ON THE GENERATION OF EXTERNAL REPRESENTATIONS OF SEMANTICALLY RICH CONTENT FOR API-DRIVEN DOCUMENT DELIVERY IN THE HEADLESS APPROACH

Hans-Werner Sehring
June 26, 2023, to June 30, 2023 – Nice, Saint-Laurent-du-Var, France

// Agenda

1. Components of digital communication systems, in particular **Content Management Systems** (**CMSs**), are often loosely coupled.
2. For communication between CMS components, there are **standard protocols**. These are based on data representation languages. One prominent example is the **JavaScript Object Notation** (**JSON**).

   To avoid ambiguity when interpreting data as content, we claim that there needs to be a central content model and mappings to external representations.

1. We use the **Minimalistic Meta Modeling Language** (**M³L**) to analyze the potential of such central models.
2. Using M³L's syntactic rules, we define **mappings** and study **mismatches** between the model and JSON.
3. We conclude with a **summary and outlook**.

// Typical (Web) Content Management Systems

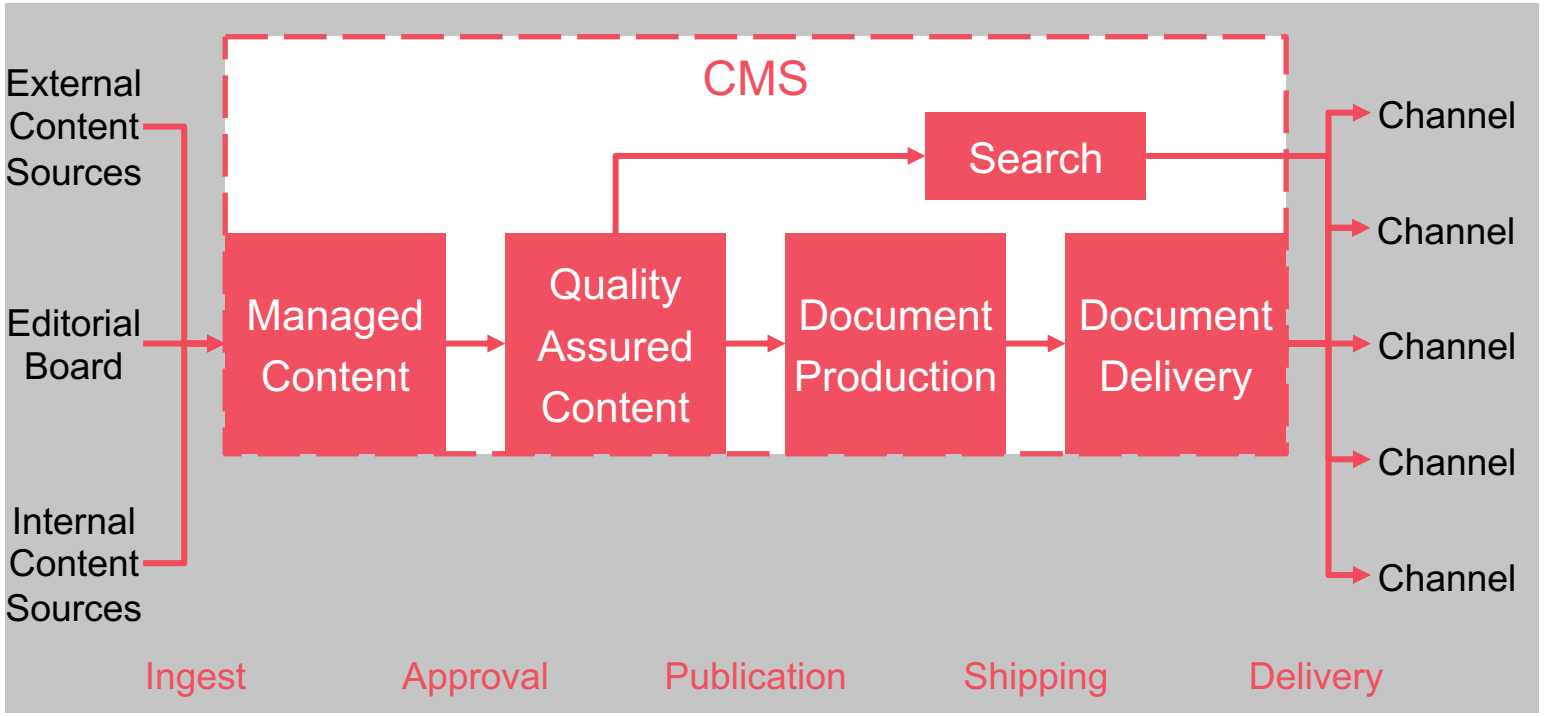
Until some years ago, content management systems supported all content-centric processes, from ingest to distribution.

The system landscape for digital communication contains multiple components for the various types of content (structured content, unstructured content, customer data, product data, etc.) and processes

For quite some time, monolithic CMS provided the basis for such systems

> Incorporate creation/ingest and editing of content, quality assurance processes as well as the creation of digital and distribution of digital documents

> Provide a platform for the integration of other systems and for custom business functionality



// Composable Architecture Based on Headless CMSs

Headless CMSs concentrate on content management functionality. They are decoupled from distribution through APIs. The same goes for other services.

In recent years, the architecture of the digital communication landscape changed to a composition of services

Well-engineered systems always based on a separation of content management and document delivery

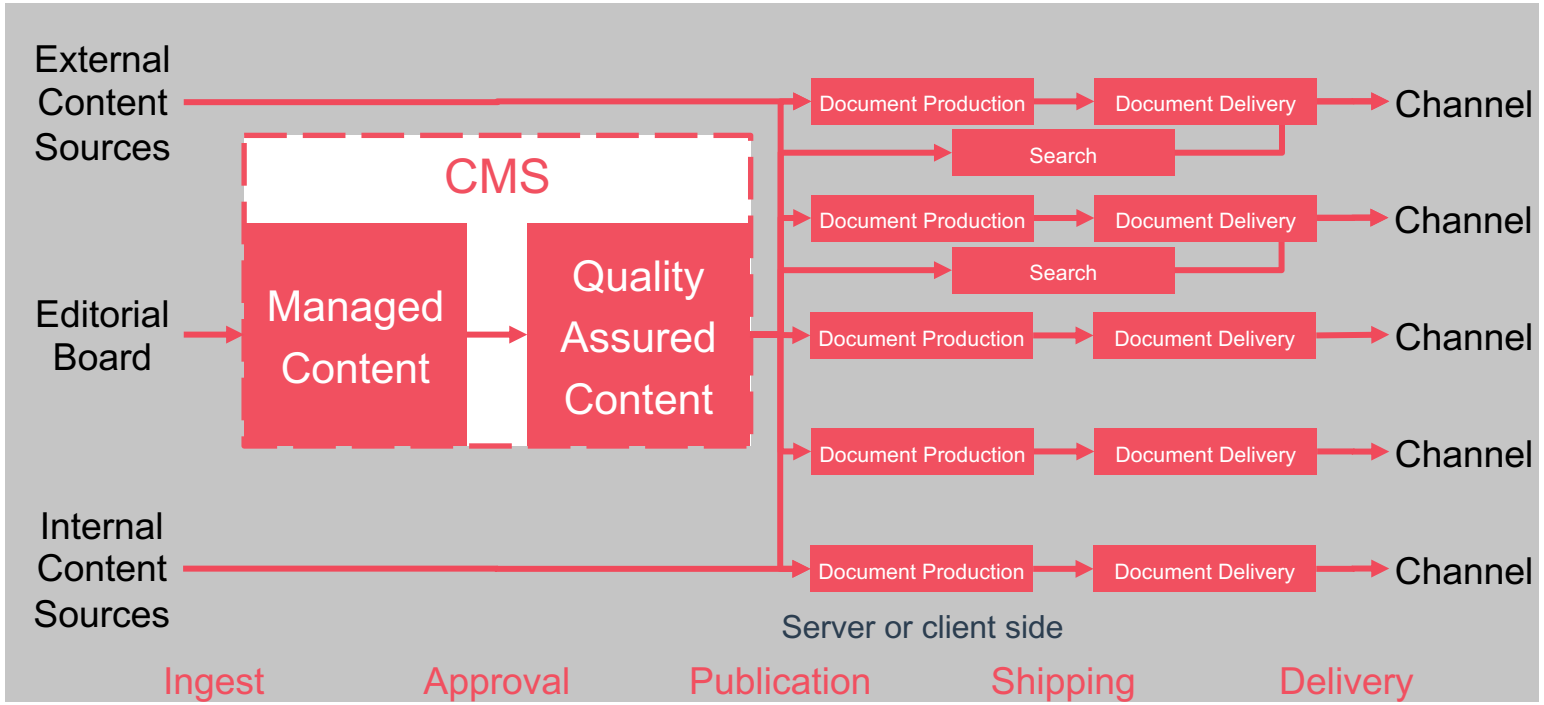
“Headless” CMSs:

> Separate components for content management and delivery

> Standardized communication between them

New chances:

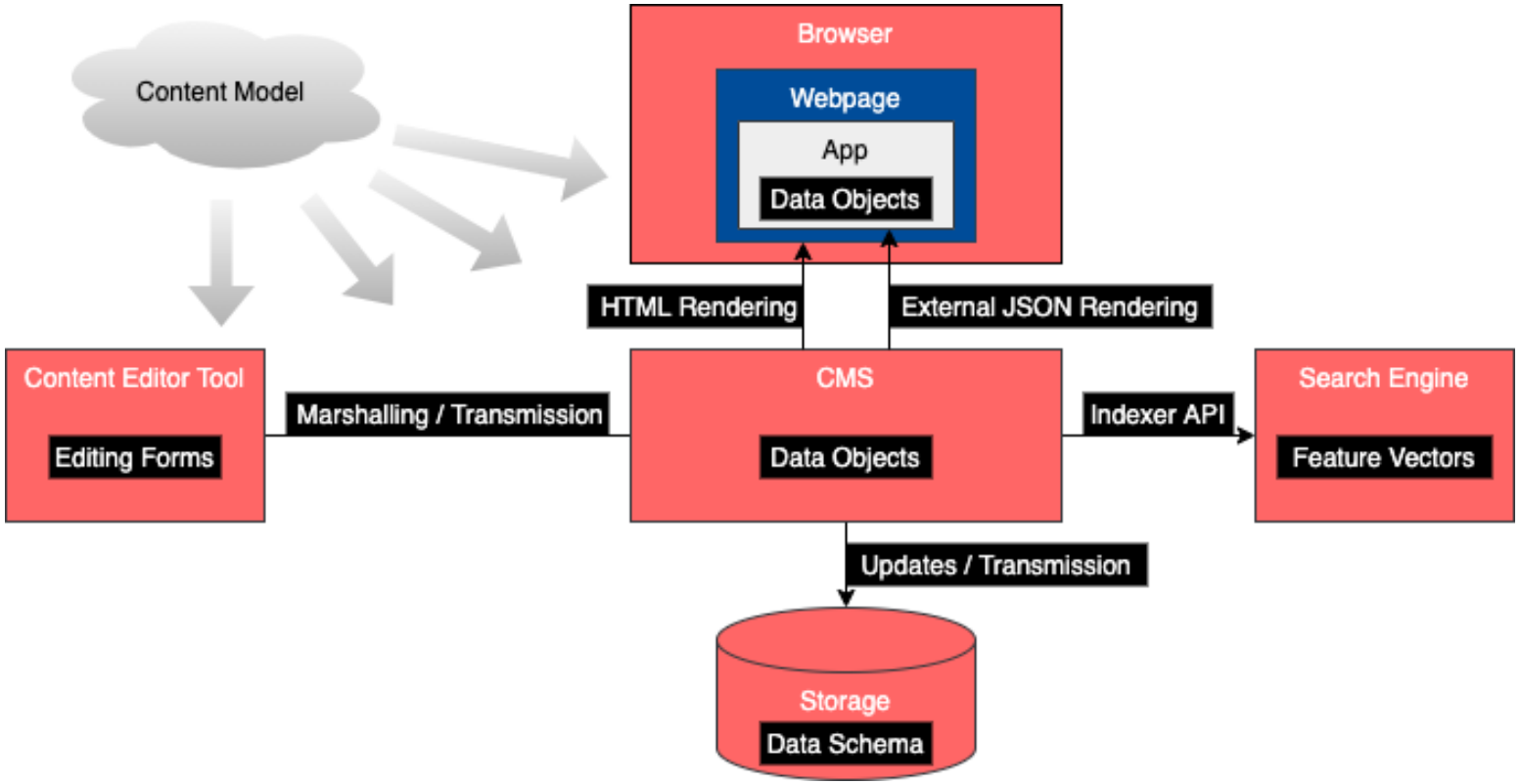
> Media agnostic content for omnichannel experiences



> “Composable architecture” (e.g., CMS and commerce system with same delivery)

// Encodings of content all over the place

Content management systems consist of various software components that use content representations in various forms



> All schemata, APIs, etc. must conform to the same underlying model

> In the case of a CMS, this is the content model

> It shows up in manifold form throughout a system

// Content Distribution Protocol: RESTful API, JSON, JSON Schema, GraphQL

> External representations of simple content

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

> Schema for the sample representation

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

> Request-dependent response formats through GraphQL queries

// Mismatch Between Data Schemas and Content Models

External content representations – the marshalling format of content – are representing content and are interpreted as content.

Content models allow rich modeling

> Domain semantics of content (articles with different kinds of markup, product descriptions, …)

> Content variants (language, channel-specific preparation)

> Separation and relationships of content with descriptions, navigation, layout, etc.

> Foundation for content editing, APIs for templates / document generation, search features, content storage / database schemas, import and export formats

JSON Schema concentrates on

> data formats (structure) and

> very little on semantics (basic types)

Simplicity is the advantage of JSON

Model information is lost when representing content in JSON.
Content semantics cannot be recovered from external representation alone.
Idea: relate content model with all external formats for consistent representation and interpretation.

// M³L

An overview over (nearly) all language constructs. More complete descriptions can be found in the paper and the literature

`> A`	> The declaration of or reference to a *concept* named **A**
`> A is a B` `A is the B`	> The *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 of a concept **A**; whenever **A** is referenced, **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 **A**; **A** is printed out as or recognized from the concatenation of the syntactic forms of concepts **E**, **F**, and **G**; if not defined, a concept evaluates to / is recognized from its name

// M³L Expression Evaluation

- The M³L has an operational semantics for expression evaluation
- It is based on (any combinations of)
  - Refinement
  - Semantic rules
  - *Visibility* rules
    - All concepts in the content of a concept are also visible in the content of refinements: `A { B }, C is an A ⇒ C { B }`
    - All concepts in the content of a concept are also visible in the contents of concepts in the context of that concept:
      `D E { F } ⇒ E { F { D } }`
  - *Narrowing*: If a concept A has a subconcept B, and if all concepts defined in the context of B are equally defined in the context of A, then each occurrence of A is narrowed down to B.

```
> 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 }

> Person {
    Peter is the Name
    42 is the Age }
  ⇒ PersonPeter

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

// M³L as a Language for Content Models

M3L can be applied to content modeling and management

Example:

> Simple content model definition

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

Concepts *Content*, *String*, and *FormattedString* may be defined elsewhere

> According content can be created

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

Note that the M³L does not distinguish between “schema” and “instance”

// Producing JSON Using the M³L

Syntactic rules to produce JSON and JSON Schema from M³L concept definitions.

Sample syntactic rule for JSON output

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

This rule produces JSON for the above example of `NewsArticle123` (rule is inherited)

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

Matching sample syntactic rule for JSON Schema

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

JSON Schema output

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

The different rules for Distinguish between outputs by contextualization

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

// Variants (Polymorphism) Using JSON's if…then

JSON Schema lacks subtyping. Structurally, variants can be expressed. Semantically, there is no counterpart to subtypes, variants, contexts.

M³L

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

In JSON Schema, an if…then construct is the closest we get

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

// Additional Aspect of a Central Model: Content Conversions and Computed Values

Mappings from and to the content model can also include conversions and computations of canonical forms.

> 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 such definitions, it is possible to state conversion rules like the following

```
Price { Value is a FloatNumber
        Currency }
|- "{\"price\": {\"value\" :" Value ", \"currency\":" Currency "}}" .

PriceInEuro is a Price { € is the Currency }

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

> With this definition, an incoming JSON for a price in a currency is directly computed to a price in Euro

// Summary and Outlook

Summary

> Content is represented in various ways in a digital communication system:
> internal and external representations, as displayable documents, APIs, in search indices, …

> To interpret data as content consistently: central content model that underlies all representations

> Study of the potential of central models and mappings to data representation languages of standard protocols for communication between CMS components using the M³L

Outlook

> Investigate the gap between rich content models and structural data schemas more deeply

> Derive properties of a JSON schema language that helps bridge the gap better

> Current syntactical rules of the M³L require one specific rule per concept. There should be templates

THANKS.