ChatGPT

تقرير تنفيذي حول مشروع Shoobydo

١. الفجوات والتعارضات بين الكود والمخطط التقني

أظهر تحليل المشروع العميق فجوات واضحة بين ما خطط في الدراسة التقنية وبين ما نفّذ في الشفرة. فقد صُمِّم المشروع أصلاً بنية خدماتية ثلاثية الطبقات (Microservices)، لكن الكود الحالي مطبّق كـ«مستودع موحد» (Monorepo) يجمع الواجهة الأمامية والخلفية في مكان واحد [1] . ففي حين اقترحت وثائق التصميم استخدام FastAPI للـ backend وواجهات حديثة مبنية بـReact/TypeScript مع مكتبات Tailwind CSS وShadcn/UI، فإن الشفرة الحالية تعتمد على Next.js للواجهة دون Tailwind (يُستخدم CSS عادي) [2] . هذا يعني أن نظام التصميم الموحد والهوية البصرية الموحدة المخطط لها لم تُطَبَّق بالكامل، والأصناف الـCSS مثل `.btn` و`.table` مُعرَّفة يدوياً في الواجهة.

من ناحية أخرى، يتكوّن كود الـAPI من مسارات لإدارة الموردين (suppliers)، والعملاء (customers)، والمنتجات (products)، والطلبات (orders)، وغيرها [3] . إلا أن العديد من الوظائف المقرّرة (مثل إدارة المستخدمين والأدوار، وتحليلات المبيعات المتقدمة) لم يتم تنفيذها بعد. وثائق الدراسة تشير إلى جداول متعددة مثل Users وOrders وCustomers وProducts لضمان تكامل بيانات شامل، بينما الشفرة السابقة افتقرت إليها. بالرغم من وجود سكريبت إنشاء للجداول يضمّ حاليًا جداول Suppliers وUsers وProducts وغيرها [4] ، إلا أن معالجة البيانات والوظائف المرتبطة بها لا تزال ناقصة مقارنةً بالمتطلبات الأصلية.

على صعيد التكامل، كانت الدراسة تتوقع مسارًا موحّدًا لواجهة برمجة التطبيقات تحت `/api/v1/...` يشمل إدارة الموردين والمنتجات والطلبات ومنصات خارجية. أما الكود الحالي فليس فيه إلا مجموعة محدودة من المسارات (تقتصر أساساً على الموردين والتقارير) [2] . فمثلًا، واجهة الـ API في الشفرة تتيح الحصول على بيانات الموردين وإعادة فهرستها والبحث عنها، لكنها تفتقر إلى مسارات لإدارة المنتجات عبر المتجر أو ربط بوابات دفع خارجية. كما أن الواجهة الخلفية الحالية لا تتضمّن حتى الآن أي نظام مصادقة أو صلاحيات، بالرغم من توجيه الـ API لمسار `/auth` [3] . هذا التفاوت بين المخطط والتنفيذ يشكّل فجوات كبيرة تعيق استكمال متطلبات الأعمال والخدمات المتوقع تحقيقها.

٢. نقاط الضعف والتناقضات البنيوية

بالإضافة إلى الفجوات السابقة، توجد في الكود الحالي العديد من نقاط الضعف والبنيوية. فغياب نظام توثيق موثوق يجعل واجهات البرمجة الحالية مفتوحة دون أي حماية أو تشفير، مما يعرّض البيانات لهجمات محتملة. التوثيق الأمني المطلوب (مثلاً جلسات JWT أو المصادقة متعددة العوامل) لم يُنفَّذ، ولم تُراعَ مبادئ الـRBAC أو جداول المستخدمين الصارمة [3] . كما أن غياب جداول المستخدمين (Users) أو دور لإدارة الأدوار (Roles) يعني انعدام القدرة على فرض صلاحيات على المسارات الحساسة.

من ناحية هندسية، يتسم الكود الحالي ببعض التناقضات مع المعايير البرمجية. فالمستودع موحد بلا فصل حقيقي للمسؤوليات (lack of separation of concerns)، حيث تحمل جميع مكوّنات الواجهة الأمامية والواجهة الخلفية معاً. هذا التصميم الهيكلي يخرق مبدأ الـModularization ويصعّب الصيانة والتوسعة. كما لا توجد حالياً معايير واضحة لكتابة الأكواد (مثل قواعد تسمية موحدة أو استخدام لقطات نمطية) ولا أطر عمل للاختبارات الوظيفية بخلاف بعض الاختبارات البسيطة للموردين. وبذلك، قد يصعب ضمان جودة الكود ورفعه في بيئات إنتاجية دون أخطاء.

ثانياً، هناك نقص في المعالجة البنيوية للبيانات. فالإكسل أو نماذج البيانات المرفقة تشير إلى جداول مترابطة مع علاقة تطبيع (Normalization) واضحة بين المنتجات والعملاء والطلبات، بينما الشفرة الحالية تفتقر إلى هذه الربطيات. وقد يؤدي هذا إلى تعارضات عند توسعة قاعدة البيانات أو إضافة ميزات جديدة. بالإضافة إلى ذلك، لم يتم تطبيق استراتيجية تخزين مؤقت شاملة (على الرغم من وجود Redis)، ولا توجد آليات لترقية العلاقات أو دعم الضبط

التلقائي للنسخ الاحتياطي. جميع هذه التناقضات البنيوية تضعف مرونة النظام وتبعده عن معايير هندسة البرمجيات المثلى.

٣. قيود على التوسع والتكامل المستقبلي

الميول الحالية تجعل النظام غير قابل للتوسع بشكل فعّال. فالبنية الأحادية Monolith تعني أن زيادة حمل المستخدمين أو إضافة خدمات جديدة يتطلب نشر تطبيق كامل كبير، بدلاً من خدمات مصغرة مستقلة. علاوة على ذلك، لا توجد مسارات منفصلة للتكامل مع خدمات الدفع مثل Stripe أو منصات التجارة مثل Shopify/ WooCommerce في النسخة الحالية. أي توسيع مستقبلي سيستلزم إعادة كتابة جوهرية.

من زاوية واجهة المستخدم، غياب مكتبات الواجهة الحديثة (Tailwind, Shadcn/UI, Framer Motion) يعني أن تحسين تجربة المستخدم سيكون مكلفاً ومرهقاً. فالتصميم الحرفي باستخدام CSS عادي يحدّ من التناسق البصري ويجعل إضافات الحركة أو السمات الجديدة أكثر تعقيداً. باختصار، الأنماط الحالية تتسم بالصعوبة في الصيانة ولا توفر قابلية التخصيص الجذري.

٤. الخطة الهندسية الممنهجة للإصلاح والتطوير

لتجاوز هذه العقبات، نقترح خطة عمل هندسية واضحة مكوّنة من عدة مراحل متتابعة بالأولويات الزمنية.

- **إعادة هيكلة وتنظيم المشروع:** تحويل المستودع الموحد إلى هيكلية معيارية (Modularization) وتنفيذ بنية خدماتية مصغرة Microservices. يُقترح تقسيم الكود إلى خدمات مستقلة (مثل خدمة للمستخدمين، وأخرى لإدارة المنتجات، وثالثة للطلبات) تتيح النشر المستقل والاختبار المنعزل. تستخدم كل خدمة إطار العمل الأنسب (FastAPI أو Node.js مثلاً) وتتواصل عبر REST أو رسائل صفية. هذا سيعالج خلل البنية الحالي ويسمح بإضافة خدمات لاحقة بسهولة [1] [3] .

- **بناء الوحدات الناقصة (modules):** التركيز على تطوير الوحدات التي وجدت الدراسة أنها مفقودة. مثلاً:

- وحدة إدارة المستخدمين والمصادقة (Users & Auth) لدعم JWT، وتسجيل الدخول/الخروج بأمان.
- وحدة إدارة المنتجات (Products Core) كاملة الخصائص (إنشاء، تعديل، حذف، استعلام) [3] .
- وحدة إدارة الطلبات (Orders Core) مع عناصر الطلب (Order Items) وحساب المجموع التلقائي [3] .
- وحدة إدارة المخزون (Inventory & Reservations) لتعقّب المخزون ومنع البيع الزائد حسب المتطلبات.
- وحدة التحليلات (Analytics/KPIs) لحساب مؤشرات الأداء وعرضها في الواجهة.

- وغيرها مثل العملاء (Customers) والعملاء المحتملين وواجهات برمجة لربط الطرف الثالث (مثل بوابات الدفع أو متاجر الويب).
بناء هذه الوحدات يغطي النقصات البنيوية ويوفر الأساس الوظيفي المطلوب.

- **تعزيز الأمان والتوثيق:** تطوير نظام مصادقة وتفويض متكامل. تطبيق JWT للتوثيق بين الخدمات، واستخدام المصادقة متعددة العوامل (MFA) لواجهات الإدارة. إضافة جداول Users وRolesg وفق خطة إدارة الهوية (IAM) لتحديد صلاحيات دقيقة لكل دور. تشفير البيانات الحساسة (مثل كلمات المرور) باستخدام خوارزميات قوية. جميع واجهات الـAPI يجب أن تُحمي بوحدة توثيق (bearer token مثلاً) وتراعي مبادئ least privilege. ستعالج هذه الإجراءات الثغرات الأمنية المعلنة وتلائم المعايير الفنية الصارمة.

- **تحسين الأداء واستراتيجية التخزين المؤقت:** الاستفادة من Redis الموجود حالياً [2] لبناء طبقة Cache فعّالة. مثلاً، يمكن تطبيق ذاكرة مؤقتة للطلبات المتكررة (كالجداول المرجعية غير المتغيرة، أو نتائج استعلامات ثقيلة) لتخفيف العبء على قاعدة البيانات. كما يجب مراقبة زمن استجابة النقاط الحرجة (مثل `/suppliers`, `/orders`) وتحسين الاستعلامات (استعمال فهارس Indexing في PostgreSQL). تنظيم طابور للمهام غير اللحظية (مثل استيراد ملفات Excel وتوليد التقارير) باستخدام Redis أو نظام رسائل (RabbitMQ/Redis Queue) سيحسن سرعة الاستجابة للمستخدم.

- **خطة التكامل مع المنصات الخارجية:** وضع محولات (Adapters) لربط النظام مع منصات التجارة الإلكترونية مثل Shopify وWooCommerce، ومحركات الدفع مثل Stripe. يتطلب ذلك تصميم واجهات API RESTful أو استخدام Webhooks للتواصل. مثلاً، بناء خدمة منفصلة تربط مكتبة Stripe API لمعالجة الدفع، وخدمة أخرى لتزامن المخزون مع Shopify. هذه الطبقة من الاندماج تضمن توسع النظام في الأسواق المختلفة وتلبية متطلبات المشاريع المستقبلية دون إعادة تصميم جوهرية.

- **تحسين الواجهة وتجربة المستخدم:** استبدال الـCSS المبسط بتطبيق إطار تصميم (Design System) موحد، مثل Tailwind CSS مع مكونات Shadcn/UI، واستخدام Framer Motion لإضفاء تفاعلية سلسة. يعمل ذلك على توحيد الهوية البصرية (الألوان والخطوط) وتسريع عملية التطوير مستقبلاً. يُنصح بفصل الواجهة إلى مكونات قابلة لإعادة الاستخدام واتباع أفضل الممارسات في React/TypeScript. هذا سيحوّل الواجهة الحالية إلى منصة جذابة وسهلة التوسع مع الحفاظ على اتساق العلامة التجارية.

- **CI/CD وجودة الكود والتوثيق:** إقامة خط أنابيب تكامل مستمر (مثلاً عبر GitHub Actions أو Jenkins) يقوم بالتنبيهات عند كل دمج (merge) ويفحص الأخطاء أوتوماتيكياً. إضافة فحص الجودة الثابت (linting) مع قواعد لـPEP8 أو Prettier/ESLint وضوابط تغطية الاختبارات. تعزيز التوثيق البرمجي (باستخدام OpenAPI/Swagger لمسارات الـAPI مثلاً) وتدوين دليل تشغيل (README) واضح. تطبيق مراجعات أكواد دورية (Code Reviews) يضمن تقاربا مع المعايير الهندسية ويوثق الكود للمطورين الجدد.

- **الإطلاق التجريبي والمراقبة:** تحضير بيئة staging مشابهة للإنتاج لاختبار الإصدار قبل الإطلاق. تفعيل أدوات مراقبة الأداء (Monitoring) مثل Prometheus/Grafana لمتابعة مؤشرات النظام (CPU، الذاكرة، زمن الاستجابة). وضع مقاييس نجاح (KPIs) واضحة للمنتج التجريبي (مثل عدد المعاملات في الساعة، زمن الصفحة الرئيسية) وتعيين تنبيهات في حال الانحراف. يساهم ذلك في ضمان ثبات الخدمات واستكشاف المشاكل قبل الانتشار الفعلي.

الجدول الزمني والأولويات

المرحلة	المدة المقدرة	الأنشطة الرئيسية	الأولوية
المرحلة ١: **الهيكلة الأساسية**	الشهر 1–2	إعادة هيكلة الكود إلى خدمات مستقلة؛ إنشاء قاعدة بيانات مهيكلة (جدول المستخدمين، العملاء، المنتجات، الطلبات) [4] ؛ بناء خطوط API أولية	عالية
المرحلة ٢: **الأمان والأداء**	الشهر 3–4	تنفيذ نظام مصادقة JWT ومصادقة متعددة العوامل؛ تفعيل Redis للكاش [2] ؛ تحسين الاستعلامات (الفهرسة، الهجرات البرمجية)	عالية
المرحلة ٣: **التكامل والواجهة**	الشهر 5–6	ربط خدمات الدفع (Stripe) والتجارة (Shopify/WooCommerce)؛ إعادة تصميم الواجهة باستخدام Tailwind وShadcn/UI؛ تطوير وحدة تحليلات KPI	متوسطة
المرحلة ٤: **التطوير التجريبي**	الشهر 7	إطلاق بيئة تجريبية (beta)؛ مراقبة الإنتاج (Logging/Metrics)؛ إصلاح العيوب وتحسين الواجهة حسب الملاحظات	عالية

التوصيات التنفيذية

بناءً على ما سبق، نوصي الفريق بالبدء فوراً في إعادة هيكلة المشروع وفق رؤية الخدمات المصغرة، مع تخصيص فرق صغيرة لكل وحدة أساسية (مستخدمين، طلبات، منتجات، إلخ). يجب أن تكون الأولوية القصوى لتأسيس بيئة الأمان والهوية (Authentication/Authorization) وضمان سلامة البيانات. بالتوازي، يُفضَّل استثمار جهود تحسين

الواجهة الأمامية لتعزيز تجربة المستخدم التجارية، خاصة قبل الإطلاق التجريبي. كل خطوة هندسية يجب أن تترافق مع توثيق واضح وفحوصات آلية لضمان الموثوقية والجودة.

المصادر: تم دعم التحليل أعلاه بمراجعة وثائق الكود الحالية للمشروع والشروحات الفنية المرافقة [1] [2] [4] [3] ، مما يضمن دقة الملاحظات المقترحة وخطوات التطوير المستقبلي.

---

[1] [2] README.md
https://github.com/abdulrhmanasami/shoobydo/blob/d1758febb20a3b5432b309aaa6179430e3db8bf3/README.md

[3] main.py
https://github.com/abdulrhmanasami/shoobydo/blob/d1758febb20a3b5432b309aaa6179430e3db8bf3/app/main.py

[4] GitHub
/https://github.com/abdulrhmanasami/shoobydo/blob/d1758febb20a3b5432b309aaa6179430e3db8bf3/apps/backend
create_tables_simple.py