BÄRNHÄCKT

«RiskGame»
«Risk-Rallye»

Technische Informationen für die Jury

Technische Informationen für die Jury

Aktueller Stand des Sourcecodes

- https://github.com/RomanRiesen/BernHackt

Ausgangslage

- Worauf habt ihr euch fokussiert?
  - Einen kurzen Fragebogen, um die Risikobereitschaft einer Person interaktiv und etwas 'spielerisch' abzufragen und einzuschätzen.
  - Eine Versicherung(-sstrategie) anhand der Fragebogen vorzuschlagen.
  - Die Entscheidung oder vorgeschlagene Versicherung spielerisch auf deren individuellen Verträglichkeit zu prüfen. Die Über- und Unterversicherten Events durchzuspielen, auch anhand des Fragebogens, und deren Wirksamkeit und finanziellen Impact zu testen.
- Welche technischen Grundsatzentscheide habt ihr gefällt?
  - Keine Standortdaten oder Adressen zu verwerten/arbeiten.
  - Am Anfang einen interactive story-driven Fragebogen auszufüllen, welcher nachher ein damit einhergehendes Risiko-Profil erstellt und eine dazu passende Versicherungspolice vorschlägt.
  - Die Auswirken und Konsequenzen, sowohl bei einer Über- als auch einer Unterversicherung transparent und spielerisch darzustellen.
  - Nur ein interaktives Frontend zu gestalten, ohne Backend.

Technischer Aufbau

- Welche Komponenten und Frameworks habt ihr verwendet? Wozu und wie werden diese eingesetzt?
  - Bei der Entwicklung einer Single Page Application (SPA) mit **Vue.js** und **Vite** spielen verschiedene technische Aspekte und Frameworks eine Rolle, die das Projekt effizienter und skalierbarer machen.
    Komponentenbasierte Architektur: Vue.js ermöglicht es, die Anwendung in wiederverwendbare und isolierte Komponenten zu unterteilen. Diese Komponenten enthalten die Logik (JavaScript), das Template (HTML) und die Styles (CSS) in einer kompakten und organisierten Struktur.
  - **Reactivity System**: Vue verwendet ein reaktives Datenbindungssystem, das Änderungen im Datenmodell automatisch in der Benutzeroberfläche widerspiegelt. Dies reduziert die Notwendigkeit, manuell DOM-Updates zu verwalten, und sorgt für eine flüssigere Benutzererfahrung.
  - **Vue Router**: Für eine SPA ist das Routing entscheidend. Vue Router ermöglicht die Navigation zwischen verschiedenen „Seiten" innerhalb der Anwendung, ohne die gesamte Seite neu zu laden. Es unterstützt auch dynamische Routen und Lazy Loading für eine bessere Leistung.
  - **Blitzschnelle Entwicklungsserver**: Vite startet einen extrem schnellen Entwicklungsserver. Anstatt die gesamte Anwendung bei jeder Änderung zu bündeln, serviert Vite native ES Modules direkt im Browser und aktualisiert nur das, was sich geändert hat. Das Ergebnis ist eine extrem kurze Wartezeit bei Code-Änderungen.
  - **HMR (Hot Module Replacement)**: Vite bietet Hot Module Replacement, wodurch Teile der Anwendung im laufenden Betrieb aktualisiert werden können, ohne dass die Seite neu geladen werden muss. Dies beschleunigt den Entwicklungsprozess erheblich.
  - **Optimiertes Build-System**: Beim Produktions-Build nutzt Vite Rollup unter der Haube, was eine hocheffiziente und bausteinbasierte Bündelung ermöglicht. Das Resultat sind kleinere und besser optimierte Dateien für die Auslieferung.
  - **Plugins und Erweiterungen**: Vite unterstützt eine Vielzahl von Plugins, einschließlich jener, die speziell für Vue.js optimiert sind. Zum Beispiel das „vite-plugin-vue" für die nahtlose Integration von Vue-Komponenten und das „vite-plugin-legacy" für Kompatibilität mit älteren Browsern.

- **Code Splitting und Lazy Loading**: Vite und Vue.js unterstützen Code-Splitting von Natur aus. Dynamische Importe ermöglichen es, JavaScript-Dateien nur dann zu laden, wenn sie benötigt werden, was die anfängliche Ladezeit der Anwendung erheblich reduziert.
- **Cypress**: Für End-to-End-Tests (E2E) ist Cypress eine leistungsstarke Lösung, die den gesamten Benutzerfluss testet. Es ist besonders nützlich, um sicherzustellen, dass die Anwendung als Ganzes wie erwartet funktioniert.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Automatisierte Build- und Deployment-Pipelines sind essenziell für eine schnelle und sichere Bereitstellung von Updates. Tools wie GitHub Actions oder GitLab CI können verwendet werden, um den Code nach jedem Push automatisch zu testen und in die Produktionsumgebung zu deployen.
- **Docker**: Containerisierung mit Docker stellt sicher, dass die Anwendung in einer konsistenten Umgebung läuft, unabhängig davon, wo sie bereitgestellt wird. Dies erhöht die Zuverlässigkeit und verringert die Risiken von Umgebungsfehlern.
- **Tree Shaking**: Vite unterstützt Tree Shaking, was bedeutet, dass ungenutzter Code während des Bundlings entfernt wird, um die endgültige Bundle-Größe zu minimieren.
- **Lazy Loading von Bildern und Komponenten**: Um die anfängliche Ladezeit weiter zu optimieren, können Bilder und Komponenten bei Bedarf nachgeladen werden, anstatt alles sofort zu laden.
- **Git**: Um den Code zu teilen und Inkonsistenzen vorzubeugen. Kollaborative zeitgleiche Zusammenarbeit und gegenseitiges bereitstellen des Codes.

Implementation

- Gibt es etwas Spezielles, was ihr zur Implementation erwähnen wollt?
  - Ein Fragebogen und ein Mini-Game
- Was ist aus technischer Sicht besonders cool an eurer Lösung?
  - Alles ist im Browser, keine Daten werden weitergegeben oder verkauft.
  - Vite erlaubt instant hot-reloading der Changes: Vite bietet Hot Module Replacement, wodurch Teile der Anwendung im laufenden Betrieb aktualisiert werden können, ohne dass die Seite neu geladen werden muss. Dies beschleunigt den Entwicklungsprozess erheblich.
  - SPL, easy deployment
  - **Integration**: Devcontainer-Konfigurationen werden typischerweise in einer .devcontainer-Ordnerstruktur abgelegt und beinhalten Dateien wie devcontainer.json, in denen Container-Spezifikationen und benötigte Entwicklungswerkzeuge definiert werden. Sie können leicht in IDEs wie Visual Studio Code integriert werden, die direkt die Container-basierte Umgebung laden können.
    CI/CD und Skalierbarkeit: Docker passt hervorragend in Continuous Integration/Continuous Deployment (CI/CD)-Pipelines. Durch die automatische Erstellung und das Testen von Container-Images kann der Deployment-Prozess erheblich beschleunigt und die Fehleranfälligkeit reduziert werden. Zudem lässt sich die Anwendung mit Docker leicht skalieren, indem Container repliziert und orchestriert werden (z.B. mit Kubernetes).

Abgrenzung / Offene Punkte

- Welche Abgrenzungen habt ihr bewusst vorgenommen und damit nicht implementiert? Weshalb?
  - Wir haben bewusst nicht alle möglichen Versicherungen in der Schweiz in allen Kantonen mit allen möglichen Versicherungsauschlüssen vorgenommen, sondern uns vor allem auf den «Auftraggeber» GVB konzentriert, um die Fragen übersichtlich und nicht eine riesige Anzahl von Versicherern über mehrere kantonale Regelungen, Wirkungsgebiete und gesetzlichen Vorschriften miteinander zu vergleichen.