GUI.py

```
import sys
import json
from PyQt5.QtWidgets import QApplication, QMainWindow, QCheckBox,
QListWidgetItem, QFileDialog, QDialog, QLabel, QVBoxLayout,
QPushButton, QMessageBox, QDesktopWidget
from PyQt5.QtCore import Qt, QTime, QTimer, QElapsedTimer, pyqtSlot,
QMetaObject
from PyQt5 import QtCore
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from PIL import ImageGrab
import pytesseract
import requests
from bs4 import BeautifulSoup
import re
import os

from PyQt5.QtCore import QThread, pyqtSignal

# Enable high DPI scaling for better display on high-resolution screens
os.environ["QT_ENABLE_HIGHDPI_SCALING"] = "1"
os.environ["QT_AUTO_SCREEN_SCALE_FACTOR"] = "1"
os.environ["QT_SCALE_FACTOR"] = "1"

# ModelTrainer class runs model training in the background on a
separate thread
class ModelTrainer(QThread):
    training_finished = pyqtSignal()  # Signal emitted when training is
complete

    def __init__(self, main_window):
        super().__init__()
        self.main_window = main_window  # Reference to the main window
that holds the model

    # This method is run when the thread starts
    def run(self):
        self.main_window.train_model()  # Call the main window's
training method
        print("start training")
        self.training_finished.emit()  # Emit the signal when training
is finished
```

```
# Import the UI files generated using PyQt Designer
from ui_Splash import Ui_SplashScreen
from ui_MainWindow2 import Ui_MainWindow
from ui_Settings2 import Ui_Settings

# Path to the settings file where user preferences are saved
SETTINGS_FILE = 'settings.json'

# Function to load settings from the settings file when the program
starts
def load_settings():
    try:
        # Open the settings file and load the content as a dictionary
        with open(SETTINGS_FILE, 'r') as f:
            settings = json.load(f)
            # Convert session times from strings to QTime objects for
easier manipulation
            for session in settings.get('sessions', []):
                if isinstance(session["start_time"], str):
                    session["start_time"] =
QTime.fromString(session["start_time"], "HH:mm")
                if isinstance(session["end_time"], str):
                    session["end_time"] =
QTime.fromString(session["end_time"], "HH:mm")
            return settings  # Return the loaded settings
    except FileNotFoundError:
        # If the settings file doesn't exist, return default settings
        return {
            "wait_time": "5",
            "sessions": [],
            "categories": {},
            "whitelisted_sites": [],
            "blacklisted_sites": [],
            "productive_sites": [],  # List of productive sites
            "override_delay": False,
            "warning_message": "Unproductive activity detected! For
your own good, please return to being productive!"
        }

# Save the current settings back to the settings.json file
def save_settings(settings):
    # Create a copy of the settings to save, converting QTime objects back to strings
    settings_copy = {
        "wait_time": settings["wait_time"],
        "sessions": [
            {
                "start_time": session["start_time"].toString("HH:mm"),
                "end_time": session["end_time"].toString("HH:mm"),
                "days": session["days"]
            }
            for session in settings["sessions"]
        ],
        "categories": settings["categories"],
        "whitelisted_sites": settings["whitelisted_sites"],
        "blacklisted_sites": settings["blacklisted_sites"],
        "productive_sites": settings.get("productive_sites", []),  # Save productive sites if available
        "override_delay": settings.get("override_delay", False),
        "warning_message": settings.get("warning_message", "Unproductive activity detected! For your own good, please return to being productive!")
    }
    # Write the settings copy to the settings.json file
    with open(SETTINGS_FILE, 'w') as f:
        json.dump(settings_copy, f, indent=4)

# Class for the SplashScreen which is displayed while the program is loading
class SplashScreen(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.ui = Ui_SplashScreen()  # Initialize the splash screen UI
        self.ui.setupUi(self)
        self.center()  # Center the splash screen on the screen

        # Make the window frameless and translucent for a modern look
        self.setWindowFlag(Qt.FramelessWindowHint)
        self.setAttribute(Qt.WA_TranslucentBackground)

        self.counter = 0  # Counter for the progress bar

        # Timer to update the progress bar every 50 milliseconds
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.progress)
        self.timer.start(50)  # Start the timer

        self.show()  # Show the splash screen

        # Prepare the main window but do not show it yet
        self.main_window = MainWindow()

        # Start training the machine learning model in a separate
thread
        self.model_trainer = ModelTrainer(self.main_window)

self.model_trainer.training_finished.connect(self.on_training_finished)
        self.model_trainer.start()  # Start the model training

    # Function to center the splash screen on the display
    def center(self):
        qr = self.frameGeometry()  # Get the geometry of the window
        cp = QDesktopWidget().availableGeometry().center()  # Get the
center of the screen
        qr.moveCenter(cp)  # Move the window to the center
        self.move(qr.topLeft())  # Move the window to the top-left
corner of the geometry

    # Update the progress bar and eventually close the splash screen
    def progress(self):
        self.ui.progressBar.setValue(self.counter)  # Set progress bar
value

        if self.counter > 100:  # If the counter exceeds 100, close the
splash screen
            self.timer.stop()  # Stop the timer
            self.main_window.showMaximized()  # Show the main window
maximized
            self.close()  # Close the splash screen

        self.counter += 1  # Increment the counter

    # Called when the model training is finished
    def on_training_finished(self):
        print("Model training completed.")
        self.main_window.on_training_finished()  # Notify the main
window that training is done
```

```
# Set up the path to the Tesseract OCR executable
tesseract_path = os.path.join(os.path.dirname(__file__),
'Tesseract-OCR', 'tesseract.exe')
pytesseract.pytesseract.tesseract_cmd = tesseract_path  # Tell
pytesseract where to find Tesseract

# Main application window class
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)  # Set up the main window UI

        self.currently_in_session = False  # Indicates whether the user
is in a session
        self.model_trained = False  # Indicates whether the machine
learning model is trained

        # Initialize the machine learning model for detecting
unproductive activity
        self.vectorizer = TfidfVectorizer()  # Convert text to numeric
features using TF-IDF
        self.model = LogisticRegression(max_iter=1000)  # Use logistic
regression with a high iteration limit

        # Load settings from the settings file
        self.settings = load_settings()
        self.wait_time = int(self.settings.get("wait_time", "5"))  #
Time before showing a warning
        self.sessions = self.settings.get("sessions", [])  # User's
work sessions
        self.categories = self.settings.get("categories", {})  #
Unproductive categories
        self.whitelisted_sites = self.settings.get("whitelisted_sites",
[])  # Whitelisted sites
        self.blacklisted_sites = self.settings.get("blacklisted_sites",
[])  # Blacklisted sites
        self.productive_sites = self.settings.get("productive_sites",
[])  # Manually added productive sites
        self.override_delay = self.settings.get("override_delay",
False)  # Whether warnings should be immediate
```

```
        self.warning_message = self.settings.get("warning_message",
"Unproductive activity detected! For your own good, please return to
being productive!")  # Warning message

        self.unproductive_flag = False  # Flag indicating if
unproductive activity is detected
        self.unproductive_timer = QElapsedTimer()  # Timer to track
unproductive activity duration
        self.warning_displayed = False  # Whether a warning message is
currently displayed

        # Connect buttons in the UI to their respective actions
        self.AddWhitelist.clicked.connect(self.add_to_whitelist)
        self.BrowseFile.clicked.connect(self.browse_file)

self.RemoveWhitelist.clicked.connect(self.remove_selected_from_whitelis
t)
        self.AddBlacklist.clicked.connect(self.add_to_blacklist)
        self.BrowseFile_2.clicked.connect(self.browse_file_blacklist)

self.RemoveBlacklist.clicked.connect(self.remove_selected_from_blacklis
t)
        self.Settings.clicked.connect(self.open_settings)

        # Unproductive category names with example site links
        self.category_sites = {
            "Social Media": ["facebook.com", "twitter.com",
"instagram.com", "discord.com"],
            "Games": ["store.epicgames.com", "store.steampowered.com",
                      "poki.com",
"minesweeper.online","nytimes.com/games/wordle",
"nytimes.com/crosswords/game/mini",

"playhop.com","coolmathgames.com","aol.com/games","solitaired.com"],
            "Video Entertainment": ["youtube.com",
"sky.com","en.tinyzone-tv.com","vimeo.com/watch","netflix.com/hk-en/bro
wse/genre/839338","ondisneyplus.disney.com","amazon.com/gp/video/storef
ront"],
            "Visual Entertainment": ["globalcomix.com", "mangadex.org",
"manganato.com", "anime-planet.com", "readallcomics.com"],
            "Forums":
["reddit.com","reddit.com/r/popular","reddit.com/r/all" "quora.com",
"threads.net/?hl=en","reddit.com/r/all/new"],
            "Online Shopping": ["amazon.com", "alibaba.com",
"aliexpress.com", "zalora.com", "eBay.com", "taobao.com",
"hktvmall.com","zalora.com.hk/s/women","hktvmall.com/hktv/en/","myntra.
com"],
        }

        # Populate the UI with data from settings
        self.populate_unproductive_categories()
        self.populate_whitelisted_sites()
        self.populate_blacklisted_sites()

        # Timer to periodically check if the user is within an active
session
        self.session_check_timer = QTimer()
        self.session_check_timer.timeout.connect(self.check_sessions)
        self.session_check_timer.start(10000)  # Check every 10 seconds

        # Timer to monitor the screen and detect unproductive activity
        self.activity_monitor_timer = QTimer()

self.activity_monitor_timer.timeout.connect(self.detect_unproductive_ac
tivity)
        self.activity_monitor_timer.start(15000)  # Check every 15
seconds

    # Center the main window on the screen
    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    # Callback when model training is finished
    def on_training_finished(self):
        self.model_trained = True
        print("Model training completed.")

    # Capture a screenshot of the entire screen
    def capture_screen(self):
        screen = ImageGrab.grab()
        return screen

    # Extract text from an image using Tesseract OCR
    def extract_text_from_image(self, image):
        try:
            text = pytesseract.image_to_string(image)
            return text
        except pytesseract.TesseractNotFoundError:
            print("Tesseract is not found in the path used")
            return ""
        except Exception as e:
            print(f"{e}")
            return ""

    # Detect unproductive activity by analyzing the text on the screen
    def detect_unproductive_activity(self):
        # Only proceed if the user is in an active session and the
model is trained
        if not self.currently_in_session:
            print("Current time is not within any session. Will not
proceed with activity detection.")
            return

        if not self.model_trained:
            print("Model is not yet trained. Will not proceed with
activity detection.")
            return

        print("Detecting unproductive activity...")
        screen_image = self.capture_screen()  # Capture the screen
        screen_text = self.extract_text_from_image(screen_image)  #
Extract text from the captured image

        if screen_text.strip():
            print("Extracted text from screen:", screen_text)
            self.unproductive_flag =
self.predict_unproductive(screen_text)  # Predict if the text is
unproductive
            print(f"Screen text is {'unproductive' if
self.unproductive_flag else 'productive'}.")

            # Handle detected unproductive activity
            if self.unproductive_flag:
                if self.override_delay:
                    print("Override delay is enabled, displaying
warning message immediately.")
```

```
                    self.display_warning_message()  # Show warning
immediately if override delay is enabled
                else:
                    if not self.unproductive_timer.isValid():
                        print("Starting unproductive timer.")
                        self.unproductive_timer.start()  # Start the
timer for unproductive activity
                    elif self.unproductive_timer.elapsed() >=
self.wait_time * 60000:
                        print(f"Unproductive timer elapsed
{self.wait_time} minutes, displaying warning message.")
                        self.display_warning_message()  # Show warning
if the wait time has passed
                        self.unproductive_timer.invalidate()  # Reset
the timer
                    else:
                        remaining_time = self.wait_time * 60000 -
self.unproductive_timer.elapsed()
                        print(f"Unproductive timer running. Time
remaining: {remaining_time // 60000} minutes and {remaining_time %
60000 // 1000} seconds.")
            else:
                print("Resetting unproductive flag and timer.")
                self.unproductive_flag = False
                self.unproductive_timer.invalidate()  # Reset the timer
if the activity is productive
        else:
            print("No text detected on screen.")

    # Show a modal dialog with a warning message
    @QtCore.pyqtSlot()
    def show_warning_message(self):
        dialog = QDialog(self)
        dialog.setWindowTitle("Warning")
        dialog.setWindowFlag(Qt.WindowType.WindowStaysOnTopHint)  #
Ensure the dialog stays on top
        dialog.showFullScreen()  # Show the dialog in full screen

        layout = QVBoxLayout(dialog)
        label = QLabel(self.warning_message, dialog)  # Display the
warning message
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        label.setStyleSheet("font-size: 24px; color: red;")
        layout.addWidget(label)

        close_button = QPushButton("Close", dialog)  # Add a close
button
        close_button.setStyleSheet("font-size: 18px; padding: 10px;")
        close_button.clicked.connect(dialog.accept)
        layout.addWidget(close_button)

        dialog.finished.connect(self.on_warning_closed)
        dialog.exec_()  # Make the dialog modal

    # Reset the state when the warning dialog is closed
    @QtCore.pyqtSlot()
    def on_warning_closed(self):
        self.warning_displayed = False
        self.unproductive_flag = False
        self.unproductive_timer.invalidate()
        print("Warning message closed. Resetting state.")

    # Populate the unproductive categories in the UI
    def populate_unproductive_categories(self):
        for category, examples in self.category_sites.items():
            checkbox = QCheckBox(category)
            checkbox.setStyleSheet("""
                QCheckBox {
                    font-family: 'Segoe UI';
                    font-size: 13pt;
                    color: #DC5F00;
                    background: transparent;
                }
            """)
            # Tooltips for each category
            if category == "Social Media":
                tooltip_text = "Examples: facebook, twitter, instagram,
github"
            elif category == "Games":
                tooltip_text = "Examples: crazygames, epicgames, steam,
wordle, spelling bee, crosswords.."
            elif category == "Video Entertainment":
                tooltip_text = "Examples: youtube, netflix, prime
video, disney plus"
            elif category == "Visual Entertainment":
                tooltip_text = "Examples: comic and manga sites"
            elif category == "Forums":
                tooltip_text = "Examples: reddit, quora"
            elif category == "Online Shopping":
                tooltip_text = "Examples: amazon, taobao, aliexpress.."

            checkbox.setToolTip(tooltip_text)  # Set the tooltip for
the checkbox
            self.Categories.addWidget(checkbox)  # Add the checkbox to
the UI
            checkbox.stateChanged.connect(lambda state, name=category:
self.update_category_state(name, state))

            # Check the checkbox if the category is in the saved
settings
            if category in self.categories:
                checkbox.setChecked(self.categories[category])

    # Update the state of a category when its checkbox is checked or
unchecked
    def update_category_state(self, category_name, state):
        self.categories[category_name] = bool(state)  # Update the
category's state
        self.settings["categories"] = self.categories  # Update
settings
        save_settings(self.settings)  # Save the new settings
        if self.model_trained:
            self.start_training()  # Retrain the model when chosen
unproductive categories change

    # Populate the whitelisted sites in the UI
    def populate_whitelisted_sites(self):
        for site in self.whitelisted_sites:
            item = QListWidgetItem(site)
            self.Whitelist.addItem(item)

    # Add a website to the whitelist
    def add_to_whitelist(self):
        url = self.EditWhitelist.text()  # Get the URL entered by the
user
        if url:
            item = QListWidgetItem(url)
            self.Whitelist.addItem(item)  # Add the URL to the list
widget
```

```
            self.whitelisted_sites.append(url)  # Add the URL to the
whitelisted sites
            self.settings["whitelisted_sites"] = self.whitelisted_sites
# Update settings
            save_settings(self.settings)  # Save the new settings
            self.EditWhitelist.clear()  # Clear the input field
        else:
            QMessageBox.warning(self, "Input Error", "Please enter a
website to whitelist.")  # Show a warning if no URL is entered

    # Browse the file system to select a file to whitelist
    def browse_file(self):
        file_path, _ = QFileDialog.getOpenFileName(self, "Select File
to Whitelist")
        if file_path:
            self.EditWhitelist.setText(file_path)  # Set the selected
file path in the text field

    # Remove the selected website(s) from the whitelist
    def remove_selected_from_whitelist(self):
        selected_items = self.Whitelist.selectedItems()  # Get the
selected items from the list
        if not selected_items:
            QMessageBox.warning(self, "Selection Error", "Please select
a link to remove.")  # Show a warning if no item is selected
            return
        for item in selected_items:
            self.whitelisted_sites.remove(item.text())  # Remove the
site from the whitelist
            self.settings["whitelisted_sites"] = self.whitelisted_sites
# Update settings
            save_settings(self.settings)  # Save the new settings
            self.Whitelist.takeItem(self.Whitelist.row(item))  # Remove
the item from the list widget

    # Populate the blacklisted sites in the UI
    def populate_blacklisted_sites(self):
        for site in self.blacklisted_sites:
            item = QListWidgetItem(site)
            self.Blacklist.addItem(item)

    # Add a website to the blacklist
    def add_to_blacklist(self):
        url = self.EditBlacklist.text()  # Get the URL entered by the
user
        if url:
            item = QListWidgetItem(url)
            self.Blacklist.addItem(item)  # Add the URL to the list
widget
            self.blacklisted_sites.append(url)  # Add the URL to the
blacklisted sites
            self.settings["blacklisted_sites"] = self.blacklisted_sites
# Update settings
            save_settings(self.settings)  # Save the new settings
            self.EditBlacklist.clear()  # Clear the input field
        else:
            QMessageBox.warning(self, "Input Error", "Please enter a
website to blacklist.")  # Show a warning if no URL is entered

    # Browse the file system to select a file to blacklist
    def browse_file_blacklist(self):
        file_path, _ = QFileDialog.getOpenFileName(self, "Select File
to Blacklist")
        if file_path:
            self.EditBlacklist.setText(file_path)  # Set the selected
file path in the text field

    # Remove the selected website(s) from the blacklist
    def remove_selected_from_blacklist(self):
        selected_items = self.Blacklist.selectedItems()  # Get the
selected items from the list
        if not selected_items:
            QMessageBox.warning(self, "Selection Error", "Please select
a link to remove.")  # Show a warning if no item is selected
            return
        for item in selected_items:
            self.blacklisted_sites.remove(item.text())  # Remove the
site from the blacklist
            self.settings["blacklisted_sites"] = self.blacklisted_sites
# Update settings
            save_settings(self.settings)  # Save the new settings
            self.Blacklist.takeItem(self.Blacklist.row(item))  # Remove
the item from the list widget

    # Open the settings page window
    def open_settings(self):
        self.settings_page = SettingsPage(self)  # Pass the main window
as the parent
        self.settings_page.showMaximized()  # Show the settings page
maximized
        self.hide()  # Hide the main window

    # Set the wait time before showing a warning message
    def set_wait_time(self, wait_time):
        self.wait_time = int(wait_time)
        self.settings["wait_time"] = wait_time  # Update settings
        save_settings(self.settings)  # Save the new settings

    # Get the current wait time
    def get_wait_time(self):
        return self.wait_time

    # Check if the current time falls within any of the user's work
sessions
    def check_sessions(self):
        current_time = QTime.currentTime()  # Get the current time
        current_day = QtCore.QDate.currentDate().dayOfWeek() - 1  # Get
the current day (0=Monday, 6=Sunday)

        in_session = False  # Flag to track if the current time is
within a session
        for session in self.sessions:
            start_time = session["start_time"]
            end_time = session["end_time"]

            # Check if the current day is part of the session's active
days
            if session["days"][current_day]:
                if end_time < start_time:  # If the session spans
midnight
                    if current_time >= start_time or current_time <=
end_time:
                        in_session = True
                        print(f"Current time {current_time.toString()}
is within session: {start_time.toString()} - {end_time.toString()}
(spans midnight)")
                        break
                else:
```

```python
                    if start_time <= current_time <= end_time:  # If the session is during the same day
                        in_session = True
                        print(f"Current time {current_time.toString()} is within session: {start_time.toString()} - {end_time.toString()}")
                        break

        # Update the session status accordingly
        if in_session and not self.currently_in_session:
            print(f"Current time {current_time.toString()} is within a session.")
            self.currently_in_session = True  # Set to True if now in a session
        elif not in_session and self.currently_in_session:
            print(f"Current time {current_time.toString()} is not within any session.")
            self.currently_in_session = False  # Set to False if no longer in a session

    # Start monitoring the user’s screen activity
    def monitor_activity(self):
        self.activity_monitor_timer.start()  # Start the activity monitoring timer

    # Predict whether the given text is unproductive using the trained model
    def predict_unproductive(self, text):
        if not self.currently_in_session or not self.model_trained:
            return False  # If not in session or model isn't trained, return False

        content_features = self.vectorizer.transform([text])  # Transform the text into numeric features
        prediction = self.model.predict(content_features)  # Use the model to make a prediction
        return prediction[0] == "unproductive"  # Return True if the prediction is "unproductive"

    # Start training the machine learning model
    def start_training(self):
        self.model_trainer = ModelTrainer(self)  # Create a new model trainer thread
        if self.model_trainer.isRunning():
            self.model_trainer.terminate()  # If already running,
terminate the old thread

self.model_trainer.training_finished.connect(self.on_training_finished)
# Connect the signal
        self.model_trainer.start()  # Start training
        print("retraining")

    # Train the machine learning model based on user preferences and
example sites
    def train_model(self):
        data = []
        labels = []

        # Use example sites from the selected unproductive categories
to train the model
        for category, examples in self.category_sites.items():
            if self.categories.get(category):
                for site in examples:
                    content = self.fetch_website_content("http://" +
site)
                    if content:
                        data.append(content)
                        labels.append("unproductive")
            else:
                for site in examples:
                    content = self.fetch_website_content("http://" +
site)
                    if content:
                        data.append(content)
                        labels.append("productive")

        # Add whitelisted and blacklisted sites to the training data
        for site in self.whitelisted_sites:
            content = self.fetch_website_content("http://" + site)
            if content:
                data.append(content)
                labels.append("productive")

        for site in self.blacklisted_sites:
            content = self.fetch_website_content("http://" + site)
            if content:
                data.append(content)
                labels.append("unproductive")

        # Add manually specified productive sites to the training data
        for site in self.productive_sites:
            content = self.fetch_website_content("http://" + site)
            if content:
                data.append(content)
                labels.append("productive")

        # Train the model using the collected data
        print("training with data")
        features = self.vectorizer.fit_transform(data)  # Transform the
data into features
        self.model.fit(features, labels)  # Train the logistic
regression model


    # Fetch the content of a website for training or prediction
    def fetch_website_content(self, url):
        if not url.startswith("http://") and not
url.startswith("https://"):
            url = "http://" + url  # Ensure the URL starts with
"http://"
        try:
            response = requests.get(url)  # Send a request to the
website
            soup = BeautifulSoup(response.text, 'html.parser')  # Parse
the HTML content
            print("fetching content")
            return ' '.join(re.findall(r'\w+',
soup.get_text().lower()))  # Return the cleaned text
        except Exception as e:
            print(f"Error fetching website content: {e}")
            return None  # Return None if there's an error

    # Display a warning message to the user
    def display_warning_message(self):
        self.warning_displayed = True  # Set the flag to indicate a
warning is being displayed
        QMetaObject.invokeMethod(self, "show_warning_message",
Qt.QueuedConnection)  # Show the warning dialog

    # Send a warning message if it hasn't already been displayed
    def send_warning_message(self):
        if not self.warning_displayed:
            self.display_warning_message()  # Display the warning
message if it hasn’t been shown yet

# Settings page window class
class SettingsPage(QMainWindow, Ui_Settings):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.ui = Ui_Settings()  # Initialize the settings page UI
        self.ui.setupUi(self)
        self.sessions = self.parent().sessions  # Get the sessions from
the main window

        # Connect settings page buttons to their respective actions
        self.ui.AddTiming.clicked.connect(self.add_session)
        self.ui.RemoveTiming.clicked.connect(self.remove_session)
        self.ui.Save.clicked.connect(self.save_settings)
        self.ui.BackHome.clicked.connect(self.go_back)  # Back button
to return to the main window

        # Link override delay and warning message to settings
        self.OverrideDelay = self.ui.OverrideDelay
        self.WarningMessage = self.ui.Warning

        self.OverrideDelay.setChecked(self.parent().override_delay)  #
Load the current override delay setting
        self.WarningMessage.setPlainText(self.parent().warning_message)
# Load the current warning message


self.OverrideDelay.stateChanged.connect(self.set_override_delay)  #
Update override delay when changed

        self.ui.Delay.addItems(["5", "10", "15", "30"])  # Add options
for delay time in the dropdown
        self.reload_settings_page()  # Load the settings into the page
        self.center()  # Center the settings window

    # Center the settings window on the screen
    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    # Add a new session to the list of work sessions
    def add_session(self):
        start_time = self.ui.Start.time()  # Get the start time from
the UI
        end_time = self.ui.End.time()  # Get the end time from the UI
        days = [self.ui.monday.isChecked(),
self.ui.tuesday.isChecked(), self.ui.wednesday.isChecked(),
                self.ui.thursday.isChecked(),
self.ui.friday.isChecked(), self.ui.saturday.isChecked(),
                self.ui.sunday.isChecked()]  # Get selected days

        session = {
            "start_time": start_time,
            "end_time": end_time,
            "days": days
        }

        self.sessions.append(session)  # Add the new session to the
list
        self.update_sessions_list()  # Update the UI to show the new
session

    # Update the list of existing sessions in the UI
    def update_sessions_list(self):
        self.ui.ExistingRunTimes.clear()  # Clear the list before
repopulating
        for session in self.sessions:
            days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
            active_days = [days[i] for i, active in
enumerate(session["days"]) if active]  # Get active days
            item_text = f"{session['start_time'].toString()} -
{session['end_time'].toString()} : {', '.join(active_days)}"
            self.ui.ExistingRunTimes.addItem(item_text)  # Add the
session to the list

    # Remove the selected session(s) from the list of work sessions
    def remove_session(self):
        selected_items = self.ui.ExistingRunTimes.selectedItems()  #
Get selected items from the UI
        if not selected_items:
            return
        for item in selected_items:
            index = self.ui.ExistingRunTimes.row(item)  # Get the index
of the selected item
            del self.sessions[index]  # Remove the session from the
list
            self.ui.ExistingRunTimes.takeItem(index)  # Remove the item
from the UI list

    # Save the current settings back to the settings file
    def save_settings(self):
        wait_time = self.ui.Delay.currentText()  # Get the selected
wait time
        self.parent().set_wait_time(wait_time)  # Update the main
window's wait time
        self.parent().sessions = self.sessions  # Update the main
window's sessions
        self.parent().settings["sessions"] = self.sessions  # Update
settings
        self.parent().settings["override_delay"] =
self.OverrideDelay.isChecked()  # Update override delay
        self.parent().settings["warning_message"] =
self.WarningMessage.toPlainText()  # Update warning message
        save_settings(self.parent().settings)  # Save the settings to
the file
        print("Settings saved")
        self.reload_settings_page()  # Reload the settings page

    # Set the override delay when the checkbox is changed
    def set_override_delay(self, state):
        self.parent().override_delay = bool(state)  # Update the main
window’s override delay
        self.parent().settings["override_delay"] =
self.parent().override_delay  # Update settings
        save_settings(self.parent().settings)  # Save the new settings

    # Reload the settings page with the most recent settings
    def reload_settings_page(self):
        self.parent().settings = load_settings()  # Reload settings
from the file
        self.sessions = self.parent().settings.get("sessions", [])  #
Get the sessions from settings
```

```
        self.update_sessions_list()  # Update the UI with the reloaded
sessions

        current_wait_time = self.parent().get_wait_time()  # Get the
current wait time
        index = self.ui.Delay.findText(str(current_wait_time))  # Find
the correct index in the dropdown
        if index != -1:
            self.ui.Delay.setCurrentIndex(index)  # Set the dropdown to
the current wait time


self.OverrideDelay.setChecked(self.parent().settings["override_delay"])
# Update the override delay checkbox

self.WarningMessage.setPlainText(self.parent().settings["warning_messag
e"])  # Update the warning message text box

    # Show the settings page
    def showEvent(self, event):
        super().showEvent(event)
        self.reload_settings_page()  # Reload the settings when the
window is shown

    # Go back to the main window when the back button is pressed
    def go_back(self):
        self.parent().override_delay = self.OverrideDelay.isChecked()
# Update override delay
        self.parent().warning_message =
self.WarningMessage.toPlainText()  # Update warning message

        self.parent().show()  # Show the main window
        self.close()  # Close the settings page

# Main entry point of the application
if __name__ == "__main__":
    app = QApplication(sys.argv)  # Create the application
    app.setAttribute(QtCore.Qt.AA_EnableHighDpiScaling)  # Enable high
DPI scaling
    app.setAttribute(Qt.AA_UseHighDpiPixmaps, True)  # Better scaling
for high-resolution screens
    window = SplashScreen()  # Create the splash screen
    sys.exit(app.exec_())  # Start the event loop
```

Settings.json

```json
{
    "wait_time": "10",
    "sessions": [
        {
            "start_time": "20:00",
            "end_time": "01:00",
            "days": [
                false,
                false,
                false,
                false,
                false,
                true,
                true
            ]
        }
    ],
    "categories": {
        "Social Media": true,
        "Games": true,
        "Video Entertainment": true,
        "Visual Entertainment": false,
        "Forums": false,
        "Online Shopping": false,
        "Productivity Killers": true
    },
    "whitelisted_sites": [],
    "blacklisted_sites": [],
    "productive_sites": [
        "docs.google.com",
        "www.workramp.com/blog/what-is-leadership-development/",
        "www.workramp.com/blog/what-is-sales-training/",
        "slides.google.com",
        "mail.google.com",
        "scholar.google.com",
        "khanacademy.org",
        "coursera.org",
        "edx.org",
        "canva.com",
        "drive.google.com"
    ],
    "override_delay": true,
    "warning_message": "No."
}
```