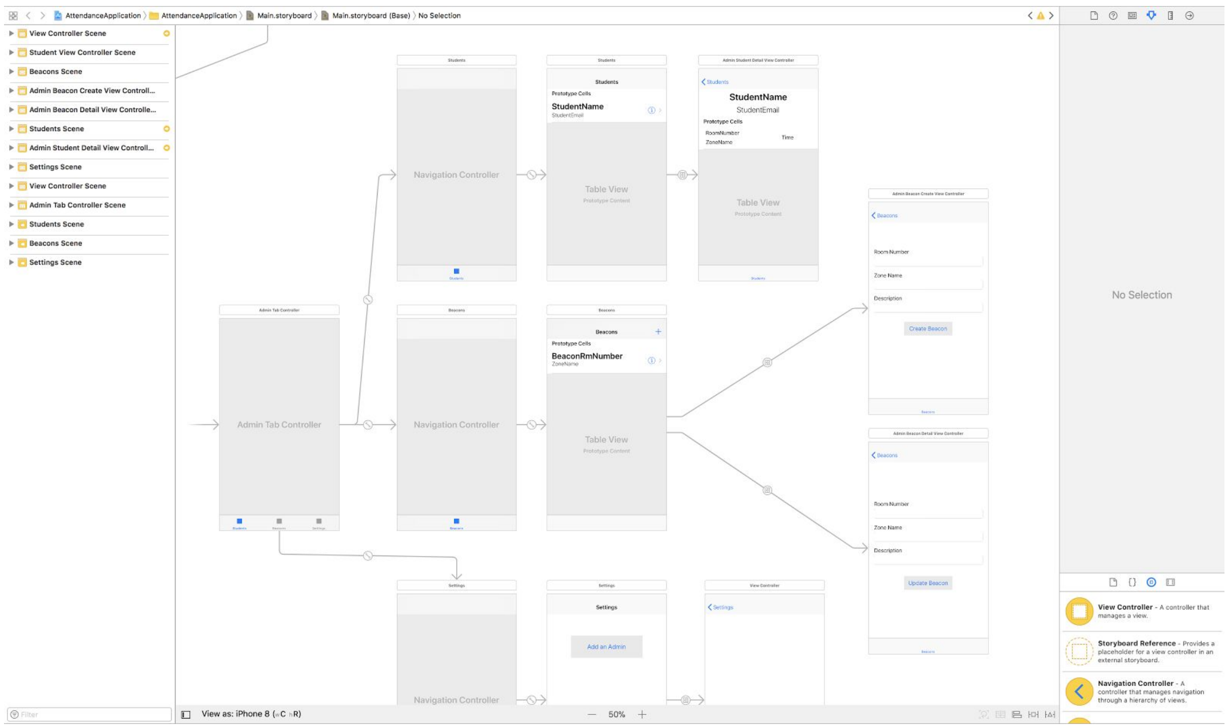
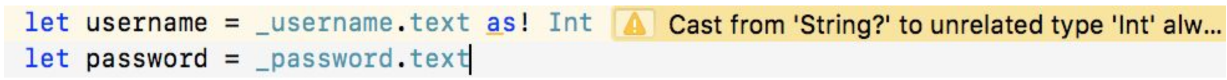
Criterion C: Development

(1000 words)

Third Party Tools

Apple Xcode IDE for Swift: The Xcode IDE provides a GUI builder that helped me create the IOS and MacOS applications. It also provides debugging tools.

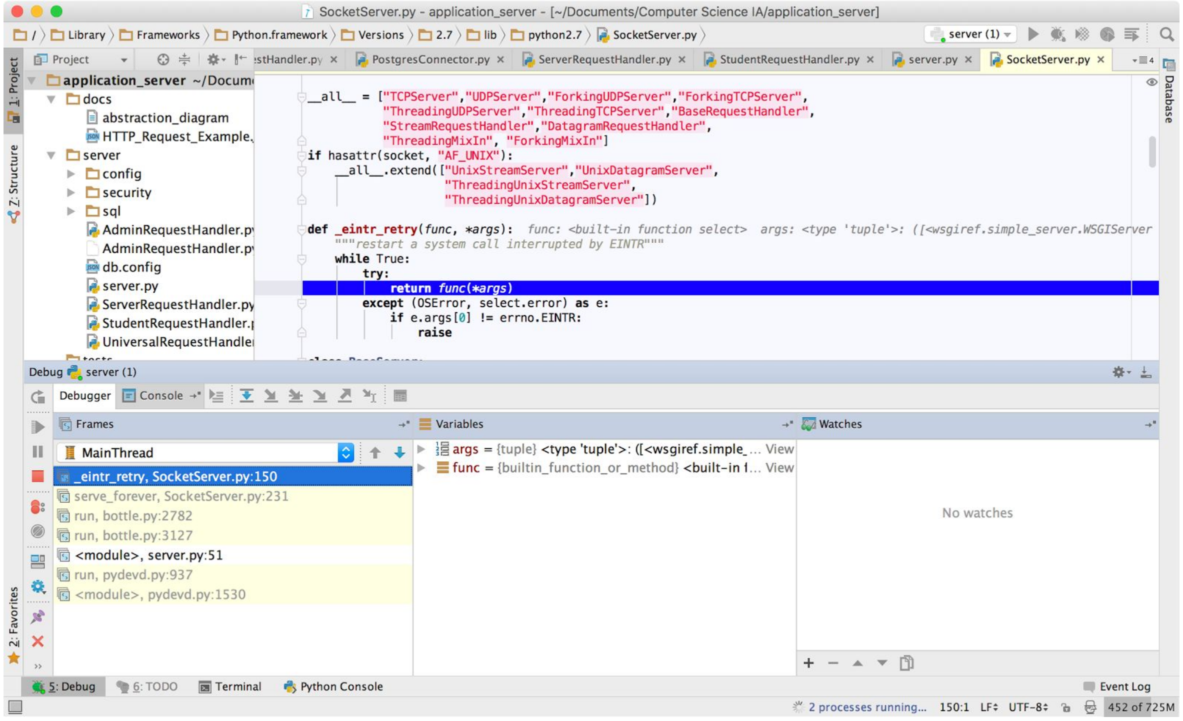

AttendanceApplication 〉 AttendanceApplication 〉 Main.storyboard 〉 Main.storyboard (Base) 〉 No Selection
View Controller Scene
Student View Controller Scene
Beacons Scene
Admin Beacon Create View Controll...
Admin Beacon Detail View Controlle...
Students Scene
Admin Student Detail View Controll...
Settings Scene
View Controller Scene
Admin Tab Controller Scene
Students Scene
Beacons Scene
Settings Scene
Filter
Navigation Controller
Students
Prototype Cells
StudentName
StudentEmail
Table View
Prototype Content
Students
StudentName
StudentEmail
RoomNumber
ZoneName
Time
Admin Tab Controller
Beacons
BeaconRmNumber
ZoneName
Room Number
Zone Name
Description
Create Beacon
Update Beacon
Settings
Add an Admin
View as: iPhone 8 (wC hR)
50%
No Selection
View Controller - A controller that manages a view.
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.
Navigation Controller - A controller that manages navigation through a hierarchy of views.



```
let username = _username.text as! Int   ⚠ Cast from 'String?' to unrelated type 'Int' alw...
let password = _password.text
```

Xcode also provides data validation. This means that it will makes it impossible to make datatype errors, enforcing proper exception handling and casting.

Pycharm IDE for Python development: The Pycharm IDE provides debugging tools for the middleware.

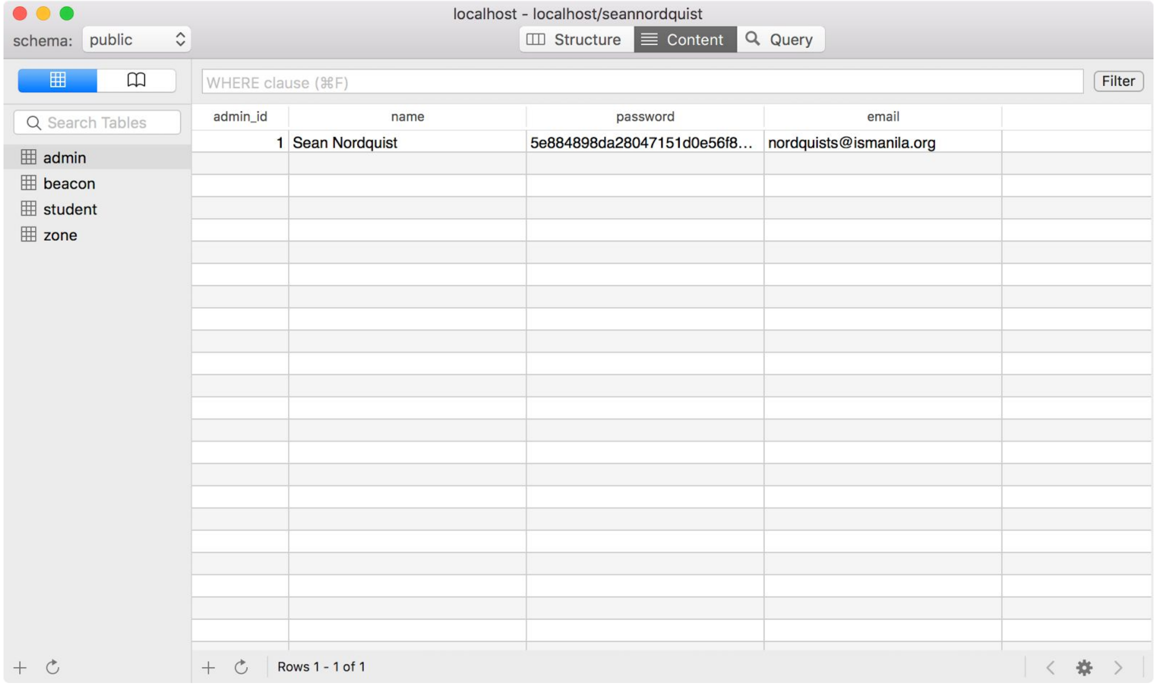


```
__all__ = ["TCPServer","UDPServer","ForkingUDPServer","ForkingTCPServer",
           "ThreadingUDPServer","ThreadingTCPServer","BaseRequestHandler",
           "StreamRequestHandler","DatagramRequestHandler",
           "ThreadingMixIn", "ForkingMixIn"]
if hasattr(socket, "AF_UNIX"):
    __all__.extend(["UnixStreamServer","UnixDatagramServer",
                    "ThreadingUnixStreamServer",
                    "ThreadingUnixDatagramServer"])

def _eintr_retry(func, *args):  func: <built-in function select>  args: <type 'tuple'>: ([<wsgiref.simple_server.WSGIServer
    """restart a system call interrupted by EINTR"""
    while True:
        try:
            return func(*args)
        except (OSError, select.error) as e:
            if e.args[0] != errno.EINTR:
                raise
```

```
_eintr_retry, SocketServer.py:150
serve_forever, SocketServer.py:231
run, bottle.py:2782
run, bottle.py:3127
<module>, server.py:51
run, pydevd.py:937
<module>, pydevd.py:1530
```

```
args = {tuple} <type 'tuple'>: ([<wsgiref.simple_... View
func = {builtin_function_or_method} <built-in f... View
```

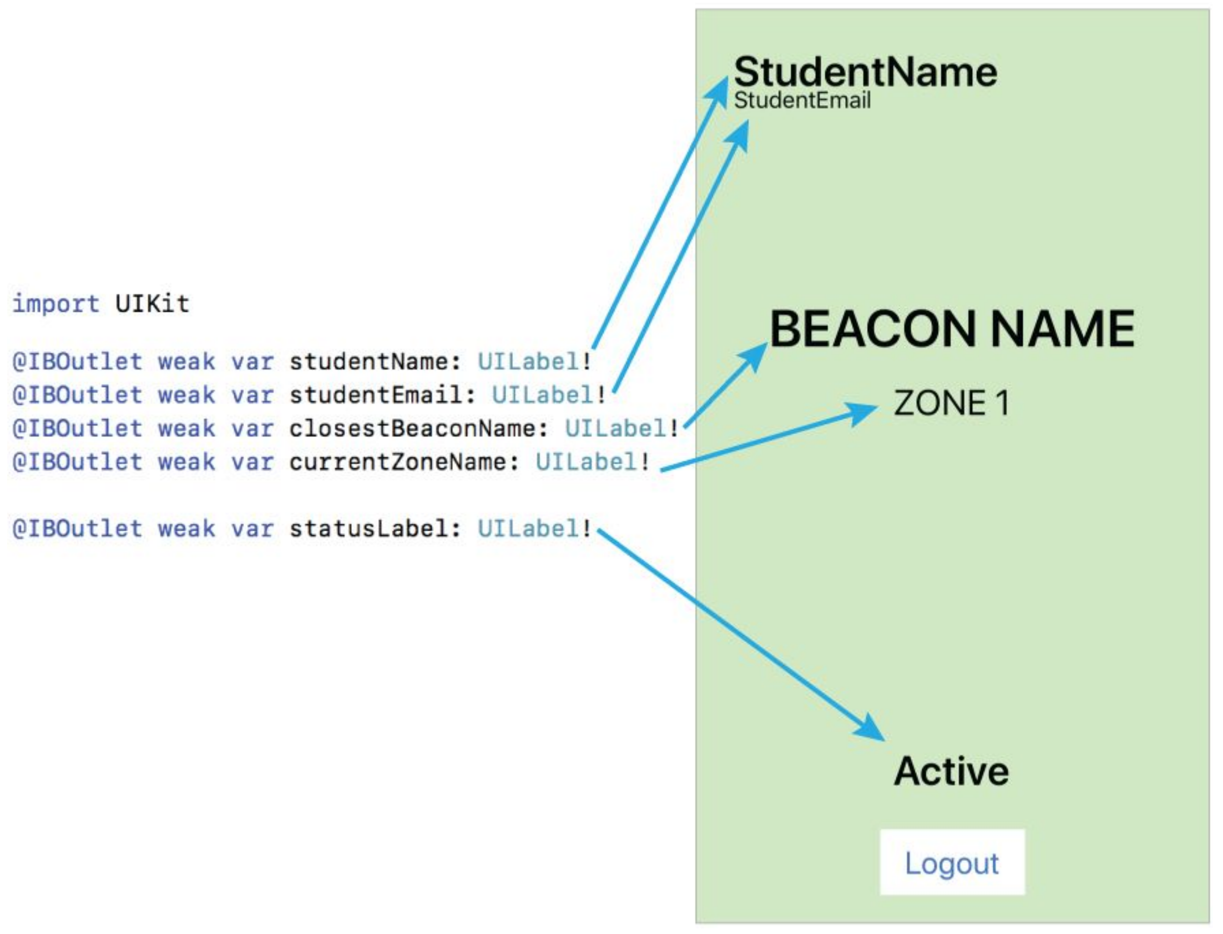
PSequel for a GUI of the Postgresql database: The GUI interface was helpful because it provided a means to see the data graphically before the GUI interface was working.



admin_id	name	password	email
1	Sean Nordquist	5e884898da28047151d0e56f8…	nordquists@ismanila.org

Third Party Libraries

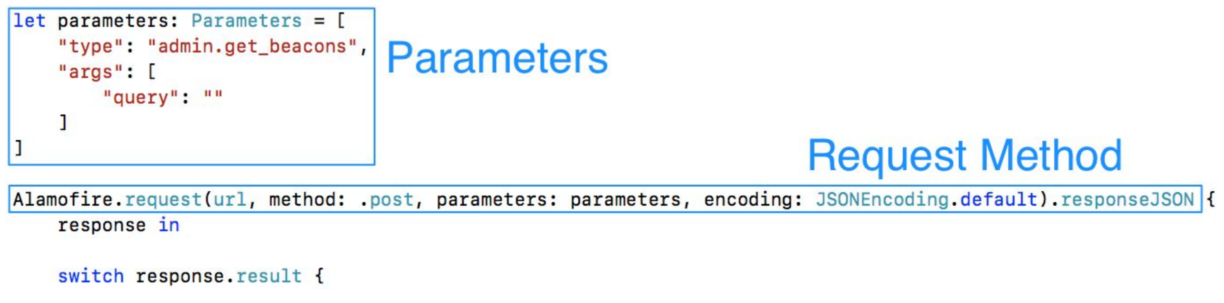
UIKit: I used UIKit, which is a GUI development library for IOS, to handle the GUI of the IOS application.



```swift
import UIKit

@IBOutlet weak var studentName: UILabel!
@IBOutlet weak var studentEmail: UILabel!
@IBOutlet weak var closestBeaconName: UILabel!
@IBOutlet weak var currentZoneName: UILabel!

@IBOutlet weak var statusLabel: UILabel!
```

All elements prefixed with UI are from the UIKit library.

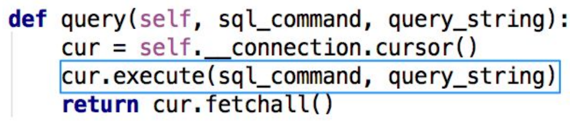
Alamofire: I used Alamofire, a Swift HTTP networking library, to send HTTP POST requests to the server. The library allowed me to add JSON parameters.



```swift
let parameters: Parameters = [
    "type": "admin.get_beacons",
    "args": [
        "query": ""
    ]
]

Alamofire.request(url, method: .post, parameters: parameters, encoding: JSONEncoding.default).responseJSON {
    response in

    switch response.result {
```

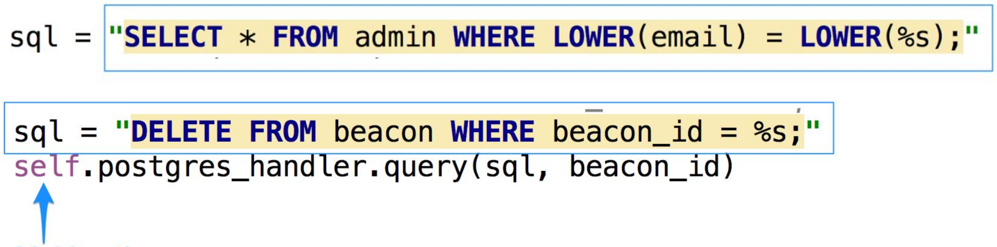
It also decodes the JSON response into a Swift format.

Psycopg2: I used Psycopg2 to securely interface with the Postgresql databases.



```
def query(self, sql_command, query_string):
    cur = self.__connection.cursor()
    cur.execute(sql_command, query_string)
    return cur.fetchall()
```

`execute()` executes the sql command passed in as a parameter.



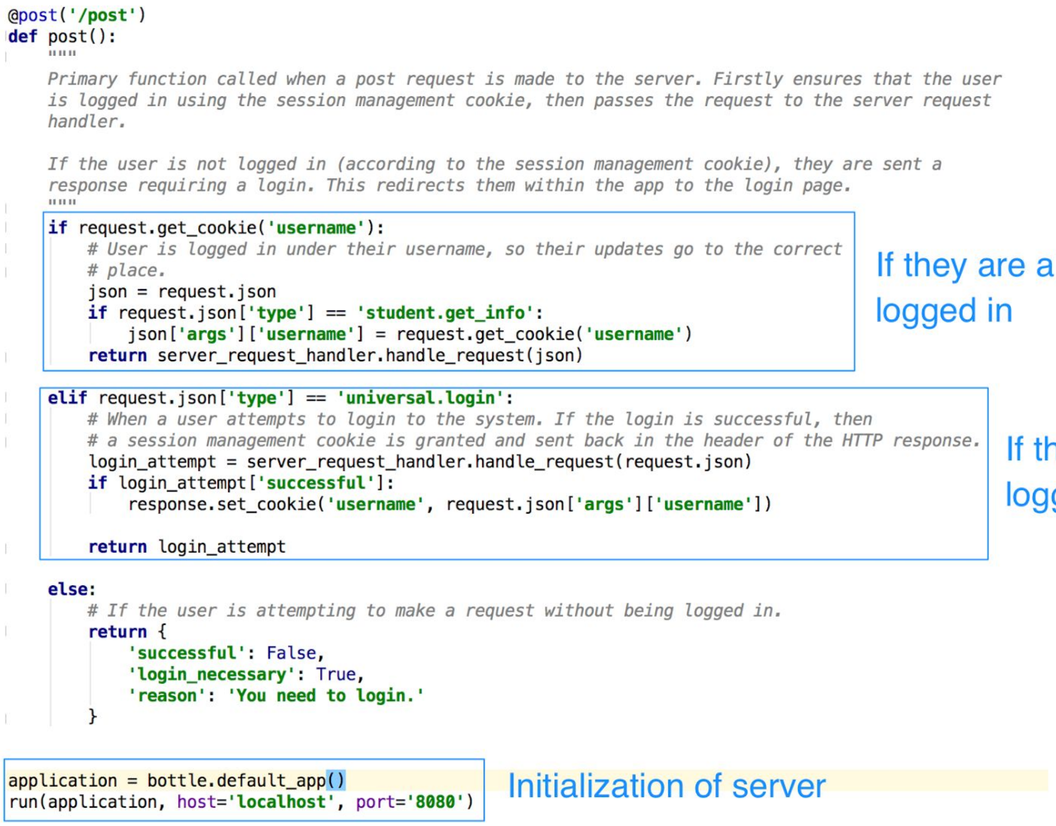
```
sql = "SELECT * FROM admin WHERE LOWER(email) = LOWER(%s);"
```

Example SQL Requests

```
sql = "DELETE FROM beacon WHERE beacon_id = %s;"
self.postgres_handler.query(sql, beacon_id)
```

Making the request.

Bottle.py: The bottle.py library provides a HTTP server. The library provides an abstraction from a typical HTTP request by providing decoding natively. The code below is mine, using the framework.



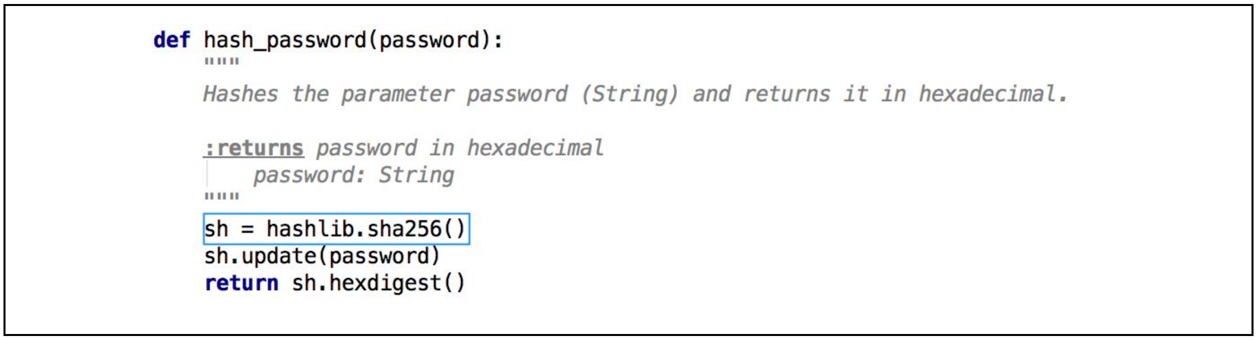
```
@post('/post')
def post():
    """
    Primary function called when a post request is made to the server. Firstly ensures that the user
    is logged in using the session management cookie, then passes the request to the server request
    handler.

    If the user is not logged in (according to the session management cookie), they are sent a
    response requiring a login. This redirects them within the app to the login page.
    """
    if request.get_cookie('username'):
        # User is logged in under their username, so their updates go to the correct
        # place.
        json = request.json
        if request.json['type'] == 'student.get_info':
            json['args']['username'] = request.get_cookie('username')
        return server_request_handler.handle_request(json)

    elif request.json['type'] == 'universal.login':
        # When a user attempts to login to the system. If the login is successful, then
        # a session management cookie is granted and sent back in the header of the HTTP response.
        login_attempt = server_request_handler.handle_request(request.json)
        if login_attempt['successful']:
            response.set_cookie('username', request.json['args']['username'])

        return login_attempt

    else:
        # If the user is attempting to make a request without being logged in.
        return {
            'successful': False,
            'login_necessary': True,
            'reason': 'You need to login.'
        }

application = bottle.default_app()
run(application, host='localhost', port='8080')
```

If they are already logged in

If they are logging in

Initialization of server

Hashlib: The hashlib library provides functions for hashing passwords. In the application, the hashing algorithm used is the SHA256.



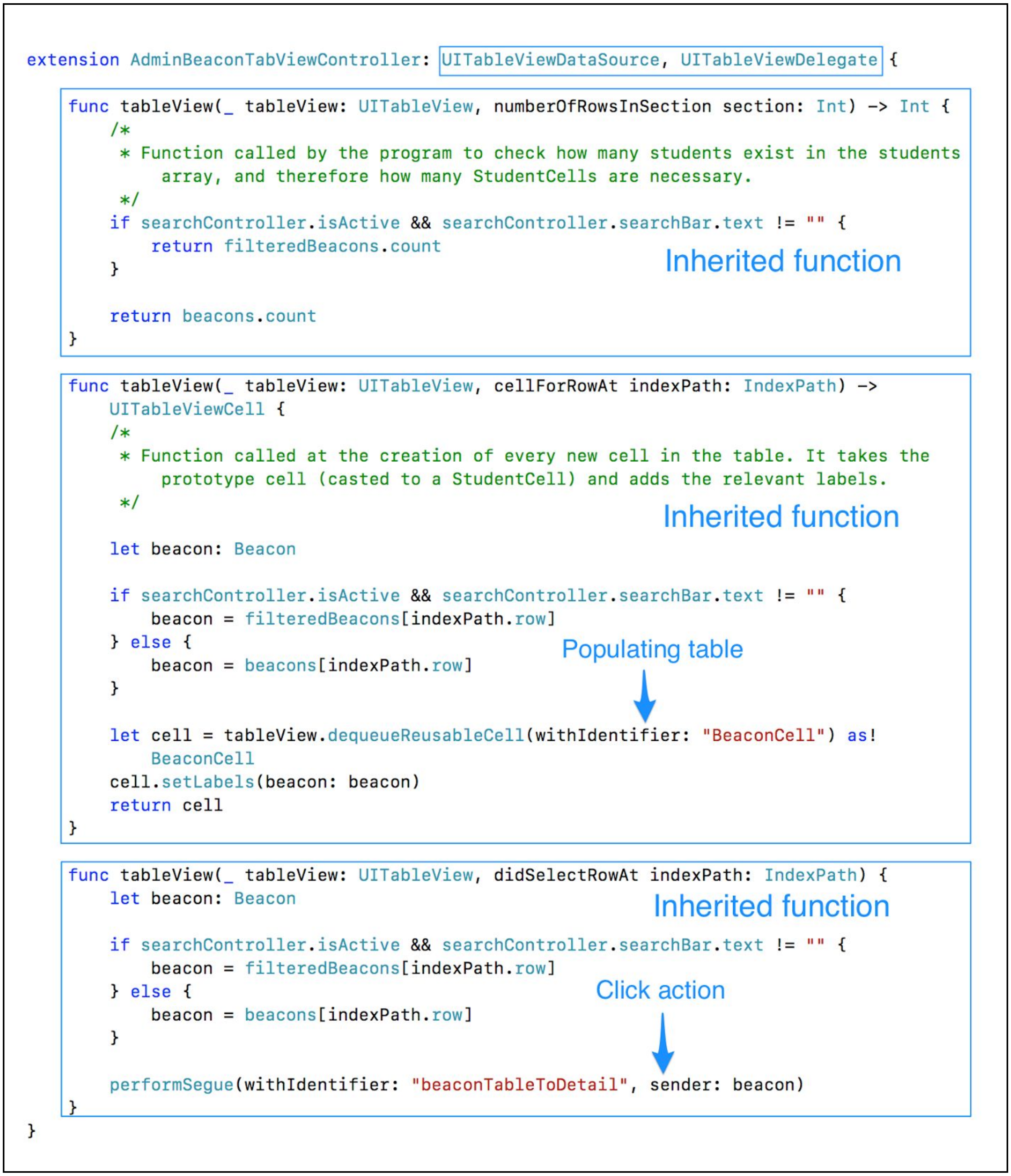
```
def hash_password(password):
    """
    Hashes the parameter password (String) and returns it in hexadecimal.

    :returns password in hexadecimal
        password: String
    """
    sh = hashlib.sha256()
    sh.update(password)
    return sh.hexdigest()
```

A modified version of BLCBeaconAdvertisement (Robinson): It is a file that allows for the formatting of the beacon packets.

Inheritance

Custom Table Views

To update the TableView in the AdminView, I needed my ViewController class to implement TableView methods. To do this, I created an extension which inherited from TableViewDataSource and TableViewDelegate.

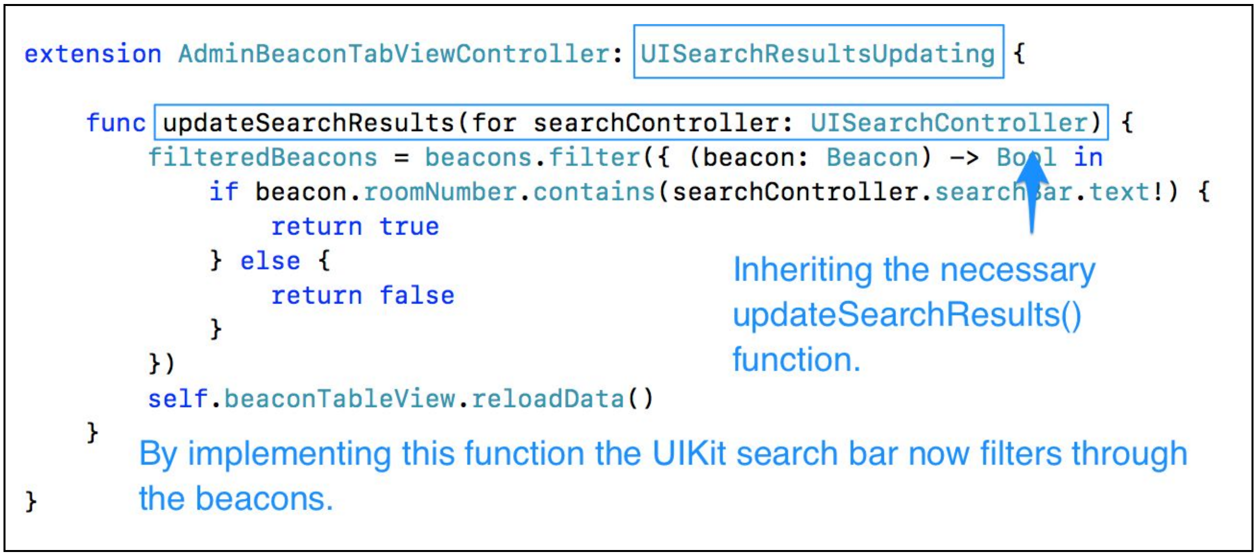


```
extension AdminBeaconTabViewController: UITableViewDataSource, UITableViewDelegate {
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        /*
         * Function called by the program to check how many students exist in the students
             array, and therefore how many StudentCells are necessary.
         */
        if searchController.isActive && searchController.searchBar.text != "" {
            return filteredBeacons.count
        }

        return beacons.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
        /*
         * Function called at the creation of every new cell in the table. It takes the
             prototype cell (casted to a StudentCell) and adds the relevant labels.
         */

        let beacon: Beacon

        if searchController.isActive && searchController.searchBar.text != "" {
            beacon = filteredBeacons[indexPath.row]
        } else {
            beacon = beacons[indexPath.row]
        }

        let cell = tableView.dequeueReusableCell(withIdentifier: "BeaconCell") as!
            BeaconCell
        cell.setLabels(beacon: beacon)
        return cell
    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let beacon: Beacon

        if searchController.isActive && searchController.searchBar.text != "" {
            beacon = filteredBeacons[indexPath.row]
        } else {
            beacon = beacons[indexPath.row]
        }

        performSegue(withIdentifier: "beaconTableToDetail", sender: beacon)
    }
}
```

Inheriting from these two classes provides methods for updating the table.

Custom Search Controller

The UIKit library provides the UI for a search bar; however, functionality must be inherited from a UISearchResultsUpdating class. Again, I created an extension to my ViewController and inherited from the UISearchResultsUpdating.



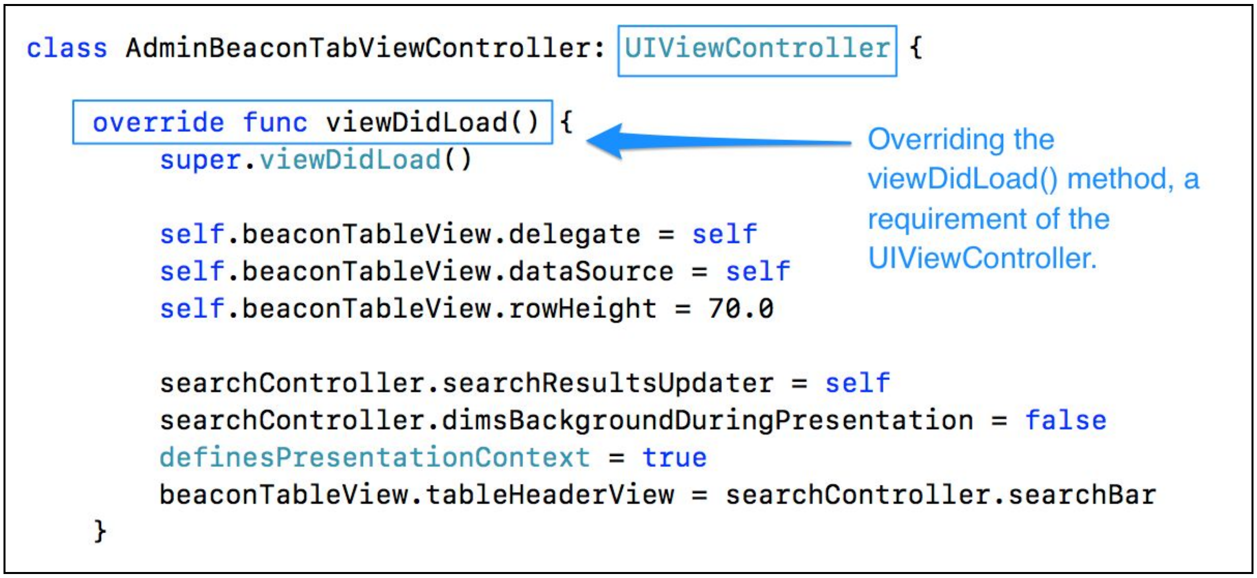
```swift
extension AdminBeaconTabViewController: UISearchResultsUpdating {

    func updateSearchResults(for searchController: UISearchController) {
        filteredBeacons = beacons.filter({ (beacon: Beacon) -> Bool in
            if beacon.roomNumber.contains(searchController.searchBar.text!) {
                return true
            } else {
                return false
            }
        })
        self.beaconTableView.reloadData()
    }
}
```

Inheriting the necessary updateSearchResults() function.

By implementing this function the UIKit search bar now filters through the beacons.

All UIViewControllers

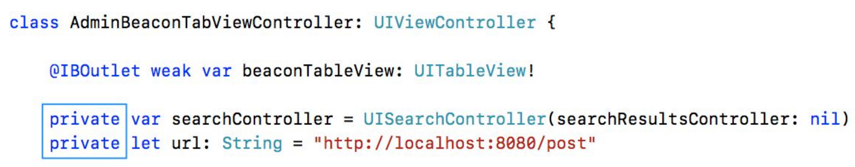
To be accepted by the Swift compiler as a runnable view controller, all view controllers must inherit and override methods from UIViewController.



```swift
class AdminBeaconTabViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        self.beaconTableView.delegate = self
        self.beaconTableView.dataSource = self
        self.beaconTableView.rowHeight = 70.0

        searchController.searchResultsUpdater = self
        searchController.dimsBackgroundDuringPresentation = false
        definesPresentationContext = true
        beaconTableView.tableHeaderView = searchController.searchBar
    }
```

Overriding the viewDidLoad() method, a requirement of the UIViewController.

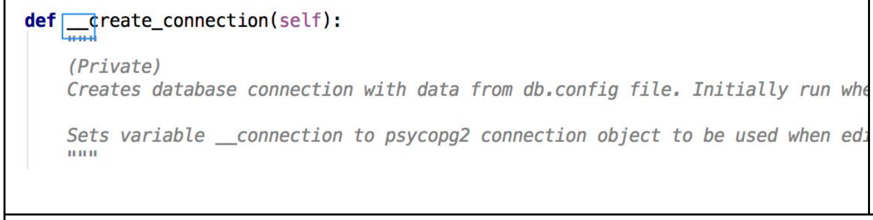
Encapsulation



```
class AdminBeaconTabViewController: UIViewController {

    @IBOutlet weak var beaconTableView: UITableView!

    private var searchController = UISearchController(searchResultsController: nil)
    private let url: String = "http://localhost:8080/post"
```

Swift

I encapsulated all fields that may create coupling problems between classes (all fields not delimited by 'private' are implicitly internal).



```
def __create_connection(self):

    (Private)
    Creates database connection with data from db.config file. Initially run whe

    Sets variable __connection to psycopg2 connection object to be used when edi
    """
```

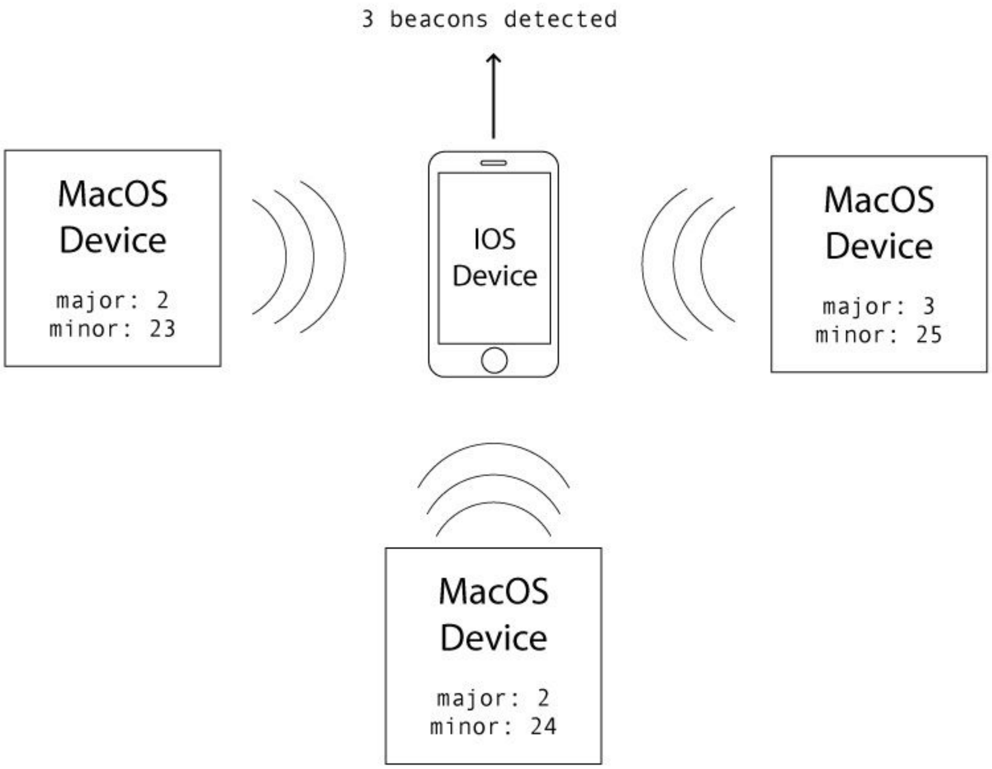
Python

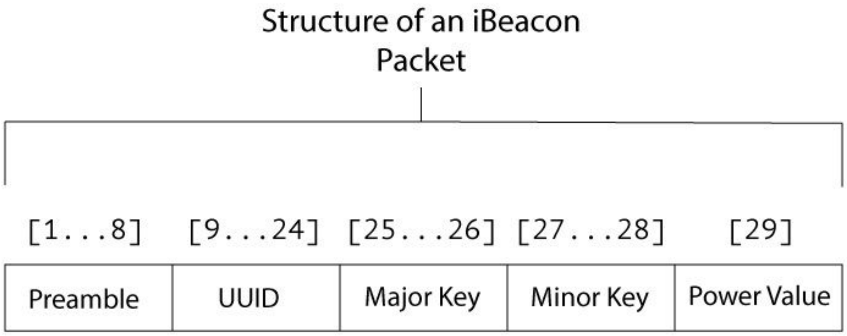
I encapsulated fields and methods for which it was necessary using the Python convention of a double-under (__).

Encapsulating data in each of the distinct elements of the program promotes extensibility by separating the front-end from the back-end, preventing coupling.

iBeacon Integration

iBeacon is a protocol for a Bluetooth advertiser-client architecture. It facilitates the detection of beacons by providing a standardized Bluetooth packet constructions.





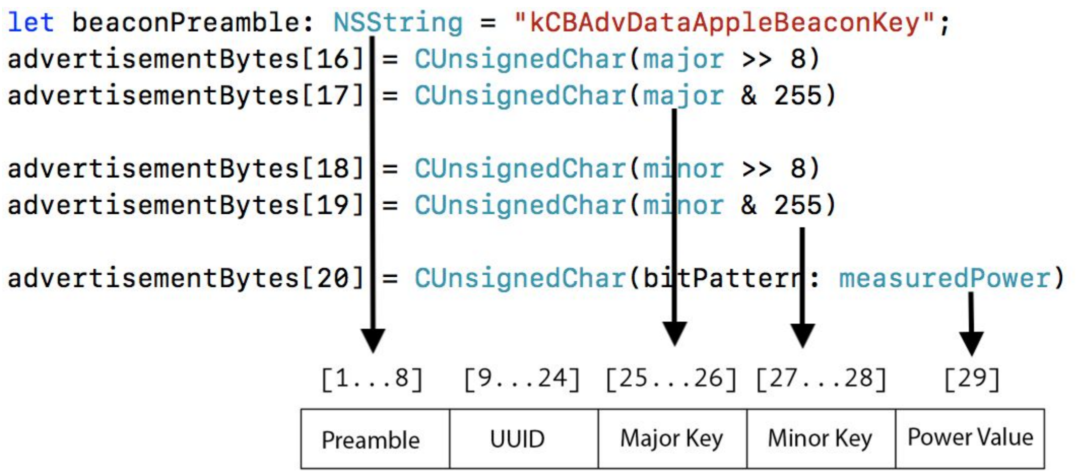
Structure of an iBeacon Packet

[1...8]	[9...24]	[25...26]	[27...28]	[29]
Preamble	UUID	Major Key	Minor Key	Power Value

A packet with a list of 29 Unsigned Chars (bytes) is used by the iBeacon protocol as the advertisement. It is this packet that will be transmitted for the clients to receive.

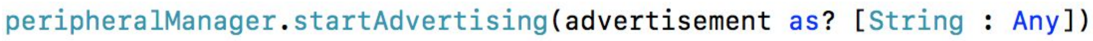
Advertiser (MacOS):
This packet must contain the the UUID of the beacon, the major key, and the minor key, which is added by casting the major:Int and minor:Int to an Unsigned Char. The UUID is added to the packet

later.

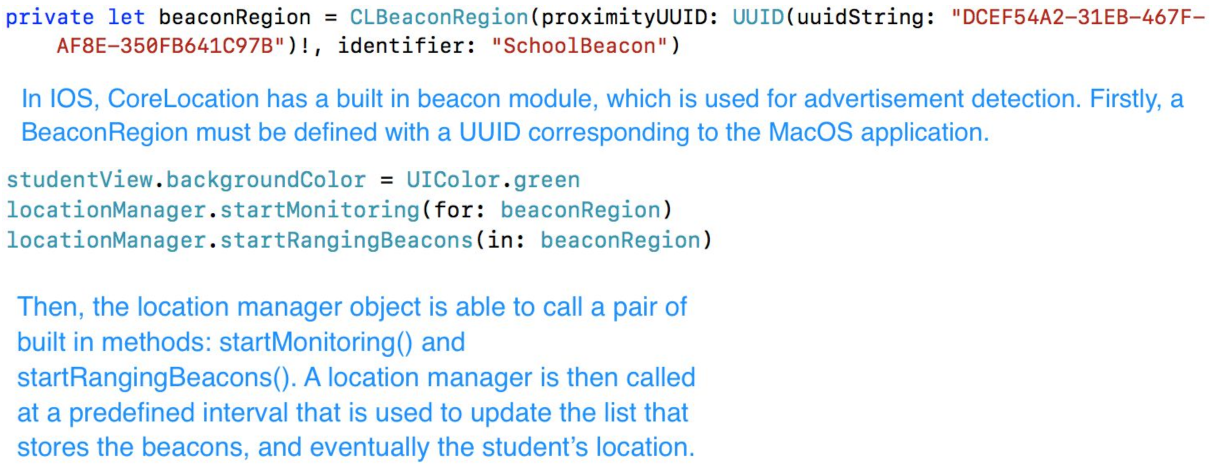


```
let beaconPreamble: NSString = "kCBAdvDataAppleBeaconKey";
advertisementBytes[16] = CUnsignedChar(major >> 8)
advertisementBytes[17] = CUnsignedChar(major & 255)

advertisementBytes[18] = CUnsignedChar(minor >> 8)
advertisementBytes[19] = CUnsignedChar(minor & 255)

advertisementBytes[20] = CUnsignedChar(bitPattern: measuredPower)
```

This packet of data can then be transmitted via a built-in CoreBluetooth class: CBPeripheralManager. This manager is instantiated and calls a method: startAdvertising(). The method takes the previously created list of UnsignedChars, and begins transmitting.



```
peripheralManager.startAdvertising(advertisement as? [String : Any])
```



```
private let beaconRegion = CLBeaconRegion(proximityUUID: UUID(uuidString: "DCEF54A2-31EB-467F-
    AF8E-350FB641C97B")!, identifier: "SchoolBeacon")
```

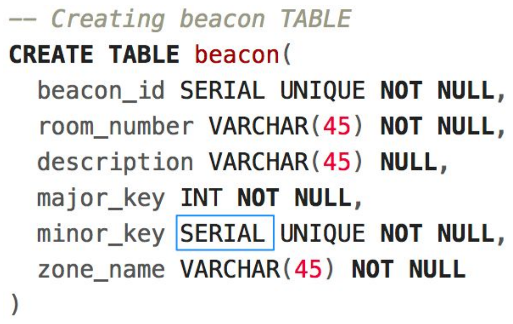
In IOS, CoreLocation has a built in beacon module, which is used for advertisement detection. Firstly, a BeaconRegion must be defined with a UUID corresponding to the MacOS application.

```
studentView.backgroundColor = UIColor.green
locationManager.startMonitoring(for: beaconRegion)
locationManager.startRangingBeacons(in: beaconRegion)
```

Then, the location manager object is able to call a pair of built in methods: startMonitoring() and startRangingBeacons(). A location manager is then called at a predefined interval that is used to update the list that stores the beacons, and eventually the student's location.

Client (IOS)

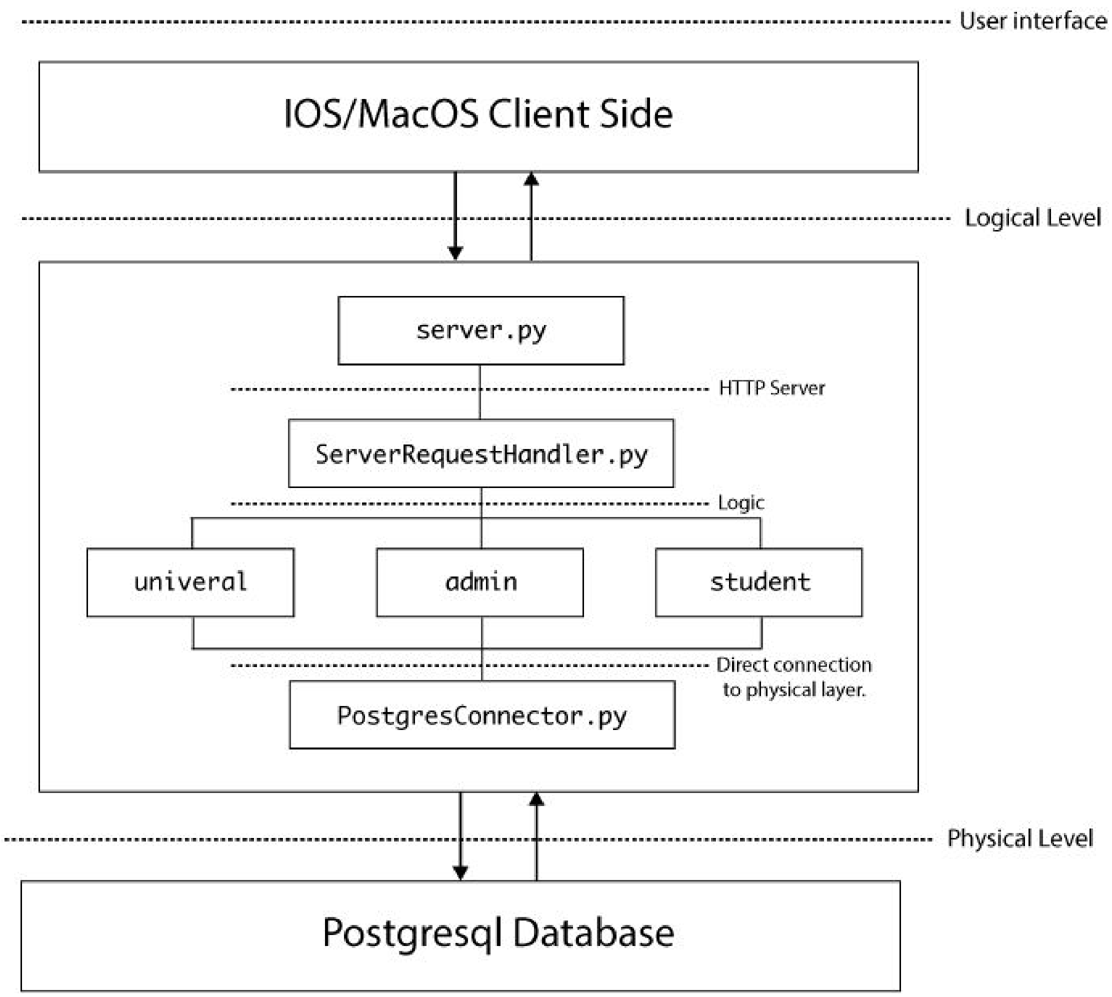
Abstraction



```
-- Creating beacon TABLE
CREATE TABLE beacon(
  beacon_id SERIAL UNIQUE NOT NULL,
  room_number VARCHAR(45) NOT NULL,
  description VARCHAR(45) NULL,
  major_key INT NOT NULL,
  minor_key SERIAL UNIQUE NOT NULL,
  zone_name VARCHAR(45) NOT NULL
)
```

iBeacon Protocol Requirements Abstraction

To increase usability, I hid the complexity of the major and minor key values from users. I did this by ensuring that all major and minor keys are assigned automatically and uniquely by using the Postgresql SERIAL macro.

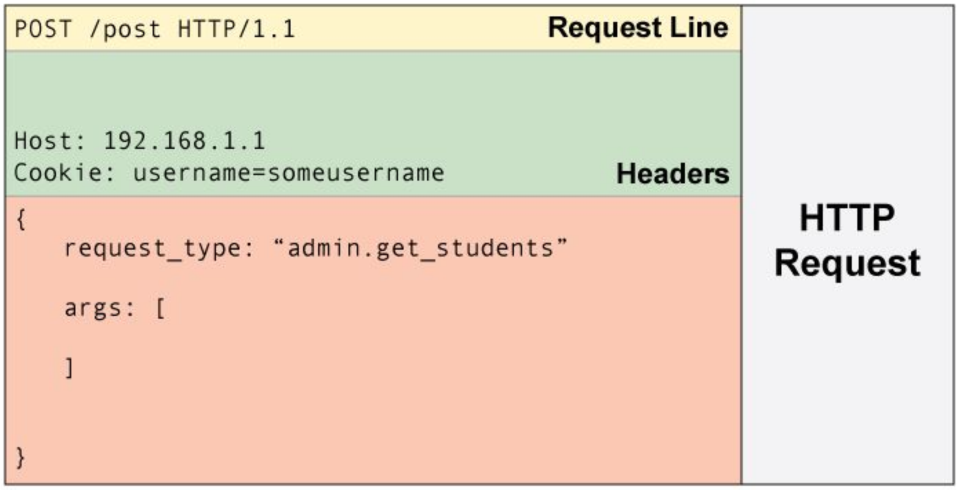


Middleware Levels of Abstraction

To make my program more friendly to other developers, I introduced levels of abstraction in my code. I did this by distinctly separating the server from the request handlers from the persistent data communicators.

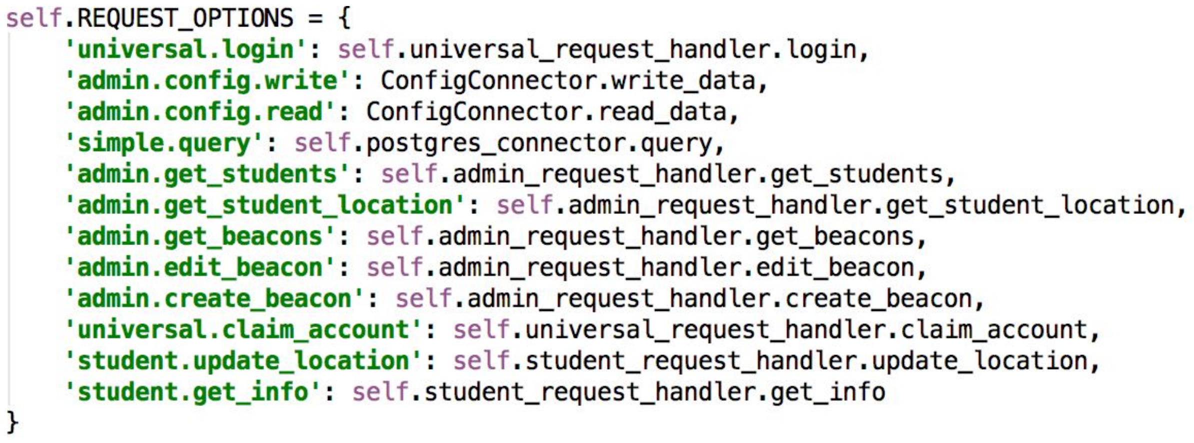
Custom Communication Protocols

HTTP Post Request Protocol



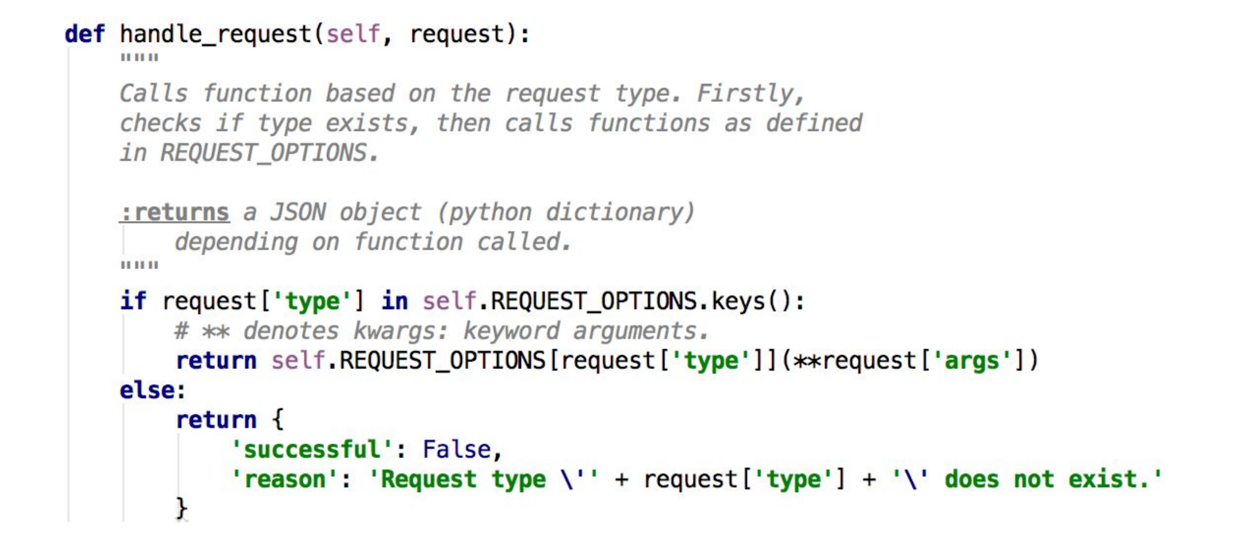
```
POST /post HTTP/1.1

Host: 192.168.1.1
Cookie: username=someusername
{
    request_type: “admin.get_students”

    args: [

    ]

}
```

I created a custom high-level protocol used for communication between the client devices and the bottle.py server. This protocol defined a distinct number of possible request types, which would be placed in POST request parameters upon making a request from a client.



```
self.REQUEST_OPTIONS = {
    'universal.login': self.universal_request_handler.login,
    'admin.config.write': ConfigConnector.write_data,
    'admin.config.read': ConfigConnector.read_data,
    'simple.query': self.postgres_connector.query,
    'admin.get_students': self.admin_request_handler.get_students,
    'admin.get_student_location': self.admin_request_handler.get_student_location,
    'admin.get_beacons': self.admin_request_handler.get_beacons,
    'admin.edit_beacon': self.admin_request_handler.edit_beacon,
    'admin.create_beacon': self.admin_request_handler.create_beacon,
    'universal.claim_account': self.universal_request_handler.claim_account,
    'student.update_location': self.student_request_handler.update_location,
    'student.get_info': self.student_request_handler.get_info
}
```

The set of possible requests is dictated by the instance dictionary REQUEST_OPTIONS by providing the request_name as a key and the function to be called as the value.



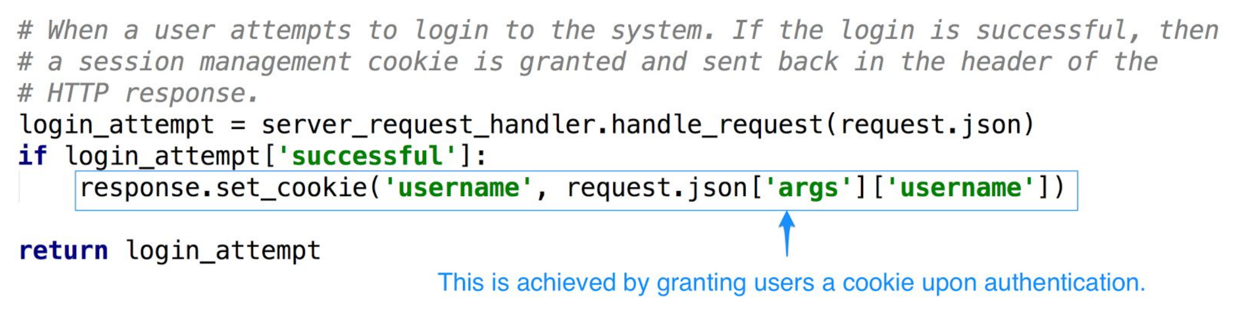
```python
    def handle_request(self, request):
        """
        Calls function based on the request type. Firstly,
        checks if type exists, then calls functions as defined
        in REQUEST_OPTIONS.

        :returns a JSON object (python dictionary)
            depending on function called.
        """
        if request['type'] in self.REQUEST_OPTIONS.keys():
            # ** denotes kwargs: keyword arguments.
            return self.REQUEST_OPTIONS[request['type']](**request['args'])
        else:
            return {
                'successful': False,
                'reason': 'Request type \'' + request['type'] + '\' does not exist.'
            }
```

Requests received would then be passed to the ServerRequestHandler's handle_request() method.

This method is an example of polymorphism, in that it takes a variety of inputs and dynamically routes them to their corresponding destinations. This saves time while programming because a specific case does not have to be made for each type of request.

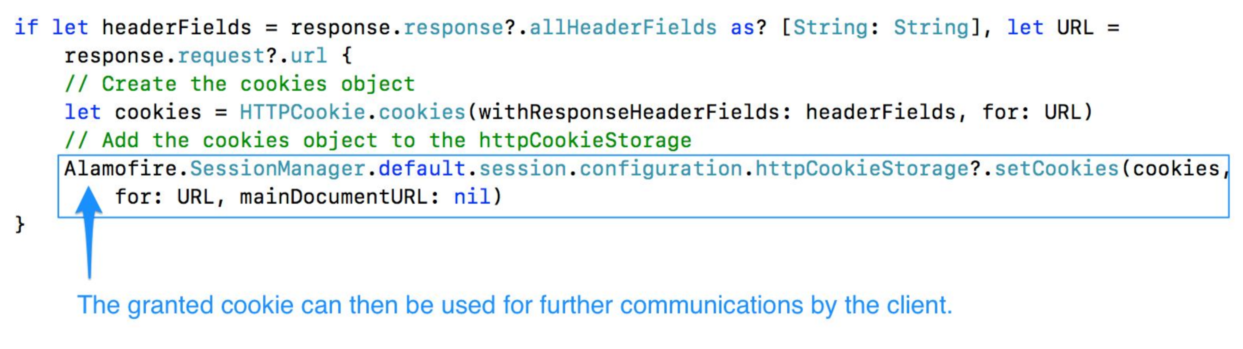
Session Management Protocol

The server communication protocol, HTTP, is sessionless; therefore, it is necessary to create a way to store the sessions of clients so that they are not required to authenticate more than once.



```python
# When a user attempts to login to the system. If the login is successful, then
# a session management cookie is granted and sent back in the header of the
# HTTP response.
login_attempt = server_request_handler.handle_request(request.json)
if login_attempt['successful']:
    response.set_cookie('username', request.json['args']['username'])

return login_attempt
```

This is achieved by granting users a cookie upon authentication.

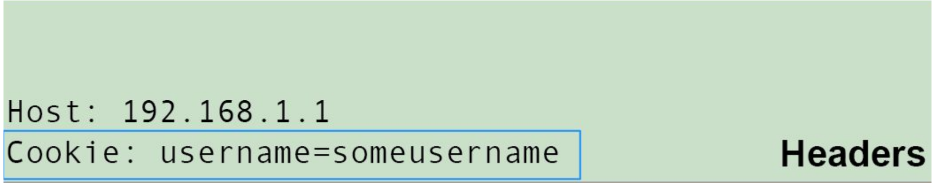


```swift
if let headerFields = response.response?.allHeaderFields as? [String: String], let URL =
    response.request?.url {
    // Create the cookies object
    let cookies = HTTPCookie.cookies(withResponseHeaderFields: headerFields, for: URL)
    // Add the cookies object to the httpCookieStorage
    Alamofire.SessionManager.default.session.configuration.httpCookieStorage?.setCookies(cookies,
        for: URL, mainDocumentURL: nil)
}
```

The granted cookie can then be used for further communications by the client.

To do this, it is added to the httpCookieStorage, which is natively built into the client. All future requests sent by the client will now be headed by the cookie:

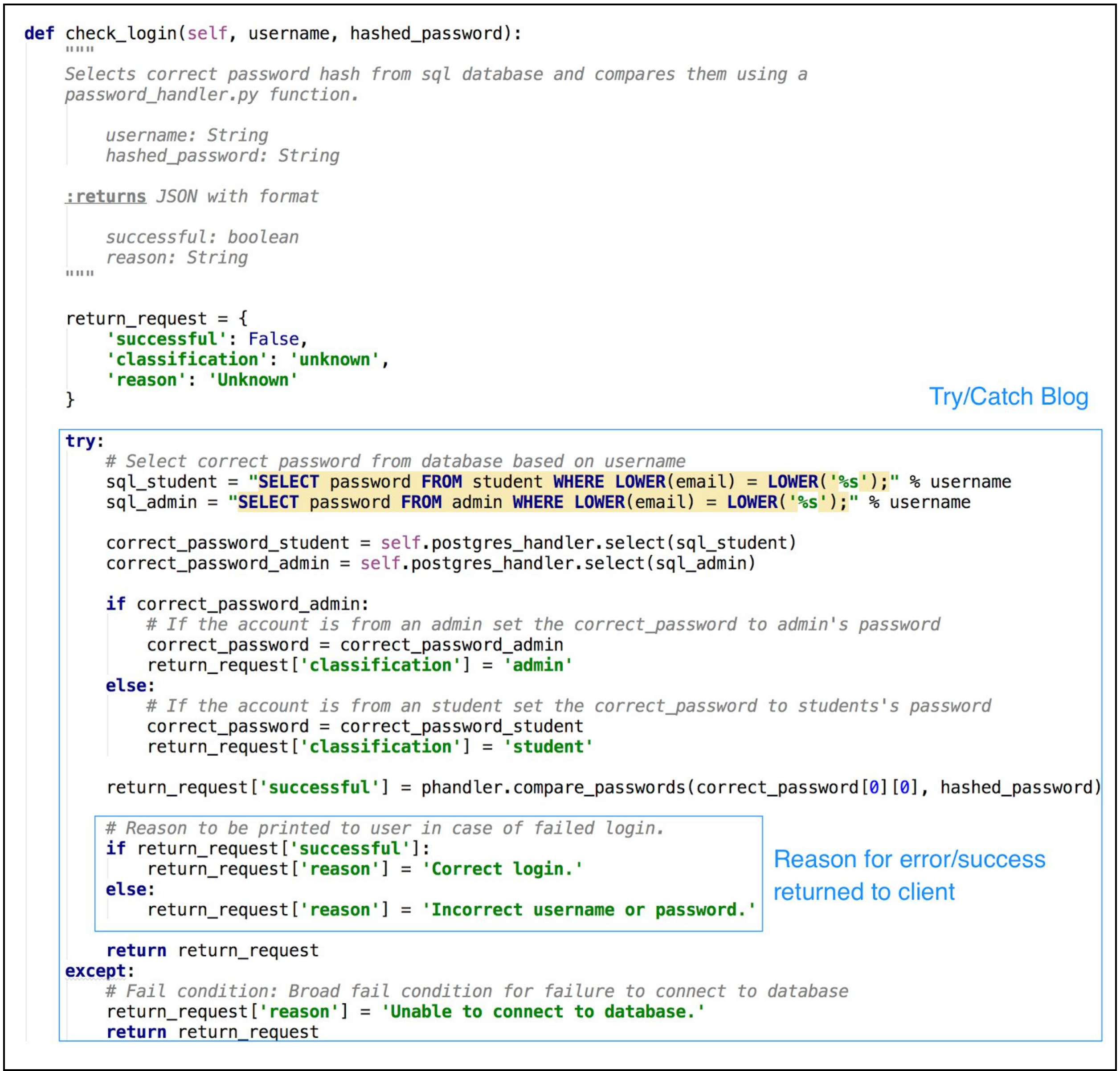
HTTP Request



```
Host: 192.168.1.1
Cookie: username=someusername
```

Error Handling

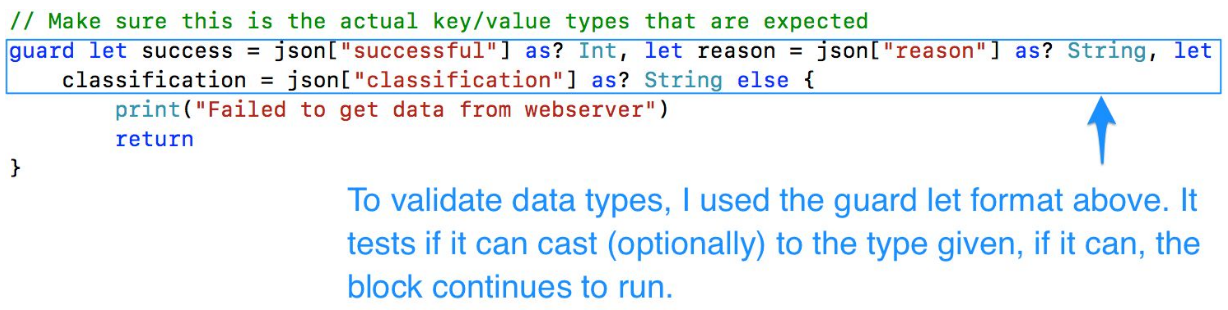
Middleware



```python
def check_login(self, username, hashed_password):
    """
    Selects correct password hash from sql database and compares them using a
    password_handler.py function.

        username: String
        hashed_password: String

    :returns JSON with format

        successful: boolean
        reason: String
    """

    return_request = {
        'successful': False,
        'classification': 'unknown',
        'reason': 'Unknown'
    }

    try:
        # Select correct password from database based on username
        sql_student = "SELECT password FROM student WHERE LOWER(email) = LOWER('%s');" % username
        sql_admin = "SELECT password FROM admin WHERE LOWER(email) = LOWER('%s');" % username

        correct_password_student = self.postgres_handler.select(sql_student)
        correct_password_admin = self.postgres_handler.select(sql_admin)

        if correct_password_admin:
            # If the account is from an admin set the correct_password to admin's password
            correct_password = correct_password_admin
            return_request['classification'] = 'admin'
        else:
            # If the account is from an student set the correct_password to students's password
            correct_password = correct_password_student
            return_request['classification'] = 'student'

        return_request['successful'] = phandler.compare_passwords(correct_password[0][0], hashed_password)

        # Reason to be printed to user in case of failed login.
        if return_request['successful']:
            return_request['reason'] = 'Correct login.'
        else:
            return_request['reason'] = 'Incorrect username or password.'

        return return_request
    except:
        # Fail condition: Broad fail condition for failure to connect to database
        return_request['reason'] = 'Unable to connect to database.'
        return return_request
```

The middleware handles errors, most often, with a try/catch block. In the event of an error, the reason is returned to the client either to be processed or displayed. This increases usability because if type errors happen on the server, the client would be otherwise unaware.

Client



```
// Make sure this is the actual key/value types that are expected
guard let success = json["successful"] as? Int, let reason = json["reason"] as? String, let
    classification = json["classification"] as? String else {
        print("Failed to get data from webserver")
        return
}
```

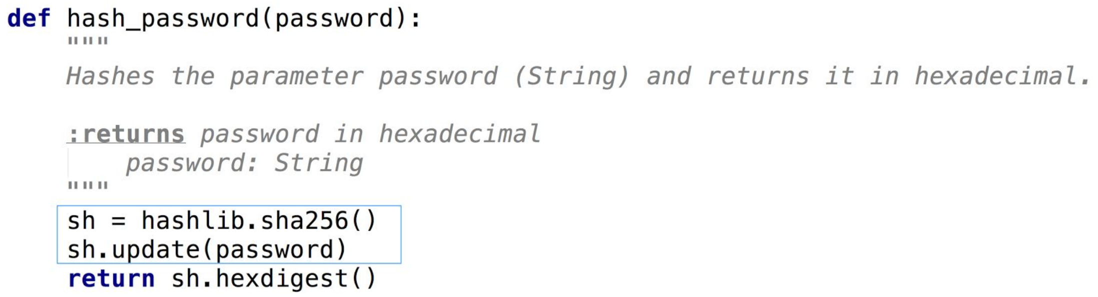
To validate data types, I used the guard let format above. It tests if it can cast (optionally) to the type given, if it can, the block continues to run.

The client handles and prevents errors by checking for failure and validating data.

This makes the program more extensible because if another developer decides to change data types the program will not crash.

Security Considerations

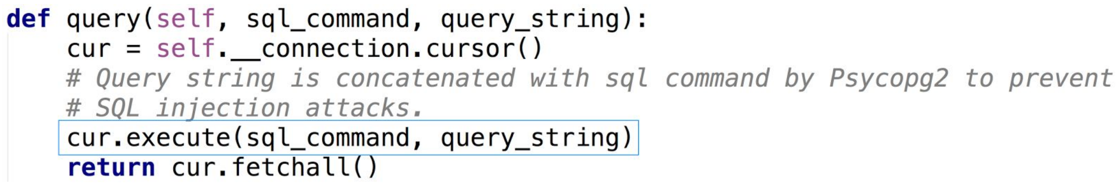
Password Hashing



```
def hash_password(password):
    """
    Hashes the parameter password (String) and returns it in hexadecimal.

    :returns password in hexadecimal
        password: String
    """
    sh = hashlib.sha256()
    sh.update(password)
    return sh.hexdigest()
```

It is proper practice to hash all passwords before placing them in a database; consequently, I used hashlib to hash passwords.

SQL Injection Prevention

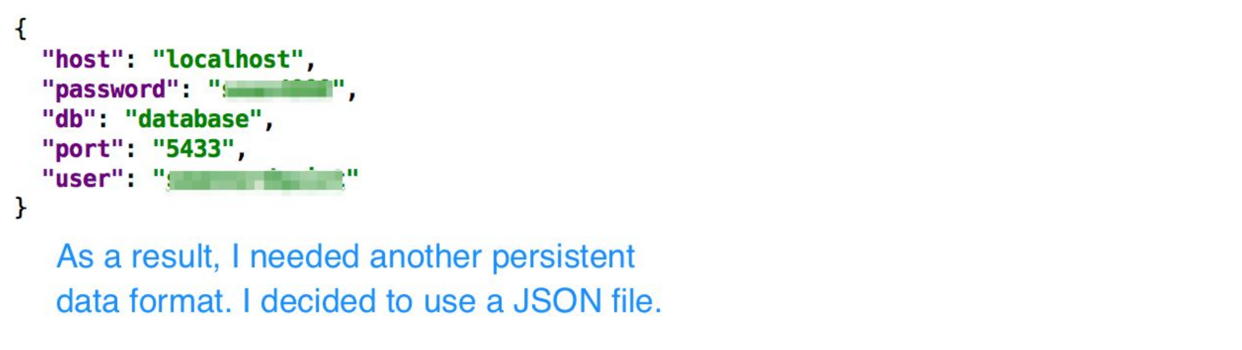


```
def query(self, sql_command, query_string):
    cur = self.__connection.cursor()
    # Query string is concatenated with sql command by Psycopg2 to prevent
    # SQL injection attacks.
    cur.execute(sql_command, query_string)
    return cur.fetchall()
```

A concern when using a SQL database with user provided queries is that it is vulnerable to an SQL injection attack. To remedy this, I used the library Psycopg2, which comes with built in protection when variables are passed correctly.

Config File Reading and Writing

To increase the extensibility of the middleware and database, I created a configuration file that dictates how to connect to the database. This is important because if an IP address changes or a password changes, the server would otherwise be unable connect.

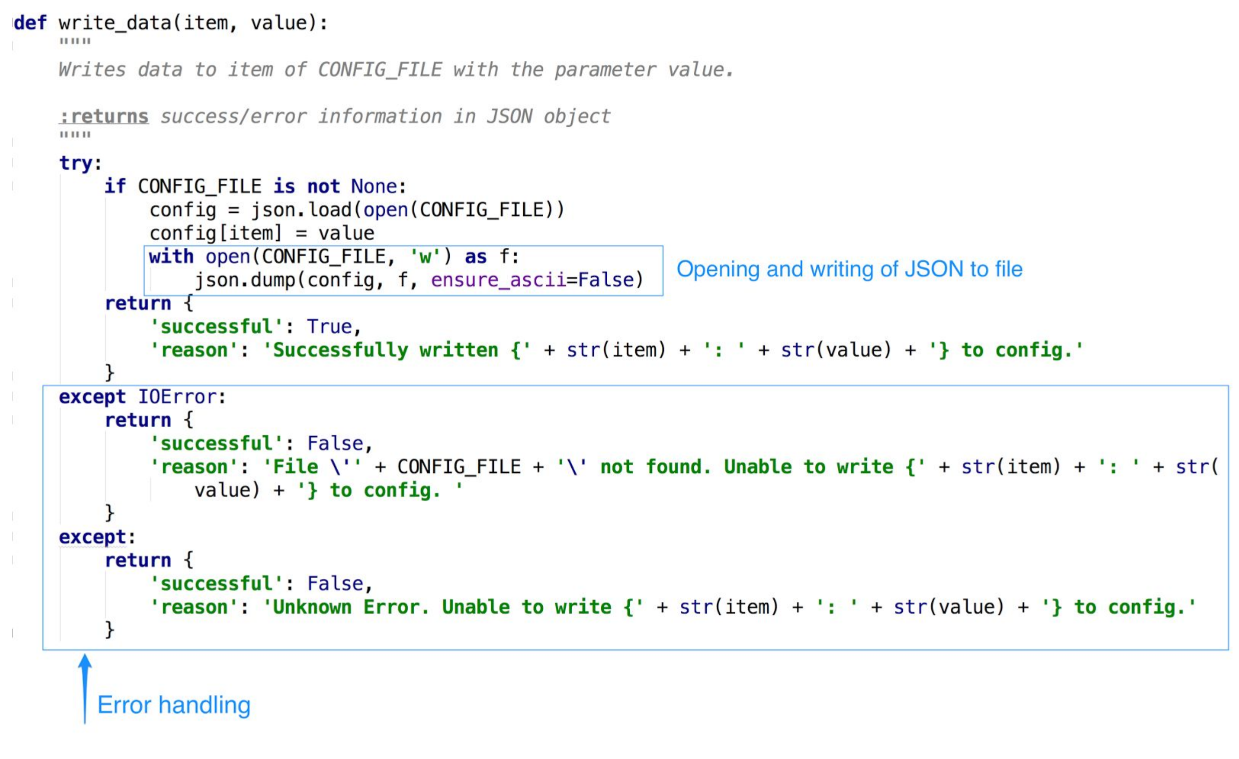


```
{
  "host": "localhost",
  "password": "[redacted]",
  "db": "database",
  "port": "5433",
  "user": "[redacted]"
}
```

As a result, I needed another persistent data format. I decided to use a JSON file.

I needed a way to read and write programmatically. I did this with the functions below.

Write Function

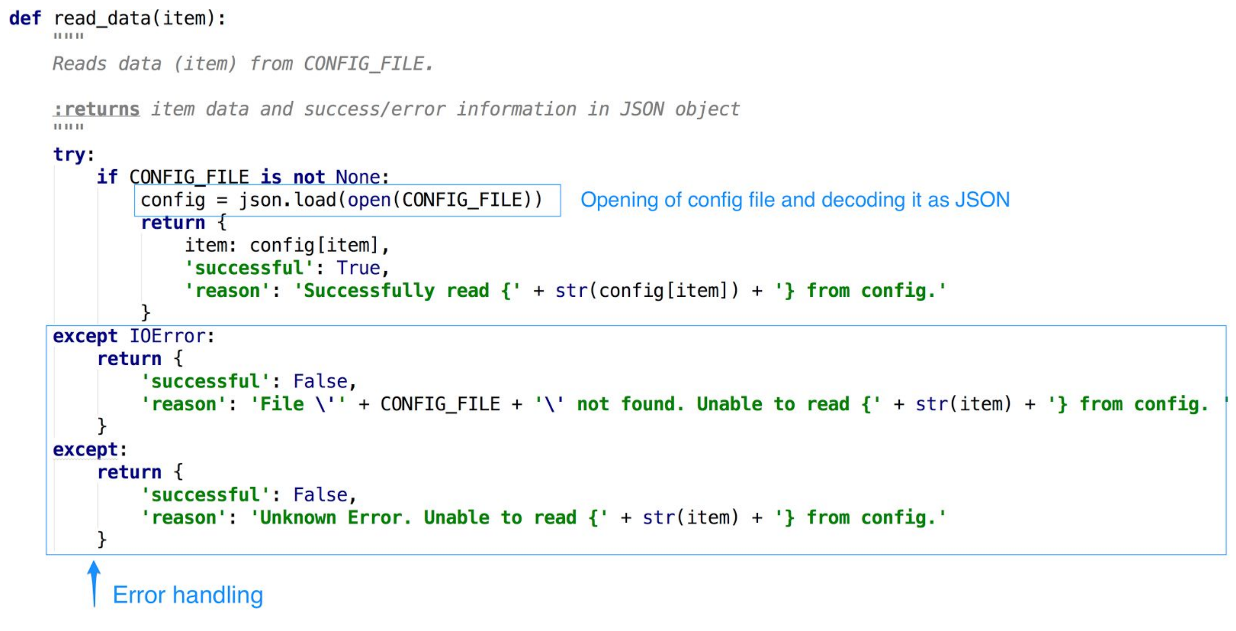


```
def write_data(item, value):
    """
    Writes data to item of CONFIG_FILE with the parameter value.

    :returns success/error information in JSON object
    """
    try:
        if CONFIG_FILE is not None:
            config = json.load(open(CONFIG_FILE))
            config[item] = value
            with open(CONFIG_FILE, 'w') as f:
                json.dump(config, f, ensure_ascii=False)
        return {
            'successful': True,
            'reason': 'Successfully written {' + str(item) + ': ' + str(value) + '} to config.'
        }
    except IOError:
        return {
            'successful': False,
            'reason': 'File \'' + CONFIG_FILE + '\' not found. Unable to write {' + str(item) + ': ' + str(
                value) + '} to config. '
        }
    except:
        return {
            'successful': False,
            'reason': 'Unknown Error. Unable to write {' + str(item) + ': ' + str(value) + '} to config.'
        }
```

Opening and writing of JSON to file

Error handling

Read Function



```
def read_data(item):
    """
    Reads data (item) from CONFIG_FILE.

    :returns item data and success/error information in JSON object
    """
    try:
        if CONFIG_FILE is not None:
            config = json.load(open(CONFIG_FILE))
            return {
                item: config[item],
                'successful': True,
                'reason': 'Successfully read {' + str(config[item]) + '} from config.'
            }
    except IOError:
        return {
            'successful': False,
            'reason': 'File \'' + CONFIG_FILE + '\' not found. Unable to read {' + str(item) + '} from config. '
        }
    except:
        return {
            'successful': False,
            'reason': 'Unknown Error. Unable to read {' + str(item) + '} from config.'
        }
```

Opening of config file and decoding it as JSON

Error handling

Works Cited:

Robinson, Matthew (2013) BeaconOSX [Objective-C Source Code] https://github.com/mttrb/BeaconOSX