TEKNISK SLUTDOKUMENTATION

MarketGrowth

En Serverless Cloud plattform för Finansiell Analys i Microsoft Azure

Författare: *Albin Stenhoff*

Datum: 2025-12-09

Kurs: Examensarbete - Cloud Developer 24

GitHub: https://github.com/OtrevligAbbe/MarketGrowth/tree/main

Sammanfattning (Abstract)

Finans Teknologi (FinTech) är en sektor som ställer exceptionellt höga krav på systemens tillgänglighet, skalbarhet och datasäkerhet. Detta examensarbete, *MarketGrowth*, utforskar hur moderna moln teknologier kan appliceras för att möta dessa krav genom en helt serverlös arkitektur ("Serverless"). Projektet omfattar design, utveckling och driftsättning av en webbaserad plattform för övervakning och analys av finansiella marknader i realtid, inklusive kryptovalutor, aktier och globala index.

Applikationen är konstruerad som en distribuerad systemarkitektur i Microsoft Azure. Frontend är byggd i **Blazor Web Assembly** (.NET 8) för att leverera en rik användarupplevelse (SPA) utan behov av server side rendering, vilket flyttar beräkningskraft från servern till klienten. Backend drivs av **Azure Functions** enligt den isolerade arbetsmodellen (.NET Isolated Worker), vilket möjliggör en händelsestyrd och kostnadseffektiv hantering av affärslogik. För datalagring används den globalt distribuerade NoSQL databaser **Azure Cosmos DB**, optimerad genom strategisk partitionering för att hantera stora volymer, läsa och skriva operationer med låg latens.

Rapporten redogör ingående för systemets designval, från jämförande studier av SQL kontra NoSQL och Blazor kontra JavaScript ramverk, till den praktiska implementationen av realtidsdata aggregering och asynkrona bakgrundsprocesser. Säkerhetsaspekter hanteras genom **Azure Key Vault** och **Managed Identities** för att eliminera hårdkodade hemligheter. Vidare beskrivs den automatiserade utvecklingsprocessen (DevOps) där **GitHub Actions** används för Continuous Integration och Continuous Deployment (CI/CD). Resultatet påvisar att en serverless arkitektur drastiskt kan reducera både initiala utvecklingskostnader och löpande driftkostnader, samtidigt som prestanda och säkerhet behålls på företagsnivå.

Conclusion (Abstract)

Financial technology (FinTech) is a sector that places exceptionally high demands on system availability, scalability, and data security. This thesis project, *MarketGrowth*, explores how modern cloud technologies can be applied to meet these requirements through a fully serverless architecture. The goal of the project was to design, develop, and deploy a web based platform for real time monitoring and analysis of financial markets, including cryptocurrencies, stocks, and global indices.

The application is constructed as a distributed system architecture within Microsoft Azure. The frontend is built using **Blazor WebAssembly** (.NET 8) to deliver a rich user experience (Single Page Application) without the need for server side rendering, thereby shifting computational power from the server to the client. The backend is powered by **Azure Functions** utilizing the isolated worker model (.NET Isolated Worker), enabling event driven and cost effective handling of business logic. For data storage, the globally distributed NoSQL database **Azure Cosmos DB** is used, optimized through strategic partitioning to handle large volumes of read and write operations with low latency.

The report details the system's design choices, ranging from comparative studies of SQL versus NoSQL and Blazor versus JavaScript frameworks, to the practical implementation of real time data aggregation and asynchronous background processes. Security aspects are managed through **Azure Key Vault** and **Managed Identities** to eliminate hard coded secrets. Furthermore, the automated development process (DevOps) is described, where **GitHub Actions** is used for Continuous Integration and Continuous Deployment (CI/CD). The results demonstrate that a serverless architecture can drastically reduce both initial development costs and ongoing operational costs, while maintaining performance and security at an enterprise level.

Innehållsförteckning

1. Inledning

1.1 Bakgrund och Motiv

Den finansiella sektorn genomgår en snabb digital transformation. Traditionella system för börshandel och analys körs ofta på lokala servrar ("On-Premise") eller via statiska virtuella maskiner (IaaS). Dessa "monolitiska" system lider ofta av höga underhållskostnader, långsamma uppdateringscykler och svårigheter att skala elastiskt vid plötsliga trafiktoppar. När marknaden rör sig kraftigt ökar antalet användare exponentiellt, vilket ställer krav på att systemet automatiskt kan allokera mer resurser utan manuell inblandning.

Samtidigt har utvecklingen inom molnteknik (Cloud Computing) möjliggjort nya arkitektoniska mönster. **Serverless computing** har seglat upp som en dominant modell för att bygga moderna applikationer där utvecklare kan fokusera uteslutande på kod och affärslogik, medan molnleverantören hanterar all underliggande infrastruktur, patchning och skalning. *MarketGrowth* föddes ur viljan att applicera dessa moderna principer för att skapa en lättviktig men kraftfull plattform för marknadsanalys.

1.2 Problemformulering

Att bygga distribuerade system medför en rad tekniska utmaningar jämfört med traditionella applikationer:

- **Statelessness:** Hur hanterar man användarsessioner och tillstånd när backend tjänsterna är helt tillståndslösa och flyktiga?
- **Datakonsistens:** Hur säkerställer man data aggregering från tredje part (API:er) på ett sätt som är resilient mot driftstörningar och hastighetsbegränsningar (Rate Limits)?
- **Säkerhet:** Hur skyddas känsliga API nycklar och databasanslutningar i en miljö där koden exekveras i en delad molninfrastruktur?
- **Latens:** Hur minimeras "Cold Starts" i en serverless miljö där funktioner kan gå ner i viloläge vid inaktivitet?

1.3 Syfte och Projektmål

Det primära syftet med examensarbetet är att designa, implementera och driftsätta en fullskalig molnapplikation i Microsoft Azure. Projektet syftar till att visa hur komponenter som Azure Functions, Cosmos DB och Static Web Apps kan samverka för att lösa ovanstående problem.

Konkreta mål:

1. Skapa en responsiv webb klient med **Blazor Web Assembly** för att utnyttja C# kompetens fullt ut.
2. Utveckla ett RESTful API med **Azure Functions** (.NET 8 Isolated Worker).
3. Implementera persistent lagring av användardata och historik i **Azure Cosmos DB**.
4. Automatisera hämtning av marknadsdata från externa källor (CoinGecko, Alpha Vantage) med robust felhantering.
5. Säkra applikationen genom **Azure Key Vault** och Managed Identities.

6. Upprätta en automatiserad leveranskedja (**CI/CD**) från GitHub till Azure.

1.4 Avgränsningar

Projektet fokuserar på den tekniska arkitekturen och implementationen av kärnfunktionalitet.

- Autentisering: En fullständig integration mot Azure AD B2C har utelämnats till förmån för en anpassad, förenklad autentiseringslösning (AuthState) för att demonstrera state hantering tydligare.
- **Betalningar:** Inga betalningslösningar är implementerade.
- **Marknader:** Datan är begränsad till utvalda kryptovalutor och aktier för demonstrationssyfte.

2. Teoretisk Referensram & Teknikval

2.1 Cloud Computing och Molnmodeller

Cloud Computing definieras som leverans av datortjänster - inklusive servrar, lagring, databaser, nätverk och mjukvara - över internet. I detta projekt används primärt **PaaS (Platform as a Service)** och **FaaS (Function as a Service)**. PaaS ger en miljö för utveckling och drift utan komplexiteten att bygga och underhålla infrastrukturen, medan FaaS tar detta ett steg längre genom att exekvera kod som svar på specifika händelser (HTTP-anrop, timers, databasändringar).

2.2 Serverless Arkitektur (FaaS)

"Serverless" innebär inte att det saknas servrar, utan att serverhanteringen är helt abstraherad från utvecklaren. Azure Functions är Microsofts implementation av FaaS.

- **Fördelar:** Automatisk skalning (från 0 till tusentals instanser), debiteringsmodell per exekvering (micro billing) och minskad administrativ börda.
- **Nackdelar:** "Cold Starts" - den fördröjning som uppstår när plattformen måste starta en ny instans för att hantera en förfrågan efter en period av inaktivitet.

I *MarketGrowth* används **.NET 8 Isolated Worker Model**. Detta innebär att funktionen körs i en separat .NET process, frikopplad från Azure Functions runtime. Det ger full kontroll över uppstarten, inklusive Dependency Injection (DI) och Middleware konfiguration, vilket är avgörande för att strukturera koden rent och testbart.

2.3 Single Page Applications (SPA) och Blazor

En SPA är en webbapplikation som laddar en enda HTML sida och dynamiskt uppdaterar innehållet när användaren interagerar med appen, vilket ger en "app liknande" känsla utan omladdningar.

Jämförelsestudie: Blazor Web Assembly vs JavaScript (React/Angular)

Marknaden för SPA domineras traditionellt av JavaScript baserade ramverk. Valet av Blazor Web Assembly för detta projekt motiveras av följande faktorer:

1. **Enhetlig teknikstack:** Genom att använda C# och .NET över hela linjen (fullstack) elimineras behovet av kontext växling mellan språk.
2. **Kod Delning (Code Sharing):** DTO:er (Data Transfer Objects) och validering logik kunde delas via ett gemensamt klassbibliotek. I en React miljö hade dessa behövt dupliceras i TypeScript, vilket ökar risken för inkonsekvens.
3. **Prestanda:** Web Assembly exekverar binär kod i webbläsaren med nära native prestanda för beräkningstunga uppgifter.

2.4 NoSQL database &CAP theorem

Till skillnad från relationella databaser (SQL) som använder strikta scheman och relationer, lagrar NoSQL databaser data i flexibla format som JSON.

Jämförelsestudie: Azure Cosmos DB vs Azure SQL

Valet av databas baserades på CAP theorem (Consistency, Availability, Partition Tolerance).

- **Azure SQL (Relationell/ACID):** Prioriterar stark konsistens. Passar bra för komplexa transaktioner men är svårare och dyrare att skala globalt.
- **Cosmos DB (NoSQL/BASE):** Prioriterar tillgänglighet och partitions tolerans. Valdes för *MarketGrowth* av följande skäl:
    - **Schema Flexibilitet:** Data från externa APIer (CoinGecko/Alpha Vantage) kan ändra struktur. NoSQL hanterar detta utan schema migrationer.
    - **Skrivprestanda:** Optimerat för extremt snabba skriv operationer och global distribution, vilket är en förutsättning för framtida skalning av realtidsdata.

2.5 DevOps och CI/CD

DevOps är en kultur som förenar utveckling (Dev) och drift (Ops). CI/CD (Continuous Integration/Continuous Deployment) automatiserar processen från kodändring till produktion. I detta projekt används **GitHub Actions** för att definiera pipelines som kod (YAML), vilket säkerställer att varje ändring automatiskt testas, byggs och publiceras.

3. Metod & Genomförande

3.1 Utvecklingsmetodik

Arbetet har bedrivits iterativt med inspiration från Scrum. Projektet delades in i två veckors långa sprintar:

1. **Sprint 1:** Infrastruktur (Azure resurser) och "Hello World" verifiering av pipeline.
2. **Sprint 2:** Backend logik, API integrationer och databaskoppling.
3. **Sprint 3:** Frontend utveckling, UI design och state hantering.

4. **Sprint 4:** Integration, säkerhetshärdning (Key Vault) och polering.

3.2 Verktyg och Miljö

- **IDE:** Visual Studio 2022 Enterprise.
- **Språk:** C# 12 / .NET 8.
- **Versionshantering:** Git & GitHub.
- **API Testning:** Postman användes för att validera API anrop oberoende av frontend.
- **Lokal Emulering:** Azure Functions Core Tools och Azurite (för lokal Storage emulering) möjliggjorde en snabb "inner development loop" utan att behöva deploya till molnet för varje liten ändring.

Initialt övervägde jag att använda en SQL databas eftersom jag är mer van vid relationsmodeller. Men efter att ha insett hur ofta datastrukturen från APIerna förändrades, tvingades jag tänka om och byta till Cosmos DB. Det krävde en viss inlärningströskel, men sparade mycket tid i längden.

4. Kravspecifikation & Design

4.1 Funktionella Krav

Följande krav identifierades och implementerades:

- **Realtidsöversikt:** Användaren ska kunna se aggregerade priser för Krypto, Aktier och Index på en och samma sida.
- **Visualisering:** Prisutveckling över 7 dagar ska visas grafiskt (Sparklines).
- **Favoriter:** Inloggade användare ska kunna spara och ta bort favoriter.
- **Alerts:** Systemet ska automatiskt upptäcka och logga stora prisrörelser (> 0.01% i demo) i bakgrunden.
- **Persistens:** Data ska sparas i databasen även om användaren laddar om sidan eller loggar ut.

4.2 Icke funktionella Krav

- **Tillgänglighet:** Systemet ska vara tillgängligt dygnet runt (99.9% SLA via Azure).
- **Säkerhet:** Inga hemligheter får finnas i källkoden. HTTPS ska användas för all trafik.
- **Prestanda:** API svar bör vara < 500ms vid cachad data.
- **Kostnadseffektivitet:** Driftskostnaden ska minimeras genom användning av Consumption Plans.

4.3 UI/UX-Design

Designen är inspirerad av moderna tradingplattformar som Capital.com, med ett mörkt tema och tydliga färgkoder (Grön/Röd) för positiva och negativa trender. Responsivitet prioriterades för att applikationen ska fungera på både desktop och mobil.

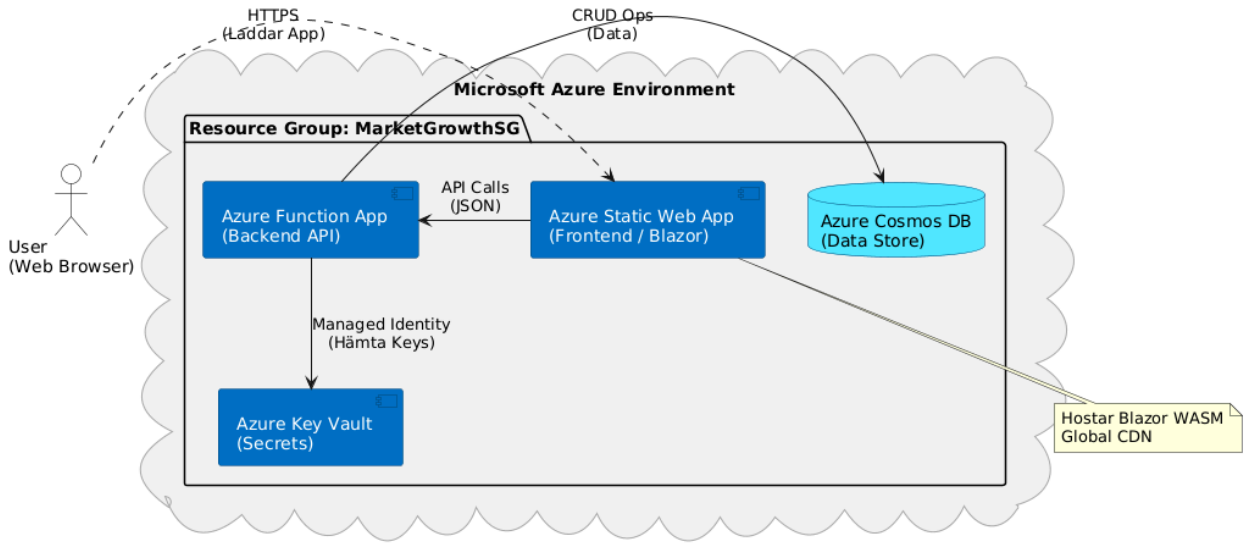
5. Systemarkitektur

5.1 Övergripande Systemdesign

Arkitekturen är baserad på principer för **Microservices** och **Serverless**, där frontend och backend är helt frikopplade från varandra för att möjliggöra oberoende skalning och utveckling.

Systemet består av följande huvudkomponenter:

- **Frontend:** Hostas på **Azure Static Web Apps**. Detta ger global distribution via Content Delivery Network (CDN), vilket säkerställer snabb laddningstid oavsett var användaren befinner sig.
- **Backend:** En **Azure Function App** agerar API Gateway och affärslogik lager. Den tar emot anrop, validerar indata, kommunicerar med databasen och externa API:er, och returnerar standardiserad JSON.
- **Data: Azure Cosmos DB** lagrar all persistent data (NoSQL).
- **Säkerhet: Azure Key Vault** hanterar alla hemligheter och nycklar, åtkomligt via Managed Identity.



5.2 Databasmodellering i Cosmos DB

För att optimera både kostnad och prestanda (Request Units - RUs) har datamodellen utformats noggrant. Data är uppdelad i fyra specifika "Containers" där valet av **Partition Key** är kritiskt för skalbarheten.

- **favorites** (Partition Key: **/UserId**)
    - Analys: Den absolut vanligaste frågan är "Hämta alla favoriter för Användare X". Genom att partitionera på **/UserId** kan Cosmos DB rikta frågan till en enda logisk partition. Detta gör läsningar extremt effektiva och kostar minimalt med RUs.
- **markethistory** (Partition Key: **/Symbol**)
    - Analys: Används för snapshots och prishistorik. Partitionering på Symbol (t.ex. "BTC") gör det snabbt att hämta all historik för en specifik tillgång vid generering av grafer.
- **marketalerts** (Partition Key: **/Symbol**)
    - Lagrar genererade larm om prisrörelser, grupperade per tillgång för snabb uppslagning.
- **useralerts** (Partition Key: **/UserId**)
    - Förberedd för framtida funktionalitet där användare kan ställa in egna bevaknings nivåer.

5.3 Applikationsflöden och Proxy mönster

Ett kritiskt flöde i arkitekturen är hanteringen av externa API anrop. Eftersom externa tjänster (till exempel Alpha Vantage) har hårda begränsningar i antal anrop (Rate Limits) och exponerar API nycklar i URL:en, kan klienten inte tillåtas anropa dem direkt.

Lösning: Backend implementerar en Proxy och Caching strategi.

1. Klienten anropar Azure Function (**/api/market/overview**).
2. Azure Function kontrollerar först sin interna **in memory cache**.
3. **Cache Träff:** Om data finns och är giltig (< 60 sekunder) returneras den direkt utan externt anrop.
4. **Cache Miss:** Om cache saknas eller gått ut, hämtar funktionen ny data från det externa API:et (med server side API nyckel), sparar resultatet i cachen och returnerar datan till klienten.

5.4 Projektstruktur och Källkodens Organisation

För att säkerställa hög läsbarhet och underhållbarhet är *MarketGrowth* strukturerat enligt etablerade principer för modern .NET utveckling (Clean Architecture). Projektet är ett "Monorepo" som innehåller både API och Frontend, men de hålls strängt separerade i källkoden.

Projektets övergripande mappstruktur illustreras nedan:

```
MarketGrowth/
│
├── api/
│   └── MarketGrowth.Api/
│       ├── Entities/        (Databasmodeller för Cosmos DB)
│       ├── Functions/       (Azure Functions Triggers)
│       ├── Models/          (DTOs för API kontrakt)
│       ├── Repositories/    (Databaslogik och access)
│       ├── Program.cs       (Dependency Injection & Config)
│       └── host.json
│
├── frontend/
│   ├── Layout/              (NavMenu, MainLayout)
│   ├── Pages/               (Vyer: Home, MarketData, Alerts)
│   ├── Shared/              (Återanvändbara komponenter, t.ex. Sparkline)
│   ├── wwwroot/             (CSS, Bilder)
│   ├── App.razor
│   └── Program.cs           (Klient konfiguration)
│
├── .github/
│   └── workflows/           (CI/CD Pipelines)
│
└── README.md
```

5.4.1 Backend struktur (Azure Functions)

Backend projektet är uppbyggt för att separera ansvar:

- Entities: Innehåller systemets datamodeller (t.ex. **FavoriteAssetEntity**) som mappar direkt mot dokumenten i Azure Cosmos DB.
- **Models:** Innehåller DTO klasser (Data Transfer Objects) som används för API kommunikation. Detta separerar den interna datalagringen från det externa kontraktet.
- **Repositories:** Här kapslas all databaskommunikation in (till exempel **MarketSnapshotRepository**). Genom att använda Repository Pattern blir affärslogiken testbar och oberoende av den underliggande databasen.
- **Functions**: Här finns systemets ingångspunkter (till exempel **GetMarketOverviewFunctionApi**). Dessa klasser hanterar HTTP anrop och Timer triggers men delegerar logiken.

5.4.2 Frontend struktur (Blazor WebAssembly)

Frontend följer Blazors standardstruktur för komponentbaserad utveckling:

- **Pages:** Innehåller de navigerbara sidorna (.razor filer). Varje sida ansvarar för sin egen vylogik.
- **Shared:** Innehåller återanvändbara UI komponenter som Sparkline.razor (grafkomponenten) och FavoritesDropdown, vilket möjliggör modulär utveckling.
- **AuthState:** En central tjänst (definierad i Program.cs) som hanterar användarens inloggningsstatus och synkroniserar favoriter mellan olika komponenter i realtid.

5.4.3 CI/CD och Automatisering

Den automatiserade leveranskedjan är definierad i .github/workflows/ och är uppdelad i två separata flöden:

1. **Frontend Pipeline:** Bygger och publicerar Blazor appen till Azure Static Web Apps.
2. **Backend Pipeline:** Bygger .NET funktionen och publicerar till Azure Function App. Detta säkerställer att varje ändring i koden automatiskt testas och driftsätts till produktionsmiljön.

6. Implementation

6.1 Backend: Azure Functions och API design

Backend koden är strukturerad med Dependency Injection (DI) i Program.cs. Här registreras CosmosClient, HttpClient och Repositories som "Singleton" eller "Scoped" tjänster.

6.1.1 Dataaggregering (GetMarketOverview)

Funktionen GetMarketOverview ansvarar för att aggregera data. För att minimera svarstiden anropas externa tjänster parallellt.

En viktig del är felhanteringen. Om det externa API:et returnerar fel (till exempel 429 Too Many Requests), används en "fallback" mekanism som returnerar den senast kända ögonblicksbilden (_lastSnapshot).

```
// Exempel på caching logik i GetMarketOverviewFunction.cs
lock (_cacheLock)
{
    // Kontrollera om data är mindre än 1 minut gammal
    if (DateTime.UtcNow - _stocksLastUpdated < TimeSpan.FromMinutes(1) &&
_cachedStocks.Count > 0)
    {
        return _cachedStocks; // Returnera cachad data direkt från minnet
    }
}
// Annars exekvera HTTP anrop mot Alpha Vantage
```

6.1.2 Favorithantering

Funktionen AddFavorite tar emot en FavoriteAssetRequest Här skapas ett unikt ID genom att kombinera användar ID och tillgångens symbol ($"{userId}_{assetId}"). Detta garanterar kvalitet och en användare kan inte lägga till samma favorit två gånger.

6.2 Frontend: State Management med AuthState

I en Blazor applikation lever användarens tillstånd i webbläsarens minne. Klassen AuthState skapades för att centralisera hanteringen av inloggning och favoriter.

Den använder C# events (Action? OnChange) för att implementera Observer mönstret. När en favorit läggs till anropas NotifyStateChanged(), vilket triggar en omritning i alla prenumererade komponenter (till exmpel Navbar och MarketData vyn).

```
// AuthStateFrontend.cs
public class AuthState
{
    public List<FavoriteAssetRequest> Favorites { get; private set; }
    public event Action? OnChange;

    public async Task ReloadFavoritesAsync() {
        // Hämta asynkront från API
        var favorites = await
_http.GetFromJsonAsync<List<FavoriteAssetRequest>>(url);
        Favorites = favorites;
        NotifyStateChanged(); // Notifiera UI
    }
}
```

6.3 Datavisualisering: Sparklines

En av de visuellt mest komplexa delarna var implementationen av Sparkline.razor. Komponenten tar emot en lista av priser. Eftersom prisnivåerna varierar kraftigt (Bitcoin ~90 000 USD vs Ripple ~1 USD) måste datan normaliseras för att passa i SVG grafen (0-100%).

Implementationen räknar ut (Value - Min) / (Max - Min) för varje punkt för att skapa en relativ Y-koordinat, och inverterar sedan denna (eftersom SVG koordinater börjar uppifrån).

6.4 Bakgrundsprocesser: Alerts och Timers

Funktionen MarketSnapshotTimer är en TimerTrigger som körs enligt CRON uttrycket 0 */5 * * * * (var 5:e minut).

Denna funktion agerar självkörande:

1. Hämtar realtidsdata för krypto.
2. Hämtar senaste snapshot från Market Snapshot Repository.
3. Jämför priserna.
4. Om förändringen är signifikant (>0.01%), skapas en Market Alert Entity och sparas i databasen.

Detta visar styrkan i en händelsestyrd arkitektur: övervakningen och analysen sker kontinuerligt på servern utan att någon användare behöver vara inloggad.

7. Säkerhet och Konfiguration

7.1 Identitetshantering och Managed Identity

En stor säkerhetsrisk i molnutveckling är hanteringen av inloggningsuppgifter (Credential Theft). I detta projekt används **Managed Identity**. Azure Function App har tilldelats en system identitet i Azure AD. Denna identitet har getts behörighet (RBAC) att läsa hemligheter från Key Vault. Det innebär att koden inte innehåller några lösenord alls.

7.2 Hemlighetshantering med Key Vault

Alla känsliga värden, såsom CosmosConnection (Connection String till databasen) och ALPHAVANTAGE_API_KEY, lagras i Azure Key Vault.

I Azure Functions konfiguration refereras dessa värden via syntaxen:

@Microsoft.KeyVault(SecretUri=https://marketgrowth-kv.vault.azure.net/secrets/CosmosConnection/...)

Vid uppstart hämtar Azure automatiskt värdet och injicerar det som en miljövariabel i processen.

7.3 Nätverkssäkerhet och CORS

Eftersom frontend (Static Web App) och backend (Function App) ligger på olika domäner, blockeras API anrop initialt av webbläsarens säkerhetsmekanism (Same-Origin Policy).

Detta löstes genom att konfigurera CORS (Cross-Origin Resource Sharing) i Function App. Endast domänen för Static Web App tilläts explicit att göra anrop, vilket förhindrar obehöriga tredjepartswebbplatser från att utnyttja APIet.

8. Testning & Kvalitetssäkring

8.1 Teststrategi

Kvalitetssäkringen av *MarketGrowth* har följt principen om "Shift Left Testing", där testning integreras tidigt i utvecklingsprocessen. På grund av systemets starka beroende av molntjänster lades tonvikten på integrationstester och manuell verifiering.

8.2 Enhetstestning och Mockning

Kritisk algoritmisk logik, såsom normaliseringen av data i sparkline.razor verifierades genom isolerade tester. Det kontrollerades att grafen renderas korrekt oavsett om trenden är positiv, negativ eller platt (för att undvika division med noll).

8.3 Integrationstestning med Postman

Backend API:et testades omfattande med verktyget Postman. En testsamling ("Collection") skapades för att systematiskt verifiera alla endpoints.

- **Happy Path:** Verifiering att GET /api/market/overview returnerar HTTP 200 och korrekt JSON struktur.
- **Felhantering:** Simulering av ogiltiga anrop (till exempel GET /api/favorites/ utan UserId)för att säkerställa att API:et returnerar HTTP 400 Bad Request istället för 500 Internal Server Error.
- **Rate Limiting:** Genom att skicka upprepade anrop i snabb följd verifierades att caching lagret i Azure Functions aktiverades korrekt.

8.4 Manuell Systemtestning (End-to-End)

System Tester utfördes i produktionsmiljön för att verifiera hela kedjan från klient interaktion till databas persistens och bakgrundsjobb. Testerna fokuserade på att validera funktionalitet och data konsistens i moln miljön.

Scenario 1: Favorit Hantering och Data Konsistens Testet syftade till att säkerställa att användarens val persisteras korrekt över sessioner och att databasens partitionering fungerar korrekt.

1. **UI interaktion:** Användare loggar in och markerar en tillgång (exempelvis Bitcoin) som favorit. UI uppdateras direkt.
2. **Databasverifiering:** Via Azure Data Explorer verifierades att ett nytt JSON-dokument skapades i containern favorites. Det kontrollerades specifikt att dokumentet tilldelades korrekt Partition Key (**/UserId**) för att säkerställa prestanda.
3. **State persistens:** Webbläsaren laddades om och användaren loggade ut och in igen. Testet verifierade att **AuthState** korrekt hämtade den sparade listan från API:et och att UI staten (gul stjärna) bestod.

Scenario 2: Det Händelsestyrda Alert systemet Testet syftade till att verifiera att bakgrundsprocesser exekveras autonomt och genererar data utan användarinteraktion.

1. Manuell Trigger: Funktionen **MarketSnapshotTimer** triggas manuellt via Azure Portal ("Code + Test") för att simulera ett tidsintervall.
2. Observability: Exekveringen övervakades i realtid via Application Insights Live Metrics. Loggflödet verifierade att funktionen hämtade föregående snapshot från **markethistory** containern och beräknade procentuell förändring korrekt.
3. **Frontend-verifiering:** En ny rad genererades i tabellen under fliken "Alerts" i frontend, vilket bekräftade att flödet (Timer -> API -> DB -> Frontend) fungerar som avsett.
    -

9. Drift, Deployment & Övervakning

9.1 Infrastruktur i Azure

Resurserna är organiserade i två separata resursgrupper i regionen Sweden Central. Detta görs för att skapa en tydlig logisk separation mellan applikationslogik (Compute) och datalagring/infrastruktur, vilket underlättar hantering och kostnadsuppföljning.

- Backend grupp (**marketgrowth-api-astenhoff_group**): Innehåller Azure Function App, App Service Plan samt Application Insights för övervakning.
- Infrastruktur grupp (**MarketGrowthSG**): Innehåller databasen (Cosmos DB), säkerhetslagring (Key Vault), lagringskonto (Storage Account) samt frontend värden (Static Web App)."
- **App Service Plan:** Linux (Consumption) - betala per exekvering.
- **Static Web App:** Distribuerar frontend globalt.
- **Application Insights:** För djupgående loggning och telemetri.

9.2 CI/CD Pipelines med GitHub Actions

Projektet använder "Infrastructure as Code" principer. Två workflows definierades i .github/workflows:

1. Frontend Pipeline: Triggas vid push till main. Sätter upp .NET miljö, bygger Blazor projektet (`dotnet publish`) och deployar till Azure Static Web Apps.

2. Backend Pipeline: Bygger Azure Functions projektet, skapar en artefakt (zip fil) och publicerar till Azure Function App.
   Detta säkerställer att produktionsmiljön alltid speglar den senaste koden i repot.

9.3 Telemetri med Application Insights

Azure Application Insights är integrerat i lösningen. Det samlar automatiskt in data om:

- **Failed Requests:** API anrop som returnerar 4xx eller 5xx fel.
- **Performance:** Exekveringstid för funktioner (viktigt för att övervaka Cold Starts).
- Custom Logs: Loggar skrivna via ILogger i koden (till exempel "CoinGecko returned 500").

10. Användarmanual

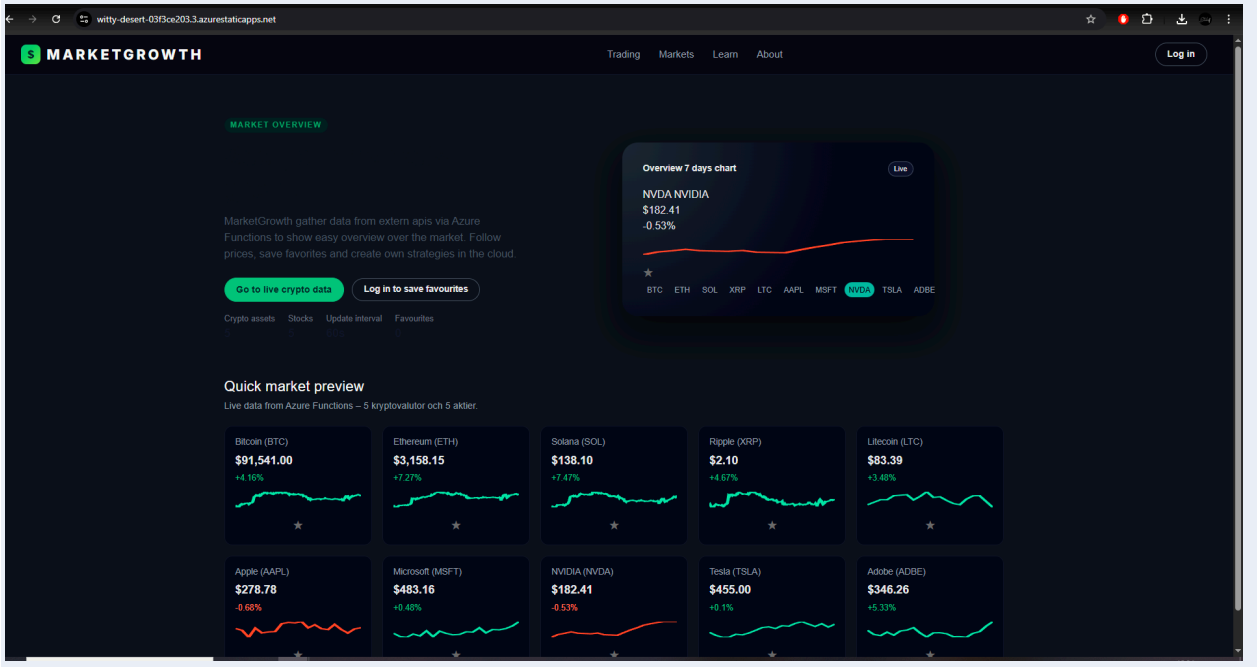
10.1 Översikt

MarketGrowth är en webbaserad plattform som nås direkt via webbläsaren. Ingen installation krävs.

10.2 Startsidan (Dashboard)

Vid första anblick möts användaren av en översikt ("Hero-sektion").

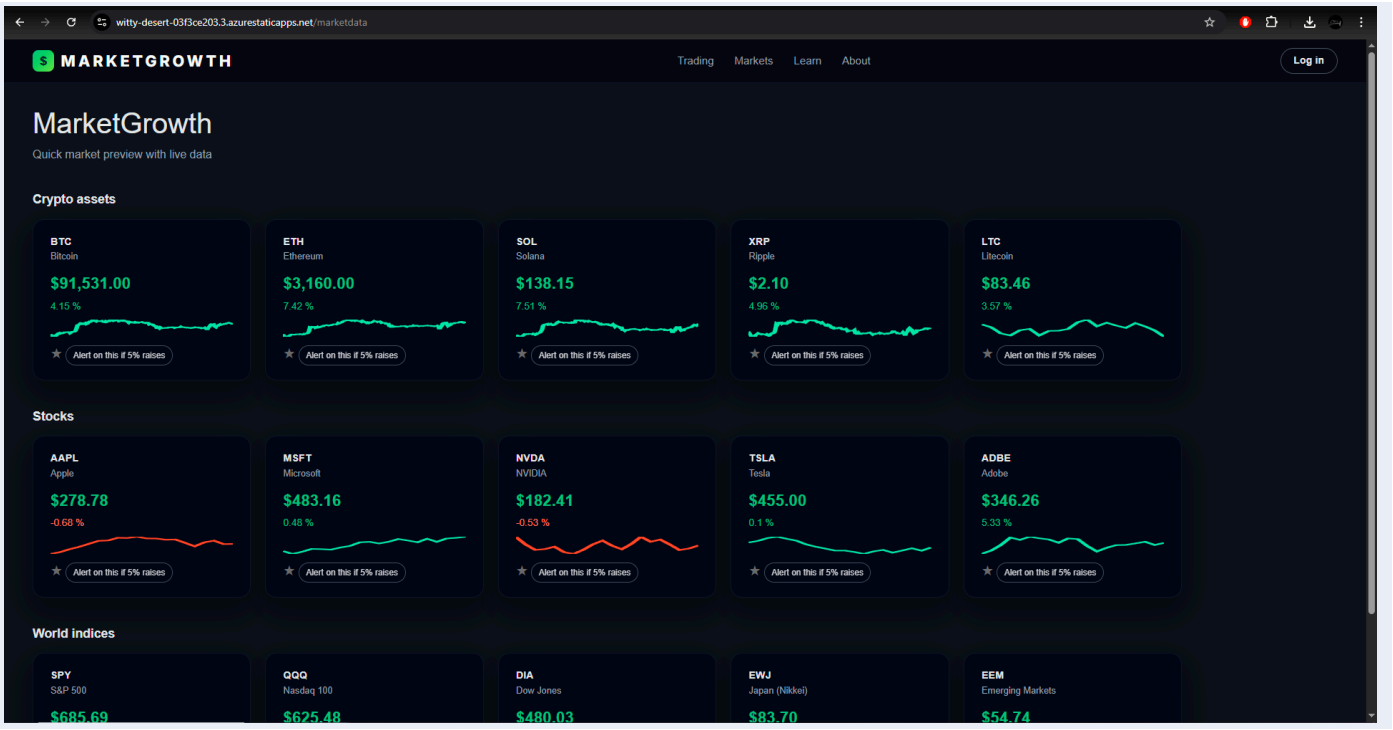
- **Live Chart:** Till höger visas en dynamisk graf över en slumpmässigt vald tillgång (Krypto eller Aktie) för att demonstrera realtidsfunktionaliteten.
- **Statistik:** Informationsboxar visar antalet tillgängliga marknader och aktuell uppdateringsfrekvens.



10.3 Marknadsdata (Market Data View)

Detta är applikationens huvudvy. Här presenteras data i tre kategorier: Crypto, Stocks och Indices.

- **Kortvy:** Varje tillgång representeras av ett kort med Namn, Symbol, Pris och en Sparkline graf.
- **Färgkodning:** Prisförändringar är färgkodade (Grönt = uppgång, Rött = nedgång).
- **Spara Favorit:** Genom att klicka på stjärn ikonen sparas tillgången till användarens personliga lista.



10.4 Inloggning och Alerts

För att spara data krävs inloggning via sidan "Authentication". När användaren är inloggad visas en favorit räknare i menyn. Under sidan "Alerts" visas en historisk tabell över signifikanta marknadshändelser som systemet automatiskt har fångat upp.

11. Resultat & Analys

11.1 Prestandaresultat

- **Frontend:** Initial laddningstid (First Contentful Paint) är under 1.5 sekunder tack vare CDN. Navigation inom appen är omedelbar.
- **API:** Svarstider för /market/overview ligger på ca 200-400ms när cachen är aktiv.

- **Cold Starts**: Vid första anropet efter inaktivitet noteras en fördröjning på ca 2-4 sekunder. Detta är en förväntad bieffekt av Consumption Plan men bedöms acceptabelt för denna typ av applikation.

11.2 Teoretisk Kostnadsmodell

En serverless arkitektur har visat sig vara mycket kostnadseffektiv.

- **Azure Functions:** De första 1 miljon anropen per månad är gratis.
- **Cosmos DB (Serverless):** Debiteras per Request Unit (RU). Genom effektiv partitionering hålls förbrukningen mycket låg (ca 3-5 RUs per läsning).
- **Total drift:** Uppskattad månadskostnad är < 50 SEK vid nuvarande trafikvolymer.

12. Diskussion & Slutsats

12.1 Reflektion kring arkitekturval

Valet av Blazor och Azure Functions visade sig vara mycket framgångsrikt för detta projekt, primärt tack vare synergien i en ren .NET miljö. Möjligheten att dela kod (DTO klasser) mellan klient och server sparade utvecklingstid och eliminerade en hel kategori av buggar relaterade till datatyp konvertering, vilket ofta är en riskfaktor i projekt med separata språk stackar (till exempel C# backend och TypeScript frontend).

Samtidigt är det viktigt att belysa att ett etablerat ramverk som React har betydande fördelar, särskilt gällande det enorma ekosystemet av tredjepartsbibliotek och community stöd. Vid komplexa UI interaktioner eller om kravet på initial laddningstid (payload size) hade varit striktare, hade React sannolikt varit ett starkare alternativ då WebAssembly fortfarande dras med en viss initial startsträcka.

För *MarketGrowth* vägde dock fördelarna med typsäkerhet och en enhetlig utvecklingsmiljö tyngre än Reacts ekosystem, särskilt givet den korta utvecklingstiden. Gällande databasvalet krävde Cosmos DB en initial mental omställning från traditionellt SQL tänk, men det visade sig överlägset för att hantera den varierande och ostrukturerade datan från externa API:er.

12.2 Hantering av tekniska utmaningar

De största utmaningarna var relaterade till integrationen av externa tjänster. API begränsningar hos Alpha Vantage ledde initialt till krascher, vilket löstes med caching och fallback snapshots. CORS konfigurationen krävde också felsökning, där Application Insights spelade en nyckelroll.

Felsökningen av CORS problematiken visade sig vara betydligt mer tidskrävande än väntat. Felmeddelandena i webbläsaren var ofta generella ('Network Error'), vilket gjorde att jag fick lägga flera timmar på att enbart analysera HTTP headers i utvecklar verktygen innan jag förstod att felet låg i Azure Functions konfiguration.

12.3 Hållbarhetsaspekter (Green Cloud)

Genom att använda en serverless arkitektur (Consumption Plan) bidrar *MarketGrowth* till minskad energiförbrukning. Till skillnad från traditionella servrar som drar ström även vid tomgång, förbrukar denna lösning endast beräkningskraft när den faktiskt används.

12.4 Framtida utveckling

För att ta *MarketGrowth* till nästa nivå föreslås:

1. **Azure AD B2C:** Ersätta den enkla inloggningen med en fullvärdig identitetslösning.
2. **SignalR:** Implementera WebSockets för att skicka alerts direkt till klienten ("Push") utan omladdning.
3. **Frontend samt backend:** Frontend och backend går alltid att bygga starkare, bättre och mer robust.

12.5 Slutsats

Det största personliga utbytet av detta projekt har inte varit att lära sig enskilda verktyg som Blazor eller Azure Functions, utan förståelsen för hur de hänger ihop i ett större ekosystem. Att gå från att skriva kod som fungerar lokalt på min maskin, till att faktiskt ansvara för en produktionsmiljö i molnet, har gett mig en helt ny respekt för DevOps och säkerhet.
MarketGrowth projektet har framgångsrikt uppfyllt alla uppsatta mål. Det demonstrerar kraften i en serverless arkitektur: genom att flytta fokus från infrastruktur till kod kunde en komplex, skalbar och säker finansiell plattform byggas på kort tid med minimala resurser. Projektet utgör en solid grund för vidare utveckling inom Cloud Development.

13. Referenser

1. **Microsoft Learn.** *Azure Functions documentation.* Hämtad från: https://learn.microsoft.com/en-us/azure/azure-functions/
2. **Microsoft Learn.** *Partitioning and horizontal scaling in Azure Cosmos DB.* Hämtad från: https://learn.microsoft.com/en-us/azure/cosmos-db/partitioning-overview
3. **ASP.NET Core Blazor.** *Blazor WebAssembly hosting models.* Hämtad från: https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models
4. **GitHub.** *MarketGrowth Repository - Källkod.* https://github.com/OtrevligAbbe/MarketGrowth/tree/main
5. **CoinGecko API.** *Documentation.* https://www.coingecko.com/en/api
6. **Alpha Vantage API.** *Documentation.* https://www.alphavantage.co/documentation/