ChatGPT

Hair Salon Website Blueprint (React + Tailwind + AI)

This guide details building a responsive hair salon site with React (frontend) and Python (AI backend). We choose **React** for a component-based frontend and **Tailwind CSS** for utility-driven styling (Tailwind provides "utility classes to speed up the development of responsive websites" [1] [2]). The site includes pages (Home, Services, Stylists, Contact, AI Try-On) and a booking flow with eCommerce-like "add-to-cart" behavior, plus Stripe payment integration. The AI backend (Python) performs face landmark detection, hair segmentation, style overlay, and haircut recommendations. We use pretrained models (no training needed) and ensure library compatibility. Below are the step-by-step implementation details.

1. Tech Stack and Project Setup

- **Frontend:** React (using Create React App or Vite), React Router for page navigation, Tailwind CSS for styling.
- **State Management:** Use React Context or a lightweight state library to manage booking cart.
- **Booking/Payment:** Stripe's official libraries (e.g. `@stripe/react-stripe-js`) for online payments [3].
- **AI Backend:** Python with Flask or FastAPI to serve AI endpoints. Libraries: MediaPipe for face landmarks, PyTorch & Hugging Face Transformers for segmentation (SegFormer model), OpenCV/Pillow for image processing.
- **Models (download links):**
- *Face Landmarker:* Provided by MediaPipe (installable via `pip install mediapipe`) [4].
- *Hair Segmentation:* Use the Hugging Face SegFormer model "jonathandinu/face-parsing" (from CelebAMask-HQ) [5] (downloadable via the HF Model Hub).
- *Hair Type Classification:* A ResNet18-based classifier (e.g. from Kavya-sree/Hair-Type-Classifier) [6].
- *(Optional:* Pretrained face-shape classifier or use custom logic.*)*
  Ensure all Python packages (mediapipe, torch, torchvision, transformers, fastapi/flask, etc.) are installed in one environment so dependencies align.

2. Frontend Setup (React + Tailwind)

1. **Initialize React App:** Create a new app (e.g. `npx create-react-app salon-app` or using Vite: `npm create vite@latest salon-app --template react`).
2. **Install Tailwind:** In the project directory, run `npm install tailwindcss postcss autoprefixer` [1]. Create `tailwind.config.js` (`npx tailwindcss init`) and set its `content` paths to your React `src` files. In `src/`, create a CSS file (e.g. `tailwind.css`) and add:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Import this CSS in your `index.js` (or `index.css`) to enable Tailwind [2]. Now Tailwind's utility classes (e.g. `p-4`, `text-center`, `grid`, etc.) can be used in JSX.

3. **Install Dependencies:** Add React Router (`npm install react-router-dom`) for multi-page routing. Add state management if needed (e.g. Context API or lightweight store). Add Stripe libraries for React (`npm install @stripe/react-stripe-js @stripe/stripe-js`). Also install a date-time picker for bookings (e.g. `react-datepicker` or a Tailwind-friendly calendar).
4. **Setup Tailwind Components:** Optionally install UI components (e.g. Headless UI or Flowbite) for accessible elements styled with Tailwind.

3. Building Frontend Pages

- **Home Page:** Create a React component (`Home.jsx`) with sections "About Us" and site intro. Use Tailwind classes to layout text/images responsively.
- **Services Page:**
- Display categories (Haircuts, Coloring, Styling, Treatment). Use state to allow **filtering** by category (e.g. tabs or dropdown).
- List services (each with name, description, price) in a grid or list using Tailwind. Each service has an "Add" button.
- When a user clicks "Add", add the service to a booking cart (maintained in React state or Context). Simulate an eCommerce cart panel (a sidebar or modal) that shows selected services.
- **Cart/Booking Panel:** This panel (or section) shows selected services ("cart") and total price. Include a "Checkout" button.
- **Stylists Page:** List stylists with photos/bios. Could be static or fetched from data.
- **Contact Page:** Simple contact form (name, email, message) and contact info. No special logic, just frontend.
- **AI Try-On Page:** This page lets users upload a photo and try hairstyles. Provide:
- An `<input type="file">` to upload an image.
- On upload, send the image to the Python backend via an API (see Section 5).
- Display returned image (after overlay) and recommendations.
- Provide buttons or dropdown to choose different hairstyle overlays (front-end triggers backend processing for each selection).

Use React Router to navigate among these pages. All pages should be mobile-responsive via Tailwind's utility classes (e.g. `md:grid-cols-2`, `flex flex-col`, etc.).

4. Booking Flow and Calendar

Implement the booking/order flow like an eCommerce checkout:

1. **Add to Cart (Panel):** Users select services (from Services page), which update a shared state (e.g. React Context). The cart/panel shows current selections.

2. **Proceed to Booking Page:** On clicking “Checkout” or “Book Now,” route to a Booking page. Transfer cart contents to this page.
3. **Calendar & Time Picker:** On the Booking page, include a calendar and time picker for scheduling. Use a React date/time picker library (e.g. react-datepicker or a Tailwind DatePicker). Allow selecting a date and time slot for the appointment.
4. **Payment Options:** Show summary (selected services + stylist if chosen) and total. Provide two options:
5. **Pay Online:** Integrate Stripe to collect card payment.
6. **Pay in Salon:** If chosen, skip payment processing (the user will pay onsite).

For online payment, use Stripe’s React SDK to create a secure payment form. On form submission, send order info (amount, email, items) to your backend to create a PaymentIntent. Then use `stripe.confirmPayment()` on the client to finalize [3]. (Alternatively, use Stripe Checkout Sessions for a faster integration.)

Ensure to secure your Stripe API keys (use environment variables: e.g. `REACT_APP_STRIPE_PK` for publishable key and keep secret key on backend).

5. Stripe Integration

Stripe provides official SDKs for web (including React) and server (Node, Python, etc.) [3]. We will use the React/JS libraries for the frontend and the Stripe Python/Node library on the backend:

- **Frontend (React):** Install `@stripe/react-stripe-js` and `@stripe/stripe-js`. Wrap the checkout form in `<Elements stripe={stripePromise}>` with your publishable key [7]. Include `<PaymentElement>` or custom card form and call `stripe.confirmPayment` on submit.
- **Backend (Python or Node):** Set up an endpoint (e.g. `/create-payment-intent`) that receives cart details (items, currency, amount) and calls `stripe.PaymentIntent.create(...)`. Return the client secret to the frontend. Use Stripe’s official Python SDK (`pip install stripe`) or Node SDK (`npm install stripe`) on the server.
- **Pay-in-Salon:** Simply finalize the order without processing Stripe. You may still create a Stripe PaymentIntent with `amount=0` or skip Stripe entirely, depending on your flow.

Refer to Stripe docs for full examples. The key point is to use official Stripe libraries so payment data is handled securely [3].

6. AI Backend: Face Detection & Segmentation

Set up a Python server (Flask or FastAPI) to handle AI tasks. Below are the components:

- **Install Dependencies:**

```
pip install mediapipe opencv-python pillow torch torchvision transformers
fastapi uvicorn
```

- **Face Landmark Detection:** Use Google's MediaPipe Face Landmarker [4]. This provides a pretrained model (bundled with the `mediapipe` package) that outputs 3D facial landmarks. For example:

```
import mediapipe as mp
mp_face = mp.tasks.vision.FaceLandmarker.create_from_options(
    mp.tasks.vision.FaceLandmarkerOptions(
        base_options=mp.tasks.BaseOptions(model_asset_path='path/to/
face_landmarker.task'),
        running_mode=mp.tasks.vision.RunningMode.IMAGE))
```

(You may need to download the `.task` model file from MediaPipe's GitHub or use their default one. MediaPipe docs explain how to get the face_landmarker.task file [8].) Landmarks include points on eyes, nose, face contour, which will help estimate face shape or overlay position.

- **Hair Segmentation:** Use a semantic segmentation model to isolate hair. A convenient option is the **SegFormer face-parsing model** (CelebAMask-HQ) from Hugging Face [5]. In Python:

```
from transformers import SegformerImageProcessor,
SegformerForSemanticSegmentation
processor = SegformerImageProcessor.from_pretrained("jonathandinu/face-
parsing")
seg_model = SegformerForSemanticSegmentation.from_pretrained("jonathandinu/
face-parsing").to(device)
```

Run the processor on the input image to get logits, then take the argmax to get a label mask. Label **13** corresponds to hair [9]. You can create a binary hair mask by `mask = (labels == 13)`. Save or return this mask image as needed for overlay.

7. Hair Style Overlay Processing

Once you have a hair mask, overlaying a new hairstyle image works as follows:

1. **Prepare Hairstyle Assets:** Have a set of hairstyle images (PNGs with transparent background) for each style. These should roughly align with a front-facing head.
2. **Alignment (Optional):** Use face landmarks (eyes, ears) to estimate head size/orientation. Optionally scale/rotate the hairstyle image to fit the user's face position. This can use affine transforms (OpenCV's `cv2.getAffineTransform` / `warpAffine`) based on three landmarks (e.g. left ear, right ear, top-of-head estimate).
3. **Overlay:** Use the hair mask to combine images. For example, convert user image to OpenCV matrix, and paste the hairstyle PNG over the hair region:

```
result = user_image.copy()
style_img = cv2.imread("chosen_style.png", cv2.IMREAD_UNCHANGED)  # RGBA
```

```
# Resize style_img to match user face width
# Create mask from style_img alpha channel
alpha = style_img[:, :, 3] / 255.0
for c in range(3):
    result_face = result[y:y+h, x:x+w, c]
    style_face = style_img[:, :, c]
    result[y:y+h, x:x+w, c] = (1-alpha)*result_face + alpha*style_face
```

Here (x,y) is where the style should be placed. You can also simply overlay by `cv2.addWeighted` or using NumPy blending, guided by the hair mask or style alpha.

No pretrained model is needed for overlay beyond image processing.

8. Hair Type Classification & Recommendation

- **Hair Type Classification:** Use an image classifier for hair texture/type (straight, wavy, curly, etc.). For instance, a ResNet18 model pretrained on hair images can classify five types [6]. You can load such a model (from the Hair-Type-Classifier repo) and run it on the user's uploaded photo or cropped hair region. This yields hair type tags.

- **Face Shape Determination:** Using facial landmarks, measure dimensions (e.g. face width vs. length, jawline angles) to classify shape (round, square, heart, oval, etc.). For example, if face width ≈ face height with soft jawline, mark as *round*; if wide jawline with equal forehead width, mark *square*, etc. (This can be rule-based without a model.)

- **Recommendation Logic:** Based on face shape and hair type, recommend styles. For example, style guides note that **round faces** benefit from hairstyles with height or asymmetry. One source suggests: *"Short hair is particularly flattering on round faces... Try a deep side part to create more angles or pull hair into a high ponytail to elongate the face"* [10]. Encode such rules: e.g. if round face & straight hair → suggest side-parted bob or layers; if square face → suggest soft bangs or textured waves; etc. These recommendations can be returned as text or style image IDs for the frontend.

9. Backend API and Integration

- **API Endpoints:** Build endpoints in Flask/FastAPI, e.g.:
- `POST /process-image`: Accepts the uploaded user image and a style choice. The backend runs the above steps: detects landmarks, segments hair, overlays the chosen style image, classifies hair type/face shape, and returns the result image and a list of recommended style names.
- `GET /download-models`: (optional) Endpoint to fetch model files if not bundled.
- `POST /create-payment-intent`: (for Stripe) takes cart details, creates a Stripe PaymentIntent, returns client secret.

- **Cross-Origin:** If frontend and backend run on different ports (e.g. 3000 & 8000), enable CORS on the Python server so React can call it (e.g. `pip install fastapi[all]` and

`add_middleware(CORSMiddleware, allow_origins=["http://localhost:3000"], allow_methods=["*"])`).

- **Environment Configuration:**

- Store API URLs (backend) and keys (Stripe public) in React's `.env` (e.g. `VITE_AI_API_URL` or `REACT_APP_API_URL`).
- In Python `.env` or config, store Stripe secret key and paths to model files.

10. Running the Site Locally

1. **Install Frontend:** In `salon-app/`, run `npm install`. Build Tailwind (if using PostCSS watch) and start React: `npm start`.
2. **Install Backend:** In `salon-backend/`, create a virtual env (`python -m venv venv && source venv/bin/activate`), then `pip install -r requirements.txt` (including fastapi, uvicorn, mediapipe, torch, transformers, etc.). Download or place model files as needed (MediaPipe face_landmarker, SegFormer weights).
3. **Start Backend:** Run (for FastAPI) `uvicorn main:app --reload` (or for Flask, `flask run`). Ensure it listens on a port (e.g. 8000) that React will use in API calls.
4. **Test Flow:** Open React frontend (http://localhost:3000) and navigate pages. Test Services → add to cart → booking → calendar → (Stripe form appears) → if testing pay-in-salon, skip payment to success.
5. **Test AI Page:** On "AI Try-On" page, upload a selfie. Verify the backend returns an image with a hairstyle overlay and some recommended styles. Switch styles to ensure overlay changes.

Each step should work without errors. If a model isn't loading, double-check its download link/path. For example, to download the face-parsing model, you can use Hugging Face's `from_pretrained` which handles it automatically, or manually download weights from the Hugging Face repository page [5]. Similarly, Stripe test keys should be obtained from your Stripe dashboard.

References

- Tailwind CSS integration in React (install steps) [1] [2]
- Stripe official SDKs for web (React) and server [3]
- MediaPipe Face Landmarker for facial landmarks [11] [4]
- Hugging Face SegFormer face-parsing model (hair segmentation) [5] [9]
- Hair type classification with ResNet18 (classifying hair into Straight, Wavy, Curly, Kinky, Dreadlocks) [6]
- Hairstyle recommendations by face shape (e.g. styles for round faces) [10]

---

[1] [2] Start a React app with Tailwind CSS in under 5 minutes | Contentful
https://www.contentful.com/blog/react-app-tailwind-css/

[3] docs.stripe.com
https://docs.stripe.com/sdks

4 8 11 Face landmark detection guide for Python | Google AI Edge | Google AI for Developers
https://ai.google.dev/edge/mediapipe/solutions/vision/face_landmarker/python

5 9 jonathandinu/face-parsing · Hugging Face
https://huggingface.co/jonathandinu/face-parsing

6 GitHub - Kavya-sree/Hair-Type-Classifier: A hair type classifier build using fastai
https://github.com/Kavya-sree/Hair-Type-Classifier

7 How I integrated Stripe Payments in React App | by Judy@webdecoded | Medium
https://judy-webdecoded.medium.com/how-i-integrated-stripe-payments-in-react-app-db24cd5c200b

10 What Hairstyle Suits Me: Best Womens Hairstyles For Different Face Shapes
https://www.luxyhair.com/blogs/hair-blog/hairstyles-for-different-face-shapes?srsltid=AfmBOopwGshkaiYF1Oe2veGnsi-unNNH8Si7gIciDhTeu7c7OqqP0WM2