Address"} },
 "$defs": {
  "Address": {
   "type": "object",
   "properties": {
    "Street": {"type": "string"},
    "City": {"type": "string"} } } } }
```

Here, the concept names *Person* and *JohnSmith* are not present in JSON. The content name *JohnSmithsAddress* is also missing; the "type" name *Address* is used instead.

Note that information is distributed over two structures, instance and schema, and declared in different languages. A JSON (instance) file does not make reference to the schema it is intended to comply with. Therefore, the matching schema has to be found by distinct means. Names – concept names in the case of the M³L – are not included in JSON, but are required for schema selection (*Person* in the above example).

Additional information that relates data to its schema has to be added on top of the standard data formats. One option is the use of a kind of envelope structure which adds type information to data, for example,

```
{ "JohnSmith": { "Firstname": "John",... },
  "type": "Person" }
```

For a second option we may introduce well-defined property names under which we give metadata as part of the data in a JSON structure. For example, assume that the property names *$name* and *$type* are defined to hold such metadata:

```
{ "$name": "JohnSmith",
  "$type": "Person",
  "Firstname": "John", ... }
```

In order to generate two external forms – JSON and JSON Schema – out of one integrated internal content representation, two lexical rules are required as mentioned in Section V-A. When parsing JSON on the receiver's side, the unrelated files need to be recombined in a content representation. JSON (Schema) provides no means to do so.

C. Capturing Type Variations

Variants of content are commonly found in CMSs since one schema alone typically does not cover all aspects under which content is used for communication. Few CMSs cover variations explicitly in content models. The M³L, however, allows reflecting variants by means of concept refinement and by contextualization.

Consider concepts modeled after an example from [12]:

```
Address {
  street_address" is a String
  city            is a String
  state           is a String
  Type }
BusinessAddress is an Address {
  Business is the Type
  Department is a String }
ResidentialAddress is an Address {
 Residential is the Type }
```

An example from [12] reflects the above M³L definitions:

```
{ "type": "object",
  "properties": {
    "street_address": {"type": "string"},
    "city": { "type": "string" },
    "state": { "type": "string" },
    "type": {
      "enum": ["residential", "business"]
    }
  },
  "required": ["street_address",
               "city", "state", "type"],
  "if": {
    "type": "object",
    "properties": {
      "type": { "const": "business" }
    },
    "required": ["type"]
  },
  "then": {
    "properties": {
      "department": { "type": "string" }
    }
  },
  "unevaluatedProperties": false }
```

In this example, an address has a type with possible values from the enumeration: *residential* and *business*.

The additional *department* property from the example above can be introduced conditionally using the *"if"..."then"..."else"* construct in JSON schema. This allows representing subtypes. The information that a *BusinessAddress* is an *Address* is lost, however, both on instance and schema level. Therefore, this is not a suitable representation of refinement that conveys semantics.

In the M³L, we have extra concepts for a *BusinessAddress* and a *ResidentialAddress*. Therefore, the *Type* is not actually needed as content. It could be generated into JSON from the concept information.

In fact, M³L would (also) work the other way round: if an *Address* with an extra *Department* is given, it is derived to be a *BusinessAddress*. The derived base type matching of the M³L – and a similar, but typically implicit behavior of typical CMS applications – makes matching JSON data to JSON Schema definitions yet more difficult.

The M³L can make those type variations explicit, though. In the above example, we can assign a *Type* property as required by JSON schema based on the presence of *Department* information:

```
Address {
  street_address is a String
  city           is a String
  state          is a String
} |= Address {
  street_address is the street_address
  city           is the city
  state          is the state
  Residential    is the Type }
BusinessAddress is an Address {
  Department is a String
} |= BusinessAddress {
  street_address is the street_address
  city           is the city
  state          is the state
  Business       is the Type }
ErrorneousAddress is an Address {
  Department  is a String
  Residential is the Type
} |= Error
```

This is not considered a typical M³L application, though.

VI. Comparison of Plain JSON and M³L Constructs

In contrast to typical data schemas, content models are not only concerned with constraints on values, references, and structure, but additionally try to capture some semantics. Furthermore, while data aims at representing one consistent state of entities, content deals with varying forms and utilizations used in communication: different communication scenarios, contexts of users who perceive content, language and other localizations, etc.

This section points out some of the differences in expressiveness of data schemas and content models using the examples of JSON schema and the M³L.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A. Subtypes

Type hierarchies allow intensional descriptions of schema elements and are, therefore, found in content models. They are not ubiquitous in data models, though. JSON schema does not feature subtyping.

JSON Schema does have means to express schema variants (*"if"*, *"dependentRequired"*) and to relate different schemas (*"dependentSchemas"*, *"allOf"*, *"anyOf"*, *"oneOf"*).

These can be used to model specializations of data as variants. An example is presented in Section V-C above.

Any forms of refinements (“subtypes”) in JSON weakens the constraints of a JSON Schema since not all properties can be *"required"* or *"additionalProperties"* and *"unevaluatedProperties"* must be allowed - very much as in the M³L.

The M³L as a notion of refinement that does not explicitly distinguish between subtyping and instantiation. In the following definitions, for example,

```
Employee is a Person {
 Salary is a Number }
JohnSmith is an Employee {
 John is the FirstName
 Smith is the LastName
 5000 is the Salary }
```

present a concept hierarchy where *Person* and *Employee* would presumably be handled on the schema level in external representations and *JohnSmith* at instance level. Then *Person* and *Employee* are in a subtype relationship with inheritance.

In JSON Schema, this would be expressed as:

```
{ "title": "Person",
  "type": "object",
  "properties": {
    "FirstName": { "type": "string" },
    "LastName": { "type": "string" },
    "Salary": { "type": "number" },
    "is_employee": { "type": "null" }
  },
  "required": ["FirstName", "LastName"],
  "dependentRequired": {
    "is_employee": ["Salary"]
  } }
```

Again, the *Employee* as a concept is not represented.

B. Single and Multi-valued Relationships

It is quite common in content models to be vague about arity. For example, some pieces of content may typically have a 1:1-relationship, making it unary in the content model. But there are exceptions of n-ary cases that also need to be covered. The M³L allows to define concepts with *is a* and *is the* to take this into account.

A typical data model would define an n-ary relationship, even though in most cases the data are 1:1.

JSON itself allows to easily vary between unary and n-ary properties by simply stating either *"a":"b"* or *"a":["b","c"]*. JSON Schema, though, needs to define arity or to define variations with *"if"..."then"..."else"*.

Consider as an example a person with two addresses:

```
Person {
  FirstName is a String
  LastName  is a String
  Address }
Address {
  Street is a String
  City is a String }
JohnSmith is an Employee {
  John               is the FirstName
  Smith              is the LastName
  JohnSmithsAddress is an Address {
    "Main Street" is the Street
    Lincolnshire  is the City }
  JohnSmithsOffice  is an Address {
    "High Street" is the Street
    Lincolnshire  is the City } }
```

A JSON structure reflecting this content is:

```
{ "FirstName": "John",
  "LastName": "Smith",
  "Address": [
    { "Street": "Main Street",
      "City": "Lincolnshire" },
    { "Street": "High Street",
      "City": "Lincolnshire" }
  ] }
```

Though this is a small change to the JSON structure, it has to be explicitly foreseen in JSON Schema. It is not as easy to vary between one or multiple addresses (in this example) as it is in content models like the M³L or the Java Content Repository [13].

C. Content Conversions and Computed Values

It is common for content models to not only contain content itself but also descriptive information about the content (sometimes referred to as metadata).

For example, a simple data property like

```
{"price": 42}
```

requires additional information to be interpreted correctly (the currency, for example). In simple data models, there is an additional documentation that establishes an agreement on how applications should deal with the data. The possibility to state the unit of measurement is typically found in *Product Information Management systems*.

In these cases, the information needs to be stated explicitly, as it is done in typical master data management systems:

```
{"price": {"value":42, "currency":"€"}}
```

Such a record allows a mutual understanding of the value. It prevents an easy mapping from JSON to a numeric *price* variable, though.

As a slight improvement, values should be replaced by named concepts. The M³L captures meaning by defining relevant concepts. For example, a concept like *EuroCurrency* as a refinement of a concept *Currency* would be used instead of the string value €.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

On top of descriptive information on content, a content model may also define a limited set of computational rules in order to define consistent arithmetics.

The M³L is expressive enough to define some (symbolic) computation. Assume, for example, a concept *Integer*, concrete “instance” concepts like *100*, and concepts describing computations like *FloatDivision*, the division of numeric values.

On the basis of such definitions, it is possible to state conversion rules like the following:

```
Price {
  Value is a FloatNumber
  Currency }
PriceInEuro is a Price {
  € is the Currency }
PriceInEuroCents is a Price {
  Value is an Integer
  Cents is the Currency }
|= PriceInEuro {
  Value is a FloatDivision {
    Value is the Dividend
    100   is the Divisor } }
```

These sample definitions describe (on schema level) how values are converted so that all clients using this model share the same arithmetics.

D. General Variations

The M³L provides extra flexibility by both considering (internal) definitions and (external) representations. Other approaches like CMSs handle these two aspects separately.

By overriding lexical rules in the M³L, the marshaling format can deviate from the schema. These rules are restricted to very simple grammar rules, though, in order to work both as producers as well as recognizers.

For example, a definition:

```
Person {
  firstName is a String
  lastName  is a String }
|- "{\"name\":\"" firstName " " lastName
   "\"}" .
```

combines first name and last name into one name field. When reading data, it will just break up the name value at the first whitespace.

The simple rules suffice in some situations. In very simple cases, they allow, for example, to adapt legacy JSON or to provide backwards compatibility to previous schema versions. They are not capable of full parsing of input data.

VII. Conclusion

We conclude with a summary and an outlook.

A. Summary

We compare rich content models – using the example of the modeling capabilities of the M³L – with typical data schemas, in particular JSON Schema. We conclude that models for meaningful content cannot adequately be expressed by data schemas alone.

JSON became a de-facto standard for content exchange. We present examples showing that the currently evolving schema language, JSON Schema, is not sufficient for content modeling in its current form.

B. Outlook

Additional research is required to identify the full expressiveness required to define external representations of content for modern content management approaches. This will guide future investigations towards a suitable set of modeling capabilities for marshaling formats.

The M³L is not intended to be a data schema language. Therefore, it lacks some features of such languages. It will be an experiment, though, to define a M³L derivative that is able to serve as an alternative schema language for JSON.

Acknowledgment

The author thanks numerous colleagues, partners, and clients for fruitful discussions on various topics centered around digital communication. The Nordakademie is acknowledged for the chance to continue work in the field.

References

[1] H.-W. Sehring, “On the Generation of External Representations of Semantically Rich Content for API-Driven Document Delivery in the Headless Approach,” Proceedings of the Fifteenth International Conference on Creative Content Technologies, CONTENT 2023, ThinkMind, 2023, pp. 17–22.

[2] H.-W. Sehring, “On the integration of lifecycles and processes for the management of structured and unstructured content: a practical perspective on content management systems integration architecture,” International Journal On Advances in Intelligent Systems, volume 9, numbers 3 and 4, pp. 363–376, 2016.

[3] H.-W. Sehring, “Architectural Considerations for the System Landscape of the Digital Transformation,” Proceedings of the Twelfth International Conference on Creative Content Technologies, CONTENT 2020, 2020, pp. 5–10.

[4] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Doctoral dissertation, University of California, 2000.

[5] A. Wright, H. Andrews, B. Hutton, and G. Dennis, “JSON Schema: A Media Type for Describing JSON Documents,” Internet Engineering Task Force, 2022.

[6] C. Andrei, D. Florescu, G. Fourny, J. Robie, and P. Velikhov, *JSound 2.0*. [Online] Available from: http://www.jsound-spec.org/publish/en-US/JSound/2.0/html-single/JSound/index.html. 2024.3.22

[7] *The most powerful schema description language and data validator for JavaScript*. [Online]. Available from: https://joi.dev/ [retrieved: May, 2023]

[8] *Mongoose*. [Online] Available from: https://mongoosejs.com/. 2024.3.22

[9] H.-W. Sehring, “On Integrated Models for Coherent Content Management and Document Dissemination,” Proceedings of the Thirteenth International Conference on Creative Content Technologies, CONTENT 2021, ThinkMind, 2021, pp. 6–11.

[10] L. Attouche et al., “A Tool for JSON Schema Witness Generation,” Proceedings of the 24th International Conference on Extending Database Technology. OpenProceedings.org, March 2021, pp. 694–697.

[11] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, “Finding Data Compatibility Bugs with JSON Subschema Checking,” Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021. Association for Computing Machinery, July 2021, pp. 620—632.

[12] M. Droettboom, “Understanding JSON schema,” Space Telescope Science Institute, 2023.

[13] D. Nuescheler et al., “Content Repository API for Java™ Technology Specification,” Java Specification Request 170, version 1.0, May 2005.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org