יומן פיתוח - מערכת הערכת רישוי עסקים

יומן זה מתעד את האתגרים המרכזיים שנתקלתי בהם במהלך פיתוח המערכת, ההחלטות שהתקבלו, והפתרונות שיושמו.

ניהול הפרויקט כ-Monorepo עם pnpm workspace

- האתגר:
הפרויקט מחולק לשני חלקים עיקריים: client ו-server. ניהול התלויות והרצת הסקריפטים בשתי סביבות נפרדות היה לא יעיל ונוטה לטעויות. לדוגמה, הייתי צריך להריץ npm install בכל תיקייה בנפרד ולפתוח שני טרמינלים שונים כדי להריץ את השרת והלקוח.
- הפתרון:
אימצתי את גישת ה-Monorepo באמצעות pnpm workspace.
1. **הגדרה ראשונית:** יצרתי קובץ pnpm-workspace.yaml בתיקייה הראשית שהגדיר את התיקיות packages/client ו-packages/server כחבילות ב-workspace.
2. **ניהול תלויות מרכזי:** מעתה, פקודת `pnpm install` אחת מהתיקייה הראשית של הפרויקט מתקינה את כל התלויות עבור כל החבילות (`client` ו-`server`) ביעילות. `pnpm` חוסך מקום בדיסק על ידי שימוש בקישורים סימבוליים ושומר על עקביות בין הגרסאות.
3. **הרצת סקריפטים יעילה:** אימצתי את השימוש בדגל `filter--` של pnpm. פקודות כמו `pnpm --filter server start` אפשרו לי להריץ סקריפטים ספציפיים מכל מקום בפרויקט, ישירות מהתיקייה הראשית. זה ייעל משמעותית את זרימת העבודה וחסך את הצורך לנווט בין תיקיות שונות בטרמינל.

פיתוח מבוסס-חוקים עם cline AI editor plugin

- **האתגר:** שמירה על עקביות וסטנדרטים גבוהים של קוד לאורך כל הפרויקט היא משימה מאתגרת, במיוחד כאשר עובדים מהר. היה צורך להבטיח שכל קומפוננטה חדשה תיכתב באותו מבנה, שהשימוש ב-TypeScript יהיה נכון, ושעקרונות המודולריות יישמרו. ביצוע ידני של כל הבדיקות הללו מאט את הפיתוח ופותח פתח לטעויות אנוש.
- **הפתרון:** הפרויקט פותח באמצעות עזרה של **cline**, תוסף פיתוח מבוסס AI. השתמשתי באדיטור windsurf בשביל ההשלמות האוטומטיות והמהירות, וב-cline כדי לכתוב קבצים ולערוך אותם מהר. היתרון המרכזי היה היכולת להגדיר "חוקים" והנחיות מותאמות אישית שה-AI פעל לפיהם באופן עקבי.
1. **הגדרת חוקים מותאמים אישית:** בתחילת התהליך, הגדרתי ל-AI סט חוקים ברור, כגון: "כתוב תמיד קומפוננטות React כפונקציות חץ עם type בשם Props", "השתמש תמיד ב-TypeScript", "עצב עם TailwindCSS", ו-"הפרד כל קומפוננטה לקובץ נפרד".
2. **יצירת קוד ושכתוב (Refactoring) מונחה-חוקים:** חוקים אלו אפשרו לי לבקש מה-AI לייצר קוד שלד, לשכתב קוד קיים, וליצור קומפוננטות חדשות שתאמו באופן אוטומטי לארכיטקטורה שהוגדרה. הדבר היה יעיל במיוחד בשלב הפירוק הראשוני של האפליקציה לקומפוננטות מודולריות.
3. **סביבת עבודה אחודה:** מעבר לכתיבת קוד, השתמשתי ב-cline כדי לייצר ולערוך את כל מסמכי התיעוד של הפרויקט, כולל קובץ ה-README ויומן פיתוח זה. הדבר יצר סביבה אחידה שבה אותו AI סייע בכל היבטי המשימה, מה שהבטיח עקביות בסגנון

ובאיכות.
- **התוצאה:** השימוש ב-cline עם מערכת חוקים מוגדרת האיץ משמעותית את זמן הפיתוח. הוא הפחית את העומס הקוגניטיבי הכרוך בבדיקה מתמדת של סטנדרטים, ואפשר לי להתרכז בלוגיקה העסקית המרכזית.

המרת מסמך הרגולציות לבסיס נתונים חכם (JSON עם Embeddings)

כחלק מדרישות הפרויקט, נדרש להמיר את קובץ הרגולציות לקובץ מיפוי וקטורי, הפכתי את מסמך הרגולציות לקובץ JSON חכם שמכיל "מיפוי וקטורי" (Embeddings) לכל סעיף. מיפוי וקטורי הוא בעצם רשימה של מספרים שמייצגת את המשמעות הסמנטית של הטקסט.

- **תהליך ההמרה:**
  1. **קריאת ה-PDF:** כתבתי סקריפט שקורא את קובץ ה-PDF ומפרק אותו לחלקים קטנים (פסקאות וסעיפים).
  2. **יצירת וקטור לכל חלק:** עבור כל חלק טקסט, הסקריפט שלח בקשה למודל שפה (Embedding Model) וקיבל בחזרה את הייצוג הווקטורי שלו (לדוגמה, רשימה של 768 מספרים).
  3. **שמירה ב-JSON:** הסקריפט יצר קובץ JSON. כל אובייקט בקובץ הכיל את הטקסט המקורי של הסעיף ואת הווקטור המספרי שמתאר את המשמעות שלו.
- **התוצאה:** בסוף התהליך, במקום קובץ PDF פשוט, היה לי קובץ JSON שמתפקד כבסיס נתונים סמנטי. קובץ זה אפשר למערכת לבצע חיפושים חכמים המבוססים על קרבה במשמעות, מה שהיווה קפיצת מדרגה משמעותית בדיוק וברלוונטיות של התשובות שסופקו למשתמש.

איזון בין פשטות למקצועיות בפלט ה-AI

- **האתגר:** הגרסאות הראשונות של הדוחות שנוצרו על ידי ה-AI היו "יצירתיות" מדי. למרות שהן היו מובנות, הן כללו שפה ציורית מדי, שימוש מופרז באימוג'ים (למשל, ⚠️🔥🧯), וטון כללי שהרגיש לא מקצועי עבור מסמך ייעוץ עסקי. המטרה הייתה שהדוח יהיה פשוט להבנה, אך עדיין יישמע סמכותי ואמין.
- **הפתרון:** הפתרון דרש שילוב של הנדסת פרומפטים וכוונון פרמטרים של ה-API.
  1. **חידוד הטון בפרומפט:** הוספתי הנחיות מפורשות לפרומפט לגבי סגנון הכתיבה הרצוי: `Maintain a professional, clear, and business-oriented tone. Avoid emojis, exclamation marks, and overly casual language. The goal is to be helpful and authoritative, not playful.`
  2. **כוונון הטמפרטורה (Temperature Tuning):** הורדתי את ערך פרמטר ה-`temperature` בקריאה ל-API לערך נמוך יותר (לדוגמה, `0.2`). פרמטר זה שולט במידת ה"אקראיות" או ה"יצירתיות" של המודל. ערך גבוה (כמו 0.9) מעודד תשובות מגוונות ומפתיעות, בעוד ערך נמוך גורם למודל להיות יותר דטרמיניסטי, ממוקד ועקבי,

ולהיצמד לסגנון ולתוכן המבוקש.
3. **תוצאה:** השילוב של שני השינויים הללו יצר פלט מאוזן - דוחות שהם עדיין קלים
4. לקריאה ומחולקים היטב, אך שומרים על טון מקצועי ורציני.

בחירת פתרון לניהול מצב גלובלי (State Management)

- **האתגר:** לאחר פירוק האפליקציה לקומפוננטות רבות, ניהול המצב הגלובלי הפך למאתגר. העברת props דרך מספר רב של רמות (תופעה המכונה "prop drilling") הפכה את הקוד למסורבל ונוטה לטעויות. היה צורך בפתרון מרכזי שיאפשר לקומפוננטות שונות לגשת למצב המשותף (כמו תוכן הדוח או מצב הטעינה) בלי לסבך את מבנה הקוד.
- **הפתרון:** בחרתי להשתמש בספריית **Zustand**. הסיבות לבחירה היו:
1. **פשטות ומינימליזם:** Zustand מציע API פשוט ואינטואיטיבי ללא כמות גדולה של קוד תבניתי (boilerplate), מה שהופך את הקמתו ושימושו למהירים מאוד.
2. **ביצועים:** הספרייה מונעת רינדורים מחדש מיותרים על ידי כך שהיא מאפשרת לקומפוננטות "להירשם" רק לחלקים ספציפיים מהמצב שהן צריכות, בניגוד ל-Context API שיכול לגרום לרינדור מחדש של כל עץ הקומפוננטות.
3. **קלות האינטגרציה:** שילוב הספרייה בפרויקט היה מיידי. הקמתי "store" מרכזי שניהל את מצב הדוח, מצב הטעינה ושגיאות, וכל קומפוננטה יכלה לגשת למידע זה ישירות באמצעות hook פשוט.
- **התוצאה:** השימוש ב-Zustand פישט משמעותית את לוגיקת ניהול המצב, ניקה את הקוד מ-prop drilling, ושיפר את ארגון וביצועי הקוד הכללי בצד הלקוח.

ארכיטקטורת הבקאנד

- **האתגר:** הלוגיקה בצד השרת דורשת תזמור (Orchestration) של מספר פעולות א-סינכרוניות מורכבות: קבלת בקשה, המרתה לווקטור, חיפוש בבסיס הנתונים הווקטורי, ולבסוף קריאה למודל שפה גדול. היה חשוב לבנות מבנה קוד נקי ומודולרי שיפריד בין האחריויות השונות ויאפשר תחזוקה והרחבה קלה.
- **הפתרון:** אימצתי תבנית עיצוב של **Controller-Service**. קובץ ה-`compliance.controller.ts` משמש כשכבת הבקרה הראשית ("התזמורת"), בעוד שהלוגיקה המורכבת מופרדת לסרוויסים ייעודיים (`vector.service`, `llm.service`).

זרימת העבודה בתוך הקונטרולר (`checkCompliance`):

1. **קבלת ועיבוד הקלט:** הקונטרולר מקבל את גוף הבקשה מהלקוח ומפרמט אותו למחרוזת טקסט (שאילתה) קריאה וברורה.
2. **יצירת Embedding מקומי:** כאן נכנסת לתמונה ספריית `@xenova/transformers`. במקום לשלוח בקשת רשת נוספת לשירות חיצוני, הספרייה מאפשרת לטעון מודל Embedding (כמו `all-MiniLM-L6-v2`) ישירות לזיכרון השרת ולהריץ אותו מקומית. זה חוסך זמן ומורכבות רשת. הקונטרולר משתמש בפונקציית ה-`pipeline` של הספרייה כדי להפוך את שאילתת המשתמש לווקטור מספרי.
3. **אחזור מידע רלוונטי:** הקונטרולר מעביר את הווקטור שנוצר לפונקציית העזר `findRelevantChunks` מתוך `vector.service`. פונקציה זו מבצעת את החיפוש הסמנטי בבסיס הנתונים הווקטורי ומחזירה את סעיפי הרגולציה הרלוונטיים ביותר.
4. **יצירת דוח חכם:** עם המידע הרלוונטי ביד, הקונטרולר קורא לפונקציית העזר

`generateComplianceReport` מתוך `llm.service`. הוא מעביר לה את נתוני העסק המקוריים ואת סעיפי הרגולציה (ההקשר). פונקציה זו אחראית על בניית הפרומפט המורכב ושליחתו למודל השפה החיצוני.
5. **החזרת תשובה:** הקונטרולר מקבל את הדוח הסופי מה-`llm.service` ושולח אותו בחזרה ללקוח כתשובת JSON.

- **התוצאה:** ארכיטקטורה זו יצרה קוד נקי וניתן לבדיקה. הקונטרולר נשאר רזה ואחראי רק על תזמור התהליך, בעוד שהלוגיקה הכבדה של חישובים וקריאות לרשת מבודדת בסרוויסים ייעודיים. השימוש בספריית Xenova לייצור Embeddings מקומי הפך את התהליך ליעיל ומהיר יותר.