Carry

Release 0.0.1

Group 13

May 24, 2020

carry

1.1 carry package

1.1.1 Submodules

1.1.2 carry.database_utils module

class `carry.database_utils.`**`DatabaseUtils`**

Bases: `object`

Database Class

Note This class only contains methods for database connection.

`addCalendarEventId` (*db*, *booking_id*, *eid*)

A function for adding a Google Calendar Event ID

Parameters
- **db** (*Database object, essential*) – Database
- **booking_id** (*int, essential*) – Booking ID
- **eid** (*int, essential*) – Google Calendar Event ID

Returns Booking Object

Return type dictionary

`cancelBooking` (*db*, *booking_id*)

A function for cancelling a booking

Parameters
- **db** (*Database object, essential*) – Database
- **booking_id** (*int, essential*) – Booking ID

Returns Booking Object

Return type dictionary

`createNewCar` (*db*, *make*, *body_type*, *colour*, *seats*, *cost*, *address*, *suburb*, *state*, *postcode*, *lat*, *lng*)

A function for creating a new car object in the database

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`make`** (*`string, essential`*) – Car Make
- **`body_type`** (*`string, essential`*) – Car Body_type
- **`colour`** (*`string, essential`*) – Car Colour
- **`seats`** (*`int, essential`*) – No. Car Seats
- **`cost`** (*`float, essential`*) – Car Hourly Cost
- **`address`** (*`String, essential`*) – Car Address
- **`suburb`** (*`String, essential`*) – Suburb
- **`state`** (*`String, essential`*) – State
- **`postcode`** (*`String, essential`*) – Postcode
- **`lat`** (*`float, essential`*) – Latitude
- **`lng`** (*`float, essential`*) – Longitude

Returns Car object if succesfully created, *None* otherwise

Return type dictionary

`findUser` (*db*, *username*)

A function for finding user by searching username or user email

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`username`** (*`string, essential`*) – Username / User email

Returns User object if succesfully found, *None* otherwise

Return type dictionary

`getAllCars` (*db*, *search_input*)

A function for getting cars by searching features

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`search_input`** (*`string, essential`*) – Search Input

Returns List of Car if succesfully found, an empty list otherwise

Return type list of dictionary

`getBookedCars` (*db*, *start_datetime*, *end_datetime*)

A function for getting booked cars by searching start_datetime and end_datetime

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`start_datetime`** (*`datetime, essential`*) – Start Datetime
- **`end_datetime`** (*`datetime, essential`*) – End Datetime

Returns List of Booked Car if succesfully found, an empty list otherwise

Return type list of dictionary

`getBooking` (*db*, *booking_id*)

A function for getting booking by searching booking_id

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`booking_id`** (*`int, essential`*) – Booking ID

Returns Booking object if succesfully found, *None* otherwise

Return type dictionary

`getCar` (*db, car_id*)

A function for getting car by searching car_id

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`car_id`** (*`int, essential`*) – Car ID

Returns Car object if succesfully found, *None* otherwise

Return type dictionary

`getCurrentBooking` (*db, user_id, car_id, now*)

A function for getting a current booking assigned to specific car and user

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`user_id`** (*`int, essential`*) – User ID
- **`car_id`** (*`int, essential`*) – Car ID
- **`now`** (*`datetime, essential`*) – Current Datetime

Returns Booking if succesfully found, None otherwise

Return type list of dictionary

`getMyBookings` (*db, user_id*)

A function for getting all bookings made by an user

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`user_id`** (*`int, essential`*) – User ID

Returns List of Bookings if succesfully found, an empty list otherwise

Return type list of dictionary

`getMyPastBookings` (*db, user_id, now*)

A function for getting all past bookings which are cancelled or finished made by an user

Parameters
- **`db`** (*`Database object, essential`*) – Database
- **`user_id`** (*`int, essential`*) – User ID
- **`now`** (*`datetime, essential`*) – Current Datetime

Returns List of Past Bookings if succesfully found, an empty list otherwise

Return type list of dictionary

`getUser` (*db, user_id*)

A function for getting user by searching user_id

Parameters
- **`db`** (*`Database object, essential`*) – Database

- **user_id** (*int, essential*) – User ID

Returns User object if succesfully found, *None* otherwise

Return type dictionary

`logging` (*db, user_id, car_id, booking_id, lat, lng, status, datetime*)
A function for logging user activities

Parameters
- **db** (*Database object, essential*) – Database
- **user_id** (*int, essential*) – User ID
- **car_id** (*int, essential*) – Car ID
- **booking_id** (*int, essential*) – Booking ID
- **lat** (*float, essential*) – Latitude
- **lng** (*float, essential*) – Longitude
- **status** (*string, essential*) – Car Status
- **datetime** (*datetime, essential*) – Current Datetime

Returns Log object if succesfully created, *None* otherwise

Return type dictionary

`newBooking` (*db, reference, user_id, car_id, start_datetime, end_datetime, booking_datetime, duration, total_cost*)
A function for creating a new booking object in the database

Parameters
- **db** (*Database object, essential*) – Database
- **user_id** (*int, essential*) – User ID
- **car_id** (*int, essential*) – Car ID
- **start_datetime** (*datetime, essential*) – Booking Start Datetime
- **end_datetime** (*datetime, essential*) – Booking End Datetime
- **booking_datetime** (*datetime, essential*) – Booking Made Datetime
- **duration** (*int, essential*) – Booking Duration
- **total_cost** (*float, essential*) – Booking Total Cost

Returns Booking Object

Return type dictionary

`register` (*db, username, email, hashed_password, firstname, lastname*)
A function for registering a new user in the database

Parameters
- **db** (*Database object, essential*) – Database
- **username** (*string, essential*) – Username
- **email** (*string, essential*) – User Email
- **hashed_password** (*string, essential*) – Hashed Password
- **firstname** (*string, essential*) – Firstname
- **lastname** (*string, essential*) – Lastname

Returns User object if succesfully created, *None* otherwise

Return type dictionary

`updateProfile` (*db*, *profile_url*, *firstname*, *lastname*, *location*)

A function for updating an existing user profile in the database

Parameters

- **`db`** (`Database object, essential`) – Database
- **`profile_url`** (`string, optional`) – Profile URL
- **`firstname`** (`string, essential`) – Firstname
- **`lastname`** (`string, essential`) – Lastname
- **`location`** (`string, essential`) – Location

Returns User object if succesfully created, *None* otherwise

Return type dictionary

1.1.3 carry.fetch module

`carry.fetch.`**`booking_detail_event`** ( )

Add a calendar event id to the existing booking if the user adds an event to Google Calendar.

`carry.fetch.`**`my_bookings_cancel`** ( )

Cancel a booking by a user request.

1.1.4 carry.forms module

class `carry.forms.`**`BookingForm`** (*formdata=<object object>*, **kwargs)

Bases: `flask_wtf.form.FlaskForm`

A flexible form validation for Booking Detail

Parameters **`formdata`** – Used to pass data coming from the enduser, usually request.POST or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings

`car_id` **= <UnboundField(HiddenField, ('Car ID',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`duration` **= <UnboundField(HiddenField, ('Duration',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`end_datetime` **= <UnboundField(HiddenField, ('End Date',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`start_datetime` **= <UnboundField(HiddenField, ('Start Date',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`submit` **= <UnboundField(SubmitField, ('Book',), {})>**

`user_id` **= <UnboundField(HiddenField, ('User ID',), {'validators': [<wtforms.validators.DataRequired object>]})>**

class `carry.forms.`**`LoginForm`** (*formdata=<object object>*, **kwargs)

Bases: `flask_wtf.form.FlaskForm`

A flexible form validation for Login

Parameters **`formdata`** – Used to pass data coming from the enduser, usually request.POST or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings

`password` **= <UnboundField(PasswordField, ('Password',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Length object>]})>**

`remember` **= <UnboundField(BooleanField, ('Remember Me',), {})>**

`submit` **= <UnboundField(SubmitField, ('Login',), {})>**

`username` **= <UnboundField(StringField, ('Email / Username',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

class `carry.forms.`**`NewBookingForm`** (*formdata=<object object>, **kwargs*)

Bases: `flask_wtf.form.FlaskForm`

A flexible form validation for New Booking

Parameters **`formdata`** – Used to pass data coming from the enduser, usually request.POST or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings

`car_id` **= <UnboundField(HiddenField, ('Car ID',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`duration` **= <UnboundField(HiddenField, ('Duration',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`end_datetime` **= <UnboundField(HiddenField, ('End Date',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`start_datetime` **= <UnboundField(HiddenField, ('Start Date',), {'validators': [<wtforms.validators.DataRequired object>]})>**

`submit` **= <UnboundField(SubmitField, ('Confirm',), {'validators': [<wtforms.validators.DataRequired object>]})>**

class `carry.forms.`**`RegistrationForm`** (*formdata=<object object>, **kwargs*)

Bases: `flask_wtf.form.FlaskForm`

A flexible form validation for Registration

Parameters **`formdata`** – Used to pass data coming from the enduser, usually request.POST or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings

`confirm_password` **= <UnboundField(PasswordField, ('Confirm Password',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.EqualTo object>]})>**

`email` **= <UnboundField(StringField, ('Email',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Email object>]})>**

`firstname` **= <UnboundField(StringField, ('First Name',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

`lastname` **= <UnboundField(StringField, ('Last Name',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

`password` **= <UnboundField(PasswordField, ('Password',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Length object>]})>**

`submit` **= <UnboundField(SubmitField, ('Sign Up',), {})>**

`username` **= <UnboundField(StringField, ('Username',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

`validate_email` (*email*)

A function for checking if email already exists in the database

Parameters **`email`** (`string, essential`) – Email

Returns ValidationError()

Return type function

`validate_username` (*username*)

A function for checking if username already exists in the database

Parameters **`username`** (`string, essential`) – Username

Returns ValidationError()

Return type function

class `carry.forms.`**`UpdateAccountForm`** (*formdata=<object object>*, ***kwargs*)

Bases: `flask_wtf.form.FlaskForm`

A flexible form validation for Profile Update

Parameters **`formdata`** – Used to pass data coming from the enduser, usually request.POST or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings

`email` **= <UnboundField(StringField, ('Email',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Email object>]})>**

`firstname` **= <UnboundField(StringField, ('First Name',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

`lastname` **= <UnboundField(StringField, ('Last Name',), {'validators': [<wtforms.validators.DataRequired object>, <wtforms.validators.Regexp object>]})>**

`location` **= <UnboundField(SelectField, ('Location',), {'choices': [('melbourne', 'Melbourne')], 'validators': [<wtforms.validators.DataRequired object>]})>**

`profile_url` **= <UnboundField(FileField, ('Update Profile Picture',), {'validators': [<flask_wtf.file.FileAllowed object>]})>**

`submit` **= <UnboundField(SubmitField, ('Update',), {})>**

`username` **= <UnboundField(StringField, ('Username',), {'validators': [<wtforms.validators.DataRequired object>]})>**

1.1.5 carry.models module

class `carry.models.`**`Booking`** (***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Model`

Booking Database Model
Each booking in the database will be assigned a unique id value, which is a primary key.

Note Reference should be unique. Booking Model has user_id and car_id fields which are foreign keys.

`booking_datetime`

`calendar_eid`

`cancelled`

`car_id`

`duration`

`end_datetime`

`finished`

`id`

`logs`

`reference`

`rendBy`

`rented`

`start_datetime`

`started`

`total_cost`

`user_id`

class `carry.models.`**`Car`** (***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Model`
Car Database Model
Each car in the database will be assigned a unique id value, which is a primary key.

`address`

`body_type`

`bookings`

`colour`

`cost`

`id`

`img_url`

lat

lng

logs

make

postcode

seats

state

suburb

class `carry.models.`**`CarSchema`** (**args*, ***kwargs*)
Bases: `marshmallow_sqlalchemy.schema.sqlalchemy_schema.SQLAlchemyAutoSchema`

class **`Meta`**
Bases: `object`

`model`
alias of `Car`

`opts` = **<marshmallow_sqlalchemy.schema.sqlalchemy_schema.SQLAlchemyAutoSchemaOpts object>**

class `carry.models.`**`Log`** (***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Model`
Log Database Model
Each Log in the database will be assigned a unique id value, which is a primary key.

Note Log Model has user_id, car_id and booking_id fields which are foreign keys.

booking_id

car_id

datetime

has

id

lat

lng

logged

logs

status

user_id

class carry.models.**User** (***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Model, flask_login.mixins.UserMixin

User Database Model

Each user in the database will be assigned a unique id value, which is a primary key.

Note Username and Email should be unique.

auth

bookings

email

firstname

id

lastname

location

logs

password

profile_url

username

carry.models.**load_user** (*user_id*)

Decorator for reloading the user from the user_id in the session :param user_id: Primary key of User :type user_id: int, essential :return User Object where id = user_id :rtype: dictionary

1.1.6 carry.routes module

carry.routes.**booking_detail** ( )

New Booking Detail Page.

carry.routes.**getDetailedMap** (*car*)

A Method for creating a detailed Google map

carry.routes.**getMap** ( )

A Method for creating a Google map

carry.routes.**home** ( )

Find Car Page.

carry.routes.**index** ( )

Landing Page.

carry.routes.**login** ( )

User Login Page.

`carry.routes.`**`logout`** ( )
User Logout Page.

`carry.routes.`**`my_bookings`** ( )
My Bookings Page.

`carry.routes.`**`new_booking`** ( )
New Booking Page.

`carry.routes.`**`profile`** ( )
User Profile Page.

`carry.routes.`**`register`** ( )
User Registration Page.

`carry.routes.`**`save_picture`** (*picture*)
A Method for saving a picture from user input

1.1.7 carry.socket module

`carry.socket.`**`background_thread`** ( )
Background Thread which runs to receive messages from Agent Pis

`carry.socket.`**`decrypt`** (*key*, *ciphertext*)
A function for decryption with shared key between Agent Pis and Master Pi

`carry.socket.`**`sendBookingID`** (*UDPServerSocket*, *address*, *request*, *user_id*, *car_id*)
A function for sending a booking id if there is current booking for specific user and car

`carry.socket.`**`sendJson`** (*UDPServerSocket*, *address*, *msgFromServer*)
A function for sending json data to Agent pi

`carry.socket.`**`statusUpdate`** (*UDPServerSocket*, *address*, *request*, *user_id*, *car_id*, *booking_id*, *lat*, *lng*, *datetime*)
A function for storing user activities in the database

`carry.socket.`**`user_credential`** (*UDPServerSocket*, *address*, *username*, *password*, *car_id*)
A function for validating user inputs from Agent Pis

1.1.8 carry.test_database_utils module

class `carry.test_database_utils.`**`TestDatabaseUtils`** (*methodName='runTest'*)
Bases: `unittest.case.TestCase`
This testcase is created by subclassing unittest.TestCase

Note To run this test class, enter this command in the project directory.

$ python3 -m unittest carry/test.py

classmethod **`setUpClass`** ( )
A class method called before tests in an individual class are run.
setUpClass is called with the class as the only argument and must be decorated as a classmethod()

classmethod `tearDownClass` ( )
A class method called after tests in an individual class have run.
tearDownClass is called with the class as the only argument and must be decorated as a classmethod():

`test_a` ( )
A test method if database configuration is set to the test database

`test_b_registerUser` ( )
A test method for user registration

`test_d_findUser` ( )
A test method for finding user

`test_e_getUser` ( )
A test method for getting user

`test_f_dupliUser` ( )
A test method for validating duplicate user

`test_g_createCar` ( )
A test method for creating a new car

`test_h_getCar` ( )
A test method for getting car

`test_i_searchCar` ( )
A test method for searching car details

`test_j_newBooking` ( )
A test method for creating a new booking

`test_k_addCalendarBooking` ( )
A test method for adding calendar event id to the existing booking

`test_l_getMyBookings` ( )
A test method for getting all bookings by the user

`test_m_getBookedCar` ( )
A test method for getting booked car during the period

`test_n_cancelBooking` ( )
A test method for cancelling the booking

`test_o_getMyPastBookings` ( )
A test method for getting past bookings

`test_p_getCurrentBooking` ( )
A test method for getting current booking allocated to specific car and user

`test_q_logging` ( )
A test method for getting current booking allocated to specific car and user

1.1.9 carry.test_routes module

class `carry.test_routes.`**`TestRoutes`** (*methodName='runTest'*)

Bases: `unittest.case.TestCase`

This class provides some basic methods for testing.

Note To run this test class, enter this command in the project directory.

$ python3 -m unittest carry/test.py

classmethod **`setUp`** ( )

The testing framework will automatically call for every single test.

`test_booking` ( )

a method which checks if booking page works properly.

Returns:

Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

`test_booking_detail` ( )

a method which checks if booking detail page works properly.

Returns:

Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

`test_home` ( )

a method which checks if find car page works properly.

Returns:

Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

`test_index` ( )

a method which checks if index page works properly.

Returns:

Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

`test_login` ( )

a method which checks if logout works properly.

Returns:
Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

test_logout ( )
a method which checks if logout works properly.

Returns:
Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

test_my_bookings ( )
a method which checks if mybookings page works properly.

Returns:
Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

test_profile ( )
a method which checks if profile page works properly.

Returns:
Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

test_register ( )
a method which checks if registration page works properly.

Returns:
Boolean

```
>>> follow_redirects=True
return True if it redirects successfully
return False if not.
```

1.1.10 Module contents

1.2 run module

1.3 sphinx module

1.3.1 Sphinx

The Sphinx documentation toolchain.

copyright Copyright 2007-2020 by the Sphinx team, see AUTHORS.

license BSD, see LICENSE for details.

`sphinx.`**`version_info`** **= (3, 0, 3, 'final', 0)**

Version info for better programmatic use.

A tuple of five elements; for Sphinx version 1.2.1 beta 3 this would be `(1, 2, 1, 'beta', 3)`. The fourth element can be one of: `alpha`, `beta`, `rc`, `final`. `final` always has 0 as the last element.

New in version 1.2: Before version 1.2, check the string `sphinx.__version__`.

- *Index*
- *Module Index*
- *Search Page*

Python Module Index

Index