Designing Connected Content for a Conference Web Site

Yannik Rauter

Designing Connected Content for a Conference Web Site

Yannik Rauter

Bachelor's Thesis

Bachelor's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Human-Centred Computing (HCC)

Graz, 24 Jan 2025

Designing Connected Content für eine Konferenzwebseite

Yannik Rauter

Bachelorarbeit

Bachelorstudium: Software Engineering and Management

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Human-Centred Computing (HCC)

Graz, 24 Jan 2025

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

Designing Connected Content (DCC) is an approach to designing and building web sites through domain and content modelling. Content is typically stored in a Content Management System (CMS), conceptually separate from any particular frontend user interface which uses and presents the content. This approach attempts to increase maintainability and reusability of content, while simultaneously reducing duplication and repetition. Content becomes available, findable, memorable, and connected.

This thesis describes the steps involved in Designing Connected Content from start to finish, taking the example of a conference web site. First, the domain is modelled, and a corresponding content model is created. Then, the content model is implemented in Contentful, a commercial Headless CMS with a fairly generous free plan, and sample conference content is created and uploaded. Finally, the Static Site Generator (SSG) Hugo is used to fetch content from the CMS and build a static web site for the conference.

Kurzfassung

Designing Connected Content (DCC) beschäftigt sich mit dem Entwurf von Websites durch Modellierung der zugehörigen Domäne rund um die Inhalte. Ein Content Management System (CMS) dient dabei als Speicher für Inhalte, wodurch diese konzeptionell von sämtlichen Frontend-Benutzeroberflächen zur Präsentation entkoppelt werden. Dieser Ansatz zielt darauf ab, die Wartbarkeit und Wiederverwendbarkeit der Inhalte zu erhöhen, während zugleich Duplikate und Wiederholungen vermindert werden. Die Inhalte sind hierdurch weniger separiert, sondern werden miteinander verbunden, um die Sichtbarkeit zu fördern.

Diese Bachelorarbeit beschreibt den gesamten Prozess von Designing Connected Content am Beispiel einer Konferenzwebseite. Anfangs wird die Domäne modelliert und ein zugehöriges Content Model erstellt. Dieses Modell wird anschließend in Contentful implementiert, einem kommerziellen Headless CMS mit großzügigem Funktionsumfang in der kostenfreien Variante. Beispielhafte Inhalte werden dort erstellt und hochgeladen. Zuletzt wird der Static Site Generator (SSG) Hugo verwendet, um die Inhalte des CMS auszulesen und damit eine statische Konferenzwebseite zu entwerfen.

Contents

List of Figures

List of Listings

Acknowledgements

To begin with, I would like to thank my parents for encouraging me throughout and enabling my academic career. Being able to enjoy this high level of education here in Austria is an opportunity which should not be taken for granted.

Next, thank you to my partner and friends, who endured my seemingly endless turmoil, yet ultimately inspired me to challenge my procrastination and finish this work of art.

I also want to acknowledge Christian Gütl and Tobias Schreck for their feedback on my domain and content models in the role of subject-matter experts for conferences.

Finally, I would like to thank Keith Andrews for supervising my thesis over the course of (too) many months, matching my perfectionist tendencies and appreciating the attention to detail.

Yannik Rauter

Graz, Austria, 24 Jan 2025

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021].

Chapter 1

Introduction

The Designing Connected Content (DCC) approach to designing and building web sites was introduced in the book of the same name by Atherton and Hane [2017]. It builds upon earlier work known under the moniker Domain-Driven Design [Millett and Tune 2015]. DCC straddles the fields of content strategy [Kissane 2011; Casey 2023] and information architecture [Rosenfeld et al. 2015; Spencer 2014].

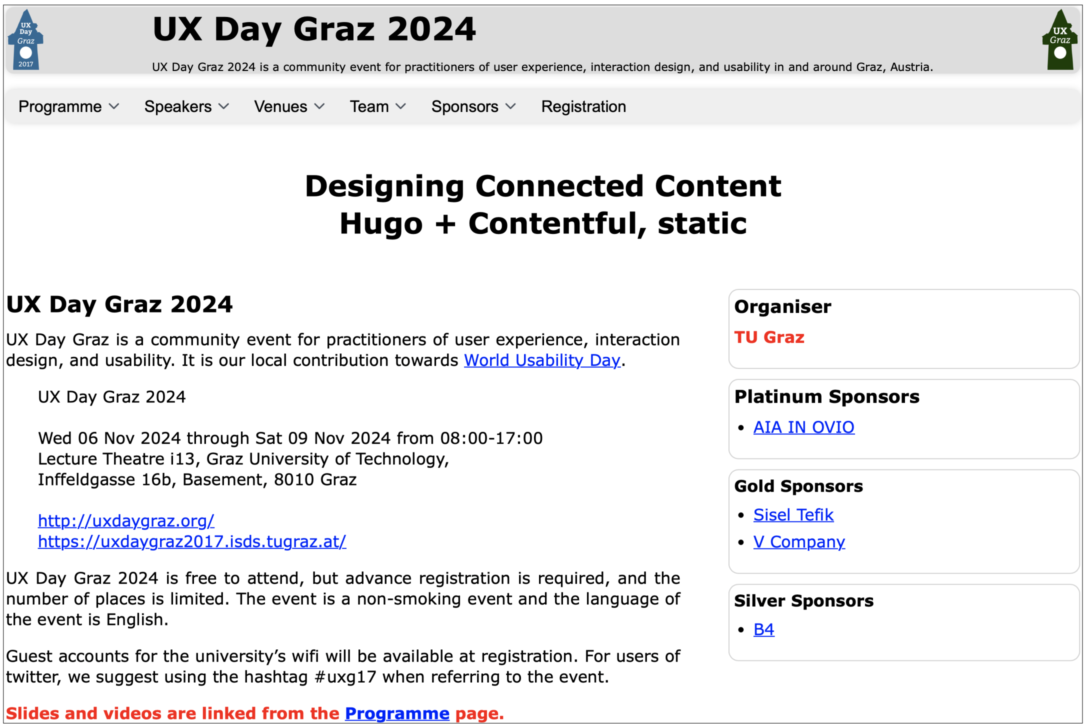
According to Atherton and Hane [2017, page 198], DCC addresses the key question of how to "structure and create your content based on the mental models of your audience *before* you start to represent it through an interface". Crucially, the design is performed bottom up. This starts with understanding the subject domain through research, aided by discussions with subject-matter experts. Through a procedure called *domain modelling*, a structure of logical concepts and their relationships is assembled. These objects are further evolved during the subsequent step of *content modelling*, by defining attributes to hold individual pieces of content. At its core, this effort aims to minimise unnecessary and error-prone duplication of data, which would inevitably lead to inconsistencies over time. Every instance of content must have its dedicated place, relating and linking to others, but ultimately existing only once [Atherton and Hane 2017, Chapters 1–6].

The derived *content model* maps the subject domain to a form which can be stored electronically in a data store. Typically, a (Web) Content Management System (CMS) is used as the data store. Once implemented, a CMS allows non-technical individuals (content creators, authors) to add, edit, and remove content through a user-friendly interface. There is no need for complex database queries and table manipulation [Barker 2016, page 80-81]. Content in this context may refer to any asset, be it writing in textual form (like names, descriptions, addresses, contact details), documents (forms, papers, slide decks), images, graphics, videos, music, audio recordings, and so forth [Naik and Shivalingaiah 2009, page 225]. Only after the content is structured and created in a meaningful way, can its presentation be considered.

Most CMSs provide an Application Programming Interface (API), over which content can be programmatically accessed and retrieved. This allows a frontend user interface to be decoupled from content in the content store. One way to build a frontend user interface is to use a Static Site Generator (SSG). This type of software tool enables the generation of individual pages for various instances of the previously defined content, based on highly customisable templates [Petersen 2016, page 9]. Any page's layout may be supplemented by reusable blocks containing content.

The next chapter, Chapter 2, provides an overview of the DCC approach. Chapter 3 describes Content Management Systems and their use as a content store for connected content. Chapter 4 describes Static Site Generators and how they can be used to provide a frontend user interface to connected content.

The remaining chapters take the example of building a web site for a small, fictitious conference, UX Day Graz 2024. Chapters 5 and 6 describe the creation of the domain and content model for the conference, respectively. Chapter 7 describes how the modern, commercial, headless CMS Contentful



UX Day Graz 2024

UX Day Graz 2024 is a community event for practitioners of user experience, interaction design, and usability in and around Graz, Austria.

Designing Connected Content
Hugo + Contentful, static

UX Day Graz 2024

UX Day Graz is a community event for practitioners of user experience, interaction design, and usability. It is our local contribution towards World Usability Day.

UX Day Graz 2024

Wed 06 Nov 2024 through Sat 09 Nov 2024 from 08:00-17:00
Lecture Theatre i13, Graz University of Technology,
Inffeldgasse 16b, Basement, 8010 Graz

http://uxdaygraz.org/
https://uxdaygraz2017.isds.tugraz.at/

UX Day Graz 2024 is free to attend, but advance registration is required, and the number of places is limited. The event is a non-smoking event and the language of the event is English.

Guest accounts for the university's wifi will be available at registration. For users of twitter, we suggest using the hashtag #uxg17 when referring to the event.

Slides and videos are linked from the Programme page.

Organiser
TU Graz

Platinum Sponsors
- AIA IN OVIO

Gold Sponsors
- Sisel Tefik
- V Company

Silver Sponsors
- B4

Figure 1.1: The UX Day Graz 2024 home page. [Screenshot taken by Yannik Rauter.]

was configured and populated for use as the content store for the conference web site. Chapter 8 describes the use of the Hugo SSG to build a static web site for the conference. The home page of this web site is shown in Figure 1.1. The code can be found at Rauter [2025]. Content is fetched from Contentful at build time to populate the various web pages. The resulting static web site can then be deployed, allowing attendees to browse the conference programme, speakers, materials, and so forth. Finally, Chapter 9 gives an outlook to potential future work.

Chapter 2

Designing Connected Content

Designing Connected Content (DCC) is about designing from the bottom up, which starts at understanding the subject domain [Atherton and Hane 2017, Chapter 1]. Attaining this basis is essential for breaking down the content into meaningful parts, thereby giving it structure. This in turn allows for more flexibility in organisation, arrangement, and ultimately, representation. Developing the facade only makes sense once the cornerstone has been set. When a piece of content is stored in one central location, the need for error-prone and inevitably inconsistent duplication disappears. At the same time, altering data becomes more convenient. In this spirit, DCC goes beyond making the right content, but extends into making the content right. Without a solid foundation, collapse is just a question of time.

Building a collective understanding of a subject domain is the role of a content strategist or information architect. Ensuring the development team and any stakeholders have a common view of the terminology and structure of the domain forms an essential foundation for any project.

Designing Connected Content (DCC) builds historically on previous work known as Domain-Driven Design (DDD): "a process that aligns your code with the reality of your problem domain" [Millett and Tune 2015]. The requirement for linking data in a machine-readable format was realised early on, accompanied by the trend of open-sourcing information as open data. This greatly improved discoverability, while simplifying redirection to related sources, thereby further extending knowledge.

DCC makes it possible to decouple content from presentation. Multiple frontends can be built, all drawing content from the same content store. Early solutions were implemented as Linked Open Data (LOD) [Yankulov 2024], with content stored in a data server (a so-called *triple store*) and accessed using the SPARQL query language [DuCharme 2013]. In the corresponding frontend, each instance of a content object is typically assigned its own URL and can be accessed independently [Andrews 2024, page 66]. Oliver [2024] describes a modern version of this approach. Most solutions today use a Content Management System of some kind to store the content. Access is provided either by a bespoke REST API or using the GraphQL query language [Porcello and Banks 2018].

2.1 Process Overview

Designing Connected Content may be summarised as a process consisting of six steps, shown in Figure 2.1:

1. *User Research*: Understanding the subject domain through research and interviews with experts in the field.
2. *Domain Modelling*: Structuring the domain into logical units of related information (*domain objects*) and their relationships to create a *domain model*.
3. *Content Modelling*: Evolving the domain objects into *content types*, adding attributes and entity relationships along the way to derive a *content model*.

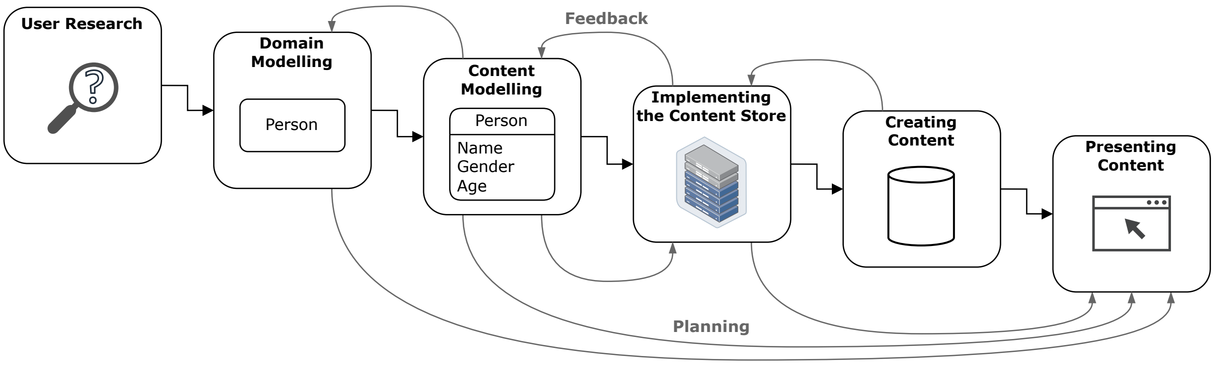


Figure 2.1: The six steps for Designing Connected Content. Insights from later steps feed back into previous steps. [Diagram drawn by Yannik Rauter.]

4. *Implementing the Content Store*: Implementing the content types in a content store, specifying data types for attributes and *entity references* for relationships.
5. *Creating Content*: Creating instances of relevant content in the content store.
6. *Presenting Content*: Designing a frontend user interface, with composable templates populated by fetching content from the content store.

2.2 User Research

The first step in the DCC process is user research. Structuring content into logical units of related information (so-called *domain objects*) requires extensive research of the subject domain, in order to gain a comprehensive understanding of the individual demands that various stakeholders have for the product. The necessary level of insight may only be achieved by preparing some probing questions and interviewing Subject-Matter Experts (SMEs), who are (in most cases) keen on having their voices be heard and gladly share intricate details. Talking to actual end users is important as well, since these conversations hold the key to unlocking the secrets of what real consumers want.

It is then the content strategist's duty to condense the resulting avalanche of information, filtering out redundancies, separating what is right from what is wrong, and identifying basic key concepts. A list of terms along with their definitions may be compiled, which will later aid the creation of an initial draft for the domain model. Product owners or managers often struggle to appreciate the importance of user research, pushing for quick progress in the interest of saving costs. Again, the content strategist must highlight and explain why this step should not be cut short.

2.3 Domain Modelling

This step builds a so-called *domain model*, an example of which will be elaborated on in Chapter 5. The two basic components of a domain model are the *domain objects* and their *relationships*. Crucially, this model does not list any attributes for its objects yet, although they may be saved for later, in case their discussion arises. The process of its creation involves all members of the team and often takes the form of an informal meeting under the content strategist's moderation. Using coloured pens and sticky notes, domain objects are selected and connected to each other through relationships, while considering cardinality and recursion. Defining strict boundary objects which mark the gateway to adjacent domains (and might just as well belong to a completely different domain) is important to prevent diverging too far. The process is described in detail by Atherton and Hane [2017, pages 74–76]

The goal is for the model to be reusable and flexible, which requires a certain level of abstraction. Therefore, the objects should be broad, real-world concepts, which are named using singular nouns. Beware of confusing objects with instances. An *object* is an abstract concept. An *instance* is a particular example of an object. Atherton and Hane [2017, pages 78–79] list some questions to help decide which concepts to select as objects.

Relationships between objects are used to build understanding. They are given names as verbs (in camel case), describing the nature of the relationship. The direction of naming follows no strict rules, but should be consistent across the model (for example, larger objects may contain smaller ones). Cardinality is expressed using one-to-many relationships in most cases, which may be optional at any of their ends and should utilise a common notation throughout. One-to-one relationships can always be omitted simply by merging the connected objects, usually without any loss of information. In contrast, many-to-many relationships should be resolved into separate one-to-many relationships by inserting a new domain object wherever possible.

The resulting domain model should be tested by trying to fill it with some instances, making adjustments as needed. It is good practice to periodically confer with SMEs, not just to gather feedback, but also for knowing when to stop. At this stage, objects and their relationships are important, methods for representing the content are not yet considered.

2.4 Content Modelling

This step creates a so-called *content model*, an example of which is given in Chapter 6. Any additional knowledge gained which necessitates updates to the preceding domain model should be viewed as cost-savings for being revealed at this stage already, as compared to surfacing even later in the process.

In contrast to the domain model, the content model strives not to map the domain, but to provide the relevant structure for the content of its objects. The general layout and approach, as well as the domain object's names are inherited. Advancing through this stage involves all team members once more, including SMEs. During another meeting (again moderated by the content strategist), the previously mapped domain objects are reconsidered and potentially filtered out as an initial step. One key question at this point is whether content exists for the concept. Which objects to keep or remove must be determined also with respect to the core business, the relevance for the audience, and the object's influence on connecting relationships.

Next, attributes (sometimes referred to as properties) are attached to the remaining domain objects, each describing a characteristic or quality. Such attributes are typically selected based on the requirements of the particular subject domain. This evolves the objects into content types, which describe a reusable container tailored to the consistent form of similar content. Any concepts which must not necessarily be displayed on their own, or end up having just one attribute, may be discarded or merged to become attributes of other objects. Likewise, discussions may lead to the need to add further content types. New insights are reflected back to the domain model. Along the way, any deciding factors should be documented for later reference.

Content becomes *connected* when one content type is referenced by adding it to another content type's list of attributes. Switching over to *designing* content in a literal sense, Atherton and Hane [2017, pages 118–120] explain that well-designed content should be useful, usable, findable, focused, targeted, distinctive, and connected. At this point, all the pieces of the puzzle start fitting together. Every entity is a specific instance of a content type. Each entity will soon be available as a specific resource via a dedicated URL.

When auditing existing content, a certain willingness to change is required. Change comes in the form of adapting the current models to newly derived insights, expanding or restructuring attributes where necessary [Atherton and Hane 2017, pages 135–137]. Some content might simply not fit the model, for example if it maps to no attribute, belongs to no objects, or is too broad for a single content type. When some information is as yet unavailable, an attribute may be left empty for the moment and be populated later. Alternatively, adding a text attribute to store a URL provides the benefit of linking to external content for further reference, thereby filling any gaps.

One crucial task is to determine the right *chunk size* for the content, striking a balance between keeping it focused and distinctive while staying useful and relevant. Chunks strive to be valuable on their own. However, only through their connection do they really come to life. Deciding on the scope for these minimal reusable units often proves difficult, but once this structure is set, the content model is ready for its actual content.

2.5 Implementing the Content Store

Nowadays, a Content Management System (CMS) is often used as a content store. They are described in Chapter 3. An implementation example using a particular CMS is described in Chapter 7.

The previously derived content model is now manually implemented in a CMS of choice by configuring all content types and their attributes (which become *fields*). This creates containers for data entries and allows individual instances of content to be entered and stored in the CMS afterwards. Incompatibilities or issues may surface at any point in this process and for various reasons. Times or dates may be split into ranges, buildings and rooms merged into a single location, or seemingly minor details like the name given to an attribute might be ambiguous or imprecise. Changes to the model caused by these differences should (as always) be documented for later reference.

This step also necessitates the specification of a data type for each field, enforcing a limitation on the allowed values according to typical restrictions like number formats, text lengths, selection of predefined values, and so forth. Moreover, it is possible to require the provision of assets in the shape of documents (forms, papers, presentations), images/graphics, videos, music, audio recordings, and more. Since content authors are sometimes overwhelmed by the task of entering data into a CMS, these data types already provide crucial indirect hints regarding the type of information contained in every field.

Entity references are the relationships between instances of content types (entities). Such a connection usually works best from a changing concept to a fixed one, meaning: If an entity `a1` of content type `A` (with its fixed set of properties) may be part of many other entities `b1`, `b2`, and `b3` of content type `B`, then that one entity `a1` should be referenced from all of those many other entities `b1`, `b2`, and `b3`.

The robustness of the model should be tested by periodically by adding sample content, aiming to discover potential flaws early on. This ensures that all types and fields are appropriately populated and serve their intended purpose.

2.6 Creating Content

To create new instances of content, creators generally use a form-based user interface to enter data into the corresponding fields. Creators sometimes struggle with adjusting to this new, structured approach of authoring. They might be in need of a *content spec sheet* as a sort of conceptual guideline. Such a document can contain a set of rules, instructions, specifications and other noteworthy considerations for every content type and its fields, thus assisting the work of producing relevant high-quality content.

2.7 Presenting Content

In the final step, content is fetched from the content store and presented to the end user by some form of frontend solution. The content may be accessed on a wide range of devices and in a number of ways.

One way is to create a static web site with a Static Site Generator (SSG). Here, the content is fetched and integrated at build time. Changes in content are incorporated in the next build. Static sites generally offer faster load times and more efficient scalability. The data from the CMS is accessed programmatically using a REST API or GraphQL. Often, JavaScript is used to fetch content and create markup and additional metadata (a typical Jamstack example [Andrews 2024, page 67]), but other languages like Python can also be used. SSGs are described in Chapter 4. An implementation example using a particular SSG is presented in Chapter 8.

As an alternative to the static approach described above, a dynamic approach allows a web frontend to query the content store and fetch content at run time. Changes in the content are reflected immediately the next time the page is loaded, rather than when the site is next built. However, in this approach, routes (URLs) to individual instances have to be generated dynamically, which has to be implemented carefully [Das 2025].

Based on the needs of the application, instances of core content types may be given their own page with their own URL. Other content is included as part of other pages. Configurable templates are used to construct and style the core content pages. Every occurrence of a core entity should be labelled with the title of the corresponding content type (by employing breadcrumbs, for example), to provide a sense of location within the web site to the user. The site's global navigation typically resides in the header and is designed around the previously selected core content types (those having their own singe page templates). Furthermore, customised lists of entities allow users to traverse the site, with contextual links placed on connected content (often via a sidebar), enabled by the relationships modelled through attributes.

The URL of every entity permits its most direct form of access. URLs should therefore be based on the core content type's title in plural (like `/sessions`), which presents a list of all corresponding elements. Particular instances are available a level deeper, each at their own unique path (like `/sessions/ux`). Incorporating volatile attributes should be avoided due to their non-persistent nature, while aliases via redirects simplify references for marketing purposes.

2.8 Discussion

In the past, not enough thought was put into structuring content, possibly for the lack of dedicated disciplines like content strategy and information architecture. Systems hardly ever shrink over time, but growth demands scalability. Often, stakeholders and key decision makers fail to focus on a product's longevity by prioritising short term gains instead.

The DCC approach forces a domain-driven focus on content modelling first, with applications building on this sound foundation. As Atherton and Hane [2017, page 147] appropriately put it, "You want to build your tool to fit the model, not model your content to fit the tool".

Chapter 3

Content Management Systems

In simple terms, a Content Management System (CMS) is a software component whose purpose is to store, manage, and provision content. It allows non-technical authors to create and maintain content (text, images, etc. and associated metadata) with a straightforward user interface which interacts with the underlying database (the backend *content layer*). This includes the ability to add new content, as well as editing and removing existing content [Naik and Shivalingaiah 2009, page 226].

As part of the setup process, an administrator or developer configures a *content model* to suit the needs of the application. It specifies the content types, their fields and attributes, and entity relationships between content types. Storage is usually handled under the hood with a relational database system, whose internal structure (tables and columns) is flexible and can therefore be customised [Barker 2016, page 80–81].

CMSs often include their own implementation of the frontend *presentation layer*, which displays content to the end user, typically in the form of web pages. Authors with appropriate access rights can enter an edit mode and make use of a WYSIWYG-style (What You See Is What You Get) [IONOS 2023] editor to change it.

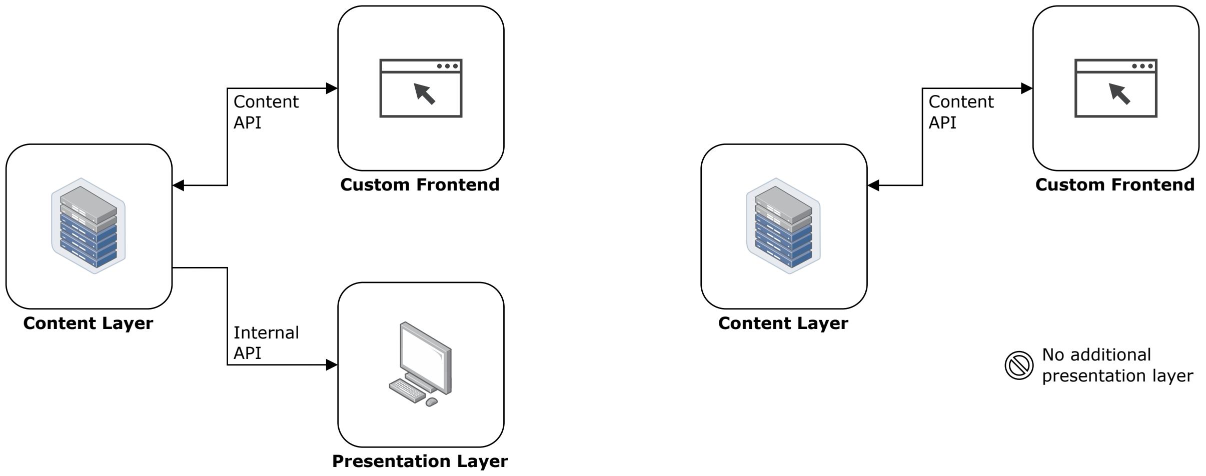
3.1 Decoupled vs. Headless CMS

A *decoupled* CMS provides its frontend presentation layer clearly separated from its backend content storage, as shown in Figure 3.1a. Content may be fetched over an API, or the provided frontend may be used instead [Howey 2023; Singh et al. 2023, page 88].

A *headless* CMS forgoes the presentation layer, completely separating content and presentation [Barker 2016, page 82], and fully relying on another frontend solution to display its content using an API [Atherton and Hane 2017, page 146], as shown in Figure 3.1b. A headless CMS usually provides a separate authenticated editor for authors to create and maintain content.

3.2 The Content API

Every CMS typically provides programmatic access to its content, usually in the form of one or more Application Programming Interfaces (APIs), which allow other software applications to interact with the content using typical create, remove, update, and delete (CRUD) operations. However, the most important aspect of this interface is the capability of *reading* data from the database (for example, in JSON format). This is what ultimately enables the creation of proprietary solutions for representing content tailored to diverse target audiences with various device types and screen sizes through omnichannel distribution [Yermolenko and Golchevskiy 2021, page 6].



(a) Decoupled CMS.

(b) Headless CMS.

Figure 3.1: Decoupled and headless CMS both provide access to their content via an API. However, a headless CMS completely lacks any form of frontend presentation layer. [Diagram drawn by Yannik Rauter.]

3.3 Entity-Based vs. Page-Based CMS

An *entity-based* CMS focuses on structuring content per resource (*entity*), which is the desired approach in domain modelling. Every resource provides the structure for its content by defining attributes and relationships with other resources. Such resources are subsequently populated with content, essentially reusing the defined structure numerous times. This approach ensures that any entity is completely independent of the page (or whatever other medium) it may be displayed on within the end product, effectively separating the content from the presentation layer [Atherton and Hane 2017, pages 143–145]. Examples of entity-based CMSs include Contentful [Contentful 2024c] and Strapi [Strapi 2024].

A *page-based* (or page-centric) CMS offers an alternative to this, in the form of structuring content per page [Barker 2016, page 84]. Markup and sometimes style are bound to and stored with the content, so each item of content is addressable with its own URL [Barker 2012]. Repopulating a page with a different instance of content overwrites the previously held data. To create multiple instances of content with a similar structure, the whole page must be duplicated first before changing the copy's content in order to retain the original copy. This strongly limits reusability and means that page-based CMS are most applicable to web site paradigms rather than more general content repurposing. Examples of page-based CMSs include Drupal [Drupal 2024] and Episerver (now Optimizely CMS) [Optimizely 2024].

3.4 Hosted vs. Self-Hosted CMS

There are many commercial providers of *hosted* CMS. These are complete server solutions maintained by the provider, often with a basic free plan and paid commercial plans, with the benefit that the provider takes care of maintenance issues such as updates, performance, availability, backups, and other issues. A hosted, headless CMS solution is sometimes referred to as *Content-as-a-Service* (CaaS) [Singh et al. 2023, page 89]. Examples include Contentful [Contentful 2024c], Sanity [Sanity 2025], and Storyblok [Storyblok 2025].

The alternative is to install and self-host one's own instance of a CMS, with the drawback of having to take care of maintenance issues oneself. Some of the many available open-source CMSs include Strapi [Strapi 2024], Directus [Storyblok 2025], and Cockpit [Cockpit 2025].

3.5 Discussion

From the perspective of supporting the Designing Connected Content (DCC) approach to modelling and building a web site, using a headless, entity-based CMS has the following benefits:

- Integrated functionality is available for creating a content model and modelling relationships between entities.
- An authoring interface allows content to be created and maintained by non-technical personnel.
- Content can be made accessible on a wide variety of platforms and devices over an API.

Chapter 4

Static Site Generators

A *web site* consists of a collection of related, interconnected *web pages* [MDN 2024b]. A Static Site Generator (SSG) builds a web site by generating a collection of static pre-rendered pages. Static pages can be sent by the web server to the browser as soon as they have been requested, without having to perform any further processing, resulting in much faster response times. There are dozens of SSGs, written in a variety of programming languages, including Jekyll (Ruby) [Jekyll 2024], Metalsmith (JavaScript) [Van Lierde 2025], Eleventy (JavaScript) [Leatherman 2025], and Hugo (Go) [Hugo 2025g].

SSGs typically support the use of a template language to include metadata and reusable pieces of code (so-called *partials)* for shared elements such as a header, footer, or navigation block [Petersen 2016, page 9]. Some SSGs may offer support for additional features like themes for layouts, multi-language support, and plug-ins to extend their capabilities in various directions.

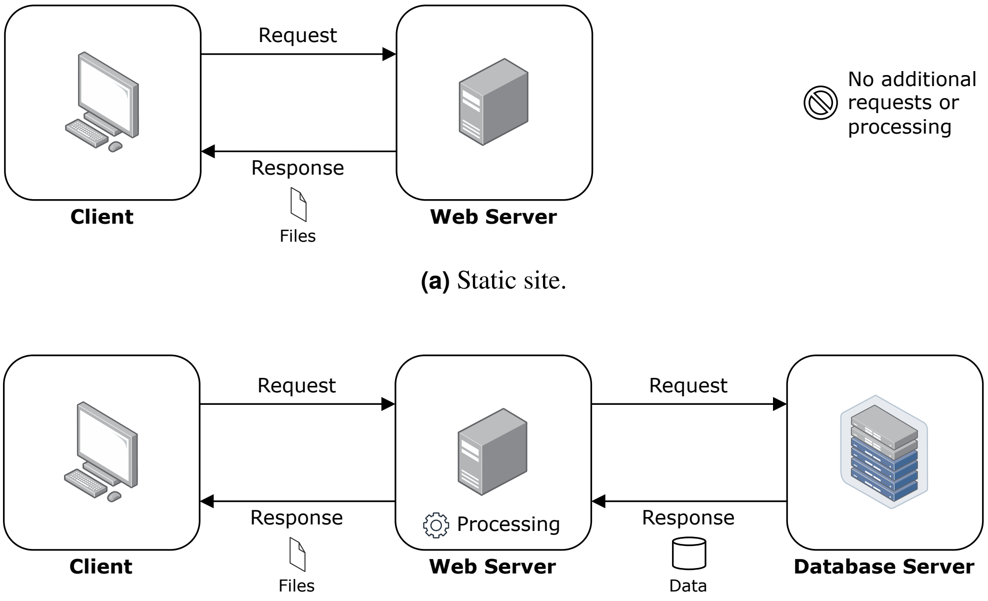
4.1 Static Sites vs. Dynamic Sites

Static sites are generated as a whole (including markup, styles, multimedia content like images, and client-side scripts) at a predetermined build time [Boiko 2004, page 76; Petersen 2016, page 7]. A static site only changes through manual intervention, either by directly altering any of its files, or by rebuilding its files following an update to the content. This characteristic enables efficient delivery by web servers, since the files are essentially pre-rendered and ready to send to the client (web browser) at any time. When loading the page, no further intermediate processing must be performed, as shown in Figure 4.1a, ensuring optimal response times. Static pages can also be efficiently cached by the web browser.

In contrast, a dynamic site only generates its final response page upon the client's request, after further backend processing. Thus, a dynamic site has the ability to adapt to user input or changing content [Petersen 2016, page 8]. It is important to note, however, that a dynamic site is by no means a CMS. While content is read from a database before being processed and assembled into a page for viewing (closely resembling a decoupled Content Management System), the similarities end there. This approach is visualised in Figure 4.1b.

4.2 The Process of Generating Static Pages

To generate a static page, an SSG requires two complementary components: one or more templates, and one or more instances of content to populate the template(s) with. A template is typically written using a templating language [Dhillon 2016], such as Nunjucks [Long 2025], Liquid [Luetke 2025], or Pug [Pug 2025]. Some SSGs support only one templating language, others support many. A template often consists of a base file defining a particular type of page's layout in general, as well as additional files including specialised instructions about where to place different types of content. In the context of Designing



(a) Static site.

(b) Dynamic site.

Figure 4.1: A static web site can serve its pages immediately upon request without any further processing. [Diagram drawn by Yannik Rauter.]

Connected Content (DCC), every content type will typically have its own specialised template file. List pages, which generate a list of instances of a particular content type, will have their own template too. Some special pages, like the landing page or the imprint page, might be hand-coded in HTML, without much templating at all.

At build time, the SSG iterates through all the provided instances of content, combining the individual parts of the corresponding template into one final static page for that instance of content [Petersen 2016, page 9]. In essence, the various raw inputs are merely converted from one format into another.

When using DCC, there are two approaches to populating pages with content from a headless CMS: either at *build time* by the SSG, or at *run time* by the browser. In the first approach, all content is fetched from the CMS at build time and is saved as static input files. Metadata from the CMS can be included in the form of so-called *front matter*, often written in YAML [Petersen 2016, page 19]. Then, the build is triggered to generate the static web site. The entire web site must be rebuilt to incorporate changes in the content in the CMS.

Alternatively, in the second approach, API calls are formulated in JavaScript to retrieve specific pieces of content or metadata from the CMS. These calls are initiated by the web browser once the page has loaded. The content of the web page on the server does not change, but pieces of content or metadata from the CMS are fetched on demand, as each particular page is displayed by the browser. In this case, changes to content in the CMS do not necessitate rebuilding and redeploying the entire web site.

The metadata accompanying instances of content can include things like:

- A unique identifier for cross-referencing between instances of content.
- The content type for determining the applicable template.
- Keywords or categories for the particular instance of content.
- A publishing date, before which the piece of content is not included.
- An alias, used to provide an alternative URL for a page.

4.3 Discussion

In terms of supporting the DCC approach to modelling and building a web site, using an SSG has the following benefits (see also Vepsäläinen and Vuorimaa [2022, page 437]):

- The use of templates drastically decreases the complexity of making changes to any shared elements (like the site layout or navigation bar) across all pages.
- Reusable blocks of content reduce the potential for inconsistency on the site.
- Listing instances of content (for example in the site navigation) can be automated, resulting in fewer repetitive processes during implementation.
- The computation of the final page only occurs once, regardless of the number of end user requests to view the page.
- Static pages can be served immediately upon request.
- No complex backend beyond a simple web server is necessary.
- Modern web browsers may benefit from an improved caching ability when frontend components (like client-side scripts) are being reused across pages.

One major drawback to using an SSG, in contrast to using the presentation layer of a CMS with an integrated editor, is that even minor changes in content may require a rebuild and new deployment of the site, which usually requires an experienced developer.

Chapter 5

Conference Domain Model

The Designing Connected Content (DCC) approach was used to create a web site for a fictitious conference called UX Day Graz 2024. As a first step, Subject-Matter Experts (SMEs) were interviewed and a domain model for a small conference was created over a period of several weeks. It was later refined to address insights and issues as they came to light. This chapter builds on the premise of domain modelling, as described in Section 2.3.

5.1 Design Choices

The various domain objects represent key concepts from the underlying conference domain. Interviews were conducted with two SMEs (Christian Gütl and Tobias Schreck), who have both attended many conferences and also organised conferences. Their input strongly shaped the model in its initial stages. Insights gained during subsequent steps further shaped the domain model into its current form.

One major decision while creating the domain model was to have Event be its own domain object. Despite holding only one instance in the final product for now (UX Day Graz 2024), this allows the domain model to be reused for other similar conferences. It also serves the dual purpose of a boundary object, marking the gateway to adjacent domains, while tying everything inside it together. The initial intention was to model a single-stream conference located in just one room. However, it quickly became apparent that support was needed for Sessions located across multiple Rooms, sometimes even spanning different Venues. The initial draft domain model can be seen in Figure 5.1.

Concerning the cardinality of connections, Role is an example for resolving a many-to-many relationship. Since a Session involves many Persons, and any Person may at the same time be part of many Sessions, an intermediary object was inserted. Role proved to be a perfect fit, since a Person may serve one Role in one Session and a different Role in another Session. Furthermore, certain Roles may concern the Event as a whole, not just a single Session.

5.2 Noteworthy Considerations

Role and Session rose to be the most prominent domain objects, around which all others revolve. Considering the subject domain from a distance, this arguably makes sense, since a conference typically involves speakers participating in a variety of sessions. All other objects enhance these core concepts, providing structure and detail to form a holistic view.

The direction of naming is generally suggested to lead from larger concepts to smaller ones. However, this principle is not always strictly followed (for example, a Role requires an Event). With an understanding of the relevance of individual objects within the domain (which is naturally developed over time), it was deemed sensible to direct from more important, central concepts (like Role) to contextually less significant ones (like Event in this example).

5.3 Resulting Model

The final version of the domain model is shown in Figure 5.2. The arrangement in the domain model diagram is purely aesthetic, based on the relationships between objects.

When comparing the two versions, it is clear to see just how much this model evolved over the course of many iterations. Some of these changes worth mentioning include:

- Session Format was initially modelled as its own domain object, before turning into an attribute of Session later on.
- Topic used to be a separate domain object, until becoming an attribute in Session and Track (the two objects it had relationships with).
- Slot was missing in the initial draft, Interestingly, it was a topic of discussion on numerous occasions, being added intermittently as Segment, only to be removed, and then finally added again as Slot.
- The relationship between Role and Track (required to declare its chair person) had not yet been considered.
- The early version is missing a legend explaining the meaning of the various cardinality symbols. Furthermore, the symbols themselves changed over time.

A full-page version of the final domain model is provided in Appendix A as Figure A.1.

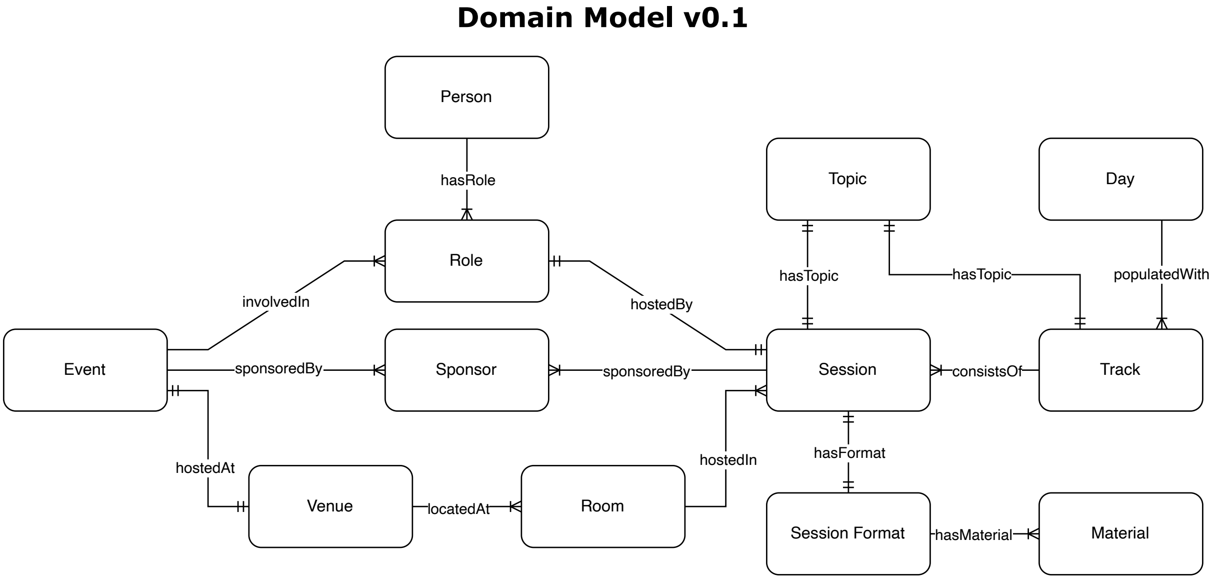


Figure 5.1: The initial draft (v0.1) of a domain model for a small conference. [Diagram drawn by Yannik Rauter.]

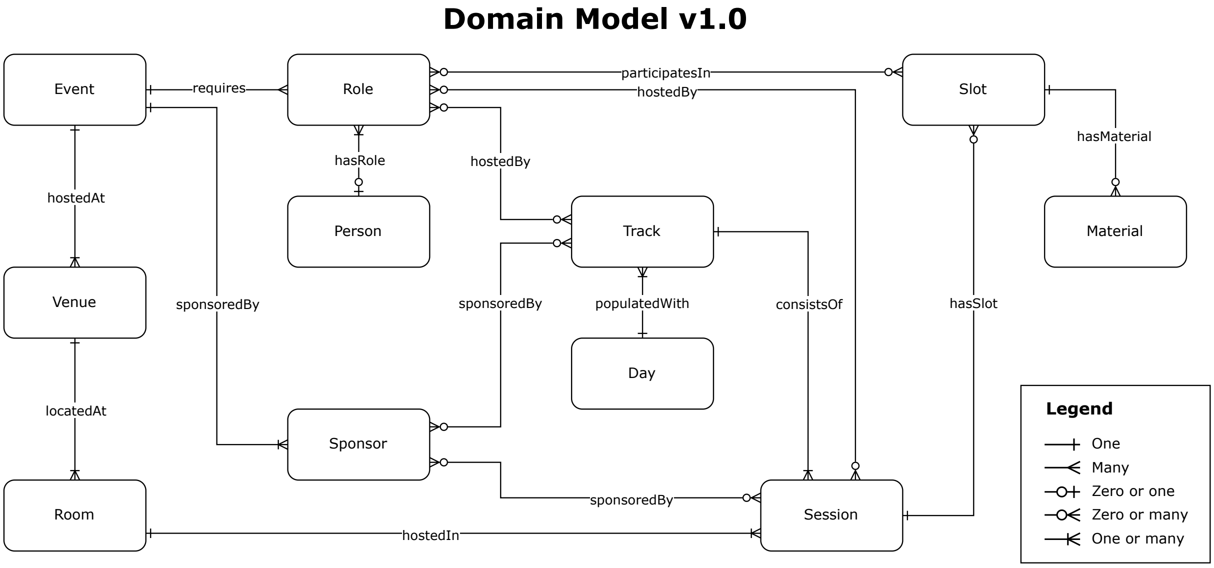


Figure 5.2: The final version (v1.0) of a domain model for a small conference. [Diagram drawn by Yannik Rauter.]

Chapter 6

Conference Content Model

Once the domain model was fairly stable, a content model was created for a small conference, following the DCC approach. Work on the content model started on the basis of version 0.6 of the domain model. To keep both models synchronised, the initial version of the content model was also denoted as version 0.6, and development continued in parallel. This chapter builds on the idea of content modelling, as described in Section 2.3.

6.1 Design Choices

None of the inherited domain objects were dropped when conceiving the first version of the content model, because at the time, content was believed to exist for all of them. After adding attributes, three content types Role, Day, and Session Format ended up with just one property. At first, all of these were supposed to be presentable on their own. Careful consideration showed that the relationships of Role play an important part in the structure of connecting adjacent objects. Thus, it was decided to retain all three as independent content types for the time being. Furthermore, no additional content types were created for the content model itself. Relationships are only modelled as connecting lines between content types at this point. The placement of entity references is deferred until implementation of the content model in the CMS later. The initial draft content model (v0.6) for a small conference is shown in Figure 6.1.

Technically, the Topic(s) attribute of Event could be implemented as an automatically generated concatenation of all the values from the Topic attributes across Session and Track. However, these topics must not coincide literally. Those in Session are more specific than their parent Track, and Event is yet another level of abstraction above. For this reason, topic is not a separate domain object, since its entities would yield limited reusability at best.

The Picture(s) attribute of Venue is designed to hold multiple assets within the CMS. If this property could be reused outside of Venue, it might have become its own content type. However, these images only have meaning in the context they relate to, hence they were modeled as attribites. Extending this principle, a Day concerns all Tracks on a specific date. Extracting this date obviates avoidable duplication, increases maintainability, and enables listing all Tracks on the same Day in the frontend later.

The DOI field in Material gives the digital object identifier (DOI) of academic documents like papers. Similar to an ISBN for a book, this identifier serves as a unique handle for one specific paper [Andrews 2021, pages 12–14]. Since not every document must have a DOI, this field is optional (and because not every Material must be a document).

The Web Site URL attribute, which is present in various content types, provides a field for linking to additional content of undefined structure. This creates room for information which does not fit into any of the available fields, thereby extending the scope of the content.

6.2 Noteworthy Considerations

Contrary to the method suggested by Atherton and Hane [2017, page 149], no spreadsheet was used in the creation of the content model. Instead, any discussions concerning individual attributes and necessary adaptations of domain objects were recorded as a bulleted list in a document. This approach worked well in the beginning (or perhaps for smaller projects like this one), but will probably fail to scale efficiently over the course of time, as remarks and insights accumulate.

Throughout the process of its inception, the suitability of provisional content was trialled on the model, to detect incompatibilities early on and reduce the need for major changes later. While the frontend is easy to adapt, redesigning the content model entails adaptations on all intermediary platforms (like the CMS) as well, which requires increased effort. Any attributes marked with an asterisk are implemented as mandatory fields in the CMS later, meaning these must always be populated when adding an entity. They also generally take precedence in the ordering of properties, which are sorted by descending relevance.

6.3 The Person – Role – Session Relationship

The cardinality between Role and Person strongly influences their threefold relationship with Session, forming an integral component within the conference subject domain together. Problems with this structure started to surface only late in the process, during the frontend implementation. However, the issue could be resolved by altering the magnitude of the connection on the Person's side. Previously, one Role could be related to zero or *many* Persons, whereas now every Role may relate to only zero or *one* Person.

First, take an example situation which produces a valid setup independent of this change. Consider a Session SA, whose Session Chair is the Person P. Consider another Session SB, whose Session Chair is also the Person P. Here, SA and SB can be related to the same Role R. That Role R then relates to the Person P.

Another common situation cannot be clearly modelled. Consider a Session SA whose Session Chair is the Person PA. Consider another Session SB whose Session Chair is the Person PB. Previously, SA might be related to Role R, while SB might simultaneously also be related to R (which causes issues soon). However, Role R could be related to Person PA *as well as* Person PB. This setup created an unclear connection from Session SA through Role R, since it is no longer distinguishable whether PA or PB is related to SA!

As mentioned before, this problematic situation was resolved by correcting the allowed cardinality from Role to Person to be at most one person. For example, consider a Session SA, whose Session Chair is Person PA. Consider another Session SB, whose Session Chair is Person PB. Now, SA must be related to Role RA, and RA must be related *only* to Person PA. Likewise, Session SB must be related to Role RB, and RB must be related *only* to Person PB. This setup creates a clear, unmistakable connection from Session SA through Role RA to Person PA (the same holds true concerning all B-entities). For this to work, the limitation of zero or one Person per Role is required.

It is not necessary to change the cardinality between Session and Role, since there may still be different Sessions sharing one Person through a Role. Furthermore, there may also still be multiple Roles (all related to their own Person) sharing the same Session, for example if two people are co-chairing a session.

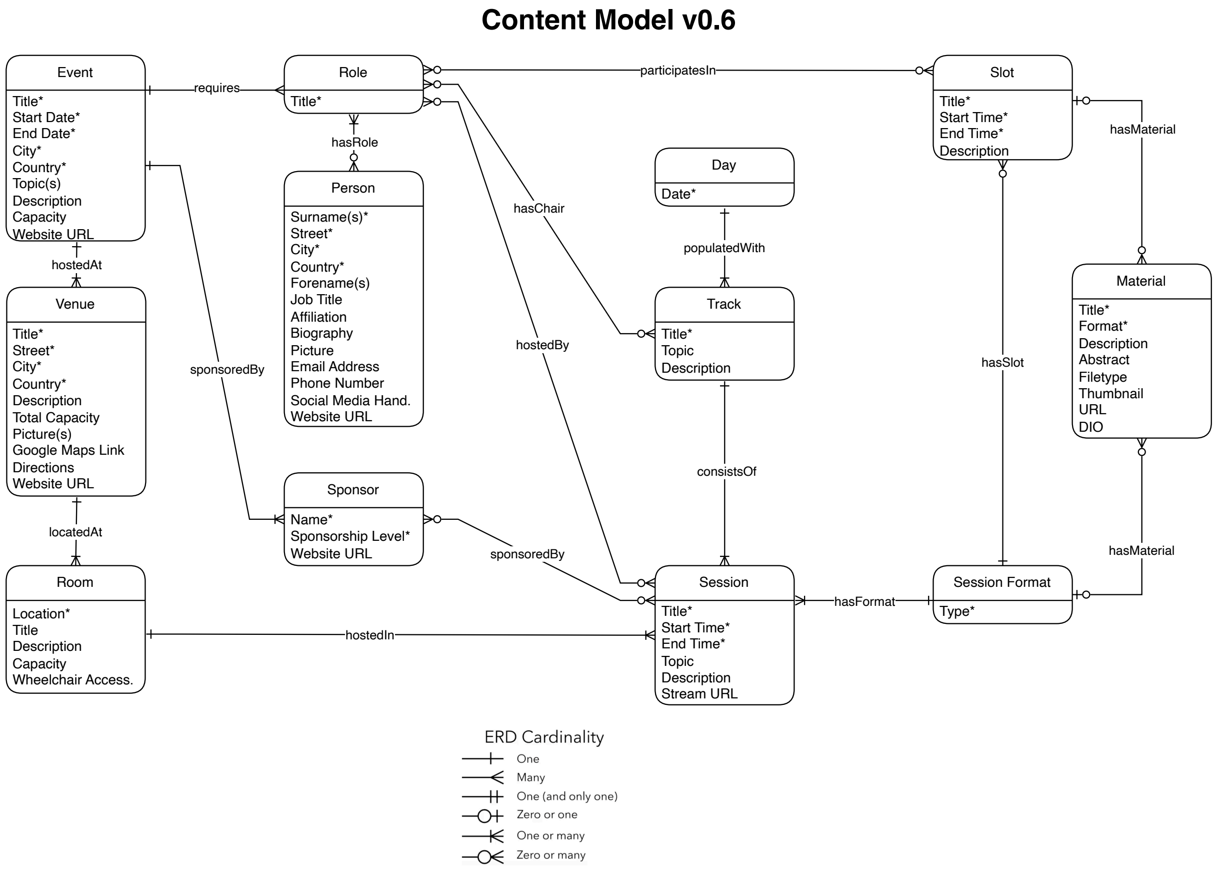


Figure 6.1: The initial draft (v0.6) of a content model for a small conference. [Diagram drawn by Yannik Rauter.]

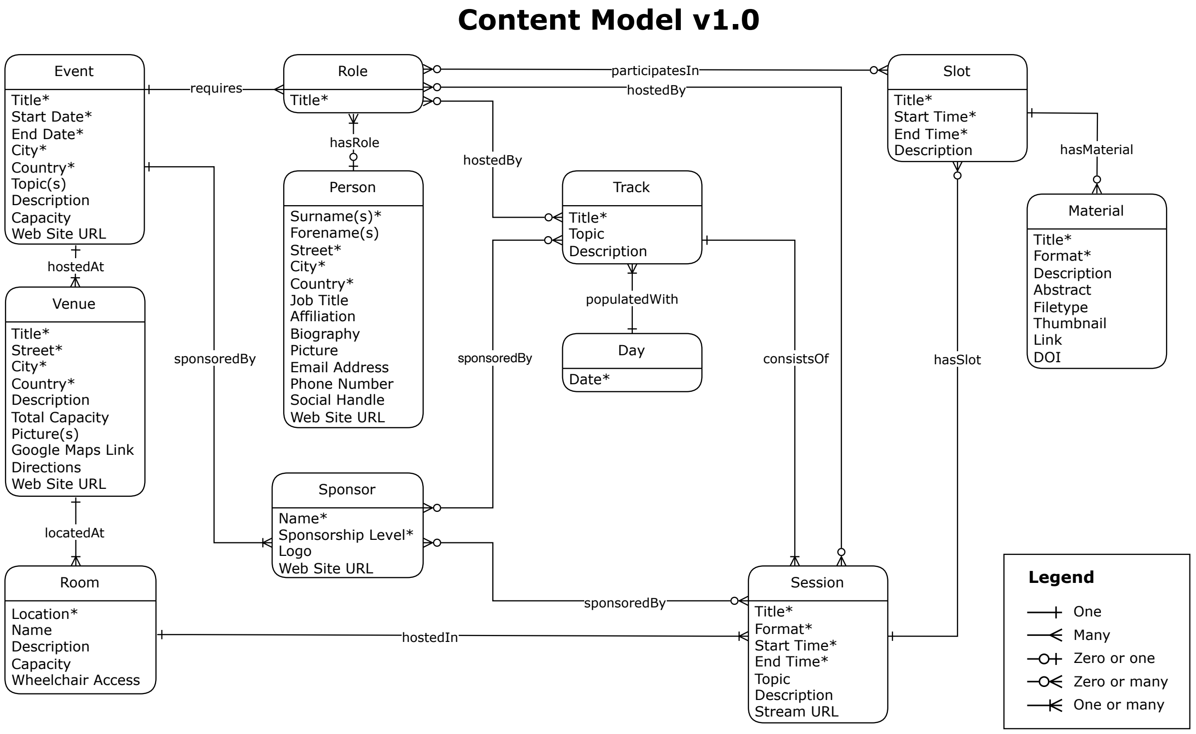


Figure 6.2: The final content model for a small conference. [Diagram drawn by Yannik Rauter.]

6.4 Resulting Model

The final content model (v1.0) for a small conference is shown in Figure 6.2. When comparing the final content model to the initial draft content model, the differences are less striking than when comparing the final and initial versions of the domain model (see Section 5.3). This is most likely because there were fewer significant structural changes to the content types along the way, since most of these alterations had already been made while creating the domain model.

Some key distinctions between the initial and final content models include:

- Session Format was initially modelled as its own domain object. After dropping the need for listing its entities, it was turned into an attribute of Session.
- A Logo attribute was added to the Sponsor content type, to provide a logo asset to the frontend implementation.
- As discussed, the cardinality of the relationship between the Role and Person content types was updated, so that every Role may only relate to at most one person.

A full-page version of the final content model is provided in Appendix A as Figure A.2.

Chapter 7

Backend: Contentful (Headless CMS)

The next steps in the project work for this thesis involved implementing the content model for a small conference in a Content Management System (CMS), and then filling it with content for the example conference web site.

7.1 Choosing a Content Management System

There is a vast array of CMSs to choose from, all with varying feature sets and popularity [Netlify 2024a]. The properties sought in a CMS for the purpose of this thesis work are firstly being entity-based rather than page-based, and secondly being headless rather than decoupled. This initially limits the choices, while further research and deliberations turned up two promising potential candidates.

Strapi [Strapi 2024] is an open-source, self-hosted, headless CMS. The choice of on-premise or cloud hosting is left to the developer. It advertises high customisability, not least through a wide range of plugins available via a centralised marketplace. Furthermore, Strapi provides integrations for many databases, frameworks (like React, Angular and Flutter), and Static Site Generators. Extensive documentation and an active community support the development process.

In comparison, Contentful [Contentful 2024c] is a cloud-hosted CMS, which touts itself as being API-first and content-centric. These traits describe its headless, entity-based nature, with a strong focus on reusability, facilitated by access to content through dedicated interfaces. Contentful offers many apps and integrations for third-party services, as well as AI suggestions and content creation features woven into its web application. Development is simplified through openly available documentation, showcases, and blog posts. Contentful offers a free tier with a limited feature set suitable for smaller projects.

Contentful requires no installation, no dedicated server, and very little configuration to get started, rooted in its approach of Content-as-a-Service (CaaS) [Singh et al. 2023, page 89]. For this reason, Contentful's free tier was selected to be used in the implementation of this project.

7.2 Setting Up Contentful

To start using Contentful, a user account must first be registered, setting up an organisation [Contentful 2024f] in the process. This serves as a home for spaces (basically a workspace), which are used to differentiate projects in development from those in production. In Contentful, an environment [Contentful 2024e] encapsulates different versions of content types and alternative configurations within a space, typically to separate development, staging, and production settings. Multiple users may have access to the same space, where roles (like Administrator, Author or Editor) optionally restrict their ability to alter content and change settings.

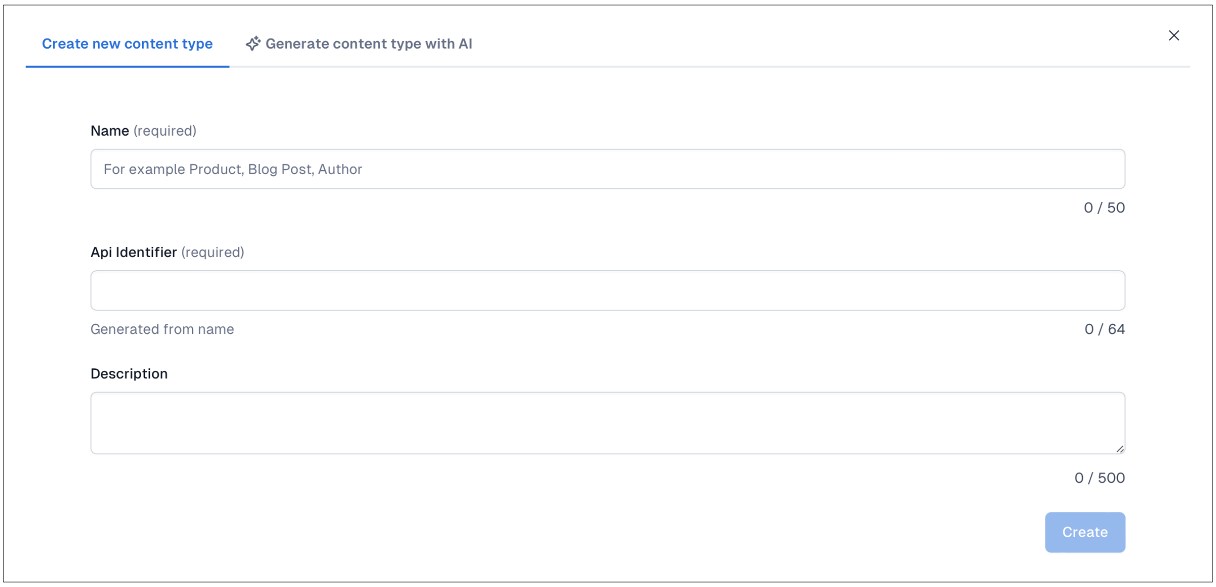


Figure 7.1: Creating a content type in Contentful. [Screenshot taken by Yannik Rauter.]

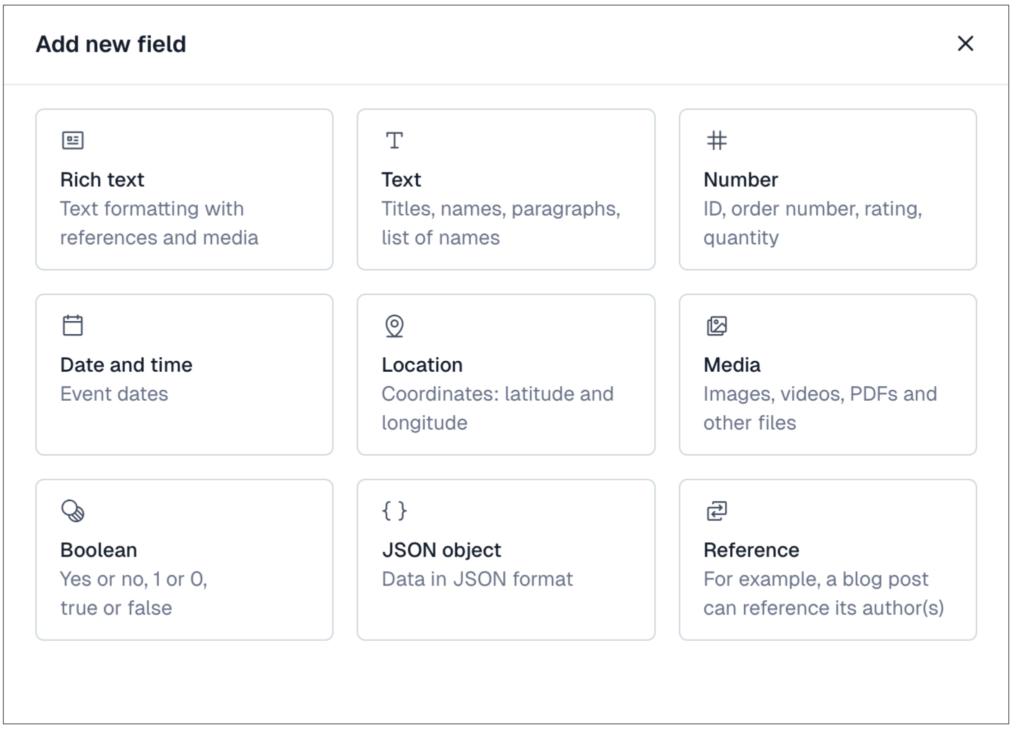
Content entities (instances) are called entries in Contentful. A space contains content types, entries, and assets. Every entry has a status, which defaults to Draft, before an entry is manually elevated by a user to being Published. Archived entries cannot be edited.

7.3 Implementing the Content Model in Contentful

A content type is created interactively in Contentful using the dialogue shown in Figure 7.1. It is given an automatically inferred API Identifier. Fields are then added to the content type using the dialogue shown in Figure 7.2.

The following data types are available in Contentful for fields:

- Rich text: Formatted text, including markup.
- Text: Unformatted text.
  - Short text: Maximum of 256 characters, enables sorting.
  - Long text: Maximum of 50.000 characters, no sorting.
- Number: Limits possible values to numbers.
  - Integer: Non-fractional numbers, positive or negative.
  - Decimal: Fractional numbers, positive or negative.
- Date and time: Date, with optional time and time zone.
- Location: Address or coordinates.
- Media: Assets of various file types.
- Boolean: True or false.
- JSON object: JSON-formatted data.
- Reference: A connection to an entry of a related content type.



Add new field

Rich text
Text formatting with references and media

Text
Titles, names, paragraphs, list of names

Number
ID, order number, rating, quantity

Date and time
Event dates

Location
Coordinates: latitude and longitude

Media
Images, videos, PDFs and other files

Boolean
Yes or no, 1 or 0, true or false

JSON object
Data in JSON format

Reference
For example, a blog post can reference its author(s)

Figure 7.2: The available data types for fields in Contentful. [Screenshot taken by Yannik Rauter.]

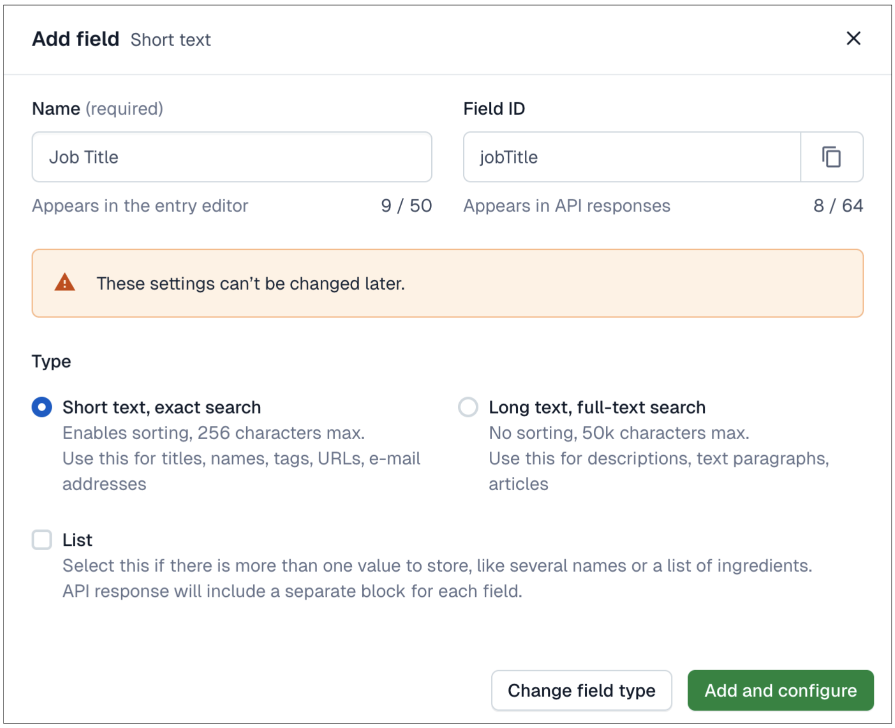
Any new field requires a Name, from which the Field ID is inferred using camel case, as shown in Figure 7.3 for a Text field. The interface points out that these settings (meaning the field type Short Text or Long Text) cannot be changed later. Attributes of a field, such as whether it is optional or required, can be set later, as shown in Figure 7.4.

All content types from the content model were implemented in Contentful. Their attributes become optional or mandatory fields, specifying data types and assets along the way. Limitations for allowed values were added to Role > Title, Session > Format, Material > Format, and Sponsor > Sponsorship Level.

In Contentful, every entry (instance) should have a non-unique Entry title whenever possible, which is used to represent an entry by giving it a name, for example to identify it in a list view. By default, the value of the first field whose type is Short text becomes the entry title. Any field of type Text (regardless of being short or long) can be specified as the Entry title for that content type [Contentful 2024b]. Other fields (including Rich text) cannot be used as the Entry title. As a consequence, entries of content types without any Text fields have no entry title, potentially leading to problems in the frontend if an entry title is always expected.

Entity references between content types are implemented by creating a field of type Reference. The resulting dialogue is shown in Figure 7.5. A decision must be made as to where to place the list of connected entities:

- One-to-one relationships should not exist (refer to Section 2.3).
- For one-to-many relationships (like the one connecting Person with Role), a single reference is always placed in the many side (in this case, Role). This way, any number of entries of the many side (any number of `Roles`) may all reference exactly one other entry (one `Person` each).
- With many-to-many relationships (like the one connecting Session with Role), a decision is required. Since duplication causes redundancy and the potential for inconsistencies, best practice dictates not to put a list of references to each other's content types on both sides of a many-to-many link (allowing



Add field Short text

Name (required): Job Title
Appears in the entry editor 9 / 50

Field ID: jobTitle
Appears in API responses 8 / 64

These settings can’t be changed later.

Type

● Short text, exact search
Enables sorting, 256 characters max.
Use this for titles, names, tags, URLs, e-mail addresses

○ Long text, full-text search
No sorting, 50k characters max.
Use this for descriptions, text paragraphs, articles

□ List
Select this if there is more than one value to store, like several names or a list of ingredients.
API response will include a separate block for each field.

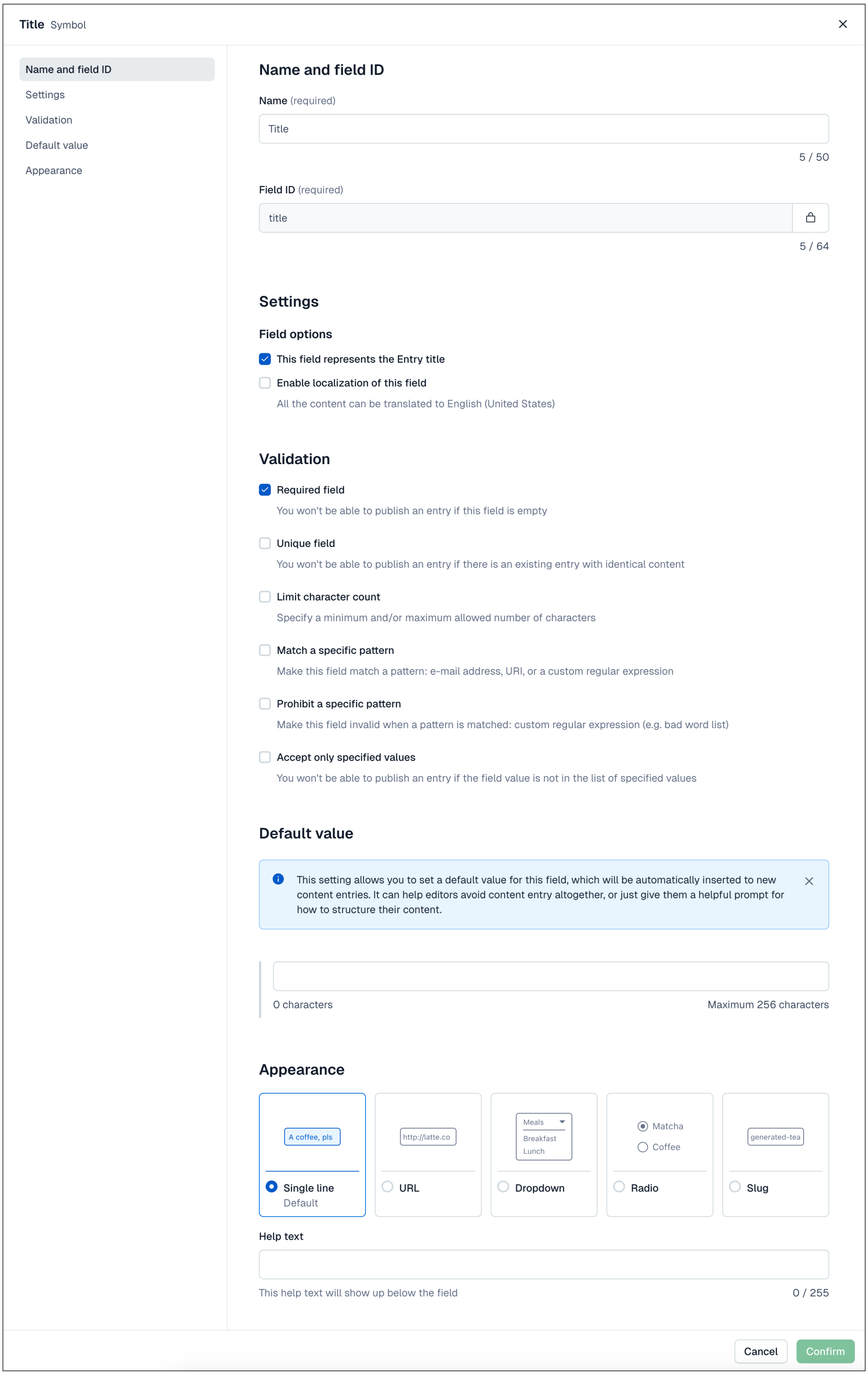
Figure 7.3: The interface for creating a field of type Text in Contentful, by example of the Job Title field in the Person content type. [Screenshot taken by Yannik Rauter.]

multiple `Sessions` in one `Role` *and* multiple `Roles` in one `Session`). Instead, the option for adding entity references should be placed on one side or the other. In this example, any `Role` lists all its connected `Sessions`. The following factors may influence this decision:

- The existing number of fields in a content type, which affects readability and clarity when editing.
- The existing number of entity references already added to a content type. Grouping together all references to other content types in one central content type simplifies references from one to all others (instead of necessitating edits in all others linking to the one).
- The later need for listing related entities may also be considered. While this is technically always possible (regardless of the placement), listing all related `Sessions` for one `Role` is programmatically simpler when Role holds the entity references to Session than the other way around.

The final implementation of the Session content type along with all its fields is shown in Figure 7.6. Lastly, Figure 7.7 gives a visual representation of all content types implemented in Contentful, along with their fields and entity references.

It is good practice to frequently try out the content model in the CMS with artificially created test content as well as real data. However, due to the approach of building the exemplary conference web site from scratch, no real, relevant content (like speaker biographies, session materials, or pictures of the venue) was available for testing. Thus, some realistic-looking example information was created to populate enough entries in Contentful to verify the model’s robustness under stress. All relevant documentation concerning the content types and their attributes is included in this thesis, so no dedicated content spec sheet was created. Throughout the process of implementing the content model in the CMS and testing the outcome with content, no changes to the content model were necessary.



Title Symbol ×

- Name and field ID
- Settings
- Validation
- Default value
- Appearance

Name and field ID

Name (required)

Title

5 / 50

Field ID (required)

title

5 / 64

Settings

Field options

☑ This field represents the Entry title

□ Enable localization of this field

All the content can be translated to English (United States)

Validation

☑ Required field

You won't be able to publish an entry if this field is empty

□ Unique field

You won't be able to publish an entry if there is an existing entry with identical content

□ Limit character count

Specify a minimum and/or maximum allowed number of characters

□ Match a specific pattern

Make this field match a pattern: e-mail address, URI, or a custom regular expression

□ Prohibit a specific pattern

Make this field invalid when a pattern is matched: custom regular expression (e.g. bad word list)

□ Accept only specified values

You won't be able to publish an entry if the field value is not in the list of specified values

Default value

This setting allows you to set a default value for this field, which will be automatically inserted to new content entries. It can help editors avoid content entry altogether, or just give them a helpful prompt for how to structure their content.

0 characters Maximum 256 characters

Appearance

● Single line Default

○ URL

○ Dropdown

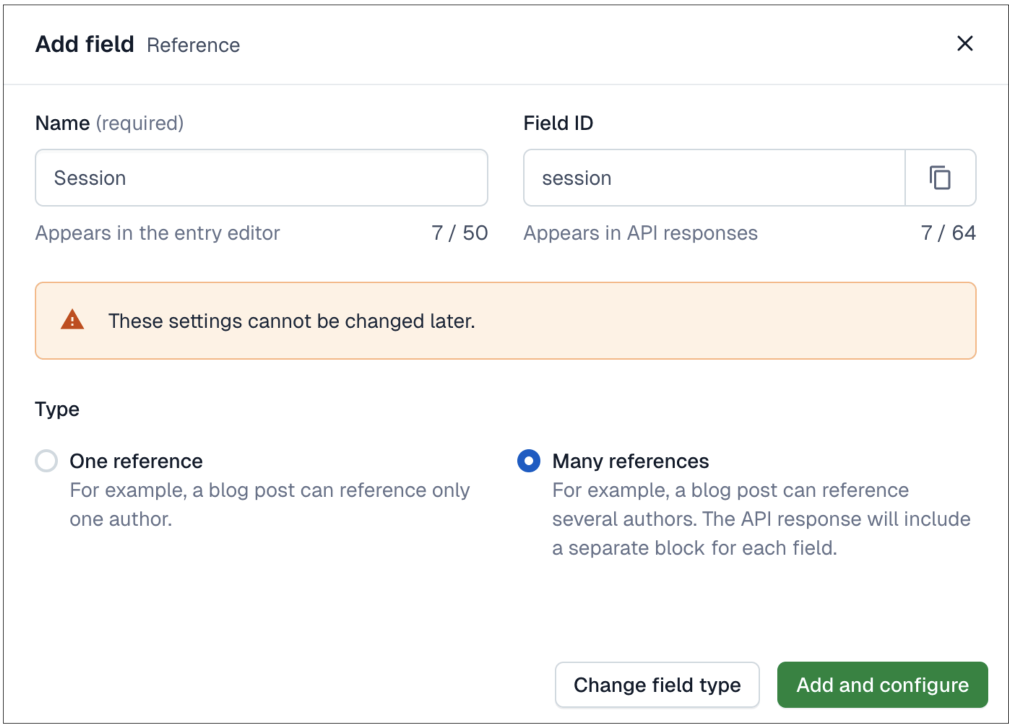
○ Radio

○ Slug

Help text

This help text will show up below the field 0 / 255

Figure 7.4: The interface for editing a field of type Text in Contentful. The field can be set as required or unique, among other things. [Screenshot taken by Yannik Rauter.]



Add field Reference

Name (required)

Session

Appears in the entry editor 7 / 50

Field ID

session

Appears in API responses 7 / 64

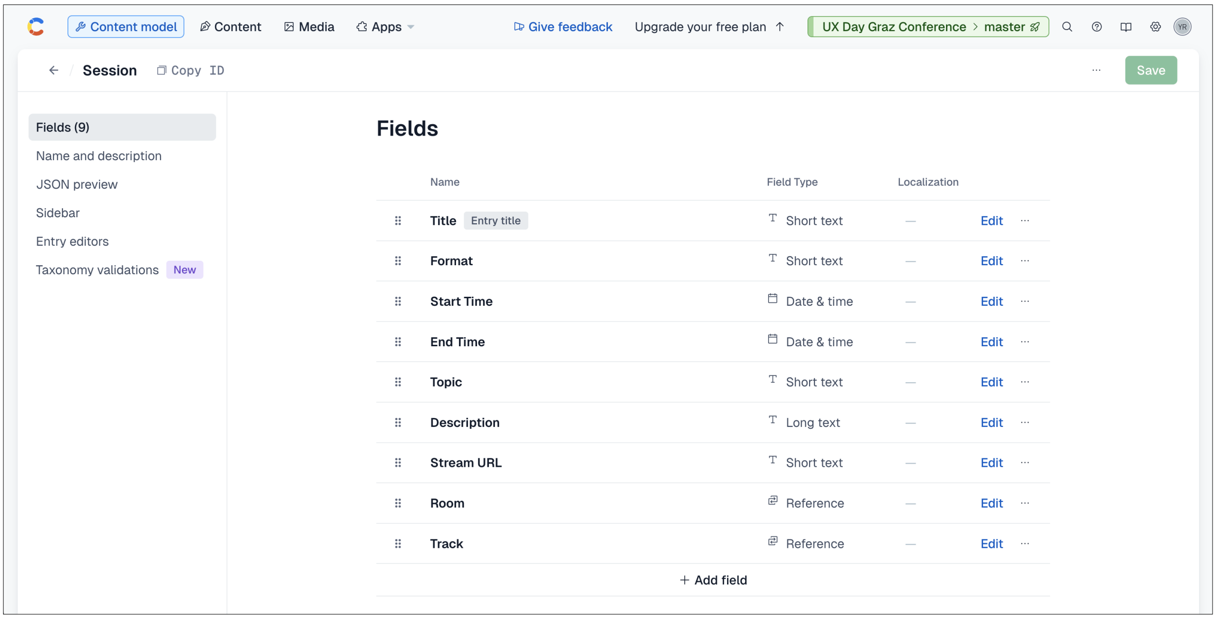
These settings cannot be changed later.

Type

○ One reference
For example, a blog post can reference only one author.

● Many references
For example, a blog post can reference several authors. The API response will include a separate block for each field.

Figure 7.5: Creating a reference in Contentful. [Screenshot taken by Yannik Rauter.]



Fields

Name	Field Type	Localization		
Title Entry title	Short text	—	Edit	···
Format	Short text	—	Edit	···
Start Time	Date & time	—	Edit	···
End Time	Date & time	—	Edit	···
Topic	Short text	—	Edit	···
Description	Long text	—	Edit	···
Stream URL	Short text	—	Edit	···
Room	Reference	—	Edit	···
Track	Reference	—	Edit	···

Figure 7.6: The interface for editing a content type and its fields in Contentful, by example of the Session content type. [Screenshot taken by Yannik Rauter.]

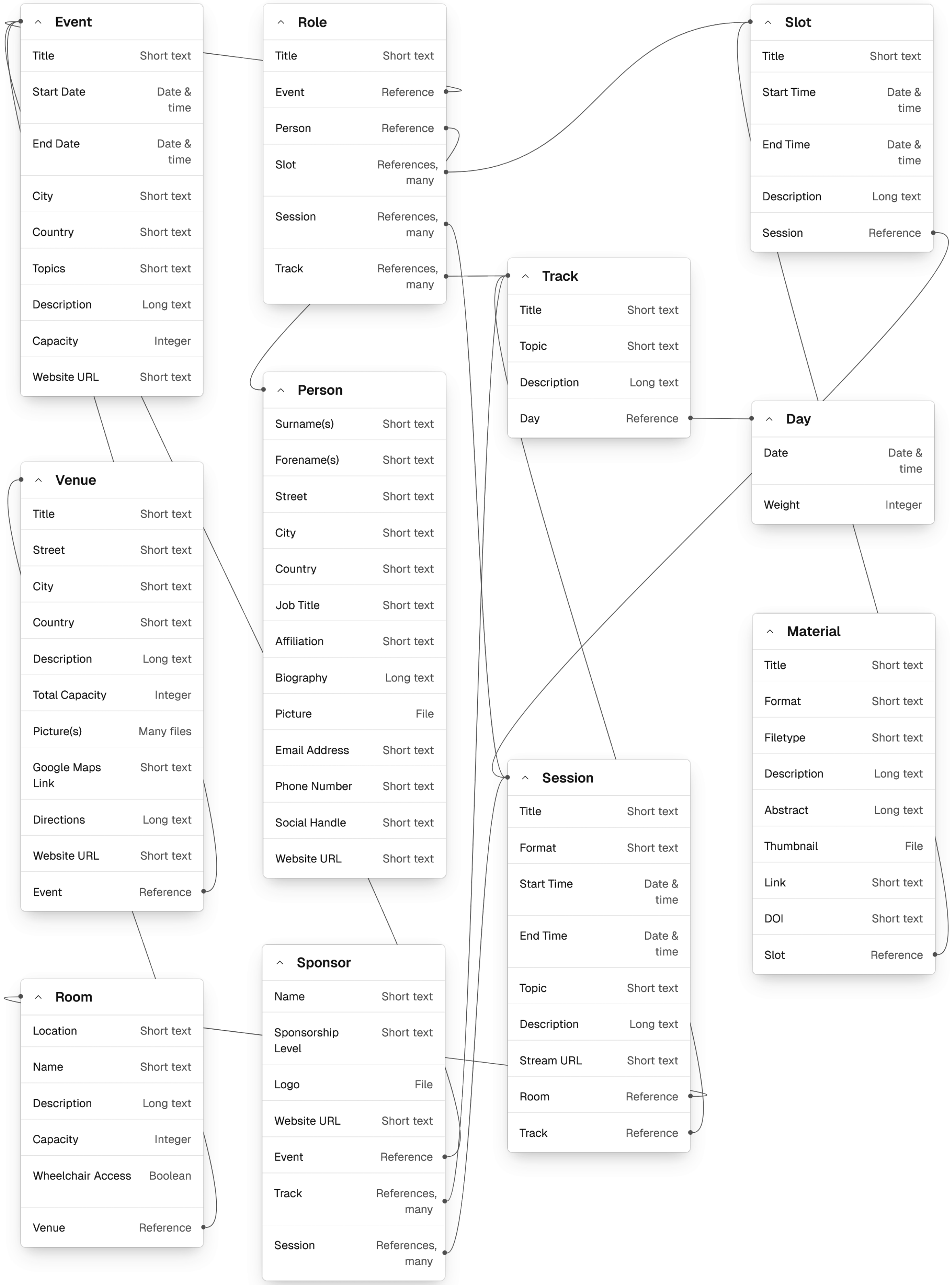
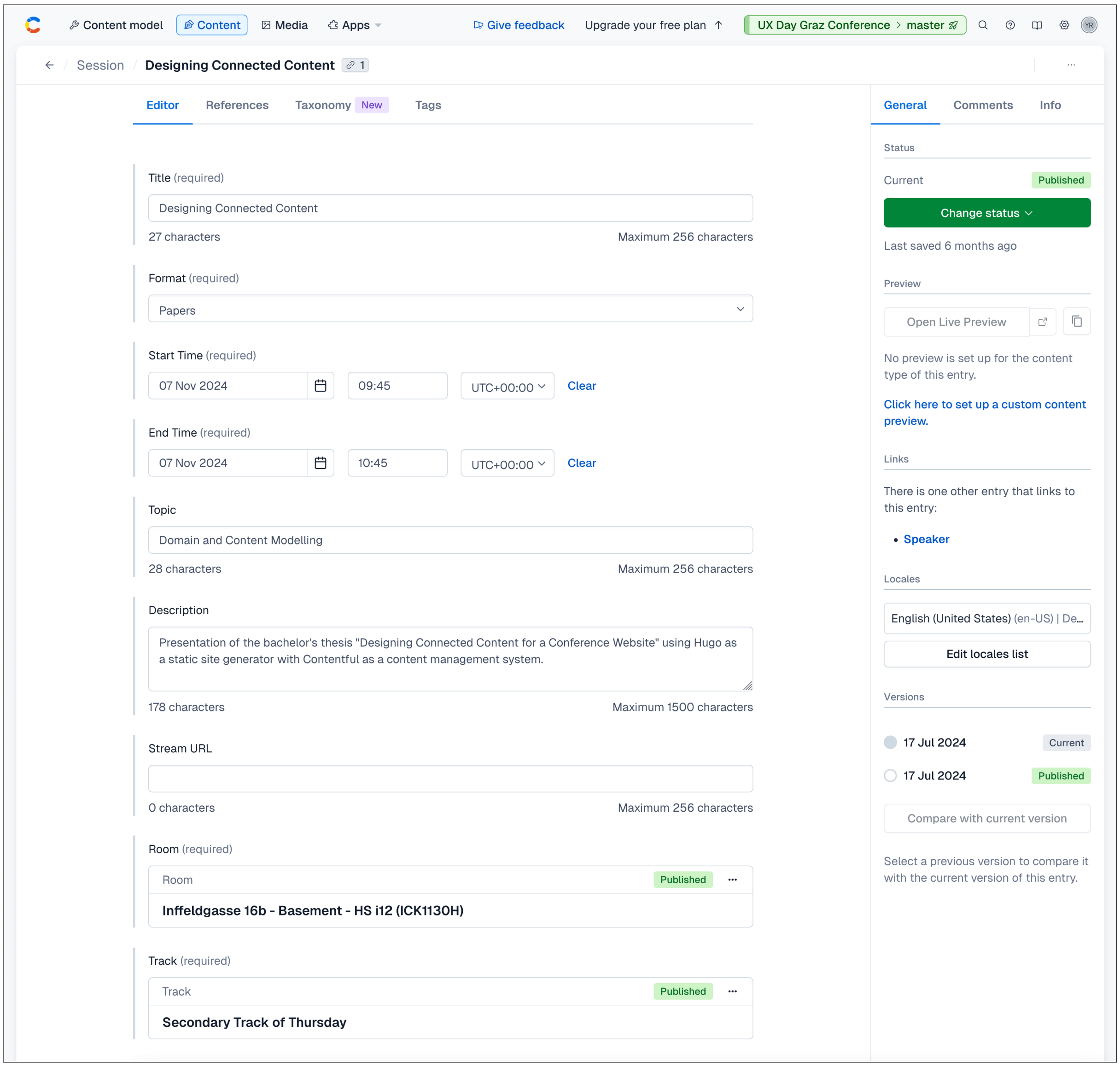


Figure 7.7: The content model for a conference web site, as implemented in and visualised by Contentful. Connecting lines cannot be redrawn manually. [Graphic exported from the Contentful's Visual Modeller.]



Content model Content Media Apps Give feedback Upgrade your free plan UX Day Graz Conference > master

Session / Designing Connected Content 1

Editor References Taxonomy New Tags

Title (required)

Designing Connected Content

27 characters Maximum 256 characters

Format (required)

Papers

Start Time (required)

07 Nov 2024 09:45 UTC+00:00 Clear

End Time (required)

07 Nov 2024 10:45 UTC+00:00 Clear

Topic

Domain and Content Modelling

28 characters Maximum 256 characters

Description

Presentation of the bachelor's thesis "Designing Connected Content for a Conference Website" using Hugo as a static site generator with Contentful as a content management system.

178 characters Maximum 1500 characters

Stream URL

0 characters Maximum 256 characters

Room (required)

Room Published ···

Inffeldgasse 16b - Basement - HS i12 (ICK1130H)

Track (required)

Track Published ···

Secondary Track of Thursday

General Comments Info

Status

Current Published

Change status

Last saved 6 months ago

Preview

Open Live Preview

No preview is set up for the content type of this entry.

Click here to set up a custom content preview.

Links

There is one other entry that links to this entry:

- Speaker

Locales

English (United States) (en-US) | De...

Edit locales list

Versions

○ 17 Jul 2024 Current

○ 17 Jul 2024 Published

Compare with current version

Select a previous version to compare it with the current version of this entry.

Figure 7.8: The interface for editing the fields of an entry in Contentful, by example of the Session content type. [Screenshot taken by Yannik Rauter.]

7.4 Creating Content for a Conference Web Site

Once the content model has been set up, content for a particular application can be entered. This is often done by non-technical staff. Figure 7.8 shows the Editor view for entering content into an instance of a Session. The values of its fields may be entered and modified. Outgoing references, such as Room and Track for a Session are also entered here. In addition, any incoming references to the entry are listed in the Links section of the right sidebar. Every new entry is assigned a randomly generated unique Entry ID (not based on any of its properties), which can be found by clicking the Info tab of the right sidebar.

7.5 Content API in Contentful

Contentful provides a variety of REST APIs [Contentful 2024a] for interacting with its content, which are available at separate endpoints. Those relevant in the scope of this thesis are:

- *Content Delivery API*: The Content Delivery API is exclusively meant for reading content. Its preferred use is to display data within a frontend application. Therefore, this is the interface mainly used in the implementation example, as described in Chapter 8.

- *Content Preview API*: The Content Preview API serves a similar purpose, but also includes unpublished entities (those with status Draft) in its response, making it suitable for development environments only.
- *Content Management API*: The Content Management API enables the manipulation of content by creating new entries or updating existing ones.
- *Images API*: The Images API allows images to be retrieved, but also manipulated in various ways such as cropping or altering the resolution.

Contentful also provides a GraphQL API, but that is not used in this project.

When requesting data, any content is delivered in JSON format, while assets are supplied as files. Across all these options, authentication is handled by including an API key in the form of a private access token with the HTTP request. This is possible using either the `Authorization` request header field, or by including the `access_token` URI query parameter along with the desired endpoint (as is done later in Section 8.4). API keys are configured within the Contentful web interface to permit access to all environments of a space.

7.6 Limitations of Contentful

Changing the data type for a field of a content type necessitates an elaborate, multi-step process. This is illustrated by the following example, using the Biography field (with a Field ID of biography) of the Person content type. To change this field's data type from Text to Rich text, the following steps are required:

1. The same Field ID cannot be used twice. Therefore, the biography field must first be renamed (including its Field ID), for example to biographyOld, in order to enable reuse of the existing biography Field ID.
2. A new field of the desired Rich text data type is created, using the previous name (and Field ID) biography, which is now available.
3. For each of the affected Person content type's entities, the contents are copied from the biographyOld field to the new biography field. This step may be assisted using the `contentful-migration` CLI [Contentful 2024d] and JavaScript.
4. The biographyOld field is deleted, since it is no longer needed.

Furthermore, Contentful limits the number of total API calls and the asset bandwidth, depending on the product tier being used [Contentful 2024g]. At time of writing, the free tier allows 100,000 API calls and 50 GB of asset downloads per month.

Chapter 8

Frontend: Hugo (SSG)

Content from a Content Management System (CMS) may be presented in various ways, only one of which is a web site. For example, Voice UIs and interactive agents are becoming increasingly popular [Atherton and Hane 2017, page 170]. For this project, the goal was to build a responsive web site for a small conference, which draws its content from the CMS described in the previous chapter. After some research, it was decided to build a static web site with a Static Site Generator (SSG). After exploring some of the available SSGs, Hugo was chosen. Designing a modern web site user interface also requires some knowledge of HTML, CSS, JavaScript, and responsive web design, as well as the mechanics of setting up an SSG.

8.1 Choosing a Static Site Generator

There are a large number of SSGs to choose from [Netlify 2024b]. The choice is largely independent of any previously selected CMS, since content will generally be accessed over an API. When fetching content at run time, it is typically fetched using JavaScript. When fetching content at build time to produce entirely static pages, it may either be fetched through a plug-in provided by the SSG, or a custom solution may be devised (say in Python over the CMS' API) to read content into separate files for parsing by the SSG.

Jekyll [Jekyll 2024] was one of the first SSGs, initially released in 2008. It is a simple, blog-aware SSG written in Ruby and utilising the Liquid templating engine [Luetke 2025]. Content may be read in either `.yaml`, `.json`, `.csv` or `.tsv` format. To this date, it remains one of the most popular choices thanks to its simplicity and performance, despite updates to its code and the addition of new features having become less frequent in recent years [Bleuzen 2023].

Many modern SSGs are written in JavaScript (or TypeScript) and use the Node environment. Metalsmith [Van Lierde 2025] and Eleventy [Leatherman 2025] are prominent examples. They are fully featured and have a large variety of plug-ins and extensions. However, Node-based SSGs have the disadvantage inherent with all Node projects that dependencies may change over time, and the SSG installation has to be maintained and kept up to date.

Hugo [Hugo 2025g] is another modern SSG solution written in Go [Go 2025], focussing on speed and flexibility. With build times of approximately one millisecond per included resource, Hugo generates even the most comprehensive web sites extremely quickly. Furthermore, Hugo strives to be highly configurable and is still actively being developed and frequently improved. It is available in two versions, standard and extended (which includes additional support for the WebP format and transpiling SASS). Hugo has a wide variety of publicly available, pre-built themes for the layout and styling of templates. Furthermore, Hugo is also distributed as a binary executable, thereby eliminating the need to maintain built-in dependencies. For these reasons, Hugo was selected to be used in the further implementation of this project.

8.2 Setting up Hugo

To start using Hugo, it is first necessary to follow the installation procedure for the applicable operating system. Next, a Hugo project must be initialised in a directory on the local file system using the command line. This creates the required directory structure [Hugo 2025b], as well as a `hugo.toml` file for the site's configuration. The most important directories in the scope of this thesis include:

- `content/`: Contains markup files, from which Hugo generates pages.
- `layout/`: Contains templates, which become pages when filled with content.
- `static/`: Contains files which are copied to the `public/` directory during the build process.
- `public/`: Contains the final static site ready for deployment, after being generated by the build process.

The site is now built by triggering the build process (utilising the command line again), which transforms content into static pages using templates. Hugo accepts a number of content formats as its input, with Markdown being among them. It creates a static output file (in this case, a web page as an `.html` file) for every `.md` file present in the `content/` directory, based on templates from the `layouts/` directory (and its subdirectories). The resulting files, along with additional, static resources like stylesheets or icons from the `static/` directory are stored in the `public/` directory, which may then be deployed on a web server.

8.3 Configuring Hugo

Configurations for Hugo are specified in either YAML, TOML, or JSON in a corresponding configuration file (`hugo.toml`, `hugo.yaml`, or `hugo.json`). Such a configuration file is required in every Hugo project. A wide range of optional parameters are used to configure various aspects of the site [Hugo 2025a]. Some are shown in the example in Listing 8.1, including:

- `cleanDestinationDir`: Removes all files from the output directory (`public`, by default) except those copied from the `static` directory.
- `disableKinds`: Disables rendering of certain unused kinds of pages, to reduce unnecessary overhead during the build process.
- `relativeURLs`: Transforms all URLs to be relative to the current page, enabling deployment regardless of the root path.
- `removePathAccents`: Removes non-spacing marks (like accents) from composite characters in the names of content files.
- `sectionPagesMenu`: Automatically defines a new parent menu entry for each top-level section of the site (i.e. for each content type).

These site parameters may be accessed from within a template as well. The `params` entry (and its children) actually serve just that purpose. For example, the list of values of a `speakerRoles` parameter determine which `Roles` are included in the Speakers navigation element.

```
cleanDestinationDir: true
disableKinds: ['taxonomy', 'term', 'RSS', 'sitemap']
languageCode: 'en-gb'
relativeURLs: true
removePathAccents: true
sectionPagesMenu: 'main'
title: 'Hugo Contentful Static - UX Day Graz 2024'

params:
  contactEmail: contact@example.com
  shortTitle: 'UX Day Graz 2024'
  speakerRoles:
    - 'Keynote Speaker'
    - 'Speaker'
```

Listing 8.1: The `hugo.yaml` file for the example conference web site, UX Day Graz 2024.

8.4 Fetching Content for Hugo

Hugo itself lacks any form of native integration for sourcing content from Contentful and storing it in individual, static content files to be used at build time. However, some third-party plug-ins, such as `contentful-hugo` [Sosso 2025] and `contentful-ssg` [JvM 2025a] are available for this. Both of these are written in TypeScript/JavaScript, and fetch content using Contentful's APIs. The content is then stored in separate files, according to Hugo's directory structure. A wide variety of options allow for proprietary configurations, tailored to a project's individual demands. Initially, `contentful-hugo` was used, but was then replaced by `contentful-ssg` for its improved customisability of file names.

By default, `contentful-ssg` only fetches textual content into files. However, content in this case is not exclusively text, but comes in various forms, including images and documents. When an entry references such an asset, the URL at which that asset is available from Contentful's API is placed as a value in the attribute by `contentful-ssg`. Therefore, the asset will be loaded from the CMS at run time, creating unwanted dynamic accesses, while simultaneously creating the potential for issues with the hosting service's Content Security Policy (CSP). It is better practice to provide these assets via the same web server which hosts the static site's markup, which requires the assets to be sourced from the CMS at build time. For this purpose, `contentful-ssg` itself requires a plug-in, `cssg-plugin-assets` [JvM 2025b]. The incompatibility of `cssg-plugin-assets` and its dependencies with the Windows operating system caused problems during the build process, leading to this plug-in (along with the requirement for static assets) being abandoned. Thus, assets in the current implementation remain being directly and dynamically accessed from the Contentful Content Delivery API at run time.

Similar to Hugo, `contentful-ssg` also comes with many options for configuration, all specified in its `contentful-ssg.config.js` file [JvM 2025c]. These also include *transform hooks*, called for every entry loaded from Contentful. These can be used to manipulate or massage the front matter entries within generated content files, for example to include a title and publication date, or to specify the ordering of items in a navigation menu (for example, days in the Programme).

Another example is the ability to infer file names from (a combination of) specific attributes per content type. Since the URL for a piece of content in Hugo depends on its file name, this enables predictable URLs. Instead of the default entry identifier (a unique, randomised, alphanumeric string automatically generated by Contentful for every entry), any attribute (like the title of a Session) may be used. The potential for duplicate file names (when different sessions have similar titles) must be considered, to avoid unintentionally overwriting files and causing other issues. Note that changing an attribute's value also breaks bookmarks and permanent links, since the URL changes as well.

Transform hooks can only operate on the content file currently being generated. They cannot operate across content files. For more flexibility, the task runner Gulp [Gulp 2024] was used in a post-processing step to further massage and manipulate content files, for example to list certain `Person`s as `Speakers` rather than `Team` members, depending on their `Role`.

For the conference web site in this project, all content fetched from the CMS was actually stored in YAML variables in the front matter of `.md` files, one file for each instance of each content type. Hence, the `.md` files themselves contain only YAML front matter; they do not contain any actual Markdown as such. An example of a content file can be seen in Listing 8.2.

8.5 Templates in Hugo

Templates are used to control the rendering of content and assets into static HTML pages. In Hugo, templates are `.html` files extended with Go code. Templates have different purposes distinguished by their file names and specific locations within the directory structure. During the build process, a template is populated with data from the content files, producing static HTML output files. An intricate, nested structure of template types combine to form a complete web site [Hugo 2025f].

Base templates serve as a wrapper for other templates, providing a frame (including meta tags, stylesheets, scripts, and perhaps site navigation) within which any page content may be placed. *Single* templates are used for the pages of individual instances of one content type. Therefore, every core content type requires its own single template. They reside as `single.html` in the subdirectory of their respective content type. A *section* template on the other hand only lists the various instances of a content type, without exposing their details. They are saved in the subdirectory of their respective content type as well, and are named `list.html`. Any of these three template types may be supplemented with one or more partial templates, which represent reusable components, and are located in the `partials` subdirectory.

When determining the correct template for a certain page, the build process follows a predefined template lookup order, highly dependent on the content type [Hugo 2025e]. Those templates placed directly in the subdirectory of a certain content type have the highest specificity and are only used for the pages of that one content type. Default templates may be added as a fallback, in case no other (more specific) template exists. However, these are rather limited in functionality, since they must be compatible with any content type. Furthermore, generic base templates can be created to serve as a wrapper, into which the more specific templates are placed. These may include additional navigation elements like breadcrumbs, used by all single and section templates in general.

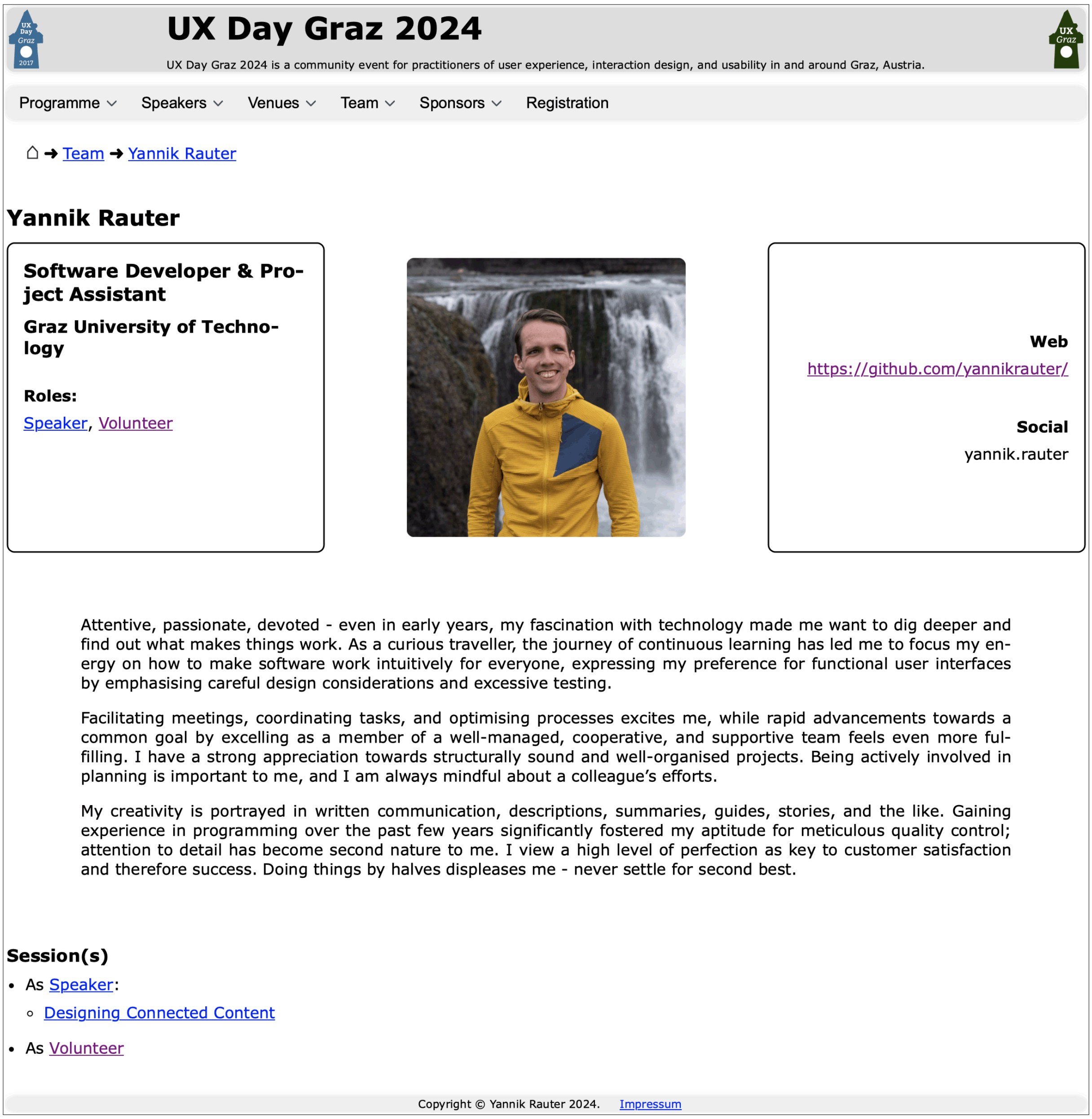
While templates mainly access and render the actual content, they may also utilise parameters from so-called *front matter* included in the content files, which is metadata prepended to the actual content [Hugo 2025c], usually in YAML format. This includes content-related attributes (such as a title, or a date of creation or last update). Additional front matter parameters might concern page setup (publishing date, menu parent, navigation weight, special template type) and also relationships with other content. Listing 8.3 shows an example of a single template file to generate a HTML page for an instance of a `Person`. It contains HTML elements with embedded instructions for including content from YAML variables.

```
---
defaultMetaFields:
  sys:
    id: 3D7dG2alyBMgUBKPupHOg7
    contentType: person
    createdAt: ’2023-06-12T21:04:47.621Z’
    updatedAt: ’2024-07-17T09:56:52.305Z’
title: Yannik Rauter
menu:
  main:
    parent: speaker
surnames: Rauter
forenames: Yannik
street: Example Street 1
city: Klagenfurt
country: Austria
jobTitle: Software Developer & Project Assistant
affiliation: Graz University of Technology
biography: >-
  Attentive, passionate, devoted - even in early years, my fascination with
  technology made me want to dig deeper and find out what makes things work. As
  a curious traveller, the journey of continuous learning has led me to focus my
  energy on how to make software work intuitively for everyone, expressing my
  preference for functional user interfaces by emphasising careful design
  considerations and excessive testing.

  Facilitating meetings, coordinating tasks, and optimising processes excites
  me, while rapid advancements towards a common goal by excelling as a member of
  a well-managed, cooperative, and supportive team feels even more fulfilling. I
  have a strong appreciation towards structurally sound and well-organised
  projects. Being actively involved in planning is important to me, and I am
  always mindful about a colleague’s efforts.

  My creativity is portrayed in written communication, descriptions, summaries,
  guides, stories, and the like. Gaining experience in programming over the past
  few years significantly fostered my aptitude for meticulous quality control;
  attention to detail has become second nature to me. I view a high level of
  perfection as key to customer satisfaction and therefore success. Doing things
  by halves displeases me - never settle for second best.
picture:
  mimeType: image/jpeg
  url: >-
    //images.ctfassets.net/8au3rnz56kwt/yannik_rauter_profile_picture.jpg
  title: Yannik Rauter Profile Picture
  description: ’’
  width: 1100
  height: 1100
  fileSize: 780281
emailAddress: example@student.tugraz.at
phoneNumber: ’+436641234567’
socialHandle: yannik.rauter
websiteUrl: https://github.com/yannikrauter/
---
```

Listing 8.2: The .md file generated for a person called Yannik Rauter, an instance of the Person content type. It contains only front matter in YAML format.

```
{{ define "main" }}
<link rel="stylesheet" property="stylesheet" href="{{ "css/person.css" | relURL }}">
<main>
  <article>
    <header>
      <h1>{{ .Title }}</h1>
      <div id="person-information-container">
        <div id="person-details">
          {{ with .Param "jobTitle" }}
            <h2>{{ . }}</h2>
          {{ else }}
            <p>No job title provided</p>
          {{ end }}
          {{ with .Param "affiliation" }}
            <h3>{{ . }}</h3>
          {{ else }}
            <p>No affiliation provided</p>
          {{ end }}
          <br>
          {{ $rolesCounter := 0 }}
          {{ range $roleE := where $.Site.RegularPages "Type" "role" }}
            {{ if eq $.Params.defaultMetaFields.sys.id $roleE.Params.person.id }}
              {{ $rolesCounter = add $rolesCounter 1 }}
            {{ end }}
          {{ end }}
          {{ if eq 0 $rolesCounter }}
            <h4>Role:</h4>
            <p>This person has no role.</p>
          {{ else if eq 1 $rolesCounter }}
            <h4>Role:</h4>
          {{ else }}
            <h4>Roles:</h4>
          {{ end }}
          <p>
            {{ range $index, $roleE := where $.Site.RegularPages "Type" "role" }}
              {{ if eq $.Params.defaultMetaFields.sys.id $roleE.Params.person.id }}
                {{ $rolesCounter = sub $rolesCounter 1 }}
                {{ if eq 0 $rolesCounter }}
                  <a href="{{ .RelPermalink }}">{{ .Params.title }}</a>
                {{ else }}
                  <a href="{{ .RelPermalink }}">{{ .Params.title }}</a>,
                {{ end }}
              {{ end }}
            {{ end }}
          </p>
        </div>
        ...
      </div>
    </header>
    ...
  </article>
</main>
{{ end }}
```

Listing 8.3: The single template file for the `Person` content type is located in the `layouts/person/single.html`. It contains HTML elements with embedded instructions for including content from YAML variables. The example has been shortened for brevity.



UX Day Graz 2024

UX Day Graz 2024 is a community event for practitioners of user experience, interaction design, and usability in and around Graz, Austria.

⌂ ➜ Team ➜ Yannik Rauter

Yannik Rauter

Software Developer & Project Assistant

Graz University of Technology

Roles:

Speaker, Volunteer

Web

https://github.com/yannikrauter/

Social

yannik.rauter

Attentive, passionate, devoted - even in early years, my fascination with technology made me want to dig deeper and find out what makes things work. As a curious traveller, the journey of continuous learning has led me to focus my energy on how to make software work intuitively for everyone, expressing my preference for functional user interfaces by emphasising careful design considerations and excessive testing.

Facilitating meetings, coordinating tasks, and optimising processes excites me, while rapid advancements towards a common goal by excelling as a member of a well-managed, cooperative, and supportive team feels even more fulfilling. I have a strong appreciation towards structurally sound and well-organised projects. Being actively involved in planning is important to me, and I am always mindful about a colleague’s efforts.

My creativity is portrayed in written communication, descriptions, summaries, guides, stories, and the like. Gaining experience in programming over the past few years significantly fostered my aptitude for meticulous quality control; attention to detail has become second nature to me. I view a high level of perfection as key to customer satisfaction and therefore success. Doing things by halves displeases me - never settle for second best.

Session(s)

- As Speaker:
  - Designing Connected Content
- As Volunteer

Copyright © Yannik Rauter 2024. Impressum

Figure 8.1: An example single web page for a Person entry on the UX Day Graz 2024 web site. A navigation bar and breadcrumb bar are included at the top of the page. [Screenshot taken by Yannik Rauter.]

8.6 Modelling a Conference Web Site in Hugo

Before creating the first template, the core content types must be selected. These include Day, Material, Person, Role, Room, Session, Slot, Sponsor and Venue. While all of these require their own templates for presentation (either single, list, or both), only some are included in the navigation. As can be seen in Figure 8.1, the navigation elements consist of Programme, Speakers, Venues, Team, Sponsors and Registration. Registration is an individual page not sourced from Contentful, similar to the Home and Impressum pages.

Speakers and Team are examples of section templates which do not simply list all entries of a content type, but instead only compile those matching certain criteria. In this case, only Persons with the Roles of Keynote Speaker or Speaker are included in the Speakers navigation element, while all other Persons are listed under the Team navigation element.

```
 1 ---
 2 defaultMetaFields:
 3   sys:
 4     id: 3bWliu1Syli9TiZ8Ciy0d8
 5     contentType: room
 6     createdAt: '2023-06-22T16:09:49.978Z'
 7     updatedAt: '2024-05-21T07:07:36.411Z'
 8 title: Inffeldgasse 16b - Basement - HS i13 (ICK1120H)
 9 location: Inffeldgasse 16b - Basement - HS i13 (ICK1120H)
10 name: Hörsaal i13
11 description: Largest lecture hall at TU Graz Campus Inffeldgasse.
12 capacity: 301
13 wheelchairAccess: true
14 venue:
15   id: 7bqSwYTFonmgJGzkYW1GuT
16   contentType: venue
17 ---
```

Listing 8.4: The Markdown file of an entry for the Room content type.

Through transclusion (pulling in related content), Programme composes a list of Days, which in turn shows a timetable of all the Sessions for that Day, along with their respective Slots. Linking to these related entries enables easy navigation and encourages explorative traversal of the site. While doing so, maintaining clarity on where the additional information is coming from is crucial to avoid confusing the user's sense of location. Thus, breadcrumbs always conveniently label the current page as well as indicate its place in the site's hierarchy.

Considering Figure 8.1 again, it becomes clear that not all fields available in the content model are also exposed on the final page. For example, Listing 8.2 contains further properties (like address and contact details) which should not be publicly exposed. Some information might also be withheld from the user temporarily, and displayed at a later point in time when it becomes relevant.

Accessing information of connected entries via references is illustrated by two examples, using the Venue and Room content types. In the first case, the reference to the related Venue entry `Inffeld` is stored directly in the source Room entry `i13`. The front matter of the Markdown file for `i13` thus includes the entry identifier (a unique, randomised, alphanumeric string automatically generated by Contentful for every entry) of `Inffeld`, as shown in line 15 of Listing 8.4. Using this ID, the corresponding entry `Inffeld` can be found, by iterating all entries of the related content type Venue. This is done in Hugo by using a combination of the `range` loop function over all pages, along with a filter through the `where` function, limiting the search to entries of the Venue content type. Finally, a simple `if` statement triggers only for the matching entry. This is shown in Listing 8.5.

In comparison, consider the inverted case of listing all related Room entries based on a source Venue entry `Inffeld`. Since entity references are always placed on just one side of the relationship, the source Venue entry `Inffeld` has no list of references to its rooms (as can be seen in Listing 8.6). Instead, each of the related rooms holds the reference to its venue `Inffeld`. The code logic remains similar. For the ID of the source Venue entry `Inffeld`, all related rooms having that source ID must be found. This requires iterating through the list of all Room entries, as shown in Listing 8.7. For improved readability, a variable name of `\$roomElement` has been assigned to the current loop element.

Nesting these structures, for example when seeking the Slots for the individual Sessions of all Days

```
{{ range where $.Site.RegularPages "Type" "venue" }}
  {{ if eq .Params.defaultMetaFields.sys.id $.Params.venue.id }}
    <h2>Venue: <a href="{{ .RelPermalink }}">{{ .Title }}</a></h2>
  {{ end }}
{{ end }}
```

Listing 8.5: An example of a loop in Hugo, used to target a specific instance of a content type.

```
---
defaultMetaFields:
  sys:
    id: 7bqSwYTFonmgJGzkYW1GuT
    contentType: venue
    createdAt: ’2023-06-22T16:05:59.998Z’
    updatedAt: ’2024-05-21T07:14:08.981Z’
title: Graz University of Technology - Campus Inffeldgasse
menu:
  main:
    parent: venue
street: Inffeldgasse
city: Graz
country: Austria
description: >-
  The Inffeldgasse Campus is the largest of the three TU Graz campus sites.
  Currently covering an area of around 124,000 square meters.

  The event takes place within the lecture halls located at this campus of Graz
  University of Technology.
totalCapacity: 509
directions: >-
  [Building number 16b](https://goo.gl/maps/mF91UKA6cFhm2V2cA) is located
  behind building number 10, so walk around that structure to find it.
  Head inside, down the stairs to your right, where you’ll find all the
  relevant lecture halls.
event:
  id: 2l2l9szONnUfytINRNo5vg
  contentType: event
---
```

Listing 8.6: The Markdown file of an entry for the Venue content type.

```
{{ range $roomElement := where $.Site.RegularPages "Type" "room" }}
  {{ if eq $.Params.defaultMetaFields.sys.id $roomElement.Params.venue.id }}
    <li>
      <a href="{{ $roomElement.RelPermalink }}">{{ $roomElement.Params.title }}</a>
    </li>
  {{ end }}
{{ end }}
```

Listing 8.7: An example of a loop in Hugo, used to list all entries of a content type matching a particular criterion.

to compile a complete conference Programme, quickly becomes convoluted and confusing, stretching the boundaries of Hugo. To alleviate this problem, partial templates help keep individual code files short, while increasing code reusability. Example applications of partials include breadcrumbs, headers, and footers. This enables the single and list templates of the Day content type to both contain only a few lines of code, by importing large amounts of content from the `day-tracks-list` with just a single line of code:

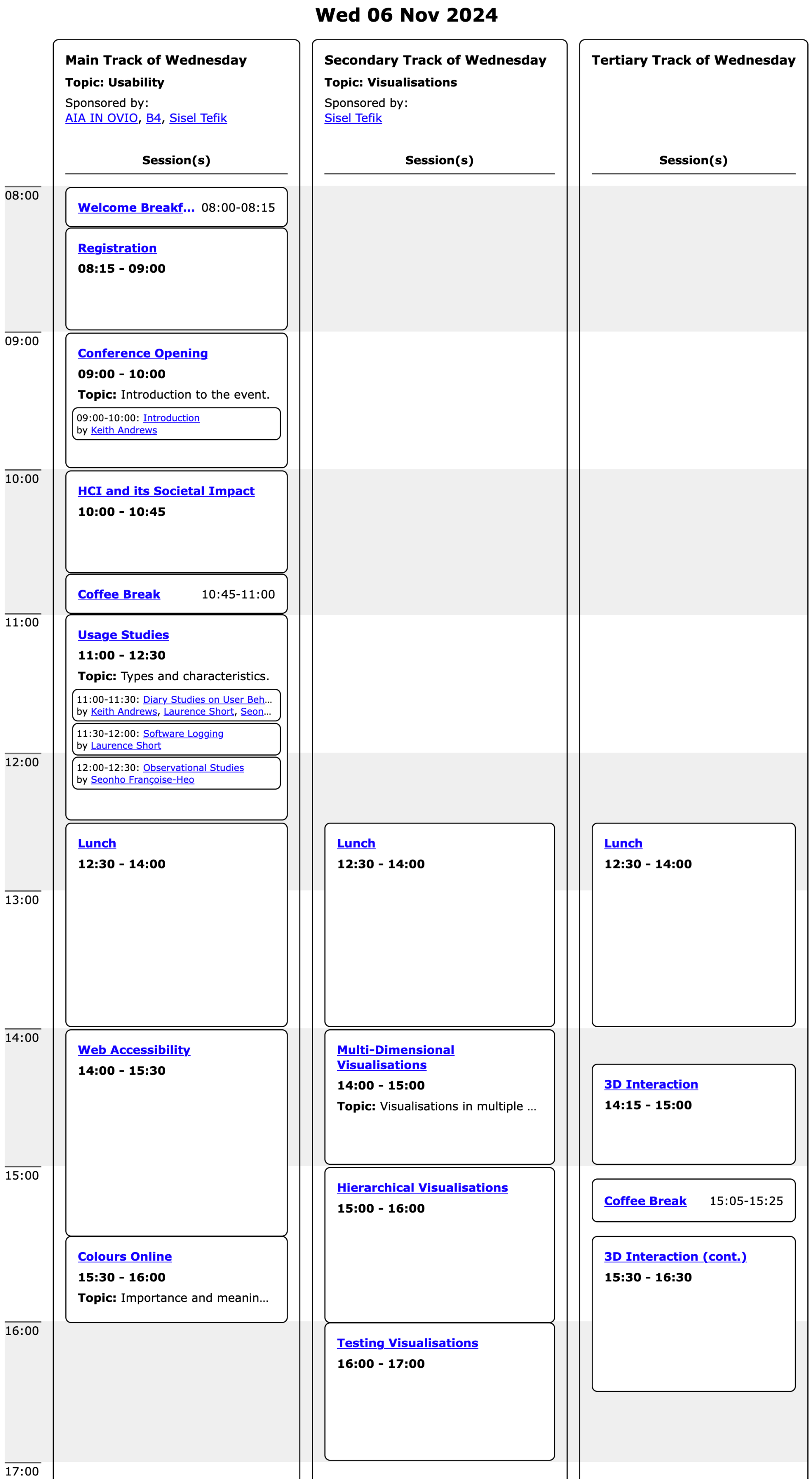
```
{{ partial "day-tracks-list" . }}
```

Functionality like this truly highlights the strengths of SSGs.

8.7 Conference Programme Using CSS Subgrid

To display the Programme of the conference, an overview of all `Sessions` was assembled. Content from a wide range of content types is collected, including `Tracks`, `Sponsors`, `Slots` and `Persons`. The Programme may be viewed for each individual `Day` alone, or for a list of all `Days` on one page. To realise this, the entire Programme of a `Day` is defined by only one partial. This partial is then used either just once (in the single template of Day), or once per `Day` (in the list template of Day) to view the whole Programme. The Programme of first `Day` of the conference is shown in Figure 8.2.

Topics and `Sponsors` of `Tracks` are included at the top, immediately below the title of the respective `Track`. Using CSS Subgrid [MDN 2024a], the `Sessions` are aligned according to their times across different `Trackss` in intervals of 5 minutes. Within each `Session`, its `Slots` are listed, along with their speakers (certain `Persons` with corresponding `Roles`). Content extending the boundaries of its block (the border around a `Track`, `Session`, or `Slot` entity) is shortened using ellipsis (except for Session titles, which may wrap lines multiple times if possible). In case of a `Session` with a duration of less than 30 minutes, a more compact layout is employed. `Slots` always have a fixed height, regardless of their title's length and their number of speakers. All this requires extensive use of variables, loops and conditional branches in Hugo code, mixed in with HTML tags to structure the page, as well as CSS classes for alignment and styling. This results in complex looking code, which requires knowledge of all these technologies to be understood. A snippet of the partial used to create the Programme is shown in Listing 8.8.



Wed 06 Nov 2024

Main Track of Wednesday

Topic: Usability

Sponsored by:
AIA IN OVIO, B4, Sisel Tefik

Session(s)

Welcome Breakf... 08:00-08:15

Registration
08:15 - 09:00

Conference Opening
09:00 - 10:00
Topic: Introduction to the event.

09:00-10:00: Introduction
by Keith Andrews

HCI and its Societal Impact
10:00 - 10:45

Coffee Break 10:45-11:00

Usage Studies
11:00 - 12:30
Topic: Types and characteristics.

11:00-11:30: Diary Studies on User Beh...
by Keith Andrews, Laurence Short, Seon...

11:30-12:00: Software Logging
by Laurence Short

12:00-12:30: Observational Studies
by Seonho Françoise-Heo

Lunch
12:30 - 14:00

Web Accessibility
14:00 - 15:30

Colours Online
15:30 - 16:00
Topic: Importance and meanin...

Secondary Track of Wednesday

Topic: Visualisations

Sponsored by:
Sisel Tefik

Session(s)

Lunch
12:30 - 14:00

Multi-Dimensional Visualisations
14:00 - 15:00
Topic: Visualisations in multiple ...

Hierarchical Visualisations
15:00 - 16:00

Testing Visualisations
16:00 - 17:00

Tertiary Track of Wednesday

Session(s)

Lunch
12:30 - 14:00

3D Interaction
14:15 - 15:00

Coffee Break 15:05-15:25

3D Interaction (cont.)
15:30 - 16:30

Figure 8.2: The Programme page for a conference, implemented using CSS Subgrid. [Screenshot taken by Yannik Rauter.]

```
<div class="subgrid-item-session" style="grid-row: {{ $lowB }} / {{ $upB }};">
  <a href="{{ $sessionE.RelPermalink }}"><h4>{{ $sessionE.Params.title }}</h4></a>
  <p><b>{{ (time.AsTime $sessionE.Params.startTime).Format "15:04" }} -
    {{ (time.AsTime $sessionE.Params.endTime).Format "15:04" }}</b></p>
  {{ with $sessionE.Param "topic" }}
    <p class="nowrap" title="{{ . }}"><b>Topic:</b> {{ . }}</p>
  {{ end }}
  <div class="slot-container">
    {{ $slotsCounter := 0 }}
    {{ $linkedCT := "slot" }}
    {{ range $slotElementFromAll :=
      sort (where $.Site.RegularPages "Type" $linkedCT) "Params.startTime" }}
      {{ if eq $sessionE.Params.defaultMetaFields.sys.id
        $slotElementFromAll.Params.session.id }}
        {{ $personsPerSlotCounter := 0 }}
        {{ $linkedCT := "person" }}
        {{ range $personElement := where $.Site.RegularPages "Type" $linkedCT }}
          {{ $linkedCT := "role" }}
          {{ range $roleElement := where $.Site.RegularPages "Type" $linkedCT }}
            {{ if eq $personElement.Params.defaultMetaFields.sys.id
              $roleElement.Params.person.id }}
              {{ range $slotElementFromRole := $roleElement.Params.slot }}
                {{ if eq $slotElementFromAll.Params.defaultMetaFields.sys.id
                  $slotElementFromRole.id }}
                  {{ if eq $personsPerSlotCounter 0 }}
                    {{ $slotsCounter = add $slotsCounter 1 }}
                    <p class="slot-container-item nowrap" id="slot-element-{{
                        $slotsCounter}}-session-{{$sessionsCounter}}">
                    {{ (time.AsTime $slotElementFromAll.Params.startTime).Format "
                        15:04" }}-{{ (time.AsTime $slotElementFromAll.Params.endTime
                        ).Format "15:04" }}:
                    <a href="{{ $slotElementFromAll.RelPermalink }}" title="{{
                        $slotElementFromAll.Params.title }}">{{ $slotElementFromAll.
                        Params.title }}</a>
                    <br>
                    by <a href="{{ $personElement.RelPermalink }}"
                      title="{{$personElement.Params.forenames}} {{$personElement.
                          Params.surnames}}">{{$personElement.Params.forenames}} {{
                          $personElement.Params.surnames}}</a>,</p>
                  {{ else }}
                    <a href="{{$personElement.RelPermalink}}" title="{{
                        $personElement.Params.forenames}} {{$personElement.Params.
                        surnames}}">{{$personElement.Params.forenames}} {{
                        $personElement.Params.surnames}}</a>,
                  {{ end }}
                  {{ $personsPerSlotCounter = add $personsPerSlotCounter 1 }}
                {{ end }}
              {{ end }}
            {{ end }}
          {{ end }}
        {{ end }}
        </p>
        ...
      {{ end }}
    {{ end }}
  </div>
</div>
```

Listing 8.8: A code snippet of the conference Programme code in Hugo using CSS Subgrid.

8.8 Limitations of Hugo

Despite its benefits, Hugo also has some shortcomings. To begin with, it does not support Contentful as a source of content by itself. This necessitates the use of third-party plug-ins like `contentful-ssg`, which are limited in support and functionality. An unwanted restriction imposed by `contentful-ssg` is the use of Markdown instead of the preferred choice of HTML. Out of the four available output formats from `contentful-ssg`, Hugo only accepts one (Markdown) as an input format, eliminating all other options. Apart from this, while Hugo is still actively being developed and regularly updated with new features, the same cannot always be said about such plug-ins.

Certain requirements of stakeholders turned out to be impossible to implement, limited by Hugo's capabilities at the time. One such example is in the navigation elements, where any `Person` entry can either be listed below Speakers or Team, but never both. It is not possible to specify two different main menu parents in Hugo. Moreover, the main menu elements are limited to a list of strings, which is compiled from those pieces of content specifying a main menu parent in their front matter. This list is iterated in a base template to derive the navigation elements. However, since this list of strings lacks any reference to the related page of each entry, the front matter parameters of those pages are inaccessible. Thus, problems arise when attempting, say, to sort the navigation entries alphabetically by surname, since the value of that parameter cannot be read.

Minor irritations are caused by Hugo's peculiar habit of leaving blank lines in the output HTML code for every line of Go code in the input template. This often results in large blocks of empty space, a feature which can only be disabled globally by minifying the output. While this solution is fine for a production environment, it remains inconvenient during development. One workaround is to use *template action delimiters* with hyphens [Hugo 2025d], a fix to be aware of sooner rather than later, since it entails large amounts of refactoring.

Finally, the URLs for Hugo's pages depend not only on the file name of the piece of content, but also on the name of the directory it resides in. When using `contentful-ssg`, this directory name is derived from the corresponding content type's name. While an alias may be used to provide an alternative URL (other than the file name) for any resource, the content type path is not configurable. For this reason, every `Person` entry will always be published at `/person/entryFileName/`, regardless of being listed in the navigation below Speakers or Team, leading to illogical URLs. Changing this `/person` path requires renaming the `person/` directory, which breaks the functionality for resolving entry references using IDs and loops (as described in Section 8.6). Once more, this shows Hugo's lack of customisability in some cases, despite its extreme flexibility in many other areas.

Chapter 9

Outlook and Future Work

As is often the case in software engineering, there are numerous ways to go about achieving a desired outcome. In this case, that outcome is enabling access to well-structured content via an easy to use, visually appealing web site for end users. The methods presented in this thesis (using Contentful as a CMS, and Hugo as an SSG) are just one example of a possible implementation. Some of the limitations imposed by Contentful and Hugo described in the previous chapters might be circumvented by employing different products instead. Further alternatives, which might be explored when considering a similar application, include:

- A static approach using another CMS, like Strapi [Strapi 2024].
- A static approach using another SSG, especially one with native support by Contentful, like Jekyll [Jekyll 2024] or Metalsmith [Van Lierde 2025].
- A dynamic approach, loading content from the CMS only upon the client's request of the page, with or without using an SSG.

The latter is quite different in nature to the solution described in this thesis. While accessing content dynamically results in fewer limitations for displaying single pages, compiling a list of navigation elements might prove difficult. Furthermore, the performance of the content API becomes a relevant factor (especially concerning assets). Depending on the CMS being used, calls to the API might be priced (see Contentful [2024g]). The process of publishing content changes must be realigned across the involved authors, considering the immediate implications of such actions on live web sites.

Using a Static Site Generator alongside dynamic requests to the Content Management System enables a sort of hybrid solution. While templates are no longer instantiated for every entity, the single active instance is instead populated directly with data from the API. Furthermore, this still maintains the advantages of reusable partials (which entails a reduced overhead in regard to recurring structures across pages) and simple, site-wide design changes.

Even though any proprietary solution like the one presented in this thesis will always come with its own individual challenges, an SSG other than Hugo has the potential to alleviate at least some of the recently highlighted issues. Modern problems require modern solutions, and software development always involves a certain level of tinkering. Finding a combination of the right technologies to aid this process makes all the difference between a good product and a great one.

Chapter 10

Concluding Remarks

This thesis described the process of Designing Connected Content with an example implementation in the form of a conference web site, using the Contentful Headless CMS as a data store and the Hugo SSG to present said content through a web static site. The code can be found at Rauter [2025].

One achievement is highlighting the advantages of this approach to creating a static web site for enabling access to structured content. While the steps to create a domain model and a content model involve significant effort, the preparations and research yield high returns when it comes to entering and later displaying content. The reasoning behind certain design decisions is explained in some detail, providing guidance for similar projects in the future.

Moreover, the practical implementation of the UX Day Graz 2024 conference web site serves as a solid foundation for any small- to medium-sized event of this category in the future. Many stepping stones along the way were only uncovered thanks to this realisation in code, showing that planning and theory alone are cannot sufficiently consider all outcomes.

Appendix A

Full Page Figures

For improved readability, Figures A.1 and Figure A.2 show the final domain model and final content model for a small conference web site, respectively, as full-page figures.

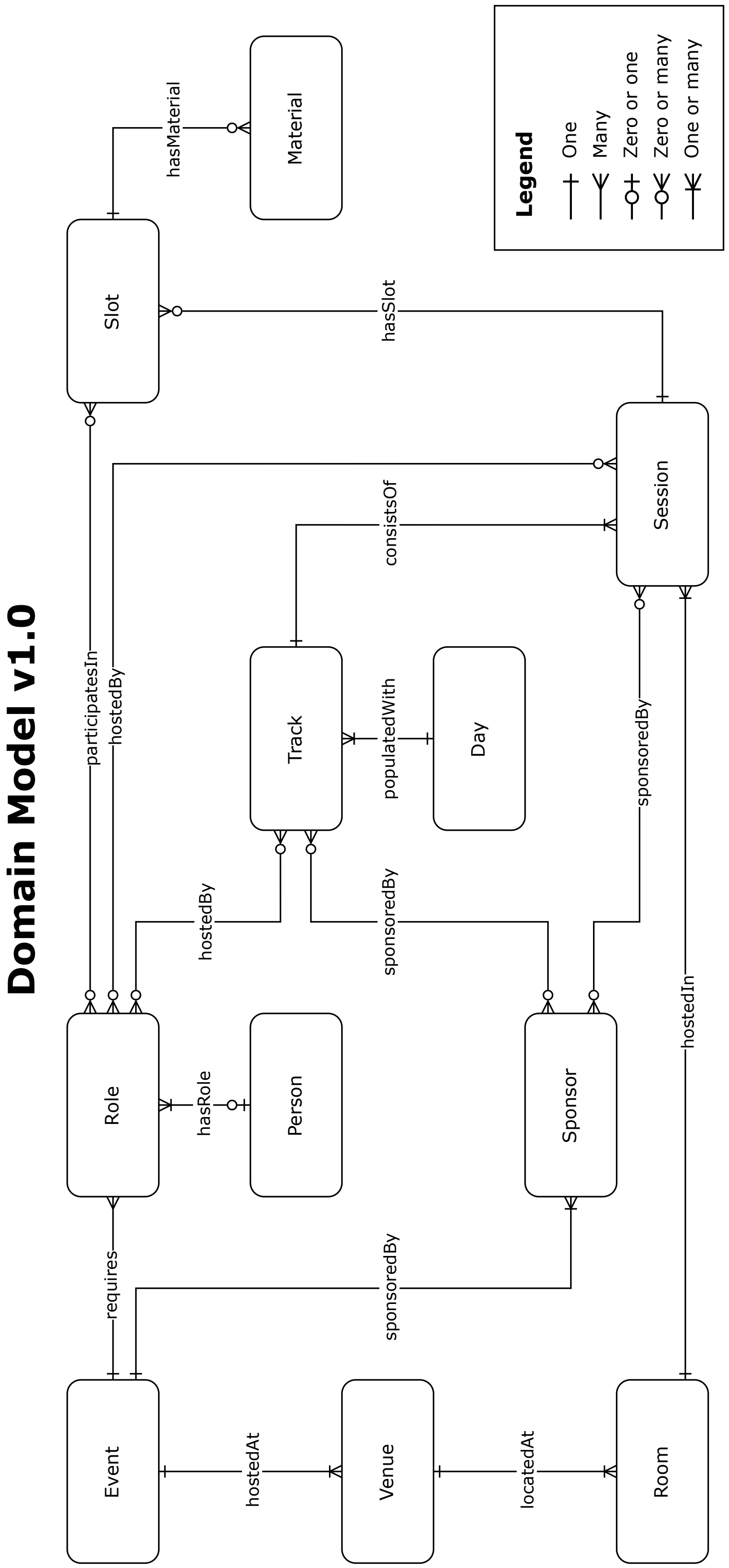


Figure A.1: The final version (v1.0) of a domain model for a small conference. [Diagram drawn by Yannik Rauter.]

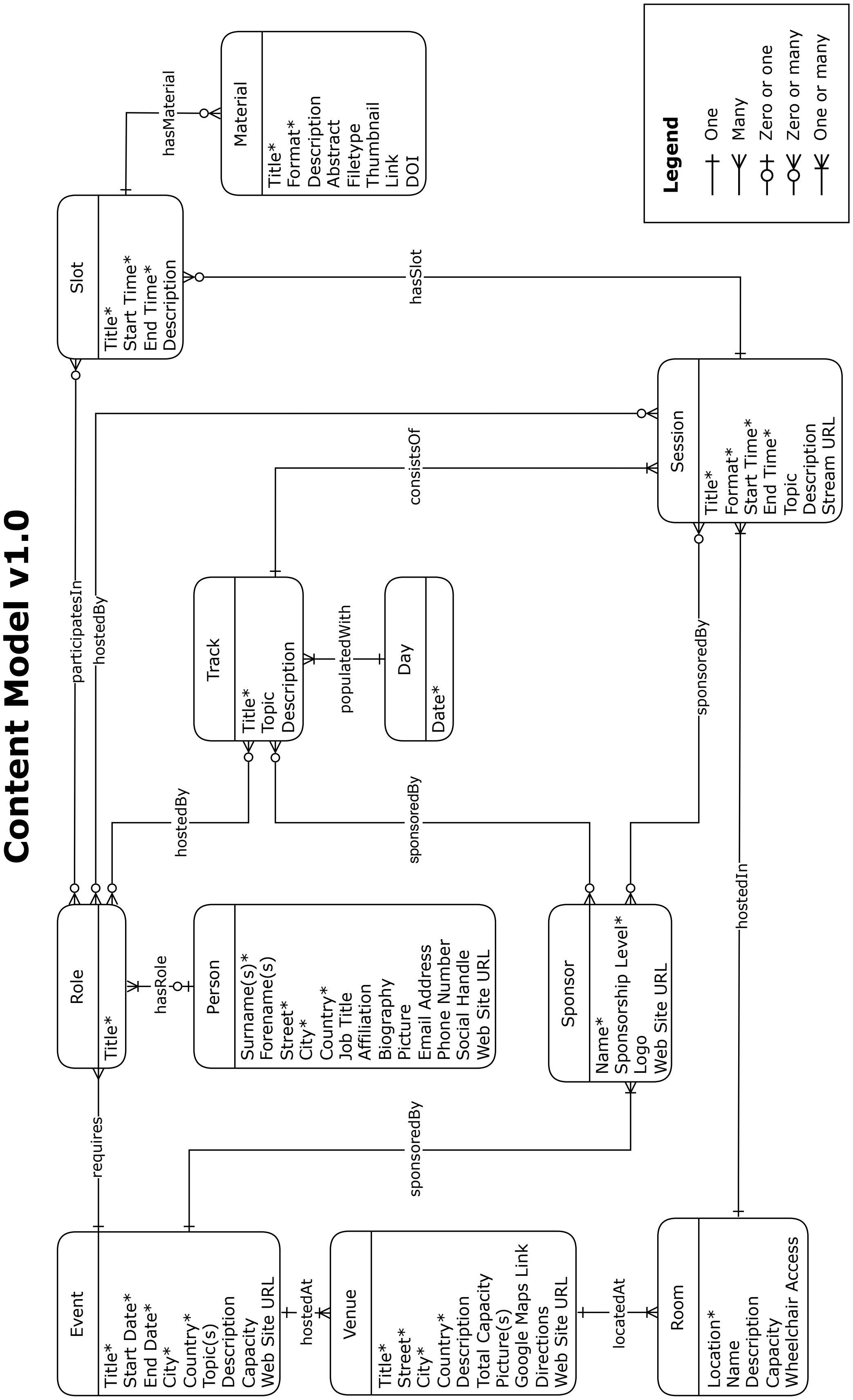


Figure A.2: The final version (v1.0) of a content model for a small conference. [Diagram drawn by Yannik Rauter.]

Bibliography

Andrews, Keith [2021]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria, 10 Nov 2021. https://ftp.isds.tugraz.at/pub/keith/thesis/ (cited on pages ix, 21).

Andrews, Keith [2024]. *Information Architecture and Web Usability: Course Notes*. 22 Oct 2024. https://courses.isds.tugraz.at/iaweb/iaweb.pdf (cited on pages 3, 7).

Atherton, Mike and Carrie Hane [2017]. *Designing Connected Content*. New Riders, 15 Dec 2017. ISBN 0134763386 (cited on pages 1, 3–5, 7, 9–10, 22, 35).

Barker, Deane [2012]. *What is a "Page-Based" CMS?* 27 Aug 2012. https://deanebarker.net/tech/blog/what-is-page-based-cms/ (cited on page 10).

Barker, Deane [2016]. *Web Content Management: Systems, Features, and Best Practices*. O'Reilly, 26 Apr 2016. ISBN 1491908122 (cited on pages 1, 9–10).

Bleuzen, Johan [2023]. *Jekyll turns 15 but for what Future?* 22 Dec 2023. https://johanbleuzen.fr/blog/jekyll-turns-fifteen-but-for-what-future (cited on page 35).

Boiko, Bob [2004]. *Content Management Bible*. 2nd Edition. Wiley, 26 Nov 2004. ISBN 0764573713 (cited on page 13).

Casey, Meghan [2023]. *The Content Strategy Toolkit: Methods, Guidelines, and Templates for Getting Content Right*. 2nd Edition. New Riders, 01 Jun 2023. 336 pages. ISBN 0138059276 (cited on page 1).

Cockpit [2025]. *Cockpit Headless Content Platform*. Agentejo, 09 Jan 2025. https://getcockpit.com/ (cited on page 10).

Contentful [2024a]. *API Basics | Contentful*. 12 Dec 2024. https://contentful.com/developers/docs/concepts/apis/ (cited on page 32).

Contentful [2024b]. *Basics | FAQ | Contentful*. 16 Dec 2024. https://contentful.com/faq/basics/#how-to-add-titles-to-entries (cited on page 27).

Contentful [2024c]. *Content that Takes you Everywhere | Contentful*. 05 Dec 2024. https://contentful.com/ (cited on pages 10, 25).

Contentful [2024d]. *contentful-migration - Content Model Migration Tool*. 06 Nov 2024. https://github.com/contentful/contentful-migration?tab=readme-ov-file#transformentriesconfig (cited on page 33).

Contentful [2024e]. *Multiple Environments | Contentful*. 13 Dec 2024. https://contentful.com/developers/docs/concepts/multiple-environments/ (cited on page 25).

Contentful [2024f]. *Spaces and Organizations | Contentful Help Center*. 13 Dec 2024. https://contentful.com/help/getting-started/spaces-and-organizations/ (cited on page 25).

Contentful [2024g]. *Usage Limits | Contentful Help Center*. 16 Dec 2024. https://contentful.com/help/admin/usage/usage-limit/ (cited on pages 33, 49).

Das, Abir [2025]. *How to Create a Custom Frontend with a Headless CMS*. PixelFree Studio, 10 Jan 2025. https://blog.pixelfreestudio.com/how-to-create-a-custom-frontend-with-a-headless-cms/ (cited on page 7).

Dhillon, Vikram [2016]. *Static Site Generators*. In: *Creating Blogs with Jekyll*. Edited by Vikram Dhillon. Apress, 11 Jun 2016. Chapter 3, pages 21–33. ISBN 148421465X. doi:10.1007/978-1-4842-1464-0_3 (cited on page 13).

Drupal [2024]. *Drupal*. 17 Dec 2024. https://drupal.org/ (cited on page 10).

DuCharme, Bob [2013]. *Learning SPARQL: Querying and Updating with SPARQL 1.1*. 2nd Edition. O'Reilly, 13 Aug 2013. ISBN 1449371434 (cited on page 3).

Go [2025]. *The Go Programming Language*. 20 Jan 2025. https://go.dev/ (cited on page 35).

Gulp [2024]. *Gulp*. 03 Dec 2024. https://gulpjs.com/ (cited on page 38).

Howey, Eric [2023]. *What is a Decoupled CMS?* Sanity, 19 Jun 2023. https://sanity.io/headless-cms/decoupled-cms (cited on page 9).

Hugo [2025a]. *Configure Hugo | Hugo*. 11 Jan 2025. https://gohugo.io/getting-started/configuration/ (cited on page 36).

Hugo [2025b]. *Directory Structure | Hugo*. 10 Jan 2025. https://gohugo.io/getting-started/directory-structure/ (cited on page 36).

Hugo [2025c]. *Front Matter | Hugo*. 11 Jan 2025. https://gohugo.io/content-management/front-matter/ (cited on page 38).

Hugo [2025d]. *Introduction to Templating | Hugo*. 12 Jan 2025. https://gohugo.io/templates/introduction/#whitespace (cited on page 47).

Hugo [2025e]. *Template Lookup Order | Hugo*. 11 Jan 2025. https://gohugo.io/templates/lookup-order/ (cited on page 38).

Hugo [2025f]. *Template Types | Hugo*. 10 Jan 2025. https://gohugo.io/templates/types/ (cited on page 38).

Hugo [2025g]. *The World's Fastest Framework for Building Websites | Hugo*. 09 Jan 2025. https://gohugo.io/ (cited on pages 13, 35).

IONOS [2023]. *What does WYSIWYG mean?* 01 Mar 2023. https://ionos.com/digitalguide/websites/website-creation/what-does-wysiwyg-mean/ (cited on page 9).

Jekyll [2024]. *Jekyll; Simple, Blog-Aware, Static Sites*. 14 Dec 2024. https://jekyllrb.com/ (cited on pages 13, 35, 49).

JvM [2025a]. *contentful-ssg - Contentful Export for Static Site Generators*. Jung von Matt, 10 Jan 2025. https://github.com/jungvonmatt/contentful-ssg (cited on page 37).

JvM [2025b]. *cssg-plugin-assets*. Jung von Matt, 10 Jan 2025. https://github.com/jungvonmatt/contentful-ssg/tree/main/packages/cssg-plugin-assets (cited on page 37).

JvM [2025c]. *Readme - contentful-ssg*. Jung von Matt, 11 Jan 2025. https://github.com/jungvonmatt/contentful-ssg/tree/main/packages/contentful-ssg#readme (cited on page 37).

Kissane, Erin [2011]. *The Elements of Content Strategy*. A Book Apart, 08 Mar 2011. ISBN 0984442553. https://elements-of-content-strategy.abookapart.com/ (cited on page 1).

Leatherman, Zach [2025]. *Eleventy - A Simpler Site Generator*. 09 Jan 2025. https://github.com/11ty/eleventy/ (cited on pages 13, 35).

Long, James [2025]. *Nunjucks - A Powerful Templating Engine with Inheritance, Asynchronous Control, and More*. 09 Jan 2025. https://github.com/mozilla/nunjucks (cited on page 13).

Luetke, Tobias [2025]. *Liquid Template Engine - Liquid Markup Language. Safe, Customer Facing Template Language for Flexible Web Apps*. 09 Jan 2025. https://github.com/Shopify/liquid (cited on pages 13, 35).

MDN [2024a]. *Subgrid - CSS: Cascading Style Sheets | MDN*. MDN Web Docs, 12 Nov 2024. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout/Subgrid (cited on page 44).

MDN [2024b]. *What is the Difference between Web Page, Website, Web Server, and Search Engine?* MDN Web Docs, 21 Nov 2024. https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/Pages_sites_servers_and_search_engines (cited on page 13).

Millett, Scott and Nick Tune [2015]. *Patterns, Principles, and Practices of Domain-Driven Design*. Wrox, 04 May 2015. ISBN 1491908122 (cited on pages 1, 3).

Naik, Umesha and D. Shivalingaiah [2009]. *Open Source Software for Content Management System*. Proc. 7th International Convention on Automation of Libraries in Education and Research (CALIBER 2009) (Ahmedabad, Gujarat, India). INFLIBNET Centre, 25 Feb 2009. https://web.archive.org/web/20091217142415/https://www.inflibnet.ac.in/caliber2009/CaliberPDF/29.pdf (cited on pages 1, 9).

Netlify [2024a]. *Headless CMS - Top Content Management Systems | Jamstack*. 12 Dec 2024. https://jamstack.org/headless-cms/ (cited on page 25).

Netlify [2024b]. *Static Site Generators - Top Open Source SSGs | Jamstack*. 17 Dec 2024. https://jamstack.org/generators/ (cited on page 35).

Oliver, Silver [2024]. *Using Domain-Driven Design and Conceptual Modelling to Support Knowledge Graph Development*. Data Language, 27 Jun 2024. https://datalanguage.com/blog/using-domain-driven-design-to-development-knowedge-graphs (cited on page 3).

Optimizely [2024]. *Episerver*. 17 Dec 2024. https://optimizely.com/episerver/ (cited on page 10).

Petersen, Hillar [2016]. *From Static and Dynamic Websites to Static Site Generators*. Bachelor's Thesis. University of Tartu, Estonia, 12 Aug 2016. 32 pages. https://core.ac.uk/download/pdf/83597655.pdf (cited on pages 1, 13–14).

Porcello, Eve and Alex Banks [2018]. *Learning Graphql: Declarative Data Fetching for Modern Web Apps*. O'Reilly, 02 Oct 2018. ISBN 1492030716 (cited on page 3).

Pug [2025]. *Pug - Robust, Elegant, Feature Rich Template Engine for Node.js*. 09 Jan 2025. https://github.com/pugjs/pug (cited on page 13).

Rauter, Yannik [2025]. *DCC Contentful Hugo*. 24 Jan 2025. https://github.com/yannikrauter/dcc-contentful-hugo (cited on pages 2, 51).

Rosenfeld, Louis, Peter Morville, and Jorge Arango [2015]. *Information Architecture: For the Web and Beyond*. 4th Edition. O'Reilly, 11 Oct 2015. 488 pages. ISBN 1491911689 (cited on page 1).

Sanity [2025]. *The Composable Content Cloud | Sanity.io*. 09 Jan 2025. https://sanity.io/ (cited on page 10).

Singh, Aniket, Anita Chaudhary, and Kirti Chaudhary [2023]. *Content Management System*. Global Journal of Enterprise Information System 15.1 (31 Mar 2023), pages 87–92. ISSN 0975-153X. https://gjeis.com/index.php/GJEIS/article/view/713/653 (cited on pages 9–10, 25).

Sosso, Joshua [2025]. *contentful-hugo*. 10 Jan 2025. https://github.com/modiimedia/contentful-hugo (cited on page 37).

Spencer, Donna [2014]. *A Practical Guide to Information Architecture*. 2nd Edition. ebook. UX Mastery, 2014. ISBN 0992538025. https://uxmastery.com/practical-ia/ (cited on page 1).

Storyblok [2025]. *Storyblok - Headless CMS with Visual Editor*. 09 Jan 2025. https://storyblok.com (cited on page 10).

Strapi [2024]. *Strapi - Open Source Node.js Headless CMS*. 05 Dec 2024. https://strapi.io/ (cited on pages 10, 25, 49).

Van Lierde, Kevin [2025]. *Metalsmith - An Extremely Simple, Pluggable Static Site Generator for NodeJS*. 09 Jan 2025. https://metalsmith.io/ (cited on pages 13, 35, 49).

Vepsäläinen, Juho and Petri Vuorimaa [2022]. *Bridging Static Site Generation with the Dynamic Web*. Proc. 22nd International Conference on Web Engineering (ICWE 2022) (Bari, Apulia, Italy). Springer, 01 Jul 2022, pages 437–442. ISBN 3031099168. doi:10.1007/978-3-031-09917-5_32 (cited on page 15).

Yankulov, Milen [2024]. *What Are Linked Data and Linked Open Data?* Ontotext, 15 Dec 2024. https://ontotext.com/knowledgehub/fundamentals/linked-data-linked-open-data/ (cited on page 3).

Yermolenko, Andrei and Yuriy Golchevskiy [2021]. *Developing Web Content Management Systems - from the Past to the Future*. Proc. 1st International Conference on Economics, Management and Technologies (ICEMT 2021) (Yalta, Crimea, Ukraine). SHS Web of Conferences, 11 Jun 2021. doi:10.1051/shsconf/202111005007 (cited on page 9).