AI-AgroBot Universal AI-based Agricultural Assistant

Imports & setup

```
1  from flask import Flask, render_template, request, redirect, url_for, flash, session, jsonify,
send_file
2  from io import BytesIO, StringIO
3  import csv
4  import os


5  from database import init_db, db, User, ChatHistory
6  from chatbot_model import process_message


7  app = Flask(__name__)
8  app.secret_key = os.getenv("FLASK_SECRET_KEY", "dev_secret_key")
9  init_db(app)
```

Explanations

1. Imports core Flask objects:
    - Flask — class to create the app object.
    - render_template — render an HTML template.
    - request — access incoming request data (form, query string, etc.).
    - redirect / url_for — generate redirects and URLs for routes.
    - flash — store short messages to show to the user (usually in templates).
    - session — server-signed cookie storage for user session data.
    - jsonify — return JSON responses (useful for AJAX).
    - send_file — send files (e.g., the CSV export) to the client.
2. BytesIO/StringIO — in-memory file-like objects: StringIO for text, BytesIO for binary. Useful for building CSV in memory and then returning it.
3. csv module — to write CSV rows.
4. os — to read environment variables (used for secret key).

5. Import helpers & models from your database module:
    - init_db(app) — function that initializes DB with Flask app (e.g., binds SQLAlchemy).
    - db — the SQLAlchemy (or similar) DB instance.
    - User, ChatHistory — ORM models for users and chat records.
6. process_message — function from chatbot_model that takes user input and returns the bot response (possibly handles translation, AI calls, etc.).
7. Create the Flask app instance. __name__ lets Flask locate templates/static relative to the module.
8. Set the secret key for sessions / flashing. It tries to get FLASK_SECRET_KEY from environment; otherwise falls back to "dev_secret_key" (dev-only fallback; not secure for production).
9. Initialize the database with the app — typically registers db.init_app(app) or similar inside init_db.

User routes — login (index)

```
11 @app.route("/", methods=["GET", "POST"])
12 def index():
13     """Login page"""
14     if request.method == "POST":
15         username = request.form.get("username", "").strip()
16         password = request.form.get("password", "").strip()

17         if not username or not password:
18             flash("Please enter username and password", "warning")
19             return redirect(url_for("index"))

20         # Admin shortcut
21         if username == "admin":
22             return redirect(url_for("admin_login"))
```

```
23        user = User.get_by_username(username)
24        if user and user.check_password(password):
25            session["user_id"] = user.id
26            session["username"] = user.username
27            return redirect(url_for("chat"))
28        flash("Invalid username or password", "danger")
29    return render_template("index.html")
```

Explanations

11. Registers the root URL / accepting GET and POST — acts as the login page.
12. Define the index() view function.
13. Docstring: short description.
14. Checks if the incoming request is a POST (form submission).

15–16. Read username and password from the POSTed form. .get(..., "") avoids None. .strip() removes surrounding whitespace.
17–19. If either field is empty, flash a warning and redirect back to the login page.
20–22. **Admin shortcut:** if username equals "admin", redirect to the admin login page (this does not automatically authenticate — it just sends the user to the admin login route).
23. Calls User.get_by_username(username) — helper that should return the User object or None.
24. If user exists and password check passes (user.check_password(password) should verify hashed password), then:
25–26. Store user_id and username in the session so the app knows the user is logged in.
27. Redirect logged-in user to the chat page.
28. If auth fails, show an error flash message.
29. For GET (or after POST fallback), render index.html (the login template).

User route — registration

```
31 @app.route("/register", methods=["GET", "POST"])
32 def register():
33    """User registration"""
34    if request.method == "POST":
35        username = request.form.get("username", "").strip()
36        password = request.form.get("password", "").strip()
37        if not username or not password:
38            flash("Please enter username and password", "warning")
```

```
39            return redirect(url_for("register"))
40        if User.get_by_username(username):
41            flash("Username already exists", "danger")
42            return redirect(url_for("register"))
43        User.create(username, password)
44        flash("Registered successfully — please login", "success")
45        return redirect(url_for("index"))
46    return render_template("register.html")
```

Explanations
31. Route /register handles user signup.
34. On POST, read and strip username/password from the form.
37–39. If fields are empty, flash and redirect back to registration.
40–42. If a user with that username already exists (via User.get_by_username), flash and redirect.
43. Call User.create(username, password) — should create the user, hash the password, and commit to DB.
44. Flash success message.
45. Redirect to login page after successful registration.
46. For GET, render the registration form template.

User route — chat page + message handling

```
48 @app.route("/chat", methods=["GET", "POST"])
49 def chat():
50    """Chat page — GET shows UI + user's past chats, POST handles a message"""
51    if "user_id" not in session:
52        return redirect(url_for("index"))


53    # POST: incoming message (AJAX form)
54    if request.method == "POST":
55        user_input = request.form.get("message", "").strip()
56        lang = request.form.get("lang", "en")
57        if not user_input:
58            return jsonify({"response": "Please enter a message."})
```

```
59        # Process message -> returns bot response translated to dest_lang
60        bot_response = process_message(user_input, dest_lang=lang)

61        # Save conversation in DB (visible to admin)
62        ChatHistory.create(session["user_id"], user_input, bot_response)

63        return jsonify({"response": bot_response})

64    # GET: show chat UI + previous messages for this user
65    chats = ChatHistory.query.filter_by(user_id=session["user_id"]).order_by(ChatHistory.timestamp.asc()).all()
66    return render_template("chat.html", username=session.get("username"), chats=chats)
```

Explanations

48. Route /chat supports GET (show page) and POST (submit message via AJAX).
51–52. If the user is not logged in (no user_id in session), redirect to login.
54. If POST — the code expects an AJAX form that sends message and lang.
55. Read the user's message and strip whitespace.
56. Read lang form field (default "en" if not provided) — used for language/translation selection.
57–58. If message is empty, return a small JSON response prompting user to enter a message.
60. Call process_message(user_input, dest_lang=lang) — this should call your AI model and return a bot reply. The comment implies the result is translated to dest_lang already.
62. Save the exchange to the DB — ChatHistory.create(user_id, message, response) should persist it.
63. Return the bot response as JSON. Client-side JS will display it without page reload.
65. For GET: load all chat rows for this user ordered ascending by timestamp (oldest → newest).
66. Render chat.html, passing username and chats list to display past conversation.

User route — logout

```
68 @app.route("/logout")
69 def logout():
```

```
70    session.clear()
71    flash("Logged out", "info")
72    return redirect(url_for("index"))
```

Explanations
68. Route /logout.
70. session.clear() removes all data from the session, effectively logging out the user.
71. Flash a small informational message.
72. Redirect to the login page.

Admin login

```
75 @app.route("/admin", methods=["GET", "POST"])
76 def admin_login():
77    """Simple admin login (username=admin / password=admin123 by default)"""
78    if request.method == "POST":
79        username = request.form.get("username", "").strip()
80        password = request.form.get("password", "").strip()
81        if username == "admin" and password == "admin123":
82            session["admin"] = True
83            return redirect(url_for("admin_dashboard"))
84        flash("Invalid admin credentials", "danger")
85    return render_template("admin_login.html")
```

Explanations
75. Admin login route /admin supports GET and POST.
79–80. Read admin credentials from the form.
81–83. Basic (and insecure for production) check: if username and password match hard-coded values, set session["admin"] = True and redirect to admin dashboard.
84. If credentials are wrong, show error message.
85. GET renders the admin login template.

Admin dashboard (view + search)

```
88 @app.route("/admin/dashboard")
89 def admin_dashboard():
90    if not session.get("admin"):
```

```
91        return redirect(url_for("admin_login"))


92     q = request.args.get("q", "").strip()
93     if q:
94        # join with users so we can search username + message + response
95        chats = (ChatHistory.query
96               .join(User, ChatHistory.user_id == User.id)
97               .filter(
98                  (User.username.ilike(f"%{q}%")) |
99                  (ChatHistory.message.ilike(f"%{q}%")) |
100                 (ChatHistory.response.ilike(f"%{q}%"))
101             )
102             .order_by(ChatHistory.timestamp.desc())
103             .all())
104   else:
105      chats = ChatHistory.query.order_by(ChatHistory.timestamp.desc()).all()


106   return render_template("admin_dashboard.html", chats=chats, query=q)
```

Explanations
88. Admin dashboard route.
90–91. Protects the page: if session["admin"] is not truthy, redirect to admin login.
92. Read optional q query parameter for search (e.g., /admin/dashboard?q=something).
95–103. If q exists, build a query:

- Join ChatHistory with User so you can search usernames too.
- Filter where username, message, or response ILIKE (case-insensitive LIKE) the query string (wrap with % for substring match).
- Order results by timestamp descending (newest first) and get .all().

105. If no q, just fetch all chats ordered newest-first.

106. Render the admin template and pass the chats + the q string (so the UI can show current search term).

Admin download (CSV export)

```
109 @app.route("/admin/download")
110 def admin_download():
111     if not session.get("admin"):
112         return redirect(url_for("admin_login"))

113     chats = ChatHistory.query.join(User, ChatHistory.user_id == User.id).order_by(ChatHistory.timestamp.desc()).all()

114     output = StringIO()
115     writer = csv.writer(output)
116     writer.writerow(["ID", "User ID", "Username", "Message", "Response", "Timestamp"])
117     for c in chats:
118         writer.writerow([c.id, c.user_id, c.user.username if c.user else "Unknown", c.message, c.response, c.timestamp])

119     mem = BytesIO()
120     mem.write(output.getvalue().encode("utf-8"))
121     mem.seek(0)
122     output.close()
123     return send_file(mem, mimetype="text/csv", as_attachment=True, download_name="chat_history.csv")
```

Explanations
109. Route /admin/download to export chat history as CSV.
111–112. Block non-admins as before.
113. Fetch chats joined with users, ordered newest-first.
114. Create a StringIO() to write a text CSV in memory.
115. Create a CSV writer bound to that text buffer.
116. Write the header row to the CSV.
117–118. Loop through chat rows and write a CSV row per chat. Uses c.user.username if c.user else "Unknown" in case the relationship is missing.
119. Create a BytesIO() because send_file expects bytes (binary).
120. Convert output text to bytes (UTF-8) and write to the binary buffer.
121. seek(0) rewinds the in-memory file to the start so Flask can read from it.

122. Close the StringIO() buffer.
123. send_file() returns the BytesIO as a file download named chat_history.csv. as_attachment=True forces download.

Note: Using StringIO then converting to BytesIO is a common pattern because csv.writer wants a text file-like object while send_file prefers binary.

Admin: clear all history

```
126 @app.route("/admin/clear_history", methods=["POST"])

127 def clear_history():

128     if not session.get("admin"):

129         return redirect(url_for("admin_login"))

130     ChatHistory.query.delete()

131     db.session.commit()

132     flash("Chat history cleared", "success")

133     return redirect(url_for("admin_dashboard"))
```

Explanations
126. Route /admin/clear_history that only accepts POST (safer than GET for destructive action).
128–129. Admin-only check.
130. ChatHistory.query.delete() issues a bulk delete deleting all rows in the chat_history table (note: bypasses model .delete() hooks — see caution below).
131. Commit the DB transaction to persist the deletion.
132. Flash success message to admin.
133. Redirect back to admin dashboard.

Caution: query.delete() is a bulk operation and may bypass some ORM-level cleanup (and does not automatically cascade via relationships in the same way row-by-row deletes might). If you need safe/complex cleanup, consider iterating .all() and deleting each instance via db.session.delete(instance).

Run the app (development)

```
136 if __name__ == "__main__":

137     app.run(debug=True)
```

Explanations
136. Standard Python module guard — only run this block if the file is executed as script, not when imported.

137. Start Flask’s built-in development server with debug=True (enables reloader, interactive debugger). **Do not** run debug=True in production.