

University of Edinburgh

School of Informatics

A Storage Framework to Promote Reuse of Modular Content

4th Year Project Report Software Engineering

Niall Napier

June 5, 2008

Abstract: This project proposes a new, flexible, way of structuring the content held in a content repository by making the relationships between individual entries the basis for organisation. These relationships can be used to simplify the creation of recommender systems and provide clear ways to combine content across the framework. A new data model was designed and implemented along with a Content Management System (CMS) to showcase the solution. The implementation of the model is not production ready; however, it illustrates the flexibility of the model and how it can be used to promote the reuse of modular content.

Acknowledgements

I would like to thank Stratis Viglas, project supervisor, for his help and guidance over the course of this project. I would also like to thank Peter Buneman for his feedback during group meetings.

Contents

1	Introduction	1
2	Background	3
2.1	Data Models	3
2.1.1	Relational Model	3
2.1.2	Domain-Specific Models	4
2.2	Object-Relation Mapping	4
2.3	Metadata	5
2.4	Scripting in Java	5
2.5	Related Work	6
3	Specification	7
3.1	Framework	7
3.1.1	Objectives	7
3.1.2	Data Model	8
3.1.3	Type System	11
3.1.4	Associations	12
3.2	Content Management System	13
4	Implementation	15
4.1	Framework	15
4.1.1	Persistence Layer	15
4.1.2	Code-Bits	17
4.1.3	Type System	18
4.1.4	Associations	19
4.2	Content Management System	20
5	Evaluation	23
5.1	Specification Objectives	23
5.2	Engineering Criteria	24
5.2.1	Fitness	24
5.2.2	Dependability	25
5.2.3	Maintainability	25
5.2.4	Scalability	25
5.2.5	Security	28
5.2.6	Cost	28
6	Further Work	29

7 Conclusion	31
Bibliography	33
Appendices	37
A Code Examples	37
A.1 Code Bit: Search By Tags	37
A.2 Code Bit: Render Test	38
B Test Results	41
B.1 Render Time Results	41
C Screenshots	43
C.1 Content Management System	43

1. Introduction

Content Management Systems (CMSs) are used globally to provide users with a non-technical way of maintaining website content. Their use is varied and can be seen everywhere from large corporations to personal websites. Using tools in the CMS users can include images, links and other resources, enhancing the reader's experience by recommending items they may be interested in. In e-commerce applications, recommendations are common-place and suggest products, based on what the user has viewed [11] and/or the purchase history of other users. Recommendations have been replicated in other environments to provide suggestions and optimised ways of navigating content through attributes like the popularity of the page or the navigation of other users [3]. While accurately targeting recommendations is a challenging task, it becomes even more difficult when there is a lack of historical data, such as in the case of new or unpopular items. Several research projects have investigated how existing structural information such as categories can be used to improve recommendations [7].

The majority of websites partition their content in a book-like structure, though rather than chapters and sections, websites provide a hierarchy of pages to represent these concepts. The popularisation of the wiki format saw many documentation projects shift to an index based approach to content organisation. In this format, articles are single items of content written about some subject, and are added ad-hoc to a structureless pool of content. For multimedia repositories content is typically partitioned into types.

While these structures may be suitable in environments where the type of content rarely changes, there is little flexibility in their organisation and content reuse is rigid. This report proposes a method for organising items of content based on the relationships between them. By defining these relationships explicitly, there are clear ways to show users content related to what they are viewing. These explicit relationships can then be augmented by dynamically calculated, implicit, relationships providing a potential means to rank their relevance. Items of content are stored independently of one another in the lowest logical form, such as the text of an article, or the image file of a picture. These modules can then be reconstructed into articles or other more complex content when displayed to users.

In this report, a new data model is presented that holds content associations at the storage level. This allows associations to occur between any types of content and for those associations to define the organisation of content in the model. To allow generic content types without repeated changes to the data store, a dynamic type system was devised which allows content to programmatically influence how it or

other content is displayed. The framework was implemented in Java and a basic CMS was also developed to show that the framework could be used successfully in an application.

Unlike common approaches to content storage the framework does not store content in a predefined structure and allows items to be included at any number of points in the framework. While this leaves elements in the framework with no sense of location, it makes it possible for users to define long complex associations between items. The dynamic type system allows users to customise how content is displayed and therefore makes it possible for the system to be used in domain specific content applications.

The remainder of this report explains the key concepts used in the implementation of the framework, how the framework is structured, how it and the CMS were implemented and how this implementation compared with the original objectives of the project. Finally, the report concludes with some suggestions for work that could be done in the future to improve the framework.

2. Background

This chapter provides an introduction to the concepts and technologies used in implementing this project; namely, the construction of data models, how to access data from an object oriented framework and how to achieve scripting inside a Java application.

2.1 Data Models

Data models are used to represent a logical structure to the information contained in some domain. According to the American National Standards Institute (ANSI) framework, data models can be considered at three levels: Conceptual, Logical and Physical. The conceptual level considers entities and their relationships at a high level. For example, a Vendor *sells* Hotdogs *to* a Customer. At this level we are interested in the main objects, their attributes, and their relationships. The logical level considers how the conceptual model can be implemented in terms of some underlying framework, such as a relational database, and how the limitations of the framework may limit what is possible in the conceptual model. This logical model is typically documented in what is known as a schema. The physical level is handled by a database or other persistence mechanism and describes the way information is physically stored in the system.

2.1.1 Relational Model

The majority of general-purpose databases use a relational model to store information. In this model, data is held in tables consisting of one or more columns. A relationship is formed between two tables when a column in one table is used to store values associated with another table. For example, if we have a table called *Cats* and another table called *People*, we can have a column in the *Cats* table called *owner_name* and a column in *People* table called *name*. This common attribute could then be used to combine the two tables and show all the cats owned by each person. It is also possible to search tables based on the values held in columns.

Relational databases have been prevalent since the inception of System R in the 1970s, yet they have recently been criticised for failing to adapt to new hardware capabilities [12]. While relational databases are still relevant, it becomes increasingly important to consider if the conceptual model of a system is best served by this technology.

2.1.2 Domain-Specific Models

Domain-specific models work under the assumption that the way you store data should be entirely defined by its purpose. Historically, relational databases have stored their data in a format optimised for efficiency under disk I/O operations – this increases the efficiency of joins and queries. Extensible Markup Language (XML) stores data in a plain text, human-readable format which allows any application to parse the data stored within – this makes it suitable for exchanging information between applications.

By adopting a customised data model an application can overcome the constraints associated with adopting, say, the relational model. However, the cost of developing a customised storage engine is likely to outweigh that convenience.

While this project uses a relational database to store its underlying data there is a strong focus on the design of the conceptual model and it would also be possible to implement the model using some other underlying representation.

2.2 Object-Relation Mapping

Relational databases are supported in Java by using the Java Structured Query Language (SQL) Application Programming Interface (API). Typically this is implemented by a Java library from the database vendor and allows developers to manipulate the database without having to communicate with it on a low-level. While this provides a high level of control over the database it does not map well to typical object-oriented programming. Queries are specified using SQL and results are returned as `ResultSet` collections which may require some manipulation before they can be handled like Plain Old Java Objects (POJOs).

Object relational mapping tools allow the developer to relinquish this responsibility and treat objects mapped in the database as POJOs. Hibernate, a popular Object-Relation Mapping (ORM) tool, facilitates this by allowing the developer to generate mapping files. These are XML representations of how objects in an application relate, and are used to validate the objects against the database and save any changes. Where there is a 1-1 correspondence between two objects, object A simply stores object B as a field in its class. Where 1-n correspondences occur, these are modelled with Java `Set` collections on the parent side of the relation, and single objects on the child side.

Querying the database is still possible, but rather than using the Java SQL API the Hibernate query framework is used. This allows users to query the database in terms of the objects they have defined in their application. When possible,

queries return results as a collection of application objects, rather than an abstract collection of fields.

While ORM allows the developer to code in a more natural, object-oriented style, it can limit the transparency and sometimes the efficiency of an application. As communication with the database is handled entirely by the ORM system, it can be difficult to debug problems with a particular query or transaction. Additionally, where the objective and relational models do not align, developers must consider the performance of their approach.

2.3 Metadata

Metadata is a set of attributes used to describe a piece of data and is frequently used to describe music (artist, album, track name), files (owner, date-created, file type) and other media. The metadata that is stored for an item depends entirely on the implementation and setting it occurs in. The Dublin Core standard [10] provides a common method for describing data and consists of fifteen metadata elements, all of which are optional and repeatable. These metadata elements can then be used to partition, search and order data into useful sets.

With the advent of Web 2.0 ‘tagging’ has become a popular way to describe content by associating it with keywords or phrases. This is especially popular for audio-visual content which, unlike text-based content, is not self-describing. Tags are often stored independently of each other as a set and this allows them to be easily cross referenced against items with matching tags. Tagging provides a method to define loose relationships between content based on semantic description, but more complex relationships can also be defined or inferred from this data. This type of data description allows the system to model an extensible set of metadata where an item of content may have zero or more of the available attributes.

The “Vocabulary Problem” [5] raised by Furnas et al states that since a single user will only ever tag a piece of content with words that they know, they may not cover all or even the most appropriate keywords for some item of content. By allowing multiple users to associate tags with some piece of content it is possible to combat this issue.

2.4 Scripting in Java

The Java 6.0 Scripting Framework Specification [6], part of the Java 6.0 Runtime Environment, provides a standardised way, regardless of scripting language, for

developers to execute scripts from within an application. While it was not used in this implementation, the scripting method employed uses a very similar paradigm and was chosen, in part, for its forward compatibility.

This project makes use of Groovy [4], an established scripting language closely related to Java and well supported in the open source community. It allows developers to embed or load scripts and have them execute from within a Java application. For this to be useful, scripts may return a value to the calling method, and variables may be bound to an instance prior to execution. These bindings allow the code inside a script to access objects outside of its regular scope. From the script's point of view, bound variables can be used as if they were previously declared in the script and can be of any type.

2.5 Related Work

A wide variety of commercial and open-source CMSs are available on the market. While they are typically coupled as a combined CMS and repository, there are some exceptions to this. In the last few years Java has seen an effort to harmonise content repositories so that they can be accessed by a common API. This has resulted in a new standard, The Content Repository for Java Technology [9]. This specification defines content as a series of nodes arranged into a tree hierarchy. This can be compared with a UNIX file system where the root node is / and child nodes are sub-directories of that node /child/child2. The content itself is expressed as a property of some particular node. While this is not especially flexible, it provides a well-formed means of storing and retrieving content that is familiar to most web developers. The type of content that can be stored in this model is defined entirely by the implementation.

Drupal, a popular open-source CMS, uses a modular architecture such that new content types can be added at run time [2]. However, to support these, changes are made to the underlying storage mechanism, such as adding new tables or columns to the repository database. By storing new types in type-specific tables it can become easier to optimise and maintain that data, however, these modules still require the user to implement handlers, design tables to store that data in and package all the required parts before it can be deployed on the server. Thanks to the large community that has evolved around Drupal, there are many existing modules available and administrators can download and install pre-made modules.

3. Specification

This chapter defines the goals of the project and outlines the design of the data model, the framework and the CMS.

3.1 Framework

3.1.1 Objectives

In the introduction, this report claims that by using the framework proposed in this project, developers can promote modularity and reuse of content in their applications. In order to achieve that aim, the framework must be able to satisfy a set of objectives.

To demonstrate modularity and flexibility the framework must be able to:

- Add new content types and content handlers without requiring changes to the framework.
- Allow a highly flexible organisation of content.

To demonstrate that the framework promotes reuse of content it must be able to:

- Allow content to appear at various points in the framework.
- Display content *explicitly* and *implicitly* associated with that currently being viewed.

These objectives, along with a set of engineering criteria, will be used to assess the potential of the framework and its implementation.

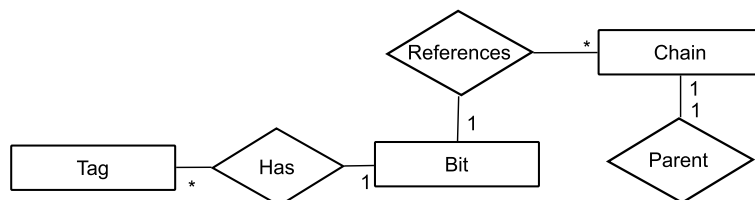


Figure 3.1: Entity relationship diagram of the framework data model.

3.1.2 Data Model

In the field of content storage, two main requirements were identified. Firstly, for each item the framework must store a *body of content*. This is defined as a space to either declare the main item of content or a reference to that item. For example, an article may store its text as the body of content whereas an image may instead store a URL which can be used to locate the required image file. Secondly, the framework must provide a means of storing metadata associated with each item. This is required to locate, partition and sort the content in the repository.

To encourage modularity and content reuse some additional requirements were added. Firstly, any item of content must be able to create an association with any other item of content regardless of its type or any other attribute. Secondly, the system should be able to handle new types of content without requiring changes to the underlying data store or compiling changes to the source code. Lastly, it should be possible to define non-fixed, optional attributes for items of content.

Table 3.1: Common class fields.

Attribute	Type
ID	UUID String
Owner	User
Date Created	Date Time
Version	Integer

By simplifying the data model in this manner it is possible to identify what aspects of content management are essential to the framework. Using these requirements, the data model shown in figure 3.1 was devised. At the storage layer there are three basic classes: *Bits*, which store the content or data of the framework; *Tags*, which store optional attributes of content; and *Chains* which define associations between content.

All of these components share some basic fixed attributes as shown in table 3.1. These allow the framework to process foreign key joins, basic ownership and temporal ordering regardless of the content that has been defined. The Version attribute is used to ensure consistency during transactions. A *User* class was also defined so that the system could be extended to support multiple users and maintain ownership of class objects.

Table 3.2: Bit class fields

Attribute	Type
Content	String
Date Modified	Date Time

Bits

Bits are used to store items of content and form the basis of the framework. The ‘Content’ field reflects the body of content and contains either content itself or a reference to some external resource such as a file or website. Since bits store all content for the repository, there is no distinction between the types of content that can be stored in the content field. At the storage level the only requirement is that the content field be text-based. The distinction of types is handled by the type system at the application level.

Since content can be anything that can be represented using text, it is possible to use bits to define parts of the application – essentially creating a plugin architecture for the system. Bits used in this way are referred to as ‘code-bits’ and are essentially scripted components that work on a set of input parameters and produce an output. In the case of this framework, the output is always a string that can be used to display the required information. Code-bits form the basis of the pluggable types system and are discussed in section 3.1.3.

It is worth noting that while the framework allows for a diverse range of content types, the CMS using the framework may impose restrictions on what a user can view or add to the framework; especially if the framework provides permission-based control over content.

The fields of the bit class are given in table 3.2.

Tags

Table 3.3: Tag class fields

Attribute	Type
Content	String(128)
Bit	Bit

Tags are keywords or phrases which specify some attribute of a bit. They allow the framework to support optional attributes of bits and can be partitioned into two types.

Semantic tags allow users to associate keywords or phrases with a bit to describe its content. This is a concept familiar to internet users and is popular for describing content which is not text-based, for instance images or videos. It can also provide a means of loosely categorising content whereby items can be assigned tags on an ad-hoc basis and those tags can be used as a grouping. An example of a semantic tag would be to give a video of the sun going down the tag “sunset”. By allowing any user to add to the tags associated with a particular item of content the framework can provide a solution to the Vocabulary Problem discussed earlier.

System tags are used to define non-fixed system attributes such as content types, handlers and other implementation-specific items. System tags must begin with the framework name-space, this allows them to be differentiated from semantic tags. System tags should not be explicitly created by users but instead created by the CMS where appropriate. For example, if a user wants to create a new piece of content, rather than have them explicitly add a tag to that content specifying its type, the CMS would allow them to select from a list of available types and add the tag automatically. The type system is described in section 3.1.3.

The fields of the tag class are given in table 3.3.

Chains

Table 3.4: Chain class fields

Attribute	Type
Parent	Chain
Bit	Bit

Chains define the associations between bits and allow the framework to express content relationships while also loosely defining the structure of data. Chains are organised into a tree hierarchy through parent-child relationships and are each associated with a single bit. This allows any bit to be referenced at multiple locations in the chain and therefore at multiple locations in the content structure. This structuring can be used to retrieve context-sensitive and in-sensitive associations as well as infer associations between content.

While it is possible to create bits without assigning them to a chain, the implementation of the framework may not display them when the user browses the content repository as they have no context. Chains must always reference a bit in the framework. If the bit a chain references is removed then the chain and all its children should be removed as there is no longer a full context. Similarly, if a chain is removed, then all of its children should also be removed though the

bit should not be deleted. The associations modelled by chains are detailed in section 3.1.4.

The fields of the chain class are given in table 3.4.

Users

Table 3.5: User class fields

Attribute	Type
Handle	String
Email	String
Password	String
Date Modified	Date Time

The user class is a special case which allows the system to store information about users and maintain information about ownership of Bits, Chains and Tags. The user entity inherits the same properties from the generic set with the exception of owner. The fields of the user class are given in table 3.5.

3.1.3 Type System

To allow any item of content to be associated with any other item requires thought at the logical level of data modelling. Conceptually, there is a master class ‘content’ and from that subclasses can be defined as required to implement each individual type. Using an object-oriented approach we can even subdivide the type tree into groupings such as binary and text-based content, video, articles etc. This, however, is difficult to maintain on a logical level, particularly when one of the earlier objectives states that the framework “...*must be able to: Add new content types and content handlers without requiring changes to the framework.*”

This is solved by using dynamic typing in the data model. Dynamic typing is achieved in the storage layer by moving the type information of content from its data structure into an attribute. The content is then stored as a single body complimented by more, optional, fields to store related metadata. While this allows more flexibility in the type system it is important to note that if the underlying data schema doesn’t constrain types then the framework must provide a method for doing so.

In the framework, content type attributes are optionally declared using system tags. If the user wanted to declare the type of some bit as an image, a tag would be added to that bit with the content:

```
tmb.content.type=image
```

In the case where ‘image’ is not a supported content type, the default content type is used. Support for different types of content can be added by adding a code-bit as a content handler. To declare a bit as a content handler, in this case for the type *image*, a tag is added to that bit:

```
tmb.content.handler=image
```

When an item of content is to be displayed to the user, the system must check to see if it has an associated type. If it does, the system loads the content handler for that type and uses it to render the content in an appropriate manner. If the handler encounters an error, it degrades gracefully and returns an error message rather than throwing an exception. Since the handler may be called as one in a sequence of recursive renders this is particularly important. Returning a string error message ensures that the user understands there has been an error for some part of the content but the rest of the content still displays as expected. This process is detailed in section 4.1.3.

3.1.4 Associations

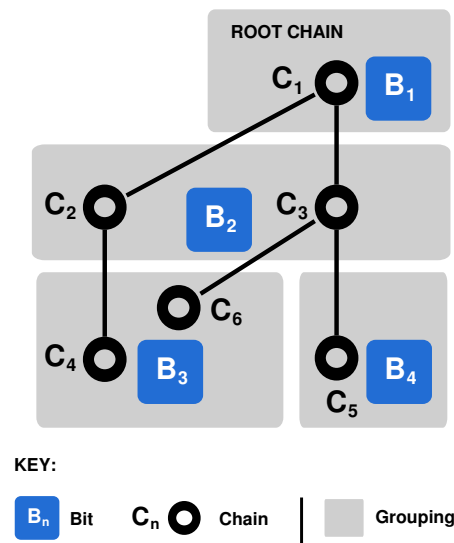


Figure 3.2: A possible instance of the framework, focusing on associations. (*Note that the grouping in this diagram shows how chains reference bits and is not a part of the data model.*)

Chains implement the unique structure of the framework by allowing associations between any two items of content, regardless of their type. Conceptually, chains are in fact links between bits in the framework which collectively form a chain of

content. Each chain contains a reference to a parent chain and some bit. This self-referencing parent-child relationship allows associations to be free-standing of the content they represent. This also presents some constraints on how chains can be defined. All chains, with the exception of the root chain, must reference another chain as their parent; this ensures that the graph of associations never becomes partitioned. Since data typically shares a grouping at some level, this does not pose a problem for disparate content. Should the user wish to have no grouping, it is still possible to create a root node to represent ‘data’ and chain all content from there. By using a tree-like structure rather than a mesh to define associations, the model allows the user to retrieve associated content on a contextual or context free basis.

Any chain or association defined between two bits is contextual. Figure 3.2 illustrates direct associations between C_2 and C_4 and again between C_3 and C_5 , and C_3 and C_6 . Note that in the context of C_3 , B_2 and B_4 are related, yet in the context of C_2 they are not.

By storing associations in this fashion it is possible to achieve higher granularity when assessing relationships between items of content. Through simple queries it is still possible to retrieve all related content regardless of context and, with more complex queries the context of associations can be used to rank the relevance of related content.

To illustrate this, consider the case where B_2 is a picture of a frog, B_3 a toad and B_4 a fish. In this instance it is possible to imagine a context where these bits are and are not related. At C_2 a user may only be interested in images of amphibians, hence they see the pictures of the frog and the related picture of a toad. At C_3 the user may be interested in images of pond-life or animals hence they see the frog, toad and fish images.

Since associations are informal, they do not represent defined categories, yet being contextual they can be used to form content structures. With some extension to the implementation model they could also be used to store learned associations or strengths of association. This model can be loosely compared to that of a neural network. However, there is no capacity for the model to respond to multiple inputs, nor for it to perform any analysis about the quality or relevance of its knowledge.

3.2 Content Management System

To demonstrate that the framework can be used in applications, a basic CMS was developed. The system allows the user to create and edit bits, create and remove chains and add and remove Tags. This demonstrates that the framework

can be used for basic data storage. To show that the framework maintains a mapping of associated content, pages always provide links to content related to that currently being displayed. This gives the user a means by which to navigate the site and discover new items of interest. The framework also suggests items of content derived from implicit associations which can be viewed through a link in the CMS.

To demonstrate that the system can dynamically define types, the CMS provides a method of declaring them by implementing code-bits and afterwards, adding a system tag to identify the bit as a type handler. The system also provides a search feature so that users can find bits based on their associated tags.

Since the structure of content is derived from its relationships, the CMS includes a set of initial data so that the system has some organisation and some existing content handlers in order to display basic types of content.

4. Implementation

This chapter discusses the implementation specific parts of the framework and CMS including the various issues faced during development.

4.1 Framework

The framework was implemented using Java 5.0. Java was chosen as it was a familiar language with a full set of libraries that could help to deliver the data persistence layer. While Java 6.0 has been available for some time, it was not used due to lack of support on some platforms.

4.1.1 Persistence Layer

In this implementation all entities are stored in a relational database. This allows the framework to perform efficient queries in an environment familiar to many developers. To manage the storage and retrieval of objects from the database the framework uses Hibernate, a Java ORM library.

In this implementation, the database schema is defined by the Hibernate mappings which allow the framework to potentially use any database supported by Hibernate. These are defined based on the properties of the main data classes and allow developers to treat the classes like POJOs.

Access to the framework data is managed by a set of persistence classes, maintained within the `com.tagmybits.dao` package. Each object class - bit, tag and chain - has a corresponding Data Access Object (DAO) which allows the developer to retrieve, save and remove objects as well as perform queries specific to that class. In combination with the `HibernateUtil` class the DAO classes also manage Hibernate sessions which allow the application to connect to the database. These DAOs are invoked using a factory class¹ which simplifies the process of accessing the data store. To illustrate this, an example call to the framework is given in listing 4.1.

```
1 // Instantiate the DAO factory
2 DAOFactory daoFactory =
```

¹The factory and `HibernateUtil` classes used in this implementation were adapted from the Hibernate documentation [1] and were originally written by Christian Bauer

```

3     DAOFactory.instance (DAOFactory.HIBERNATE);
4     // Get the bit DAO
5     BitDAO bitDAO = daoFactory.getBitDAO ();
6     // Start a new transaction
7     HibernateUtil.beginTransaction ();
8     try {
9         // Find the required bit
10        bit = bitDAO.findById (bitId , false );
11
12    } catch (NoSuchItemException e){
13        throw new InterfaceException ("Could not "
14            + "update the required bit");
15    }
16
17    // Make changes
18    bit.setContent (content );
19    // Commit the transaction
20    HibernateUtil.commitTransaction ();

```

Listing 4.1: Calling the framework through the DAO.

Because the framework is manipulated through the DAO mechanism rather than directly, the ORM implementation can be changed without affecting other areas of the system.

Entity DAOs provide a generic query method which allows the user to query the framework based on the field values of an object supplied as a parameter. Bit DAOs also provide simple query methods which allow the user to query the framework based on some common parameters such as a keyword or tag. Since the project does not implement a query language these methods are necessary to provide users with access to the framework's data.

To simplify and secure data access from code-bits, another DAO was implemented which includes a subset of functions from each of the main DAO classes. This DAO can retrieve and query entities but cannot save, remove or update them. This ensures that code-bits cannot change the state of the framework but that they can still perform more complex queries.

Transactions are supported through the Hibernate framework and allow the user to develop applications which may depend on more than one database action to complete some task. The transaction manager is invoked through the `HibernateUtil` class but does not depend directly on the DAO. In this implementation Hibernate sessions are managed on a per-thread basis using the Thread Local Session pattern. This is a common approach in Hibernate application design and allows the user to start, commit and rollback transac-

tions without having a handle to the current session object. This can be seen in listing 4.1 where a transaction is started and committed using the static `HibernateUtil.beginTransaction()` and `HibernateUtil.commitTransaction()` methods. Until a transaction is committed, no changes are persisted and if an error occurs when committing the transaction, the transaction is automatically rolled back and an exception thrown.

4.1.2 Code-Bits

As detailed in the specification, the framework is able to display various types of content by using code-bits to compute the view. In this implementation, code-bits are implemented using the Groovy scripting language and are able to access other data items through a series of variable bindings. These allow the code-bit to access external objects from within its own scope. The objects bound to the code-bit execution include the chain object for the current chain, a set of parameters and the `codeDAO`. The additional binding `chainContent` provides a shortcut to retrieving the content of the current chain being rendered while the boolean `selfRendering` tells the code-bit if it is being called to render itself or another bit. An example using all of these bindings is given in appendix listing A.1.

By providing access to the chain object, code-bits can retrieve and display content from further down the context chain. This can be useful in situations where content is better displayed as a set of items rather than individually. For example, if we have a chain that represents pictures taken on holiday. While it makes sense to store the pictures as individual items, they are also part of the collection ‘holiday snaps’. If we define a bit to encompass this information, we can then code a bit which displays the title of the album and then uses the chain object to find the child chains, the individual pictures, and display their thumbnails.

To display a child chain requires that the code-bit has access to the chain DAO as this is where the render method is located. As mentioned earlier, this is inadvisable as it would allow the code-bit to change the state of the framework. Instead, the the system binds the code-bit specific `codeDAO` which provides the code-bit access to the render method in a sand-boxed environment.

Due to the tree structure of chains it is not possible to create a looped chain, however, it would be possible to create an unnecessarily large chain. In the situation where each chain depends on the chain below it, this results in a large number of calls to the render method in order to display the topmost element. Since it is unlikely that a user is interested in content many links below the current item in the chain, a depth limit was introduced. This ensures that any call to the render method only recurses to a limited depth in the chain before

displaying the result to the user. All new calls to the render method inherit the depth of the current render to insure that the depth restriction is enforced for all recursions.

Parameters extend the model by allowing users, or system properties to influence how content is displayed. In the CMS this allows features which depend on user input to be entirely handled by a code-bit. For example, parameters from the querystring such as a search query could allow a code-bit to display search results. Parameters are stored as key-value pairs in a HashMap object and passed to the chain being rendered.

4.1.3 Type System

Content is always rendered based on its context in the chain and is displayed by calling the method `chainDAO.render(ChainId)`. Since displaying (rendering) a piece of content requires that the system resolve its type, the render method is implemented in the DAO which allows it to perform queries on the underlying data store.

Code-bit parameters, passed to the type handler can be used to specify how an item should be displayed. For example, by default a type handler may display an image full-size. However, if the parameter `size=small` is supplied, the type handler will instead display the thumbnail version of the image.

When a code-bit is used as a content handler it is rendered in-place as if it were the original item of content. Essentially the handler inherits all the information that is associated with the chain it is rendering. The type handler has access to this through the chain binding.

In figure 4.1 the picture example mentioned earlier is shown. B_2 and B_3 contain the filenames of two images and have an image type with handler B_5 . B_1 holds the title of the album and has an album type with handler B_4 . The handler specified in B_4 loads any pictures chained to the current chain (C_1) and calls the render method on them. In this example the image type handler converts the filename to an HTML image tag and returns this. These are then combined with the title of the album to create a total render for the gallery.

Since types are declared as optional attributes of content it is possible for content to not have an associated type. If this happens, the default type handler is used to display content. In this implementation this is defined as the `article` type.

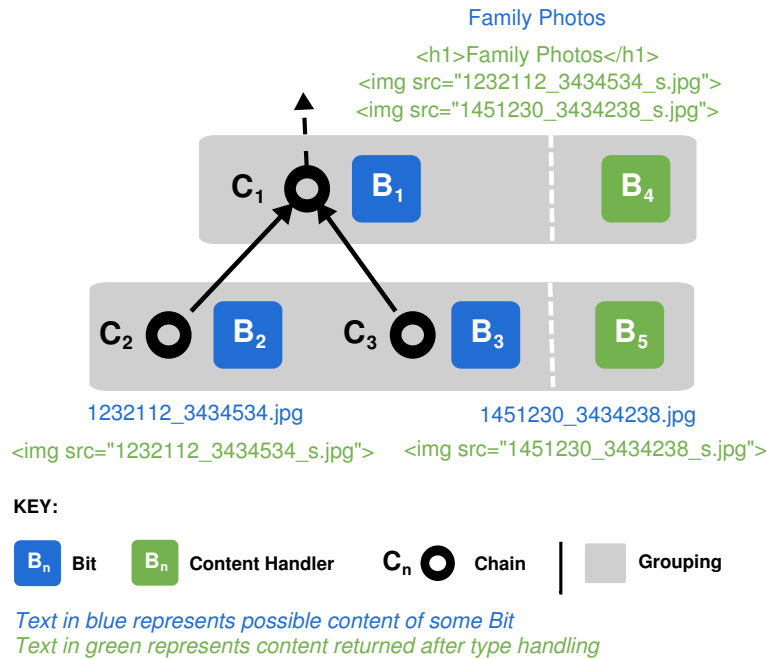


Figure 4.1: A possible instance of the framework, focusing on content rendering. (Note that the grouping in this diagram represents how bit types are associated with content handlers.)

4.1.4 Associations

Associations, defined as chains in the framework, are implemented in the database using a self-referencing relation. This is mapped by the ORM system as a tree of objects where each chain object contains a set of child chains. While the chain objects provide a method to return any direct child of that chain, methods for more complex operations are handled by the `ChainDAO` which has access to perform queries on the underlying database and is therefore more efficient.

The framework supports three methods of retrieving children for a given chain. As detailed in section 3.1.4, the way the framework stores associations results in context-sensitive relationships, and typically children are returned using the `getChildren([Chain])`² method. While this provides a high degree of granularity to the system, the framework also provides methods to present a context-free environment. This was achieved by adding another method, `getAllChildren(Chain)`, in the `ChainDAO` which, given some chain, returns the children of all chains which reference the same associated bit. Looking back to the example shown in figure 3.2, calling `getAllChildren(C2)` will return `C4`, `C5` and `C6`.

²The parameter 'Chain' is only required when the method is called from the `ChainDAO`. The method can also be called on a chain object itself (e.g. `chain.getChildren()`).

The third and final method for retrieving chain children is the `getImplicitChildren(Chain)` method. Implicit children are defined as chains which may not carry an explicit association, nor an associated bit but must share some common attribute. In this implementation implicit children are defined as chains sharing some semantic tag with the original chain³. This should return items which have been tagged with the same keyword regardless of whether or not they are explicitly linked.

4.2 Content Management System

A web CMS was developed to show that the framework could be effectively used in an application. While this was a success it did uncover some weaknesses in the implementation that would need to be addressed in further development.

The system was written as a Java Enterprise application using a combination of Java Server Pages (JSPs) and Servlet technology. JSPs were used to handle presentation and exceptions while a servlet filter was used to establish the Hibernate session for page requests. This model was chosen as it provided a fast and straight-forward method to implement the main functionality of the CMS.

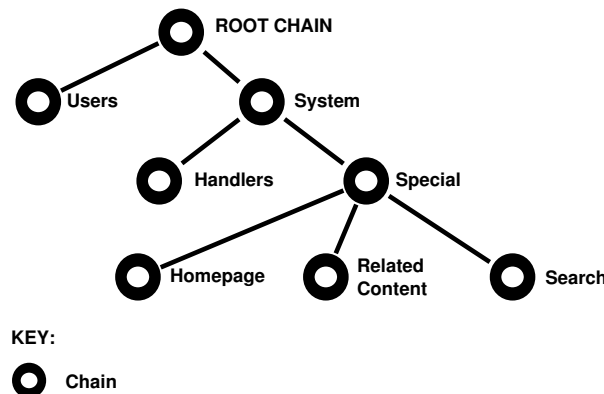


Figure 4.2: The content structure of the implemented CMS.

The CMS structure was based on that shown in figure 4.2. Since the framework allows the CMS to define the structure of content, this is entirely at the discretion of the implementation. This structure was chosen so that there was a clear distinction between bits that are fundamental to the system and bits that are purely content. Each user has a chain under the ‘users’ branch and any content that they create is chained to that branch. The ‘special’ branch is used to store bits which represent special pages of the content framework such as the homepage, the search results page and a bit to display the related content of some other chain.

³Note that it is in fact the bit associated with the chain, rather than the chain itself, which holds the tag.

To present the capabilities of the framework's type system, three type handlers were implemented: the default type, `article`, allows articles to be written in a very basic wiki markup and have it rendered as HTML; the image type, which produces an HTML image tag for the specified filename; and the Youtube [8] video type, which displays an embedded Youtube video for the specified Youtube URL. The image type is shown in listing 4.2.

```

1 // Define the output string
2 def output = ""
3
4 // If this bit is being called to render itself
5 if (selfRendering) {
6
7     output="Dont chain content handlers."
8 } else {
9     // Check for size parameter
10    String size = parameters.get("size")
11    if (size==null) size = "medium"
12
13    output("<img src=\"common/images/archive/"
14         + size + "/" + chainContent + "\">"
15    )
16    return output;

```

Listing 4.2: Image type handler.

On every page the CMS shows items related to the one currently being viewed, as illustrated in figure 4.3. This is not affected by the type of the current item and is implemented separately from the type handler. Since the framework models associations top-down, the parent of any element is not generally considered an associated item. As a result, the parent of a chain is not included when displaying related items of content.

The search results page is implemented as a code-bit and is an example of a 'special' chain discussed above. To pass the search query from the CMS to the code-bit, the parameter feature is used. The source code for the code-bit is shown in listing A.1. To give the CMS easy and optimised access to special chains the `ChainDAO` has a method `getSpecialChain(String)`. This holds the special pages in a cache for faster access. Bits for special chains are marked with the tag:

```
tmb.framework.feature
```

where 'feature' is the name of the special page, 'homepage', for example. This is passed as the `String` parameter to `getSpecialChain(String)` and the requested

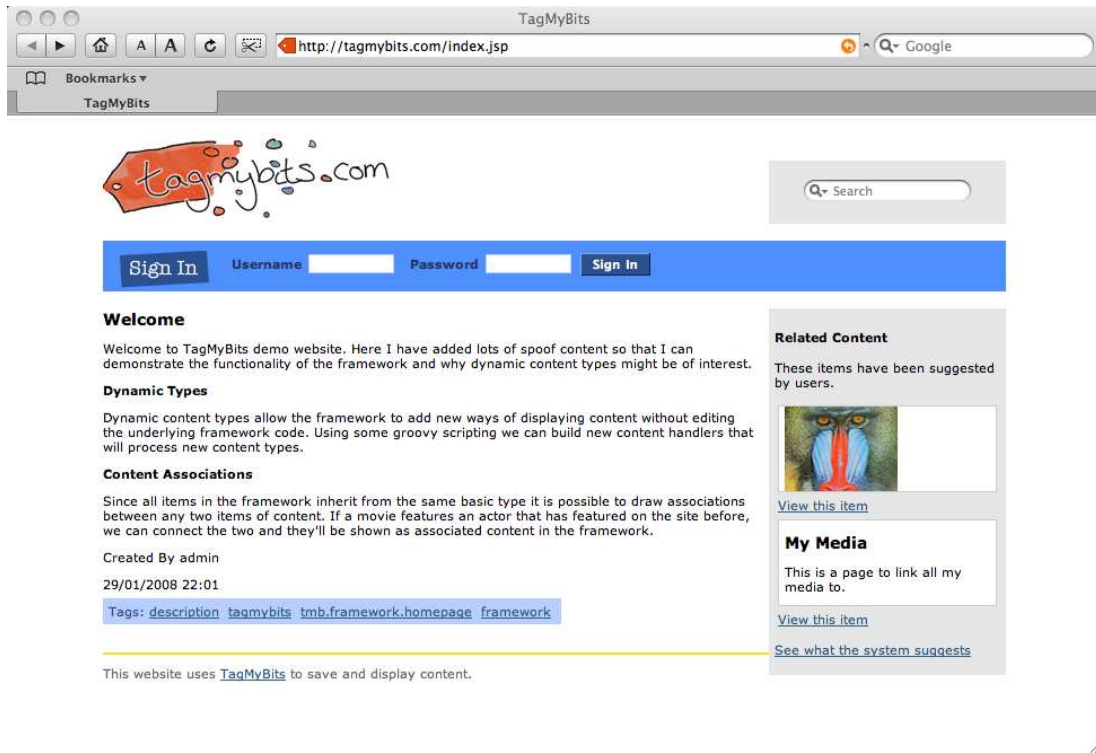


Figure 4.3: Screenshot of the CMS homepage.

chain is returned.

Constructing an interface for some aspects of the system proved challenging as it was often difficult to refer to chains or bits in a way that the user could readily understand. Since bits have no title or description, there was no obvious way to summarise their content. This was especially noticeable when implementing the interface to add and remove bits to/from chains. Since the bits did not have any context, or were code-bits, they could not be rendered as they would be in the site. Where possible, content was rendered as it would appear on the site, but in a confined space. This is shown in figure C.1. In a setting where this page would be viewed by many users there could be an impact in performance as it requires that every chain be individually rendered.

Another issue that was highlighted in the CMS was the difficulty of implementing a standard navigation panel. In this implementation, almost all navigation is done through the related content items, or by searching the site. Even then, the search feature has to have its special chain value hardcoded into the CMS. While this is not strictly the responsibility of the content repository, it does not present a clear way to implement navigation in the CMS.

5. Evaluation

The objectives of this project are defined in section 3.1.1. This chapter evaluates the implementation of the framework against the objectives as well as relevant engineering criteria to determine the potential of the framework.

5.1 Specification Objectives

To demonstrate modularity and flexibility the framework must be able to:

Add new content types and content handlers without requiring changes to the framework. This has been achieved by the dynamic type system implemented in the data model. New types are handled by code-bits and subsequently the data model does not require alteration to cope with new types.

Allow a highly flexible organisation of content. Since there is no prescribed structure to content, any structure that the framework presents is entirely derived from the associations created within the data model. As these are generated by the user or CMS, the organisation of content is entirely flexible.

To demonstrate that the framework promotes reuse of content it must be able to:

Allow content to appear at various points in the framework. This is achieved by using associations to form relationships between content. Where two items of content are linked they can be shown together, or as items the user may be interested in. Code-bits may even render associated items in-place. Since associations may be made between items and may also be independent of existing associations, content can essentially be reused anywhere within the repository.

Display content *explicitly* and *implicitly* associated with the content currently being viewed. The framework models an explicit association as a child of a chain, these are used to calculate sets of related content. This was highlighted in the CMS implementation. In this implementation, implicit content associations can be retrieved using the `chainDAO` method `getImplicitChildren(Chain)`. While the CMS does not display these at the same time as the initial content, this is primarily due to lack of screen real-estate.

5.2 Engineering Criteria

The CMS developed using this framework demonstrates that it is possible to use this data model for a practical application. To some degree this gives an indication as to how easy it is to develop applications based on this framework. However, the account may be biased as the application was developed by the creator of the framework (Unfortunately, an independent test of this type was not feasible within the time constraints of the project). Therefore, while the CMS shows the possibility of building an application based on this framework, an assessment of engineering properties afford us a fairer evaluation of the framework's suitability for certain tasks. This assessment was based on a number of factors:

- Fitness
- Dependability
- Maintainability
- Scalability
- Security
- Cost

In comparison to traditional engineering requirements, the usability of the framework was not evaluated as it would have required time and resources outside the scope of this project.

5.2.1 Fitness

As established earlier in the report, this storage framework is specifically designed as a content repository and provides the basic features for content manipulation and storage. Users may create, read, update and delete content items and the framework also provides a means to describe content through tags. Content items contain a body-of-content and fields to store when the content was created and last updated, as well as basic ownership information. This makes it at least suitable for use in a basic CMS.

The framework does not provide features for workflow management, content permissions, or versioning and therefore would not be suitable for use in an Enterprise CMS which requires support for business protocol during document generation.

5.2.2 Dependability

The dependability of the framework was assessed through a series of unit tests to ensure that the framework produced reliable results. This provides an estimate of how reliably the framework behaves. These tested the DAO classes and the data model classes. Any subsequent implementations should be tested against these unit tests to ensure that changes they make do not break the existing functionality of the framework.

Users intending to create their own types for content should be encouraged to test their implementations to ensure they do not affect the stability of the application and that they return reliable results. Currently there is no method to support this in the framework.

It does not consider how robust the framework is to failure under certain circumstances.

5.2.3 Maintainability

Because of the dynamic type system the framework uses, content in the framework is more complicated to maintain in the underlying data structure. To perform data-cleanup or optimisations on data types would require database join operations to retrieve type information. Conversely, there are fewer tables that require maintenance and so operations that do not rely on type data may be simpler to execute.

The current framework implementation has no support for content versioning and consequently there is no way to roll-back erroneous changes. Versioning could be achieved with few changes to the implementation.

5.2.4 Scalability

In this implementation of the framework, Hibernate is used for ORM and allows the framework to make use of object-caching. This allows the system to make fewer calls to the database as objects can be stored in-memory across requests, hence improving performance. Under heavy use it would be advisable to run several instances of the framework on independent systems to spread the load across several systems. Using the framework it is possible to run the database system on a separate server. Since the implementation was developed in Java, it could also be used with a distributed memory toolkit such as Terracotta [14] to manage object serialisation and transactions across networked systems.

Assuming that content is always used in at least one location, each addition of content will require two new records in the database: one to represent the bit, another to represent the chain. Any time an item of content is reused in the system, only one entry is needed to create a new chain. The addition of tags requires one entry per tag, per bit. Since the addition of content relies solely upon the insertion of rows into the database the framework will only be limited in size by the constraints of the underlying database.

Displaying an item requires that the framework render the bit of a chain and, potentially, the children of that chain recursively. In this model it is possible to generate infinitely long chains so a limit of 3 was applied to the render recursion depth. This means that only the original chain, its children, and its children's children will be rendered.

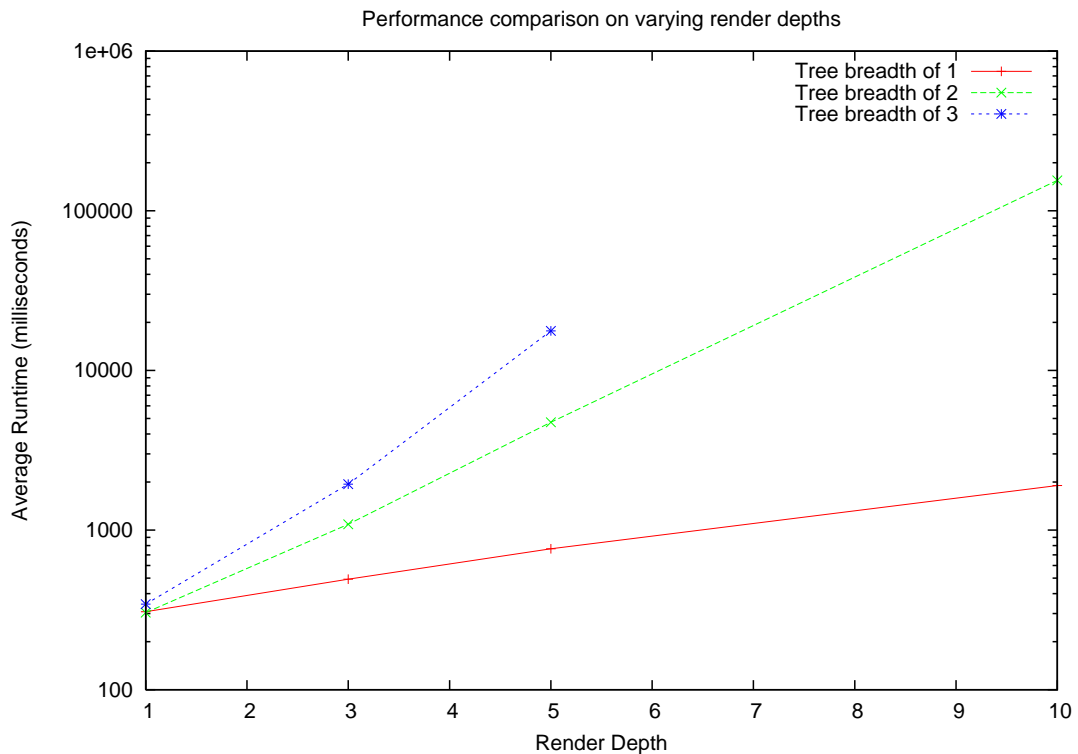


Figure 5.1: Results of the render test with respect to depth of chain.

Two sets of tests were performed to calculate how long or wide chains impact render time. These were all performed on the same system, a MacBook 2GHz Core 2 Duo with 2Gb main memory, and were sampled 5 times to obtain an average runtime. To provide a recursively rendering chain, a simple code-bit was created; this is given in appendix listing A.2. Figure 5.1 shows how the render time of a chain changed as the depth of the render increased; these results are detailed in appendix B. This test was based on chains 1, 2 and 3 children wide

at every node. This structure grows exponentially, as consistent with the size of n-ary trees [13]:

$$N_{elements} = \frac{k^h - 1}{k - 1}$$

Where k is the breadth and h is the depth of the chain.

due to this the breadth of chains was not increased past three children¹. Figure 5.1 illustrates that the time required to render a chain increases exponentially with respect to the the depth of the chain. However, the time required to render a chain grows only linearly with respect to the number of elements in the chain.

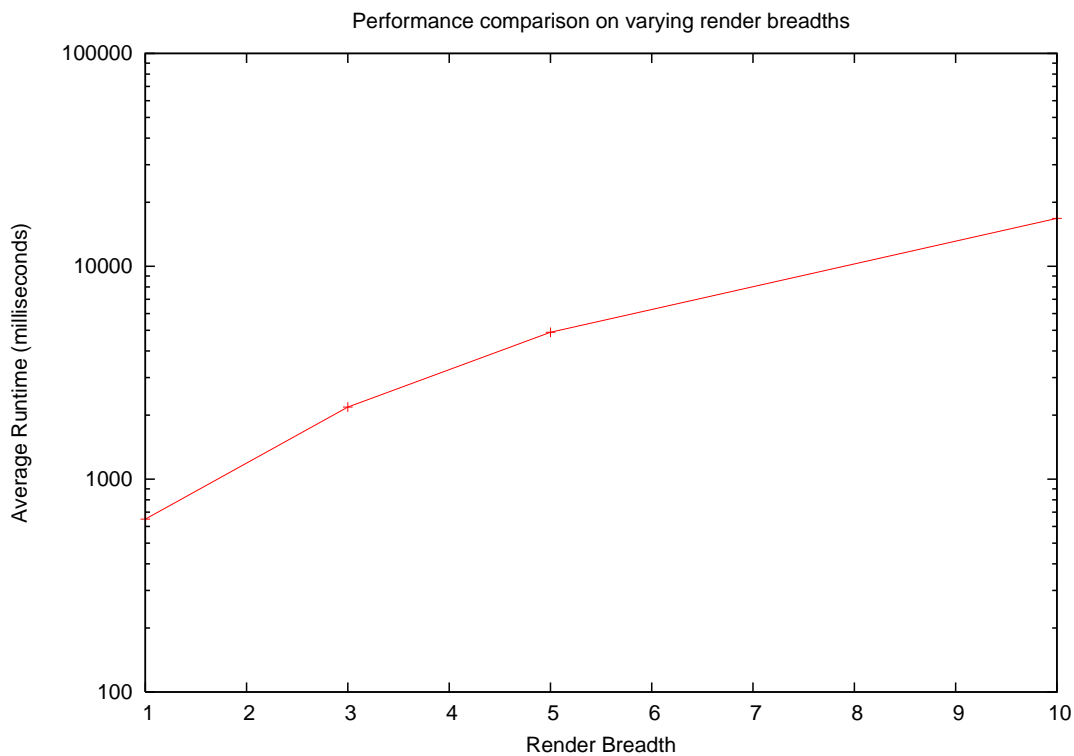


Figure 5.2: Results of the render test with respect to breadth of chain.

The impact of breadth on render time was evaluated against chains within the depth limit of the framework. Figure 5.2 shows how render time changed as the breadth of the chain increased. This test was based on chains 1, 3, 5 and 10 children wide. The depth in this test was fixed at 3 and so the number of chains to be rendered grew linearly, again consistent with the size of n-ary trees.

These results show that it is not feasible to render content to arbitrary depths. Since the strength of associations is likely to get weaker the deeper the render

¹The test for a chain with depth 10 and width 3 was omitted due to the massively large data set required to evaluate its performance.

is from the original item, it is logical to introduce a depth limit at some point. While the limit confines the depth of chains, the implementation does not limit the breadth of chains that can be rendered. In the worst-case, each chain would be linked to every bit in the framework, however this is unlikely to happen in practice as associations should only be defined between items of related content.

5.2.5 Security

Code-bits are currently implemented using the scripting language Groovy and allow the implementation to tackle the problem of pluggable types. While this provides a simple and natural way for developers to define custom types – Groovy is very similar to Java – it does not restrict the operations that a developer can perform. Consequently, it is possible to write code including loops, recursion and other exploits which may result in memory overflows, non-terminating code or even provide access to the server filesystem. Clearly this is not an ideal solution and a scripting language that limits this functionality would be more appropriate.

The parameter functionality of the Hibernate query framework was used to ensure that the system was not susceptible to SQL injection; an attack used to circumvent the framework and perform queries or updates directly on the underlying database. Therefore, it should not be possible for malicious users to circumvent the framework and perform their own operations on the database.

Taking into account these security considerations, the framework could be deployed publicly, but it would not be advisable to allow untrusted users to create content, especially type handlers.

5.2.6 Cost

The framework requires no specialised hardware and is built using free, open source components. Development time should be reduced as the developer does not need to deal with content persistence or retrieval though some time will be required to implement type handlers that are appropriate to the application being developed. This should be low as handlers are written in a well documented scripting language.

6. Further Work

Through the evaluation and testing of the framework, a number of areas were identified where further development could improve the quality and functionality of the system and are discussed below.

The framework presents an entirely new data model and, while this implementation uses a relational database as a backend, it could be implemented using a variety of different storage mechanisms. A custom built storage engine would allow the framework to benefit from performance optimisations and, perhaps, a more natural mapping of the data model to the underlying data store. With or without a custom storage engine, the framework would benefit from its own query language. This would allow developers using the repository to construct their own queries without relying on the query methods provided by the relevant DAO. It may also lead to a more appropriate method of querying the data model from within code-bits.

Code-bits are currently implemented using the scripting language Groovy. As discussed in the evaluation, this is not ideal as it provides malicious users a mechanism to attack the system. A preferable solution that was out of the scope of this project would be to implement a custom scripting or markup language that allows code-bits to process information in a way that does not permit loops, recursion or access to the server. It may be useful to allow a tiered system for code-bits so that users with permission to administrate the system may include loops or other scripting features, whereas users that are not administrators can only create code-bits containing markup.

Due to the flexibility of the type system it is difficult for applications using the repository to enforce constraints on content data. Without a way of checking that content is well-formed it is possible for a user to define some item of content as any type regardless of its actual type. By introducing a method of type-templating it would be possible to run checks on the form of data and use them to provide feedback to the application. For example, if a user specifies an image URL that does not exist, the application could warn the user, or highlight the image upload utility.

The above suggestions are specific to this framework. However, the current implementation would benefit from enhancements that are more general to the field of content management. Versioning would provide the system with a means to roll-back to any previous state of an article and such a feature is crucial in environments where multiple users edit the same item of content. A permissions system would allow the framework to enforce restrictions on who can manipulate items in the framework and how. Workflow management would allow users in

corporate environments to enforce manager or moderator approvals before new versions of content are available to other users.

7. Conclusion

In this project, a new data model was presented for organising items of content based on the relationships between them. These relationships are defined explicitly and allow the system a means by which to recommend related items of content to the user. By using these associations to combine items it is possible to generate sophisticated displays of content with minimal effort. This data model differs greatly from the traditional hierarchy or index-based approach to content organisation and allows the CMS or even the user to define how content is organised.

There are some weaknesses in the model as developed at this stage. Omitting title and description metadata has made it difficult to design certain elements of the CMS interface as there is no way of summarising content. Since the organisation of content is flexible it becomes less obvious how to create a standard navigation panel and, more generally, by radically simplifying the data model, the implementation of the framework has become significantly more complex for what should be simple cases.

Deficiencies in the implementation of this model were also identified. While precaution was taken to shield the framework from SQL injection attacks, relying on a fully featured scripting language to implement types and code-bits leaves the system unnecessarily exposed to malicious users or even the malice of untested code. Additional features such as versioning and a permissions system would also be desirable to realise this framework as a fully featured content repository, but they are not part of this implementation.

Chapters 4 and 6 discuss these challenges and how they can be overcome with further development.

Overall, the framework detailed in this report meets the objectives of the project by providing a flexible approach to organising content storage and allowing modular content to be easily reused across the framework.

Bibliography

- [1] Christian Bauer and Gavin King. *Hibernate in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.
- [2] Drupal. Drupal handbook. <http://drupal.org/handbooks>, 2008.
- [3] Magdalini Eirinaki, Michalis Vazirgiannis, and Dimitris Kapogiannis. Web path recommendations based on page ranking and markov models. In *WIDM '05: Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 2–9, New York, NY, USA, 2005. ACM.
- [4] Codehaus Foundation. Groovy homepage. <http://groovy.codehaus.org/>, 2008.
- [5] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.
- [6] Mike Grogan and JCP Group. Scripting for the java platform, final draft specification. <http://www.jcp.org/en/jsr/detail?id=223>, 2006.
- [7] Eui-Hong (Sam) Han and George Karypis. Feature-based recommendation system. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 446–452, New York, NY, USA, 2005. ACM.
- [8] Youtube LLC. Youtube - broadcast yourself (homepage). <http://www.youtube.com/>, 2008.
- [9] David Nuescheler, Peeter Piegaze, and JCP Group. Content repository for java technology specification. <http://www.jcp.org/en/jsr/detail?id=170>, 2005.
- [10] National Information Standards Organization. The dublin core metadata element set. <http://www.niso.org/standards/resources/Z39-85-2007.pdf>, 2007.
- [11] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.
- [12] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

- [13] James A. Storer. *An Introduction to Data Structures and Algorithms*. Birkhauser Boston, 2001.
- [14] Open Terracotta. An introduction to terracotta dso. <http://terracotta.org/confluence/display/orgsite/Whitepapers>, 2007.

Appendices

Appendix A. Code Examples

A.1 Code Bit: Search By Tags

This example shows how a code-bit can be used to search the framework based on some query parameter, and display the results.

```
1 import com.tagmybits.model.*;
2 def output = "";
3
4 // If this bit is being used to render itself
5 if (selfRendering) {
6     output+= "<h1>Tag Search</h1>\n"
7     output+= "<p>You searched for bits with the tag <i>"
8         + parameters.get("qsQuery") + "</i></p>\n"
9     // Search for bits with the query tag
10    Set<Bit> bits = codeDAO.findBitsWithTag(
11        parameters.get("qsQuery"), 50,0);
12    // If there are matches
13    if (bits.size()>0) {
14        // Create render parameters
15        HashMap<String, String> newParameters =
16            new HashMap<String, String>();
17        newParameters.put("size", "small");
18        boolean toggle = false;
19        // For each matching bit
20        for (Bit bit: bits) {
21            List<Chain> instances =
22                codeDAO.findBitInstances(bit.getId(), 1,0);
23            // If it has a chain instance show it
24            if (instances.size()>0){
25                Chain instance = (instances).get(0);
26                if (instance!=null) {
27                    if (toggle)
28                        output += "<div class=\"row\">\n"
29                    else {
30                        output += "<div class="
31                            + "\"row-toggle\">\n"}
32                output += "<div class=\"content\">\n"
33                output += codeDAO.renderChain(instance,
```

```

34         newParameters)
35         output += "</div>\n"
36         output += "<a href=\"index.jsp?\"
37             + \"chainId=\"
38             + instance.getId() + \"\>\"
39             + \"View this bit</a>\"
40         output += "</div>\n"
41     }
42 }
43 }
44 // If no bits match the query
45 } else
46     output += "<p class=\"empty\">No bits were "
47         + "found with that tag.</p>\n"
48 // If this bit is being used to render another bit
49 // display an error message
50 } else {
51     output += "This bit should be self rendered."
52 }
53 // return the result
54 return output;

```

Listing A.1: Code-bit to display search results.

A.2 Code Bit: Render Test

This code-bit was used as a content handler for the render test discussed in section 5.2.4. This code-bit prints its ID and calls the render method on any children it has.

```

1 import com.tagmybits.model.*
2 def output = ""
3 if (selfRendering) {}
4 else {
5     output += chain.getId()
6     Set<Chain> children = chain.getChildren()
7     for (Chain chain: children)
8         output += " " + tmbDAO.renderChain(chain, null)
9     output += "\n"
10 }
11 return output;

```

Listing A.2: Code-bit for recursive render test.

Appendix B. Test Results

B.1 Render Time Results

Table B.1: Results of the render test with respect to depth of chain where **breadth is 1**.

Depth	Elements	Average Runtime (ms)	Runtime/Elements
1	1	308	308
3	3	492	164
5	5	764	153
10	10	1901	190

Table B.2: Results of the render test with respect to depth of chain where **breadth is 2**.

Depth	Elements	Average Runtime (ms)	Runtime/Elements
1	1	304	304
3	7	1087	155
5	31	4732	153
10	1023	155154	152

Table B.3: Results of the render test with respect to depth of chain where **breadth is 3**.

Depth	Elements	Average Runtime (ms)	Runtime/Elements
1	1	344	344
3	13	1940	149
5	121	17694	146
10	29524	<i>Not Calculated</i>	<i>Not Calculated</i>

Table B.4: Results of the render test with respect to breadth of chain where **depth is 3**.

Breadth	Elements	Average Runtime (ms)	Runtime/Elements
1	3	649	216
3	13	2179	168
5	31	4896	158
10	111	16781	151

Appendix C. Screenshots

C.1 Content Management System

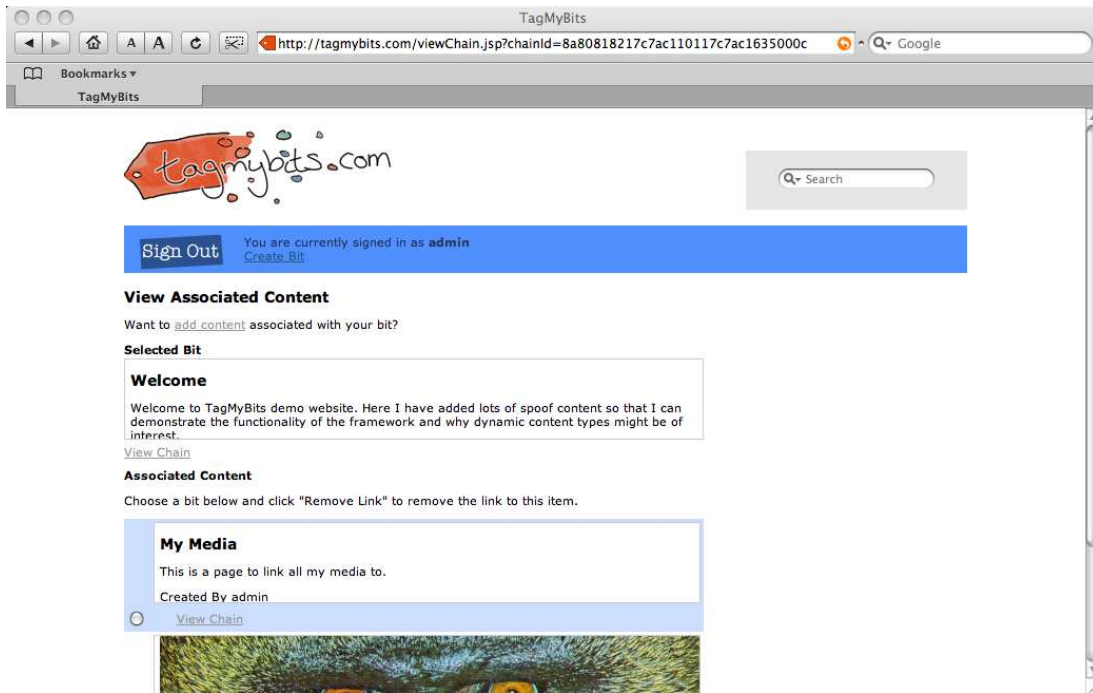


Figure C.1: Screenshot of the CMS showing cropped versions of rendered content.