GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Uploaddatum: 20.09.2025
Uploadzeit: 19:41

Dies ist ein von FlexNow automatisch beim Upload generiertes Deckblatt. Es dient dazu, die Arbeit automatisiert der Prüfungsakte zuordnen zu können.

**This is a machine generated frontpage added by FlexNow.
Its purpose is to link your upload to your examination file.**

Matrikelnummer: 21678578

GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

Design and Implementation of a Modern Database and Web API for the GroIMP Gallery

David Forys

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

20. September 2025

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor:	Prof. Dr. Winfried Kurth
Second Supervisor:	Hon.-Prof. Eckart Dr. Modrow
Additional Technical Guidance:	Dr. Gaëtan Heidsieck & Tim Oberländer

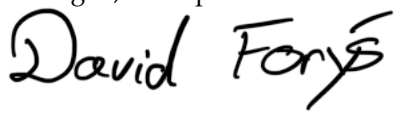
I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules. Annex: Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this paper, I have used ChatGPT or another AI as follows.:

- ☐ not at all
- ☐ during brainstorming
- ☐ when creating the outline
- ☐ to write individual passages, altogether to the extent of . . . % of the entire text
- ☐ for the development of software source texts
- ☐ for optimising or restructuring software source texts
- ■ for proofreading or optimising
- ■ further, namely: bug fixing aid, Latex formatting

I hereby declare that I have stated all uses completely. Missing or incorrect information will be considered as an attempt to cheat.

Göttingen, 20. September 2025



Abstract

The open-source Growth Grammar-related Interactive Modelling Platform (GroIMP) serves as a comprehensive framework for functional-structural plant modelling, supporting a diverse community of researchers and practitioners. However, the existing project sharing infrastructure relies on outdated manual processes that create significant barriers to community engagement and model accessibility. This work addresses these limitations by designing and implementing a modern, database-driven gallery system that facilitates efficient sharing, discovery, and management of GroIMP modelling projects.

The implemented solution comprises three core components: a MongoDB database with a structured schema for storing project metadata and relationships, a Representational State Transfer (REST)ful Application Programming Interface (API) built with Node.js and Express for programmatic access, and an intuitive web interface for browsing and managing projects. The system introduces version control capabilities, enabling users to maintain multiple iterations of their models, and implements comprehensive search and filtering functionality through categorization and tagging systems. Authentication and authorization mechanisms ensure secure access control while maintaining ease of use.

The work included the successful migration of over 300 legacy projects from the file-based storage system to the new database architecture. Custom Python scripts were developed to convert existing metadata formats and automate the upload process, ensuring continuity with historical community contributions.

The resulting platform significantly improves upon the previous infrastructure by eliminating manual processing bottlenecks, providing structured search capabilities, and offering an accessible interface that serves both experienced model developers and domain expert users who primarily seek to apply existing models. The system reduces administrative overhead while enhancing discoverability and knowledge exchange within the GroIMP community, establishing a robust foundation for future collaborative features and community growth.

Contents

1 Introduction **2**

1.1 Background and Motivation . . . 2
1.2 Related Work . . . 3
1.3 Objectives . . . 4
1.4 Scope and Limitations . . . 4
1.5 Thesis Structure . . . 5

2 Analysis **6**

2.1 Existing Infrastructure and Old Data . . . 6
2.2 Requirements . . . 6
2.2.1 Database . . . 7
2.2.2 API . . . 7
2.2.3 Web Page . . . 7

3 Design **8**

3.1 Database . . . 8
3.2 API and Web Page . . . 10
3.2.1 API . . . 10
3.2.2 Web Page . . . 11
3.3 File Management . . . 11
3.4 Supplementary Scripting and Testing Tools . . . 12

4 Implementation **13**

4.1 Server Overview . . . 13
4.1.1 Server Initialization and Database Setup . . . 13
4.1.2 Middleware Stack and Security . . . 14
4.1.3 Routing Architecture . . . 14
4.2 Database . . . 15
4.3 Middleware . . . 15

4.3.1 Authentication and Authorization Middleware . . . 16
4.3.2 Input Validation and Sanitization . . . 17
4.3.3 Data Processing Middleware . . . 17
4.3.4 Utility Middleware . . . 18
4.4 Routing . . . 18
4.4.1 Route Organization . . . 18
4.4.2 Project Data Access . . . 19
4.5 API Workflow Example . . . 19
4.5.1 User Authentication . . . 19
4.5.2 Create Categories . . . 19
4.5.3 Fetch All Categories . . . 20
4.5.4 Create New Project . . . 20
4.5.5 Create a Referencing Project . . . 21
4.5.6 Fetch Specific Project . . . 22
4.5.7 Version Management Operations . . . 22
4.5.8 Session Termination . . . 23
4.6 Web Interface . . . 23
4.6.1 Navigation and Functionality . . . 23
4.6.2 Technical Background . . . 35
4.7 Legacy Data Migration . . . 39
4.7.1 Uploading the Legacy Data to the New Server . . . 41
4.8 Conclusion . . . 42

Bibliography **45**

A Legacy Data **46**
A.1 DESC File Pattern . . . 46
A.2 DESC to JSON Conversion . . . 47
A.2.1 DESC File . . . 47
A.2.2 JSON File after Refactoring with Python Script . . . 47
A.2.3 Database JSON Document . . . 48
A.3 Categories . . . 49

B Database Models **51**
B.1 GroIMP Projects Model . . . 51
B.1.1 Schema Fields . . . 51
B.2 Versions Model . . . 52
B.2.1 Schema Fields . . . 52
B.2.2 File Storage . . . 52
B.3 Tags Model . . . 53

B.3.1 Schema Fields . . . 53
B.4 Categories Model . . . 53
B.4.1 Schema Fields . . . 53
B.5 Groups Model . . . 53
B.5.1 Schema Fields . . . 53
B.6 Database Relationships Overview . . . 53

C API Reference Documentation **54**
C.1 Authentication . . . 54
C.1.1 Login . . . 54
C.1.2 Logout . . . 54
C.2 Project Management (JSON API) . . . 55
C.2.1 Retrieve Projects . . . 55
C.2.2 Create Project . . . 55
C.2.3 Update Project . . . 56
C.2.4 Delete Project . . . 57
C.2.5 Version-Specific Operations . . . 57
C.3 Version Control . . . 58
C.3.1 Retrieve Version . . . 58
C.3.2 Create Version . . . 58
C.3.3 Update Version . . . 59
C.3.4 Delete Version . . . 59
C.3.5 Download Version Files . . . 59
C.4 Category Management . . . 60
C.4.1 Retrieve Categories . . . 60
C.4.2 Create Category . . . 60
C.4.3 Delete Category . . . 61
C.5 Tag Management . . . 61
C.5.1 Retrieve Tags . . . 61
C.5.2 Create Tag . . . 62
C.5.3 Update Tag . . . 62
C.5.4 Delete Tag . . . 62
C.6 Project Gallery (Web Interface) . . . 63
C.6.1 Gallery Views . . . 63
C.6.2 Project Detail Views . . . 64
C.7 Security and Authorization . . . 64
C.7.1 Authentication Requirements . . . 64
C.7.2 Authorization Levels . . . 65
C.7.3 Input Validation and Security . . . 65

C.8 Error Handling 65
C.8.1 HTTP Status Codes 65
C.8.2 Error Response Format 65
C.8.3 File System Error Handling 66
C.9 API Usage Examples 66
C.9.1 Authentication Flow 66
C.9.2 Project Creation Example 66
C.9.3 Query Examples 66

D HTML Sanitization and Allowed Tags **67**
D.1 Text Formatting 67
D.2 Structure and Lists 67
D.3 Links 68

List of Abbreviations

GroIMP	Growth Grammar-related Interactive Modelling Platform	2
API	Application Programming Interface	6
DBMS	Database Management System	8
ER model	Entity-Relationship model	8
IDE	Integrated Development Environment	10
REST	Representational State Transfer	11
JWT	JSON Web Token	16

Chapter 1

Introduction

1.1 Background and Motivation

The open-source Growth Grammar-related Interactive Modelling Platform (GroIMP) is a comprehensive software framework for functional-structural plant modelling that combines rule-based modelling approaches with advanced 3D visualization capabilities. First developed at the BTU Cottbus and currently under continued development at the University of Göttingen, GroIMP enables researchers and practitioners to create complex models of plant growth and architecture using XL programming language, a Java-based extension of the L-system formalism [1]. The platform has been widely adopted in the plant modelling community for applications ranging from crop simulation to forest ecosystem modelling, e.g. for the modelling of Norway spruce trees [2] and for the development of new modelling techniques and frameworks like FSPM-P [3].

The Growth Grammar website (grogra.de) serves as the central hub for the GroIMP community, providing documentation, tutorials, and a project gallery where users can share their modelling work. The project gallery serves a particularly crucial role in community development, offering concrete examples of functional models that demonstrate best practices and implementation strategies. These examples possess significant didactic value, providing new users with tangible starting points to understand complex modelling concepts and techniques. For newcomers to functional-structural plant modelling, such examples are invaluable in bridging the gap between theoretical documentation and practical application, thereby facilitating community growth and knowledge transfer.

The GroIMP community encompasses two distinct user groups with different needs and technical expertise levels. Model developers are typically experienced programmers and modelers who create new models from scratch or extend existing frameworks. In contrast, model users are often domain experts—such as biologists, agronomists, or ecologists—who primarily seek to apply existing models to their research questions without necessarily developing deep programming

expertise. These model users frequently require only specific configurations of established models or need to adjust particular parameters for their use cases. However, the current infrastructure inadequately serves this user group, as accessing and utilizing existing models often requires extensive communication with developers or learning complex software functionalities that extend far beyond their actual needs.

Currently, the project sharing mechanism represents a significant bottleneck in community engagement and accessibility. To publish a project on the existing gallery, users must fill out an email form and wait for manual processing by the website maintainer. Downloaded files are stored directly within the server's file system, making both modification and search operations cumbersome and error-prone. This outdated workflow not only inhibits the collaborative potential of the GroIMP community but also creates substantial barriers for model users who need straightforward access to existing models. The lack of searchable metadata and organized categorization particularly disadvantages newcomers and non-developer users who cannot easily discover relevant models or assess their applicability without extensive investigation.

Both the software platform and its accompanying website infrastructure are undergoing major modernization and refactoring work. This includes a new modernized website designed with WordPress, a comprehensive Wiki for GroIMP users, refactoring of GroIMP's legacy Java codebase to support the latest Java versions and new features such as optimized support for point clouds with new point cloud manipulation tools [4].

1.2 Related Work

The concept of centralized repositories for sharing academic and research resources is well-established across various domains. Most notably, GitHub [5] has revolutionized code sharing and collaboration in software development through sophisticated version control, search capabilities, and community features. In the scientific domain, Zenodo [6] provides researchers with a platform to share datasets, publications, and software with comprehensive metadata support and persistent identifiers.

Within the specialized field of biological and agricultural modelling, the BioModels Database [7] serves as a prominent example of a successful model repository, offering searchable access to computational models in systems biology with standardized formats and extensive metadata.

These existing platforms demonstrate common features essential for community adoption: intuitive web interfaces, robust search and filtering capabilities, version control, and standardized metadata schemas. However, none specifically address the unique requirements of functional-structural plant models created with rule-based approaches like those used in GroIMP.

1.3 Objectives

The primary objective of this work is to implement a modern, database-driven architecture that provides a centralized platform for saving, sharing, and discovering GroIMP modelling projects. This platform aims to serve both user groups effectively by providing intuitive access for model users while maintaining comprehensive functionality for model developers. The specific objectives include:

- Development of a relational database schema to store project metadata, user information, and file associations with rich categorization and search capabilities
- Implementation of a RESTful API to enable programmatic access to the repository. For explanation see REST API Architecture 3.2.1.
- Creation of a user-friendly web interface for browsing, searching, and downloading projects that prioritizes accessibility for non-developer users
- Integration of user authentication and project management capabilities
- Establishment of a scalable file storage system to handle diverse project assets
- Design of metadata structures that facilitate discovery and understanding of models for users with varying technical expertise

The resulting platform should significantly reduce the administrative overhead of project sharing while enhancing discoverability and promoting knowledge exchange within the GroIMP community. Particular emphasis is placed on lowering barriers for model users who seek to apply existing models without requiring extensive technical knowledge or prolonged communication with developers.

1.4 Scope and Limitations

Due to scope limitations, not every decision regarding tools and packages can be discussed in detail. However, key architectural components and design choices deemed particularly significant or non-trivial are explained more thoroughly. In general, technology selections were made based on availability, development simplicity, and adequate feature completeness to support the core objectives of this work.

The implementation focuses primarily on the core repository functionality and does not address advanced features such as collaborative editing, integrated version control, or computational model execution within the platform. Integration with external authentication providers and advanced analytics capabilities are also beyond the current scope.

1.5 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 presents the analysis of existing infrastructure and defines the system requirements for database, API, and web interface components. Chapter 3 details the design decisions, including the selection of technologies and architectural considerations for each system component. Chapter 4 describes the implementation of the server infrastructure, database models, API endpoints, web interface, and legacy data migration process. Chapter 5 concludes with a discussion of the achieved outcomes, lessons learned during development and an outlook for future development goals.

Chapter 2

Analysis

2.1 Existing Infrastructure and Old Data

The website `grogra.de` was originally hosted on a Windows-based virtual machine and generated with *Apache Forrest*, which is no longer supported [8]. During this project's development time, the site was migrated to a more modern environment and is now hosted on a Linux-based XAMPP server stack, integrating an Apache web server, a MySQL database, and PHP to support a WordPress installation with built-in user management [9].

The old gallery contained over 300 different projects. These projects, from here on called *legacy data*, are saved as source code files in `.rgg` file format (pure code) or `.gsz` (an archive-like file-format for a whole project that additionally includes other files and meta data). The projects are structured in a filesystem according to their topic where the topic is the corresponding folder name. In addition to the project's source code file there are two more files per project. First a simple `.desc` text-file that holds a number of different text-based meta information including a description, the author, a list of other models/projects that are related and a date without further description, but assumed to be the date of the creation or upload of the model. For a full list of options in the `.desc` please see Appendix A.1. The second file is an image, usually a snapshot of the model render in one of three image-formats: `.png`, `.gif` or `.jpg`. All files are placed together in the same folder, with the exception being `.gif` files which have a dedicated folder further up in the folder structure.

2.2 Requirements

A database is appropriate for proper data structuring, searching, filtering, ordering and management of the data. The data should be accessible through an Application Programming Interface (API) for both GroIMP as well as the newly to be created, dedicated gallery web page. Both

should have a modern state-of-the-art implementation and allow for simple use, maintenance and new features in the future.

In addition to the existing legacy project structure, it should be possible to save multiple distinct versions of the same project.

2.2.1 Database

The database needs to hold text-based data from the description file as well as references to binary data, in form of images and archive files, or hold the data itself.

2.2.2 API

The API needs to be accessible from a dedicated gallery web view as well as from within GroIMP. Every interaction with the database should happen through the API, this should include functionality for accessing, editing and deleting data and the corresponding authentication and authorisation of users. For access control the MySQL Wordpress [9] user data is to be used.

2.2.3 Web Page

The user should be able to browse and view existing projects through a graphical web interface, this includes searching, filtering and sorting of the data. It should also be possible to add new projects or versions and edit or delete existing ones.

The view should include all the features that where present in the old gallery, this includes showing the image, showing all the different meta information from the description file as well as providing a file download for each of the project's version specific models.

Chapter 3

Design

This chapter outlines the design decisions made for the system, focusing on the selection of technologies and tools for each major component of the architecture. The discussion covers the database structure and choice of Database Management System (DBMS), the design of the backend API and web interface, as well as supporting scripts and testing environments. Emphasis is placed on practicality, maintainability, and integration between components.

3.1 Database

Figure 3.1 shows an Entity-Relationship model (ER model) [10] as a design draft for the database. An Entity-Relationship Model (ER model) uses rectangles to represent entities (real-world objects or concepts), diamonds to represent relationships between entities, and ovals to represent attributes (properties of entities), with key attributes underlined to indicate primary keys that uniquely identify entity instances. Relationships are characterized by multiplicities (1:1, 1:n, or n:m) that define how many instances of one entity can be associated with instances of another entity, specifying the cardinality constraints between related entities.

The model contains only a limited number of entities. Most of its structure is derived directly from the legacy data format, with the most significant addition being the concept of versions. Categories represent the topics under which the projects were classified and correspond to their placement in the folder structure of the original server files (see appendix for a list of categories). Tags are a newly introduced feature, allowing users to add searchable keywords as a form of informal categorization, without expanding the fixed category set maintained by the admin group. Users and groups are intended to enable access control, although group support was ultimately considered an optional extension for a later development stage and user data will be fetched from a pre-existing system.

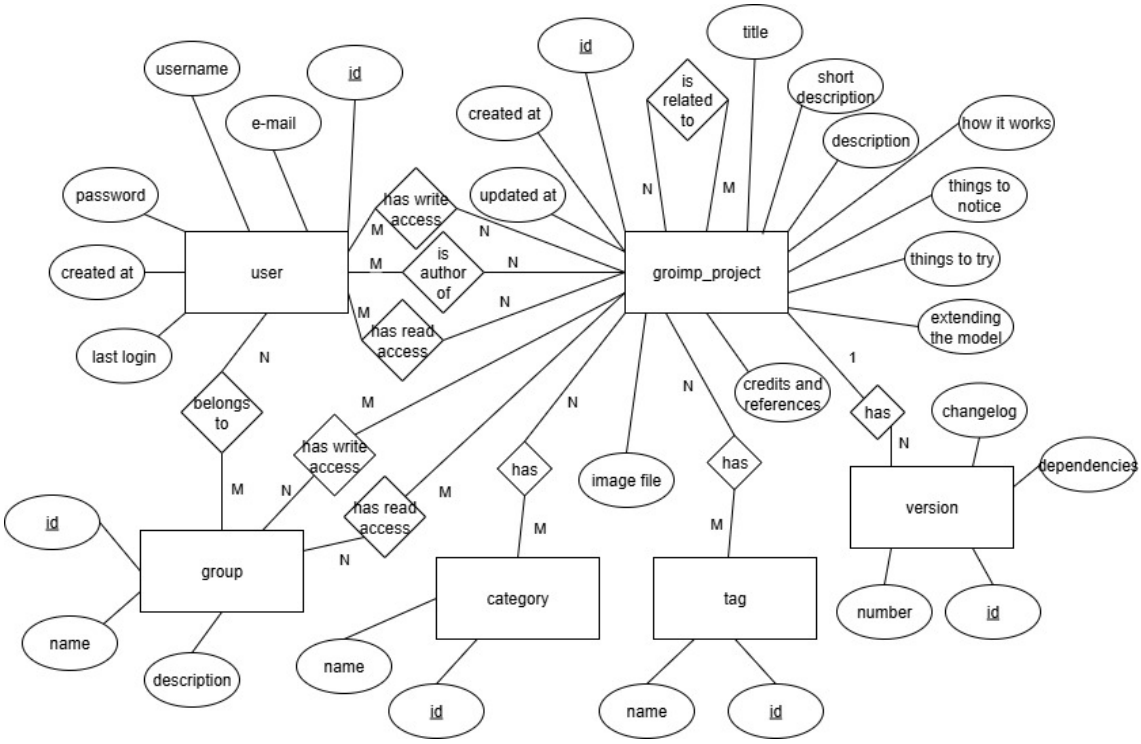


Figure 3.1: Entity-Relationship model for the database

For implementation, MongoDB and MySQL are two highly popular open-source choices, especially for smaller, self-hosted DBMSs. Both systems are viable, offering extensive documentation and good scalability. MySQL is a conventional SQL-based relational DBMS that performs well with highly interconnected data. [11] In contrast, MongoDB excels with large, loosely connected datasets. While performance is a key advantage, MongoDB also provides several other benefits: it integrates easily with JavaScript (see next section), stores data in JSON format, and simplifies data fetching and updating. Although it does not follow a relational model, many relational concepts can still be enforced in MongoDB, albeit with potential efficiency trade-offs, which are acceptable given the expected volume and size of the data. [12]

In summary, MongoDB is selected for its simplicity and seamless integration with the JavaScript-based components of the system for the gallery database. While no separate MySQL server will be set up during this project, MySQL will play a role in user authentication by leveraging the existing WordPress database infrastructure to avoid separate user management systems and provide users with a unified authentication experience across GroIMP.

3.2 API and Web Page

The system architecture divides functionality between two main components: a backend API and a frontend web interface. The API is designed to handle all data operations including project storage, retrieval, user authentication, and file management through standardized endpoints. The web interface provides browsing capabilities, search functionality, project upload forms, and administrative tools for content management.

Communication between these components follows a request-response cycle where user interactions trigger HTTP requests that pass through validation and processing layers before generating appropriate responses. This separation enables independent development of each component while maintaining clear interfaces and allows for future integration with external clients such as the GroIMP software itself.

To implement this architecture efficiently, it is advantageous to use a single programming language for both the frontend and backend. JavaScript is the most prominent candidate for this purpose.

JavaScript offers a wide array of popular tools, libraries, and frameworks that support full-stack development, while remaining relatively easy to learn [13]. Although alternatives such as TypeScript—a statically typed superset of JavaScript with an expanding user base—offer more robust error checking and additional features, they also come with a steeper learning curve [14]. For this reason, and to avoid unnecessary complexity in a first-time project of this size, plain JavaScript was chosen.

The de facto standard Integrated Development Environment (IDE) for JavaScript development is Visual Studio Code, which provides a lightweight yet powerful environment with excellent support for JavaScript [15].

3.2.1 API

The following sections outline the key architectural decisions and technology selections for implementing the backend API infrastructure.

Node.js Runtime Environment

Node.js was chosen as the runtime environment due to its widespread industry adoption and strong ecosystem for developing scalable web applications. As a well-established platform for executing JavaScript on the server, it offers high performance, extensive library support, and compatibility with modern development practices [16]. Given that JavaScript was already selected as the main language, Node.js is a natural choice for the backend.

REST API Architecture

A RESTful API architecture is planned to structure the communication between frontend and backend. Representational State Transfer (REST) is a widely adopted architectural style for web services that relies on standard HTTP methods such as GET, POST, PUT, DELETE, and PATCH. In this approach, each endpoint corresponds to a resource—such as a user, dataset, or document—which can be accessed or modified using these methods. The HTTP methods follow semantic conventions where GET requests handle the retrieval of data without modifying server state, POST requests handle the creation of new resources, PUT requests handle the replacement of existing resources in their entirety, PATCH requests handle partial updates to existing resources, and DELETE requests handle the removal of resources from the server. This clear and predictable interface simplifies client-server communication, improves modularity, and allows for independent development of system components [17].

Express Backend Framework

The Express framework is considered as the backend framework because of its minimal design, low setup overhead, and flexibility in defining custom routing and middleware. This framework is well-suited to building RESTful services and is expected to contribute to a clean and maintainable backend structure [18].

3.2.2 Web Page

For server-side rendering of dynamic web content, the EJS templating engine is used. Although not inherently part of Express, EJS integrates seamlessly via Express's view engine interface, enabling HTML pages to be generated with embedded JavaScript logic. This approach supports consistent rendering and simplifies template management within the JavaScript-based stack. Naturally, these templates use standard HTML and CSS to structure and style the web pages, ensuring compatibility with all modern browsers and ease of design. [19]

3.3 File Management

Uploaded files are stored on the server filesystem using the `FILE_LOCATION` environment variable defined in the `.env` configuration file, which specifies the root directory for the project file structure containing all uploaded project assets and associated image files.

The root directory contains individual project folders, each named according to its unique project identifier. Within each project folder, you will find the associated image files alongside version subdirectories, which are named using their respective version identifiers. Each version subdirectory houses the corresponding project files. In Illustration of this pattern can be seen below:

```
FILE_LOCATION/
└── ⟨project_id⟩/
    ├── ⟨imagefile⟩
    └── ⟨version_id⟩/
        └── ⟨projectfile⟩
```

The following structure demonstrates this organization with two example projects: one containing a single version and another containing two versions:

```
FILE_LOCATION/
├── 507f1f77bcf86cd799439011/
│   ├── Image1.jpg
│   └── 65f8a2b3c4d5e6f7a8b9c0d1/
│       └── project1.rgg
└── 507f191e810c19729de860ea/
    ├── Image2.gif
    ├── 65f8a2b3c4d5e6f7a8b9c0d2/
    │   └── project2.rgg
    └── 65f8a2b3c4d5e6f7a8b9c0d3/
        └── project3.gsz
```

3.4 Supplementary Scripting and Testing Tools

Some preprocessing and scripting was required to convert the legacy data into the desired format and to automate uploading it to the new API. For this task, simple Python scripts were used, which play only a minor role in the overall system but are nevertheless part of the process.

Additionally, a local test environment simulating the XAMPP setup with WordPress was created. This environment supports testing all frontend features, particularly those related to compatibility and user access control, which is handled through a separate MySQL-based DBMS.

Chapter 4

Implementation

This chapter details the practical implementation of the GroIMP gallery system, covering the development of each major component designed in Chapter 3. The implementation follows a modular approach, with distinct sections addressing server infrastructure, database integration, API development, web interface creation, and legacy data migration. Each section explains the key implementation decisions, code structure, and technical challenges encountered during development.

All the relevant files can be found in the `gallery_db_server` folder, which serves as the project's root directory and is publicly available in the GitLab repository at `https://gitlab.gwdg.de/davidjulian.forys/groimp_gallery`.

4.1 Server Overview

This section provides a brief overview of the server architecture and core functionality. Detailed implementation specifics for individual features are covered in later sections.

4.1.1 Server Initialization and Database Setup

The application can be started using two different commands depending on the development context:

- `run start` launches the production server using Node.js
- `run devStart` runs the server with nodemon for development, enabling automatic server restarts when saving server file changes to improve the debugging process

At startup, the server establishes pooled connections to both MySQL for user authentication and MongoDB for the actual data handling. This is handled in the top-level `server.js` file. MongoDB is accessed via Mongoose [20] with a configured `maxPoolSize`, while MySQL uses the

`mysql2/promise` [21] library to create a connection pool with defined concurrency limits. Both connection pools are attached to `app.locals` so that they can be reused across route handlers without repeatedly opening new connections. To ensure system reliability, a periodic health check is executed every five minutes to verify that both databases remain reachable and operational.

Once started, the application listens on port 3000, providing both the API and the web interface through a single, integrated access point at `http://localhost:3000/`. This server is designed for local access and requires proxy forwarding from a hosting server to be publicly accessible. Currently, this is implemented at `gallery.grogra.de`, where the application is made available to the public through the `grogra.de` hosting infrastructure.

4.1.2 Middleware Stack and Security

The server's middleware stack ensures correct and secure processing of requests. Body parsing is provided by `express.json()` and `express.urlencoded()` to support both JSON and form-encoded request bodies. Cookie handling is implemented via `cookie-parser` to manage authentication state and other persistent session data.

Input sanitization is performed using `sanitize-html`, configured to allow only a restricted subset of HTML tags and attributes. This prevents cross-site scripting (XSS) when displaying user-generated content. Static asset delivery is handled by `express.static()`, which serves files such as CSS and images from the `/static` directory.

4.1.3 Routing Architecture

Routing follows a modular structure, with each API resource or feature implemented as an independent router in the `routes/` directory. The main routes include:

- `project_json` for creation, retrieval, updating, and deletion of project metadata and associated files
- `project_gallery` for serving the HTML gallery interface
- `versions` for managing version metadata
- `categories` and `tags` for providing categorization and tagging functionality
- `login` and `logout` for implementing authentication endpoints that allow API clients and the web interface to obtain and invalidate session tokens

The root route (`/`) redirects users to `/project_gallery`, which acts as the main entry point for the public-facing web interface. All API endpoints return JSON-encoded responses, whereas web routes render EJS templates with sanitized dynamic content.

4.2 Database

The relevant server files for the database can be found in the sub folder `db_models`. All the files correspond to their respective collections inside the database and have an id attribute that has unique ids:

`categories.js` This includes a descriptive name.

`groimp_projects.js` This includes multiple String based metadata fields as well as well as references (IDs) to versions, related projects, categories and tags as well as an access field that for now only dictates if it is accessible by non admin/uploader users. This will handle group access in the future.

`groups.js` Not implemented yet, only a placeholder file.

`tags.js` This includes a descriptive name.

`versions.js` This includes a version number, a description and a list of dependencies.

The schema definitions always start by importing the Mongoose library which provides a simple way to define a schema. See Appendix B for a full list of the fields of all the database models.

4.3 Middleware

Middleware in Express.js refers to functions that execute during the request-response cycle, having access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. These functions can execute code, make changes to the request and response objects, end the request-response cycle, or call the next middleware function in the stack. In this implementation, middleware functions serve critical roles in authentication, authorization, input validation, logging, and data processing.

To illustrate the middleware execution pattern, consider a typical POST request to create a new project version at `/version/:project_id`. The request traverses through a defined middleware stack where each component performs specific operations before passing control to the next:

```
1 router.post('/:project_id', jwtCheck, uploaderCheck,
  ↪ uploadFiles.single('gszfile'), sanitizeFile, sanitizeInput,
  ↪ logRequestResponse, async (req, res) => {
2     // Route handler logic
3 });
```

The execution flow proceeds sequentially: `jwtCheck` validates the user's authentication token and sets `req.isAuthenticated` and user information; `uploaderCheck` verifies that the authenticated user has permission to modify the specified project; `uploadFiles.single('gszfile')` processes the file upload and attaches it to `req.file`; `sanitizeFile` sanitizes the uploaded

filename; `sanitizeInput` processes and sanitizes all request body parameters; and finally `logRequestResponse` logs the request details before the route handler executes. Each middleware calls `next()` to advance to the subsequent component, or terminates the request early by sending an error response if validation fails.

The middleware components are organized in the `middleware` directory and consist of seven distinct modules: authentication checking, model retrieval, JSON Web Token (JWT) verification, request-response logging, version creation processing, input sanitization, and uploader authorization checking.

The following sections detail their functionality with accompanying code examples.

4.3.1 Authentication and Authorization Middleware

The `jwtCheck.js` middleware handles user authentication by verifying JWTs stored in HTTP cookies. This middleware extracts the token from the request, validates it against the server's secret key, and enriches the request object with user information. The core authentication logic uses Node.js's `jwt.verify()` method to validate the token:

```
jwt.verify(token, process.env.SECRET_KEY, async (err, decoded) => {
if (err) {
req.isAuthenticated = false;
req.isAdmin = false;
return next();
}
req.tokenPayload = decoded;
req.userId = decoded.userId;
req.isAuthenticated = true;
// Additional database queries for admin privileges...
});
```

When token verification succeeds, the middleware extracts the user information from the decoded payload and sets authentication flags on the request object. The `req.isAuthenticated` flag indicates successful authentication, while `req.userId` provides access to the user's unique identifier for subsequent operations. In cases where verification fails, the middleware gracefully handles the error by setting both authentication and admin flags to false, allowing the request to continue with limited privileges. The middleware then queries the MySQL database to determine user capabilities, particularly checking for administrator privileges by deserializing PHP-serialized capability data from the WordPress user metadata. The `uploaderCheck.js` middleware provides fine-grained authorization control by ensuring that only the original uploader of a project or system administrators can modify project data. This middleware retrieves the project from the database and compares the authenticated user's username with the project's `uploadedBy` field, implementing a role-based access control system that prevents unauthorized modifications while

maintaining administrative oversight capabilities.

4.3.2 Input Validation and Sanitization

The `sanitize.js` middleware implements comprehensive input sanitization to prevent cross-site scripting (XSS) attacks and ensure data integrity. This middleware processes all incoming data through the `express-validator` library [22], sanitizing HTML content while preserving safe formatting tags such as bold, italic, and paragraph elements. The middleware handles both form data and file uploads, sanitizing filenames to prevent directory traversal attacks.

The sanitization process converts newline characters to HTML break tags and restricts allowed HTML tags to a predefined whitelist, ensuring that malicious script injection is prevented while maintaining basic text formatting capabilities. See Appendix D for a comprehensive list.

4.3.3 Data Processing Middleware

The `processVersionCreation.js` middleware encapsulates the complex logic required for creating new project versions. This middleware handles file system operations, database record creation, and directory structure management. The implementation follows a structured approach where request data is first extracted and parsed from the JSON payload:

```
1  const processVersionCreation = async (req, res, next) => {
2  const { project_id, version, dependencies, description} =
   ↪  JSON.parse(req.body.data);
3  const newVersion = await Version.create({
4  dependencies, version, description,
5  });
6  const versionFolder = path.join(projectFolder, newVersion._id.toString());
7  await fsExtra.ensureDir(versionFolder);
8  // File movement and cleanup operations...
9  res.locals.createdObjectId = newVersion._id;
10 next();
11 };
```

After parsing the incoming data, the middleware creates a new version record in the MongoDB database using the `Version.create()` method. The newly created version object provides a unique identifier that is used to establish a corresponding directory structure in the file system. The `path.join()` operation constructs the full path to the version-specific folder, while `fsExtra.ensureDir()` guarantees that the directory hierarchy exists before any file operations occur. Finally, the middleware stores the created object's ID in `res.locals` for use by subsequent middleware in the chain and calls `next()` to continue processing. This middleware demonstrates the separation of concerns principle by isolating complex file and database operations into a reusable component that can be called from multiple routes.

4.3.4 Utility Middleware

The `getModel.js` middleware provides a standardized mechanism for retrieving project models from the database based on URL parameters. This middleware reduces code duplication across routes by centralizing the model retrieval logic and error handling, attaching the retrieved model to the response object for subsequent middleware and route handlers.

The `logRequestResponse.js` middleware implements comprehensive request and response logging, capturing HTTP method, URL, query parameters, headers, response status codes, and request duration. This middleware is essential for debugging, performance monitoring, and security auditing, providing detailed insights into API usage patterns and potential issues.

The middleware architecture ensures that cross-cutting concerns such as authentication, authorization, validation, and logging are handled consistently across all routes while maintaining code modularity and reusability.

4.4 Routing

The routes are split into different files located in the `routes` folder. This improves code maintainability by organizing related endpoints together, making it easier for future developers to locate and modify specific functionality without navigating through a monolithic route file.

The gallery server implements a RESTful API architecture with multiple route modules, each handling specific resource types. The server follows standard REST conventions where route names correspond to the resources they manage, and HTTP methods indicate the type of operation being performed.

4.4.1 Route Organization

The application organizes routes into logical modules based on resource types:

- `/project_json` - Provides JSON API endpoints for project data manipulation
- `/project_gallery` - Serves the web-based gallery interface for projects
- `/version` - Manages project version control and history
- `/category` - Handles project categorization endpoints
- `/tags` - Manages project tagging system
- `/login` - Authentication endpoint for user login
- `/logout` - Authentication endpoint for user logout

4.4.2 Project Data Access

The server provides two distinct approaches to accessing project data:

Project JSON Routes (`/project_json`) deliver raw project data in JSON format, designed for programmatic access and API consumption. These endpoints follow the standard REST patterns.

Project Gallery Routes (`/project_gallery`) serve rendered HTML pages for human interaction, providing a visual interface for browsing and managing projects. This route handles the web-based user interface and will be examined in detail in the following *Web Page* section.

4.5 API Workflow Example

This section demonstrates a complete API workflow, showing how to authenticate, create categories and projects, manage versions, and establish relationships between models. The example walks through creating two projects with different versions, illustrating a full exemplary cycle from user login to project creation and management. For a comprehensive API documentation see Appendix C. All examples use `localhost:3000` as the domain, but as already stated this can be replaced with `gallery.grogra.de` where the API is currently hosted.

4.5.1 User Authentication

The authentication process requires sending user credentials via a POST request to establish a session token that will be automatically included in subsequent requests.

```
# Login with credentials
POST http://localhost:3000/login
Content-Type: application/x-www-form-urlencoded

username=researcher&password=your_password

# Response sets HTTP-only authentication cookie automatically
# Response: {"message": "Login successful"}
```

4.5.2 Create Categories

Before uploading projects, categories must be created to enable proper project classification and organization within the gallery system.

```
# Create first category
POST http://localhost:3000/category
Content-Type: application/json
# Authentication cookie included automatically

```

```
{
  "name": "Simulation"
}

# Create second category
POST http://localhost:3000/category
Content-Type: application/json

{
  "name": "Plants"
}
```

4.5.3 Fetch All Categories

This request retrieves all available categories from the system, providing both their database IDs and display names for use in project creation.

```
# Retrieve all available categories
GET http://localhost:3000/category

# Response includes ObjectIds and names:
# [
#   {"_id": "64a7b8c9d1e2f3g4h5i6j7k8", "name": "Simulation"},
#   {"_id": "507f1f77bcf86cd799439011", "name": "Plants"}
# ]
```

4.5.4 Create New Project

The project creation request uses multipart form data to handle both metadata and file uploads. The request is structured into distinct parts: `projectdata` contains the main project metadata including title, description, categories, and access settings, while `versiondata` holds version-specific information such as version number, dependencies, and version description. Additionally, two files are uploaded: `imagefile` for the project preview image and `gszfile` containing the GroIMP model file.

Note: Categories are provided as **names** (not IDs) and are automatically converted to their respective ObjectIds internally.

This example demonstrates creating a complete project entry with both metadata and file uploads using multipart form data encoding.

```
# Create project with multipart form data
POST http://localhost:3000/project_json
Content-Type: multipart/form-data
```

```
# Authentication cookie included automatically

# Form fields:
projectdata: {
  "title": "Tree Growth Simulator",
  "shortdescription": "Models realistic tree development",
  "description": "Detailed project description here",
  "categories": ["Simulation", "Plants"],
  "howitworks": "Implementation details",
  "access": {"public": true}
}

versiondata: {
  "version": "1.0.0",
  "dependencies": ["plugin1"],
  "description": "Initial version release"
}

# File uploads:
imagefile: [binary file data - project preview image]
gszfile: [binary file data - GroIMP model file]

# Response includes both project and version IDs:
# {
#   "newProject": {"_id": "64a7b8c9d1e2f3g4h5i6j7k8", ...},
#   "newVersion": {"_id": "507f1f77bcf86cd799439011"}
# }
```

4.5.5 Create a Referencing Project

This request creates a second project that establishes a relationship with the previously created project through the `relatedmodels` field.

```
# Create second project that references the first
POST http://localhost:3000/project_json
Content-Type: multipart/form-data

projectdata: {
  "title": "Advanced Tree Growth Model",
  "shortdescription": "Enhanced version with additional features",
  "description": "Extended project description",
  "categories": ["Simulation", "Plants"],
  "relatedmodels": ["64a7b8c9d1e2f3g4h5i6j7k8"], #ID of above project
```

```
  "access": {"public": true}
}

versiondata: {
  "version": "1.0.0",
  "dependencies": ["plugin2"],
  "description": "Initial version with enhanced features"
}

imagefile: [binary file data]
gszfile: [binary file data]
```

Important Note: The `relatedmodels` field creates a **unidirectional reference**. The first project's `relatedmodels` array is **not automatically updated** - a manual update via a PATCH-request is needed if bidirectional references are desired.

4.5.6 Fetch Specific Project

This GET request retrieves detailed information about a specific project, including all associated metadata, version references, and relationship data.

```
# Retrieve the first project with all details
GET http://localhost:3000/project_json/64a7b8c9d1e2f3g4h5i6j7k8

# Response includes populated project data with:
# - All project metadata
# - Array of version ObjectIds
# - Category ObjectIds (resolved from names during creation)
# - Related model references
```

4.5.7 Version Management Operations

The following examples demonstrate adding new versions to existing projects and removing outdated versions from the system.

```
# Add new version to the project
POST http://localhost:3000/version/64a7b8c9d1e2f3g4h5i6j7k8
Content-Type: multipart/form-data

version: 2.0.0
dependencies: plugin3
description: Major update with performance improvements
gszfile: [binary file data - updated model]

```

```
# Response: {"newVersion": {"_id": "new_version_object_id"}}
```

This DELETE request removes a specific version from the project, cleaning up both database records and associated files from the filesystem.

```
# Delete the original version (first version)
DELETE http://localhost:3000/version/507f1f77bcf86cd799439011

# This removes:
# - Version document from database
# - Associated files from filesystem
# - Version ID from project's versions array
```

4.5.8 Session Termination

The logout request terminates the user session by invalidating the authentication cookie and clearing it from the client.

```
# Logout and clear authentication cookie
POST http://localhost:3000/logout

# Response: {"message": "Logout successful"}
# Authentication cookie is cleared (maxAge: 0)
```

4.6 Web Interface

The project gallery web interface represents the primary user-facing component of the application and is currently hosted on `gallery.grogra.de/project_gallery`. It can be also accessed from `grogra.de` via the menu point *Gallery*.

This module provides project browsing, detailed project viewing, and user interaction capabilities through a web browser interface.

Some functionality requires a `grogra.de` user account, namely uploading and editing projects. The rest of the features is available for accountless users as well, including searching, filtering, viewing and downloading projects.

4.6.1 Navigation and Functionality

This section demonstrates the functionality of the new gallery and explains how to navigate its interface.

Main Page

When opening the GroIMP Project Gallery on `gallery.grogra.de`, the user is greeted by its main page as shown in Figure 4.1.

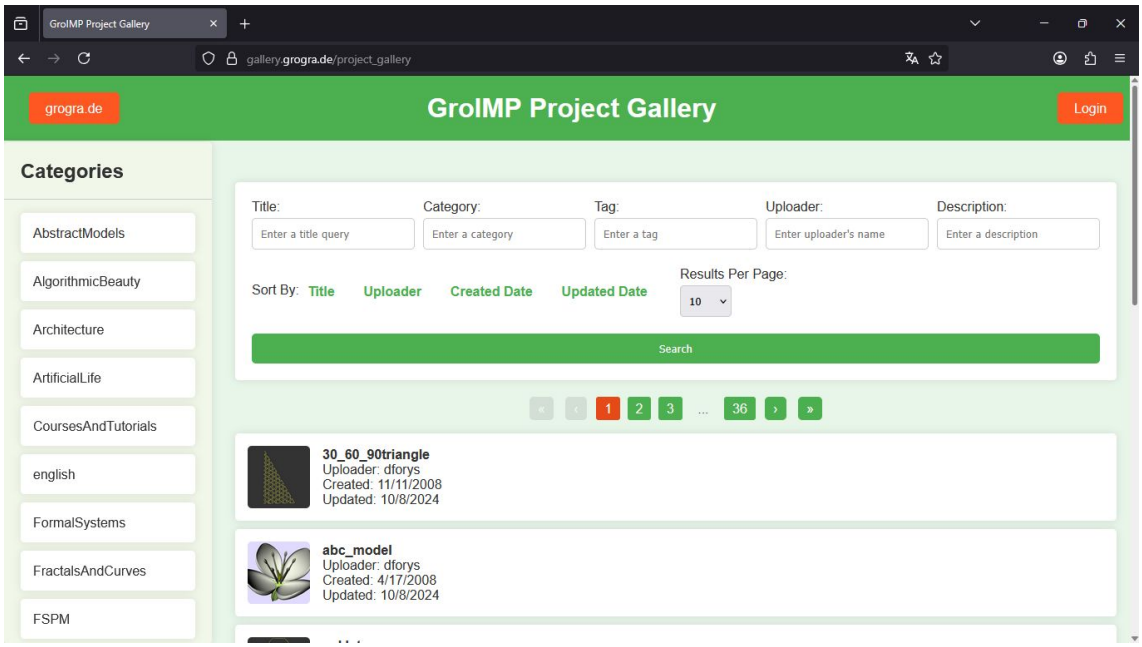


Figure 4.1: The Main Page of the gallery as it is displayed when not logged in.

At the top is a **banner** displaying "GroIMP Project Gallery". Clicking the title leads to a clean version of `gallery.grogra.de` without any enabled filters, sorting options, search terms, and other metadata. To the left of the title is a button that leads back to `grogra.de`. Another button on the right leads to the Login Page. The banner changes depending on the user's authentication status. An authenticated user sees a banner as shown in Figure 4.2 with additional buttons: The *Upload Project* button is displayed next to the *grogra.de* button and leads to the Upload Page. A *Logout* button replaces the *Login* button next to the logged-in user's username. Clicking this button logs the user out, clearing the login cookie and terminating the session on the server side, returning to the main page shown in Figure 4.1.

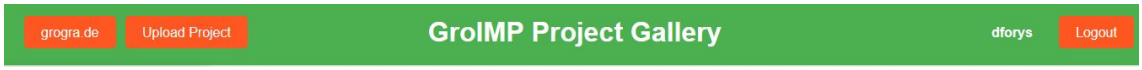


Figure 4.2: The banner has additional buttons after login.

On the left is a **sidebar** titled *Categories*. It scrolls independently of the rest of the website and displays all available categories for projects. Clicking a category reloads the main page with the selected category as a set filter option.

The **search box** below the banner provides options for entering search terms, filtering, and sorting. It is possible to search by title, uploader, and description. While the former two directly correspond to the projects' metadata fields, the description field searches all text-based metadata information fields (see Appendix B for a full list of fields). The search implementation uses MongoDB aggregation pipelines for efficient filtering (see Main Gallery Endpoint for technical details).

Categories and tags have a **suggestions** feature. When typing, the server suggests existing categories and tags (note that there are currently no tags) in a small pop-up as shown in Figure 4.3. Clicking a suggestion auto-completes the user input. Multiple categories and tags can be added separated by commas and receive separate suggestions (see Live Suggestion System for technical implementation details).

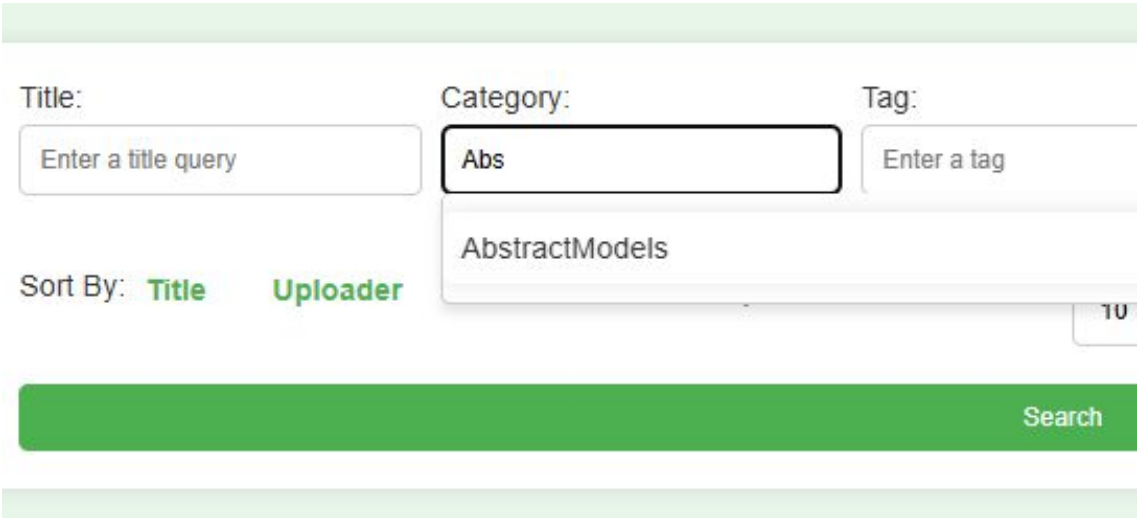


Figure 4.3: Example for a suggestion pop-up for the category "AbstractModels"

It is also possible to **sort** by title, uploader, creation date, and date of last update. By clicking one of the options once, the options are sorted in ascending order indicated by a small triangle pointing up next to the clicked option. Another click reverses the order (descending) indicated by the triangle now pointing down.

The **number of results** per page can be set by choosing an option (10, 20, 50, 100) from the drop-down menu.

The **results** appear below the search box. They can be navigated using page numbers located

above and below the results and arrows. The single arrows are for *next page* (>) and *previous page* (<) while the double arrows are for *first page* («) and *last page* (»). For every result, a title in bold as well as the uploader, creation date, and date of last update are displayed next to the image representing the project. Clicking a result redirects to the corresponding Project Detail page.

Login Page

The login page can be seen in Figure 4.4. The banner is replaced by a *Return to Overview* button in the top left that leads back to the Main Page on click.

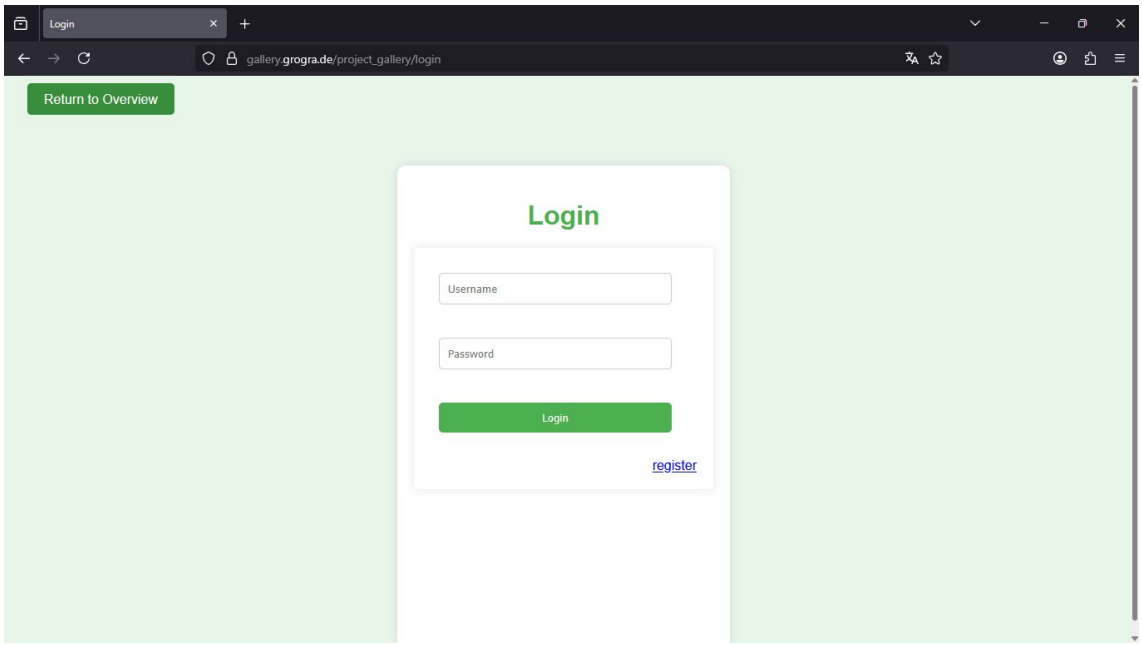


Figure 4.4: On the Login Page the user can log in with his `grogra.de` credentials.

The login interface in the middle of the page titled *Login* has two fields for username and password respectively, a *Login* button to confirm the user input, and a *register* hyperlink that leads to the `grogra.de` page for registering, though it is currently disabled. Instead, for registration, a request needs to be made by contacting the department (see `grogra.de` for contact details). After a successful login, the user is redirected back to the Main Page with added functionality (see Authentication Routes for technical implementation details).

Upload Page

The upload page shown in Figure 4.5 serves a form that allows a logged-in user to upload a new project to the gallery. The banner's title is changed to *Upload New Project* and the button that links

`grogra.de` is replaced by a *Return to Overview* button similar to the login page. The top right still shows the username and *Logout* button similar to the main page banner.

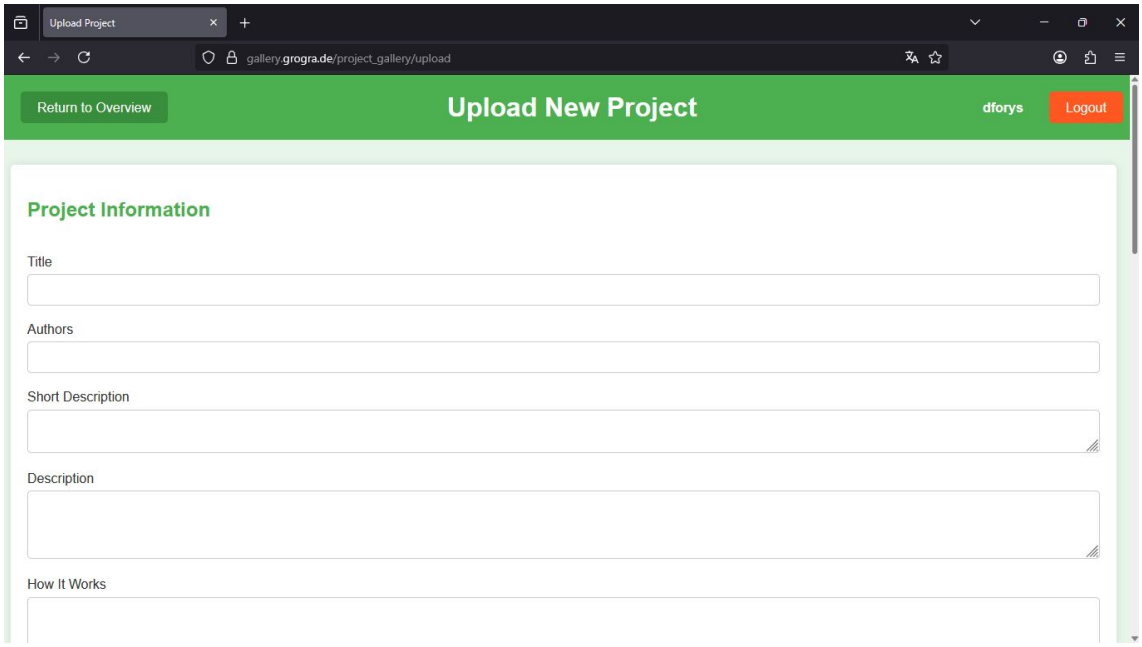


Figure 4.5: On the Upload Page a logged in user can upload his projects.

The form is split into two parts. The first contains all the fields for the general project metadata. The second part contains all the version metadata. Most fields are plain text fields, like *description* that also allow for simple HTML formatting (see Appendix D for allowed HTML tags) to stylize the description. *Categories* and *Tags* have the suggestion system (Section 4.6.1) implemented here as well (see Live Suggestion System for technical implementation details). The *Related Models* field only accepts valid project IDs separated by comma that will then directly link the corresponding projects on the Project Detail Page of the created project. The version field is pre-filled with "1.0" but can still be changed.

The only required fields are the *title* and the two file uploads at the bottom of the form. To upload a file, the user has to click the *Choose File* button; then the standard file selection window for the browser and operating system in use will pop up as shown in Figure 4.6. The project file is expected to be a `.GSZ` or `.RGG` file, while the image file is expected to be `.JPG`, `.PNG`, or `.GIF` format and is set up this way in both file selection windows respectively.

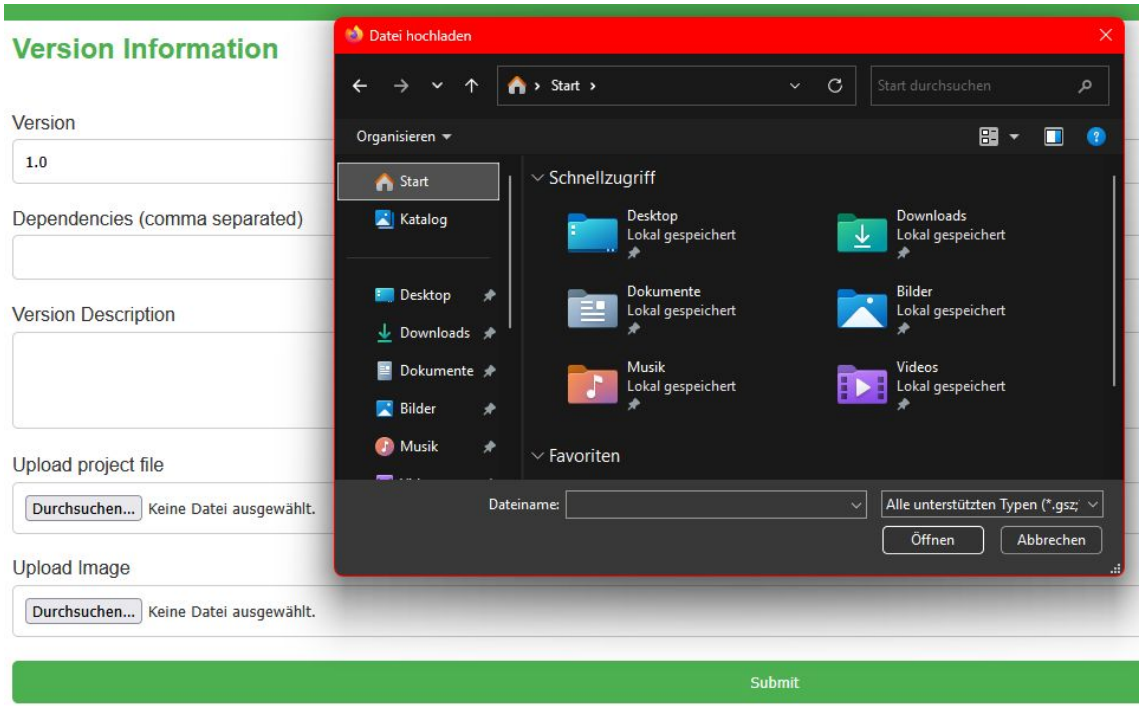


Figure 4.6: Choosing a project file for upload in Mozilla Firefox under Windows 11.

After all required fields are filled and other fields are filled to the user's satisfaction, clicking the *Submit* button sends the form. If the project is created successfully, the user is then redirected to the just created Project Detail Page.

Project Detail Page

When viewing a project in detail, the page looks similar to Figure 4.7; in this example, the project "ant" is shown. The page dynamically loads project metadata and version information through dedicated route handlers (see Project Detail View for implementation details). In the banner, the *Return to Overview* button (linking the Main Page) as well as either username and *Logout* button (logging out and redirecting to the Main Page) or the *Login* button (linking the Login Page) are shown.

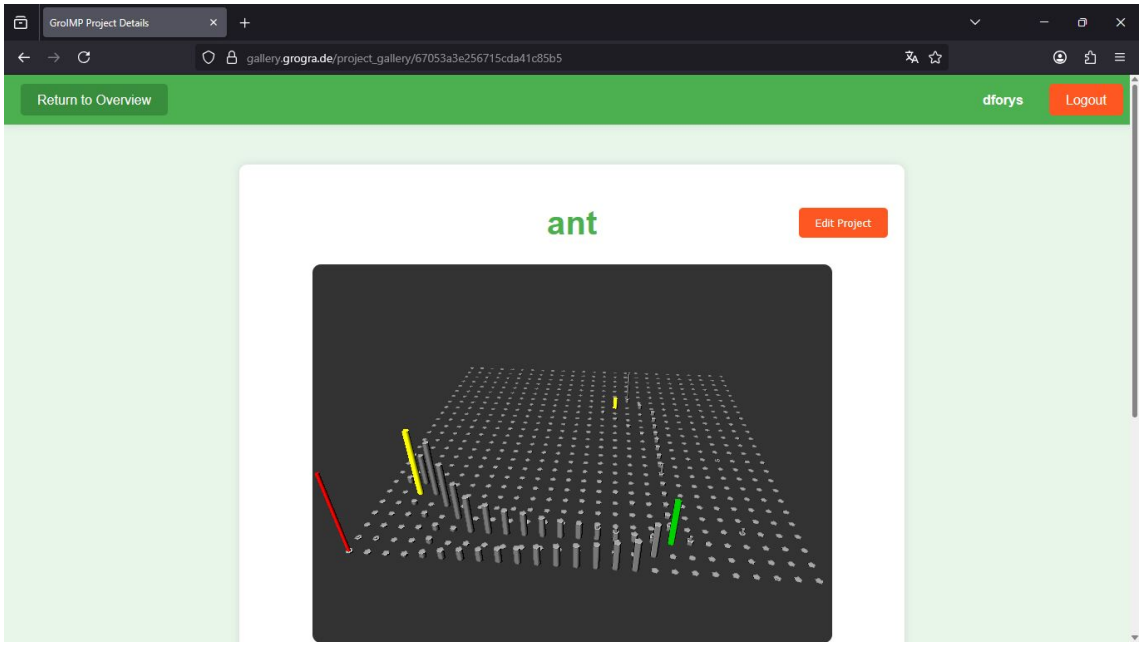


Figure 4.7: The Project Detail page for the project *ant* as an example.

If the user account has the rights to edit the project, an *Edit Project* button is displayed. Clicking the button opens an editing window as shown in Figure 4.8.

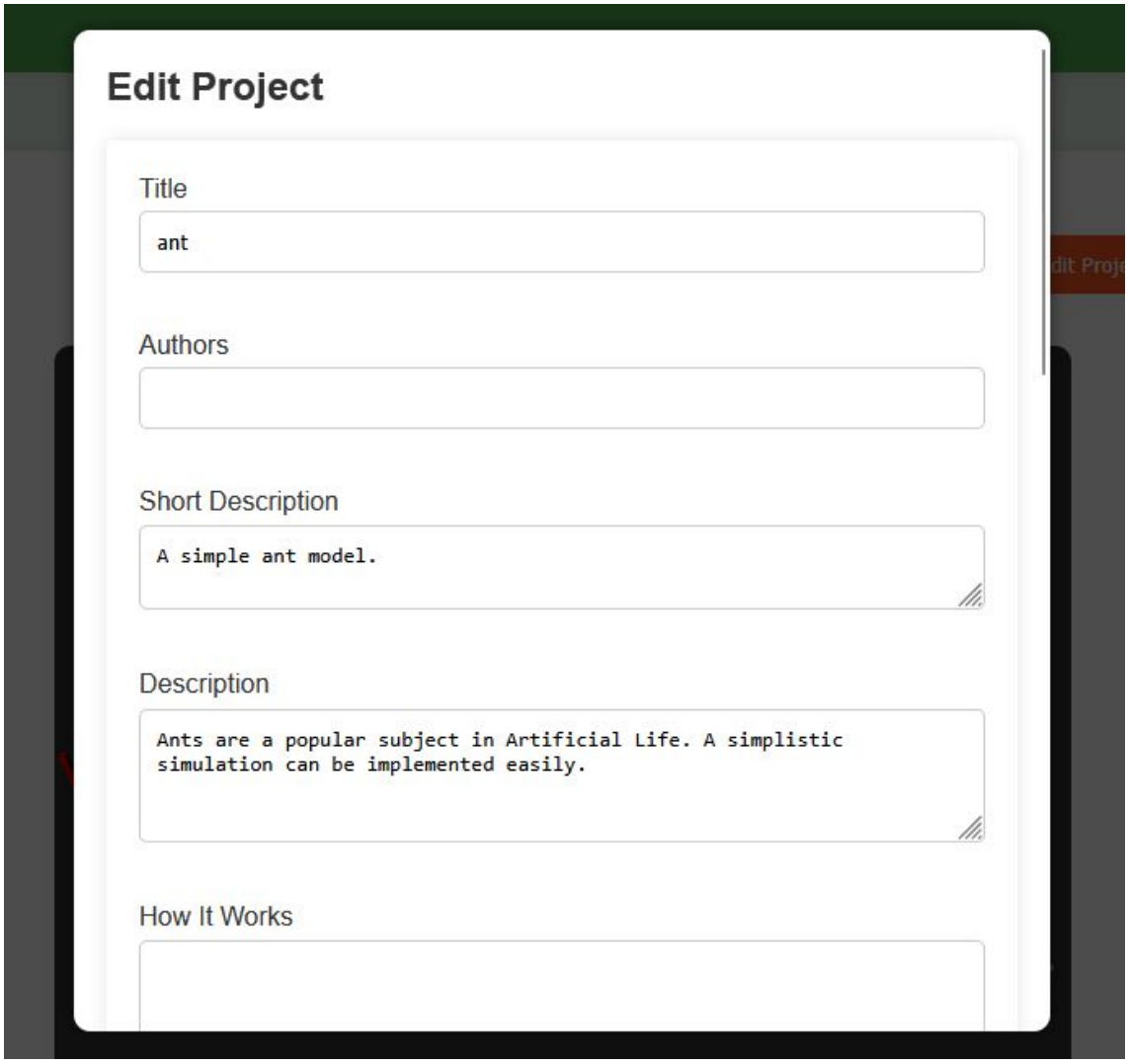


Figure 4.8: Editing window for editing *ant*

Here the project's information can be edited, including the upload of a new image (replacing the old one). At the bottom, below the fields, are three buttons as shown in Figure 4.9.

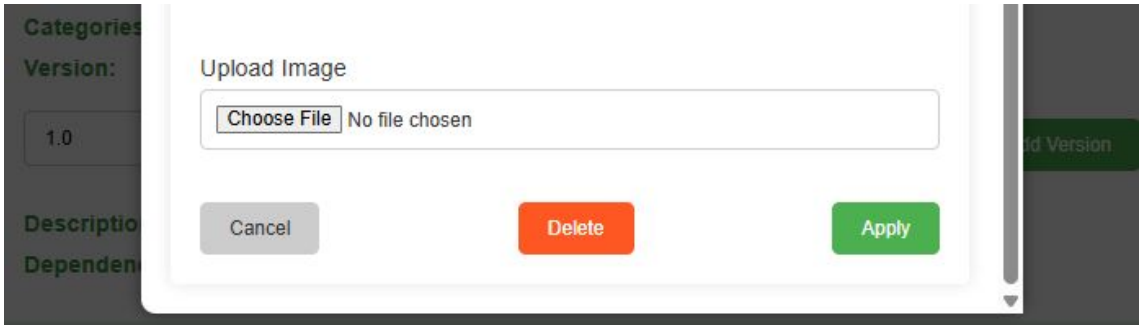


Figure 4.9: The three buttons at the bottom of the project edit form

Cancel cancels any changes, *Delete* deletes the entire project, and *Apply* applies any changes. All three options need confirmation.

Further down on the page, all metadata fields that are in use by the project are displayed as seen in Figure 4.10. In this example, we have the following fields:

ID This field shows the unique project ID that can be referenced in the *Related Models* field of other projects and shared with other users for immediate access to the project by navigating to `gallery.grogra.de/project_gallery/<Project-ID>` (this is the same as the project details website link in the browser's address field).

Uploader This shows the user account that uploaded this project. The name links to the main page with the username set as a search term.

Short Description A short text-based description.

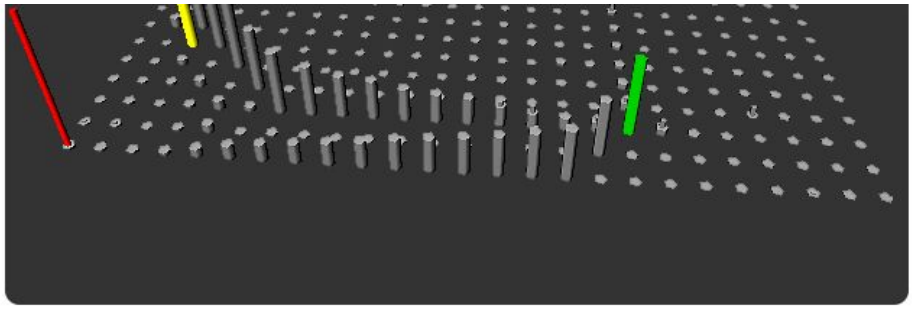
Created At This displays the date when the project was added to the gallery. Note that this project is part of the *Legacy Data*.

Updated At This displays the date of the last update.

Description This is the most general of the many different text-based fields.

Related Models Here are all the projects linked that are somehow related. Clicking a name links to the respective project detail page.

Categories Here is a list of the categories this project is part of. Clicking a category links to the Main Page with the category as an enabled filter.



ID: 67053a3e256715cda41c85b5
Uploader: dforys
Short Description: A simple ant model.
Created At: 4/20/2009, 1:00:00 AM
Updated At: 10/8/2024, 3:57:23 PM
Description: Ants are a popular subject in Artificial Life. A simplistic simulation can be implemented easily.
Related Models: antb, ant_simulation
Categories: ArtificialLife
Version:

Select Version | Download | Edit Version | Add Version

Figure 4.10: The full project information for *ant* is shown below its image.

At the bottom of the page is the **Version** section. Here it is possible to select a version via the drop-down list as shown in Figure 4.11.

Figure 4.11: A version can be selected from the drop-down list

Doing so enables both the *Download* button and, if the user account has the rights to edit this project, also the *Edit Version* button, as shown in Figure 4.12. Additional version information fields are shown for the selected version below the drop-down menu.

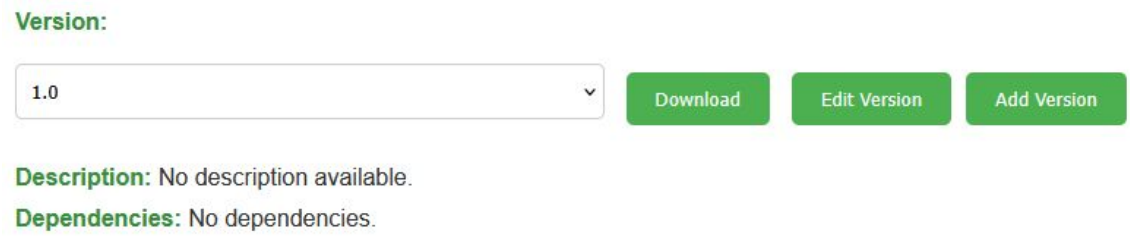


Figure 4.12: Selecting a version enables version information, downloading and editing.

Clicking the *Download* button prompts a download of the project file of the selected version for the project through the browser. If available to the user, the *Edit Version* button opens a window for editing the selected version similar to the project editing window as shown in Figure 4.13.

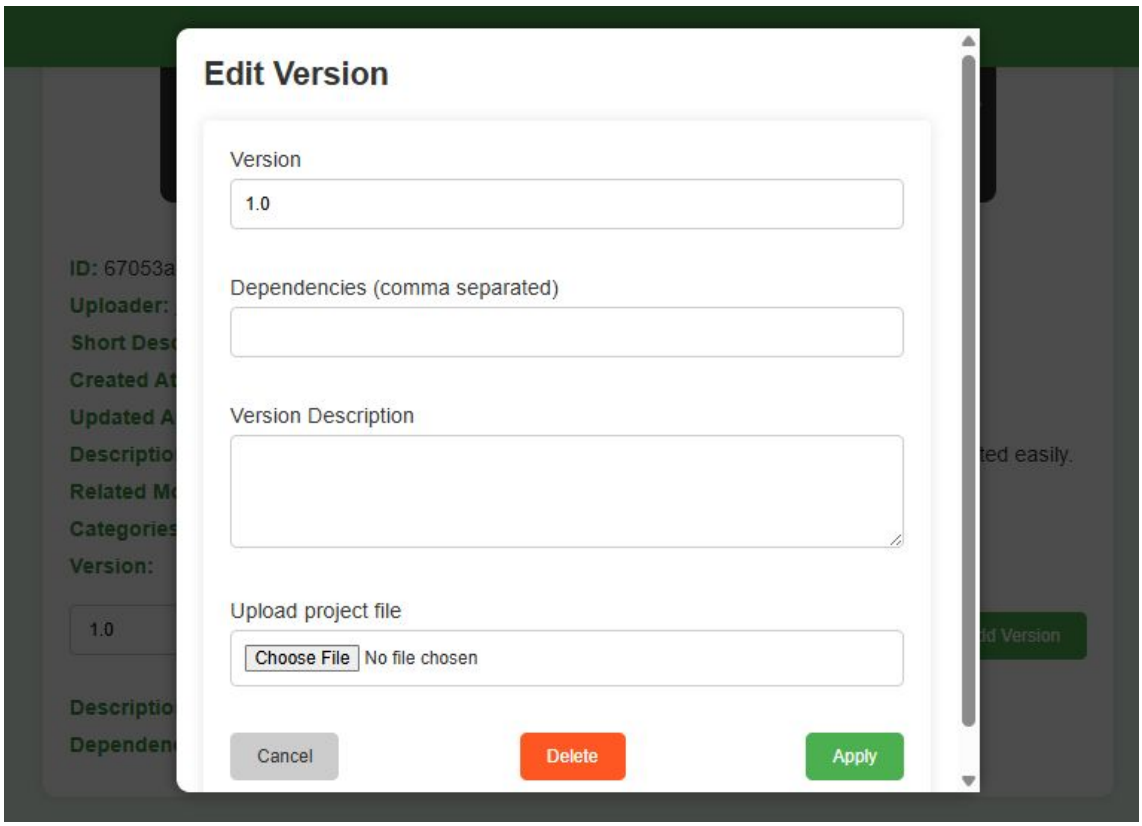


Figure 4.13: The selected version can be modified in the editing window.

Here the version metadata can be changed as well as a project file uploaded to replace the existing project file. This form has the same three buttons for canceling and applying changes as well as deleting the version as the project's edit page has.

Alternatively to editing versions, an authorized user can also add a new version to the project by clicking the *New Version* button. As shown in Figure 4.14, the form for adding versions is very similar to the edit form.

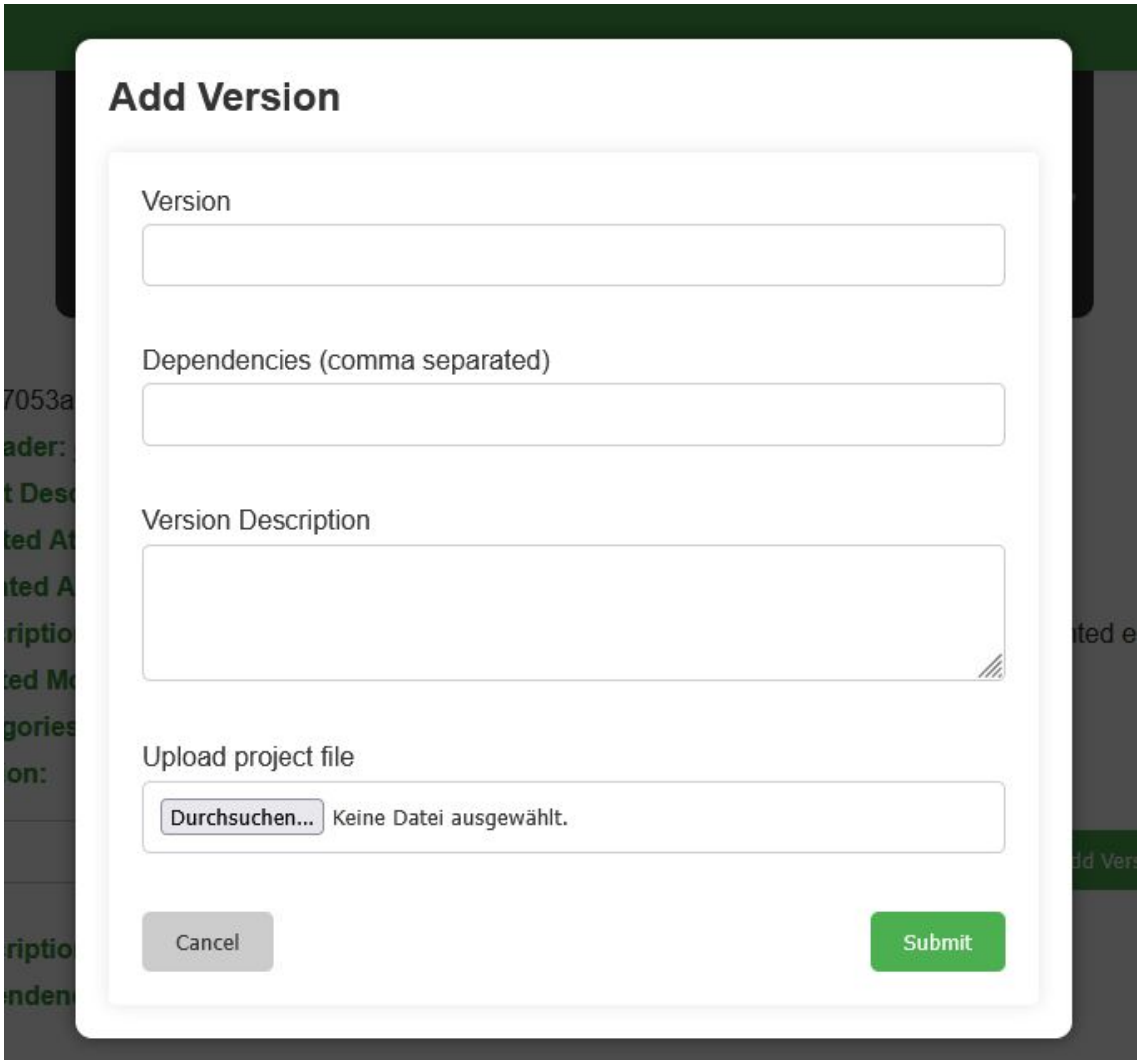


Figure 4.14: New versions can be added to the project in the Add Version window.

After a version number is entered, a project file is uploaded, and any other optional information is added, the form can be submitted via the *Submit* button or the process can be aborted via the

Cancel button (see Template System Implementation for technical implementation details).

4.6.2 Technical Background

The web view is implemented through the `project_gallery.js` route handler and its associated view templates. This router (located in the `routes` folder) serves as the central controller for all web-based project interactions, implementing six primary endpoints that handle different aspects of the gallery functionality.

Main Gallery Endpoint (`GET /`) implements the core project browsing functionality with filtering and pagination capabilities. The route processes multiple query parameters including search terms, category filters, tag filters, and uploader-specific views. For example, a request like `/project_gallery?search=leaf&category=plants&page=2` triggers a MongoDB aggregation pipeline that combines text search across project titles and descriptions with category matching and pagination logic. The implementation constructs dynamic aggregation pipelines to handle complex filtering scenarios:

```
// Building dynamic aggregation pipeline
const pipeline = [];

if (searchTerm) {
    pipeline.push({ $match: { $text: { $search: searchTerm } } });
}

if (categoryFilter && categoryFilter.length > 0) {
    pipeline.push({ $match: { categories: { $in: categoryFilter } } });
}

if (tagFilter && tagFilter.length > 0) {
    pipeline.push({ $match: { tags: { $in: tagFilter } } });
}

pipeline.push({ $skip: (page - 1) * limit });
pipeline.push({ $limit: limit });
```

The search functionality employs MongoDB text indexing for efficient full-text search across project titles and descriptions. The route implements pagination with configurable page sizes and generates navigation metadata for the frontend, including total project counts and page boundaries.

Project Detail View (`GET /:project_id`) provides detailed individual project information retrieval. The route utilizes the `getModel` middleware to validate project existence and retrieve project data, then performs additional aggregation operations to gather related version information:

```
router.get('/:project_id', jwtCheck, sanitizeInput, getModel,
    logRequestResponse, async (req, res) => {
    try {
        const project = res.entry;
        const versions = await Versions.find({
            project_id: project._id
        }).sort({ createdAt: -1 });

        res.render('gallery/show', {
            project,
            versions,
            isAuthenticated: req.isAuthenticated,
            isAdmin: req.isAdmin,
            userId: req.userId,
            tokenPayload: req.tokenPayload
        });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});
```

Upload Interface (`GET /upload`) serves the project submission form with authentication requirements. The route verifies user authentication status through the `jwtCheck` middleware and renders the upload interface with user context information.

Authentication Routes includes login interface serving (`GET /login`) and forbidden access handling (`GET /forbidden`), providing consistent error messaging and user guidance for authentication-related scenarios.

Resource Serving (`GET /:project_id/image`) handles project image delivery for displaying on the web page.

Category Endpoint (`GET /categories`) is a placeholder for a more advanced categories page in the future.

Each route implements the complete middleware pipeline including JWT authentication checking, input sanitization, request logging, and appropriate error handling to ensure secure and reliable operation (see middleware section for more information).

Template System Implementation

The web interface employs four EJS templates (all located in the `views/gallery` folder) that work in conjunction with the route handlers to generate dynamic HTML content.

Gallery Main Page Template (`index.ejs`) implements the main project browsing interface

with client-side functionality. The template generates responsive project listings with conditional rendering based on authentication status:

```
<% if (isAuthenticated) { %>
    <span class="username">Welcome, <%= tokenPayload.username %></span>
    <a href="/logout" class="logout-btn">Logout</a>
<% } else { %>
    <a href="/project_gallery/login" class="login-btn">Login</a>
<% } %>
```

The template implements integrated search forms with real-time suggestions, and provides navigation controls including pagination and filtering options. For instance, when a user clicks on a category filter, the interface constructs URLs like `/project_gallery?category=Plants` and updates the display accordingly.

Project Detail Template (`show.ejs`) renders detailed project information including metadata display, version management interfaces, and conditional edit capabilities. The template implements overlay-based editing interfaces for both project information and version management, with form submission handling through asynchronous JavaScript. Permission-based rendering ensures that edit controls are only displayed to authorized users (project uploaders or administrators).

Upload Interface Template (`upload.ejs`) provides a complete project submission form with integrated file upload capabilities and metadata input fields. The template implements client-side form validation, progress feedback during file uploads, and integration with the live suggestion system (see below) for categories and tags.

Authentication Templates includes the login interface (`login.ejs`) with WordPress data base integration for user login and the forbidden access page (`forbidden.ejs`) for consistent error handling.

Live Suggestion System

The `suggestions.js` (located in the `static` folder) module implements a client-side suggestion system that enhances user experience during data entry. The system provides real-time suggestions for category and tag input fields through integration with backend API endpoints.

The **Core Functionality** of the live suggestions centers around the `attachLiveSuggestions` function, which accepts an input field identifier and a corresponding API endpoint:

```
function attachLiveSuggestions(inputId, suggestionsEndpoint) {
    const input = document.getElementById(inputId);
    const suggestionBox = document.createElement('ul');
    suggestionBox.classList.add('suggestion-box');
    input.parentNode.appendChild(suggestionBox);

```

```
    let debounceTimeout;

    const debounce = (func, delay) => {
        clearTimeout(debounceTimeout);
        debounceTimeout = setTimeout(func, delay);
    };

    input.addEventListener('input', () => {
        const query = input.value.trim();
        const parts = query.split(',');
        const currentPart = parts[parts.length - 1].trim();

        if (currentPart.length > 0) {
            debounce(() => fetchSuggestions(currentPart), 300);
        } else {
            suggestionBox.style.display = 'none';
        }
    });
}
```

The function creates dynamic suggestion interfaces by generating unordered list elements positioned adjacent to the input field. The implementation supports multi-value input through comma-separated parsing. For example, when a user types `"Plants, LeafModels"` in a category field, the system parses the current segment (`"LeafModels"`) and provides suggestions for that term while preserving the previously selected `"Plants"` category.

Performance Optimization is achieved by employing a debouncing mechanism to prevent excessive API calls during rapid user input. The system implements a 300-millisecond delay before triggering another suggestion request, with automatic cancellation of pending requests when new input is detected. This approach ensures responsive user interaction while minimizing server load.

For **API Integration** it utilizes asynchronous fetch requests to retrieve suggestion data from backend endpoints:

```
async function fetchSuggestions(query) {
    try {
        const response = await
        ↪  fetch(`${suggestionsEndpoint}?q=${encodeURIComponent(query)}`);
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const suggestions = await response.json();

        suggestionBox.innerHTML = '';
```

```
10          suggestions.forEach(suggestion => {
11              const listItem = document.createElement('li');
12              listItem.textContent = suggestion;
13              listItem.addEventListener('click', () => {
14                  selectSuggestion(suggestion);
15              });
16              suggestionBox.appendChild(listItem);
17          });
18
19          suggestionBox.style.display = suggestions.length > 0 ? 'block' :
            ↪  'none';
20      } catch (error) {
21          console.error('Error fetching suggestions:', error);
22          suggestionBox.style.display = 'none';
23      }
24  }
```

For instance, typing `"Ar"` triggers a request to `/category/search?q=ar`, which returns suggestions like `Architecture` and `ArtificialLife`. The system handles API responses through JSON parsing and implements error handling to maintain system stability during network issues or API failures.

Finally for **User Interface Integration** it dynamically generates suggestion lists with a click-to-select functionality, allowing users to quickly select from available options. The suggestion interface includes proper styling integration and responsive behavior to maintain consistent user experience.

Static Resource Management

The application serves static resources through Express.js static middleware, with the primary stylesheet (`styles.css`) providing basic styling for interface components. The stylesheet implements responsive design principles and defines consistent visual themes for form elements, navigation components, and interactive elements. Visual design complexity is intentionally minimized to focus on functional requirements, with button styling supporting different contexts including navigation (`.return-btn`), authentication (`.login-btn`, `.logout-btn`), and primary actions (`.upload-btn`, `.grogra-btn`).

4.7 Legacy Data Migration

To migrate the old data to the new gallery, several important refactoring steps were required. First, the `.DESC` files were converted into the `.JSON` format. The Python `json` package [23] provides functionality to convert dictionaries into JSON, so a dictionary was created using keys derived from the different metadata in the pattern file. Some keys were renamed to better reflect their

meaning in the database context, and additional keys were introduced to provide new information, such as access control. The date field was split into two parts, `createdAt` and `updatedAt`, and reformatted to match the MongoDB standard. Furthermore, a clear reference to the image file was added to support personalized naming schemes and enable the possibility of multiple image uploads in the future (see Appendix A.2 for a full example comparison).

For that purpose, the `desc_to_jsonv4_1.py` script was created. It reads the `.DESC` files and the folder structure in which they are placed, identifies the associated images, and constructs the `.JSON` files from the `.DESC` files.

How to run the Script

The script has the following start options mostly for debugging purposes:

-i for importing directly into mongodb, for that the `CLIENT`, `DB` and `COLLECTION` global variables need to be filled with the respective information in the main function from the mongoDB database. They are pre-filled with my testing database as can be seen below.

```
1       global CLIENT
2       CLIENT = MongoClient('mongodb://localhost:27017/')
3       global DB
4       DB = CLIENT['gallery']
5       global COLLECTION
6       COLLECTION = DB['groimp_projects']
```

-r for searching sub folders of the current folder as well, which is needed for searching the server files properly. It is optional to give a path for the folder that will contain the `.JSON` files as a last argument, the default path is `./json_files`.

-a for only searching the current folder, used mainly for testing purposes. Using this option in conjunction with '-r' can lead to errors or unintended behavior.

neither -r nor -a for converting a single `.DESC` file to `.JSON`. This expects a single `.DESC` file name as an argument.

For working with the old database files, the script needs to be placed next to or inside the old gallery server folder `gallerySRC` and run the file with the following command:

```
python ./desc_to_json_v4_1.py -r
```

The code then sifts through the old server files looking for `.DESC` files and their associated `.RGG` or `.GSZ` file as well as the the image file in either `.PNG`, `.JPG` or `.GIF` file format. All the files are then copied to new folders at the location of the script. The `.RGG` and `.GSZ` files are placed in the `project` folder, the images are placed in the `images` folder and the newly created `.JSON` files are placed in the `json_file` folder (if this was not changed).

Before rerunning the script any existing `.JSON` files need to be deleted or moved, as they will NOT be replaced, leading to duplicates with slightly altered names.

4.7.1 Uploading the Legacy Data to the New Server

To automate the upload of the legacy data to the new server, I created another script `legacy_data_api_requests_v4_1.py`. This is a python module that is to be used in the interactive python shell. The script needs to be placed in the same folder as the folders created by the previous script. To start the shell with the module use the following command:

```
python -i -m legacy_data_api_requests_v4_1
```

The module contains multiple functions.

```
List of available functions:
setEnv() - set Environment to prod(uction) or test.
login() - logs in to the API providing a web token for authentification.
addCategories() - Adds categories to the API.
uploadOne(json_file_path) - Uploads a single JSON file to the API.
uploadRelated() - Uploads JSON files with non-empty 'relatedmodels' field
                  and updates related projects.
uploadAll() - Uploads all JSON files to the API.
uploadNoRelation() - Uploads JSON files with empty 'relatedmodels' field.
version_test() - Sends a test request to the 'version' endpoint.
upload_queue_test() - Tests the upload queue functionality.
process_upload_queue() - Processes the upload queue.
help() - Displays this help message.
```

The following assumes a freshly set up server: Before uploading the data via the API the user needs to set the correct environment first, by running the `setEnv()` function. Providing 'test' as a parameter or none at all will set the `api_endpoint` to `test_api_endpoint` while providing 'prod' as a parameter sets it to `prod_api_endpoint`. The current `test_api_endpoint` is set to a locally set up test server while the `prod_api_endpoint` is set to the new `grogra.de` gallery on `gallery.grogra.de`.

The next step is to log in on the API. For that the user needs an existing `grogra.de` account, run the `login()` function and then enter the `grogra.de` login credentials. The session token will be saved automatically and is available as long as the interactive python environment is open or until it expires on the server side (4 hours).

Now the user can simply run `addCategories()` to upload all the pre-defined categories from the script and then `uploadAll()` to start the upload process for the legacy data. This will first upload all projects without relations to other projects by calling `uploadNoRelation()`.

After that `uploadRelated()` is run to upload all Projects with relation and setting the relation in the relatedmodels field to the correct ID. When the related Project can not be found it it probably is a bi-directional relation, so the ID that is returned on database entry creation and information about the related project are pushed into the upload queue as a tuple.

When all projects are uploaded the `processUploadQueue()` function is called which works through the previously created upload queue and sets all the bi-directional relations for the projects.

The projects are now successfully uploaded via API to the server.

4.8 Conclusion

This work successfully achieved its primary objective of designing and implementing a modern, database-driven gallery system for the GroIMP community. The developed platform introduces significant improvements through its structured database schema, RESTful API architecture, and intuitive web interface.

A key accomplishment was the successful migration of legacy data, ensuring continuity with existing community contributions while enabling enhanced search capabilities, categorization, and version management. The technology stack of JavaScript, Node.js, Express, and MongoDB proved effective for rapid development. However, the considerable time spent debugging runtime errors highlighted that TypeScript would have been a more suitable choice, as its static type checking would have prevented many issues while maintaining JavaScript's flexibility [24].

Several areas remain for future development to fully realize the platform's potential. The tagging system requires complete web interface integration, and group functionality with authorization controls needs implementation to enable collaboration and selective project sharing. Most importantly, tighter integration with the GroIMP software itself would provide substantial benefits through direct API access for seamless project loading and saving.

In summary, this work establishes a robust foundation for a modernized GroIMP gallery system that substantially improves accessibility, maintainability, user experience and collaboration within the GroIMP community.

Bibliography

[1] O. Kniemeyer, "Design and implementation of a graph grammar based language for functional-structural plant modelling," PhD thesis, Brandenburg University of Technology Cottbus-Senftenberg, 2008. [Online]. Available: https://opus4.kobv.de/opus4-btu/frontdoor/index/index/docId/462

[2] M. Fabrika, L. Scheer, R. Sedmák, W. Kurth, and M. Schön, "Crown architecture and structural development of young Norway spruce trees (*Picea abies* Karst.): A basis for more realistic growth modelling," *BioResources*, vol. 14, no. 1, pp. 908–921, 2019. [Online]. Available: https://bioresources.cnr.ncsu.edu/resources/crown-architecture-and-structural-development-of-young-norway-spruce-trees-picea-abies-karst-a-basis-for-more-realistic-growth-modelling/

[3] M. Henke, W. Kurth, and G. H. Buck-Sorlin, "Fspm-p: towards a general functional-structural plant model for robust and comprehensive model development," *Frontiers of Computer Science*, vol. 10, no. 6, pp. 1103–1117, 2016. [Online]. Available: https://doi.org/10.1007/s11704-015-4472-8

[4] G. Heidsieck, T. Oberländer, T. Hay, and W. Kurth, "Pointcloud: Implementation of point clouds as graphs in the 3d plant modeling platform groimp," *Journal of Open Source Software*, vol. 10, no. 110, p. 8062, 2025. [Online]. Available: https://doi.org/10.21105/joss.08062

[5] GitHub, Inc., "GitHub: Where the world builds software," https://github.com, 2024, accessed: September 20, 2025.

[6] CERN and OpenAIRE, "Zenodo - Research. Shared." https://zenodo.org, 2024, digital repository for research data and software. Accessed: September 20, 2025.

[7] N. Le Novère, B. Bornstein, A. Broicher, M. Courtot, M. Donizelli, H. Dharuri, L. Li, H. Sauro, M. Schilstra, B. E. Shapiro, J. L. Snoep, and M. Hucka, "BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems," *Nucleic acids research*, vol. 34, no. suppl_1, pp. D689–D691, 2006, https://www.ebi.ac.uk/biomodels/.

[8] The Apache Software Foundation, "Apache forrest: A publishing framework," Feb. 2011, available at: https://forrest.apache.org. Retired project; version 0.9 (2011-02-07). Accessed on 2025-09-16.

[9] WordPress Foundation, "Wordpress developer resources," https://developer.wordpress.org/, accessed: 3 September 2025.

[10] P. P. Chen, "The entity–relationship model: Toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, 1976, available at: https://dl.acm.org/doi/10.1145/320434.320440, accessed: 3 September 2025.

[11] Oracle Corporation, "Mysql 8.0 reference manual," https://dev.mysql.com/doc/refman/8.0/en/, accessed: 3 September 2025.

[12] MongoDB, Inc., "Mongodb documentation," https://www.mongodb.com/docs/, accessed: 3 September 2025.

[13] Mozilla Contributors, "Javascript | mdn web docs," https://developer.mozilla.org/en-US/docs/Web/JavaScript, accessed: 3 September 2025.

[14] Microsoft Corporation, "Typescript documentation," https://www.typescriptlang.org/docs/, accessed: 3 September 2025.

[15] ——, "Visual studio code documentation," https://code.visualstudio.com/docs, accessed: 3 September 2025.

[16] OpenJS Foundation, "Node.js documentation," https://nodejs.org/en/docs, accessed: 3 September 2025.

[17] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000, available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, accessed: 3 September 2025.

[18] Express.js Contributors, "Express.js documentation," https://expressjs.com/, accessed: 3 September 2025.

[19] EJS Contributors, "Ejs: Embedded javascript templates," https://ejs.co/, accessed: 3 September 2025.

[20] Valeri Karpov and contributors, "Mongoose — elegant mongodb object modeling for node.js," https://mongoosejs.com, 2025, accessed on 2025-09-16.

[21] Sidorares, Andrey and contributors, "mysql2/promise — mysql client for node.js," https://github.com/sidorares/node-mysql2, 2025, accessed on 2025-09-16.

[22] E. V. Contributors, "express-validator," 2024, zugriff am 18. September 2025. [Online]. Available: https://www.npmjs.com/package/express-validator

[23] P. S. Foundation, "json — json encoder and decoder," 2025, zugriff am 18. September 2025. [Online]. Available: https://docs.python.org/3/library/json.html

[24] D. Fateh, "TypeScript vs. JavaScript: Explaining the differences," Oct. 2023, accessed: 29 October 2025. [Online]. Available: https://www.contentful.com/blog/typescript-vs-javascript-explaining-the-differences/

Appendix A

Legacy Data

A.1 DESC File Pattern

```
TITLE:

SHORTDESCRIPTION:

DOWNLOAD: <a href="*.gsz">*.gsz</a>

DATE:

AUTHOR:

DESCRIPTION:

HOWITWORKS:

THINGSTONOTICE:

THINGSTOTRY:

EXTENDINGTHEMODEL:

RELATEDMODELS:

CREDITSANDREFERENCES:
```

A.2 DESC to JSON Conversion

A.2.1 DESC File

```
TITLE: ant

SHORTDESCRIPTION: A simple ant model.

DOWNLOAD: <a href="ant.gsz">ant.gsz</a>.

DATE: 20.04.2009

AUTHOR: O. Kniemeyer

DESCRIPTION: Ants are a popular subject in Artificial Life. A simplistic
↪ simulation can be implemented easily.

HOWITWORKS:

THINGSTONOTICE:

THINGSTOTRY:

EXTENDINGTHEMODEL:

RELATEDMODELS: <a href="antb.html">antb</a>, <a
↪ href="ant_simulation.html">ant_simulation</a>

CREDITSANDREFERENCES:
```

A.2.2 JSON File after Refactoring with Python Script

```
{
    "title": "ant",
    "shortdescription": "A simple ant model.",
    "projectfile": "ant.gsz",
    "author": "O. Kniemeyer",
    "description": "Ants are a popular subject in Artificial Life. A simplistic
    ↪ simulation can be implemented easily.",
    "howitworks": "",
    "thingstonotice": "",
    "thingstotry": "",
    "extendingthemodel": "",
```

```
    "relatedmodels": [
        "antb",
        "ant_simulation"
    ],
    "creditsandreferences": "",
    "createdAt": "2009-04-20T00:00:00.000+01:00",
    "updatedAt": "2009-04-20T00:00:00.000+01:00",
    "categories": [
        "ArtificialLife"
    ],
    "access": {
        "public": true,
        "readAccess": [],
        "writeAccess": []
    },
    "imagefile": "ant.png"
}
```


A.2.3 Database JSON Document


```
{
    "access": {
    "public": true,
    "readAccess": [],
    "writeAccess": []
    },
    "_id": "67053a3e256715cda41c85b5",
    "title": "ant",
    "shortdescription": "A simple ant model.",
    "uploadedBy": "dforys",
    "description": "Ants are a popular subject in Artificial Life. A simplistic
    ↪  simulation can be implemented easily.",
    "howitworks": "",
    "thingstonotice": "",
    "thingstotry": "",
    "extendingthemodel": "",
    "relatedmodels": [
    "67053a3e256715cda41c85bb",
    "67053a3e256715cda41c85c7"
    ],
    "creditsandreferences": "",
    "categories": [
    "6705382e256715cda41c7dec"
```

```
    ],
    "tags": [],
    "imagefile": "ant.png",
    "versions": [
    "67053a3e256715cda41c85b7"
    ],
    "createdAt": "2009-04-19T23:00:00.000Z",
    "updatedAt": "2024-10-08T13:57:23.252Z",
    "__v": 0
}
```


Associated Version


```
{
    "_id":"67053a3e256715cda41c85b7",
    "dependencies":[],
    "version":"1.0",
    "description":"",
    "__v":0
}
```


A.3 Categories

```
AbstractModels
AlgorithmicBeauty
Architecture
ArtificialLife
CoursesAndTutorials
FSPM
FormalSystems
FractalsAndCurves
Games
GroPhysics
LeafModels
Others
PlantStands
Plants
StepByStep
StepByStep2
Teaching
Technics
```

```
XL4C4D
english
```

Appendix B

Database Models

B.1 GroIMP Projects Model

File: `db_models/groimp_projects.js`
Collection Name: `groimp_projects`
Main Entity: Core project model representing GroIMP modelling projects

B.1.1 Schema Fields

Basic Information

- **title** (String) – Project title
- **shortdescription** (String) – Brief project summary
- **authors** (String) – Project authors/creators
- **uploadedBy** (String) – User who uploaded the project
- **imagefile** (String) – Filename of project preview image

Detailed Documentation

- **description** (String) – Full project description
- **howitworks** (String) – Technical explanation of project mechanics
- **thingstonotice** (String) – Important observations for users
- **thingstotry** (String) – Suggested experiments/modifications
- **extendingthemodel** (String) – Guide for extending the project
- **creditsandreferences** (String) – Attribution and citations

Relationships (ObjectId References)

- **relatedmodels** (Array) – References to other `groimp_projects`
- **categories** (Array) – References to `categories` collection
- **tags** (Array) – References to `tags` collection
- **versions** (Array) – References to `versions` collection

Access Control System

- **access.public** (Boolean, default: false) – Public visibility flag
- **access.readAccess** (Array) – References to `groups` with read permissions (not in Use yet)
- **access.writeAccess** (Array) – References to `groups` with write permissions (not in Use yet)

Automatic Timestamps

- **createdAt** – Auto-generated creation timestamp
- **updatedAt** – Auto-generated last modification timestamp

Format Example:"2023-10-03T14:30:00.000+00:00"

B.2 Versions Model

File: `db_models/versions.js`
Collection Name: `Version`
Purpose: Stores different versions/iterations of project files

B.2.1 Schema Fields

- **dependencies** (Array of Strings) – Required dependencies for this version
- **version** (String) – Version identifier (e.g., "1.0.0", "v2.1", defaults to 1.0 for first project upload)
- **description** (String) – Version-specific description/changelog

B.2.2 File Storage

- Actual files (.gsz, .rgg, .zip) stored in filesystem: `FILE_LOCATION/project_id/version_id/`
- Database only stores metadata, not file content

B.3 Tags Model

File: `db_models/tags.js`
Collection Name: `tags`
Purpose: tagging system for user made categorization of projects

B.3.1 Schema Fields

- **name** (String, required, unique) – Tag name/label

B.4 Categories Model

File: `db_models/categories.js`
Collection Name: `categories`
Purpose: Admin made categorization system for projects

B.4.1 Schema Fields

- **name** (String, required, unique) – Category name

B.5 Groups Model

File: `db_models/groups.js`
Collection Name: `groups`
Purpose: User group management for access control, still work in progress

B.5.1 Schema Fields

- **name** (String) – Group identifier
- **description** (String) – Group purpose/description
- **members** (Array) – References to `users` collection

B.6 Database Relationships Overview

```
groimp_projects (1) ↔ (many) versions
groimp_projects (many) ↔ (many) categories
groimp_projects (many) ↔ (many) tags
groimp_projects (many) ↔ (many) groimp_projects (self-reference)
groimp_projects (many) ↔ (many) groups (via access control)
groups (many) ↔ (many) users (external reference)
```

Appendix C

API Reference Documentation

This appendix provides a complete reference for all available API endpoints in the gallery database server. All endpoints follow RESTful conventions with appropriate HTTP methods and status codes.

C.1 Authentication

C.1.1 Login

Endpoint: `POST /login`

Description: Authenticate user credentials and establish session

Content-Type: `application/x-www-form-urlencoded`

Authentication: None required

Request Body:
- `username` (string, required) – User login name
- `password` (string, required) – User password

Response: Sets HTTP-only authentication cookie

Status Codes: 200 (Success) | 403 (Authentication failed) | 500 (Server error)

C.1.2 Logout

Endpoint: `POST /logout`

Description: Terminate user session and clear authentication cookie

Authentication: None required

Response: Clears authentication cookie

Status Codes: 200 (Success)

C.2 Project Management (JSON API)

C.2.1 Retrieve Projects

Endpoint: `GET /project_json`

Description: Retrieve all projects with optional filtering

Authentication: None required

Query Parameters:
- `category` (ObjectId) – Filter by category ID
- `uploadedBy` (string) – Filter by uploader username
- `access.public` (boolean) – Filter by public access
- Any other project field for exact matching

Status Codes: 200 (Success) | 404 (No projects found) | 500 (Server error)

Endpoint: `GET /project_json/:project_id`

Description: Retrieve specific project by ID

Authentication: None required

Path Parameters: `project_id` (ObjectId, required) – Unique project identifier

Status Codes: 200 (Success) | 404 (Project not found) | 500 (Server error)

C.2.2 Create Project

Endpoint: `POST /project_json`

Description: Create new project with initial version and file uploads

Content-Type: `multipart/form-data`

Authentication: JWT token required

Form Fields: `projectdata` (JSON string, required) – Project metadata:
- `title` (string) – Project title
- `shortdescription` (string) – Brief description
- `description` (string) – Detailed description
- `categories` (array of strings) – Category names

- `howitworks` (string) – Implementation details
- `access` (object) – Access control settings
- `relatedmodels` (array of ObjectIds) – Related project references

`versiondata` (JSON string, required) – Initial version metadata:

- `version` (string) – Version identifier
- `dependencies` (array of strings) – Required dependencies
- `description` (string) – Version description

File uploads:

- `imagefile` (file, required) – Project preview image
- `gszfile` (file, required) – GroIMP model file

File Limits: 1GB maximum per file

Status Codes: 201 (Created) | 400 (Bad request) | 401 (Unauthorized) | 500 (Server error)

C.2.3 Update Project

Endpoint: `PUT /project_json/:project_id`

Description: Complete replacement of project metadata

Content-Type: `application/json`

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Request Body: Complete project object

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

Endpoint: `PATCH /project_json/:project_id`

Description: Partial update of project metadata with optional image replacement

Content-Type: `multipart/form-data`

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Form Fields:

- `title` (string, optional) – Updated project title
- `shortdescription` (string, optional) – Updated brief description

- `categories` (string, optional) – Comma-separated category names
- `relatedmodels` (string, optional) – Comma-separated project IDs
- `current_imagefile` (string, optional) – Current image filename for replacement
- `imagefile` (file, optional) – New project preview image
- Any other project field for partial updates

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

C.2.4 Delete Project

Endpoint: `DELETE /project_json/:project_id`

Description: Delete project and all associated versions and files

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Side Effects:
- Deletes all project versions from database
- Removes all associated files from filesystem
- Cascades deletion to version directory structure

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

C.2.5 Version-Specific Operations

Endpoint: `PATCH /project_json/:project_id/:version_id`

Description: Update specific project version with file replacement

Content-Type: `multipart/form-data`

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Form Fields:
- `imagefile` (file, optional) – New model file for version
- Version metadata fields for updates

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

Endpoint: `DELETE /project_json/:project_id/:version_id`

Description: Delete specific project version

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Side Effects: Removes version files and directory structure

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

C.3 Version Control

C.3.1 Retrieve Version

Endpoint: `GET /version/:id`

Description: Retrieve specific version metadata by ID

Authentication: None required

Path Parameters: `id` (ObjectId, required) – Version identifier

Status Codes: 200 (Success) | 404 (Not found) | 500 (Server error)

C.3.2 Create Version

Endpoint: `POST /version/:project_id`

Description: Add new version to existing project

Content-Type: `multipart/form-data`

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Form Fields:
- `version` (string, required) – Version identifier
- `dependencies` (string, required) – Comma-separated dependencies
- `description` (string, required) – Version description
- `gszfile` (file, required) – GroIMP model file

Side Effects: Updates parent project's versions array

Status Codes: 201 (Created) | 400 (Bad request) | 401 (Unauthorized) | 403 (Forbidden) | 500 (Server error)

C.3.3 Update Version

Endpoint: `PATCH /version/:version_id`

Description: Update version metadata and optionally replace model file

Content-Type: `multipart/form-data`

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Form Fields:
- `version` (string, optional) – Updated version identifier
- `dependencies` (string, optional) – Updated dependencies
- `description` (string, optional) – Updated description
- `project_id` (ObjectId, required) – Parent project identifier
- `gszfile` (file, optional) – New model file

Side Effects: Replaces existing version files when new file provided

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

C.3.4 Delete Version

Endpoint: `DELETE /version/:version_id`

Description: Delete version and associated files

Authentication: JWT token required (owner or admin)

Authorization: Owner or administrator privileges

Side Effects: Removes version directory and all contained files

Status Codes: 200 (Success) | 401 (Unauthorized) | 403 (Forbidden) | 404 (Not found) | 500 (Server error)

C.3.5 Download Version Files

Endpoint: `GET /version/download/:project_id/:version_id`

Description: Download model files (.gsz, .rgg, .zip) for specific version

Authentication: None required

Path Parameters:
- `project_id` (ObjectId, required) – Project identifier
- `version_id` (ObjectId, required) – Version identifier

Response: Binary file download with appropriate content headers

Status Codes: 200 (Success) | 404 (File not found) | 500 (Server error)

C.4 Category Management

C.4.1 Retrieve Categories

Endpoint: `GET /category`

Description: Retrieve all available categories

Authentication: None required

Response: Array of category objects with _id and name fields

Status Codes: 200 (Success) | 500 (Server error)

Endpoint: `GET /category/search?q=:query`

Description: Search categories by name prefix (autocomplete functionality)

Authentication: None required

Query Parameters: `q` (string, required) – Search prefix (case-insensitive)

Response: Limited array of matching categories (max 10 results)

Status Codes: 200 (Success) | 400 (Missing query) | 500 (Server error)

Endpoint: `GET /category/:name`

Description: Retrieve specific category by name identifier

Authentication: None required

Path Parameters: `name` (string, required) – Category identifier

Status Codes: 200 (Success) | 404 (Not found) | 500 (Server error)

C.4.2 Create Category

Endpoint: `POST /category`

Description: Create new category

Content-Type: `application/json`

Authentication: JWT token required

Request Body: `name` (string, required) – Category name (must be unique)

Status Codes: 201 (Created) | 400 (Bad request) | 401 (Unauthorized) | 500 (Server error)

C.4.3 Delete Category

Endpoint: `DELETE /category/:name`

Description: Delete category by name identifier

Authentication: JWT token required

Path Parameters: `name` (string, required) – Category identifier

Warning: Does not automatically remove category references from existing projects

Status Codes: 200 (Success) | 401 (Unauthorized) | 404 (Not found) | 500 (Server error)

C.5 Tag Management

C.5.1 Retrieve Tags

Endpoint: `GET /tags`

Description: Retrieve all available tags

Authentication: None required

Response: Array of tag objects with _id and name fields

Status Codes: 200 (Success) | 500 (Server error)

Endpoint: `GET /tags/search?q=:query`

Description: Search tags by name prefix (autocomplete functionality)

Authentication: None required

Query Parameters: `q` (string, required) – Search prefix (case-insensitive)

Response: Limited array of matching tags (max 10 results)

Status Codes: 200 (Success) | 400 (Missing query) | 500 (Server error)

Endpoint: `GET /tags/:id`

Description: Retrieve specific tag by ObjectId

Authentication: None required

Path Parameters: `id` (ObjectId, required) – Tag identifier

Status Codes: 200 (Success) | 404 (Not found) | 500 (Server error)

C.5.2 Create Tag

Endpoint: `POST /tags`

Description: Create new tag

Content-Type: `application/json`

Authentication: JWT token required

Request Body: `name` (string, required) – Tag name

Status Codes: 201 (Created) | 400 (Bad request) | 401 (Unauthorized) | 500 (Server error)

C.5.3 Update Tag

Endpoint: `PUT /tags/:id`

Description: Complete replacement of tag name

Content-Type: `application/json`

Authentication: JWT token required

Request Body: `name` (string, required) – New tag name

Status Codes: 200 (Success) | 401 (Unauthorized) | 404 (Not found) | 500 (Server error)

Endpoint: `PATCH /tags/:id`

Description: Partial update of tag name

Content-Type: `application/json`

Authentication: JWT token required

Request Body: `name` (string, optional) – Updated tag name

Status Codes: 200 (Success) | 401 (Unauthorized) | 404 (Not found) | 500 (Server error)

C.5.4 Delete Tag

Endpoint: `DELETE /tags/:id`

Description: Delete tag by ObjectId

Authentication: JWT token required

Path Parameters: `id` (ObjectId, required) – Tag identifier

Status Codes: 200 (Success) | 401 (Unauthorized) | 404 (Not found) | 500 (Server error)

C.6 Project Gallery (Web Interface)

C.6.1 Gallery Views

Endpoint: `GET /project_gallery`

Description: Render paginated gallery interface with filtering and search

Authentication: JWT token required

Query Parameters:
- `title` (string, optional) – Search in project titles
- `category` (string/ObjectId, optional) – Filter by category name or ID
- `tag` (string/ObjectId, optional) – Filter by tag name or ID
- `uploader` (string, optional) – Filter by uploader username
- `description` (string, optional) – Search in all description fields
- `page` (integer, optional) – Page number for pagination (default: 1)
- `limit` (integer, optional) – Items per page (default: 10)
- `sortBy` (string, optional) – Sort field name
- `order` (string, optional) – Sort order: 'asc' or 'desc'

Response: Rendered HTML gallery page with pagination controls

Status Codes: 200 (Success) | 401 (Unauthorized) | 500 (Server error)

Endpoint: `GET /project_gallery/categories`

Description: Render categories overview page

Authentication: None required

Response: Rendered HTML page listing all categories

Status Codes: 200 (Success) | 500 (Server error)

Endpoint: `GET /project_gallery/upload`

Description: Render project upload form

Authentication: JWT token required

Response: Rendered HTML upload form

Status Codes: 200 (Success) | 401 (Unauthorized)

Endpoint: `GET /project_gallery/login`

Description: Render login form

Authentication: None required

Response: Rendered HTML login form

Status Codes: 200 (Success)

Endpoint: `ALL /project_gallery/forbidden`

Description: Render access denied page

Authentication: None required

Response: Rendered HTML forbidden access page

Status Codes: 200 (Success)

C.6.2 Project Detail Views

Endpoint: `GET /project_gallery/:project_id`

Description: Render detailed project view with versions, related models, categories, and tags

Authentication: JWT token required

Path Parameters: `project_id` (ObjectId, required) – Project identifier

Response: Rendered HTML project detail page with populated relationships

Status Codes: 200 (Success) | 401 (Unauthorized) | 404 (Not found) | 500 (Server error)

Endpoint: `GET /project_gallery/:project_id/image`

Description: Serve project preview image file

Authentication: None required

Path Parameters: `project_id` (ObjectId, required) – Project identifier

Response: Binary image file stream

Status Codes: 200 (Success) | 404 (Image not found)

C.7 Security and Authorization

C.7.1 Authentication Requirements

JWT Tokens: Managed via HTTP-only cookies for security

Token Expiration: 4 hours from login

Secure Cookies: HTTPS-only in production environment

Session Management: Automatic cookie clearing on logout

C.7.2 Authorization Levels

Public Access: GET operations for projects, categories, tags, versions, and downloads

Authenticated Users: Project creation, gallery access, upload interface

Resource Owners: Full read/write access owned projects and versions

Administrators: Full system access, user management capabilities

C.7.3 Input Validation and Security

Sanitization: All user inputs processed through sanitization middleware

File Upload Limits: 1GB maximum file size per upload

XSS Protection: HTML content sanitization for multiline text fields

Path Traversal Protection: Secure file handling with controlled directory access

SQL Injection Prevention: Parameterized queries for MySQL authentication

C.8 Error Handling

C.8.1 HTTP Status Codes

200 OK	Successful operation
201 Created	Resource created successfully
400 Bad Request	Invalid request format or missing required fields
401 Unauthorized	Missing or invalid authentication credentials
403 Forbidden	Insufficient permissions for requested operation
404 Not Found	Requested resource does not exist
500 Internal Server Error	Server-side processing error

C.8.2 Error Response Format

All error responses follow a consistent JSON format:

```
{
  "message": "Human-readable error description",
  "error": "Technical error identifier" // Optional
}
```

C.8.3 File System Error Handling

Upload Failures: Automatic cleanup of partially uploaded files

Directory Operations: Graceful handling of file system permission errors

File Replacement: Atomic operations to prevent data corruption

Cleanup on Error: Rollback of file operations when database operations fail

C.9 API Usage Examples

C.9.1 Authentication Flow

Example login request:

```
POST /login
Content-Type: application/x-www-form-urlencoded

username=john.doe&password=securepassword123
```

C.9.2 Project Creation Example

Example project creation with file upload:

```
POST /project_json
Content-Type: multipart/form-data
Authorization: Bearer <jwt-token>

projectdata={"title":"Plant Growth Model","shortdescription":"Simulates plant growt
versiondata={"version":"1.0","dependencies":["GroIMP 2.3"],"description":"Initial r
imagefile=<binary-image-data>
gszfile=<binary-model-data>
```

C.9.3 Query Examples

Retrieve public projects in a specific category:

```
GET /project_json?access.public=true&category=<category-id>
```

Search projects by uploader:

```
GET /project_json?uploadedBy=researcher123
```

Appendix D

HTML Sanitization and Allowed Tags

This appendix details the HTML sanitization policies implemented in the application to prevent XSS attacks while maintaining essential formatting capabilities. The system employs a whitelist approach, where all HTML tags, CSS styling, and JavaScript event handlers are automatically removed from user input to ensure application security. Only the following whitelisted elements are permitted:

D.1 Text Formatting

- **`<b>`** – Bold text formatting
- **`<i>`** – Italic text formatting
- **`<em>`** – Emphasized text
- **`<strong>`** – Strong importance text

D.2 Structure and Lists

- **`<p>`** – Paragraph elements
- **`<br>`** – Line breaks
- **`<ul>`** – Unordered lists
- **`<ol>`** – Ordered lists
- **`<li>`** – List items

D.3 Links

- **`<a>`** – Hyperlinks (only `href` attribute allowed)