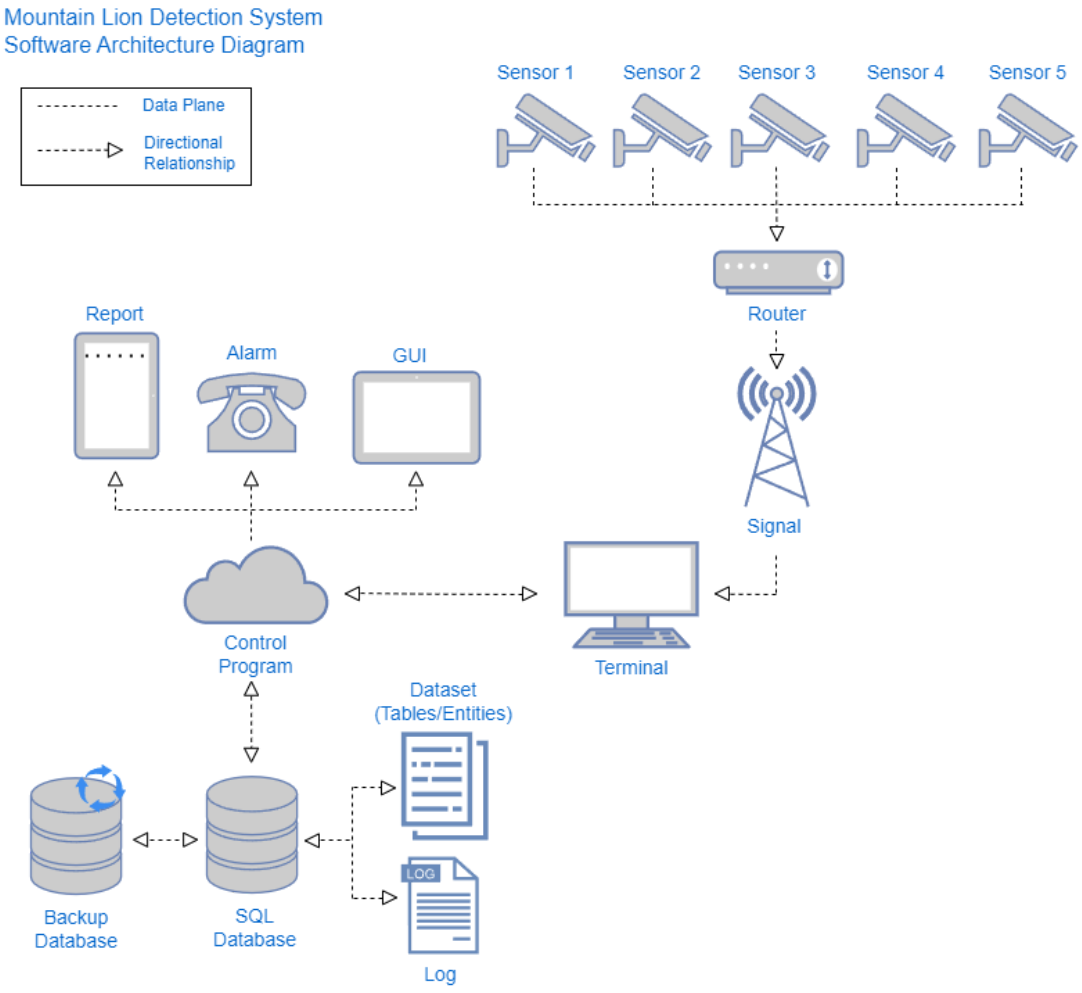
Mountain Lion Detection System

Daniel Aguilar, Yulianna Izaguirre, Conner Sommerfield

1. System Description

This is a mountain lion detection system, where the purpose is to detect any mountain lions nearby and if detected an alarm is sounded. This is needed to protect our civilians in parks all over California and allow our rangers to be more aware of what can possibly be out there in the parks. Though this system is for mountain lions it will also be able to detect different types of noises and animals that will directly notify the rangers to do what they need to do. This document will discuss precisely what requirements this system will need and how this system will be able to run smoothly.

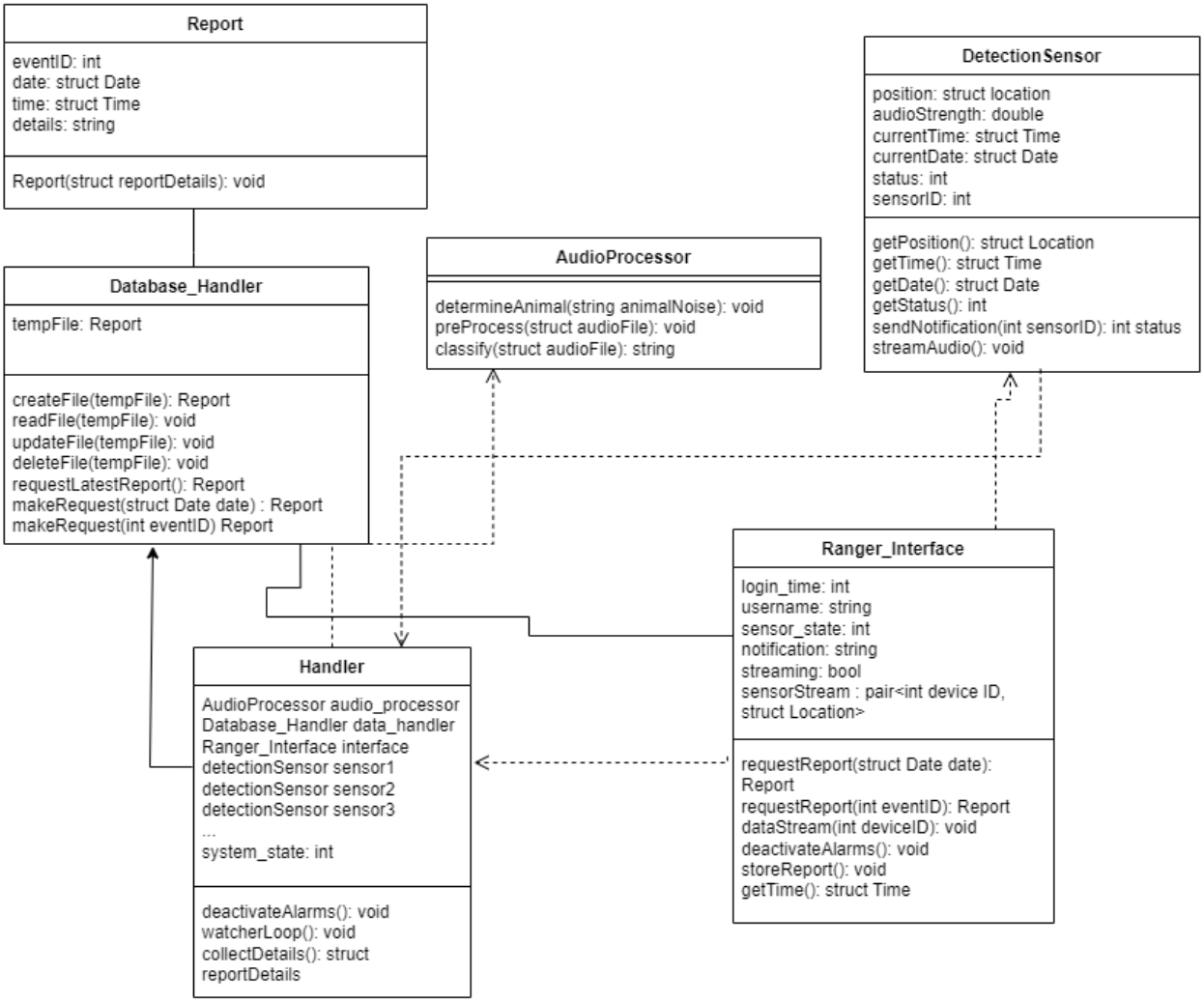
2. Software Architecture Overview



The software system includes two main components which are the animal detection system and the control program. The animal detection system consists of five noise detection sensors. The sensors detect the strength, type, and location of the noise within a 3-meter radius. This information is sent to the terminal to be processed into a structured alter message. The terminal receives the alert message from the sensors based on both the type of animal noise and strength. The control program which is on the terminal has multiple functions for the ranges. Once an alert has been received from the animal detection system the control program will initialize the alarm to sound off. The alarm must be turned off manually by a ranger, otherwise it will continue to sound. The control program allows for the ranger to review and classify each alert message as definite, suspected, or false. The terminal saves all mountain lion alerts along with a summary of alert information to the SQL database. Because the control program has access to the database it allows the rangers to request several types of reports, and display information with a graphical interface. Note that the control program will display any information such as the GUI and reports in the terminal.

The SQL database contains a log and dataset of several entities specific to the Mountain Lion Detection System, refer to section 5 for more details on the data management. Along with the main database there is also a backup database that is synced with the information that is updated or changed by the rangers through the terminal.

UML Diagram



The handler will be the entry point of the program that runs the main watcher loop. This watcher loop will use an interrupt system to wait for notifications from the sensors (instead of something like polling) so that the rangers can also interact with the terminal while the server is waiting for a notification. These interactions may include things like generating reports. When an interrupt occurs, the program flow will be given to the appropriate function. The handler will have instances of the other classes as attributes so that it can call the necessary functions. It will have a state attribute that will change as the system goes through different stages such as all clear, warning, and emergency. It will also be in charge of giving the official notification to sound the alarm.

When the detection_sensor senses strange activity, it will send a flag to the main server along with its sensor id (this will be an integer that identifies which sensor is which). This causes the main server to query the sensor and record a brief audio. The sensor can give back information from its attributes such as the approximate position of the animal, the intensity of the sound recorded, and the time during recording.

The handler will pass the audio information from the sensor to an audio_processor class which will then perform a variety of functions to prepare the audio for classification of the animal. It will have a preprocess() function for cleaning up noise and then a classify() method which will return the believed animal type using a deep learning algorithm. For testing, a known animal noise can be passed to show that it is classifying animals correctly.

The Ranger_Interface class will have the functions that are tied to the buttons of the graphical interface. It will keep track of information on the ranger's login session through its attributes like the login_time, username of the ranger, sensor state (all of which to be shown on screen), and any notification to be displayed. It will provide a graphical link to the database functions for rangers to be able to request reports using a few different parameters, for example one could request reports based on the date it happened or by an event id (each event will have an id attached). There will also be a dataStream(int deviceID) method that allows the ranger to watch a certain sensor on screen at will by clicking the desired sensor.

The Database_Handler class will simply define CRUD operations for interacting with the database and functions that wrap specific SQL commands to easily get specific information without having to design requests. Each report will be an object defined by the File class, this way the reports will have a consistent format. These files in the database can be created, read, updated, and deleted. These may be expanded to perform more specific and useful functions like SQL requests that get all the events for a certain day or to get the most recent event, etc.

3. Development Plan and Timeline

PROJECT TIMELINE
Approximate Budget: 3 million USD
APRIL 2023 - JULY 2023

Month of April (Starting April 3)

The engineers will first have to install all the appropriate hardware to support the system. We'll have one person for field setup such as the sensor and alarm devices, with a group of three for the mainframe server and network setup. The equipment will take about two weeks to configure to a basic level but will also need to be fortified to withstand various weather conditions such as rain, snow, and heavy winds, so we can expect hardware setup to take about three weeks. (ETA April 24 - May 1)

Month of May

Before starting with the higher-level development, we will give developers three weeks to get the most basic functionality running and do robust testing to ensure that our interfaces are working. The developers will set up unit tests to make sure devices have basic capabilities such as having the server successfully recognize a sensor message (without location), having dummy entries successfully saved to the database, and having the main server accept login information from rangers. With these foundations in place, higher-level development should be started by the beginning of June (June 1 - June 8).

Months of June and July

Developers will be able to use existing infrastructure and low-level capabilities to start plugging in higher-level functionality. This will include creating the classes and functions that will perform the needed functionality instead of using dummy data. The major aspects to work on will be the ranger interface, the main loop/sensor interrupt functionality, the CRUD operations to perform on the database using different querying parameters (event id, date, etc.), and the audio processing functionality which will involve extensive training to classify animals correctly. Expect 8 - 10 weeks to have the system working to an acceptable level for deployment (Delivery: August 5 - August 12)

Maintenance

We'll have to put measures into place to keep the system updated and add features as needed without complications. The system will be designed to accept new sensors or alarms with no changes to the existing infrastructure. Hardware checks on the devices will have to be performed every week to check for cracks or damage to devices as well as electrical issues. The audio processing will have to be fine-tuned to accommodate unforeseen circumstances and improve accuracy.

4. Verification Test Plan

Unit Testing

- **Unit: Test Set 1**

  determineAnimal(string Animal)

  Input: Roar

  Expected Output: AudioProcessor.determineAnimal = Mountain Lion

  The function determine Animal() will be passed a string from the Database_Handler which the Ranger_Interface can establish the animal data categorized in relation to the sound identified.

- **Unit: Test Set 2**

  void updateFile(tempFile)

  Input: Input of the temporary file created from an instance of the class, Report

  Expected Output: Output does not return anything but the file is has new material in the file that was set with the parameter

Functional/Integration Testing

- **Integration: Test Set 1**

  Classes: Ranger_Interface/Database_Handler

  Feature Description: Ranger interface report request should make request for a report with the database handler and display the data on

  Test 1:

  Input: Date 03/22/2021

  Feature: Ranger_Interface.requestReport(03222021)

  Expected Output:

  Database_Handler -> tempFile = Report

  Database_Handler.readFile(Report)

  Test 2:

  Input: Invalid Date 05/32/2070

  Feature: Ranger_Interface.requestReport(05322070)

Expected Output:

Database_Handler -> tempFile = null

Database_Handler.readFile(Report) = null

Description: tempFile will be null if there is no Report to read that has an ID of the Date specified in requestReport(), otherwise if there is a valid ID with a Report in Database_Handler then tempFile : Report and the Ranger_Interface has access to read file.

- **Integration: Test Set 2** -> Detection_Sensor/Ranger_Interface Integration
  Feature: Detection sensor activation should prompt data stream from respective sensor to Ranger Interface

  Test 1: Existent Sensor
  - Input:
    - integer 2

    Feature:
    - Detection_Sensor.sendNotification(2)
  - Expected Output:
    - Ranger_Interface -> sensor_state = 1
    - Ranger_Interface -> streaming = True
    - Ranger_Interface -> sensor_stream = 2, (50,100,50)

  Description: Sensor_state should be set to 1 to signify warning state (0 all clear, 1 is warning, 2 is emergency). Because Detection_Sensor.sendNotification() occurs, Ranger_Interface.dataStream() should be called with the appropriate deviceID to stream the current audio data to the ranger interface for the sensor with suspicious activity. This should cause the sensorStream attribute to change to signify the streaming sensor (2 in this case) as well as that sensor's x,y,z location (sensor 2 located at 50,100,50).

  Test 2: Non-Existent Sensor Edge Case
  - Input:
    - Integer 0

    Feature:

        - Detection_Sensor.sendNotification(0)
    - Expected Output:
        - Ranger_Interface -> sensor_state = 0
        - Ranger_Interface -> streaming = False
        - Ranger_Interface -> sensor_stream = 0, (0,0,0)

Sensor 0 is nonexistent / invalid input

Description: Sensor_state should still be 0 to signify all clear state. Because Detection_Sensor.sendNotification() was called, with a non-existent sensor. Ranger_Interface.dataStream() will not be called and streaming will stay as False (default), sensorStream attributes should remain at defaults since no sensor is streaming.

System/Acceptance Testing

- **System: Test Set 1**

  When an audio is detected from the detection sensor an alarm is set off the rangers then have to be able to turn off the sensor and then handle the information to what they need to do.

  Input:
    - login_time
        - 1401
    - username
        - ranger101
    - sensor_state
        - 1
    - notification
        - “audio detected”
    - streaming
        - true

  Feature:
    - Ranger_Interface.requestReport(eventID) -> struct Report tempfile
    - DetectionSensor.dataStream(sensorID)
    - DetectionSensor.getStatus() -> 1
    - Handler.deactivateAlarms()

- Ranger_Interface.storeReport()
- DetectionSensor.getTime() -> struct Time time

Expected Output:

- tempfile -> eventID = 1234
- login_time = 1401
- username = ranger101
- sensor_state = 0
- streaming = false

DetectionSensor.getStatus is a method that gets the status of the detection sensor 0 being off and 1 being on.

Handler.deactivateAlarms() is called to deactivate the alarms that have been set off this would change the sensor state to off this case being 0, this would also save the username of and login time of who accessed this and when did they.

Ranger_Interface.requestReport(eventID) is a method that is able to request access to the report that was generated from this instance of the detection sensor going off.

- **System: Test Set 2**

  Audio files of dangerous animals should cause a report to be generated for rangers to view at a later time with the correct details of the event.

  Input:

  - Audio file of known mountain lion roar
  - Mock Detection_Sensor
    - currentTime
      - Year: 2020
      - Month: 10
      - Day: 10
    - currentDate
      - Hour: 10
      - Minute: 10
      - Second: 10
    - ID
      - 10
    - Location

        - [10,10,10]
    - eventID -> 99999

Feature:

- Audio_Processor.classify(processedAudioFile) -> string “Mountain Lion”
- Detection_Sensor.getPosition -> [0,0,0]
- Detection_Sensor.getDate -> struct Date date
- Detection_Sensor.getTime -> struct Time time
- Handler.collectDetails() -> struct ReportDetails reportDetails
- Constructor Report(reportDetails)
- Database_Handler.createFile(tempFile)
- Ranger_Interface.generateReport(99999+1) -> struct Report report

Expected Output:

- report -> eventID = 100000
- report -> date.year = 2020
- report -> date.month = 10
- report -> date.day = 10
- report -> time.hour = 10
- report -> time.minute = 10
- report -> time.second = 10
- report -> location = [10,10,10]
- report -> details = “Mountain Lion”

Normally Database_Handler.requestLatestReport().eventID would be called to find the most recent eventID. This number would be incremented to be passed as the next report’s ID. In this case we pass in 99999 as a dummy theoretical return value from this function (high number to not overwrite other reports), meaning the next report should be generated using ID 100000.

A mock sensor Object will be created as input to see if its details are displayed accurately in the report.

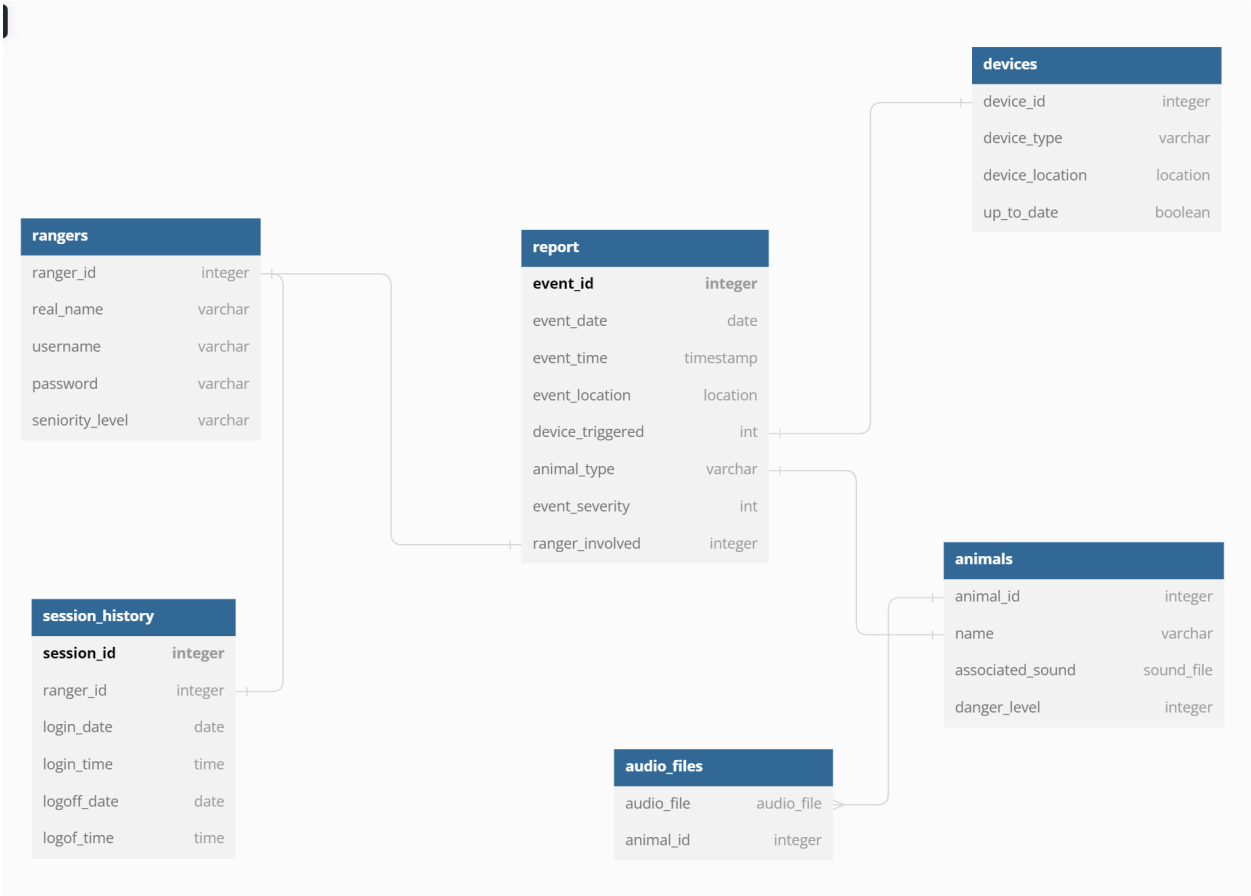
The audio_processor will be passed a known mountain lion audio file; this should be classified correctly as a mountain lion

The handler.collectDetails() method should get details returned from the audio_processor and detection_sensor objects as well as the incremented eventID and place them all into one struct. This will be passed to the Report constructor which will create a report ready to be stored in the database

The Database_Handler.createFile() method will be called and passed in the tempFile attribute which will be set as the Report that was generated. This will store the file to the database.

Finally The Ranger_Interface.generateReport(100000) should now return the report that was just saved to the database and should hold all the correct dummy details about the event.

5. Data Management



Our data management strategy will revolve around an SQL-based database for its robustness and simplicity. The main object of interest in this database are the reports that are generated for each event. This is the main information that the ranger will be interacting with and is important for gathering statistics and preparing for future events. These reports will contain all the important information regarding an event, and will therefore need to pull from a few different tables to populate its fields.

These other tables include the animals table, the rangers table, and the devices table. The system will have to be able to recognize a large amount of animals which will have to store potentially massive amounts of audio files for training and sound recognition. We also need to pre-classify the danger level of each animal so that a sensor trigger can be correctly classified and set off the alarm if needed. We will also have a large number of personnel that we will have to keep track of along with usernames and large hashes for passwords. Then we have to keep track of each of our devices such as motion sensors, alarms, and auxiliary devices along with their relevant information like its location. Each report will link to resources from these tables; for example, a report will want details on the device, but we can compartmentalize this into its own table and then simply pull the row we need and use the relevant information from there in our report.

It's also worth mentioning the one-to-many relationship that is held between an animal item and its audio files. As mentioned earlier, there may be a large number of audio files we have to store for each animal. SQL is robust, but can also be rigid, and we don't want to have to have a predefined number of columns to represent a certain number of acceptable files (for example, without this relationship, we would have to say maybe each animal has 3 columns for audio files). The solution is that we can make another table of audio files. Each entry in the audio file database will have an animal_id associated with it. Let's say a mountain lion has animal_id 1. We may have 20 audio files for mountain lions, but we will place the animal_id 1 as one of the columns for each of these audio files, meaning they will be linked to that animal. If we put a different id like 2, we would associate the given audio file with an animal that isn't the mountain lion. In this way, we can store a large amount of audio files for each animal.

Several alternatives for the technology that we could have implemented into our detection system are the use of just noise detectors. Another option we could have used is a specific camera with computer software, such as AI, in order to detect and classify the kind of animal and the level of danger. The benefits would provide the rangers with a lighter workload as the computer software would be handling large portions of work. Some issues that may arise with this alternative technological implementation into the detection system is misclassification. If there was low accuracy in classifications, this would have to be checked and fixed by rangers, leading to redundant work. With the software, we can use a different application to store files, for example the use of JSON. The data management would take human language and would be able to store different objects in different places. In this case, it would be very easy and flexible to store different audio files of animals since we are able to use a variable that can hold different

columns. This could be a better alternative to the data management organization we currently have as it allows for the convenient use of separate files.