URL: TODO

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 uesr has access to edit the MedImage
contribution_id	ObjectId	_id of 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 listed below **Failure**are simply the 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 end

point 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:**
        - **user_id** *Required:* _id of User to check
        - **image_id** *Required:* _id of MedImage to check
    - **Response:**
        - **Success:**Contribution Access
        - **Failure:**
            - Invalid user_id
            - Invalid image_id
- POST */contributions*
    - **Description:**Adds user to collaboration on medical image with given id
    - **Parameters:**
        - **user_id:** *Required:* _id 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 user_id
            - 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/{user_id}/medimages*
    - **Description:**Gets the MedImages for a user
    - **Parameters:**None
    - **Response:**
        - **Success:**Array[MedImage]
        - **Failure:**
            - Invalid user _id
- POST */medimages*
    - **Description:**Creates a MedImage
    - **Parameters:**
        - **user_id** *Required:* _id 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
            - Invalid user_id
            - title not non empty
            - Given user does not exist
- 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)

        - MedImage not found
- DELETE *medimages/{image_id}*
    - **Description:**Deletes a MedImage
    - **Parameters:**None
    - **Response:**
        - **Success:**None
        - **Failure:**
            - Invalid image _id
            - Medimage does not exist

Tags

- GET */tag/{image_id}*
    - **Description:**Get all the tags of the medical image with the given id
    - **Parameters:**None
    - **Response:**
        - **Success:**Array[Tag]
        - **Failure:**
            - Invalid image _id
- POST */tag/{image_id}*
    - **Description:**Create or add tag for photo with given id
    - **Parameters:**
        - **tag** *Required:* Name of tag
    - **Response:**
        - **Success:**None
        - **Failure:**
            - Invalid image _id
            - Tag already exists for the image
- DELETE */tag/{image_id}*
    - **Description:**Remove tag off of photo with given id
    - **Parameters:**
        - **tag** *Required:* Name of tag
    - **Response:**
        - **Success:**None
        - **Failure:**
            - Invalid image _id

Uploads

- GET */uploads/images/{user_id}/{image_name}*
    - **Description:**Gets an image file for a MedImage based off the url in MedImage
    - **Parameters:**None
    - **Response:**
        - **Success:**Image File
        - **Failure:**
            - Invalid Image Request (type of image not png or jpg)
            - Image not found

        - App environment invalid

Users

- GET */users*
    - **Description:**Gets all existing users
    - **Parameters:**None
    - **Response:**
        - **Success:**Array[User]
        - **Failure:**Mongoose error only
- POST */users*
    - **Description:**Creates a new user
    - **Parameters:**
        - **first_name** *Required:* First name of user
        - **last_name** *Required:* Last name of user
        - **username** *Required:* Username of user
        - **password** *Required:* Account password
    - **Response:**
        - **Success:**None
        - **Failure:**
            - One of the parameters was empty
            - User already exists
- PUT */users/{username}*
    - **Description:**Edit an existing user
    - **Parameters:**
        - **first_name:** *Optional:* First name of user
        - **last_name:** *Optional:* Last name of user
        - **password:** *Optional:* Account password
    - **Response:**
        - **Success:**None
        - **Failure:**
            - User could not be found

Design

Design Challenges

- Challenge 1: Figuring out how to represent the various users and their rights to view and edit photos. → result: “Contributions”
- 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:**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.
- Challenge 3: Representing data on an image → Result: we decided to go with tags representing higher goals that can be searched on an image and annotation represent direct markings on an image.

Design Justifications

- **MedImage being relational vs. embedded.**

  Tags:we definitely wanted tags relational because we wanted the search feature to be mainly dependent on the tags associated with each image. 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. Thus, either the tag or the image can exist in multiple tags in the collection. This allows us to only have to query for Tags where the tag_name matches one of the ones given. Retrieving the tags associated with an image is also very simple. Annotations: There were several reasons we chose relational. For one, it allowed us to do subclassing. 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) and 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.

- **Contribution: relational vs. embedded**

  Our two choices we were debating between was 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. We considered two actions that we wanted to do with contributors in deciding which:

    - 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.

- 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).
