**Tech Stack Overview and Scalability Plan**

To build a scalable client portal that integrates with our existing systems, I'm proposing we leverage a modern tech stack that aligns with our current use of Vite, React, and TypeScript. Below is the proposed tech stack and how each component contributes to scalability and maintainability.

**Frontend**

* **Vite**: A fast build tool that serves our code via native ES modules during development and bundles it for production. It significantly improves development speed compared to traditional bundlers.
* **React**: A robust JavaScript library for building user interfaces. React's component-based architecture promotes reusability and maintainability.
* **TypeScript**: A statically typed superset of JavaScript that catches errors at compile-time, enhancing code quality and developer productivity.
* **State Management**: Utilize **Redux** or React's **Context API** for managing global state, especially useful for handling user authentication status and shared data across components.
* **Routing**: Implement **React Router** for client-side routing to create a single-page application experience.

**Backend**

* **Node.js**: A JavaScript runtime built on Chrome's V8 engine. It allows for building scalable network applications.
* **Express.js**: A minimal and flexible Node.js web application framework that provides a robust set of features for building APIs.
* **TypeScript**: Using TypeScript on the backend ensures type safety and consistency across the full stack.
* **Database**:
  + **PostgreSQL**: A powerful, open-source relational database system. It offers robustness and scalability.
  + **TypeORM**: An ORM that supports TypeScript and allows for seamless interaction with PostgreSQL using entity models.
* **Authentication and Authorization**:
  + **JWT (JSON Web Tokens)**: For stateless authentication, facilitating horizontal scalability.
  + **Passport.js**: Middleware for authentication in Node.js, supporting various strategies including JWT.
* **Integration with HubSpot**:
  + Use HubSpot's APIs and webhooks to synchronize data between our client portal and HubSpot CRM.
  + Implement OAuth 2.0 for secure API access.

**Deployment and Infrastructure**

* **Netlify**: Continue using Netlify for frontend deployment, offering continuous deployment and serverless functions.
* **Backend Hosting**:
  + **Heroku** or **DigitalOcean App Platform**: For ease of deployment and scalability without managing infrastructure.
  + **Docker**: Containerize the backend application for consistent deployment across environments.
* **Continuous Integration/Continuous Deployment (CI/CD)**:
  + **GitHub Actions**: Automate the build, test, and deployment pipeline for both frontend and backend.
* **Scaling Considerations**:
  + Design stateless backend services to allow horizontal scaling.
  + Use load balancers provided by the hosting platform.
  + Implement caching strategies using **Redis** to reduce database load.

**Security Strategy**

* **Data Protection**:
  + **SSL/TLS Encryption**: Ensure all data in transit is encrypted.
  + **At-Rest Encryption**: Use PostgreSQL's encryption features or disk-level encryption provided by the hosting service.
* **Authentication and Authorization**:
  + **Password Hashing**: Use bcrypt to securely hash user passwords.
  + **Role-Based Access Control (RBAC)**: Define user roles (e.g., client, intake agent, QC team) and permissions.
* **Secure Coding Practices**:
  + **Input Validation and Sanitization**: Prevent SQL injection and XSS attacks by validating all inputs.
  + **CORS Configuration**: Properly configure Cross-Origin Resource Sharing to prevent unauthorized access.
* **Compliance**:
  + **HIPAA Compliance**: Since medical records are involved, adhere to HIPAA regulations.
  + **GDPR Compliance**: If handling data from EU citizens, ensure GDPR compliance.
* **Infrastructure Security**:
  + **Firewalls and Security Groups**: Restrict access to databases and servers.
  + **Regular Security Audits**: Perform vulnerability scanning and penetration testing.

**Code Breakdown and Detailed Explanation**

Below is a breakdown of how the code works, structured like a presentation for clarity.

**1. Frontend Architecture**

**Entry Point (main.tsx):**

* Initializes the React application.
* Wraps the App component with necessary providers (e.g., Redux Provider).

typescript

Copy code

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import { Provider } from 'react-redux';

import store from './store';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

**App Component (App.tsx):**

* Sets up routing using React Router.
* Defines global components like navigation bars or footers.

typescript

Copy code

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Login from './components/Login';

import Questionnaire from './components/Questionnaire';

import Dashboard from './components/Dashboard';

const App: React.FC = () => (

<Router>

<Switch>

<Route path="/login" component={Login} />

<Route path="/questionnaire" component={Questionnaire} />

<Route path="/dashboard" component={Dashboard} />

</Switch>

</Router>

);

export default App;

**Login Component (Login.tsx):**

* Renders a form for user login.
* Dispatches an action to authenticate the user.

typescript

Copy code

import React, { useState } from 'react';

import { useDispatch } from 'react-redux';

import { login } from '../actions/authActions';

const Login: React.FC = () => {

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const dispatch = useDispatch();

const handleSubmit = (e: React.FormEvent) => {

e.preventDefault();

dispatch(login({ email, password }));

};

return (

<form onSubmit={handleSubmit}>

{/\* Input fields and submit button \*/}

</form>

);

};

export default Login;

**Questionnaire Component (Questionnaire.tsx):**

* Presents questions to the user.
* Collects responses and sends them to the backend.

typescript

Copy code

import React, { useState } from 'react';

import { useDispatch } from 'react-redux';

import { submitQuestionnaire } from '../actions/questionnaireActions';

const Questionnaire: React.FC = () => {

const [responses, setResponses] = useState({});

const dispatch = useDispatch();

const handleChange = (questionId: string, answer: string) => {

setResponses({ ...responses, [questionId]: answer });

};

const handleSubmit = () => {

dispatch(submitQuestionnaire(responses));

};

return (

<form onSubmit={handleSubmit}>

{/\* Render questions and collect responses \*/}

</form>

);

};

export default Questionnaire;

**State Management with Redux:**

* **Store (store.ts):** Combines reducers and applies middleware.

typescript

Copy code

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;

* **Reducers:** Manage slices of state (e.g., authReducer, questionnaireReducer).
* **Actions:** Define action creators for dispatching actions (e.g., login, submitQuestionnaire).

**2. Backend Architecture**

**Server Setup (server.ts):**

* Initializes Express app.
* Applies middleware for JSON parsing, CORS, and security headers.

typescript

Copy code

import express from 'express';

import cors from 'cors';

import helmet from 'helmet';

import authRoutes from './routes/authRoutes';

import questionnaireRoutes from './routes/questionnaireRoutes';

const app = express();

app.use(cors());

app.use(helmet());

app.use(express.json());

app.use('/api/auth', authRoutes);

app.use('/api/questionnaire', questionnaireRoutes);

// Error handling middleware

app.listen(process.env.PORT, () => {

console.log(`Server running on port ${process.env.PORT}`);

});

**Authentication Routes (authRoutes.ts):**

* **POST /login:** Authenticates the user and returns a JWT.
* **POST /register:** Registers a new user.

typescript

Copy code

import { Router } from 'express';

import { login, register } from '../controllers/authController';

const router = Router();

router.post('/login', login);

router.post('/register', register);

export default router;

**Auth Controller (authController.ts):**

* Handles logic for user authentication and registration.

typescript

Copy code

import { Request, Response } from 'express';

import bcrypt from 'bcrypt';

import jwt from 'jsonwebtoken';

import User from '../models/User';

export const login = async (req: Request, res: Response) => {

// Authenticate user and generate JWT

};

export const register = async (req: Request, res: Response) => {

// Hash password and save user to the database

};

**Questionnaire Routes (questionnaireRoutes.ts):**

* **POST /submit:** Receives questionnaire responses and processes tiering.

typescript

Copy code

import { Router } from 'express';

import { submitQuestionnaire } from '../controllers/questionnaireController';

import { authenticate } from '../middleware/authenticate';

const router = Router();

router.post('/submit', authenticate, submitQuestionnaire);

export default router;

**Questionnaire Controller (questionnaireController.ts):**

* Processes responses and determines the tier.

typescript

Copy code

import { Request, Response } from 'express';

import Questionnaire from '../models/Questionnaire';

export const submitQuestionnaire = async (req: Request, res: Response) => {

// Save responses and determine tier

};

**Middleware (authenticate.ts):**

* Verifies JWT and attaches user to the request object.

typescript

Copy code

import { Request, Response, NextFunction } from 'express';

import jwt from 'jsonwebtoken';

export const authenticate = (req: Request, res: Response, next: NextFunction) => {

// Verify token and handle authentication

};

**Database Models (e.g., User.ts, Questionnaire.ts):**

* Define schemas using TypeORM decorators.

typescript

Copy code

import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()

export class User {

@PrimaryGeneratedColumn()

id: number;

@Column()

email: string;

@Column()

passwordHash: string;

@Column()

role: string;

}

**3. Integration with HubSpot**

**HubSpot Service (hubspotService.ts):**

* Encapsulates API calls to HubSpot.

typescript

Copy code

import axios from 'axios';

export const createContact = async (contactData) => {

// Use HubSpot API to create a new contact

};

export const updateContact = async (contactId, updateData) => {

// Update existing contact information

};

**Usage in Controllers:**

* After user registration or questionnaire submission, synchronize data with HubSpot.

typescript

Copy code

import { createContact } from '../services/hubspotService';

export const register = async (req: Request, res: Response) => {

// Register user

// ...

await createContact({ email: req.body.email, ...otherData });

};

**Strategy for Secure Data Handling**

* **Data Encryption:**
  + **In Transit:** Enforce HTTPS using SSL certificates.
  + **At Rest:** Utilize PostgreSQL's encryption features or encrypt sensitive fields within the application.
* **Access Control:**
  + Implement role-based access controls within the application.
  + Use middleware to enforce permissions on API routes.
* **Password Security:**
  + Use bcrypt with a strong salt factor to hash passwords.
  + Never store plaintext passwords.
* **Environment Variables:**
  + Store secrets like API keys and database credentials in environment variables.
  + Use a package like dotenv to manage them in development.
* **Logging and Monitoring:**
  + Use a logging library like winston for server-side logging.
  + Monitor application performance and errors with tools like Sentry or New Relic.
* **Regular Audits:**
  + Conduct security audits and code reviews.
  + Use automated tools to scan for vulnerabilities (e.g., npm audit, Snyk).
* **Compliance Measures:**
  + Document and enforce policies required by HIPAA and GDPR.
  + Train staff on data handling procedures.

**Project Timeline and Scaling Plan**

* **By End of November:**
  + Complete core functionalities:
    - User authentication and authorization.
    - Questionnaire flow with tiering logic.
    - Integration with HubSpot for initial data synchronization.
  + Deploy the MVP to Netlify (frontend) and chosen backend hosting platform.
* **Post-November Scaling:**
  + Implement intake agent and QC team interfaces.
  + Enhance security measures, including 2FA and detailed audit logs.
  + Optimize performance with caching and load balancing as user base grows.
  + Continuously integrate feedback and iterate on features.

**Leveraging Bolt.new and Cursor AI**

* **Code Generation:**
  + Use Bolt.new to quickly scaffold components, routes, and models.
  + Accelerate development of repetitive boilerplate code.
* **Code Refinement:**
  + Utilize Cursor AI to optimize existing code.
  + Automatically refactor code for better performance and readability.
* **Testing Assistance:**
  + Generate unit and integration tests with AI assistance.
  + Ensure high code coverage and reliability.

**Final Thoughts**

By utilizing this tech stack and following best practices, we can build a robust, scalable, and secure client portal that integrates seamlessly with our existing processes and tools like HubSpot. The use of modern development tools like Bolt.new and Cursor AI will expedite the development process, allowing us to aim for an MVP release by the end of November.

**Next Steps:**

* Finalize the detailed requirements and user stories.
* Set up project repositories and CI/CD pipelines.
* Begin development with a focus on core functionalities.