Radiographic Hardware Identification in Distal Radius Surgery

Detailed Design

JID Team 1332

Bharat Goyal, Blake Sanie, Dylan Mace,

Ethan Wells, Jessie Everett

Date of Submission: 04/17/2022

Client: Dr. William Runge, M.D.

GitHub Repository: Link

Table of Contents

List of Figures

Terminology

Application Programming Interface (API)

A developer-friendly interface with consistently documented functions and specifications for a program/service.

Convolutional Neural Network (CNN)

A type of machine learning model well-suited for image classification tasks. The convolutional layers of the model operate by applying a filter across the input data to draw out trends or outliers in the data.

CoreML

A framework for developing machine learning models for Apple devices, including iPhones, iPads, and Macs.

Distal Radius Implant

Medical devices (metal plates) that are implanted into the forearm because of a bone break or fracture, appearing at the head of the wrist.

FastAPI

Python framework for creating organized web server APIs to be accessed over the internet.

GitHub

An industry-standard cloud-based version control system that stores source code, enables concurrent work through branching, and merging of the work of collaborating members.

Heroku

An online cloud service platform based on Amazon Web Services where our API is hosted.

JSON

JavaScript Object Notation: a standard data/file type used to encode information about hierarchical objects in a human-interpretable manner.

Machine Learning (ML)

An industry standard problem-solving paradigm that involves training a model to map inputs to outputs from a known dataset, then applying the model to unseen inputs to solve unseen problems.

OpenCV

An industry standard computer vision library for Python with common image manipulation operation.

Python

A programming language that in our case is used to develop ML and CV models, and our web-server API.

Swift

A programming language for developing iOS applications.

Teachable Machine

A framework for creating machine learning networks through the cloud, built by Google. The project acts as a wrapper GUI for TensorFlow.

TensorFlow

An industry standard ML library for Python developed and maintained by Google.

1. Introduction

1.1. Background

Distal radius fractures are a common occurrence for orthopedic surgeons. As such, they have clear, common treatment plans. However, the problem arises when patients need to have their implant removed by an office which is separate from that which installed the implant, as no universal implant attachment hardware exists for distal radius implants. Implants are made by a host of companies, all with differing screws and bolts used for attaching the implant. Due to the abundance of implant models, there is simply no way surgeons can always accurately identify different types of implant models from an x-ray. In the event that the implant manufacturer cannot be identified and still requires immediate removal, surgeons must make do with existing tools via unconventional and crude means.

Our team has created an iOS application which will (from a photo of the patient x-ray) identify the manufacturer and model of the implant, allowing the surgeon to enter the operating room with the correct tools. In designing our application, we first analysed our user base: doctors and others in the medical field. We determined our application needed to meet three main requirements:

Speed: our application needs to make predictions quickly to seamlessly integrate into the surgeon's workflow.

Reliability: our application needs to predict the correct manufacturer, otherwise it is useless.

Security: to ensure the usefulness of our application, all data needs to be secured.

Our application is implemented in three main parts: an iOS frontend client implemented in Swift, a backend API written using the FastAPI wrapper and Python, and a prediction model written in Python as well. There are a few avenues we're testing for the model which involve CoreML, TensorFlow, and Teachable Machine. The main portions of our app include an onboarding/setup flow, a camera capture and pre-processing page, a results page, and a detailed breakdown of each company's main implants and required tools.

1.2. Summary

This report documents the Radiographic Hardware Identification in Distal Radius Surgery project in five main parts:

The *System Architecture* section describes the static and dynamic elements of our system's iOS application, API, and ML model, as well as how these portions interact with each other within the system.

The *Data Storage Design* section described the structure of our (lack of) databases, due to the nature of our application storing nearly zero data to fulfil privacy requirements. However, a detailed analysis of how our static API data is stored is included here.

The *Component Design Detail* section details the relationships between the client, API, and model, and how the different API calls and data classes interact with each other to form a cohesive system.

The *UI Design* section presents the current state of the five major UI screens that our users will interact with, and the changes we plan to implement as we continue into later sprints. The states of these pages will be reflected by video/photo examples.

The *Appendix* section details additional information including but not limited to a complete API specification sheet including response codes, JSON return structures, etc.

2. System Architecture

2.1. Introduction

This application is defined by our iOS application, our machine learning model, and our Heroku server. We have provided two diagrams, a static design diagram and a dynamic system architecture diagram. The static system architecture diagram will provide an overview of how the different components of our application interact from the developer's side, and the dynamic system architecture diagram will detail how a user of the application interacts with the interface and API.

2.2. Static System Architecture

As noted in the introduction, our application architecture can be split into two main portions: a client-facing iOS application and a Heroku server that hosts our API and is connected to our machine learning model.

Inside our iOS application, the general flow is as follows:

1. User interacts with the UI to execute a specific action (such as clicking a button)
2. The UI interacts with the API Calling Service, native iOS data components, or third-party libraries to fetch the required info
3. The UI is updated to display the new information to the user

2.2.1. Local Storage

Diving deeper into the application flow, local key/value storage (known on Apple devices as UserDefaults or AppStorage) is utilized to store certain application milestone markers, such as whether a user has completed the tutorial stage of the application. This local storage persists across app launches and can be used as a "check" of sorts when displaying certain UI elements. Additionally, the permission authorization system built into all Apple devices is utilized to ensure user consent before performing certain operations. In our case, we utilize this service to determine whether users allow camera capture from our application.

2.2.2. Third-Party Frameworks

To avoid re-inventing the wheel, we have integrated two third-party libraries into our codebase. They were behind The BetterSafariView library is utilized to integrate native modal Safari popup views into a SwiftUI application (NOTE: this was solved in iOS 15.0+ devices, however we have integrated this to ensure backwards compatibility for iOS 14 devices). We have also integrated SlidingRuler into our application to beautify the rotation engine, allowing us to match the native iOS photo editing experience.

2.2.3. Implant Classification

To perform classification, the iOS application relies on Apple's CoreML framework. This callable client-side module instantiates the trained model and performs the classification task. Within our code, we can collect the image captured by the user, compute its classification result via CoreML, and display results within the UI itself.

2.2.4. API and Server Interaction

To retrieve dynamic information at the request of the user, the iOS application interacts with the Heroku server over an established internet connection. This resource is used to obtain company information stored off-device. Once the server receives the request, a Unicorn load balance routes the request to one of several concurrent FastAPI instances. The selected instance then further routes the request to the corresponding endpoint. The endpoints either load a memory from static storage, in the case of /implantExamples/images, or queries the PostgreSQL database to extract company and implant data. The response is converted into a JSON-formatted response, and the endpoint directly communicates the response back to the API Calling Service within the iOS application. Lastly, the application converts the response back into swift datatypes so that its contents can be interpreted as needed by the user interface and its backing logic.

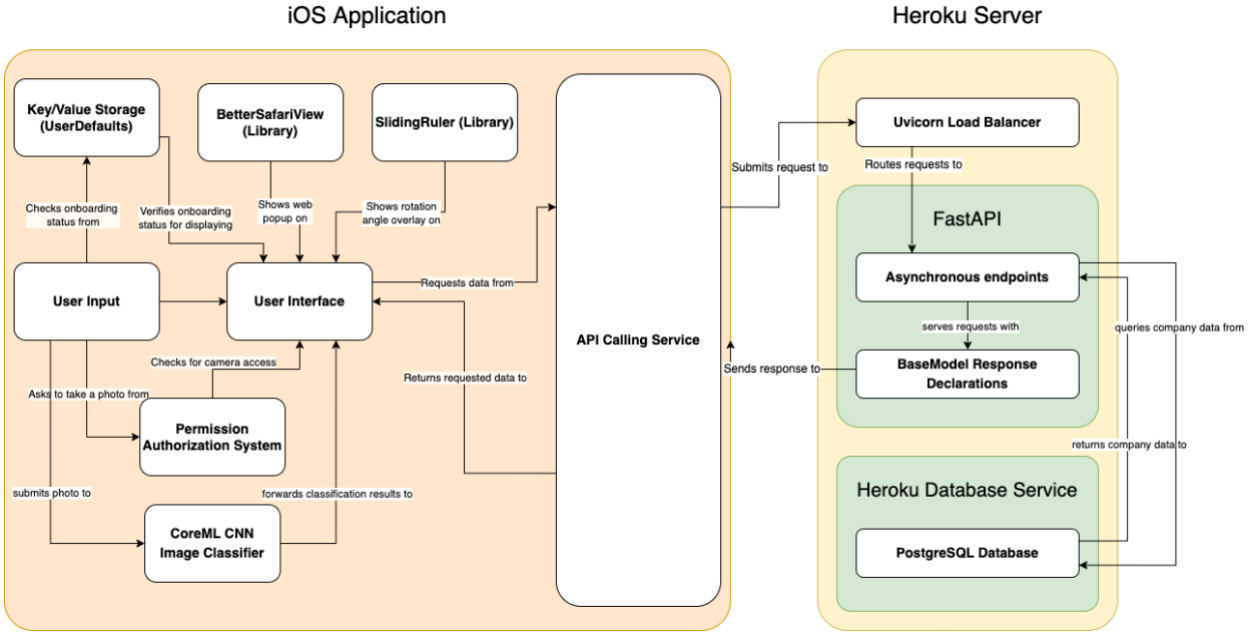

Figure 2.1 - Static System Architecture Diagram

2.3. Dynamic System Architecture

Our application flow begins with the user opening the app. At this point, UserDefaults is checked to determine the onboarding status of the user. If they have not completed the onboarding process, then the user must walk through the process. When they have completed the process, the user will be redirected to the home screen. At the same time, the onboarding key is updated to reflect that the user has been shown the sequence.

When the user chooses to classify a new implant, they are directed to take a photo. At this point, the UI retrieves the camera authorization status from the user. If the status has not been set, the user is prompted to accept the camera permission popup. From here, the user can take a photo of an x-ray and rotate/crop it until it is satisfactory.

Once the user is satisfied with their image, they can submit the image. The application passes this image to the CoreML service, which propagates the image pixels through the instantiated CNN. CoreML then formats the classification results for consumption by the results screen within the UI.

Now that the results have been determined, the application can fetch the example images for the predicted company. An API endpoint, hosted on Heroku, is queried from the calling service. Once the images are returned, they are displayed on-screen.

When the user selects one of the companies from the detail list, information about that company's main implants is requested from the server via the API calling service. The server retrieves the information from the PostgreSQL database and transmits the response back to the application, where it is displayed onscreen. When clicking on the technique guide associated with one of the fetched implant examples, the BetterSafariView library is utilized to display a modal view with the technique guide PDF.

We would like to specifically note that we have left the SlidingRuler and BetterSafariView libraries out of the dynamic system diagram as they are functionally part of the user interface.

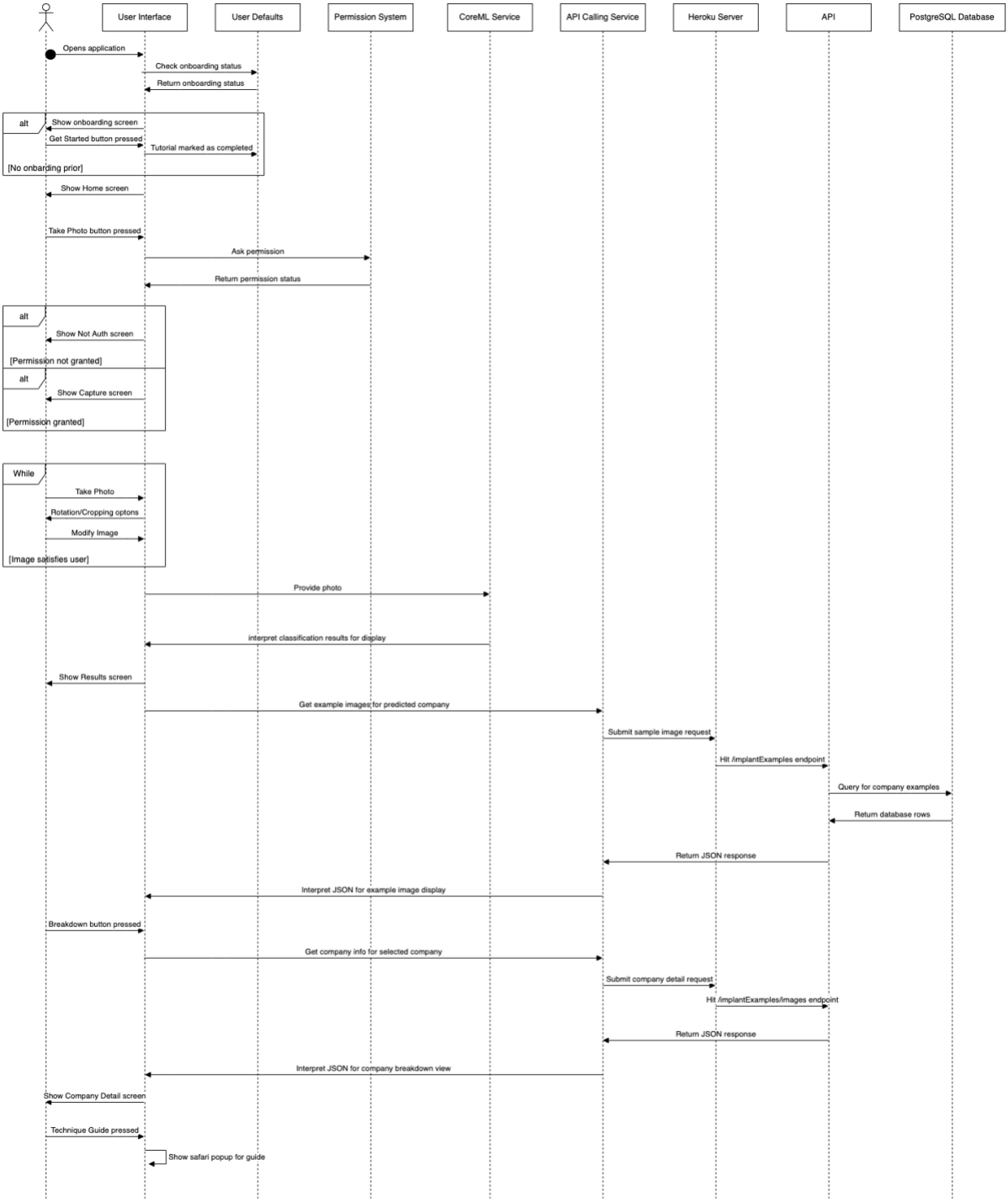

Figure 2.2 - Dynamic System Architecture Diagram

2.4. Local Environment Information

Aside from the iOS application, Heroku server and their corresponding entities, we must outline a few key independent processes that drive the lifecycle of the system. These systems do not actively support the end-user experience but provide the backbone necessary for the system upon initial start-up.

2.4.1. Image Preprocessing

Firstly, creation and training of the model occurs in a local environment. More specifically, the developer pulls the latest model source code from the shared GitHub repository and loads in an image dataset (organized with subfolders to represent image classes). With these artifacts now saved to the local filesystem, the image preprocessing pipeline is then applied to the images. The pipeline consists of the following steps: image reading from the filesystem, color space inversion if plate shows as black on x-ray scan, thresholding out unnecessary detail, edge detection, contour formation, contour straightening, and finally, cropping to the detected object of interest. Once the set of transformations are applied, the new images are saved to the *processed_images* folder and may also be pushed to a remote storage service for version control if chosen.

2.4.2. Model Development and Training

Machine learning model development occurs in the same environment. The local dataset of cleaned images is split into training, testing, and validation sets. The Python-based Convolutional Neural Network model is then trained to find optimal correspondence between the image input and manufacturer label output. Once trained, the model is converted to an iOS .mlpackage file to be pushed to the remote repository. This allows the model to be fine-tuned in the future.

2.4.3. API Redeployments

The remote GitHub repository, in addition to managing source code versions, plays an important role in propagating changes initiated from the local environment to the server instance. GitHub has an active version observer on the main branch; when a new version is pushed, the Heroku server and its FastAPI contents are redeployed. In other words, whenever a new change is pushed to the repository, the server is updated using the most recent source code. This process is automated for system reliability and developer experience purposes.

3. Data Storage Design

3.1. Database Use

Company, implant, and tool information is stored and modelled through the relational paradigm using a PostgreSQL database hosted on Heroku. This database is protected by a randomly generated username/password credential pair; these credentials are read in through the API's environment, meaning credentials are never committed to shared repositories. All information in the database is publicly available, and credentials are automatically regenerated every six months to further reinforce security.

Each table corresponds to its proper entity and holds only the fields essential to describing such entity. Fields can be combined are aggregated by performing join queries, using the SQL querying language.

Below is our Entity Relationship Diagram (ERD) showcasing these entity schema and relationships:

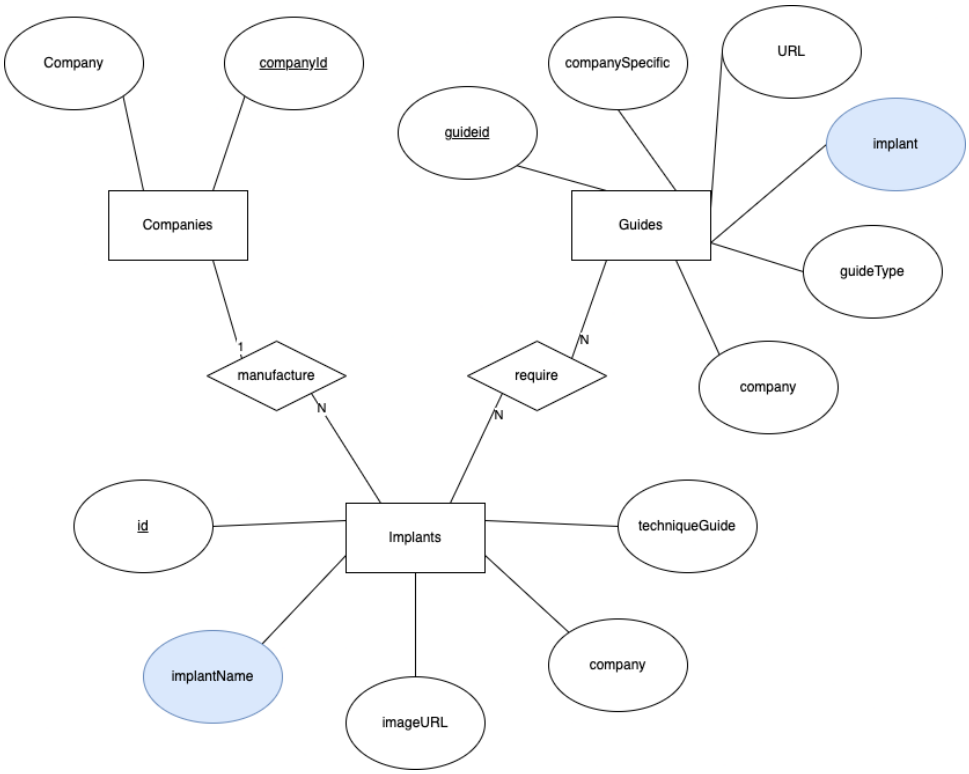

Figure 3.1 – Entity Relationship Diagram (ERD)

3.2. File Use

All x-ray images are stored statically on Heroku and accessed via our API. Any images captured and stored client-side are only stored temporarily, until the client's session ends.

3.3. Data Exchange

The images the user wants to submit for classification are sent to the server using HTTPS. The classification information computed by the model is formatted as a JSON file which is not stored in the backend (i.e., as soon as it's generated it's transmitted to the iOS client). Additional information regarding the tools suggested by the manufacturer and the sample image are retrieved from the database (which is also hosted on the Heroku server) through the the corresponding API endpoint.

Currently, this application does not involve user authentication and session management, though this feature may be included in future versions.

4. Component Design

4.1. Dynamic Component Design

Our robustness diagram shows the kind of interactions that occur between the user and the interfaces/control objects in our project. When the user opens the application for the first time, they are walked through the tutorial/onboarding sequence. This is a series of screens that are interlinked without the need for a designated controller.

Once that is completed, the view router control object directs the user to the home page. The user has the option to view the results for X-rays that were classified in the same instance of the application (these results are deleted every time the instance is closed). The view router object handles this functionality based on user inputs. In addition, the user has the option at any point during the flow of the application to go back to the onboarding sequence. This transition is also handled by the ViewRouter object.

In case the user wishes to classify a new image, they can select the corresponding option on the home screen which prompts the ViewRouter to change the interface to the capture screen. From here on, once the user clicks a satisfactory image, the classification controller transmits the information to the model API which then accesses the locally stored classification model to get the results. Thereafter, the ViewRouter updates the interface to the company information screen where the classification controller uses the company information API to retrieve information from the Postgres database. In addition to the instructions provided by the manufacturers, the database also stores example images so that the user can draw visual similarities between their X-ray and samples to gain confidence in the classification.

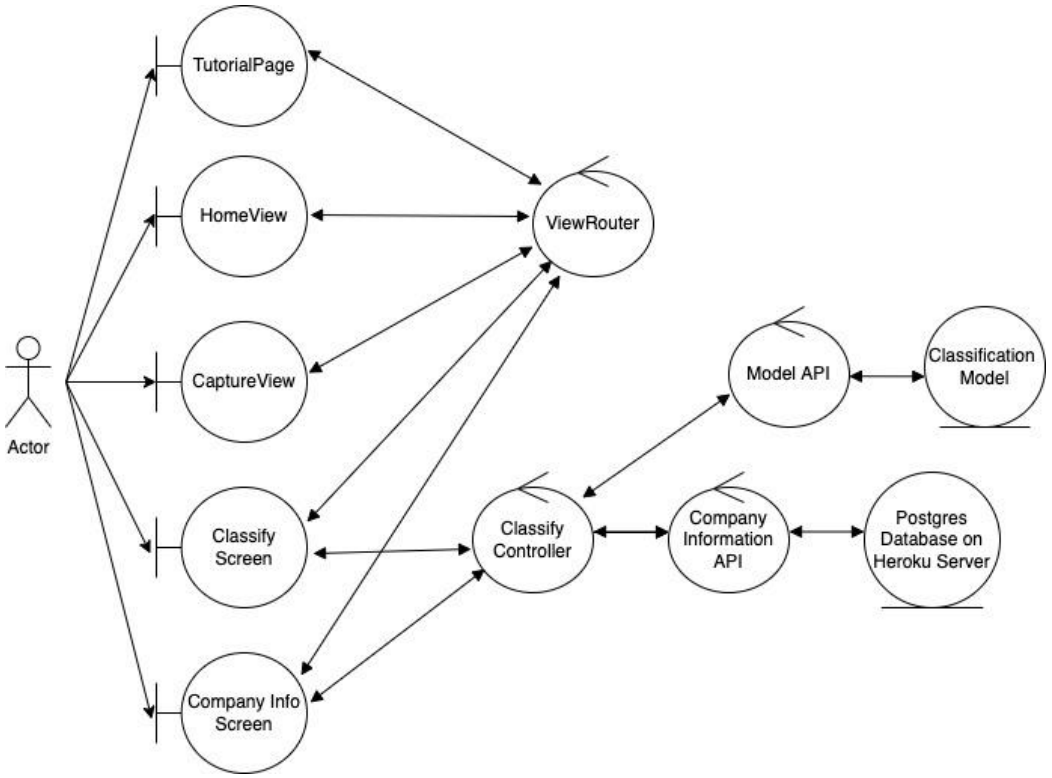

Figure 4.1 – Dynamic Robustness Diagram

4.2. Static Component Design

Our component design class diagram represents the static components of our application system. Our system architecture consists of two main sections: an iOS client, represented in the yellow section, and a web server, represented in the green section.

4.2.1. Web Service

The web server can be split into two main endpoints: fetching implant examples and fetching example images. Both endpoints internally fetch information about the plants released by the queried company, populated in our Heroku SQL database. The database design section above explains the tables, entities, and relationships encapsulated by Heroku.

The first endpoint returns all available information about all available implant classes manufactured by the queried company. This endpoint fetches all available information, and returns this information formatted as a list of ExampleImplant objects. Each ExampleImplant object contains an implant name, a URL for an image of that implant, a URL to the technique guide for that implant, and a list of required tools (formatted as RequiredTool objects), with each tool containing a tool name and a URL to an image of the tool.

The second endpoint just returns URLs for implant images from the queried company. These are formatted as a list of ImplantImage objects, where each object contains the name of an implant and its corresponding image URL.

4.2.2. iOS Application

Before the details of our application flow are explained, we would like to explain the general structure of our application. Each segment of the application contains a view and view model. The view displays elements onscreen, and the view model fetches data and connects components. If elements need to be fetched from the machine learning model or the API, it is completed in the view model and returned to the view to be displayed. Data returned from the API, or the classifier service is converted from JSON into Swift structs, handled internally by the APIService or the ClassifierService.

Diving into the iOS Application, the outermost layer consists of a ViewRouter object. This acts as a wrapper dedicated to managing which application page is onscreen. There are two main flows in the application: the tutorial flow and the classify flow.

The tutorial screens are contained in a TutorialView objects, which is a tab view containing the 5 tutorial pages. This object has a reference to the ViewRouter object to push itself off the navigation stack once the tutorial has been completed. It also contains a tabSelection variable which manages which page index the tab view displays.

After the tutorial is completed, the user is moved to the HomeView. This view displays a list of previous classifications, fetched from the ClassificationModel. From here, a user can move to capture a photo for classification; this occurs in the CaptureView. The capture view contains a CameraFrameViewModel, which is responsible for managing the camera popup which occurs when the user to selects a photo. This photo is passed back to the view for displaying, after which the user can use built-in controls to resize and crop the captured image to a square.

Once the user is satisfied with their classification, they can progress to the results screen. This screen accepts the user image as a parameter and queries the ClassificationService for prediction through the predict(image:) function. If an error is returned, the results view displays the error. If a successful classification is returned, it is stored in the ClassificationModel classifications variable and displayed onscreen. Each classification contains the predicted company, the predicted likelihood, and the breakdown of each company and the corresponding percentage. Once the classification is retuned, the ResultsView fetches all example images from the predicted company through the APIService and displays it through the ExampleImagePageView.

To gain additional insights about company implants, users can click on each company in the breakdown list to view more information. This causes a segue to the CompanyDetailView. This view contains a link to the CompanyDetailViewModel, which fetches all implant information as soon as the view is called. This returns a list of ExampleImplant objects, each of which contains an implant name, a link to an implant image, a technique guide url, and a list of tools. The user can expand and collapse the information for each variety of implant fetched through the getImplantExamples() function, with the images displayed at the top of the screen in a new ExampleImagePageView. If the user chooses to view the technique guide, a Safari popup appears displaying the contents of the technique guide URL.

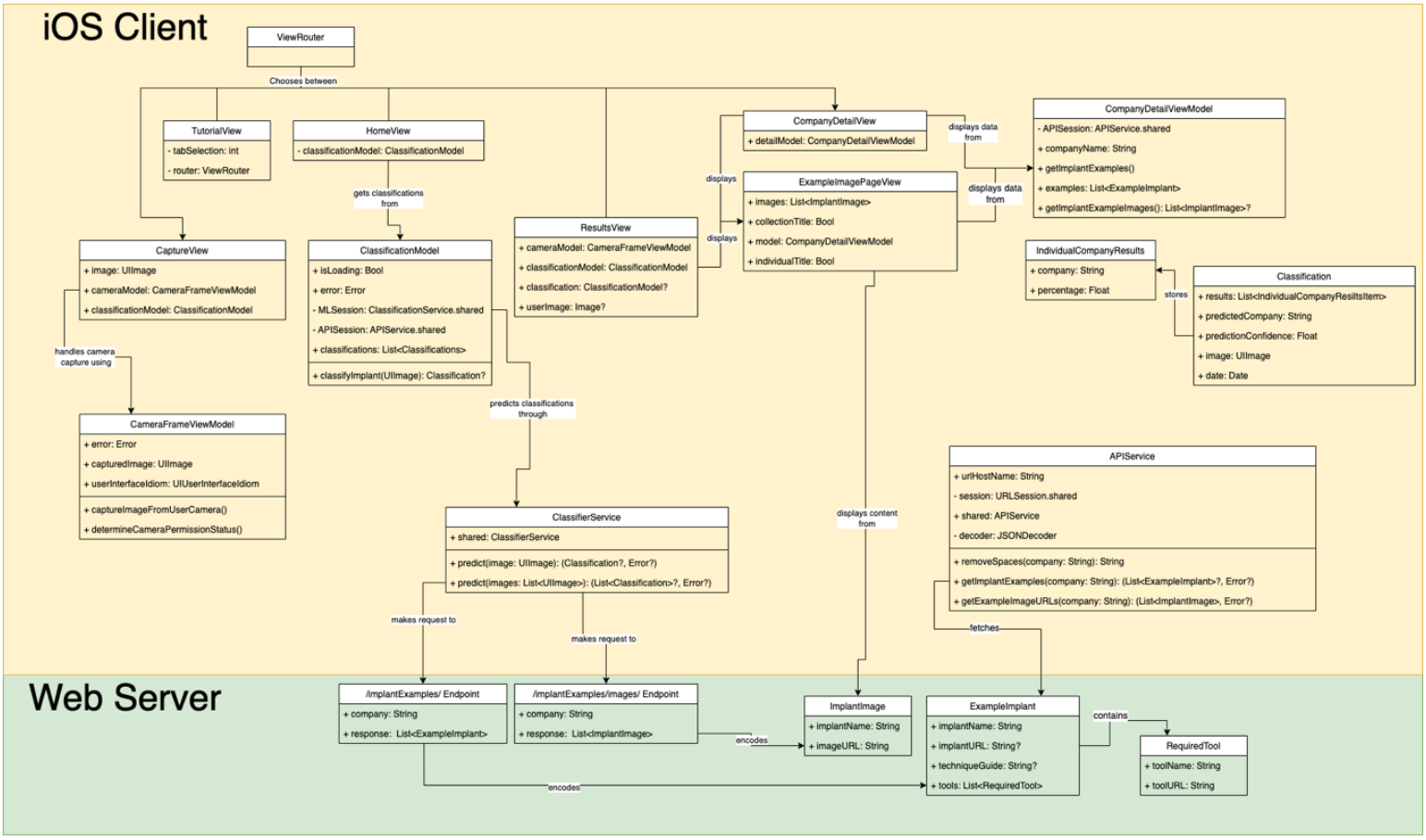

Figure 4.2 – Static Component Design Class Diagram

5. UI Design

In this section we present the User Interface (UI) of our mobile application. This application allows users to submit photos for local classification as well as see information on companies and their main implant examples. In this section we discuss the decisions behind our design choices and demonstrates all major screens associated with the application.

Figure 5.1 – Onboarding Sequence

When the user opens the application for the first time, they are directed to an onboarding sequence. This sequence consists of a greeter and explains the general flow of the application. Through this section users can understand the classification process and understand a simplified view of the results. Once the user has completed the tutorial, they can proceed to

the home screen. Additionally, the tutorial can be viewed at any time by clicking the “i” icon on the top right of all major screens in the application.

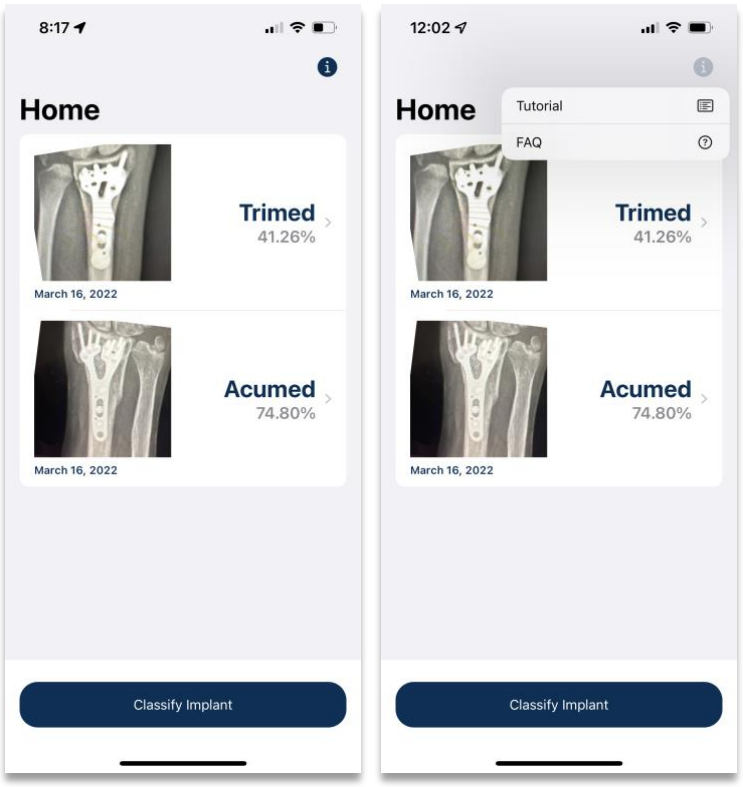

Figure 5.2 - Home Screen

After the tutorial sequence has been completed, and on every subsequent app launch, users are routed to the home screen. Here, they can see the list of classifications completed on the current app session (classifications are cleared on app close for privacy reasons). From here, a user can either select the information button or choose to begin a classification. When the information button is selected, a popover menu appears allowing users to view the tutorial and FAQ screens.

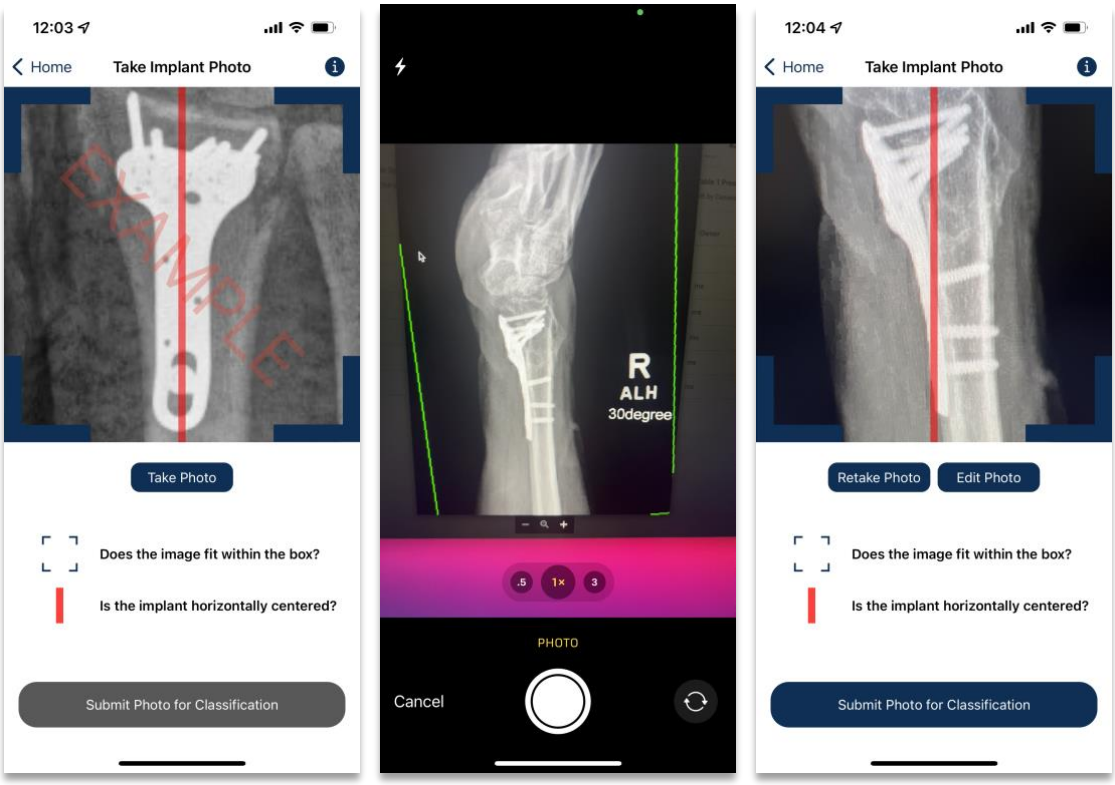

Figure 5.3 – Capture Screen

After the user begins the process of classifying an implant, they are directed to the capture screen. Here, the user can select the "take photo" button to capture a photo using the second of the three views above. An example is shown prior to capture for users to understand the process. After the user has captured an image, they can view their image in the viewfinder. They then have the option to retake or edit the photo. Retaking the photo takes the user back to the second of the above screens; editing a photo is shown in Figure 5.4 below.

Figure 5.4 - Edit Photo Screen

It can be very difficult for a user to eyeball the correct positioning for the implant in an image. For this reason, we have implemented an editing screen for those on iOS 15 and above. This screen allows users to rotate and crop their image to be aligned correctly. Once they are finished, the user can simply click “Done” and they will be redirected back to the last of the capture screens.

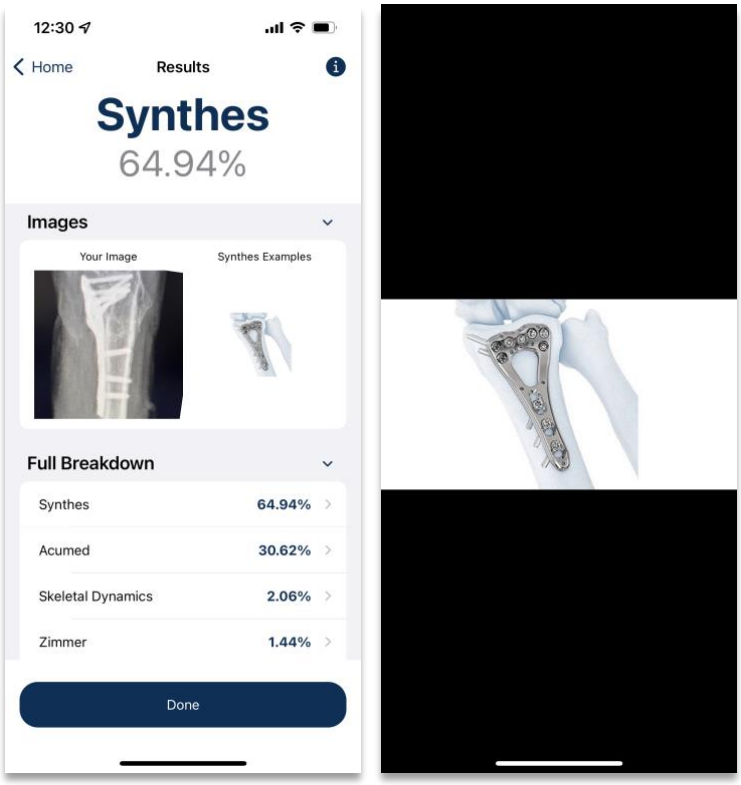

Figure 5.5 - Results Screen

Once the user chooses to classify their captured image, they are directed to the results screen. At the top of the screen, the user can see the predicted company and the associated confidence level. Additionally, for all available companies, a confidence level for each company is shown. This is helpful if the predictions are close between two companies, allowing doctors to explore both companies and make the final decision for themselves. Finally, the user can see their captured image and images associated with common implants made by the predicted company. To see these images in greater detail, the user can tap on each for a full-screen popup where the user can zoom and pan around.

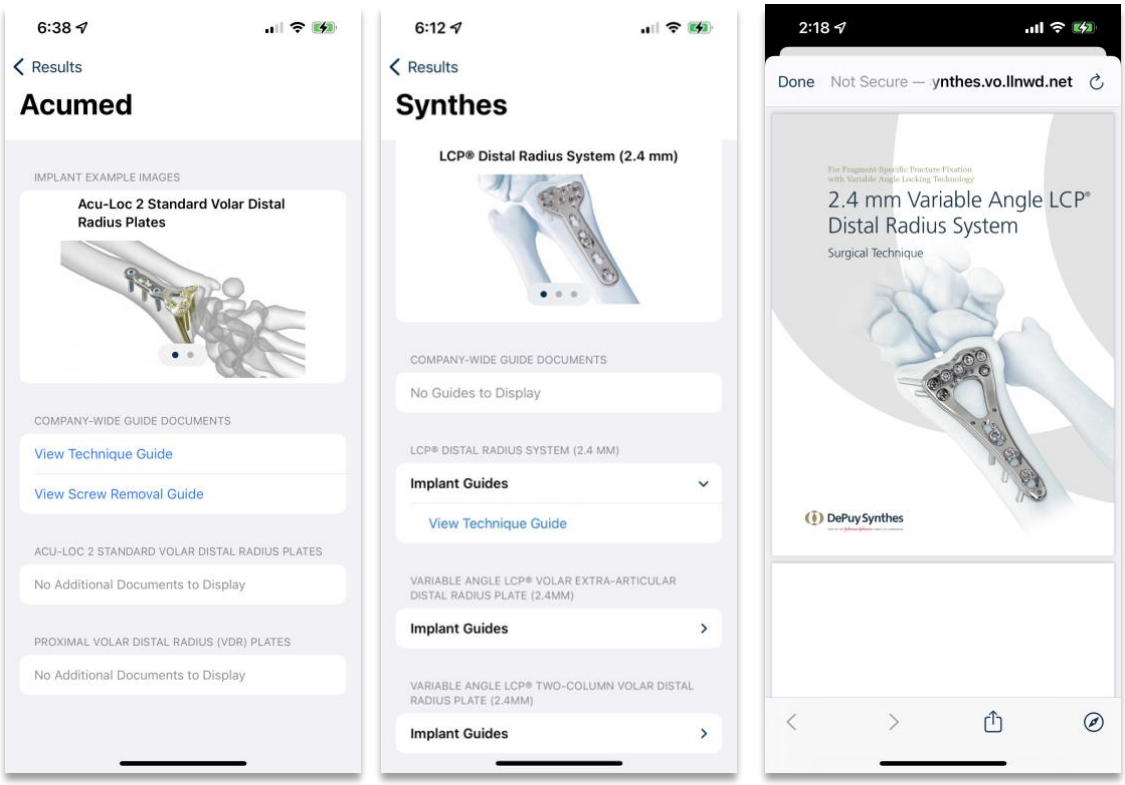

Figure 5.6 - Company Detail Screen

To learn more about the various implants a company produces, they can click on the company name from the results screen. This takes them to a detail page containing images and surgical guides for each main implant the manufacturer creates. Below the example images, a section for company-wide technique guides is attached for documents focused on all implants the company creates. To gain more information, documents associated with each specific implant are embedded into the list and when clicked will create a Safari popup showing the document as a PDF. This can be shared or open in the Safari app for more actions. The user can dismiss the popup using the “Done” button and can return to the results screen through the back button in the top left corner.

6. Appendix A

Below is the complete specification for our API web service. Following the OpenAPI standard, the documentation outlines request endpoints, parameters, bodies, responses, and response codes. An interactive live version can be viewed at https://distalradius.herokuapp.com/docs.

```
paths:
  /:
    get:
      summary: Root
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema: {}

  /implantExamples/{company}:
    get:
      summary: Getimplantexamples
      parameters:
        - required: true
          schema:
            title: Company
            type: string
          name: company
          in: path
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/CompanyImplant'
        '422':
          description: Validation Error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/HTTPValidationError'
```

/implantExamples/images/{company}:

```
  get:
    summary: Get implant image examples for a specific company
    parameters:
      - required: true
        schema:
          title: Company
          type: string
        name: company
        in: path
    responses:
      '200':
        description: Successful Response
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/ImplantImage'
      '422':
        description: Validation Error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/HTTPValidationError'
```

```
components:
  schemas:
    CompanyImplant:
      title: CompanyImplant
      required:
        - implantName
      type: object
      properties:
        implantName:
          title: Implantname
          type: string
        implantURL:
          title: Implanturl
          type: string
        techniqueGuide:
          title: Techniqueguide
          type: string
        tools:
          title: Tools
          type: array
          items:
            $ref: '#/components/schemas/Tool'
    ImplantImage:
      title: ImplantImage
      required:
        - implantName
        - imageURL
      type: object
      properties:
        implantName:
          title: Implantname
          type: string
        imageURL:
          title: Imageurl
          type: string
      description: Implant Image model
```

HTTPValidationError:

```
title: HTTPValidationError
type: object
properties:
  detail:
    title: Detail
    type: array
    items:
      $ref: '#/components/schemas/ValidationError'
```

ValidationError:

```
title: ValidationError
required:
 - loc
 - msg
 - type
type: object
properties:
  loc:
    title: Location
    type: array
    items:
      type: string
  msg:
    title: Message
    type: string
  type:
    title: Error Type
    type: string
```