2021

Bitrem Tracer

TECHNICAL DOCUMENTATION

SERGIO MATHURIN – BITREM DEVELOPMENT TEAM

1. File Structure

The application's file structure is as follows:

```
contact_tracing_desktop_application/
    Bitrem Tracer.spec
    config.yaml
    files_rc.py
    main.py
    requirements.txt
    tasks.py
    installer/
        Bitrem Tracer.exe
        build.ifp
        IBScanDriver Setup.msi
    lib/
        ibscan-1.0.0-py3-none-any.whl
    pages/
        AboutPage.py
        HomePage.py
        ScanPage.py
        __init__.py
    ui/
        __init__.py
        assets/
            files.qrc
            fonts/
                segoeui.ttf
                segoeuib.ttf
            images/
                fingerprint_black.png
                fingerprint_green.png
                finger_img.png
                icon.ico
                icon.png
                noun_Fingerprint_120965.png
        pyui/
            about_page.py
            animations.py
            error_window.py
            home_page.py
            log_page.py
            main_window.py
            scan_page.py
            splash_screen.py
            ui_styles.py
            widget_page.py
            __init__.py
        raw/
            about_page.ui
            error_window.ui
            home_page.ui
            log_page.ui
            main_window.ui
            scan_page.ui
            splash_screen.ui
            widget_page.ui
    users/
        UsersNew.py
        __init__.py
    utils/
        config.py
        contants.py
        exceptions.py
        image.py
        server.py
        utils.py
        __init__.py
    windows/
        ErrorWindow.py
        MainApplicationTrayIcon.py
        MainApplicationWindow.py
        MainApplicationWindowFunctions.py
        SplashScreenWindow.py
        __init__.py
    workers/
        fetch_device_id.py
        fetch_users.py
        identity_verification.py
        scanning.py
        startup.py
        __init__.py
```

Root Level Application Files

Name	Type	Description
Bitrem Tracer.spec	.spec file	PyInstaller configuration file
config.yaml	.yaml file	Default config file
files_rc.py	.py file	Python file mapping icons to python code.
requirements.txt	.txt file	List of all required python packages
tasks.py	.py file	Script file for automatically running application build

installer/	directory	Holds both some predefined build settings for using install forge as well as the final exe file from doing so and a setup file for installing the required drivers for Integrated Biometrics scanners.
lib/	directory	Holds any required libraries for this project.
pages/	Python module	Contains all defined application pages that can be used in the main window
ui/	Python module	Holds all user interface material from assets, to implemented GUI windows and pages built with Python.
users/	Python module	Code for dealing with user data
utils/	Python module	Utility functions
windows/	Python module	Contains all defined application windows
workers/	Python module	Thread workers

2. Build

2.1. Build Tools

The following tools are required when building the application via a computer using the Windows OS.

Tool	Version	Download Location
Python	3.0+	https://www.python.org/downloads/
Virtualenv **(Optional but recommended)**	20.2.2+	https://docs.python.org/3/tutorial/venv.html
Multiple Python Packages	Package version info in requirements.txt	The included requirements.txt
PyIBScan	1.0.0	Wheel (.whl) file found in project's lib folder. Present in requirements.txt but can be installed separately as well.
InstallForge	1.4.2	https://installforge.net/download/

2.2. Build Process

The build process for this project is done via the PyInstaller package. It can then be distributed as a folder, or using an installable application using InstallForge.

2.3. How to Run a Build

- *Install Build Tools*

  Ensure the appropriate build tools listed above are installed. These can each be found and installed using the instructions at the download locations provided.

  Note: As said above, a virtual environment is not necessary, but it is highly recommended that one be set up prior to installing all other Python dependencies and be used as the project's Python interpreter.

- *Run Build*

  So long as all the required tools are successfully installed the build process is simple.

  1. ***Build Installable Application***

     Simply navigate to the root directory of the project and run the **pyinstaller** command with the desired options. See https://www.pyinstaller.org/ for more information on PyInstaller settings.

  2. ***Add PyIBScan to Installable Application***

     Copy the ibscan directory from either your project's virtual environment, or from the PyIBScan project directory. Either will work but an instance of the ibscan module must be present in the built application's root directory.

  3. ***Build Installer for Application***

     Use InstallForge (or any other installer builder) to wrap the bult application into an installable package. The **build.ifp** file in the installer folder can be used for this or as a base. See https://installforge.net/ for more details of using InstallForge.

     Note: Any values that utilise absolute paths will have to be adjusted when using the *build.ifp* script on a new machine for the first time. This includes all files for the installable package in the *files section*, as well as the installer and uninstaller icons found in the *build section*.

  Note: Steps 1 and 2 of the build process can be done automatically using pre-configured PyInstaller settings using the invoke command whilst in the project's root directory.

  Example:

  ```
  invoke build
  ```

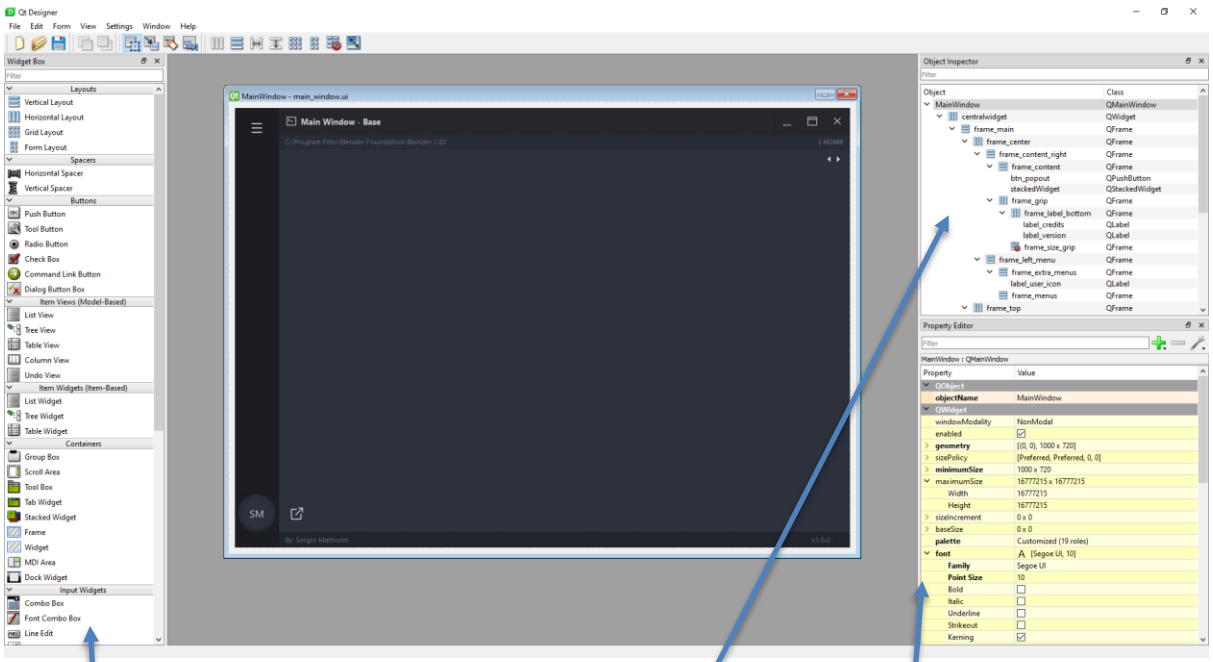
  For building the application with a debug console or

  ```
  invoke build-no-console
  ```

  For building the application without a debug console.

2.4. Using Qt Designer

Qt Designer is the primary tool used to for editing and creating this application's GUI. The basic layout of Qt Designer is:



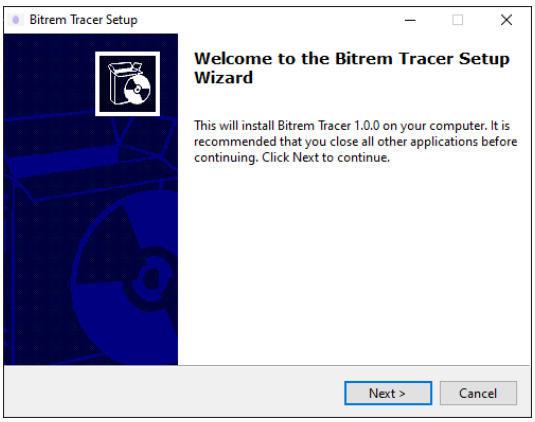
Qt Object Inspector
Used to view and select widget component

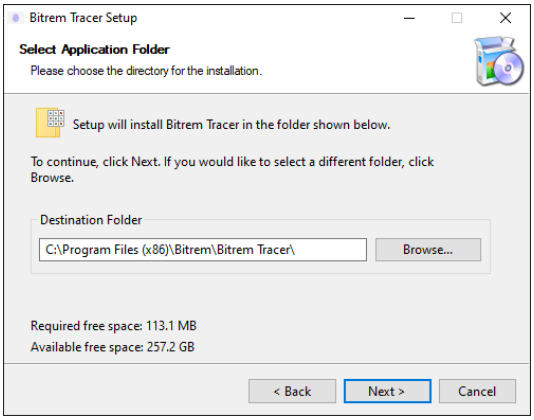
Qt Widget Box/Library
Used to add new widgets to interface

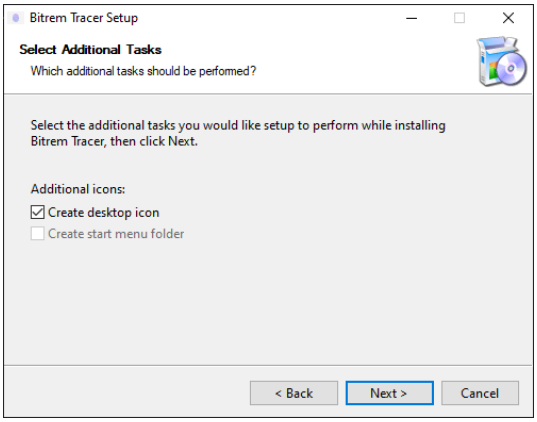
Object Properties
Used to modify widget properties

3. Installation

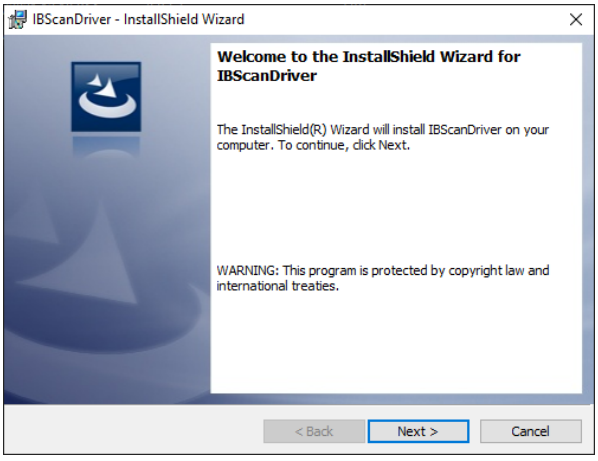
After building the application, it can be installed using the generated exe.

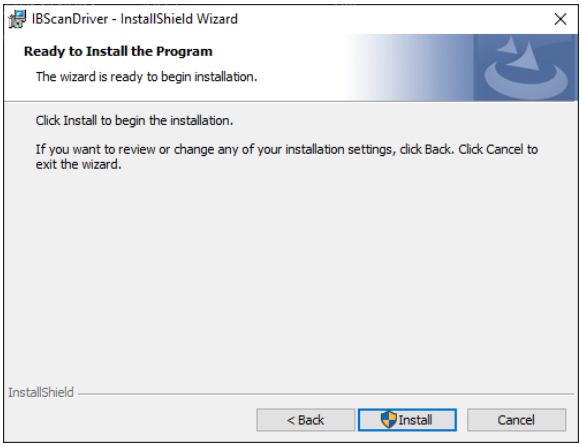






It is recommended that the Integrated Biometrics Drivers be installed as well, using the included MSI (.msi) file, installer/IBScanDriver Setup.msi.





4. Application Configuration

This application can have multiple settings manually configured by utilising the config.yaml file found in the application's install directory. Any changes done before building and installing the application will be reflected in the default config.yaml file available immediately after installation.
Settings that can be configured are as follows:

Name	Default Value	Description
DEVICE_ID	<Dataset Name>-registration	Unique ID given to each client application is installed on.
FINGERPRNT_EXT	Png	Format images are automatically saved as.
PRODUCTION	True	Whether production mode. Is on or off. [Deprecated – Previously used for connection to local database].
REGISTRATION_URL	https://identitysecure.bitrem.co/BitremWebApplication/ContactTracingScannerServlet	Url used for contacting server for multiple functions.
VERIFICATION_URL	https://identitysecure.bitrem.co/BitremWebApplication/AttendanceServlet	Url used for verifying user fingerprints using server.
VERSION	1.0.0	Current application version.

5. Application Structure

The application is built using the Qt Framework through the PySide2 package for Python. All components are made using Qt Widgets, each designed using QT Designer.

The application's main sections are built by extending the QT Window class and are built in a modular fashion to allow their internal widgets to be swapped in and out.

Though Windows and Pages shown are responsible for everything the user sees they, all underlying functionality is handled by the worker classes that extend the QRunnable class, allowing them to run on thread separate from the main application. This allows the main application to continuously update, providing a smooth user interface while, more complex underlying operations take place simultaneously.

5.1. Main Windows

Main GUI views. Users will always be presented with one of these when interacting with the application.

- ### UIFunctions

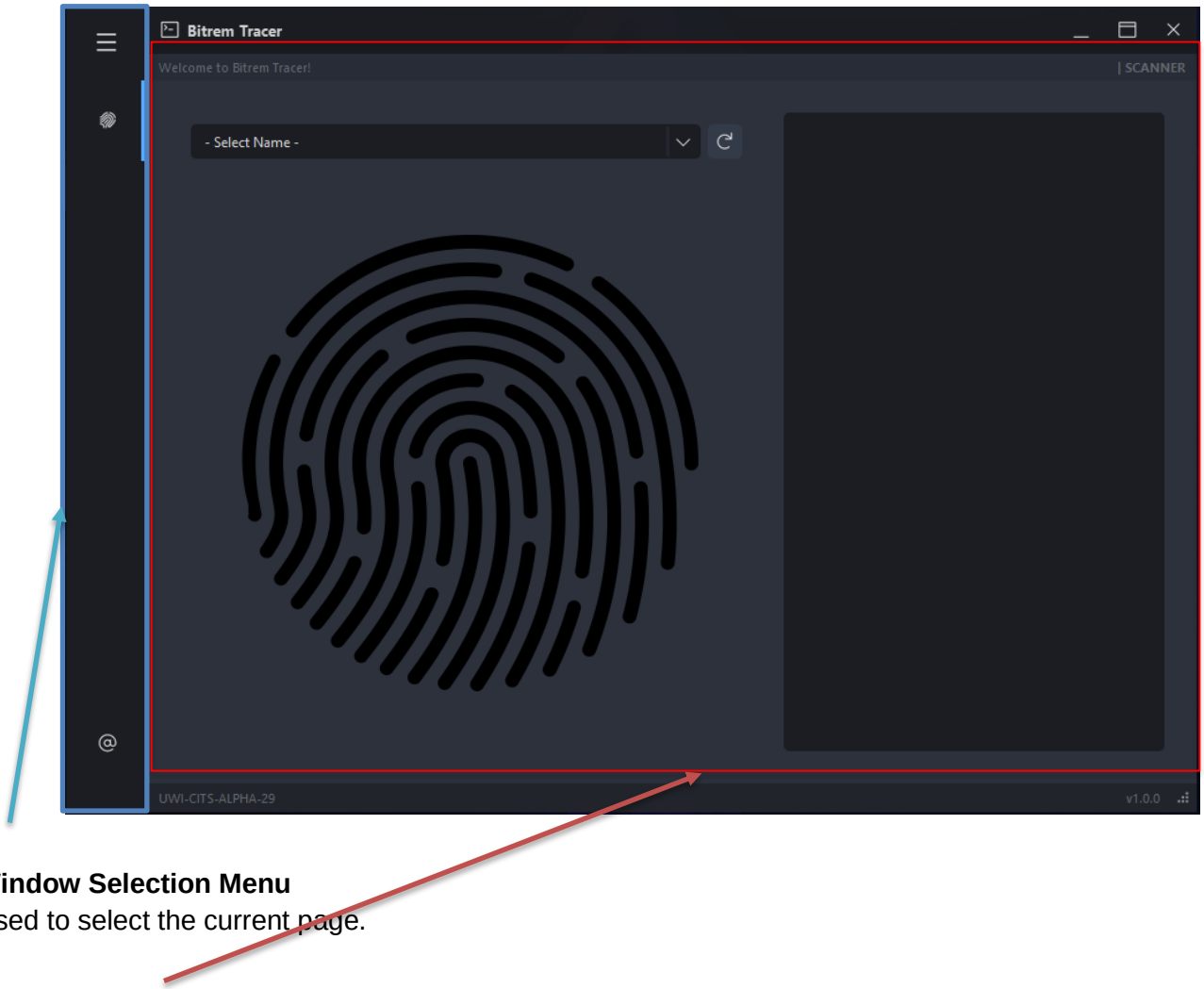
  All main windows apart from SplashScreen utilise the UIFunctions class located in windows/MainApplicationWindowFunction.py.

Main Window Function Methods

Name	Description
addNewMenu	Adds a new icon/button to side menu
deselectMenu	Deselect button/icon on side menu
enableMaximumSize	Enable maximum size
labelCredits	Set label credits text
labelDescription	Set label description text
labelPage	Sets page label text
labelTitle	Set label title text
maximize_restore	Maximize or restore window
removeTitleBar	Removes title bar
resetStyle	Reset style to default
returnStatus	Return current maximization state
selectMenu	Select button/icon on side menu
selectStandardMenu	Select standard menu from left menu
setStatus	Set current maximization state
toggleMenu	Show or hide menu using toggle button
uiDefinitions	Set up and connect all UI elements
userIcon	Set user icon. Can be set to initials, an icon, or hidden

- Primary Application Window

  The primary window is where a user will spend most of their time when interacting with this application, and has multiple functions depending on the current page widget selected.



Window Selection Menu
Used to select the current page.

Selected Page
As stated earlier, this application uses a modular design pattern to allow for its components to be swapped in and out at necessary. To this end the main application window consists of numerous pieces, with its primary component being a QStackedWdidget. This widget holds multiple predefined widgets and displays the desired widget based on the choice selected using the menu located on the far left.

Primary Window Components
All components can be viewed using Qt Designer's Object Inspector.

Primary Window Methods

Name	Description
Button	Triggered when side menu icon/button is pressed and switches current page to page associated with that

	button. Essentially allows you to swap between pages using the side menu
close	Closes off/terminates the application.
closeEvent	Triggers when close icon in top right is pressed. If the application's tray icon is available will hide the main window essentially minimizing it, otherwise it will close the entire application.
eventFilter	Filters events to determine if action should be taken when an event occurs
keyPressEvent	Triggered when key is pressed
mousePressEvent	Triggered when mouse button is pressed
resizeEvent	Triggered when user tries to resize the window
resizeFunction	Used when resizing window

Adding Pages to Primary Window:

1. Add an instance the page to be added to the window's stacked widget.
   Example:

```
self.page_about = AboutPage()
self.stackedWidget.addWidget(self.page_about)
```

2. Use the **addNewMenu** method to create a new entry in the side menu, specifying its name, button name, icon and whether it's laid out from the top of the screen or not.
   Example:

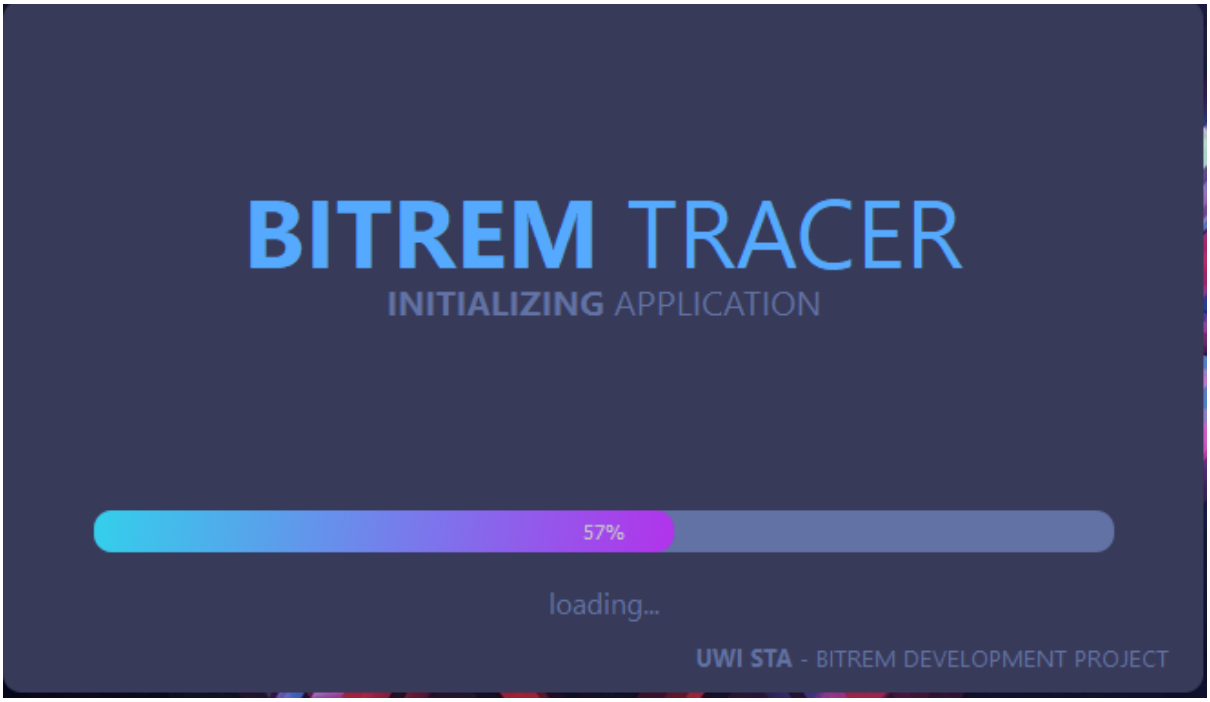
```
UIFunctions.addNewMenu(self, "About", "btn_about",
"url(:/16x16/icons/16x16/cil-at.png)", False)
```

3. Update the **Button** method's to automatically switch to the desired widget when a button name specified in step 2 is selected.
   Example:

```
# PAGE ABOUT
if btnWidget.objectName() == "btn_about":
    self.stackedWidget.setCurrentWidget(self.page_about)
    UIFunctions.resetStyle(self, "btn_about")
    UIFunctions.labelPage(self, "About")
    btnWidget.setStyleSheet(UIFunctions.selectMenu(btnWidget.styleSheet()))
```

- ## Splash Screen

  The splash screen serves as a pre-loader that pops up prior to application start up. Functions are run during this process, such as pre-loading the user list for example. If an error occurs during the pre-loading process, the Error Screen is presented, otherwise the Primary Application Window is presented.



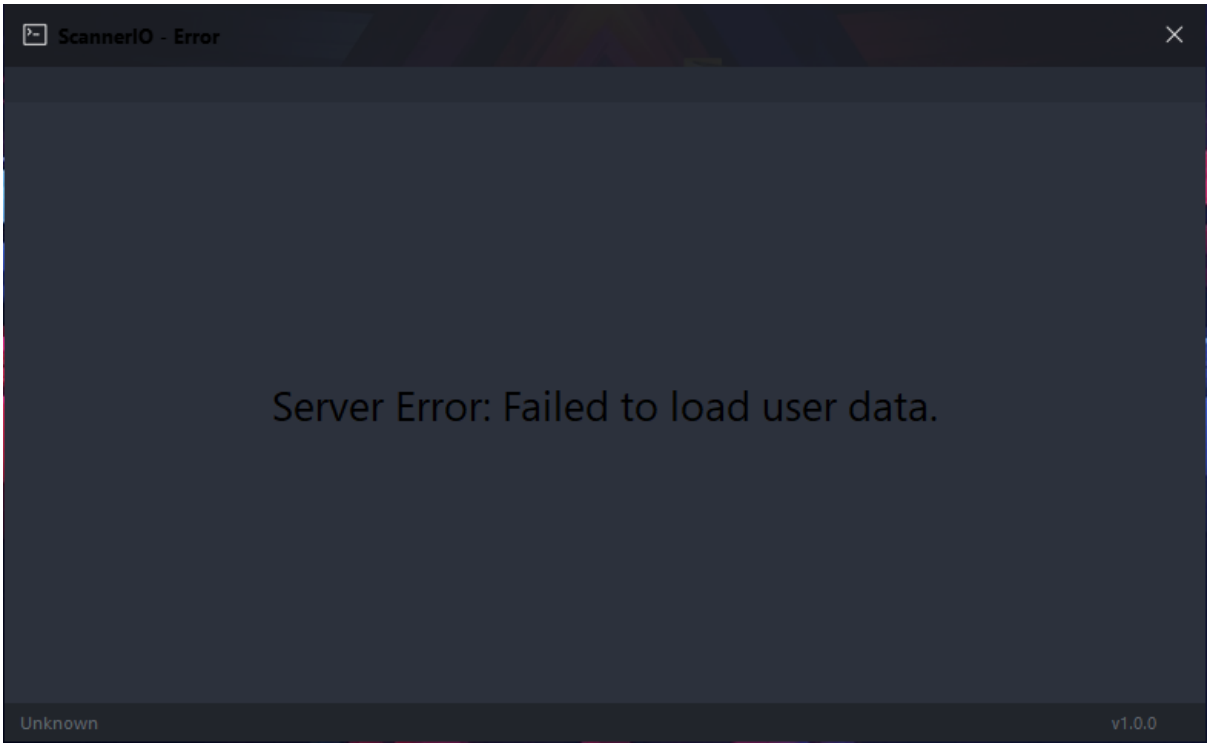
Splash Screen Components
All components can be viewed using Qt Designer's Object Inspector.

Splash Screen Methods

Name	Description
progress	Function used to increment progress bar and check and check if startup functions have completed successfully
on_start_startup	Called when startup worker thread begins startup process
on_stop_startup	Called when startup worker thread completes startup process
on_error_startup	Called when startup worker thread encounters an error during the startup process
on_progress_startup	Called when each time an underlying startup function is complete

- Error Screen

  The Error Screen is presented, and error occurred during the Splash Screen. It will show a simple message to summarize the error that occurred.



Error Screen Components
All components can be viewed using Qt Designer's Object Inspector.

5.2. Tray Icon

The main application also has a tray icon that extends the **MainApplicationTrayIcon** class and can be used when minimizing the application. The application must be closed using this tray icon.

Tray Icon Components
All components can be viewed using Qt Designer's Object Inspector.

Tray Icon Methods

Name	Description
close_application	Terminates entire application
on_activated	Opens main application window when icon is double clicked
open_application	Opens main application window
setupMenu	Sets up try icon menu on startup

5.3. Pages

These pages can be selected and viewed using on the Primary Application Window.

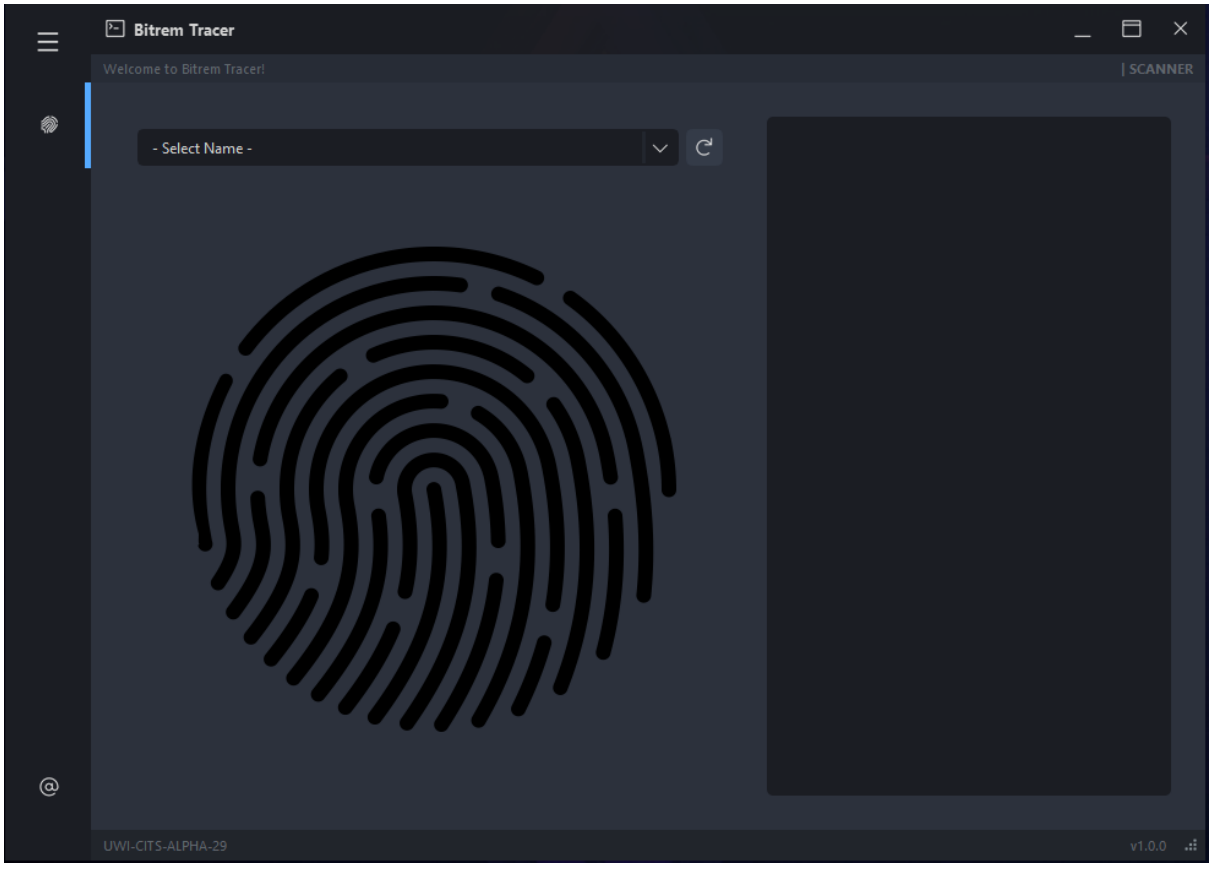
- Scan Page

  This page handles the scanning and contact tracing for this application.

Scanning Process:

1. A scan is triggered when a new name is selected from the dropdown menu.
2. The user is then asked to place their finger onto the scanner.
3. The scan is automatically completed when a good enough fingerprint image is obtained from the scanner.
4. This image is then sent off to the application server where it is used to validate the identity of the user. If the response returned from the server has a status code of 200, the application has successfully determined that the person is who they claim to be. If the response code is 403, they are not who they claim to be, and we assume an error took place otherwise.
5. An appropriate message is then displayed depending to the user, telling them if they were successfully validated or not.

All user scanning process is handled by an instance of the ScanWorker class, whilst the server communication is handled by an instance of the IdentityVerificationWorker class. Both of which work on separate threads than the main application.



Scan Page Components

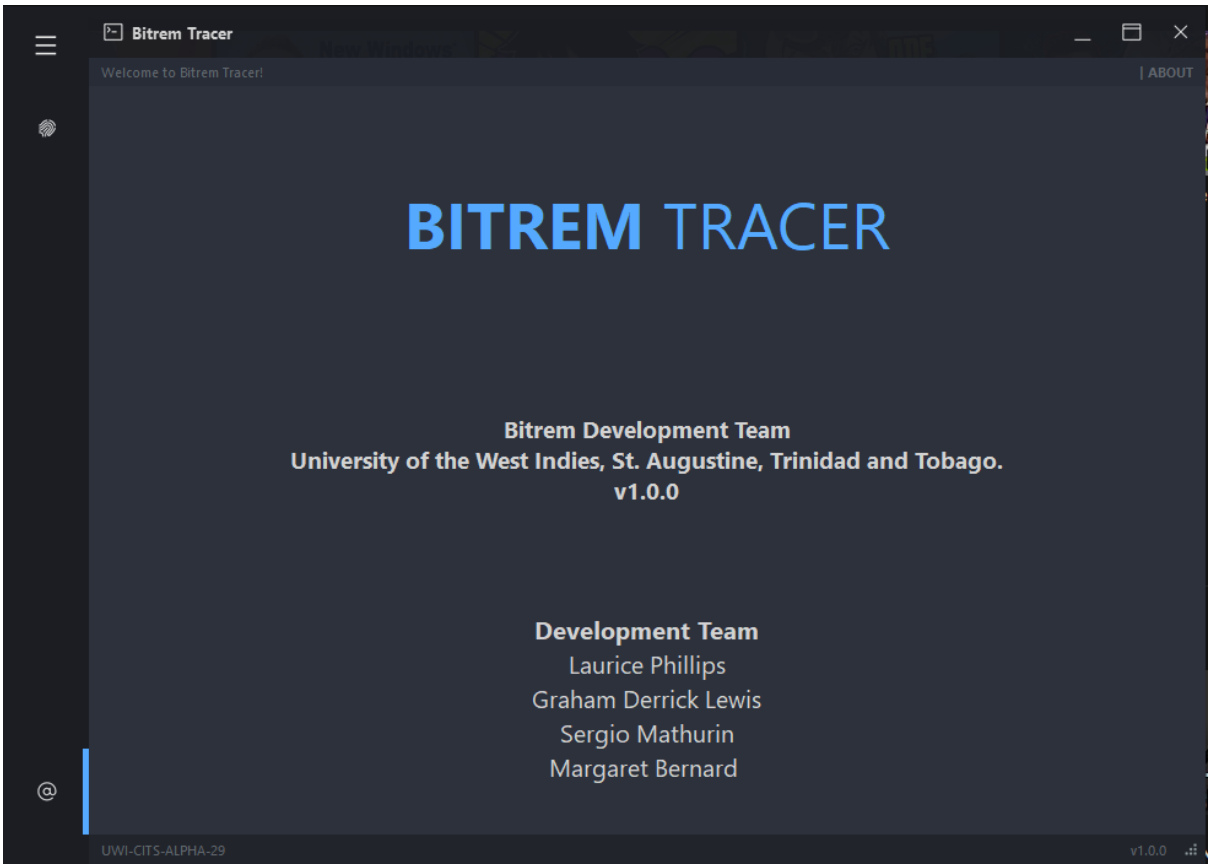
All components can be viewed using Qt Designer's Object Inspector.

Scan Page Methods

Name	Description
cancel_scan	Cancel the scanning process
error_fetch_users	Trigger a fetch users error in the fetch users thread
on_cancel_scan	Triggered when underlying scanner thread sends signal that scan has been cancelled
on_error_fetch_users	Triggered when the underlying fetch users thread sends the signal that an error occurred during the fetch users process
on_error_identity_verification	Triggered when underlying identity verification thread sends signal that an error occurred during the identity verification process.
on_error_scan	Triggered when underlying scanner thread sends the signal that an error has occurred during the scanning process
on_finger_touched	Triggered repeatedly by the underlying scanner thread informing the UI that a finger is present on the scanning platen or not
on_start_fetch_users	Triggered when the underlying fetch users thread sends the signal that the underlying fetch users process has started
on_start_identity_verification	Triggered when underlying identity verifcation thread sends signal that the identity verification process has started
on_start_scan	Triggered when underlying scanner thread sends signal that scan has started
on_stop_fetch_users	Triggered when the underlying fetch users thread sends the signal that the underlying fetch users process has stopped
on_stop_identity_verification	Triggered when underlying identity verification thread sends signal that identity verification process has stopped.
on_stop_scan	Triggered when underlying scanner thread sends the signal that scan has been stopped
reset	Resets the UI and scanning page to it's default state, ready to start a fresh scanning process.
start_fetch_users	Start the fetch users process.
start_identity_verification	Start identity verification process using currently selected user and most recent scan results
start_scan	Begin the scanning process
stop_fetch_users	Stop/Finish the fetch users process
stop_identity_verification	Stop/Finish identity verification process
stop_scan	Stop/Finish the scanning process
update	Updates the UI with any changes made recently.
update_user_list	Updates current list of available users

- About Page

  This page provides some basic information about the application and development team.



About Page Components

All components can be viewed using Qt Designer's Object Inspector.

5.4. Thread Workers

These are worker classes used to handle underlying functionality separate from the GUI of the application. These classes extend the QRunnable class and run-on separate threads from the main application. This allows them to seamlessly run concurrently with the user interface. The thread worker classes communicate with the main application via QSignals, that can be emitted at any time.

- ### ScanWorker

  Located in workers/scanning.py. This class handles all interaction with fingerprint scanners.

  ScanWorker Methods

Name	Description
cancel_scan	Cancels scan process
error_scan	Error while scanning
finger_touched	Status of whether a finger is on the finger platen during scan process.
run	Automatically runs when thread starts
scan_results	Converts byte string obtained from scan process into a usable image
start_scan	Starts scan process
stop_scan	Stops scan process

- ### IdentityVerifcationWorker

  Located in workers/identity_verification.py. This class handles all interaction with the application server regarding verification of a user's identity using an obtained fingerprint.

  IdentityVerificationWorker Methods

Name	Description
clear	Signals main application to reset to default in preparation for a new scan
error_verification	Error during identity verification process
run	Automatically runs when thread starts
start_verification	Starts identity verification process
stop_verification	Stops identity verification process

- StartupWorker

  Located in workers/startup.py. This class handles all processes carried out during the application's pre-loading/splash screen.

  StartupWorker Methods

Name	Description
error_startup	Error occurred during startup process
progress_startup	Sends a signal to main application informing it that a startup process has successfully been completed
Run	Automatically runs when thread starts
start_startup	Starts startup process
stop_startup	Stops startup process

- FetchUsersWorker

  Located in workers/fetch_users.py. This class handles all interaction with the application server regarding obtaining and refreshing the user data required by the application.

  FetchUsersWorker Methods

Name	Description
clear	Signals main application to reset to default in preparation for a new scan
error_fetch_users	Error occurred during fetch users process
run	Automatically runs when thread starts
start_fetch_users	Starts fetch users process
stop_fetch_users	Stops fetch users process
update_fetch_users	Sends signal to main application to update the user list with new data alongside the new data to be used.

6. Miscellaneous

6.1. Config

Located in utils/config.py. Deals directly with the config.yaml file.

Config Functions

Name	Description
load	Reads data from config.yaml file and places it into a usable dict
write	Writes dict data to config.yaml file

6.2. Constants

Located in utils/constants.py. Constant values used multiple times throughout application.

Defined Constants

Name	Description
ASSET_DIR	Asset directory.
BASE_DIR	Root directory.
FONTS_DIR	Fonts directory.
HTTP_HEADERS	Application header for all outgoing http requests.
ICONS_DIR	Icons directory.
IMAGES_DIR	Images directory.

6.3. Exceptions

Located in utils/exceptions.py. Constant values used multiple times throughout application.

Defined Exceptions

Name	Description
DeviceIdValidationError	Thrown when an error occurs when a valid device id is not found and cannot be obtained from the server.
FetchUsersError	Thrown when user data cannot be obtained from the server.
UserVerificationError	Thrown when a user cannot be verified using their registration id and a fingerprint scan. This does not mean the scan scan came back as a negative match, rather that the matching process was not successfully completed at all.

6.4. Image

Located in utils/image.py. Utility image processing functions.

Defined Functions

Name	Description
byte_arr_to_pil_img	Converts a byte array to a PIL Image.
decode_base64	Decodes a base64 string to a PIL Image.
encode_base64	Encodes a PIL Image as a base64 string.
pil_img_to_byte_arr	Converts a PIL Image to a byte array.

6.5. Server

Located in utils/server.py. Utility server management classes and functions. Currently this consists only of an Enum class used to define the functions that can be used on the REGISTRATION_URL.

ServerFunctions Enum

Name	String Value
REGISTRATION	registration
VERIFICATION	verification
USER_LIST	userlist

6.6. Utils

Located in utils/utils.py. Utility functions used throughout application.

Defined Functions

Name	Description

fetch_users	Uses application server to obtain a list of users.
	Server request uses the following format:
	
	Method: POST
	Uri: <REGISTRATION_URL>
	JSON Body: {
	'device_id': "<DEVICE_ID>",
	'function': "userlist"
	}

populate_users	Returns sorted list of User objects from data obtained in fetch_users.
register_device_id	Uses application server to obtain a new device id when presented with a default id following the pattern <dataset>-registration.

	Server request uses the following format:
	
	Method: POST
	Uri: <REGISTRATION_URL>
	JSON Body: {
	'device_id': "<DEVICE_ID>",
	'function': "registration"
	}

verify_user	Uses application server to verify a user's identity using their registration id and a fingerprint scan.
	Server request uses the following format:
	
	Method: POST
	Uri: <VERIFICATION_URL>
	JSON Body: {
	'img': "<base64 encoded fingerprint image>",
	'registration_id': "<selected user's registration_id>",
	'device_id': "<DEVICE_ID>",
	'function': "userlist"
	}

6.7. Users

Located in users/UsersNew.py. Class used for handling user data.

User Fields

Name	Description
dataset	User's dataset
fname	User's first name
lname	User's last name
oname	User's other name

6.8. UI

The application's user interface module contains multiple assets in the form of images, fonts and icons in their respective directories within the ui/assets/ folder.
It also contains the raw UI (.ui) files for the application's windows and pages, and the generated Python classes for those UI files in the ui/raw and ui/pyui directories, respectively.

Adjusting Old and Building New Interfaces:
Any of the application's UI elements can be modified using Qt designer, which comes downloaded with PySide2.

1. Find your installation of PySide2 in your Python interpreter's site-packages. A file called **designer.exe** will be in that directory and will open the Designer application when used.
2. Use Qt Designer to make open any older UI files and make changes or build new user interfaces.
3. Use P**ySide2's pyuic command** to generate/regenerate Python classes equivalent to the UI files.
   Example:

```
pyside2-uic ui/raw/main_window.ui > ui/pyui/main_window.py
```

2. Use or extend the newly generated Python classes.