ChatGPT

Full-Stack App Development Plan

This plan outlines a step-by-step approach to building a React frontend and Django REST API backend for an e-commerce site (with routes like Home, Signup, Cart, etc.), integrating JWT auth, email/OTP verification, payments (Razorpay/Juspay), and a Google Gemini chatbot. We'll cover project setup, folder structure, routing, APIs, security, and best practices with code snippets and tables for clarity.

Frontend: React Application

Project Initialization and Folder Structure

- **Initialize the project** using a modern React toolchain (e.g. `create-react-app` or Vite). Create a `src/` folder for all source code.
- **Feature-based folders:** Organize code by feature (domain) rather than by technical type. Store **reusable UI components** in `src/components/`, and feature-specific code in `src/features/<feature>/**` [1] [2]. For example, create folders like `src/features/auth/`, `src/features/cart/`, `src/features/categories/`, etc. Each feature folder contains its React components, Redux slice, and related logic.

- **Example structure:**

```
src/
  ├── App.js              # Root component with Router
  ├── index.js            # ReactDOM render
  ├── components/         # Reusable UI components (buttons, inputs, etc.)
  ├── features/
  │   ├── auth/           # Authentication (login, signup, verification)
  │   │   ├── LoginPage.js
  │   │   ├── SignupPage.js
  │   │   ├── authSlice.js   # Redux slice (login/signup state)
  │   │   └── ...
  │   ├── home/
  │   │   ├── HomePage.js
  │   │   └── homeSlice.js
  │   ├── cart/
  │   │   ├── CartPage.js
  │   │   ├── cartSlice.js
  │   │   └── ...
  │   ├── categories/
  │   │   ├── CategoriesPage.js
  │   │   ├── CategoryDetail.js
  │   │   └── categoriesSlice.js
  │   ├── payment/
  │   │   ├── PaymentPage.js
  │   │   ├── paymentSlice.js
  │   │   └── ...
  │   ├── support/
  │   │   ├── ChatbotPage.js
  │   │   └── supportSlice.js
  │   └── ...
  ├── app/
  │   ├── store.js      # Redux store setup
  │   └── rootReducer.js (if not using RTK configureStore)
  └── styles/          # Global CSS or theming files
```

- **Feature folders approach:** This aligns with Redux and React best practices, keeping each feature's components and state logic together [1] [2]. For instance, all authentication-related components and the Redux slice live under `features/auth/`, while cart-related code lives under `features/cart/`.

Routing with React Router DOM

- **Setup:** Install and configure React Router v6. Wrap the app in `<BrowserRouter>` (in `index.js` or `App.js`) and define routes with `<Routes>` and `<Route>` [3].
- **Routes table:** For clarity, define routes and corresponding components. For example:

Page	Route Path	Component/Page
Home	`/`	`HomePage`
Signup	`/signup`	`SignupPage`
Login	`/login`	`LoginPage`
Cart	`/cart`	`CartPage`
Categories (List)	`/categories`	`CategoriesPage`
Category (Detail)	`/categories/:id`	`CategoryDetailPage`
Payment	`/payment`	`PaymentPage`
Email Verification	`/verify-email`	`VerifyEmailPage`
OTP Verification	`/verify-otp`	`VerifyOTPPage`
Chatbot Support	`/support` (or `/chat`)	`ChatbotPage`

- **Example code snippet:** In `App.js`, use React Router as follows [3]:

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import HomePage from './features/home/HomePage';
import SignupPage from './features/auth/SignupPage';
import LoginPage from './features/auth/LoginPage';
// ... other imports

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/signup" element={<SignupPage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/cart" element={<CartPage />} />
        <Route path="/categories" element={<CategoriesPage />} />
        <Route path="/categories/:id" element={<CategoryDetailPage />} />
        <Route path="/payment" element={<PaymentPage />} />
        <Route path="/verify-email" element={<VerifyEmailPage />} />
        <Route path="/verify-otp" element={<VerifyOTPPage />} />
        <Route path="/support" element={<ChatbotPage />} />
        {/* Fallback 404 */}
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </Router>
  );
}
export default App;
```

This configures all required routes; dynamic segments (e.g. `:id`) handle category details. The router must wrap the app (as shown) to enable navigation.

Redux State Management

- **Redux Toolkit:** Use Redux Toolkit (RTK) to simplify setup. Create a **Redux slice** for each feature (e.g. `authSlice.js`, `cartSlice.js`). Each slice contains state, reducers, and actions in one file ("ducks"/slice pattern) [2].
- **Store setup:** In `app/store.js`, configure the store (optionally using `configureStore`) and include all feature reducers. For example:

```
import { configureStore } from '@reduxjs/toolkit';
import authReducer from '../features/auth/authSlice';
import cartReducer from '../features/cart/cartSlice';
import categoriesReducer from '../features/categories/categoriesSlice';
// ...

export const store = configureStore({
  reducer: {
    auth: authReducer,
    cart: cartReducer,
    categories: categoriesReducer,
    // ...
  }
});
```

- **Feature folder logic:** Place each slice file inside its feature folder. The official Redux style guide recommends this feature-based grouping, keeping related logic together [2].
- **Example slice snippet:** For the cart feature (in `features/cart/cartSlice.js`):

```
import { createSlice } from '@reduxjs/toolkit';

const cartSlice = createSlice({
  name: 'cart',
  initialState: { items: [], total: 0 },
  reducers: {
    addItem(state, action) {
      state.items.push(action.payload);
      state.total += action.payload.price;
    },
    removeItem(state, action) {
      state.items = state.items.filter(item => item.id !==
action.payload.id);
    },
    clearCart(state) {
      state.items = [];
      state.total = 0;
    }
  }
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

- **Usage:** Components connect to Redux via `useSelector` and `useDispatch` hooks. Keep components simple by moving business logic into slices and async thunks if needed.

Pages and Components

- **Page components:** Build one main component per page (e.g. `HomePage.js`, `SignupPage.js`, etc.), placed in the respective feature folder.
- **UI components:** Put reusable UI bits (buttons, form inputs, etc.) in `src/components/`. These are imported by pages or feature components as needed [1].

- **Dropdown for Categories:** For a category dropdown, create a `CategoryDropdown` component in `components/`, used on multiple pages. Its page view (e.g. `CategoriesPage.js`) would display all categories or a selected category full-page.
- **Chatbot page:** The support page (e.g. `ChatbotPage.js`) can use local React state to manage the chat UI and dispatch or call an API when the user sends a message.

By keeping components modular and grouping code by feature, the frontend remains organized and maintainable [1] [2].

Backend: Django REST API

Project Setup and Folder Structure

- **Initialize a Django project:** Create a new Django project (e.g. `django-admin startproject backend`) and one or more apps (e.g. `python manage.py startapp accounts`, `store`, `payments`, `support`).
- **Apps by domain:** For example:
- `accounts/` – user models, authentication, email/OTP verification.
- `store/` – product and category models, cart management.
- `payments/` – endpoints for payment gateway integration.
- `chat/` or `support/` – chatbot message storage (if needed).
- **Folder structure:** An example layout:

```
backend/
  ├── manage.py
  ├── backend/                # Project settings
  │   ├── settings.py
  │   ├── urls.py
  │   └── ...
  ├── accounts/              # User auth, verification
  │   ├── models.py (User, OTP, etc.)
  │   ├── serializers.py
  │   ├── views.py
  │   ├── urls.py
  │   └── ...
  ├── store/                 # Products, categories, cart
  │   ├── models.py (Category, Product, Cart, CartItem, etc.)
  │   ├── serializers.py
  │   ├── views.py
  │   └── urls.py
  ├── payments/              # Payment processing
  │   ├── views.py
  │   ├── urls.py
  │   └── ...
  ├── support/               # Chatbot logging
  │   ├── models.py (ChatMessage)
  │   ├── views.py
  │   ├── urls.py
  │   └── ...
  └── requirements.txt
```

- **Virtual environment & dependencies:** Use a virtualenv. Install **Django**, **djangorestframework**, **djangorestframework-simplejwt** (JWT auth), **django-cors-headers** (CORS), **psycopg2** (Postgres), **google-generativeai** (Gemini API), and payment SDKs (e.g. `razorpay` Python SDK).
- **Settings adjustments:** In `settings.py`, configure installed apps (`rest_framework`, `rest_framework_simplejwt`, etc.) and CORS. For CORS:

```
INSTALLED_APPS += ["corsheaders"]
MIDDLEWARE = ["corsheaders.middleware.CorsMiddleware", ...]
CORS_ALLOWED_ORIGINS = ["http://localhost:3000"]  # React dev server
```

  Use `django-cors-headers` to allow requests from the React frontend domain.

Supabase PostgreSQL Database

- **Use Supabase:** Supabase provides a hosted PostgreSQL. Obtain the connection URI from Supabase Dashboard.
- **Configure Django `DATABASES`:** For example, using `dj-database-url` or manually:

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'postgres',
    'USER': 'your_username',
    'PASSWORD': 'your_password',
    'HOST': 'db.<PROJECT>.supabase.co',
    'PORT': '5432',
  }
}
```

  The connection string looks like `postgresql://username:password@host:5432/postgres` [4]. Store these in environment variables for security.
- **Migrations:** After configuring, run `python manage.py migrate` to set up the schema on Supabase.

JWT Authentication and User Accounts

- **Simple JWT:** Use `djangorestframework-simplejwt` for JWT auth. In `settings.py`, configure the REST framework default auth class:

```
REST_FRAMEWORK = {
  'DEFAULT_AUTHENTICATION_CLASSES': (
    'rest_framework_simplejwt.authentication.JWTAuthentication',
  )
}
SIMPLE_JWT = {
  'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
  'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
  # ... other settings ...
}
```

This matches recommended config in tutorials [5].

- **User model:** Optionally use a custom user model with email as username. Ensure email uniqueness and fields for verification (e.g. `is_active` or an email-verified flag).
- **Signup flow (link verification):**
- **Register endpoint (`POST /api/auth/signup/`):** Accept user info (email, password). Create user with `is_active=False`.
- **Email confirmation link:** Generate a unique token (e.g. using JWT or Django's `TimestampSigner`) and send an email containing a link to `/verify-email/?token=<token>`. You can embed the JWT refresh token or a specially signed URL.
- **Verify endpoint (`GET /api/auth/verify-email/`):** When the user clicks, extract and validate the token. If valid, activate the user (`user.is_active=True`) and return success.
- **OTP flow:**
- **Generation (`POST /api/auth/verify-otp/`):** Alternatively, upon signup or when requesting verification, generate a random 6-digit OTP, save it (e.g. in a model or cache) with an expiry (e.g. 5 minutes), and email it to the user.
- **Verification (`POST /api/auth/verify-otp/`):** User submits the code; verify it matches and is not expired. On success, mark email verified (`user.is_active=True` or a separate `email_verified` flag).
- Tutorials suggest using a serializer to capture the OTP and token, then checking them in a view [6].
- **Login endpoint (`POST /api/auth/login/`):** After email is verified, allow login by validating credentials and returning JWT access and refresh tokens (provided by Simple JWT).
- **JWT handling:** Store tokens securely. It is best practice to send JWTs in HTTP-only cookies or use secure storage. On the frontend, the refresh token flow should be handled (e.g. `/api/token/refresh/` endpoint from Simple JWT).

API Endpoints Overview

Expose REST endpoints for all functionality. Example endpoint table:

Method	Endpoint	Description	Auth Required
POST	`/api/auth/signup/`	Register a new user (sends verification email/OTP)	No
POST	`/api/auth/login/`	Obtain JWT tokens	No

Method	Endpoint	Description	Auth Required
GET	`/api/auth/verify-email/`	Verify account via email token	No
POST	`/api/auth/verify-otp/`	Verify account via OTP code	No
GET	`/api/products/`	List products	No (or token)
GET	`/api/categories/`	List categories	No
GET	`/api/categories/<id>/`	Get category details or products	No
GET	`/api/cart/`	Get current user's cart	Yes (JWT)
POST	`/api/cart/add/`	Add item to cart	Yes
PUT	`/api/cart/update/`	Update cart item qty	Yes
DELETE	`/api/cart/remove/`	Remove item from cart	Yes
POST	`/api/payment/create-order/`	Create payment order (server-side)	Yes
POST	`/api/payment/verify/`	Verify payment signature (callback)	Yes/CSRF-exempt
POST	`/api/chatbot/message/`	Send user message to Gemini, get reply	Yes

Adjust endpoints as needed. Use DRF **serializers** for input validation and **viewsets/views** for logic. Protect endpoints with permissions (e.g. `IsAuthenticated` for private actions). Use **pagination** or filtering on listing endpoints if data is large.

Payment Gateway Integration (Razorpay example)

- **Backend (Django):** Install the Razorpay Python SDK (`pip install razorpay`).
- **Obtain API keys** from Razorpay dashboard (`RAZOR_KEY_ID` , `RAZOR_KEY_SECRET`), store in settings/secrets.
- **Create order:** In your Django view (e.g. `create_order`), use the SDK to create a payment order before the user pays. Example code (from Razorpay docs) [7] :

```
import razorpay
from django.conf import settings
def create_order(request):
    client = razorpay.Client(auth=(settings.RAZOR_KEY_ID,
settings.RAZOR_KEY_SECRET))
    data = {"amount": 50000, "currency": "INR", "receipt":
"order_rcptid_11"}  # amount in paise
    razorpay_order = client.order.create(data=data)
    return JsonResponse({
        'order_id': razorpay_order['id'],
        'amount': data['amount'],
        'currency': data['currency'],
    })
```

This generates a `order_id` to send to the frontend. Amount is in smallest currency unit (paise) [7].

- **Frontend:** The React payment page will load Razorpay's Checkout script, taking `order_id`, `amount`, `key_id` from the backend response.
- **Payment callback:** Configure Razorpay to POST payment status to a Django endpoint (or use the checkout callback). In the callback view, verify the signature to confirm authenticity [8] [9]. Example (from Razorpay docs) [9]:

```
@csrf_exempt
def payment_verify(request):
    if request.method == "POST":
        client = razorpay.Client(auth=(settings.RAZOR_KEY_ID,
settings.RAZOR_KEY_SECRET))
        params = {
          'razorpay_order_id': request.POST['razorpay_order_id'],
          'razorpay_payment_id': request.POST['razorpay_payment_id'],
          'razorpay_signature': request.POST['razorpay_signature']
        }
        try:
            client.utility.verify_payment_signature(params)
            # On success: capture payment, update order in DB, return
success
            client.payment.capture(request.POST['razorpay_payment_id'],
amount)
            return JsonResponse({'status': 'Payment successful'})
        except razorpay.errors.SignatureVerificationError:
            return JsonResponse({'error': 'Payment verification failed'},
status=400)
```

This matches Razorpay's example of verifying signature [9]. Always capture the payment after signature verification.

- **Alternative (Juspay):** Similar flow – backend creates an order/token via Juspay API, frontend completes payment, backend verifies. The key is to handle the server-side order creation and verification securely.
- **Best practice:** Keep all secret keys on the server. Do not trust payment success messages from the frontend without server verification. Use CSRF exemptions on the webhook endpoint as Razorpay will POST without CSRF token [8].

Chatbot Integration (Google Gemini API)

- **Google Generative AI library:** Install `google-generativeai` (`pip install google-generativeai`). Configure the API key from Google Cloud in your Django settings or environment.

- **Backend endpoint:** Create an endpoint (e.g. `POST /api/chatbot/message/`) that takes user message, sends it to Gemini, and returns the AI response. For example, using code from a tutorial [10]:

```python
import google.generativeai as genai
from django.views.decorators.http import require_POST
from django.http import JsonResponse

genai.configure(api_key=settings.GEMINI_API_KEY)
model = genai.GenerativeModel("gemini-pro")

@login_required
@require_POST
def chat_message(request):
    user_text = request.POST.get("text")
    chat = model.start_chat()
    response = chat.send_message(user_text)
    # Optionally save to DB (ChatMessage model)
    return JsonResponse({'response': response.text})
```

  This uses the Gemini model named `"gemini-pro"` and sends a message, returning `response.text` [10].
- **Frontend:** On the Chatbot page, send user input via AJAX/fetch to this endpoint, display the AI's response. Use Redux or React state to store chat history (optionally).
- **Storing chats:** If desired, save each chat exchange in a Django model (e.g. `ChatMessage` with fields `user`, `input_text`, `response_text`, `timestamp`) [11]. The support page can then fetch past conversations for logged-in user.

Email Verification (Link and OTP)

- **Sending emails:** Use Django's email backend or a service (SendGrid, SMTP). On signup, generate:
- A **verification link**: e.g. `https://yourfrontend.com/verify-email?token=<token>`. The token can be a signed value (using `django.core.signing` or a JWT refresh token). When the user visits the link, your frontend can call an API to activate.
- An **OTP code**: Email a numeric code (e.g. 6 digits) stored in a temporary model or cache. Ensure the code expires (e.g. 5 min).
- **Verification endpoints:**
- **Link confirmation (`/api/auth/verify-email/` GET):** Decode token, activate user.
- **OTP confirmation (`/api/auth/verify-otp/` POST):** Accept `{email, otp}`, verify against stored OTP, then activate user. Libraries like Django OTP or custom models can be used.
- **Security:** Use HTTPS in production for email links. Avoid exposing any user data in the token. Once verified, invalidate the token/OTP.

API Security and CORS

- **CORS:** Install `django-cors-headers` and configure it to allow the React app's origin. Example:

```
CORS_ALLOWED_ORIGINS = ["http://localhost:3000"]  # or your production URL
```

- **CSRF:** For non-GET endpoints, Django's CSRF protection applies. API clients should send the CSRF token (for session auth). Since we use JWT, CSRF is not used for token auth. For the payment callback (which is a third-party POST), use `@csrf_exempt` [8].
- **Authentication:** Mark protected views with `@permission_classes([IsAuthenticated])` or use DRF viewsets with `permission_classes`.
- **HTTPS:** Ensure the app uses HTTPS in production to protect token transmission.
- **Secrets management:** Store keys (JWT secret, DB URI, API keys) in environment variables or `django-environ`, not in code.

Best Practices and Tips

- **Modularity:** Keep code DRY. Reuse components (frontend) and functions (backend). Isolate features; e.g. a payment service module, an auth service module.
- **Folder consistency:** Use consistent naming. The feature-based structure is recommended for both React and Redux [1] [2]. Similarly, in Django, group URLs, views, serializers per app.
- **Error handling:** Validate data on both frontend and backend. Use DRF serializers to enforce required fields. Return meaningful error messages.
- **State organization (Redux):** Use the Redux feature-slice pattern [2]. Keep state minimal and normalized if complex (e.g. store product list by ID).
- **Token storage:** For JWT, prefer HTTP-only secure cookies or in-memory state (for web). If using localStorage, ensure to guard against XSS. Handle token refresh smoothly on the frontend.
- **User flows:** Plan user flows clearly. For example, after signup, the UI should inform the user to check email. Handle cases like "resend OTP" or expired token gracefully.
- **Payment security:** On the backend, always verify payment signatures [8] [9]. Do not finalize orders until payment is captured.
- **Testing:** Write tests for key flows (e.g. signup/login, payment, API endpoints) on the backend. Use React Testing Library or Cypress for critical frontend components.
- **Documentation:** Document your API (e.g. with DRF's schema or a README) and state the folder conventions in a dev guide. This helps new developers.
- **Environment parity:** Use `.env` files and `.gitignore` for secrets. Keep development and production settings separate (e.g. DEBUG, CORS).
- **Performance:** For scalability, avoid deep component nesting in React [12]. On the backend, use database indexes (e.g. on email, foreign keys) and limit querysets.
- **Simplicity:** Don't over-engineer. Start with basic implementations (Django ORM models, simple serializers) and refactor when needed.

By following this plan — organizing code clearly, securing APIs, and integrating services properly — you will create a maintainable full-stack application. The tables and snippets above summarize routes, API endpoints, and folder structures to guide implementation. Keep components small and focused, protect your API with proper auth, and test each part thoroughly as you build.

Sources: We followed industry guidance on React project organization and Redux patterns [1] [2], React Router usage [3], Django REST Framework JWT setup [5], Razorpay integration [8] [7] [9], and Google Gemini API usage [10]. These practices ensure a scalable, secure application.

1 12 React Folder Structure in 5 Steps [2025]
https://www.robinwieruch.de/react-folder-structure/

2 Code Structure | Redux
https://redux.js.org/faq/code-structure

3 Your complete guide to routing in React | Hygraph
https://hygraph.com/blog/routing-in-react

4 Connect to your database | Supabase Docs
https://supabase.com/docs/guides/database/connecting-to-postgres

5 6 OTP Verification in Django REST Framework using JWT and Cryptography | GeeksforGeeks
https://www.geeksforgeeks.org/otp-verification-in-django-rest-framework-using-jwt-and-cryptography/

7 9 Integration Steps | Python SDK | Razorpay Docs
https://razorpay.com/docs/payments/server-integration/python/integration-steps/

8 Razorpay Integration in Django | GeeksforGeeks
https://www.geeksforgeeks.org/razorpay-integration-in-django/

10 11 Build your chatbot with the power of Gemini API using Django | by Abdulla Fajal | Medium
https://medium.com/@abdullafajal/build-your-chatbot-with-the-power-of-gemini-api-using-django-29f574fd2ae0