hexagonal Documentation

ionagamed

Apr 25, 2018

CONTENTS

hexagonal is an LMS (library management system) build as a first year bachelors' project at Innopolis University. Class diagram can be found at hexagonal.docs.hexagonal-classes.uml. Also a `rendered version`.

It should also be noted that as a dynamic by nature language, Python doesn't have native support for class attribute documentation. This is also a bad practice, because it hurts readability of code.

CHAPTER

ONE

REASONING

The part in which we describe why we chose what we chose.

1.1 Technology stack

Our technology stack is mainly focused on flexibility and ease-of-use:

- Flask was used over django because it is definitely more lightweight, and is more flexible.
- SQLAlchemy was used because of its, again, flexibility and speed of development. SQLAlchemy uses data-mapper model instead of Django's familiar Active Record, and it allows for some interesting features such as association proxies.
- Itsdangerous is a small library with 'good' cryptography. It is a common knowledge that you shouldn't implement cryptography by yourself, and this library contains a lot of neat things to help with that.
- PostgreSQL was chosen for backend because it is feature-rich, stable, and easy to use. SQLite didn't have ARRAY builtin type and full index support for JSON. MySQL just tends to fall apart randomly (from my experience). Other SQL (MS SQL, Oracle) are too much enterprise for such project.

CHAPTER

TWO

MODEL

This part contains overview for used model classes. For a more in-depth explanation see *Classes*

2.1 Users

The user model consists currently of five classes:

- `User` is an abstract base class for all users
- `Librarian` is a kind of an admin which can manage everything else
- `Patron` is an abstract base class for all patrons
- `StudentPatron` is a class that represents all student-patrons
- `FacultyPatron` is a class that represents all faculty-patrons

There is a single *root* user throughout the runtime, which is a librarian. This is done because only a librarian can create new users, and wee need an entry point.

2.2 Documents

The document model consists currently of five classes:

- `Document` is an abstract base class for all documents, which contains all common fields.
- `DocumentCopy` is a class for a copy of a document, which is just linked to a loan and a document.
- And a class for each document type (`Book`, `AVMaterial`, `JournalArticle`).

2.3 Booking System

The `Loan` is the single class which represents the booking system. Each *Loan* has a status (`hexagonal.model.loan.Loan.Status`) - it can be: *requested* - user wants this book. Then librarian can push a button and make the loan *approved* - which means that it is currently in patron's possession. Then, finally, user brings the book back, and it becomes *returned*, until a librarian approves the return request, and the loan is removed. This is the whole lifecycle of one *Loan*.

2.4 Priority Queue

The `QueuedRequest` is the single class responsible for the priority queue. It uses SQL *ORDER BY*'s to implement the required behaviour, and contains indices for them to be fast enough.

CHAPTER

THREE

CLASSES

This part contains in-depth view of the used classes. Subclassing with only role changing was used to implement a rather flexible permission system.

3.1 User model

In SQL users use single-table inheritance model, which allows for compact data storage, when we do not have additional fields in subclasses.

class hexagonal.model.user.**User**(***kwargs*)

Base class for all users in the system. Should not be directly instantiated. (*hexagonal.auth* currently does that, but i'm working on it)

address

User's full address.

card_number

User's library card number.

has_permission(*permission*)

Whether this user has the required permission.

By default returns whether permission is present in the class static field *permissions*.

Parameters **permission** – permission to be checked.

Returns whether the current user has the required permission.

id

Integer primary key.

login

Login of the user. Must be unique.

name

User's full name.

password

Password hash of the user.

phone

User's phone number.

reset_password

Reset password flag. If true, on next login user would be prompted to reset the password.

role
Polymorphic identity of the user. Used to implement inheritance.

class hexagonal.model.patron.**Patron**(**kwargs*)
Bases: *hexagonal.model.user.User*

Abstract base class for patrons. Should not be directly instantiated

checkout(*document_copy*)
Try to checkout the required document_copy. Raises TypeError if document_copy is not a DocumentCopy. Raises ValueError if document_copy is not available for loan.

Parameters **document_copy** – to be checked out.

Returns loan object

get_borrowed_document_copies()
Get document copies that are currently borrowed by this user.

Returns list of document_copies

get_borrowed_document_copy_count()
Get the amount of borrowed document copies.

Returns amount

get_checkout_period_for(*document*)
Get checkout period for a specific document. Abstract in User.

Parameters **document** – to be checked out

Returns timedelta

get_loan_count()
Get the amount of all approved loans for this patron.

Returns list.

get_loans()
Get all approved loans for this patron.

Returns list.

get_overdue_loan_count()
Get the amount of overdue associated items.

Returns amount

get_overdue_loans()
Get current overdue associated loans.

Returns list of loans

get_total_overdue_fine()
Get total current overdue fine across all loans. Abstract in User.

Returns total overdue fine, in rubles

loan_query()
Get BaseQuery for all approved loans with borrowed documents

Returns BaseQuery

overdue_loan_query()
Get BaseQuery for all overdue loans of this user.

Returns BaseQuery

class hexagonal.model.librarian.**Librarian**(***kwargs*)
Bases: *hexagonal.model.user.User*

Librarian type of user.

has_permission(*permission*)
Whether this user has the required permission.

Parameters **permission** – permission to be checked.

Returns whether the current user has the required permission.

class hexagonal.model.student_patron.**StudentPatron**(***kwargs*)
Bases: *hexagonal.model.patron.Patron*

Student patron user type. Internal model, gets squashed in the api.

get_checkout_period_for(*document*)
Get checkout period for a specific document.

Parameters **document** – to be checked out

Returns timedelta

class hexagonal.model.faculty_patron.**FacultyPatron**(***kwargs*)
Bases: *hexagonal.model.patron.Patron*

Faculty patron user type. All teaching body.

get_checkout_period_for(*document*)
Get checkout period for a specific document.

Parameters **document** – to be checked out

Returns timedelta

3.2 Document model

In SQL documents use joined-table inheritance, so id of any specific document is a foreign key to id of a document. All common fields are kept in the `documents` table, whereas fields unique for the specific type are held in a separate table.

class hexagonal.model.document.**Document**(***kwargs*)
Base class for all documents. Should not be instantiated directly.

Contains common fields for all documents, and inherits from *hexagonal.model.searchable.Searchable*, adding search capability.

Fuzzy search fields are *title* and *type*. Fuzzy array search fields are *keywords* and *authors*.

authors
Authors of this document.

available_copies
Hybrid property for currently available copies of this document. Available copies are copies which don't have an associated loan.

copies
Relation with copies of this document.

id
Integer primary key (referenced from subclasses)

keywords
Relation with keywords.

price
Document price (used in calculating overdue fine)

title
Document title (exists for all types)

type
Polymorphic identity column for inheritance support.

class hexagonal.model.document_copy.**DocumentCopy**(***kwargs*)
Copy of a document. References a specific document and document type by foreign key.

document
Associated document.

document_id
Foreign key to documents.

id
Integer primary key.

loan
Relation to current loan. May be None when document is available.

location
Location of the copy in the physical library.

class hexagonal.model.book.**Book**(***kwargs*)
Bases: *hexagonal.model.document.Document*

Book document type. These could be checked out for 2 weeks, if they are bestsellers. Otherwise, students check out these for 3 weeks, and faculty members check out these for 4 weeks.

bestseller
Whether this book is a bestseller.

edition
Book edition.

id
Integer primary foreign key to documents.

publisher
Relation with publisher.

publishment_year
Publishment year.

reference
Whether this book is a reference book

class hexagonal.model.journal_article.**JournalArticle**(***kwargs*)
Bases: *hexagonal.model.document.Document*

Journal article type of document. These could be checked out for two weeks by anyone.

id
Integer primary key.

issue_editor
Editor of the issue.

issue_publication_date
Publication date of the issue.

journal
Journal.

class hexagonal.model.av_material.**AVMaterial**(***kwargs*)
Bases: *hexagonal.model.document.Document*

AVMaterial document type. These could be checked out for 2 weeks by anyone.

id
Primary foreign key to documents.

3.3 Booking model

class hexagonal.model.loan.**Loan**(***kwargs*)
Model for one loan of a specific document by a specific user. Internal model, gets squashed in the api.

class Status
Loan status. Each loan can be:

- *requested* - which means that it has been requested by a patron.
- *approved* - which means that a librarian has approved the request, and the document is now in patron's possession
- *returned* - which means that the patron has supposedly brought the document into the library, and it is now waiting for approval from a librarian

can_be_renewed()
Flag for renew_document function. Gives information is renew option is it available to renew loan or not.

Returns whether the loan can be renewed.

document_copy
Loaned document_copy.

document_copy_id
Foreign key to document_copy.

due_date
Date when the document_copy must be returned.

get_overdue_fine()
Get total overdue fine for this loan. Returns 0 if it is not overdue.

Returns the overdue fine, in rubles.

static get_overdue_loan_count()
Get the amount of overdue loans.

Returns amount.

static get_overdue_loans()
Get overdue loans.

Returns list.

static get_requested_loan_count()
Get the amount of requested loans.

Returns amount.

static get_requested_loans()
Get requested loans.

Returns list.

static get_returned_loan_count()
Get the amount of returned loans.

Returns amount.

static get_returned_loans()
Get returned loans.

Returns list.

id
Integer primary key.

overdue()
Check whether this loan is overdue.

Returns whether this loan is overdue.

overdue_days()
Gives number of overdued days of loan.

Overdued or overdue? English is my second language. But Leonid Lyigin says that my eNgLiSh is finish <3

Returns number of days

static overdue_loan_query()
Get the query for overdue loans.

Returns query for overdue loans.

renew_document()
Allows user to renew his period of book checkout for one more period, without overduing the renewable document by old date

Returns new date, when book will become overdued

renewed
Whether this loan was renewed.

static requested_loan_query()
Get the query for requested loans.

Returns query for requested loans.

static returned_loan_query()
Get the query for returned loans.

Returns query for returned loans.

status
Current loan status.

user
Borrowing user id.

user_id
Foreign key to user.

3.4 Priority Queue

class hexagonal.model.queued_request.**QueuedRequest**(***kwargs*)

3.5 Search

class hexagonal.model.searchable.**Searchable**

Searchable base class. To add search functionality to an SQLAlchemy model class, just extend it from this class, and if you need fuzzy search specify *fuzzy_search_fields* and *fuzzy_array_search_fields*.

E.g.

```
class User(db.Model, Searchable):
    fuzzy_search_fields = ['name', 'role']
    fuzzy_array_search_fields = ['notes']

    name = db.Column(db.String, ...)
    role = db.Column(db.String, ...)

    notes = db.Column(db.ARRAY(db.String), ...)
```

To search by a specific field, use `search_by_<field>` (yeah, dynamic getattr)

E.g. (with the previous example)

```
user = User(...)
user.search_by_name('Alina')
```

will produce something like

```
SELECT * FROM users WHERE name ILIKE '%Alina%'
```

classmethod **_search_factory**(*field*)

Produce a function which searches the required field. (used later in __getattr__)

Parameters **field** – field in which the search will occur.

Returns searching function.

classmethod **_search_in_fields**(*term*, *fields=None*, *array_fields=None*)

Wrapper function which just returns all the records that match the query

classmethod **_search_in_fields_query**(*term*, *fields=None*, *array_fields=None*, *apply_to_query=None*)

Perform the actual search in specified fields.

Search is performed using SQL ILIKE, and all array fields are joined using a comma (‘,’). All clauses are OR’ed. E.g. `Searchable._search_in_fields('aba', ['title', 'description'], ['keywords'])` gives the following SQL:

```
SELECT * FROM whatever WHERE
    title ILIKE 'aba' OR
    description ILIKE 'aba' OR
    array_to_string(keywords, ",") ILIKE 'aba';
```

`array_to_string` is `sqlalchemy.func.array_to_string`.

`apply_to_query` can be used for chaining multiple logical searches.

```
first_clause = Model._search_in_fields_query('term', ['whatever'])
results = Model._search_in_fields_query('term2', ['whatever2'], apply_to_
↪query=first_clause).all()
```

Parameters

- **term** – term to search for.
- **fields** – usual text fields to perform search in.
- **array_fields** – array fields to perform search in.
- **apply_to_query** – chaining helper.

Returns list of results.

classmethod fuzzy_search(*term*)

Perform a search of the specified field through all defined fields.

Parameters **term** – term to search for.

Returns list of results.

INDEX

Symbols

A

B

C

D

E

F

G