FITFINDER — Project Technical Report

Generated:

Summary

This document describes the FITFINDER project: its purpose, repository layout, Python backend, frontend static files, key libraries and packages, deployment instructions, and a file-by-file explanation. It includes recommended next steps and how to present the project.

```
markdown
FITFINDER — Project Technical Report
```

Generated:

Summary

This document describes the FITFINDER project: its purpose, repository layout, Python backend, frontend static files, key libraries and packages, deployment instructions, and a file-by-file explanation. It includes recommended next steps and how to present the project.

Repository root files

- `app.py` — Main Flask backend serving APIs and static files.
- `app_clean.py` — Canonical, vetted copy of the backend used as reference.
- `requirements.txt` — Python dependencies used to run the app.
- `Procfile` — Process declaration for Heroku/Railway: runs `gunicorn`.
- `README-deploy.md` — Deployment guide (Railway/Heroku).
- `test_generate.py` — Small client script for testing `/api/generate-outfit`.
- `contacts.json` — (Optional) storage file used by contact endpoint.
- `generated_outfits/` — Folder where generated images are saved.
- `uploads/` — Folder for uploaded images.
- `static/` — Frontend static files (HTML pages): `index.html`, `about.html`, `contact.html`, `outfit-generator.html`, `tryon.html`.

High-level architecture

- A Flask application (`app.py`) exposes REST endpoints under `/api/*` and serves static pages from `static/` for the frontend.
- Image generation uses two modes:
- Demo / fallback (Pillow-based placeholder images) when no Hugging Face token is configured.
- Real AI generation via Hugging Face Inference API when `HF_API_TOKEN` is provided.
- Try-on endpoint applies a naive image compositing algorithm (Pillow) on the server; `process_tryon_with_api` exists to forward images to an external try-on API if configured.
- Deployed via platform like Railway or Heroku with `Procfile` and `gunicorn` production server.

Key libraries and why they are used

- Flask (v3.x): main web framework to define routes, handle requests, and serve responses.
- Flask-CORS: enable Cross-Origin Resource Sharing for the API when frontend and backend are served from different origins during development/testing.
- Pillow (PIL): image processing (generating demo images, resizing and compositing for try-on).
- requests: HTTP client for calling Hugging Face Inference API or external try-on services.
- gunicorn: WSGI HTTP server used in production deployments (Procfile).

Configuration via environment variables

- `HF_API_TOKEN` or `HUGGINGFACE_API_TOKEN` — Hugging Face Inference API token. When present, the backend attempts real image generation.
- `HF_MODEL` — Model ID to use with Hugging Face (default `stabilityai/stable-diffusion-2-1`).

- `TRYON_API_URL` — Optional external try-on service endpoint if you want server-to-server try-on processing.

Security considerations

- Do NOT commit API tokens to source control. Use Railway/Heroku environment settings or GitHub Secrets for CI.
- The demo images are generated locally and safe, but if you enable external model APIs, be aware of PII/privacy in uploaded images.
- For production, use HTTPS and limit access to admin endpoints (e.g., `/api/admin/stats`) with authentication.

File-by-file analysis (high-level) — key files

`app.py` (main backend)
- Purpose: Provide REST API endpoints and serve the static frontend.
- Key endpoints:
- `GET /api/health` — simple health/status JSON; also reports whether HF token is present.
- `POST /api/generate-outfit` — accepts JSON `{scene, style, gender, custom_prompt}`; builds a textual prompt and either calls the HF Inference API (if token present) or returns a Pillow demo image as a base64 data URI. Saves the generated PNG to `generated_outfits/`.
- `POST /api/tryon` — expects `person_image` and `cloth_image` files in `multipart/form-data`, tries a naive compositing approach using Pillow and returns a result image.
- `POST /api/contact` — stores contact form submissions into `contacts.json`.
- `GET /api/admin/stats` — returns counts for generated images, uploads, and contacts.
- Static serving: `GET /` and `GET /` mapped to files under `static/`.
- Important behaviors:
- If `HF_API_TOKEN` present: calls `https://api-inference.huggingface.co/models/{HF_MODEL}` with `Accept: image/png` to request image bytes; wraps them as base64 data URI and stores the output file.
- If token absent, it uses `demo_image()` which returns a simple text-based PNG image created with Pillow.
- Image files are saved to `generated_outfits` with a timestamp suffix.

`app_clean.py`
- A canonical copy used for testing and reference; has similar structure as `app.py` but is a stable version that was used to restore the corrupted `app.py`.

`requirements.txt`
- Lists dependencies: Flask, Flask-CORS, Pillow, requests, and gunicorn (for production). Optional ML libs commented out.

`Procfile`
- `web: gunicorn -w 2 -b 0.0.0.0:$PORT app:app` — instructs hosts like Railway or Heroku to run `gunicorn` binding to the provided `$PORT` environment variable.

`README-deploy.md`
- Step-by-step instructions for deploying to Railway or Heroku (already added to the repo).

`test_generate.py`
- A small client script that repeatedly attempts to POST a demo generate request to the server until it succeeds — useful for automated local smoke testing.

`static/` pages
- Simple HTML files for the frontend: `index.html`, `about.html`, `contact.html`, `outfit-generator.html`, and `tryon.html`.
- `outfit-generator.html` was modified to call the relative API endpoint `/api/generate-outfit` instead of absolute development addresses so it works when the backend is co-located.

Operational notes and run commands

- Local run (development):
```powershell
. .\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
python d:\ssr\app.py
```
- Production (Railway/Heroku): the `Procfile` starts gunicorn which serves the `app` WSGI app.
- Health check:
```powershell
Invoke-RestMethod http://127.0.0.1:5000/api/health -Method Get
```

Testing endpoints

- Generate outfit (PowerShell example):
```powershell
$body = @{ scene='casual'; style='minimalist'; gender='female' } | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri http://127.0.0.1:5000/api/generate-outfit -Body $body -ContentType
'application/json'
```
- Try-on: use `multipart/form-data` with two file fields `person_image` and `cloth_image`.

Git / repository notes

- `git` is used for version control. Key workflows done in this project:
- Created a clean canonical backend `app_clean.py` when the original `app.py` became corrupted.
- Replaced the corrupted `app.py` with the canonical content and committed the change.
- Added `Procfile` and `README-deploy.md`, committed, and pushed to `origin/main`.
- Added `PROJECT_REPORT.md` and `PROJECT_REPORT.pdf` to repo.
- To present the repo on GitHub: highlight the `README-deploy.md`, include screenshots, and point to the live Railway URL (once deployed).

Presentation tips

- Start with a short demo: show the live site (Railway URL) and call `/api/health` and `/api/generate-outfit`.
- Then walk through architecture: Flask backend, Pillow fallback, Hugging Face integration, static frontend.
- Show the code for `build_prompt()` and `generate_outfit()` to explain how prompts are assembled and how HF is called.
- Discuss scalability & production concerns: replace dev server with `gunicorn` (done), move images to S3 for persistence, add authentication for admin endpoints.

Next steps / recommended improvements

- Add persistent storage for generated images (S3 or cloud storage) and store metadata in a small DB (SQLite/MySQL/Postgres) for indexing.
- Add user/authentication flow if you want to persist user favorites or profiles.
- Improve try-on algorithm by integrating a specialized try-on model or third-party service; add `TRYON_API_URL` integration.
- Add logging and monitoring (structured logs, Sentry) for production.
- Add unit tests for endpoints and integration tests for deployment.

Appendix — Important code excerpts

`app.py` (top):

    python

```
markdown
FITFINDER — Project Technical Report
```

Generated:

Summary

This document describes the FITFINDER project: its purpose, repository layout, Python backend, frontend static files, key libraries and packages, deployment instructions, and a file-by-file explanation. It includes recommended next steps and how to present the project.

Repository root files

- `app.py` — Main Flask backend serving APIs and static files.
- `app_clean.py` — Canonical, vetted copy of the backend used as reference.
- `requirements.txt` — Python dependencies used to run the app.
- `Procfile` — Process declaration for Heroku/Railway: runs `gunicorn`.
- `README-deploy.md` — Deployment guide (Railway/Heroku).
- `test_generate.py` — Small client script for testing `/api/generate-outfit`.
- `contacts.json` — (Optional) storage file used by contact endpoint.
- `generated_outfits/` — Folder where generated images are saved.
- `uploads/` — Folder for uploaded images.
- `static/` — Frontend static files (HTML pages): `index.html`, `about.html`, `contact.html`, `outfit-generator.html`, `tryon.html`.

High-level architecture

- A Flask application (`app.py`) exposes REST endpoints under `/api/*` and serves static pages from `static/` for the frontend.
- Image generation uses two modes:
- Demo / fallback (Pillow-based placeholder images) when no Hugging Face token is configured.
- Real AI generation via Hugging Face Inference API when `HF_API_TOKEN` is provided.
- Try-on endpoint applies a naive image compositing algorithm (Pillow) on the server; `process_tryon_with_api` exists to forward images to an external try-on API if configured.
- Deployed via platform like Railway or Heroku with `Procfile` and `gunicorn` production server.

Key libraries and why they are used

- Flask (v3.x): main web framework to define routes, handle requests, and serve responses.
- Flask-CORS: enable Cross-Origin Resource Sharing for the API when frontend and backend are served from different origins during development/testing.
- Pillow (PIL): image processing (generating demo images, resizing and compositing for try-on).
- requests: HTTP client for calling Hugging Face Inference API or external try-on services.
- gunicorn: WSGI HTTP server used in production deployments (Procfile).

Configuration via environment variables

- `HF_API_TOKEN` or `HUGGINGFACE_API_TOKEN` — Hugging Face Inference API token. When present, the backend attempts real image generation.
- `HF_MODEL` — Model ID to use with Hugging Face (default `stabilityai/stable-diffusion-2-1`).
- `TRYON_API_URL` — Optional external try-on service endpoint if you want server-to-server try-on processing.

Security considerations

- Do NOT commit API tokens to source control. Use Railway/Heroku environment settings or GitHub Secrets for CI.
- The demo images are generated locally and safe, but if you enable external model APIs, be aware of PII/privacy in uploaded images.
- For production, use HTTPS and limit access to admin endpoints (e.g., `/api/admin/stats`) with

authentication.

File-by-file analysis (high-level) — key files

`app.py` (main backend)
- Purpose: Provide REST API endpoints and serve the static frontend.
- Key endpoints:
- `GET /api/health` — simple health/status JSON; also reports whether HF token is present.
- `POST /api/generate-outfit` — accepts JSON `{scene, style, gender, custom_prompt}`; builds a textual prompt and either calls the HF Inference API (if token present) or returns a Pillow demo image as a base64 data URI. Saves the generated PNG to `generated_outfits/`.
- `POST /api/tryon` — expects `person_image` and `cloth_image` files in `multipart/form-data`, tries a naive compositing approach using Pillow and returns a result image.
- `POST /api/contact` — stores contact form submissions into `contacts.json`.
- `GET /api/admin/stats` — returns counts for generated images, uploads, and contacts.
- Static serving: `GET /` and `GET /` mapped to files under `static/`.
- Important behaviors:
- If `HF_API_TOKEN` present: calls `https://api-inference.huggingface.co/models/{HF_MODEL}` with `Accept: image/png` to request image bytes; wraps them as base64 data URI and stores the output file.
- If token absent, it uses `demo_image()` which returns a simple text-based PNG image created with Pillow.
- Image files are saved to `generated_outfits` with a timestamp suffix.

`app_clean.py`
- A canonical copy used for testing and reference; has similar structure as `app.py` but is a stable version that was used to restore the corrupted `app.py`.

`requirements.txt`
- Lists dependencies: Flask, Flask-CORS, Pillow, requests, and gunicorn (for production). Optional ML libs commented out.

`Procfile`
- `web: gunicorn -w 2 -b 0.0.0.0:$PORT app:app` — instructs hosts like Railway or Heroku to run `gunicorn` binding to the provided `$PORT` environment variable.

`README-deploy.md`
- Step-by-step instructions for deploying to Railway or Heroku (already added to the repo).

`test_generate.py`
- A small client script that repeatedly attempts to POST a demo generate request to the server until it succeeds — useful for automated local smoke testing.

`static/` pages
- Simple HTML files for the frontend: `index.html`, `about.html`, `contact.html`, `outfit-generator.html`, and `tryon.html`.
- `outfit-generator.html` was modified to call the relative API endpoint `/api/generate-outfit` instead of absolute development addresses so it works when the backend is co-located.

Operational notes and run commands

- Local run (development):
```powershell
. .\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
python d:\ssr\app.py
```
- Production (Railway/Heroku): the `Procfile` starts gunicorn which serves the `app` WSGI app.
- Health check:
```powershell
```

```
Invoke-RestMethod http://127.0.0.1:5000/api/health -Method Get
```

Testing endpoints

- Generate outfit (PowerShell example):
```powershell
$body = @{ scene='casual'; style='minimalist'; gender='female' } | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri http://127.0.0.1:5000/api/generate-outfit -Body $body -ContentType 'application/json'
```
- Try-on: use `multipart/form-data` with two file fields `person_image` and `cloth_image`.

Git / repository notes

- `git` is used for version control. Key workflows done in this project:
- Created a clean canonical backend `app_clean.py` when the original `app.py` became corrupted.
- Replaced the corrupted `app.py` with the canonical content and committed the change.
- Added `Procfile` and `README-deploy.md`, committed, and pushed to `origin/main`.
- Added `PROJECT_REPORT.md` and `PROJECT_REPORT.pdf` to repo.
- To present the repo on GitHub: highlight the `README-deploy.md`, include screenshots, and point to the live Railway URL (once deployed).

Presentation tips

- Start with a short demo: show the live site (Railway URL) and call `/api/health` and `/api/generate-outfit`.
- Then walk through architecture: Flask backend, Pillow fallback, Hugging Face integration, static frontend.
- Show the code for `build_prompt()` and `generate_outfit()` to explain how prompts are assembled and how HF is called.
- Discuss scalability & production concerns: replace dev server with `gunicorn` (done), move images to S3 for persistence, add authentication for admin endpoints.

Next steps / recommended improvements

- Add persistent storage for generated images (S3 or cloud storage) and store metadata in a small DB (SQLite/MySQL/Postgres) for indexing.
- Add user/authentication flow if you want to persist user favorites or profiles.
- Improve try-on algorithm by integrating a specialized try-on model or third-party service; add `TRYON_API_URL` integration.
- Add logging and monitoring (structured logs, Sentry) for production.
- Add unit tests for endpoints and integration tests for deployment.

Appendix — Important code excerpts

`app.py` (top):

```python
from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS
from PIL import Image, ImageDraw
import os, io, base64, time, json

app = Flask(__name__, static_folder='static')
CORS(app)

app.config['UPLOAD_FOLDER'] = 'uploads'
app.config['GENERATED_FOLDER'] = 'generated_outfits'
```

`Procfile`:
```
web: gunicorn -w 2 -b 0.0.0.0:$PORT app:app
```

Line-by-line annotated `app.py`

The following section reproduces `app.py` with a concise explanation for each line. Line numbers are included for quick reference.

1: `"""FitFinder - Clean AI-capable backend (single canonical file)`
- Module docstring: describes the purpose of the file and lists endpoints and modes (demo vs HF).

2: `` (blank line) — formatting separator for readability.

3: `This backend is self-contained and avoids duplicate route definitions.`
- Continuation of module docstring.

4: `Endpoints:`
- Docstring lists available HTTP endpoints for quick reference.

5-11: `- GET /api/health ...` (three lines)
- Docstring enumerating the public endpoints.

12: `` (blank line)

13: `import os`
- Standard library module for filesystem/environment operations.

14: `import io`
- Provides `BytesIO` used for in-memory byte buffers when manipulating images.

15: `import json`
- For reading/writing `contacts.json` and other JSON data.

16: `import time`
- Used to timestamp generated filenames.

17: `import base64`
- For encoding/decoding image bytes to/from base64 data URIs.

18: `from datetime import datetime`
- For human-readable timestamps saved with contacts and status responses.

19: `` (blank line)

20: `import requests`
- External HTTP client library used to call Hugging Face Inference API.

21: `from flask import Flask, request, jsonify, send_from_directory`
- Flask imports for creating the app, accessing requests, returning JSON responses, and serving static files.

22: `from flask_cors import CORS`
- Enables Cross-Origin Resource Sharing when frontend and backend might be on different origins.

23: `from PIL import Image, ImageDraw`
- Pillow imports for generating demo images and processing/compositing uploaded images.

24-25: `` (blank lines)

26: `app = Flask(__name__, static_folder='static')`
- Create the Flask application and set `static` as the folder for static assets.

27: `CORS(app)`
- Apply a permissive CORS policy to the app (useful in dev; consider restricting in production).

28: `` (blank line)

29: `app.config['UPLOAD_FOLDER'] = 'uploads'`
- Path where uploaded files (e.g., try-on inputs) will be stored.

30: `app.config['GENERATED_FOLDER'] = 'generated_outfits'`
- Path where generated images are saved.

31: `os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)`
- Ensure the `uploads` directory exists; no exception if already present.

32: `os.makedirs(app.config['GENERATED_FOLDER'], exist_ok=True)`
- Ensure the `generated_outfits` directory exists.

33: `` (blank line)

34: `CONTACTS_FILE = 'contacts.json'`
- File used to persist contact form submissions.

35: `` (blank line)

36: `HF_MODEL = os.environ.get('HF_MODEL', 'stabilityai/stable-diffusion-2-1')`
- Default Hugging Face model ID, overridable via environment variable.

37: `HF_TOKEN = os.environ.get('HF_API_TOKEN') or os.environ.get('HUGGINGFACE_API_TOKEN')`
- Read the HF token from `HF_API_TOKEN` or `HUGGINGFACE_API_TOKEN` env vars. If unset, demo mode is used.

38: `` (blank line)

39: `SCENES = {'casual', 'work', 'date-night', 'workout', 'formal', 'party'}`
- Allowed scene keywords validated in generate endpoint.

40: `STYLES = {'minimalist', 'vintage', 'streetwear', 'comfort', 'bohemian', 'artistic'}`
- Allowed style keywords.

41: `GENDERS = {'female', 'male', 'unisex'}`
- Allowed gender keywords to influence prompt wording.

42: ``

43-61: `SCENE_PROMPTS = { ... }`
- Mapping of `scene` → human-readable prompt fragments used to assemble the final text prompt for HF or demo generation.

62-80: `STYLE_PROMPTS = { ... }`
- Mapping of `style` → prompt fragments that describe the visual style to request.

81-87: `GENDER_PROMPTS = { ... }`
- Map that adjusts the prompt to mention the model's gender or androgynous phrasing.

88: ``

89: `QUALITY = 'high quality, professional fashion photography, realistic fabric texture, studio lighting'`
- Common quality descriptor appended to prompts to bias outputs toward polished images.

90: ``

91: `def build_prompt(scene, style, gender, custom=''):`
- Helper function that composes a textual prompt from the selected options and optional custom text.

92: ` parts = [GENDER_PROMPTS.get(gender, 'fashion model'), 'wearing', SCENE_PROMPTS.get(scene, ''), STYLE_PROMPTS.get(style, '')]`
- Assemble prompt parts using the configured mappings; fallback values keep prompt readable.

93: ` if custom:`
- If user provided a `custom_prompt`, include it.

94: ` parts.append(custom)`
- Add custom prompt text to prompt parts.

95: ` parts.append(QUALITY)`
- Always append quality descriptor to improve visual results.

96: ` return ', '.join([p for p in parts if p])`
- Join non-empty parts with commas into a single string returned to callers.

97: ``

98: `def save_data_uri(data_uri: str, out_dir: str, stem: str) -> str:`
- Helper to decode a `data:image/png;base64,...` string and save it as a timestamped PNG file.

99: ` if not data_uri.startswith('data:image'):`
- Validate incoming string looks like a data URI.

100: ` raise ValueError('expected data uri')`
- Early error if the format is unexpected.

101: ` b64 = data_uri.split(',', 1)[1]`
- Extract the base64 payload portion after the comma.

102: ` raw = base64.b64decode(b64)`
- Decode the base64 into raw bytes.

103: ` os.makedirs(out_dir, exist_ok=True)`
- Ensure output directory exists.

104: ` path = os.path.join(out_dir, f"{stem}_{int(time.time())}.png")`
- Create a unique filename using the provided `stem` and current epoch time.

105: ` with open(path, 'wb') as f:`
- Write the decoded bytes to disk.

106: ` f.write(raw)`
- Save file contents.

107: ` return path`
- Return the absolute path where the image was saved.

108: ``

109: `def demo_image(scene, style, gender, note='demo'):`
- Create a small placeholder PNG that indicates the requested options — used when HF is not configured

or fails.

110: ` img = Image.new('RGB', (768, 512), color=(240, 240, 250))`
- Create a pastel background image using Pillow.

111: ` d = ImageDraw.Draw(img)`
- Create a drawing context for rendering text.

112: ` lines = [gender.upper(), f'{scene} | {style}', 'FitFinder Demo', note]`
- Text lines printed on the demo image for clarity.

113: ` y = 140`
- Vertical offset where text drawing begins.

114: ` for L in lines:`
- Iterate lines and draw them onto the image.

115: ` d.text((40, y), L, fill=(30, 30, 60))`
- Draw each text line at fixed x offset.

116: ` y += 40`
- Advance vertical position for next line.

117: ` buf = io.BytesIO()`
- Prepare an in-memory buffer to hold the PNG bytes.

118: ` img.save(buf, format='PNG')`
- Save the Pillow image to the buffer as PNG.

119: ` return 'data:image/png;base64,' + base64.b64encode(buf.getvalue()).decode('utf-8')`
- Return a base64 data URI representing the image.

120: ``

121: `@app.get('/api/health')`
- Define a simple health check route that is useful for uptime monitoring and local verification.

122: `def health():`
- Handler for the health route.

123: ` return jsonify({'status': 'healthy', 'hf': bool(HF_TOKEN), 'time': datetime.now().isoformat()})`
- Return JSON indicating service state and whether HF is configured.

124: ``

125: `@app.post('/api/generate-outfit')`
- Endpoint to request an outfit image; accepts JSON describing scene/style/gender.

126: `def generate_outfit():`
- Handler implementing generate logic and saving results.

127: ` data = request.get_json(silent=True) or {}`
- Parse incoming JSON safely; fallback to empty dict.

128: ` scene = data.get('scene')`
- Extract requested scene.

129: ` style = data.get('style')`
- Extract requested style.

130: ` gender = data.get('gender')`
- Extract requested gender.

131: ` custom = (data.get('custom_prompt') or '').strip()`
- Optional custom prompt text; ensure it's a stripped string.

132: ` if scene not in SCENES or style not in STYLES or gender not in GENDERS:`
- Validate inputs; reject unknown options early.

133: ` return jsonify({'success': False, 'error': 'invalid scene/style/gender'}), 400`
- Return a 400 error for invalid selections.

134: ` prompt = build_prompt(scene, style, gender, custom)`
- Build the textual prompt to send to the model (or to annotate the demo image).

135: ` try:`
- Start a try/except to catch generation or network failures.

136: ` if HF_TOKEN:`
- If token present attempt a real HF model call.

137: ` url = f'https://api-inference.huggingface.co/models/{HF_MODEL}'`
- Construct the HF Inference endpoint for the configured model.

138: ` headers = {'Authorization': f'Bearer {HF_TOKEN}', 'Accept': 'image/png'}`
- Authorization header and request accept header to request PNG bytes.

139: ` resp = requests.post(url, headers=headers, json={'inputs': prompt}, timeout=120)`
- POST the prompt and wait up to 120s for a model response.

140: ` if resp.status_code == 200:`
- Check for success; HF returns raw image bytes on success here.

141: ` img_uri = 'data:image/png;base64,' + base64.b64encode(resp.content).decode('utf-8')`
- Wrap returned bytes in a base64 data URI for client preview and saving.

142: ` else:`
- HF responded with error; fallback to demo.

143: ` img_uri = demo_image(scene, style, gender, note='hf-failed')`
- Create a demo image indicating HF failure.

144: ` else:`
- No HF token configured: use demo mode.

145: ` img_uri = demo_image(scene, style, gender, note='no-hf')`
- Produce a demo image explaining HF is not configured.

146: ` saved = save_data_uri(img_uri, app.config['GENERATED_FOLDER'], f'outfit_{scene}_{style}')`
- Save the data URI to disk and get the saved path.

147: ` return jsonify({'success': True, 'file': os.path.basename(saved), 'image': img_uri})`
- Return a JSON success response with filename and inline image data URI.

148: ` except Exception as e:`
- Catch-all to avoid exposing stack traces and to return a sane JSON error.

149: ` return jsonify({'success': False, 'error': str(e)}), 500`
- Return 500 with the error message (consider logging in production instead).

150: ``

151: `@app.post('/api/tryon')`
- Try-on endpoint accepts multipart file uploads and attempts a simple compositing.

152: `def tryon():`
- Handler implementing naive overlay-based virtual try-on.

153: ` if 'person_image' not in request.files or 'cloth_image' not in request.files:`
- Validate that both required files are present in the request.

154: ` return jsonify({'success': False, 'error': 'upload both files'}), 400`
- Return a 400 error if missing uploads.

155: ` person = Image.open(request.files['person_image'].stream).convert('RGBA')`
- Load person image into RGBA for compositing.

156: ` cloth = Image.open(request.files['cloth_image'].stream).convert('RGBA')`
- Load cloth image (also RGBA) to preserve alpha if present.

157: ` pw, ph = person.size`
- Person image width and height.

158: ` cw, ch = cloth.size`
- Cloth image width and height.

159: ` target_w = int(pw * 0.55)`
- Heuristic: scale cloth to ~55% of person width.

160: ` scale = target_w / max(cw, 1)`
- Compute scaling factor; avoid division by zero.

161: ` new_size = (max(int(cw*scale),1), max(int(ch*scale),1))`
- New cloth size, ensuring at least 1px dimension.

162: ` cloth_r = cloth.resize(new_size, Image.LANCZOS)`
- Resize cloth using a high-quality resampling filter.

163: ` x = (pw - new_size[0]) // 2`
- Center cloth horizontally on person image.

164: ` y = max(int(ph * 0.25) - new_size[1]//8, 0)`
- Vertical placement heuristic to position cloth over the torso.

165: ` comp = person.copy()`
- Create a copy of person to composite onto.

166: ` comp.alpha_composite(cloth_r, (x,y))`
- Overlay the cloth onto the person respecting alpha channels.

167: ` buf = io.BytesIO()`
- Buffer to hold final PNG bytes.

168: ` comp.convert('RGB').save(buf, format='PNG')`
- Convert to RGB and save to buffer as PNG.

169: ` img_uri = 'data:image/png;base64,' + base64.b64encode(buf.getvalue()).decode('utf-8')`
- Wrap composite image as base64 data URI.

170: ` saved = save_data_uri(img_uri, app.config['GENERATED_FOLDER'], 'tryon')`
- Save to disk via helper; filename uses `tryon` stem.

171: ` return jsonify({'success': True, 'file': os.path.basename(saved), 'image': img_uri})`
- Return success JSON with filename and inline image.

172: ``

173: `@app.post('/api/contact')`
- Endpoint to receive contact form submissions and persist them to `contacts.json`.

174: `def contact():`
- Handler for contact form.

175: ` data = request.get_json(silent=True) or {}`
- Parse JSON body safely.

176: ` name = (data.get('name') or '').strip()`
- Extract and normalize `name` field.

177: ` email = (data.get('email') or '').strip()`
- Extract and normalize `email` field.

178: ` message = (data.get('message') or '').strip()`
- Extract and normalize `message` field.

179: ` if not (name and email and message):`
- Validate required fields are provided and non-empty.

180: ` return jsonify({'success': False, 'error': 'all fields required'}), 400`
- Return 400 if validation fails.

181: ` contacts = []`
- Prepare to load existing contacts.

182: ` if os.path.exists(CONTACTS_FILE):`
- If contacts file exists attempt to read it.

183: ` try:`
- Guard against malformed JSON.

184: ` contacts = json.load(open(CONTACTS_FILE, 'r', encoding='utf-8'))`
- Load JSON array of previous contacts.

185: ` except Exception:`
- If reading fails, fall back to empty list.

186: ` contacts = []`
- Reset contacts on read error to avoid breaking.

187: ` contacts.append({'name': name, 'email': email, 'message': message, 'time': datetime.now().isoformat()})`
- Append new submission with timestamp.

188: ` with open(CONTACTS_FILE, 'w', encoding='utf-8') as f:`
- Persist updated contacts list back to disk.

189: ` json.dump(contacts, f, indent=2)`
- Write pretty-printed JSON for readability.

190: ` return jsonify({'success': True})`
- Return success response for the contact form.

191: ``

192: `@app.get('/api/admin/stats')`
- Admin endpoint returning simple usage/health metrics (counts of files, contacts).

193: `def admin_stats():`
- Handler for admin statistics.

194: ` gen = len([f for f in os.listdir(app.config['GENERATED_FOLDER']) if f.endswith('.png')])`
- Count generated PNG files.

195: ` up = len([f for f in os.listdir(app.config['UPLOAD_FOLDER']) if f.endswith('.png')])`
- Count uploaded PNG files.

196: ` contacts = 0`
- Default contacts count.

197: ` if os.path.exists(CONTACTS_FILE):`
- If contacts file exists attempt to count entries.

198: ` try:`
- Guard reading JSON.

199: ` contacts = len(json.load(open(CONTACTS_FILE, 'r', encoding='utf-8')))`
- Compute number of contact submissions.

200: ` except Exception:`
- If reading fails fall back to zero.

201: ` contacts = 0`
- Ensure contacts is numeric.

202: ` return jsonify({'success': True, 'generated': gen, 'uploads': up, 'contacts': contacts})`
- Return collected counts as JSON.

203: ``

204: `@app.get('/')`
- Serve the SPA index page at the root.

205: `def index():`
- Handler for root.

206: ` return send_from_directory('static', 'index.html')`
- Return `static/index.html` content.

207: ``

208: `@app.get('/')`
- Generic static file handler for other frontend assets.

209: `def static_files(filename):`
- Handler that forwards the path to the static folder.

210: ` return send_from_directory('static', filename)`
- Send the requested static asset.

211: ``

212: `if __name__ == '__main__':`
- Standard Python module entrypoint guard for running locally.

213: ` print('FitFinder starting. HF configured:', bool(HF_TOKEN))`
- Print simple startup log indicating whether HF token is present.

214: ` app.run(host='0.0.0.0', port=5000, debug=False, use_reloader=False)`
- Start Flask dev server bound to all interfaces on port 5000; `use_reloader=False` avoids duplicate processes.

*** End of report content ***