Emergent Architecture Design

For MoodCat

Eva Anker (eanker 4311426)
Gijs Weterings (gweterings 4272587)
Jaap Heijligers (jheijligers 4288130)
Jan-Willem Gmelig Meyling (jgmeligmyling 4305167)
Tim van der Lippe (tvanderlippe 4289439)

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Emergent Architecture Design

For MoodCat

by

Eva Anker (eanker 4311426)
Gijs Weterings (gweterings 4272587)
Jaap Heijligers (jheijligers 4288130)
Jan-Willem Gmelig Meyling (jgmeligmyling 4305167)
Tim van der Lippe (tvanderlippe 4289439)

Supervisor:	Cynthia Liem	TU Delft
Teaching Assistants:	Friso Abcouwer,	TU Delft
	Abhishek Sen,	TU Delft

An electronic version of this document is available at
https://github.com/MoodCat/MoodCat.me/.

TU Delft Delft University of Technology

Contents

1

Introduction

This document provides a sketch of the system that is going to be built during the context project multimedia services. The architecture of the syste is defined by the system's high level components. These components are split in to sub components and sub-systems.

1.1. Design goals

The following design goals will be preserved throughout the project:

- **Deployability**
  The system will be developed in such a way that we can always deploy the most current version (Continuous Integration [1]). Being able to deploy the system at any time allows us to keep the work required for a release managable. Because the current version should be deployable at anytime, developers are enforced to only alter the system in a non-breaking way. Another advantage is that smaller releases[1] imply a lower possibility of introducing bugs.

- **Portability**
  The frontend should work on *Evergreen* browsers [2]. The backend should run on both Windows and Linux server environments capable running Java software.

- **Simplicity**
  The system will be developed with simplicity in mind. Existing libraries and frameworks will be investigated and used if applicable, so we can focus on developing the system rather than the tools required to build the system. This will keep our codebase relatively small and allows us to focus on the user experience and underlying algorithms rather than reinventing the wheel for the used techniques.

- **Object-oriented programming**
  Object-oriented programming[2] will be used to develop the system. We will separate the system into subsystems and components (packages, classes and interfaces). This should enable code reuse and extensibility, and improve the testability of the sytem, because each component can be tested individually[3].

- **Usability**
  Users should be able to use Moodcat intuitively. All features of MoodCat should work for modern browsers. Moodcat should work properly with accessibility tools like screenreaders.

[1] We define *smaller releases* as a relatively small change to the codebase.

[2] «The term "evergreen browser" refers to browsers that are automatically upgraded to future versions, rather than being updated by distribution of new versions from the manufacturer, as was the case with older browsers.» - http://www.techopedia.com/definition/31094/evergreen-browser

2

Software architecture views

This chapter discusses the architecture of the system. The system is first decomposed into smaller subsystems and the dependencies between the subsystems are explained. In the second paragraph the relation between the hardware and software of the system is elaborated. The third paragraph illustrates the data management of the system.

2.1. Subsystem decomposition

Moodcat is subdivided into two components: the frontend website and the backend processing service. In this chapter we will explain how these components are composed and how they interact with each other.

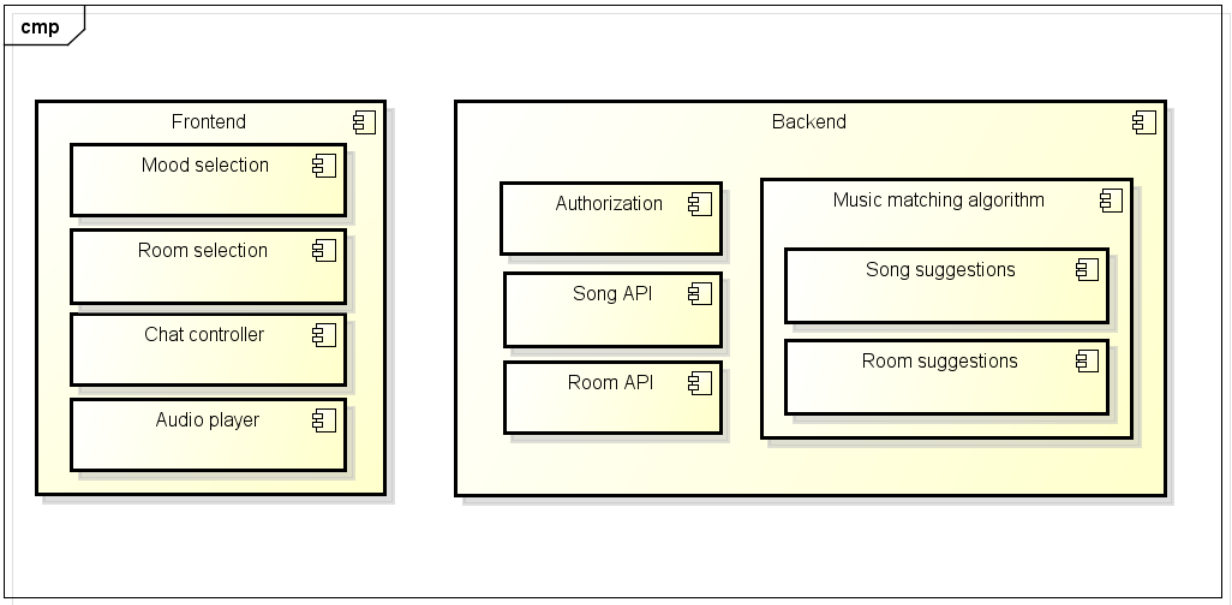

Figure 2.1: Architecture overview for Moodcat

2.1.1. Moodcat Frontend

The frontend will be the visible part for the user. After logging in to our services, the user will be able to select his mood and based on that MoodCat will provide several music listening rooms. In these rooms the user can interact with other users through the chat while listening to the music.

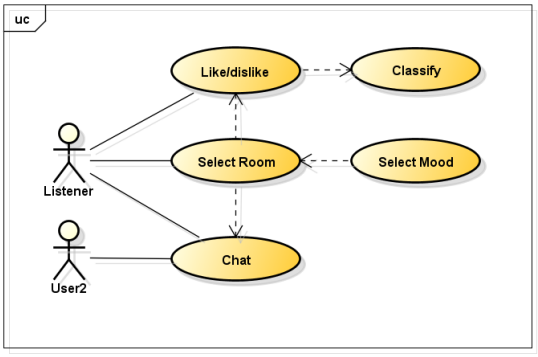

Figure 2.2: Use case diagram for Moodcat

The MoodCat frontend will be web-based. This allows us to support all modern devices which enables a large target audience. We will develop the frontend using modern techniques as HTML[4], Javascript and CSS[5]. Twitter Bootstrap will be used as boilerplate for our own customized CSS theme.

AngularJS[6] will be used as web framework, providing us two-way data binding between the user interface and backend, and a standardized way of composing user interface controlling logic through *controllers*, *services* and *directives*.

The Object-oriented[2] and Dependency Injection[7] driven nature of AngularJS makes it suitable for unit testing[3]. Testing the frontend codebase will be done using the Karma[8] testrunner, the Jasmine[9] testing framework and the PhantomJS[10] headless[1] browser.

Frontend structure

Angular works with *controllers*, *services* and *directives*. Directives can thought of as physical elements on the webpage, yet they have *controller* logic attached to it. Directives can be double binded to a scope. This means that the directive can both read and write to the scope. Scopes in Angular are ViewModels, and can contain both data and methods. A scope can recursively resolve data from its parent *scope*, this is implemented through JavaScripts Prototype inheritence. The top level scope is the so-called *$rootScope*. Services are singleton objects that can be used for shared logic between controllers.

The system will be build around *services* that connect with our backend API, for example the **RoomService** that queries rooms and the **ChatService** that manages chat messages within a room. The *$rootScope* will be used to store data such as the selected moods, current room and the current song.

2.1.2. Moodcat Backend

The Moodcat backend has several subcomponents, each with their own responsibility:

- the *Static File Server* that serves the frontend website on a webserver;
- the *REST API* which is used for communication about rooms and songs between the backend and frontend;
- the *Music Matching Algorithm* which is used to generate music suggestions for rooms and to find a room for a user.

In the following paragraphs we will explain the last two subcomponents.

REST API

The backend keeps track of which rooms are active and what song is currently played in the room. When the backed is started, it creates a room instance for each room in the database. For each room instance, the backend keeps track of a current song, a history of previously played songs and a playing queue for future tracks to play. The information of a room and its current song can be requested by

[1] JavaScript runs in browsers and the written logic needs a DOM to interact with. A headless browser does not actually render the page and is therefore much faster in intergration tests.

clients through the API. Next to requesting data, the API can process user votes and classifications in order to improve the system. When an API call results in the change of an attribute of a room, the room is updated in the database. Furthermore, the backend is responsible for transmitting the chat messages between the users in the room. An example interaction between the frontend and the backend is given in 2.3.

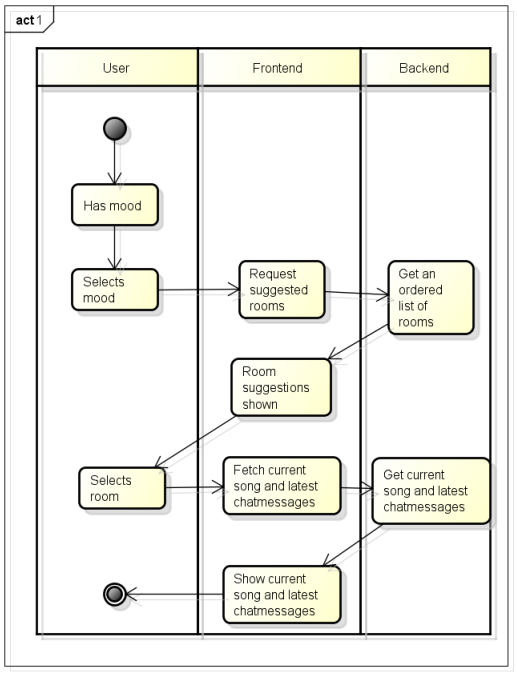

Figure 2.3: Example interaction frontend and backend

The backend and the frontend communicate over HTTP requests in the JSON format. The REST API will be written in Java using JAX-RS and Jackson. Jackson serializes Java objects to a JSON string based on their attributes, and vice versa: it is able to deserialize a Java object from a JSON string. This allows us to use JSON in the frontend, which is ideal in Javascript, but use classes and objects in Java, which greatly improves the testability of the responses. The backend will be tested with unit tests written using JUnit and Mockito.

Music matching algorithm

The music matching algorithm determines and compares moods for songs, rooms and users. The classification of songs will be done by classification from the user, which will be encouraged with gamification. Moodcat will improve over time, because the more people classify the more accurate it gets. All entities are modeled with a valence and arousal vector. Valence and arousal are measured in a range of -1 to 1. By default, a song will be classified with the default vector: valence 0 and arousal 0. This would mean an absolute neutral mood. This classification (mapping) will be updated over time by processing user feedback. Therefore the longer the system is running, the more accurate the classifications become.

When a user votes down, the user will be requested to classify the song. If there are more up-votes than down-votes the classification will not have effect, because the majority of the audience says the song fits the room.

Using a nearest-neighbour algorithm we will determine which song fits a room and which room fits a user. This can ultimately be improved by personalising according to the listening history of the user.

2.2. Hardware/software mapping

For navigating through the user interface and posting messages in the chatroom a keyboard and mouse, or a touch screen with onscreen keyboard is required. We will annotate our elements with proper Aria

accessibilty attributes so that assistive technologies, such as screen readers, will work with our service properly.

Moodcat requires an audio interface to play music. To interact with the audio interface from the browser, the Web Audio API[11] and ngAudio[12] are used. The users browser should support these technologies in order to use Moodcat.

2.3. Persistent data management

Moodcat has to handle both song data and chatroom data.

We try do deduplicate artists from the Soundcloud data as much as possible. We identify artists in our database by an unique identifier. Besides this identifier, we store the name for the artist. The name will be used to display in the frontend.

For the song we store the basic properties, like artist, song title and duration. A song has an unique id in our system and also an id from Soundcloud, in order to fetch data from Soundcloud. We also include an url to display the artwork of a song.

Furthermore each song has its own valence and arousal value, this is updated when users classify the song. This classification will be done in the Music Matching Algorithm as described above. In order to prevent incorrect or unwanted classifications, we also have to store the netto value of up-votes compared to the down-votes.

Each room has an unique room identifier, which is used to find the room requested from the API call. We also store some properties: The name of the room to display on the frontend, the currently playing song including its metadata, and a playing queue and history of played songs. Furthermore, the arousal and valence values of the room are stored.

For a chatmessage we store the author, actual message, timestamp and the room in which the message was posted.

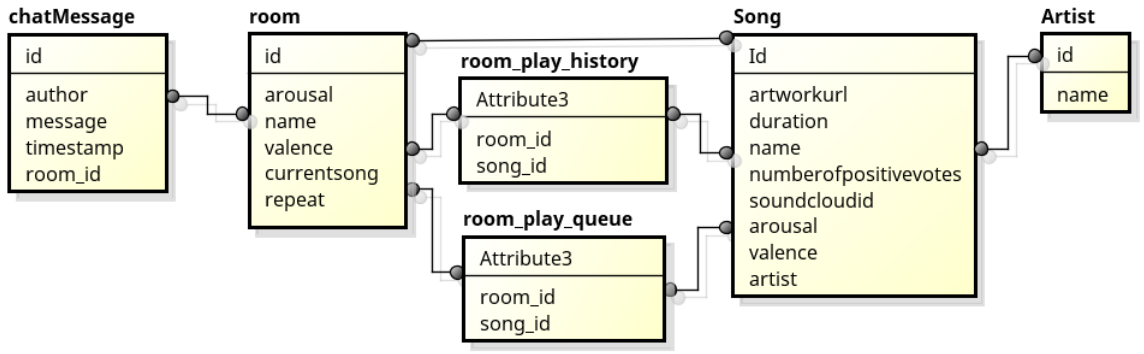

Figure 2.4: ER diagram for Moodcat

Most of the currently used data will be stored in memory, as it would be intensive, for example, to query the database for each chat message that comes in. However, the data will eventually be persisted in a PostgreSQL[13] Relational database. On one hand, it would be impractical to store our entire dataset in memory, on the other hand, then the system can continue where it has left off after a restart.

We will connect our Java backend server to the PostgreSQL database using the Java Persistence API (JPA) and the Postgres JDBC driver. The Hibernate[14] Object-relational mapping (ORM) framework will be used to map Java class instances to tables. QueryDSL[15] JPAQuery will be used to build SQL queries from Java code. We have chosen for these techniques because they provide a nice and clean API to bind Java objects to database objects.

2.4. Concurrency

In order to scale up to thousands of connected clients, we should develop with concurrency in mind from the very first start. Moodcat uses the Soundcloud API in order to stream media to the clients. This means that the audio stream does not have to pass through our servers. We do however have

to handle the various API calls to interact with the backend, for example to play the next song or to broadcast a message in a room.

It is important that these requests do not block each other, therefore we have chosen a servlet container, Jetty, that uses asynchronous sockets - which will not block if there is no data to receive yet.

Besides the communication with the clients, Moodcat should also keep its algorithm (described in 2.1.2) running in the background. This algorithm will run in seperate threads to optimize throughput. These seperate threads will get a specific task assigned, which has his local data to compute on. Deadlocks will be prevented because the spawned threads will be joined before acquiring locks in synchronized statements. This will happen when we persist the algorithm output in the database.

2.5. Scalability

Concurrent clients are one thing, but we also have to scale the algorithms for growth. Initially, we planned on a nearest neighbours algorithm to match songs to rooms. Later on, we found out that (while this works well for small datasets,) finding the nearest neighbours in a system of a few million songs can prove to be a lot harder. The goal is clear: Find a way for the system to scale further. We went over several iterations of the system. Once we realised the issue with plain Nearest Neighbours, we went for an ANN[2] algorithm to be able to use the nearest neighbour algorithm. This initially was planned to be implemented using a KD-Tree [16]. However, this would cause memory issues due to a O(N) space complexity of the algorithm. The second suggestion was retrieving a random subset of songs to run the NN algorithm on. We ran some calculations, and it looked promising, but it underestimated the locality clustering of the data. We also considered LSH[17], but this is better suited for high-dimension data, and MapReduce, but this wasn't feasible for the amount of time we had. In the end, we threw away the original NN implementation we had and use spatial indexing in the database, using RTREE [18]s.

2.6. Design Patterns

Due to the usage of various libraries and frameworks, our codebase does not contain a lot of helper mechanisms. Therefore the number of design patterns implemented is lower, because we use a lot of patterns defined in the frameworks.

The design patterns we use, as they are defined in the frameworks, are:

1. Lazy initialization (Hibernate):
   OneToMany-relations are fetched from the database when they are needed. This not only makes retrieval of entities that use embeddables faster, we can also database calls. For example most of the time we are not interested in the songs that an artist produced, just his/her name.

2. Object pooling (RestEasy, Guice):
   The http-connection to SoundCloud and database-connection to our PostgreSQL database are pooled for re-use. This way we only have to set up the connection once using said frameworks and reduce object retrieval time. The **SoundCloudAPIConnector** uses a RestEasyClient. The database connection is handled by C3PO which is an extension of Hibernate, our Object-relational mapping framework.

3. Memento (JPA, Guice):
   All database updates and retrievals are transactions, which implement the Memento design pattern. This makes sure that all database changes are ACID[**?**]. Classes instantiated through Guice dependency injection can annotate methods with @Transactional, so that the method is intercepted to enforce that the method is ran in a database transaction. This starts a new transaction if no transaction is active for the current thread. If any exceptions occur (either in the Java or database layer - for example unmet key constraints) the transaction will automatically be rolled back after the exception has been propagated upwards. All extensions of **AbstractDAO** have transactions.

4. Builder (QueryDSL, HttpClient):
   We use QueryDSL in order to easily create SQL queries that map to our Hibernate entities.

[2] Approximate nearest neighbour

QueryDSL allows you to write SQL queries using a builder pattern. For each database entity, QueryDSL generates a few classes (so named QClasses) which allow you to write SQL queries using a builder pattern. This introduces type safety, because schema changes will now cause compile errors if a query is not updated.

The HttpClient that we use for our HTTP calls to the SoundCloud API also uses a builder pattern to set the path, parameters and headers for the request to be made.

The design patterns we implemented ourselves are:

1. Singleton:
   In order to enforce a single controlling unit, all of our backends [**?**] are singleton. For example the **ChatBackend** contains all **RoomInstances**, which should be stateful[3] in order to maintain cached collections of **ChatMessages**. This speeds up message retrieval for users that join a room. It also reduces the amount of database updates, which lowers the network traffic. Also, the Angular *services* for the backend, described in 2.1.1, follow the singleton pattern as well.

2. Facade:
   All APIs and backends implement the Facade design pattern. They are a single end-point which can connect several backends or DAOs. Therefore we can change the actual storage- and retrieval-structure, while making sure our API end-points and backends still provide the correct data in the correct structure.

3. Observer (not implemented yet):
   We will use the Observer pattern after converting our currently poll-based architecture to a stream pipeline using websockets. This way the frontend can continuously receive updates whenever the backend notifies the clients. Therefore chat messages can be instantly received and no unnecessary calls shall be made to the frontend. Currently the frontend polls every second the backend to check if there are updates.

[3] Stateful room instances should not be confused with the stateless nature of the REST-api. From the clients point-of-view all requests are stateless and don't influence future requests.

Glossary

Aria Accessible Rich Internet Applications defines ways to make Web content and Web applications (especially those developed with Ajax and JavaScript) more accessible to people with disabilities. 4
arousal How calm or excited you are. 5

class An extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions, methods). 1
Continuous Integration The practice of merging all developer working copies with a shared mainline several times a day. 1
CSS Cascading Style Sheets is a style sheet language used for describing the look and formatting of a document written in a markup language. 3

framework An abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. 1

HTML HyperText Markup Language is the standard markup language used to create web pages. 3
HTTP The Hypertext Transfer Protocol is an application protocol for distributed, collaborative, hypermedia information systems. 4

interface A common means for unrelated objects to communicate with each other. These are definitions of methods and values which the objects agree upon in order to cooperate. 1

Javascript A dynamic programming language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed. 3, 4
JSON JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. 4

library A collection of non-volatile resources used by computer programs. 1

Object-oriented programming Programming paradigm based on the concept of ”objects”, which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. 1
Object-relational mapping A programming technique for converting data between incompatible type systems in object-oriented programming languages. 5, 6

package A Java package is a mechanism for organizing Java classes into namespaces similar to the modules of Modula, providing modular programming in Java. 1
Prototype inheritence Prototype-based programming is a style of object-oriented programming in which behaviour reuse (known as inheritance) is performed via a process of cloning existing objects that serve as prototypes. . 3

Relational database A relational database is a digital database whose organization is based on the relational model of data, as proposed by E.F. Codd in 1970. This model organizes data into one or more tables (or ”relations”) of rows and columns, with a unique key for each row. 5

Soundcloud An online music streaming service. 5
SQL Structured Query Language is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). 5

valence The amount of attraction you have towards a certain event. 5
ViewModel The view model is an abstraction of the view that exposes public properties and commands. . 3

Bibliography

[1] P. Duvall, *Continuous Integration: Improving Software Quality and Reducing Risk* (Addison-Wesley, Reading, Massachusetts, 2007).

[2] R. Wirfs-Brock, *Designing Object-Oriented Software* (Prentice Hall PTR, Upper Saddle River, New Jersey, 1990).

[3] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Pearson Education, Upper Saddle River, New Jersey, 1999).

[4] *HTML,* http://www.w3.org/TR/html5/, accessed: 2015-04-28.

[5] *CSS,* http://www.w3.org/Style/CSS/, accessed: 2015-04-28.

[6] *AngularJS,* http://www.angularjs.org, accessed: 2015-04-28.

[7] Angular, *Angularjs: Developer guide: Dependency injection,* https://docs.angularjs.org/guide/di, accessed: 2015-04-28.

[8] Karma, *Karma - spectacular test runner for javascript,* http://karma-runner.github.io/0.12/index.html, accessed: 2015-04-28.

[9] Jasmine, *Jasmine - behavior-driven javascript,* http://jasmine.github.io/2.0/introduction.html, accessed: 2015-04-28.

[10] PhantomJS, *Phantomjs - full web stack no browser required,* http://phantomjs.org, accessed: 2015-04-28.

[11] W3C, *Web audio api,* https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html, accessed: 2015-06-05.

[12] ngAudio, *ngaudio - angular directive for playing sounds,* https://github.com/danielstern/ngAudio, accessed: 2015-04-28.

[13] PostgreSQL, *Postgresql - the world's most advanced open source database,* http://www.postgresql.org, accessed: 2015-06-05.

[14] HibernateORM, *Hibernate orm - idiomatic persistence for java and relational databases.* http://hibernate.org/orm/, accessed: 2015-06-05.

[15] QueryDSL, *Unified queries for java. querydsl is compact, safe and easy to learn.* http://www.querydsl.com, accessed: 2015-06-05.

[16] K. Z. et al, *Real-time kd-tree construction on graphics hardware,* http://research.microsoft.com/pubs/70568/tr-2008-52.pdf (), accessed: 2015-06-05.

[17] L. A. et al, *Locality sensitive hashing: a comparison of hash function types and querying mechanisms,* https://hal.archives-ouvertes.fr/inria-00567191/document (), accessed: 2015-06-05.

[18] A. Guttman, *R-trees. a dynamic index structure for spatial searching,* http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf, accessed: 2015-06-05.