# Pacemaker DCM (Group 10)

Graham Power, Hamza Ashraf

Nov 26, 2020

# CONTENTS:

CHAPTER

# ONE

# FLASKAPP APP

## 1.1 Main Application

The main implementation of the DCM flaskapp. Handles rendering off all the endpoints as well as communication between the frontend and backend.

`app.`**`home`**`()`

home The route to the homepage of the flask application

Renders the homepage of the flask app and manages post requests. Supported post request are 'login' and 'account creation' requests, which when completed successfully will redirect the user to their user specific page. Any failed post request will result in an error message flashed to the screen.

**Returns** The render template used by the homepage, in this case the 'index.html' template

**Return type** `flask.render_tmeplate`

`app.`**`logout`**`()`

logout The route to the logout page of the flask application

This page acts as an intermediary between a user page that requires login and the homepage of this application. This page logs the user out and immediately redirects to the homepage, flashing a logout message after the redirect.

**Returns** A redirect to the homepage of the application

**Return type** `flask.redirect`

`app.`**`open_browser`**`()`

open_browser Opens the flask app automatically in the web brower.

So the user doesn't have to memorize the url and port the application is being hosted on.

`app.`**`random`**`()` → x in the interval [0, 1).

`app.`**`user_connect`**`()`

user_connect The route to the user connect page of the flask application

Renders the user connect page of the flask app. This page allows the user to view the status of the pacemaker in different pacing modes. Pacing modes can be changed via a selection box at the top of the page. NOTE: This page will correctly change between different pacing mode states, though these states and their corresponding rendered templates are black as no serial communication with the pacemaker has been implemented. As a result changing modes will have little visible effect on the rendered content of the application.

**Returns** The render template used by the user connect page, in this case the 'user_connect.html' template

**Return type** `flask.render_tmeplate`

`app.`**`user_data`**`()`

user_data The route to the data request uri for the egram

Will return a json response containing updated point values to be added to the live graph.

**Returns** A json response, containing updated point values

**Return type** json

`app.`**`user_egram`**`()`

user_egram The route to the user history page of the flask application

Renders the user egram page of the flask app. This page allows the user to view a live graph (or egram) of the pacemakers atrium and ventrical chambers. NOTE: The graph displayed is an adaptable graph and will take up to one minute to adjust both the graphs domain and range values.

**Returns** The render template used by the user connect page, in this case the 'user_egram.html' template

**Return type** `flask.render_tmeplate`

`app.`**`user_history`**`()`

user_history The route to the user history page of the flask application

Renders the user history page of the flask app. This page allows the user to view the entire history of their pacemaker parameters in a convenient table. This table can be exported to a csv file if the user wishes, which will appear in the downloads file.

**Returns** The render template used by the user connect page, in this case the 'user_history.html' template

**Return type** `flask.render_tmeplate`

`app.`**`user_page`**`()`

user_page The route to the user specific page of the flask application

Renders the user specific page of the flask app. This page acts as an intermediary between the 'login' state and any other states which require login, such as the user parameters and pacing mode specific states.

**Returns** The render template used by the user specific page, in this case the 'user.html' template

**Return type** `flask.render_tmeplate`

`app.`**`user_parameters`**`()`

user_parameters The route to the user parameters page of the flask application

Renders the user parameters page of the flask app. This page allows the user to view and modify their pacemaker parameters through a submittable form. Users can modify some or all parameters and submit the changes through a post request. Changes will be made immediately in the flask app and database, and the user parameters page updated with these changed values.

**Returns** The render template used by the user parameters page, in this case the 'user_parameters.html' template

**Return type** `flask.render_tmeplate`

CHAPTER

TWO

# CONFIG CONFIG_MANAGER

## 2.1 Config Manager

A collection of configuration setup classes, to initialize the config variables and setup application logging as well as allow access to both the config variables and logging function directly form anwhere within the application.

**class** `config.config_manager.`**`Config`**

The configuration class, holds all the configuration information

Manages the configuration and provides access to the config through a singleton implementation. Its get method can be called from anywhere within the application the Config class is imported.

**Raises**

- **`Exception`** – This class is a singleton!
- **`Exception`** – Configuration can only be read once!
- **`Exception`** – Configutation has not been read, read_config() must be called first!

**`get`**(*section*, *variable*)

get Gets a variable from the configuration

Given a configuration section and variable name within that section will return the value of the variable as a string

**Parameters**

- **`section`** (`str`) – The section in the configuration file to find the variable in
- **`variable`** (`str`) – The variable name to read from the section

**Raises** **`Exception`** – Configutation has not been read, read_config() must be called first!

**Returns** The string representation of the variable

**Return type** str

**static** **`getInstance`**()

getInstance Gets the current instance of Config

If no instance exists, it will create the initial instance before returning it.

**Returns** A singleton instance of the Config class

**Return type** `config_manager.Config`

**`get_all`**(*section*)

get_all Gets all variable and values within a section

Given a configuration section from the configuration files will return a dictionary containing all variable names and values from within that section. All varaible names and values are stored in their string representation within the dictionary.

**Parameters** **`section`** (*`str`*) – The section of the configuration file to read

**Raises** **`Exception`** – Configutation has not been read, read_config() must be called first!

**Returns** A dictionary conatianing variable name, value pairs

**Return type** dict

**`getboolean`**(*section*, *variable*)

getboolean Gets a variable from the configuration

Given a configuration section and variable name within that section will return the value of the variable as a boolean. The value in the configuration file can only be in the form [yes/no, true/false, on/off, 1/0], for any other values this method will throw an error

**Parameters**

- **`section`** (*`str`*) – The section of the configuration file to find the variable in
- **`variable`** (*`str`*) – The variable to read form the section

**Raises** **`Exception`** – Configuration has not been rean, read_config() must be called first!

**Returns** The boolean representation of the variable

**Return type** bool

**`read_config`**(*cfg_files=['D:\\School\\Year 3\\Semester 1\\3K04\\PacemakerProject\\DCM\\flaskapp\\config\\application.ini']*)

read_config Reads the configuration files

Called on instance initialization. Will search the config directory for all .ini files and read their variable values into the Config Manager. Raises an exception if called more than once, to ensure variables are not changed mid execution.

**Parameters** **`cfg_files`** (*`list, optional`*) – A list of additional ini files to read, application.ini is added automatically, defaults to []

**Raises** **`Exception`** – Configuration can only be read once!

**class** `config.config_manager.`**`Logger`**

The Logger class, holds all logging information

This class can initialize a logger through a singleton implementation. Its method can be called from anywhere within the application where the Logger class is imported.

**Raises**

- **`Exception`** – This class is a singleton!
- **`Exception`** – Logger can only be started once!

**static getInstance**()

getInstance Get the current instance of Logger

If no instance exists it will create the initial instance before returning it

**Returns** A singleton instance of the Logger class

**Return type** `config_manager.Logger`

**`log`**(*level*, *msg*)

log Will log a message

Given a message and the level of that message, this method will log the message to all the Loggers handlers (i.e. file and terminal) only if the log level is equal to or higher than the Logger's log level defined in the applications configuration.

**Parameters**

- **`level`** (*`str`*) – The level of the message being logged can take the values: DEBUG, INFO, WARN, ERROR, CRITICAL
- **`msg`** (*`str`*) – The message to be logged by the Logger

**`start_logger`**(*config*)

start_logger Starts the logger

This method must be called by the main block of the application. It will create the log file and start the logging process both to the log file and the terminal

**Parameters** **`config`** (`config_manager.Config`) – The Instance of Config, which must be created and initialized before the Logger is started

**Raises** **`Exception`** – Logger can only be started once!

CHAPTER

# THREE

# CONFIG DECORATORS

## 3.1 Decorators Library

A collection of decorators used by the main app, created to avoid the unnecessary reproduction of code.

`config.decorators.`**`login_required`**(*f*)

login_required Ensures that a user is logged in before granting access to user restricted pages

Will check if a user is logged in and only allow a redirect to the user restricted page if there is a user logged in. If no user is logged in it will chenge the redirect to the home page and flash a login required messege, letting the user know they must login before accessing that endpoint. NOTE: This function should never be called as a function, only used as a decorator above functions who require its functionality to be implemented on entry. Always use a decorator (i.e. @login_required) to invoke this function.

**Parameters** **f** (*function*) – The function being decorated. This will be automatically filled when using the @ tag

**Returns** The wrap function, whose contents are the input function, modified to add the functionality of this decorator

**Return type** function

`config.decorators.`**`logout_required`**(*f*)

logout_required Ensures that a user is logged out before granting access to non user restricted pages

Will check if a user is logged out and only allow a redirect to the non user restricted page if there is no user logged in. If a user is logged in it will simply log them out automatically before redirecting to the requested endpoint. No logout action on the part of the user is necessary. NOTE: This function should never be called as a function, only used as a decorator above functions who require its functionality to be implemented on entry. Always use a decorator (i.e. @logout_required) to invoke this function.

**Parameters** **f** (*function*) – The function being decorated. This will be automatically filled when using the @ tag

**Returns** The wrap function, whose contents are the input function, modified to add the functionality of this decorator

**Return type** function

CHAPTER

# FOUR

# DATA DATABASE

## 4.1 Database Library

A collection of functions capable of interacting with a sqlite3 single file databse. NOTE: Users are returned as lists, whose entries are in the following order

[ _userid, username, password, LowerRateLimit, UpperRateLimit, AtrialAmplitude, AtrialPulseWidth, AtrialRefractoryPeriod, VentricularAmplitude, VentricularPulseWidth, VentricularRefractoryPeriod ]

`data.database.`**`find_user`**(*cursor*, *username=None*, *password=None*)

find_user Given search parameters will find all matching users in the database

Given one or more of the optional search parameters will return a list of all users matching that search criteria. Accepted search parameters are username and password. If neither optional parameters are given, the function will return None

**Parameters**

- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database to search for the user in
- **`username`** (*`str, optional`*) – The username of the user to search for, defaults to None
- **`password`** (*`str, optional`*) – The password of the user to search for, defaults to None

**Returns** A list of tuples containing all users matching the search query

**Return type** list

`data.database.`**`get_rows`**(*cursor*)

get_rows Returns the number of rows (.i.e users) in the database

**Parameters** **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database

**Returns** the number of rows contained in the user table of the database

**Return type** int

`data.database.`**`get_user`**(*cursor*, *id*)

get_user Returns a complete users information given their unique ID

Return type is a list of users. Since the search is done by unique ID, this list is garunteed to be either of length one, if a user with matching unique ID is found, or zero, if no user with matching unique ID is found.

**Parameters**

- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database the user can be found in
- **`id`** (*`int`*) – The unique ID of the user to search for

**Returns** A list of tuples containing the contents of the first item matching the search query

**Return type** list

`data.database.`**`get_user_history`**(*cursor*, *id*)

get_user_history Returns a complete list of all the users past parameters

Return type is a list of all past parameters, including the current ones. This function has no garunteed list size, as it depends on how many history entries the user has made.

**Parameters**

- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database the user can be found in
- **`id`** (*`int`*) – The unique ID of the user to search for

**Returns** A list of all the users past pacemaker parameters

**Return type** list

`data.database.`**`get_user_parameters`**(*cursor*, *id*)

get_user_parameters Returns a complete list of the users most recent parameters

Return type is a list of parameters. Since the search is done by unique ID, this list is garunteed to be of constant length, defined by the parameter list in the application configuration.

**Parameters**

- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database the user can be found in
- **`id`** (*`int`*) – The unique ID of the user to search for

**Returns** A list of the users current pacemaker parameters

**Return type** list

`data.database.`**`init_db`**(*file*)

init_db Initializes a database located at a given file location

The file location should be specified relative to the ~/3K04-Pacemaker/DCM/flaskapp/data directory. The file should also have a supported sqlite3 extension (.db .db3 .sdb .s3db .sqlite .sqlite3) and if the file does not already exist it will be created and populated with a new databases.

**Parameters** **`file`** (*`str`*) – The relative file location of the single file sqlite3 database

**Returns** A tuple containing the databases connection handler and cursor (sqlite3.Connection, sqlite3.Cursor)

**Return type** tuple

`data.database.`**`insert_user`**(*conn*, *cursor*, *username*, *password*)

insert_user Given a username and password of a new user, will insert the user into the database

This function will create a new entry in the database of a user with the given username and password. Only the users username and password are initialized upon user creation, the pacemaker parameters will default to None, forcing the user to manually enter their parameters. NOTE: This function does no check for conflicting users in the database before inserting a new user. It is up to the user of this function to check for conflicts (if they wish to do so) before calling this function.

**Parameters**

- **`conn`** (`sqlite3.Connection`) – The connection handler for the database to insert a new user into

- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database to insert a new user into
- **`username`** (*`str`*) – The username for the new user
- **`password`** (*`str`*) – The password for the new user

`data.database.`**`update_pacemaker_parameters`**`(`*conn*, *cursor*, *id*, *values*`)`

update_pacemaker_parameters Given a list of pacemaker parameters, updates the database values

When given handler to the database and the unique ID of the user being affected, will update the users pacemaker parameters to match the input list. NOTE: No complete check is done to ensure the validity of the input, it is up to the method user to ensure the lists correctness.

**Parameters**

- **`conn`** (`sqlite3.Connection`) – The connection handler for the database whos contents to change
- **`cursor`** (`sqlite3.Cursor`) – The cursor handler for the database whos contents to change
- **`id`** (*`int`*) – The unique ID of the user whos parameters should be changed
- **`values`** (*`list`*) – A list of pacemaker parameters, whos order matches the databases contents

CHAPTER

FIVE

# DATA USER

## 5.1 User Class

The class to represent a user of this application. Capable of initializing and accessing the database specified in the application configuration, as well as loggin in and out, creating an account, and modifying the pacemaker parameters of the currently logged in user.

**class** `data.user.`**`User`**

This is a class representation of a simple flask app user

The class to represent a user of this application. Capable of initializing and accessing the database specified in the application configuration, as well as loggin in and out, creating an account, and modifying the pacemaker parameters of the currently logged in user.

**Parameters** **`config`** (class:*configparser.ConfigParser*) – A handle to the `configparser.ConfigParser` config object initialized by the main application on startup

**`create_account`**(*username*, *password*)

create_account Creates a new user account

Checks the database to ensure no user with the same username exists, and that the maximum allowable local-agents has not been exceeded (defined in the application.ini). If no conflicts exist, a new user is created then both inserted into the database and logged in.

**Parameters**

- **`username`** (*`str`*) – The username of the new user being created
- **`password`** (*`str`*) – The password of the new user being created

**Returns** True if the account creation was successful, False otherwise

**Return type** bool

**`create_history_file`**()

create_history_file Creates a csv file containing the users parameter history

**`get_history`**()

get_history Get the users history

Gets the users curretn history by calling the database helper function

**Returns** A list fo the users history, including the current pacemaker values

**Return type** list

**`get_limits`**()

get_limits Get the users parameter limits

Returns a dictionary of all the users parameter limits as they were defined in the application configuration

**Returns** A dictionary of the users parameter limits

**Return type** dict

**`get_pacemaker_mode`**()
get_pacemaker_mode Returns the current pacemaker mode

**Returns** The current pacemaker mode, can be either a String or None

**Return type** str

**`get_pacemaker_parameters`**()
get_pacemaker_parameters Returns a dictionary of this users pacemaker parameters

**Returns** A dictionary of this users pacemaker parameters

**Return type** dict

**`get_username`**()
get_username Returns this users username

**Returns** This users username

**Return type** str

**`is_loggedin`**()
is_loggedin Checks if this user is logged in

**Returns** True if the user is logged in, False otherwise

**Return type** bool

**`login`**(*username*, *password*)
login Attempts to log a user in, given their username and password

Searches the database to check if the user with matching username and password exists. No conflict management (i.e. ensuring only one user matches that username) is necessary since it is handled on account creation. If a matching user is found the `data.user` is updated with that users information and the user is logged in.

**Parameters**

- **`username`** (*`str`*) – The username of the user trying to login
- **`password`** (*`str`*) – The password of the user trying to login

**`logout`**()
logout Logs out the currently logged in user

**`update_all_pacemaker_parameters`**(*values*)
update_all_pacemaker_parameters Updates all pacemaker parameters

Given a list of pacemaker parameters, in the same order as the values of the pacemaker parameters dictionary, will update every value of the dictionary. NOTE: No complete check is done to ensure the validity of the input, it is up to the method user to ensure the lists correctness.

**Parameters** **`values`** (*`list`*) – An ordered list of the updated pacemaker parameters

**Returns** True if the pacemaker parameters were updated sucessfully, False otherwise

**Return type** bool

**`update_pacemaker_mode`**(*mode*)
update_pacemaker_mode Changes the pacemaker mode

Given a mode that is contained in the application configurations list of allowed modes, will change the current pacemaker mode to the new one. If no valid mode is given will do nothing and return False.

**Parameters** **`mode`** (*`str`*) – The pacemaker mode to change to

**Returns** True if the pacemaker mode was successfully changed, Fale otherwise

**Return type** bool

**`update_pacemaker_parameter`**(*key*, *value*)

update_pacemaker_parameter Updates a single pacemaker parameter

Given a valid key (one already contained in the pacemaker parameters dictionary), will update the value of that key with the passes in value.

**Parameters**

- **`key`** (*`str`*) – A key already contained in the pacemaker parameters dictionary
- **`value`** (*`int`*) – An updated value for the associated key

**Returns** True if the parameter was sucessfully updated, False otherwise

**Return type** bool

CHAPTER

SIX

# GRAPHS GRAPHING

## 6.1 Graphing Library

A collection of graphiung functions used by the app to generate both data points to be rendered live, as well as publish a graphs history to a csv file.

`graphs.graphing.`**`publish_data`**`(`*username*`)`

publish_data Publishes the current (or most recent) live graphs data to a csv file

Will generate a csv file containing all the data from when the current (or most recent) graph was started.

`graphs.graphing.`**`random`**`()` → x in the interval [0, 1).

`graphs.graphing.`**`set_start_time`**`()`

set_start_time Sets the start time for the graph

Called when starting a new graph, will reset the running timer allowing the new graph to start at the zero value.

`graphs.graphing.`**`temp_serial_placeholder`**`()`

temp_serial_placeholder Placeholder for serial functionality

Generates a sine and cosine point output to be rendered to the live graphs to test their functionality without relying on the serial communications.

**Returns** A list containing a sine and cosine point in this format [ sine(current_time), cos(current_time)]

**Return type** list

`graphs.graphing.`**`update_data`**`()`

update_data Returns a new data point for the graph

Called to request a new data point for the live graph. Will return a list of lists, each internal list containing a point with a timestamp (x-value) and parameter value (y-value). The timestamp is determined by the time difference between 'now' and when the set_start_time() function was last called.

**Returns** A list of lists containing a new point to be added to each line being rendered by the live graph

**Return type** list

CHAPTER

SEVEN

# TESTS TESTS

# CHAPTER EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX