Implemented security & privacy features (what, where, why)

1) Password hashing

- **Where:** `init_db.py` / `app.py` — `werkzeug.security.generate_password_hash` and `check_password_hash`.
- **Why / mitigates:** protects user credentials when DB is leaked (no plaintext passwords).
- **Notes:** uses Werkzeug salted hashing (PBKDF2 by default).
- **Severity handled: High** (password theft risk).
- **Improvement:** enforce stronger password policy (length/complexity), enable account lockout after repeated failures.

---

2) Role-based access control (RBAC)

- **Where:** `app.py` — checks `current_user.role` at endpoints (`worker_dashboard`, `claim`, `complete`, `admin`, `cleanup`).
- **Why / mitigates:** prevents unauthorized users from performing privileged actions (claiming/completing jobs, running cleanup).
- **Severity handled: High** (privilege escalation / unauthorized access).
- **Improvement:** centralise permission checks (decorators) and log admin actions.

---

3) Minimal PII storage & pseudonymisation (email hashing)

- **Where:** `models.py` (field `email_hash`), `app.py` function `hash_email()` used on register.

- **Why / mitigates:** lowers risk of raw personal identifiers being exposed; hashed emails reduce re-identification risk.
- **Severity handled: Medium** (data exposure / privacy).
- **Improvement:** consider storing email separately for communications but encrypted; hashing reduces utility for emailing, so clarify tradeoff.

---

4) Ephemeral visit tokens (privacy-preserving check-ins)

- **Where:** `models.py` booking field `ephemeral_visit_id` (UUID); displayed in `my_bookings.html`.
- **Why / mitigates:** enables contact-tracing / check-in without directly sharing user identity; tokens are random and unlinkable without the mapping.
- **Severity handled: Medium/High** for privacy-sensitive flows.
- **Improvement:** present tokens as QR codes; rotate tokens per visit; limit token lifetime.

---

5) Encrypted sensitive mapping table (`VisitMapping`) with Fernet

- **Where:** `models.py` (`VisitMapping.encrypted_user_id`), `app.py` uses `cryptography.Fernet` to encrypt/decrypt mappings when creating mappings and when reporting exposure.
- **Why / mitigates:** prevents plaintext linking between ephemeral tokens and user IDs in DB; an attacker with DB dump cannot directly map tokens→users without the key.
- **Severity handled: High** (linkage attack / deanonymization).
- **Improvement:** secure key storage (not in repo or env on shared systems), key rotation, use HSM/KMS for production.

---

6) Retention policy & manual cleanup endpoint

- **Where:** `app.py` route `/admin/cleanup` deletes `VisitMapping` entries older than 14 days.
- **Why / mitigates:** limits data retention window to reduce privacy risk and regulatory exposure.
- **Severity handled: Medium** (privacy / data minimization).
- **Improvement:** automate via cron / background job; log deletion events for audit.

---

7) CSRF protection & form validation (server-side)

- **Where:** `forms.py` uses `FlaskForm` (Flask-WTF) which provides CSRF token; templates include `form.hidden_tag()`; WTForms validators used.
- **Why / mitigates:** prevents Cross-Site Request Forgery and enforces input constraints to reduce malformed inputs.
- **Severity handled: High** (CSRF can perform actions as victims).
- **Improvement:** ensure `WTF_CSRF_SECRET_KEY` and `SESSION_COOKIE_SECURE` set in production.

---

8) Session management via Flask-Login

- **Where:** `app.py` — `LoginManager`, `login_user`, `logout_user`, `@login_required` decorators.
- **Why / mitigates:** consistent authentication/session handling; prevents anonymous access to protected routes.
- **Severity handled: High** (unauthorized access).
- **Improvement:** set secure cookie flags, session timeouts, and optionally server-side session store for invalidation.

---

9) Use of SQLAlchemy ORM (prepared statements)

- **Where:** `models.py`, all DB access in `app.py` uses SQLAlchemy ORM sessions.
- **Why / mitigates:** protects against SQL injection and eases safe parameter handling.
- **Severity handled: High** (SQLi risk).
- **Improvement:** continue to avoid raw SQL; if used, parameterize queries.

---

10) Template auto-escaping (XSS mitigation)

- **Where:** Jinja2 templates (`templates/*.html`) auto-escape by default (e.g., `{{ b.service_type.name }}`).
- **Why / mitigates:** defends against stored/reflected XSS when user content displayed.
- **Severity handled: High** (XSS can hijack sessions).
- **Improvement:** still sanitize any HTML allowed in user input; use CSP headers in production.

---

11) Environment-based secrets (SECRET_KEY, FERNET_KEY)

- **Where:** `.env.example` and `app.py` `load_dotenv()` usage.
- **Why / mitigates:** avoids hardcoding secrets in source code (if used correctly).
- **Severity handled: High** for key leakage prevention.
- **Improvement:** do **not** use `.env` in production — use secret manager (AWS KMS/Secrets Manager, Azure Key Vault) and never fallback to a generated key in production (the code currently generates a fallback Fernet key when none provided — insecure).

---

12) In-app notifications (avoids external PII leakage)

- **Where:** `Notification` model and dashboard UI.
- **Why / mitigates:** keeps exposure notifications inside app instead of emailing SMS (which could leak or be intercepted).
- **Severity handled: Low/Medium** depending on config.
- **Improvement:** if using external channels, ensure secure transport and consent.

Summary: 22 security features (implemented vs new)

- **Already implemented (6):**
    1. Password hashing
    2. Role-based access control (RBAC)
    3. Minimal PII/pseudonymisation (email hashing)
    4. Ephemeral visit tokens
    5. Encrypted visit mapping with Fernet
    6. CSRF via Flask-WTF + session management via Flask-Login
- **Add / implement now (16)** to reach ~22 total:
    7. Enforce presence of secrets (no fallback keys) *(fix)*
    8. HTTPS enforcement + secure headers (HSTS, CSP, X-Frame-Options, etc.)
    9. Secure cookie settings (Secure, HttpOnly, SameSite)
    10. Session timeout / idle logout
    11. Login rate limiting & brute-force protection
    12. Account lockout after failed attempts
    13. Password strength policy (and password reuse prevention hint)
    14. Email verification (account activation)

15. Two-Factor Authentication (TOTP) optional (pyotp)
16. Audit logging and immutable admin action log
17. Automated retention job (background task) + scheduled purge
18. Key management improvements + key rotation support (versioned keys)
19. Input sanitization for free-text fields (prevent stored XSS)
20. Content Security Policy (CSP) and other response headers (via Flask-Talisman)
21. Rate-limited APIs + CORS policy
22. File upload restrictions & virus-scan / allowed types (if you accept uploads)

Below you'll find detailed steps and code snippets for each.

1–6: (Already implemented — quick references)

No code needed; these are in your repo:

- Password hashing: `werkzeug.generate_password_hash` — in `init_db.py` / `app.py`.
- RBAC: `current_user.role` checks — `app.py`.
- Minimal PII: `email_hash` and `hash_email()` — `models.py` / `app.py`.
- Ephemeral visit tokens: `Booking.ephemeral_visit_id` — `models.py`.
- Encrypted mapping: `VisitMapping.encrypted_user_id` via `cryptography.Fernet` — `models.py` / `app.py`.
- CSRF: Flask-WTF forms with `form.hidden_tag()` — `forms.py` / templates.

---

7 — Require secrets (no fallback Fernet key)

Why: Prevents accidental weak encryption keys; prevents silent insecure fallback.
Where: replace fallback code in `app.py` (FERNET_KEY handling).

Change in `app.py`:

```
FERNET_KEY = os.getenv("FERNET_KEY")
if not FERNET_KEY:
    raise RuntimeError("FERNET_KEY not set in environment. Generate
with: from cryptography.fernet import Fernet;
print(Fernet.generate_key().decode())")
fernet = Fernet(FERNET_KEY)
```

Notes: Fail fast in dev — this forces secure practices.

---

8 — HTTPS enforcement + secure headers (HSTS)

Why: Ensure TLS; HSTS prevents protocol downgrade.
Dependency: `Flask-Talisman`
Install: `pip install flask-talisman` → add to `requirements.txt`.

Where: `app.py` after `app` creation:

```
from flask_talisman import Talisman

# enforce HTTPS and default secure headers
# In dev you can set content_security_policy=None for quick testing
csp = {
    'default-src': ["'self'"],
    'script-src': ["'self'"],
    'style-src': ["'self'","'unsafe-inline'"]
}
Talisman(app, content_security_policy=csp, force_https=True,
strict_transport_security=True)
```

Note: For local dev you may want to disable `force_https` or run with a dev cert.

---

9 — Secure cookie settings

Why: Prevent JavaScript access and cross-site cookie leakage.
Where: set in `app.config` in `app.py`:

```
app.config.update(
    SESSION_COOKIE_SECURE=True,   # send cookies only over HTTPS
    SESSION_COOKIE_HTTPONLY=True, # not accessible to JS
    SESSION_COOKIE_SAMESITE="Lax" # or "Strict" per UX
)
```

---

10 — Session timeout / idle logout

Why: Limits session hijacking window.
Where: `app.py` add session lifetime and a small middleware to refresh.

```
from datetime import timedelta
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(minutes=30)

# when logging in:
login_user(LoginUser(u))
session.permanent = True
```

Optionally check last activity in `before_request`:

```
from flask import session
@app.before_request
def session_management():
    session.modified = True
```

---

11 — Login rate limiting & brute-force protection

Why: Stop credential stuffing and brute-force.
Dependency: `Flask-Limiter` (`pip install Flask-Limiter`)
Where: `app.py`

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(app, key_func=get_remote_address,
default_limits=["200 per day", "50 per hour"])
# Protect login route:
@app.route("/login", methods=["GET","POST"])
@limiter.limit("5 per minute")
def login():
    ...
```

---

12 — Account lockout after repeated failures

Why: Prevent account compromise after repeated failures.
Approach: simple DB fields: add `failed_attempts`, `locked_until` to `User` model.

models.py additions (User):

```
from sqlalchemy import DateTime
failed_attempts = Column(Integer, default=0)
locked_until = Column(DateTime, nullable=True)
```

app.py: on failed login increment `failed_attempts` and set `locked_until` after threshold:

```
from datetime import datetime, timedelta
MAX_FAIL = 5
```

```
LOCK_FOR = timedelta(minutes=30)

u = s.query(User).filter_by(username=...).first()
if u:
    if u.locked_until and u.locked_until > datetime.utcnow():
        flash("Account locked. Try later.", "danger")
    elif not check_password_hash(u.password_hash, password):
        u.failed_attempts += 1
        if u.failed_attempts >= MAX_FAIL:
            u.locked_until = datetime.utcnow() + LOCK_FOR
            u.failed_attempts = 0
        s.commit()
```

Reset `failed_attempts` on successful login.

---

13 — Password strength policy

Why: Prevent weak passwords.
Where: `forms.py` / registration flow.

Use `password-validator` or simple checks:

```
def is_strong_password(pw):
    return len(pw) >= 10 and any(c.isupper() for c in pw) and
any(c.isdigit() for c in pw)
# In register route:
if not is_strong_password(form.password.data):
    flash("Choose a stronger password (min 10 chars, uppercase,
number).", "danger")
```

---

14 — Email verification (account activation)

Why: Verify ownership & prevent fake accounts.
Dependencies: `Flask-Mail` (or print token for demo), `itsdangerous` (built-in in Flask).
Install: `pip install Flask-Mail`

Flow: on register, generate token, send link `GET /verify/<token>`. Use `itsdangerous.URLSafeTimedSerializer`.

Snippet:

```
from itsdangerous import URLSafeTimedSerializer
s = URLSafeTimedSerializer(app.config['SECRET_KEY'])
token = s.dumps(user.email_hash or user.username, salt='email-verify')
# verification route:
@app.route("/verify/<token>")
def verify(token):
    try:
        data = s.loads(token, salt='email-verify', max_age=3600*24)
        # mark user verified in DB (add field email_verified=True)
    except:
        flash("Invalid or expired token", "danger")
```

---

15 — Two-Factor Authentication (TOTP)

Why: Strong second factor for sensitive roles (workers/admin).
Dependency: `pyotp` (`pip install pyotp qrcode[pil]` if generating QR).
Where: Add fields `totp_secret` & `totp_enabled` to `User`.

Snippet (setup):

```
import pyotp
secret = pyotp.random_base32()
# show QR / secret to user to configure authenticator app
```

Verify during login when user has `totp_enabled`.

---

16 — Audit logging & immutable admin action log

Why: Accountability for admin actions and forensic analysis.
Where: add `AuditLog` model.

models.py

```
class AuditLog(Base):
    __tablename__ = "audit_logs"
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, nullable=True)
    action = Column(String(200))
    details = Column(Text)
    created_at = Column(DateTime(timezone=True),
server_default=func.now())
```

Use: on admin cleanup or role change, insert audit log entry. Do not allow easy deletion (or log deletions elsewhere).

---

17 — Automated retention job (background + scheduled purge)

Why: Ensure mappings are purged automatically.
Options: `cron` on host, or `APScheduler` for in-app scheduled jobs.

Install: `pip install APScheduler`

Snippet:

```
from apscheduler.schedulers.background import BackgroundScheduler

def purge_old_mappings():
    s = SessionLocal()
    cutoff = datetime.utcnow() - timedelta(days=14)
```

```
    s.query(VisitMapping).filter(VisitMapping.created_at <
cutoff).delete()
    s.commit()
    s.close()

scheduler = BackgroundScheduler()
scheduler.add_job(func=purge_old_mappings, trigger="interval", days=1)
scheduler.start()
```

---

18 — Key management & rotation support (versioned Fernet keys)

Why: Enable key rotation for long-term safety.
Approach: store keys with version IDs in config and try-decrypt with all active keys.

app.py (example with list of keys):

```
from cryptography.fernet import MultiFernet, Fernet
# FERNET_KEYS env variable: comma-separated keys
KEYS = os.getenv("FERNET_KEYS", "")
if not KEYS:
    raise RuntimeError("FERNET_KEYS missing")
fernet_list = [Fernet(k) for k in KEYS.split(",")]
multi = MultiFernet(fernet_list)
# encrypt: multi.encrypt(...)
# decrypt: try each with fallback? MultiFernet handles rotation if
first key is newest
```

Note: rotate by prepending new key.

---

19 — Input sanitization for free-text (prevent stored XSS)

Why: Prevent scripts in descriptions/notes.
Dependency: `bleach` (`pip install bleach`)

Where: sanitize `description` before saving:

```
import bleach
clean = bleach.clean(form.description.data, tags=[], strip=True)
booking.description = clean
```

---

20 — Additional response headers & CSP (via Talisman)

(Some overlap w/ #8)
Why: Mitigate XSS, clickjacking, MIME sniffing.

Talisman already sets many headers; also set `X-Content-Type-Options` etc. Talisman does this; if not, set manually:

```
@app.after_request
def set_security_headers(resp):
    resp.headers['X-Content-Type-Options'] = 'nosniff'
    resp.headers['X-Frame-Options'] = 'DENY'
    resp.headers['Referrer-Policy'] = 'no-referrer'
    return resp
```

---

21 — Rate-limited APIs & CORS restriction

Why: Limit abuse and define allowed origins.
Dependency: `Flask-CORS` for CORS (`pip install Flask-Cors`) — but prefer whitelist.

Snippet:

```
from flask_cors import CORS
CORS(app, resources={r"/api/*": {"origins":
["https://yourdomain.edu"]}})
```

APIs can be rate-limited with `Flask-Limiter` as shown in #11.

---

22 — File upload restrictions & antivirus (if you accept uploads)

Why: Prevent malicious uploads.
Where: If you add file upload (e.g., workers upload certificates), enforce allowed extensions, limit size, and scan with clamd/virus scanner.

Snippet for allowed extensions:

```
ALLOWED_EXT = {'pdf','png','jpg'}
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.',1)[1].lower() in
ALLOWED_EXT
# in upload route:
if not allowed_file(file.filename): abort(400)
# save to secured folder with safe filename
from werkzeug.utils import secure_filename
filename = secure_filename(file.filename)
```

Note: For real security, integrate an antivirus service (ClamAV) and quarantining.

Additional small but useful changes (implementation notes)

- **Centralize permission checks** into a decorator `@role_required('admin')` to avoid code duplication.

- **Use server-side session store** for easy session invalidation (Redis). (Add `Flask-Session` and configure `SESSION_TYPE='redis'`).

- **Log security events** (failed login, password change, token use) with structured logs and send to central logging (ELK/CloudWatch) in production.

- **SAST/DAST test**: add CI step to run bandit / semgrep (document in README).

- **Privacy notice & consent**: add a UI page with data/retention explanation and an acceptance checkbox during registration.