MedImage

Design Doc

Contributors
Danny Sanchez
Victoria Gong
Calvin Li

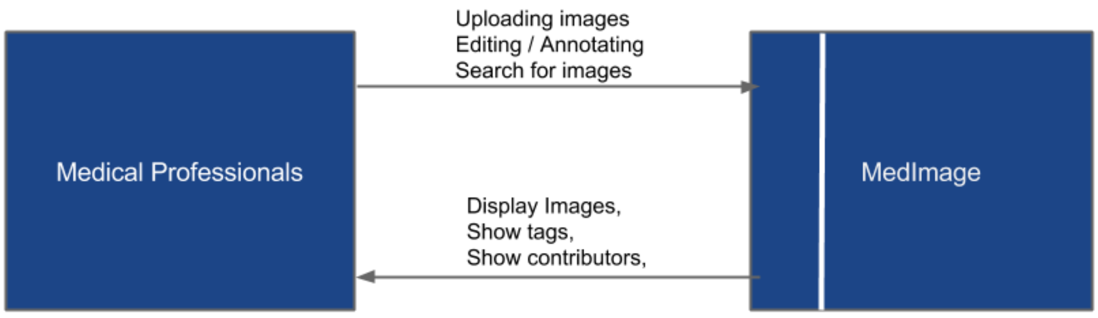
Purpose

1. Allow medical professionals to include important data on medical images
   - Medical professionals want to take notes on specific portion(s) of their medical images denoting certain characteristics or problems. Consequently, they can look at the image in the future and have a clearer understanding of their notes related to the image.

2. Allow medical professionals to share collective knowledge
   - Medical professionals want to collectively work together in sharing their knowledge when analyzing an image, allowing for more collaboration and teamwork.

3. Allow medical professionals to find relevant medical images
   - Medical professionals want to find related medical images to their desire searches so they can compare images and draw conclusions from those comparisons.

Context



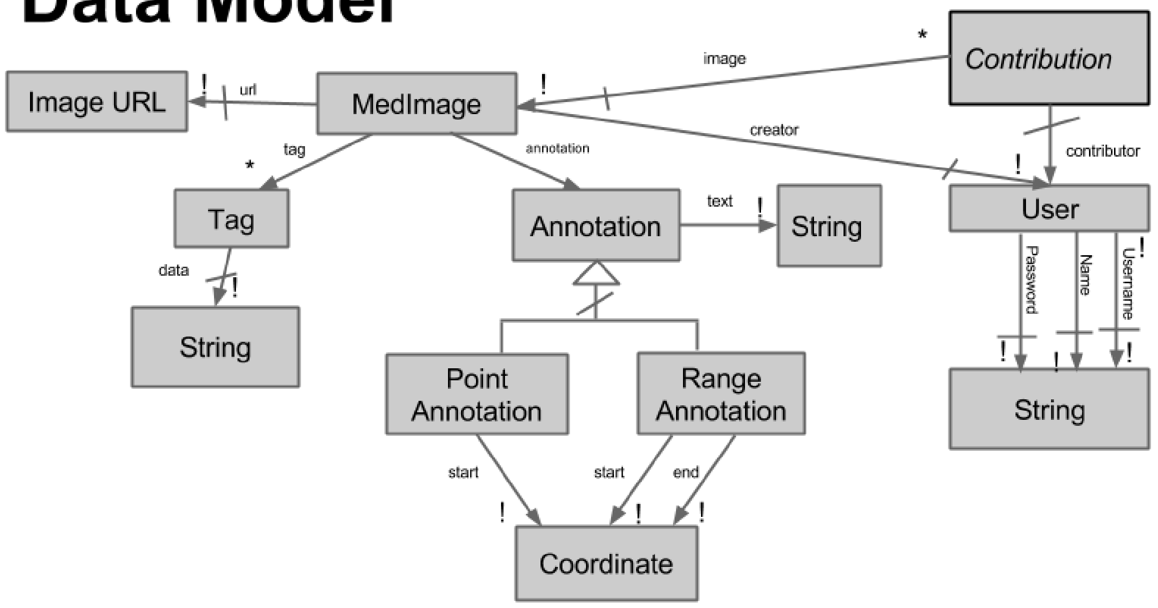
In our project, there will now only be one group interacting with our program: medical professionals. The people will be the ones uploading images, editing them and annotating them. Although one might argue that medical assistants may not edit other doctor's images necessarily, the same is possibly true for doctors outside the collaboration scope. In essence, both groups are still medical professionals. Editing and image rights are instead left to the design of the program.

Concepts

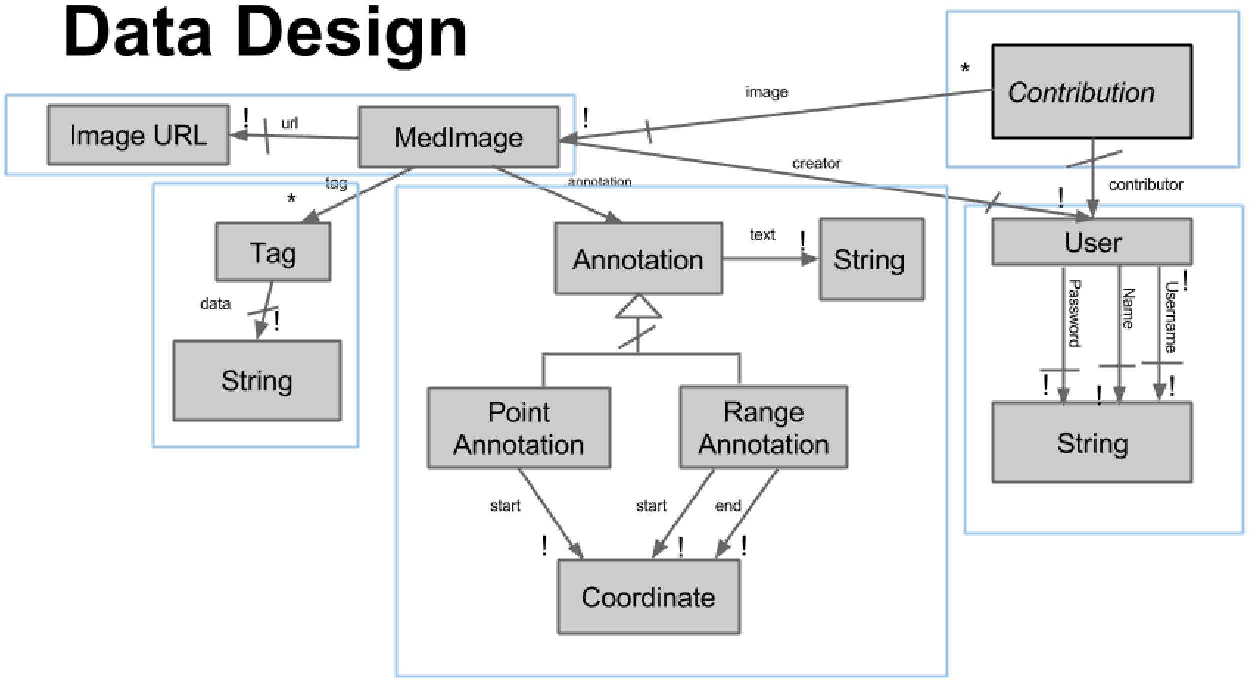
1. Annotations – writing notes on a portion of the image (purpose 1)
2. Tagging – placing a keyword describing a detail of the image (purpose 1,2,3)
3. Images – the medical images (purpose 2)

Design

Data Model



Data Design



Design Justifications

MedImage: Tags, Annotations, Contributions – being relational vs. embedded

- **Tags**
  - Need: A search feature that would be mainly dependent on the tags associated with each image. Tags to be displayed on each associated image.
  - Options: MedImage contain a list of embedded Tags vs. A separate collection just holding Tag objects containing an ImageMed id and the associated tag name. Thus, either the tag or the image can exist in multiple objects in the Tag collection.
  - Result: A separate Tag collection (relational)
  - Thought: If we have MedImage contain a list of Tags, we would have to search through each MedImage and then search through their respective tag list in order to find all the images with the tags given. We felt this was much less efficient than having Tag be its own Collection where it contains a tag and its associated image id. This is because MongoDB cannot search through nested data structures as well as non- nested. Also, since the Tag is structured such that it contains only two pieces of information – the tag and the image id, in order to retrieve a list of Images associated with a tag we only need to query for Tags where the tag_name matches one of the ones given. Retrieving the tags associated with an image is similar.

- **Annotations**:
  - Need: We wanted to have two different types of Annotations (Point and Range, where the former represents a point reference and a range represents a space reference). Annotations needed to be displayed per each image.
  - Choices: Embedded list of Annotations per ImageMed or a relational version where Annotation is a separate Collection.
  - Result: Annotation as a separate collection.
  - Explanation: The first reason was that a relational database would allow for subclassing. Having hierarchy would allow changes to either class or both classes to be much easier and cleaner. Secondly, we wanted to leave the possibility of having search also be based on annotation text for the future. (As explained above, having an embedded list is not optimal for searching). Meanwhile, retrieving the annotations associated with an image is very easy -- it is a simple query given a defined image id parameter.

- **Contributions:**
  - Need: (1) Knowing if a user has editing rights to an image, (2) Displaying a list of contributors for each image, (3) Adding and removing contributors to an image.
  - Choices: Having each MedImage have a list of contributors, or have Contribution be a separate collection where each object only contains the image id and a single contributor.
  - Result: Contributions as a separate collection
  - Explanation:
    - For 1) an embedded model would require us to retrieve the medical image object and then go down another level to the list of contributors and search for the name there. For a relational model, we just have to check if an object with that image id and that user id exists, which is a simple upper-layer query.
    - For 2) an embedded model would be very simple as we just have to retrieve the list of contributors and display it. However, the relational model is not complicated either as the retrieval just requires querying the Contributor collection for objects with image_id = the given image id. Although the former is slightly faster, they both run pretty fast.
    - For 3) Removing contributors is similar to searching (i.e. 1) as we would have to first find the object in the embedded version and then remove it and thus it is more efficient and easier to use relational. For

adding, it is slightly faster than removing for embedded because we do not need to look through the list of Contributors inside medical image, but we do need to append to the end of the list. For relational, however, it is a simple creation of a Contributor object and just saving it to the collection.

Thus, we have chosen relational for med image because the benefits for relational outweigh the benefits for embedded. Note: we didn't consider the option of having ImageMed have a list of users that has contributed and User have a list of images it contributes to because multiple copies of the same information leaves room for data inconsistency (such as the case where one fails to update).

Design Challenges

Challenge 1: Figuring out how to represent the various users and their rights to view and edit photos.

Initially, we first thought that there might be two kinds of users, Medical Professionals with M.D (like doctors, surgeons) and Medical assistants (nurses, etc.). The M.D. would give certain users a signal if that person was 'experienced' enough to be annotating medical images. The Medical assistants, on the other hand, might not be given permission, but would still be allowed to view and search. However, we came across a huge problem: Certifying that a doctor was indeed a doctor and the degree was real. To counter this problem, we thought back to the core problem: We wanted to differentiate between M.D. and non-M.D. in order to help verify if a user has authority to annotate and edit medical images. We realized that there is a much simpler goal we were trying to solve: Who exactly can annotate and edit medical images. To solve this, we came up with the idea of a "Contributor." In essence, the creator of an image had all rights to dictate who can edit their images. Each user can add the contributors and most likely that user will probably know the other user and know if they are trustworthy or not. Moreover, the concept of contributors can better guarantee that a random person isn't editing the annotations in a way that the creator did not intend for. Thus, we came up with putting the access rights of an image into the creators' hands.

Challenge 2: Search (populate vs. aggregate)

For search, I wanted to be able to display a list of photos ordered by relevance to the given tag names. In order to implement this there were two choices:

1. Do a query for the tag objects that match one of the tag names, and use populate to retrieve the image url's at the same time. This would return a list of json objects which contain the tag name, the image id and the image url. In order to order it by relevance, I would have to go through each of the objects and group the items by the image id and then sort by size of group.
2. Run an aggregate function on Tag which allows me to query for all tags that matches the tag names given, group them by image id's and also sort by size of group all within the function. This would return a sorted list of json objects which contains the photo id, , list of tags matched, number of tags matched and the image id. I would need to access Image collection in order to get the image url.

Result: Implemented option (2)

- **Reasoning:** Benefits of 1 is that there is one call to the database, but there would be post-processing of grouping and sorting. Benefit of 2 is that there will be 2 calls to database and less post- processing of grouping and sorting. We felt that it was easier that we did not have to post-process and sort by relevance ourselves and just have one extra database call. Moreover, the database often makes sorting most efficient.

Challenge 3: Representing data on an image
There are numerous ways you can represent data on an image. For example, drawings, text, or even a single word.

In order to represent this the most simply, we decided that tags would represent higher topics that can be searched on an image, while annotations to represent direct markings on an image. Annotations would be split into point and range so that text can be associated with specific areas of the image.

API Specification

Response Models

Point annotation

Key	Format	Description
text	String	Text in the annotation
image_id	ObjectId	_id of the associated MedImage
start_point	JSON	{x: Number, y: Number} – coordinate of annotation on image

Range annotation

Key	Format	Description
text	String	Text in the annotation
image_id	ObjectId	_id of the associated MedImage
start_point	JSON	{x: Number, y: Number} - coordinate of annotation on image
end_point	JSON	{x: Number, y: Number} - coordinate of end point of annotation

Contribution

Key	Format	Description
image_id	ObjectId	_id of the associated MedImage
user_id	ObjectId	_id of the associated MedImage

Contribution Access

Key	Format	Description
has_access	Boolean	Whether the user has access to edit the MedImage
contribution_id	ObjectId	_id of the associated contribution

MedImage

Key	Format	Description
_creator	ObjectId	_id of user who created MedImage
title	String	Title of image to display
Image_url	String	url of actual image file

MedImage Response

Key	Format	Description
title	String	Title of image to display
image_url	String	url of actual image file

User Model

Key	Format	Description
_id	ObjectId	Default Mongoose id
first_name	String	First name of user
last_name	String	Last name of user
username	String	Username of user
password	String	Password of user
v	versionKey	Default Mongoose version key

Tag

Key	Format	Description
_image	ObjectId	_id of image to which this tag is applied
tag_name	String	Name of tag to display

Default Error

Key	Format	Description
status	Number	HTTP status code of error
name	String	Name of error
message	String	Description of error

Mongoose Error: Defined by Mongoose

None: Empty JSON

API Endpoints

Note: All endpoints can have a Mongoose Failure, which always returns a Mongoose Error response model. All other errors will return the Default Error response model. The responses are listed below. **Failure** is simply the list of reasons that a failure could occur, and each returns a Default Error.

Annotations

- GET /medimages/{image_id}/annotations/
  - **Description:** Gets all the annotations of the medical image with the given image_id
  - **Response:**
    - **Success:** Array[Point Annotation, Range Annotation]
    - **Failure:**
      - Invalid Id

- POST /annotations
  - **Description:** Creates a new annotation
  - **Parameters:**
    - **text***(Required):* Text to display in annotation
    - **image_id***(Required):* _id of image where annotation was created
    - **type***(Required):* "point" || "range", designates the type of annotation to be created
    - **start_point***(Required):* Coordinates of start point of annotation
    - **end_point***(Required if type == range):* Coordinates of endpoint of annotation
  - **Response:**
    - **Success:** None
    - **Failure:**
      - Invalid image_id
      - type not given

- PUT /annotations/{annotation_id}
  - **Description:** Edits the annotation with the given _id
  - **Parameters:**
    - **type** Required: "point" || "range", designates type of annotation to edit
    - **text** Optional: Text to display in annotation
    - **start_point** Optional: Coordinates of start point of annotation
    - **end_point** Optional: Coordinates of end point of annotation
  - **Response:**
    - **Success:** None
    - **Failure:**
      - Invalid annotation _id
      - type not given

- DELETE /annotations/{annotation_id}
  - **Description:** Deletes the annotation with the given _id
  - **Parameters:**
    - **Type**: Required: "point" || "range", designates type of annotation to delete
  - **Response:**
    - **Success:** None
    - **Failure**:
      - Invalid annotation _id
      - type not given

Contributions

- GET /contributions/access
    - **Description:** Sees if user has access to edit medical image with the given id
    - **Parameters:**
        - **username** Required: username of User to check
        - **image_id** Required: _id of MedImage to check
    - **Response:**
    - **Success:** Contribution Access
    - **Failure:**
        - User does not exist
        - Invalid image_id

- POST /contributions
    - **Description:** Adds user to collaboration on medical image with given id
    - **Parameters:**
        - **username:** Required: username of User to add
        - **image_id:** Required: _id of MedImage to add user to
    - **Response:**
        - **Success:**
            - **Contribution exists:** None
            - **Contribution does not already exist:** Contribution
        - **Failure:**
            - Invalid image_id
            - image does not exist
            - user does not exist

- DELETE /contributions/{contribution_id}
    - **Description:** Deletes contribution with given _id
    - **Parameters:** None
    - **Response:**
        - **Success:** None
        - **Failure:**
            - Invalid contribution_id

MedImages

- GET /users/{username }/medimages
    - **Description:** Gets the MedImages for a user
    - **Parameters:** None
    - **Response:**
        - **Success:** Array[MedImage]
        - **Failure:**
            - User doesn’t exist

- POST /medimages
    - **Description:** Creates a MedImage
    - **Parameters:**
        - **username** Required: username of user who created image
        - **title** Required: Title to display for MedImage
        - **medImage** Required: Actual image file to created MedImage from
    - **Response:**
        - **Success:** MedImage Response
        - **Failure:**
            - File type not jpeg or png
            - User doesn’t exist

            - title not non empty

- PUT */medimages/{image_id}
    - **Description:** Edits a MedImage
    - **Parameters:**
        - **Title** Required: Title to display for MedImage
    - **Response:**
        - **Success:** None
        - **Failure:**
            - Invalid MedImage _id
            - Invalid title (empty or whitespace)M
            - MedImage not found
- DELETE medimages/{image_id}
    - **Description:** Deletes a MedImage
    - **Parameters:** None
    - **Response:**
        - **Success:**None
        - **Failure:**
            - Invalid image _id
            - Medimage does not exist