بسم الله الرحمن الرحیم

مستند اسکرام

Scrum Document

تاریخ: ۱۴۰۳/۰۹/۲۱

معرفی کلی پروژه

نام پروژه: یکی کمه

فناوری‌های استفاده شده به شرح زیر است:

Frontend: React.js

Backend: Django

Database: PostgreSQL

API: GraphQL

Scalability: RabbitMQ

Use Docker

GitHub Organization

معماری پروژه ماژولار به صورت یکپارچه یا *Modular Monolithic* است.

تعداد اسپرینت‌های پروژه: ۶

مدیریت پروژه با استفاده از *YouTrack* است.

توضیح کار نرم افزار

نرم افزار «یکی کمه» یک نرم افزار تحت وب برای ایجاد، عضویت و مدیریت رویدادهای اجتماعی است. مثلا برای یک مسابقه فوتبال آخر هفته، یک مسافرت آفرود، یک رویداد خیریه، یک رویداد تحصیلی یا تحقیقاتی محور و... کاربر وارد سایت شده و می‌تواند رویدادها را ایجاد، مدیریت و ... کند. کاربر با استفاده از این نرم افزار می‌تواند افراد دیگری که علاقه‌مند به شرکت در چنین رویدادهایی هستند را پیدا کند تا این رویداد با افراد بیشتری انجام شده و در نهایت به تعاملات اجتماعی کمک شایانی کند.

ایده این نرم افزار الهام گرفته از نرم افزار مشابه خارجی به نام *TeamSnap* است که کاربرد آن در ایجاد، عضویت و مدیریت در رویدادهای صرفا ورزشی است ولی در نرم افزار «یکی کمه» دسته‌بندی‌های بسیار جامع‌تری درنظر گرفته شده است.

توضیحاتی راجب معماری ماژولار یکپارچه

معماری *Modular Monolithic* یک یک رویکرد در طراحی و ساخت نرم‌افزار است که ترکیبی از دو مفهوم اصلی "ماژولار (Modular)" و "یکپارچه (*Monolithic*)" را در بر می‌گیرد. این معماری بیشتر در زمینه توسعه نرم‌افزار و سیستم‌های پیچیده مورد استفاده قرار می‌گیرد، به ویژه زمانی که قصد داریم از مزایای هر دو مدل ماژولار و یکپارچه بهره‌برداری کنیم.

در ادامه معماری ماژولار و معماری یکپارچه را توضیح می‌دهیم:

۱) معماری ماژولار (*Modular Architecture*):

در این معماری، سیستم به واحدهای کوچک و مستقل به نام ماژول تقسیم می‌شود.

هر ماژول عملکرد خاص خود را دارد و می‌تواند به طور جداگانه توسعه داده، آزمایش شده و نگهداری شود. این ماژول‌ها معمولاً به گونه‌ای طراحی می‌شوند که به راحتی با سایر ماژول‌ها تعامل کنند.

۲) معماری یکپارچه (*Monolithic*):

در معماری یکپارچه، تمامی اجزای سیستم در یک برنامه واحد و یکپارچه (Monolithic) پیاده‌سازی می‌شود. در این مدل، همه‌ی بخش‌ها در یک پایگاه کد قرار دارند و نمی‌توانند به صورت مستقل از یکدیگر توسعه یا تغییر کنند.

در ادامه به بررسی ترکیب این دو معماری با هم می‌پردازیم

در معماری *Modular Monolithic*، شما می‌توانید مزایای هر دو دنیا را داشته باشید. این معماری اجازه می‌دهد که اجزای مختلف سیستم به صورت ماژولار طراحی و پیاده‌سازی شوند، اما همچنان همه‌ی ماژول‌ها در یک سیستم مونوئیتیک واحد قرار دارند و در کنار یکدیگر عمل می‌کنند.

ویژگی‌های این معماری به شرح زیر است:

۱) ماژول‌های مستقل: هر بخش از سیستم به صورت ماژولار طراحی شده است، به این معنی که می‌توان هر ماژول را به طور مستقل تغییر داد یا بهبود بخشید.

۲) یکپارچگی سیستم: برخلاف سیستم‌های میکروسرویس، تمامی ماژول‌ها در یک برنامه واحد قرار دارند. بنابراین، نیازی به مدیریت پیچیدگی‌های متعدد و نقاط تعامل بین سرویس‌ها نیست.

۳) توسعه یکپارچه: برخلاف معماری‌های کاملاً ماژولار یا یکپارچه، توسعه‌دهندگان می‌توانند کد را به صورت ماژولار بنویسند، اما همچنان از دیدگاه یکپارچگی سیستم، آن را در یک محیط مشترک پیاده‌سازی کنند.

اما سوال اساسی که وجود دارد این است که «چرا در این پروژه از معماری ماژولار یکپارچه استفاده کردیم؟»

برای پاسخ به این سوال جمله معروف مارتین فالر رو با هم مرور می‌کنیم:

«شما نباید یک پروژه جدید را با میکروسرویس‌ها شروع کنید، حتی اگر مطمئن باشید که اپلیکیشن شما به اندازه کافی بزرگ خواهد بود که استفاده از آن‌ها ارزش داشته باشد.»

این جمله به خوبی این نکته را بیان می‌کند که حتی اگر بدانیم این پروژه در آینده باید به معماری *Micro Service* مهاجرت کند در ابتدا نیازی که توسعه آن هم به صورت میکروسرویس باشد.

معمولا در همان ابتدا که مقیاس پروژه پایین است و پروژه *Feature*های زیادی ندارد بهتر است از معماری سرویس‌گرا یا میکروسرویس استفاده نشود و از معماری جایگزینی مثل *Modular Monolithic* استفاده شود.

در این پروژه نیز چون مقیاس پروژه بالا نبوده و بسیاری از Featureهای این پروژه نیز به طور کلی قابلیت *Reusable* ندارد از معماری سرویس‌گرا یا *Micro Service* استفاده نشده است.

Architectural Characteristic	Star Rating
Maintainability	★ ★ ★
Testability	★ ★ ★
Deployability	★ ★ ★
Simplicity	★ ★ ★ ★
Evolvability	★ ★ ★
Performance	★ ★ ★
Scalability	★
Elasticity	★
Fault Tolerance	★
Overall Cost	$ $

These fare better than in the layered architectural style.

Most monolithic architectures perform well, especially if well designed.

Overall, more expensive than layered architectures. Modular monoliths require more planning, thought, and long-term maintainance.

شاخص‌های ویژگی این معماری به شرح بالا هستند. همانطور که می‌بینید این معماری قابلیت بسیار مناسبی در *Maintainability*, *Deployability*, *Testability* دارد و معماری بسیار قابل درک و مناسبی است و پیچیدگی کمتری نسبت به معماری سرویس‌گرا و میکروسرویس دارد.

همچنین این معماری از نظر عملکرد نیز بسیار قابل قبول و بهینه است.

اما همانطور که مشاهده می‌کنید در حالت پایه این معماری قابلیت *Scalability, Elasticity and Fault Tolerance* خیلی بالایی ندارد. (هر چند که جدول بالا بر اساس مقایسه این معماری با میکروسرویس است که حالت ایده‌آل شاخص‌های بالا را داراست)

برای بهبود این شاخص‌ها در این پروژه از *Docker* برای فرآیند *CI/CD* انجام شده است و همچنین *Logging* به طور مناسب در *Docker* پیاده‌سازی شده است که تا حد زیادی *Fault Tolerance* بهبود یافته است.

همچنین برای بهتر شدن *Scalability* با استفاده از *RabbitMQ* بخش‌هایی از بک‌اند پروژه که ممکن است بر اثر Stress Test درخواست زیادی داشته باشند به صورت *Automatic* مقیاس پذیر شده‌اند.

توجه شود که این معماری قطعا مانند معماری میکروسرویس از قابلیت‌های بالا به صورت کاملا ایده‌آل بهره نبرده و قطعا دارای یکسری معایبی است که این معایب در معماری میکروسرویس به طور کامل برطرف خواهند شد.

توضیحاتی راجب به GraphQL

در این پروژه به جای استفاده از *REST API* تصمیم گرفته شد که از *GraphQL* استفاده شود.

برای دریافت داده‌های مورد نیاز باید چندین درخواست *REST API* ارسال میشد برای همین *GraphQL* اختراع شد.

GraphQL یک *syntax* است که نحوه ارسال درخواست دقیق داده‌ها را توصیف می‌کند.

پیاده‌سازی *GraphQL* برای مدل داده‌ای یک برنامه با موجودیت‌های پیچیده که به یکدیگر ارجاع می‌دهند، ارزشمند است.

GraphQL با ساخت یک *schema* شروع می‌شود، که توصیفی از تمام درخواست‌هایی است که می‌توان در یک *GraphQL API* انجام داد و تمام انواع داده‌هایی که این درخواست‌ها باز می‌گردانند.

ساخت *schema* دشوار است زیرا نیاز به *SDL* (*Schema Definition Language*) دارد.

با داشتن *schema* قبل از ارسال درخواست، کلاینت می‌تواند درخواست خود را اعتبارسنجی کرده و اطمینان حاصل کند که سرور قادر به پاسخ‌دهی به آن خواهد بود.

پس از رسیدن به برنامه بک‌اند، یک عملیات *GraphQL* در برابر کل *schema* تفسیر شده و با داده‌ها برای برنامه فرانت‌اند حل می‌شود.

با ارسال یک *query* بزرگ به سرور، یک پاسخ *JSON* باز می‌گرداند که دقیقاً مطابق با شکل داده‌ای است که درخواست کرده‌ایم.

مزایا استفاده از *GraphQL* به شرح زیر است:

۱) *Typed Schema*

GraphQL از پیش آنچه که می‌تواند انجام دهد را منتشر می‌کند که باعث بهبود کشف‌پذیری (*Discoverability*) آن می‌شود.

۲) تناسب با داده‌های گرافی

داده‌هایی که به روابط پیچیده متصل هستند، بسیار خوب با *GraphQL* تطابق دارند، اما برای داده‌های صاف (*flat*) مناسب نیست.

۳) *No Versioning*

بهترین روش در نسخه‌بندی، عدم نسخه‌بندی *GraphQL API* به طور کامل است.

۴) پیام‌های خطای دقیق

مشابه *SOAP* جزئیات دقیقی از خطاهایی که رخ داده را ارائه می‌دهد.

پیام خطای *GraphQL* شامل تمام *resolver*ها و اشاره به بخش دقیق درخواست اشتباه است.

۵) مجوزهای انعطاف‌پذیر

GraphQL اجازه می‌دهد که توابع خاصی به‌طور انتخابی منتشر شوند در حالی که اطلاعات خصوصی حفظ می‌شود.

این درحالی است که معماری *REST* داده‌ها را به صورت کلی نشان می‌دهد، به این صورت که یا همه داده‌ها یا هیچ داده‌ای نیست!

نکته) سرعت و *Performance* این *API* بسیار به طراحی *Database* وابسته است.

معایب استفاده از *GraphQL* به شرح زیر است:

۱) *Performance issues*

GraphQL پیچیدگی را به نفع قدرت خود مبادله می‌کند.

داشتن فیلدهای تو در تو زیاد در یک درخواست می‌تواند منجر به سربار زیاد روی سیستم شود.

۲) پیچیدگی *Caching*

از آنجا که *GraphQL* ز مفاهیم کشینگ *HTTP* استفاده نمی‌کند، نیاز به تلاش سفارشی برای کشینگ دارد.

۳) نیاز به آموزش پیش از توسعه زیادی دارد

بسیاری از پروژه‌ها به دلیل کمبود زمان برای فهمیدن عملیات‌های خاص *GraphQL* و *SDL* تصمیم می‌گیرند همان *REST* بمانند.

توضیحاتی راجب فیچرهای پروژه

مجموعه کارهایی که کاربر در نرم افزار باید بتواند انجام دهد شامل موارد زیر است:

- امکان ثبت‌نام و ورود کاربر با استفاده از شماره موبایل و تائید شماره موبایل با استفاده از کد *Otp* ساختگی
- امکان دید رویدادها با استفاده از فیلتر کردن شهرها
- امکان فیلتر کردن با موارد مختلف رویدادها، دیدن جزئیات رویدادها، درخواست عضویت و عضویت در یک رویداد
- امکان ایجاد یک رویداد
- کاربر باید امکان دیدن رویدادهای ایجاد کرده، رویدادهایی که در آن عضویت دارد و نقش عادی یا مالک (سازنده) دارد را داشته باشد.
- دارای یک پنل مدیریتی برای ادمین‌های اصلی سایت
- دارای پنل کاربری مناسب برای ایجاد، مشاهده رویدادهای عضو شده و ویرایش اطلاعات کاربری
- نرم افزار به طور کلی باید دارای قابلیت‌های *Scalability*, *Logging and Reliable* باشد.

توضیحاتی راجب بک‌لاگ هر اسپرینت

مجموعه کارهایی که در هر اسپرینت باید انجام شود به صورت کلی به شرح زیر است:

اسپرینت اول

انتظار داریم که تا پایان اسپرینت اول کاربر بتواند وارد صفحه اصلی شده و بتواند با استفاده از شهرها فیلترهایی را روی رویدادها بر اساس شهر انجام دهد و سپس به صفحه لیست رویدادها (بر اساس فیلتری که روی شهرها اعمال شده) برود.

در این اسپرینت انتظار نمی‌رود که صفحه لیست رویدادها پیاده سازی شود.

کاربر باید بتواند در سایت با استفاده از شماره موبایل خود ثبت نام کند و سپس برای کاربر کد *Otp* ارسال شود تا شماره موبایلش احراز هویت گردد و همچنین کاربر به دو صورت به سایت لاگین کند. حالت اول با استفاده از شماره موبایل و رمز عبور است و حالت دوم با استفاده از شماره موبایل و ارسال کد یکبار مصرف یا همان *Otp* است.

کاربر پس از ورود به سایت باید به صفحه پنل کاربری هدایت شود.

لازم به ذکر است که در این اسپرینت قرار نیست که صفحه پنل کاربری پیاده سازی شود.

در این اسپرینت همچنین صفحه *Admin Panel* پیش فرض جنگو پیاده سازی و شخصی سازی گردد.

اسپرینت دوم

در این اسپرینت انتظار داریم که کاربر بتواند با استفاده از پنل کاربری خود یک رویداد را ایجاد کند.

در این اسپرینت باید صفحه لیست رویدادها پیاده سازی گردد و کاربر بتواند یکسری فیلترها روی رویدادها انجام دهد.

کاربر پس از کلیک بر روی یک رویداد باید وارد صفحه جزئیات رویداد شود.

لازم به ذکر است که صفحه جزئیات یک رویداد در این اسپرینت پیاده سازی نمی‌شود.

بخش سوالات متداول به صفحه اصلی اضافه شده و جدول و منطق آن در سمت پنل مالکین سایت پیاده سازی شود.

اسپرینت سوم

در این اسپرینت باید صفحه جزئیات رویداد پیاده سازی شود. این صفحه باید شامل همه جزئیات رویداد شود. کاربر اگر ثبت نام و لاگین شده بود بتواند با زدن روی دکمه عضویت به صفحه شرکت در رویداد برود. در این صفحه کاربر اگر نیاز است باید اطلاعاتی را پر کند و درخواست عضویت خود را ارسال کند.

اگر کاربر لاگین و ثبت نام نکرده بود ابتدا باید ثبت نام یا لاگین کند و سپس به صفحه جزئیات رویداد برگردد.

اسپرینت چهارم

کاربر پس از تکمیل مرحله عضویت در یک رویداد باید صبر کند تا مالک رویداد درخواست عضویت آن را بررسی کند و سپس در رویداد ثبت خواهد شد.

مالک رویداد درخواست عضویت کاربران را بررسی کرده و آن‌ها را تائید یا رد می‌کند.

در این اسپرینت همچنین باید قابلیت‌های Logging و Scalability نیز پیاده سازی گردد.

برای قابلیت Logging داکرایز آن نیز انجام شود اما برای Scalability صرفا بک‌اند پروژه آماده سازی گردد.

اسپرینت پنجم

در ایـن اسـپرینت کـاربر بایـد بتوانـد لیسـتی از رویـدادهایی کـه مالـک آن و یـا عضـو عـادی آن اسـت را ببینـد و اطلاعـات کـاربری خـود را در پنـل کاربری خود بتواند مشاهده، تکمیل و تغییرات دهد.

همچنین در این اسپرینت باید قابلیت Scalability داکرایز گردد.

اسپرینت ششم

صفحه سیاست حریم خصوصی (*Privacy & Policy*) و صفحه شرایط و ضوابط استفاده (*Terms & Condition*) ایجاد شود.

در این اسپرینت صفحه درباره ما نیز اضافه خواهد شد.

در این اسپرینت همچنین دیباگ و تست و دیپلوی نهایی پروژه نیز انجام خواهد شد.

نکته) جزئیات هر اسپرینت به صورت *Task*های مختلف در *YouTrack* قابل دسترسی خواهد بود.

نکته) جزئیات Burn Down Chart نیز در فایلی مجزا پیوست شده است.