The Librarian - code review documentation

Overview

This part of code responsible for working with library user abstract model
The project runs on *Django* framework and use build-in *REST API* system
Also this code contains *permissions system* example

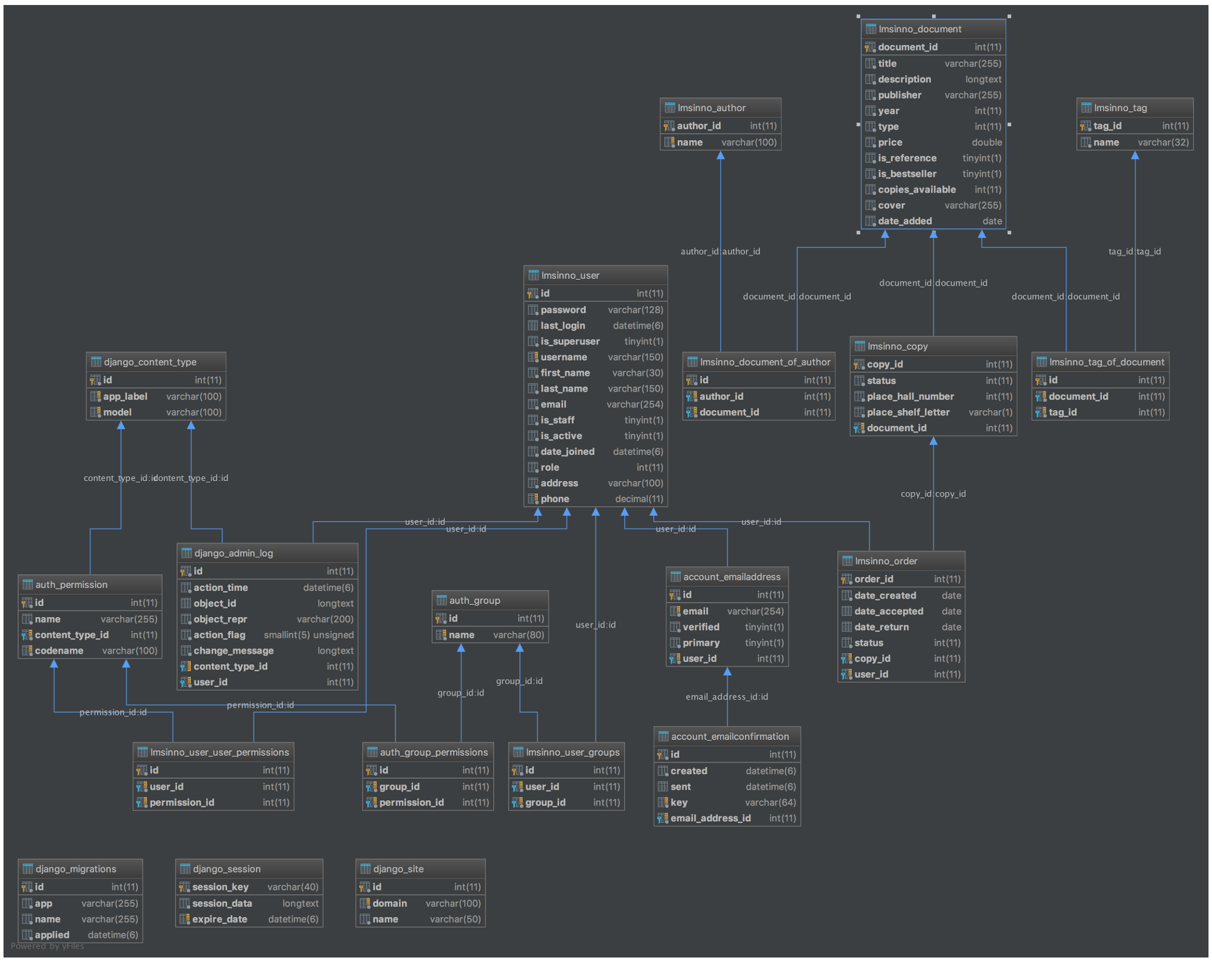
Goals

Build flexible system using only framework functionality

Usage Scenarios

Receive, update or delete information about user in library

Class Diagrams



Code

Categories for review

🟥 - Error 🟧 - Warning 🟩 - Well done!

1. Design decisions

1. **Reviewer1**: Methods reuses properly, all common things splitted into constants in same methods

2. **Reviewer1**: Sensitive constants are in the corresponding class
3. **Reviewer2**: Using separate class for constant

2. API design

1. **Reviewer1**: All used API methods and classes are in the following implementation (for review)
2. **Reviewer1**: Well-designed API implementation as prod ready client-server behaviour

3. Architecture (including inheritance hierarchy, if any)

1. **Reviewer1**: Modules are created separately, smart usage of permission methods
2. **Reviewer1**: May use decorators for some shells (e.g permission control), as far as Python is wrapper-friendly

   Owner: Ok! *[TO_BE_FIXED]*

4. Implementation techniques, in particular choice of data structures and algorithms

1. **Reviewer1**: Algorithms use as less resources as possible
2. **Reviewer1**: Smart choice to use implemented DS in MySQL database for data queries
3. **Reviewer2:** Using `@staticmethod` for saving memory

5. Exceptions handling - Contracts

1. **Reviewer3**: `Try - catch` envelope for error handling
2. **Reviewer3**: Different exception types for different kind of exceptions
3. **Reviewer3**: Serializers for type checking and contract resolution
4. **Reviewer3**: JWT token usage

6. Programming style, names

1. **Reviewer3**: Readable code
2. **Reviewer3**: Constants are in special class for reuse
3. **Reviewer4:** Smart usage of blank lines. Makes code readable.
4. **Reviewer4:** All requirements of python style are fulfilled (classes' names in camel case, snake case for variables and functions' names, ect.)
5. **Reviewer3**: Next line (if condition is too long) in `if` branching should be shifted to the start but not to the very end of line => less readable (lines 28, 31...)

   Owner: Ok! *[TO_BE_FIXED]*

7. Comments and documentation

1. **Reviewer3**: DocStrings are used properly
2. **Reviewer3**: Very informative comments
3. **Reviewer4:** It is better to write in DocStrings only that arguments which must be described or describe all of them, empty description does not look good.

   Owner: Ok! *[TO_BE_FIXED]*
4. **Reviewer4:** Return position in DocStrings are informative