Functionality

- Ensure that you complete the project and incorporate the core requirements specified in the brief.
- Add all data models from task 1
- Actually make the program work.
- Use more iterations

Code Organisation

- Don't add files until I'm going to use them (e.g., auth.py and validation.py)
- Add more functions and reduce code duplication.
- Store secret key in external instance (redo if you switch computers):

1. Create config.py in the instance folder.
2. Add: *SECRET_KEY = "your_secret_key_here"*
3. Load it in app.py: *app.config.from_pyfile("config.py")*
4. Add to .gitignore: instance/config.py

- Make comments that explain why the code was written, not just what it does.

User Experience

- Make sure the system is robust.
- Add an error handler page for 500 errors.
- Client: HTML type="email" + simple regex check ; Server: regex + reject invalid input with an error message
- Front-end JS: on submit → set button disabled + show spinner
- Add arial-label where labels aren't visible.
- Add a proper <label for="..."> for every input.
- Password: JS checks length and complexity in real-time, displaying a strength bar.
- Dates: use date picker
- Phone: mask like 07xxx xxxxxx.
- Add forgot password functionality; store token in DB with expiry; send reset link; allow new password to be set.; Mention rate limiting to prevent abuse.
- Add short helper text under complex fields:

* “Password must include...”
* “Date must be in the future”
* “Rewards points are awarded after payment confirmation”

- Server-side validation + email regex

Legal and regulatory guidelines and Standards

- Implement cookies
- Privacy policy, terms and conditions
- Add alt text for images
- Https i
- UserWay Accessibility Pricing
- Implements HTTPS (Flask SSL context for development; trusted SSL certificate in production)
- Cross-browser testing (Chrome, Firefox, Edge)
- Add **CSRF tokens** to forms
- Validate token on submission
- **Rate limiting** on login
- **ORM parameterised queries** (SQL injection prevention)

Suitability of test data

- Add extreme data tests
- Cover calculations and data filtering
- Validation tests for SQL injection attempts / XSS attacks
- Show how issues were identified (manual vs automated testing unclear)

Use of testing to inform the iterative development process

- Include logic errors
- Detail in comments: how errors were identified
- Add a “Version” or “Iteration” column to my test column
- Improve wording for fixes; ~~Fix user dashboard route~~, “Refactored dashboard routing logic to correct redirect path.”
- Add versions x.2, etc., for fixes

Quality of the iteration development process

- Rationale should go more in-depth
- Describe why decisions were made

ADAM NOTES

Build code that isnt specific to a task. Login, registration code, and documentation.

Documentation. Don't overcomplicate it. Treat examiners like they're stupid, why and how you've done this. Don't repeat yourself.

Structure everything beforehand.

Time to set yourself time goals.

Review the mark scheme and allocate more time to areas that earn higher marks.

Backend before frontend.

Things I have to learn how to add aswell

- registration and login with third-party apps (Gmail & X-Twitter)
- Show bug fixes in my documentation, not just the test log

Minimal “Any OS Exam” Distinction build (if you want the shortest winning set)

If you can build these end-to-end, you’re covered for most scenarios:

1. Register/Login + roles
2. Search availability + Booking (with double-booking/capacity rules)
3. Payment status flow (simulated) + receipt
4. Dashboard cards pulling real DB data
5. Accessibility + security basics (validation, CSRF, safe sessions)
6. Rewards/XP system
7. Analytics page (user + admin summary)
8. Testing evidence + fixes