

Softwareudvikling bachelor rapport



BetterBoard

Automatiseret End-to-End Test System

TEAM:

ANDERS STRÆDE BJERREGAARD NIELSEN

ASBN26877@EDU.UCL.DK

AFLEVERINGSDATO:

3. JUNI 2024

UDDANNELSE:

PBA I SOFTWAREUDVIKLING

INSTITUTION:

UCL, ERHVERVSAKADEMI & PROFESSIONSHØJSKOLE, ODENSE SEEBLADSGADE

VEJLEDER:

KENNETH JEPSEN CLAUSEN

KJCL@UCL.DK

ANSLAG: 58.655

SOURCE CODE: [HTTPS://GITHUB.COM/ANDERSBJERREGAARD/BETTERBOARD-BA](https://github.com/ANDERSBJERREGAARD/BETTERBOARD-BA)

INDHOLDSFORTEGNELSE

Indholdsfortegnelse	2
Forord	4
Indledning	5
Anti-Pattern	5
BetterBoard	5
Problemstilling	6
Problemformulering	6
Afgrænsning	7
Automatiseret Acceptance Test	7
Cloud Provider	7
Tech Stack	7
Container Engine	7
Metodologi	8
CI / CD Pipelines	8
Secret / Key Management	9
Containerisering	10
Azure Cloud & Tooling	10
Infrastructure-as-Code	10
Analyse	12
Source Control	12
Azure Repos	12
GitHub	13
Kvalitetssikring	13
Testniveau & -Type	14
Test Miljø & Stadie	15
Automatiseret Web Interaktion	19
Playwright	20
Nightwatch	21
Cypress	22
Selenium	22
Azure Service Type for Selenium Grid	23
Azure Kubernetes Service	23

Azure Container Apps	24
Udrulning af Selenium Grid	24
Infrastructure as Code	25
.NET Aspire	26
Tests.....	30
Implementering.....	31
CI / CD.....	34
Automatiseret Acceptance Tests.....	36
Konklusion	38
Litteraturliste	39
Bilag	41

FORORD

Kode eksempler i denne rapport kan forekomme med følgende snippet: '{...}'. Dette fremviser en kollapsed metode, constructor, klasse eller lignende. Hvore den indre logik er blevet undladt med vilje, til gavn for læsbarhed og forkortelse. Optræder den nævnte snippet, er det fordi koden derunder ikke menes at være relevant for den kontekst, det bliver beskrevet i.

INDLEDNING

ANTI-PATTERN

Mange moderne applikationer involverer adskillige bevægelige dele og er komplekse at udrulle. Alligevel er der mange organisationer der udruller software manuelt (Humble & Farley, 2011). Dette er typisk et afkom af at skulle manuelt verificere om softwaren kører korrekt.

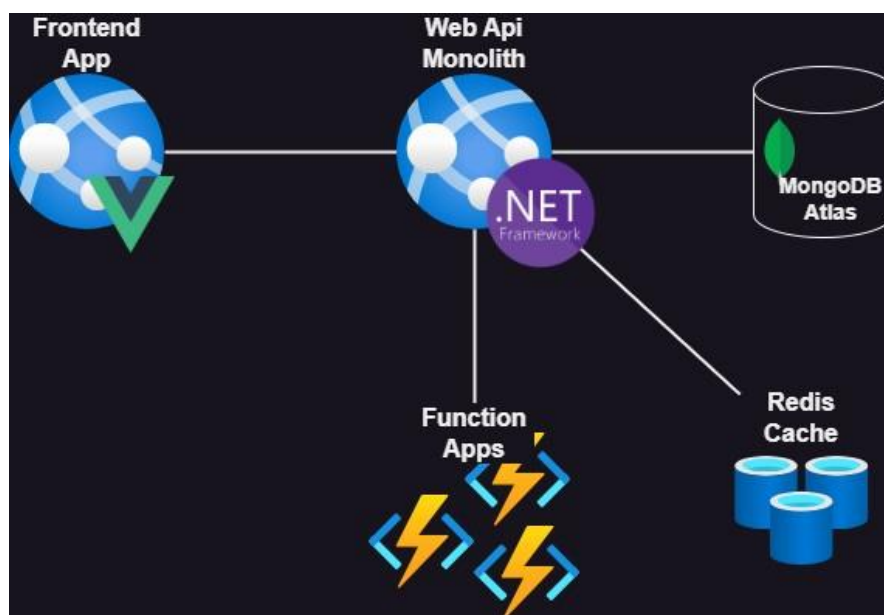
Manuel udrulning er sårbar for menneskelige fejl. Automatisk udrulning løser disse sårbarheder, ved at dokumentere processen i form af scripts der giver et forudsigeligt resultat.

Dog, er en omfattende automatisk kvalitetssikringsproces nødvendigt for ikke at udrulle fejlagtig software.

BETTERBOARD

BetterBoard ApS – eller BB som forkortelse – er en software virksomhed der udvikler og sælger online software til bestyrelser, foreninger og andre grupper. Bestyrelsessoftware strukturerer og forbedrer aktiviteter i forbindelse med planlægning, eksekvering og opfølgning af bestyrelsesarbejdet og andre ledermøder (BetterBoard, 2024).

Fra et overbliksperspektiv har BB's software systemer følgende arkitektur:



Figur 1: BetterBoard software system arkitektur

Software kvalitetssikringsprocessen består af en suite af unit tests på web api monolitten, og en manuel smoke test efter hver udrulning. I øvrigt er udrulningsprocessen for systemerne kun af 'continuous delivery' karakteristika. Med andre ord: Produktionsklar software artefakter bliver genereret men ikke udrullet automatisk.

PROBLEMSTILLING

BetterBoard har en ambition om at kunne udrulle ændringer, lavet i monolittens kodebase, som minimum 8 gange om dagen.

Udviklingsafdelingen får skabt nok ændringer i løbet af en dag, til ovenstående mål. Men den eksisterende CI / CD pipeline udfører kun continuous delivery, og ikke continuous deployment. Dette er et bevidst valg, da der ikke er høj nok tillid til den eksisterende automatiske kvalitetssikring for softwareproduktet.

Derudover står majoriteten af BB's distribueret softwareinfrastruktur udokumenteret. Da det primært er blevet kreeret gennem brugergrænsefladen på Azure. Det er der interesse i at afvige fra: Ved at diktere at al fremtidig distribueret software infrastruktur skal dokumenteres, som minimum med et passende infrastructure-as-code værktøj.

PROBLEMFORMULERING

Denne opgave ønsker at opsætte accept kriterier for en forbedret automatisk software kvalitetssikringsproces. Til formål for at garantere firmaets tillid til en automatisk continuous deployment. Opgaven indebærer også at implementere tilstrækkelig tests til at opfylde nævnte kriterier.

Opgaven ønsker ydermere at ekspandere på eksekveringsplanen for det nuværende CI / CD flow, med den nyligt implementeret kvalitetssikring.

Derudover vil al ny software, der skal udrulles til cloud provideren i denne opgave, dokumenteres med passende IaC.

Opsummeret set undersøger denne opgave følgende:

Hvordan kan BetterBoard sikre en høj kvalitet i det leverede produkt, når der er en ambition om at opdatere det mindst 8 gange om dagen – og ikke har et team af testere siddende.

Ud fra ovenstående udliciteres følgende underspørgsmål:

Hvordan kan BB kvalitetssikre dets softwareprodukter, løbende og kontinuerligt, når softwaren opdateres eller ændres, før det når ud til slutbrugeren?

Hvad skal der til for at implementere en automatiseret kvalitetssikring, der som minimum dækker de manuelle tests der bliver foretaget før hver udrulning?

Hvad skal der til at dokumentere den automatiseret drift?

AFGRÆNSNING

Denne opgave går i dybden med en række emner. Nogle af disse emner – som kan ses i underoverskrifterne forneden – vil være afgrænset til en mere snæver specifikation. Formålet er at give læseren indblik i den specifikke kontekst, baseret på terminologien.

Dette er et afkom af at visse ord kan have en bred betydning.

AUTOMATISERET ACCEPTANCE TEST

Automatiseret acceptance tests er et emne der bliver berørt ofte i denne opgave. Ordet 'acceptance test' kan have en bred betydning. Men i denne rapport, går det ud fra den definition at det er forretningsorienterede tests der understøtter udviklingsteamet (Crispin & Gregory, 2009).

CLOUD PROVIDER

På baggrund af at denne opgave er i samarbejde med et firma, som distribuerer sin software gennem Microsoft Azure. Skal det forstås at Azure er cloud provideren, når der tales om udrulning eller distribuering af software.

TECH STACK

Førnævnte firma har en del eksisterende software skrevet i en bestemt tech stack. Denne opgave vil forsøge at holde sig tæt på C# .NET, i de systemer hvor det er passende, af den årsag at produkterne beskrevet i denne rapport har til hensigt at kunne driftes af eksisterende udviklere i virksomheden.

CONTAINER ENGINE

Docker er den valgte container engine i produkterne beskrevet i denne opgave. På baggrund af at det efter seneste statistik, er den mest udbredte teknologi inden for dette emne (Baresi, Quattrocchi & Rasi, 2023). Samt er det – på tidspunktet at denne opgave blev udført – den bedst supporteret container runtime på Microsoft Azure og dets dokumentation.

METODOLOGI

Den fundamentale teori bag de væsentligste værktøjer og emner bliver gennemgået i dette afsnit. Formålet er at bringe læseren bedst muligt ind i analyse, refleksion og konklusionerne i denne rapport.

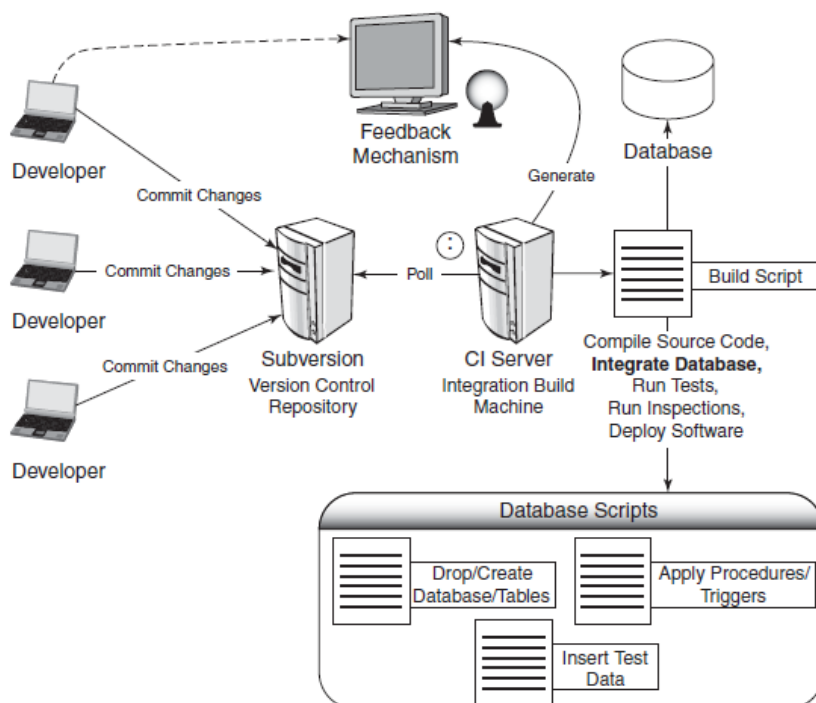
CI / CD PIPELINES

Continuous integration, delivery & deployment eller CI / CD pipelines. Er et værktøj der bruges til at integrere kildekodeændringer, kompilere systemartefakter eller udrulle applikationer. I takt med disse primære mål, inkorporeres der ofte automatiseret og kontinuerlig database integration, testing, inspicering, udrulning og feedback i en CI / CD pipeline. Herved reduceres risici og der skabes en mere tillidsfuld distribuering (Duvall, Matyas & Glover, 2007).

Pipelines er ofte defineret som en række 'trin' eller 'jobs' i en yaml fil. Udtrykt efter en specifik skema reference, eksempelvis Azure Pipelines eller GitHub Action Workflows. Trinnene består typisk af ovennævnte primære mål og inkorporeringer. Derudover har en pipeline typisk en 'trigger', der beskriver hvornår den skal eksekveres. Eksempelvis ved en pull request mod en branch i source control.

Pipelines bliver eksekveret på en CI server. Disse er oftest virtuelle maskiner, som er baseret på en bestemt CPU arkitektur og operativsystem, alt efter pipeline kravene. I tilfælde af at der er specifikke software krav til at eksekvere en pipeline, kan det pre-installeres på CI serveren eller installeres dynamisk: Som for eksempel ved brug af 'tasks' i Azure Pipelines (UseDotNet@2, 2024).

En CI server kan hostes af en cloud provider, eller på egen hånd. Eksempelvis udstiller GitHub og Microsoft softwaren til at eksekvere deres tilsvarende CI server logik. GitHub bruger udtrykket 'self-hosted GitHub actions runner' og Microsoft 'self-hosted agent' til dette.



Figur 2: CI / CD eksempel (Duvall, Matyas & Glover, 2007)

SECRET / KEY MANAGEMENT

Kritiske oplysninger skal håndteres sikkert for at undgå lækage og misbrug. I stedet for at hardkode sådanne oplysninger direkte i kildekoden, anvendes dedikerede værktøjer og metoder til at administrere dem. Dette gør det muligt at opbevare og bruge nøgler på en sikker måde, hvilket øger sikkerheden og gør det lettere at open-source kildekode uden at kompromittere sikkerheden.

Med Azure som cloud provider, kan man benytte Azure Key Vault til at gemme og hente sine kritiske oplysninger.

Som følge af Azure's ekstensive Access Control logik, kan det med høj nøjagtighed specificeres, hvem eller hvad har adgang til hvilke informationer.

Forneden ses et eksempel på en Azure Pipeline, som henter nogle kritiske oplysninger fra en Azure Key Vault. Disse oplysninger bliver da gemt in-memory på CI serveren så længe pipelinen er under eksekvering.

```

1  trigger:
2    - main
3
4  stages:
5    - stage: Deploy
6      displayName: 'Deploy Resources'
7      jobs:
8        - job: AzureCLI
9          pool:
10             name: Default
11          steps:
12            - task: AzureKeyVault@2
13              inputs:
14                KeyVaultName: 'keyvault_name'
15                SecretsFilter: '*'
16                RunAsPreJob: false
17              displayName: 'Retrieve Key Vault Secrets'
18            - task: Bash@3
19              inputs:
20                filepath: './deploy.sh'
21              env:
22                APP_ID: $(AppId)
23                APP_SECRET: $(AppSecret)
24                TENANT_ID: $(TenantId)
25              displayName: 'Deploy script'

```

Figur 3: Azure Key Vault eksempel i Azure Pipelines

Pipelines i Azure har ofte en tilkoblet service connection, som beskriver hvilke ressourcer den specifikke pipeline har adgang til (Service Connections, 2024). Dette resulterer i at adgangen er implicit, og ikke behøves at blottes i yaml filen.

CONTAINERISERING

I en software kontekst: Containerisering er handlingen at pakke software med kun operativsystem biblioteker og afhængigheder, som er påkrævet at eksekvere softwaren i en enkelt letvægts executable – en container – der kører konsistent på enhver infrastruktur (IBM, 2024).

Baseret på industristandarden sat af Docker Engine, containeriseres software oftest gennem en Dockerfile som beskriver et udgangspunkt efterfulgt af en række trin, til at opsætte ens software. Essensen af containerisering er at det er idempotent: Det bliver det samme resultat, uanset hvor mange gange det udføres.

AZURE CLOUD & TOOLING

INFRASTRUCTURE-AS-CODE

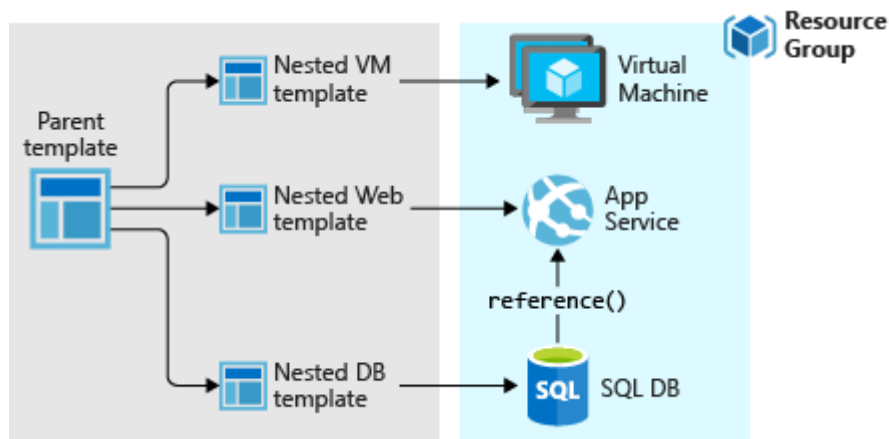
Azure's primære Infrastructure-as-Code værktøj, eller IaC som forkortelse, er Azure Resource Manager templates eller ARM templates. Disse skabeloner beskriver Azure ressourcer og deres relationer i JSON.

```
1  "resources": [  
2    {  
3      "type": "Microsoft.Storage/storageAccounts",  
4      "apiVersion": "2022-09-01",  
5      "name": "mystorageaccount",  
6      "location": "centralus",  
7      "sku": {  
8        "name": "Standard_LRS"  
9      },  
10     "kind": "StorageV2"  
11   },  
12 ]
```

Figur 4: ARM template eksempel af et Azure Storage Account (Azure Resource Manager, 2024)

Når man udruller en ARM template, bliver det konverteret til http requests op mod Azure Resource Manager REST API.

ARM understøtter også at have en 'parent' template der refererer til andre 'nested' templates, til formål for at kunne dokumentere en multi-ressource applikation.



Figur 5: Multi-tier solution (Azure Resource Manager, 2024)

Dog, er ARM templates i JSON sjældent brugt for sig selv. Microsoft har udviklet et Azure native IaC sprog ved navn Bicep, der kompilerer til ARM.

Bicep bliver nu promoveret som det primære IaC værktøj til Azure. Hvis man skal dokumentere sin infrastruktur og automatisere sin udrulning. Sproget erstatter at skrive ARM templates manuelt, idet det har bedre tooling og en officiel Language Server Protocol. Som hjælper med at fange fejl allerede ved compile-time.

```

1  targetScope='subscription'
2
3  param resourceGroupName string
4  param resourceGroupLocation string
5
6  resource seleniumRG 'Microsoft.Resources/resourceGroups@2023-07-01' = {
7    name: resourceGroupName
8    location: resourceGroupLocation
9  }
```

Figur 6: Bicep eksempel på et Azure Storage Account

ANALYSE

SOURCE CONTROL

Et aspekt der bør overvejes, er noget så simpelt som hvor kodebasen skal ligge for projektet. Det vil have indflydelse på mange dele af projektet, da man hæfter sig på en værktøjslinje der integrerer med den valgte source control provider. Eksempelvis er der værktøjer i Azure der kun kan integrere med en kodebase i Azure Repos.

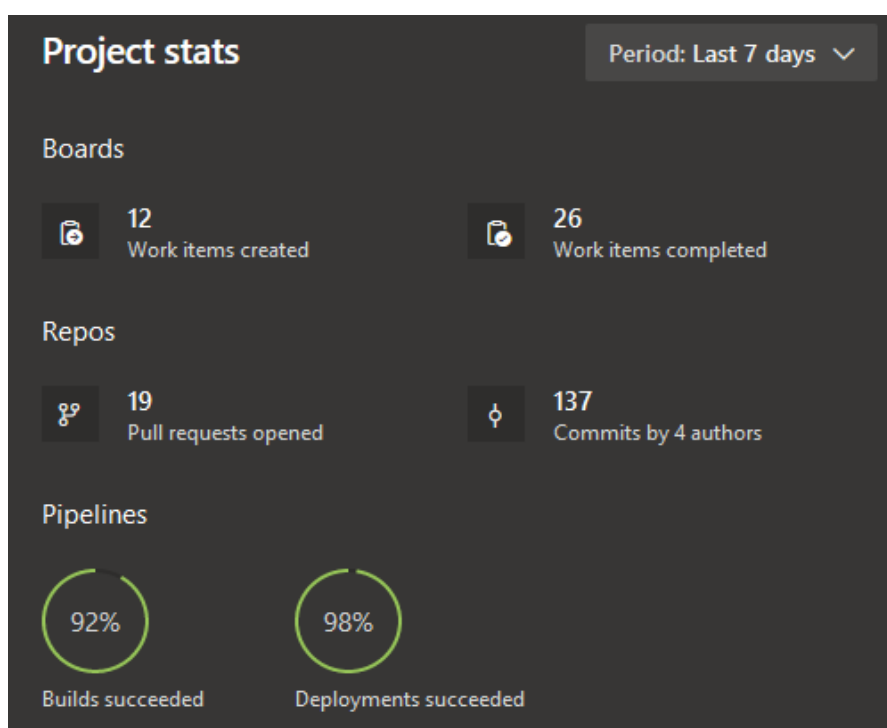
Ydermere, har valget af source control provider indflydelse på tilgængelighed.

Disse aspekter er noget som vil blive analyseret igennem underliggende overskrifter.

AZURE REPOS

Azure Repos er Microsoft Azure's udstillede værktøj til at hoste kildekode. Det er en del af en større værktøjslinje "Azure DevOps".

Hvis et projekt bliver udført af en organisation eller et team med flere medlemmer, kan Azure Repos være et godt valg. Især fordi det kommer med en række værktøjer der kan give indsigt i projektets proces, ved at konfigurere forskellige metrikker at måle på:



Figur 7: Azure Repos Project stats eksempel

Ydermere til projektstyring er Azure DevOps Boards. Som er et gennemført projektstyringsværktøj, direkte integreret med kildekode. Således at man kan linke arbejdsopgaver til pull requests eller commits.

Derudover giver Azure Repos også lejlighed til at benytte Azure Pipelines som CI / CD værktøj.

Er Azure den valgte cloud provider, er der gevinster ved også at holde sig til samme tech stack, og vælge Azure Repos som source control provider.

Dog er Azure Repos efter standard låst for anonym adgang til et givent repository. Skulle et projekt være til hensigt at være open-source, kræver det en hel del opsætning, med hensyn til visibility og access control (Azure Project Visibility, 2023).

GITHUB

GitHub, derimod, tager altid udgangspunkt i at et projekt skal være open-source. Og letter dermed processen at opsætte projektet til disse vilkår.

Det har ikke så gennemført en integreret værktøjskasse som Azure DevOps, men har nogle unikke indsigts metrikker tilgængelig: Som for eksempel Dependency Graph, som giver oversigt over tredjeparts værktøjer som ens kildekode er afhængig af.



Figur 8: GitHub Dependency Graph

Derudover har GitHub også lignende alternativer til DevOps, såsom GitHub Projects, -Wiki og -Actions.

Den væsentligste forskel mellem GitHub og Azure Repos, er at Repos integrerer "out-of-the-box" med Azure som cloud provider. Det kræver en del mere opsætning at skulle interagere med distriberede Azure ressourcer gennem GitHub – Som vil blive gennemgået under afsnittet "CI / CD" i denne rapport.

Azure Repos vurderes til at være den mest velegnet source control provider til dette projekt. Men på baggrund af tilgængeligheden, blev GitHub den valgte provider.

KVALITETSSIKRING

Som benævnt i problemformulerings afsnittet. Er formålet med projektet at ekspandere den automatiske kvalitetssikring.

Dette er et væsentligt mål, da det hænder at eksisterende- eller ny funktionalitet slår fejl i nye udrulninger.

En mere gennemført automatisk kvalitetssikring vil formindske regressionsfejl i nye udrulninger. Derudover, vil det skulle skabe en højere tillid til fremtidige udrulninger. Således at systemet kan begynde at blive udrullet automatisk i en CI / CD pipeline.

Dermed skal det undersøges hvorledes kvalitetssikring kan implementeres, til formål for at minimere regressionsfejl i eksisterende- eller ny funktionalitet.

TESTNIVEAU & -TYPE

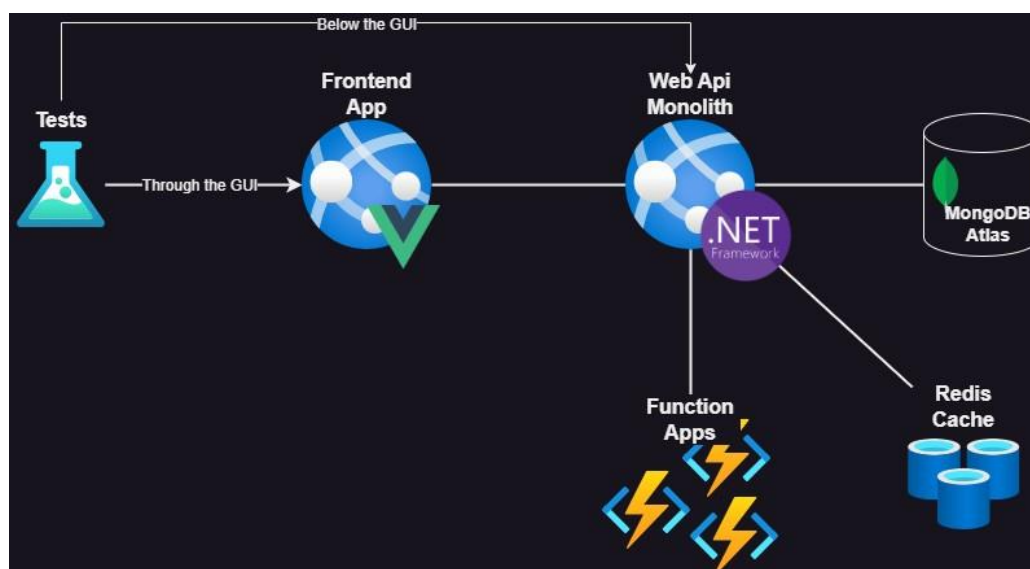
På baggrund af at problemstillingen indebærer funktionel regressionsfejl. Vurderes det at en implementering af funktionel acceptance tests, er en passende strategi.

Det skal understreges, at det er væsentligt at implementeringen bør kunne eksekveres automatisk, da det opfylder et anti-pattern at udføre disse tests manuelt. For at forhindre at udrulle fejl, bør acceptance tests gennemføres hver gang der udrulles (Humble & Farley, 2011).

Ydermere er det vigtigt at disse tests, så vidt muligt, simulerer slutbrugernes interaktion med systemet. Som følge af et citat Humble & Farley's Continuous Delivery: *"Acceptance tests are intended to simulate user interactions with the system in a manner that will exercise it and prove that it meets its business requirements."* (Humble & Farley, 2011).

Opsummeret set kan funktionel acceptance tests implementeres med 2 forskellige indgangsvinkler på systemet:

- Brugergrænsefladen (UI-laget)
- API'et (Applikationslaget)



Figur 9: Test grænseflader

Tests op mod applikationslaget vil kræve at den individuelle test skal holde stadiet mellem api-kald, på baggrund af at applikationslaget er implementeret som en RESTful API – hvilket resulterer i en stateless web api. Ydermere kræver det delvist også at systemet under test er blevet udviklet med end-to-end test i mente. Dette er desværre ikke sagen for systemet under test i dette projekt. Ydermere vurderes det også at forretningslogikken ikke kan valideres fuld ud, med et testniveau der ikke interagerer fra samme perspektiv som slutbrugeren.

Derfor er brugergrænsefladen valgt som indgangsvinklen for testene berørt i dette projekt. Dog rejser det nogle andre komplekse vanskeligheder, end en indgangsvinkel mod applikationslaget. Dette vil blive gennemgået senere i denne rapport.

TEST MILJØ & STADIE

Det skal overvejes hvorledes miljø og stadie håndteres i en funktionel acceptance test kontekst.

BetterBoad's nuværende software miljøer inkluderer: Produktion, Dev og den individuelle udviklers lokale miljø. Dev er et distribueret miljø på Azure, som kan modtage ændringer fra den individuelle udvikler – uden at skulle have bekræftelse gennem et pull request. Til formål for at teste logik i et distribueret miljø. Derfor skal det understreges at Dev-miljøet, ikke kan garanteres som et stabilt testmiljø.

Derfor kunne det være en mulighed at distribuere et staging miljø mellem Dev og Produktion. Til formål for at have en målskive for atomisk isoleret tests af forskellige niveauer. Dog, ville dette være en kompleks affære. Da størstedelen af den eksisterende infrastruktur står udokumenteret – altså manglende infrastructure-as-code. På baggrund af at det er blevet kreeret gennem Azure's brugergrænseflade. Dette betyder at det ville være vanskeligt at genskabe et miljø der er produktionsnært. Ydermere at vedligeholde stadiet for sådan et staging miljø, til at være så produktionsnært som muligt.

Det skal dog understreges at det er vigtigt at have et produktionsnært miljø at teste op mod, når konteksten er funktionel acceptance tests. Netop fordi det ideelle formål er at kunne teste slutbrugernes interaktioner med systemet.

På baggrund af rammerne for den eksisterende system og infrastruktur, samt best-practice i funktionel acceptance testing: Vurderes det at test miljøet for dette projekt bør være det eksisterende produktionsmiljø.

Det rejser da følgende udfordring: Hvordan kan der opsættes og nedrives et isoleret og atomisk stadie for disse tests?

Dette er en vigtig strategi at få implementeret. Da det lader testene have reproducerbare resultater. Ydermere, hvis isolationen er implementeret korrekt, giver det mulighed for at kunne eksekvere testene parallelt – så flaskehalsen ikke vil være mængden af tests.

Dog, på baggrund af at målskiven for testene er produktionsmiljøet. Betyder det at der er en afhængighed af den eksisterende infrastruktur. Hvilket resulterer i at test stadietopsætningen ikke kan være afhængig af at skulle udrulle en ny database, eller bruge rollback i den eksisterende database.

En anden mulighed for at sætte stadie op, er at udvide applikationslaget på systemet under test, til at kunne opsætte og nedrive stadie til tests. Dette medfører også at acceptance testene vil fungere som validering af applikationslaget.

Dermed vurderes det at API'en i systemet under test, skal være opsætning- og nedrivningsværktøjet for stadie til testene i dette projekt.

Hertil udvikles nogle simple endpoints, der tager ansvar for forretningslogik, væk fra testene:

```

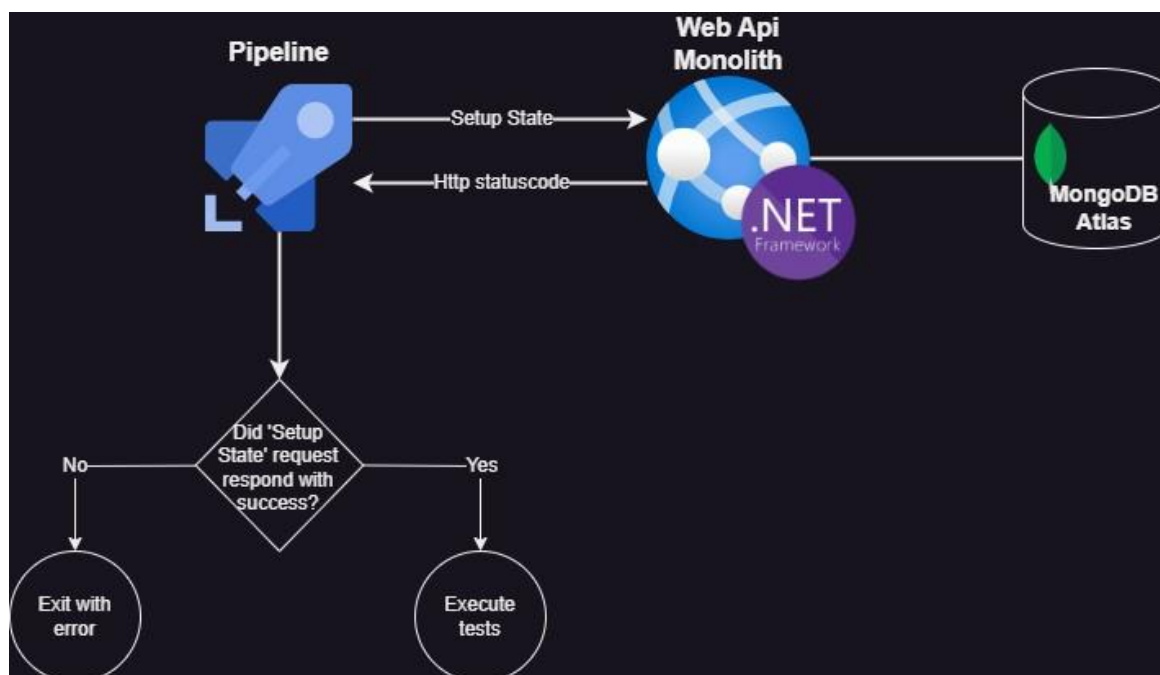
1  [HttpPost]
2  [Route(@"testtransaction/start")]
3  [ApiKeyAuthorization]
4  public async Task<IHttpActionResult> Start([FromBody] TestRunIdentifier runI-
dentifier) {
5      try {
6          await _transactionService.Seed(runIdentifier);
7      }
8      catch (Exception e) {
9          string returnMessage = ExceptionMessageHelper.ConcatenateException-
Message(e);
10         if (e.GetType() == typeof(InvalidTransactionException)) {
11             return BadRequest(returnMessage);
12         }
13         return InternalServerError(e);
14     }
15     return StatusCode(HttpStatusCode.Created);
16 }
17
18 [HttpDelete]
19 [Route(@"testtransaction/cleanup")]
20 [ApiKeyAuthorization]
21 public async Task<IHttpActionResult> CleanUp([FromBody] TestRunIdentifier run-
Identifier) {
22     try {
23         await _transactionService.CleanUp(runIdentifier);
24     }
25     catch (Exception e) {
26         string returnMessage = ExceptionMessageHelper.ConcatenateException-
Message(e);
27         if (e.GetType() == typeof(InvalidTransactionException)) {
28             return BadRequest(returnMessage);
29         }
30         return InternalServerError(e);
31     }
32     return StatusCode(HttpStatusCode.NoContent);
33 }

```

Figur 10: Test Transaction Endpoints

Logikken er at have et enkelt endpoint der skal kaldes før nogle tests bliver eksekveret. Som da tager hånd om at opsætte et nødvendigt stadie i systemet, for at kunne eksekvere testene. Dernæst kan HttpDelete endpointet kaldes for at rydde op efter test eksekvering. Denne implementering medfører at testene, isoleret set, ikke behøver noget ansvar for at kreere eller oprydde stadie.

Derfor forestilles der følgende kommunikationssnitflade til at sætte stadie op for tests:



Figur 11: Test stadiet setup

På baggrund af at begge endpoints udfører kritisk forretningslogik. Som, i en normal kontekst, ville skulle kræve autorisering fra en superadmin i systemet. Er blevet dekoreret med en `ApiKeyAuthorization` attribut. Som er en implementering af .NET's `AuthorizationFilterAttribute`, til formål for kun at tillade autoriseret kald til disse endpoints. Hvilket bliver opretholdt ved at have en privat nøgle gemt i API'et og en key vault:

```

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited = true,
AllowMultiple = true)]
public sealed class ApiKeyAuthorizationAttribute : AuthorizationFilterAttribute
{
    private readonly string _apiKey;

    public ApiKeyAuthorizationAttribute()
    {
        _apiKey = ConfigurationManager.AppSettings["TransactionApiKey"];
    }

    public override void OnAuthorization(HttpContext actionContext)
    {
        IEnumerable<string> apiKeyHeaders = Enumerable.Empty<string>();
        if (!actionContext.Request.Headers.TryGetValue("X-API-Key", out apiKey-
Headers))
        {
            actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized, "Mis-
sing API Key");
            throw new UnauthorizedAccessException("Missing API Key.");
        }

        var apiKey = apiKeyHeaders.FirstOrDefault();
        if (!IsValidApiKey(apiKey))
        {
            actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized, "Inva-
lid API Key");
            throw new UnauthorizedAccessException("Invalid API Key.");
        }
    }
}

```

```

    public override Task OnAuthorizationAsync(HttpContext actionContext, CancellationTok
    en cancellationToken) {...}

    private bool IsValidApiKey(string apiKey) {...}

    public Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(HttpContext actionCon
    text actionContext, CancellationToken cancellationToken, Func<Task<HttpResponseMessage>>
    continuation) {...}

    private async Task<HttpResponseMessage> ExecuteAuthorizationFilterAsyncCore(Http
    pActionContext actionContext, CancellationToken cancellationToken, Func<Task<HttpRespon
    seMessage>> continuation) {...}
}

```

Figur 12: Api nøgle authorization attribut

Begge endpoints modtager data gennem http body, i form af klassen TestRunIdentifier:

```

public class TestRunIdentifier
{
    public string Id { get; private set; } = string.Empty;
    public string CompanyId { get; private set; } = string.Empty;

    public TestRunIdentifier(string id, string companyId)
    {
        Id = id;
        CompanyId = companyId;
    }
}

```

Figur 13: Test Transaction Endpoint HTTP body domæne klasse

Klassen foroven fremviser det eneste data påkrævet at opsætte stadie til acceptance tests. Id propertyen er til hensigt at illustrere den specifikke kørsel af en test eksekvering. Til formål for at kunne aflæse en specifik testeksekverings interaktioner med systemet, skulle nogle tests fejle eller lignende. Således at man kan komme til bunds i hvorfor en test potentielt fejlede.

CompanyId propertyen refererer til det specifikke firma der vil blive kreeret under testen – og opryddet derefter. Det skal understreges, at alt domænelogik tilnærmelsesvis kan kobles til ét eller flere firmaer. Dermed giver det en høj anskuelighed, at logge det specifikke test firma der vil blive oprettet og interageret med under en testeksekvering.

Seed metoden, som kaldes fra HttpPost endpointet nævnt tidligere, tager højde for at opsætte stadie til alle acceptance tests. Det kan anskues som at gå i strid mod 'single responsibility principle' fra SOLID. Dog, giver det en klar indikator om der er 'grønt lys' for at kunne eksekvere acceptance tests op mod systemet, før nogle tests er kørt. I stedet for at skulle lave isoleret stadie opsætning og nedrivning for hver test case.

Test data setup logikken er implementeret således at det er unikke data hver gang det køres.

```

    /// <exception cref="InvalidTransactionException"></exception>
    public async Task Seed(TestRunIdentifier runIdentifier)
    {
        ValidateCaller();

        ValidateInput(runIdentifier, out Guid identifier, out ObjectId companyId);

        // Create Company
        var existingCompany = await _companyService.SearchAsync(
            new CompanySearchDto { CompanyName = COMPANY_NAME_PREFIX + runIdentifier
});

        if (existingCompany.Any())
        {
            throw new InvalidTransactionException($"A company with the name '{COMPANY_NAME_PREFIX + runIdentifier.Id}' already exists.");
        }

        var company = new Company {...};

        company = await _companyService.Create(company, ObjectId.Empty.ToString());

        // Create Board
        var board = new Board { BoardName = "E2E Test Board - " + identifier.ToString("N") };

        board = await _boardService.CreateBoard(companyId.ToString(), board, ObjectId.Empty.ToString());

        var boardId = board._id;

        // Create Users
        var userBase = new UserBase {...};

        var member = new MemberBuilder(userBase).Build();

        _ = await AddPresetMemberToBoard(companyId.ToString(), boardId.ToString(), member, identifier);
    }

```

Figur 14: Test seed metode implementering

Logikken foroven er, som benævnt tidligere, kritisk forretningslogik – især i et produktionsmiljø. Dermed indføres metoden `ValidateCaller` i starten af eksekveringen. Som håndhæver en sikkerhedsmekanisme baseret på reflection: At kun klassen selv, controlleren der på nuværende tidspunkt kalder `Seed` og `Cleanup` metoden, og den tilsvarende unit test klasse, må invokere metoder på denne klasse. Dette sørger for at der i fremtidig udvikling eller drift, ikke ved et uheld bliver koblet til denne kritiske logik.

Et omfattende sekvensdiagram for første iteration af test stadie setup kan findes i bilag 1 (Test State Setup Sekvensdiagram).

AUTOMATISERET WEB INTERAKTION

Som benævnt under 'Testniveau & -Type' afsnittet, skal implementeringen for kvalitetssikringen i dette projekt kunne interagere med en web-baseret brugergrænseflade.

Dette scope har mange løsningsstrategier, og en lang række værktøjer til at hjælpe på vej. Dertil er der blevet analyseret på 4 forskellige værktøjer: Playwright, Nightwatch, Cypress og Selenium.

PLAYWRIGHT

Playwright er et cross-browser automatiserings framework til at end-to-end teste webapplikationer (playwright.dev, 2024). Det supporterer at skrive playwright tests i JavaScript, Python, Java og .NET.

Playwright er en af de end-to-end test værktøjer der eksekverer i en selv-isoleret tilstand. Hvilket betyder at det ikke afhænger af en anden infrastruktur til at levere browser webdrivere, som for eksempel Selenium gør. Det resulterer i at man kan have en temmelig kompakt CI / CD pipeline til at eksekvere ens playwright tests:

```
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup dotnet
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: 8.0.x
      - run: dotnet build
      - name: Ensure browsers are installed
        run: pwsh bin/Debug/net8.0/playwright.ps1 install --with-deps
      - name: Run your tests
        run: dotnet test
```

Figur 15: Playwright tests i GitHub Actions (Playwright, 2024)

Nogle kerneværktøjer som playwright udstiller er Codegen og Trace Viewer. Codegen genererer test kode baseret på video optaget handlinger. Dette kan lette arbejdet i at skulle skrive boilerplate kode. Trace Viewer fanger information fra en fejlet test, alt fra screencast til DOM snapshot (Playwright, 2024).

Dog er denne salgspitch, at playwright er enormt selv-isoleret, kan også være dens største udfordring. Som det kan ses figur 9, kræver det at CI serveren, der eksekverer disse tests, har de nødvendige software afhængigheder installeret. Til formål for at kunne starte browsere op. Denne afhængighed kan godt løses ved at implementere en pipeline strategi der udnytter playwright's docker image

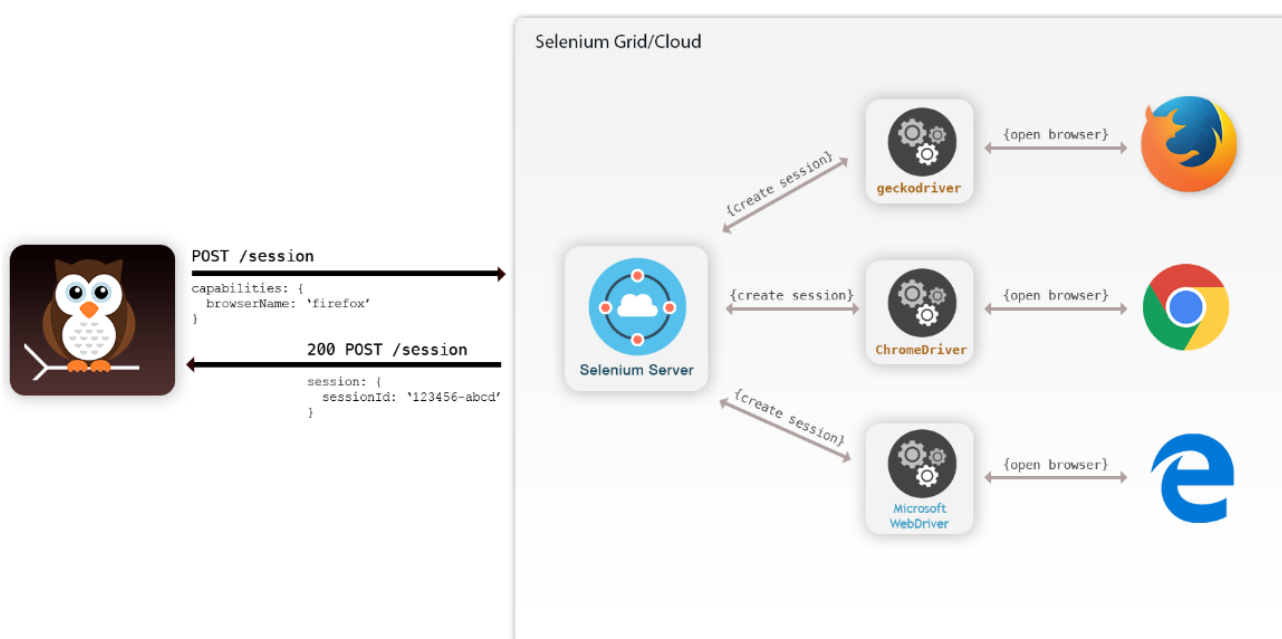
'mcr.microsoft.com/playwright/dotnet'. Men begge disse strategier lægger krav på høj compute kraft fra CI serveren. Da browser emulering kræver regelmæssig høj memory. Og playwright afhænger af at kunne instantiere browser kontekster i den samme eksekverings kontekst som testene. Med andre ord: Playwright kører browser sessions på samme maskine som eksekverer playwright tests.

Et positivt aspekt af ovenstående er dog at der er minimal opsætning påkrævet, før der kan fås udbytte af acceptance tests.

En unik feature ved playwright er måden parallelisering af tests er implementeret på. Netop at browser sessions bliver lavet i inkognito mode, og hver test case får en fane i den givne browser session. I stedet for en browser session til hver test case. Dette giver en bedre udnyttelse af ressourcer, men kan komme i strid mod test isolering. Især hvis ens test case afhænger af cookies eller flere faner.

NIGHTWATCH

Nightwatch er et javascript framework til at automatisere end-to-end tests på webapplikationer (Nightwatch, 2024). Nightwatch er anderledes fra Playwright, i det omfang at det supporterer at teste op mod distribuerede browser sessions. Det integrerer per standard med Selenium Grid – som vil blive uddybet under 'Selenium' afsnittet længere nede i rapporten – og cloud-baseret testing platformer.



Figur 16: Nightwatch tests op mod distribuerede browser sessions (Nightwatch, 2024)

Denne infrastruktur forskel giver mulighed for en langt bedre koordineret skaleringsplan for test eksekvering. På baggrund af at der er en afkobling mellem test projektet indeholdende Nightwatch scripts, og infrastrukturen der sørger for browser sessions. Sæt man har 10 forskellige test cases skrevet i Nightwatch, der ønskes at kører parallelt i 3 forskellige browsere: Lægges der ikke krav på at CI Serveren skal koordinere 30 browser sessions parallelt. Derimod kan der implementeres en skaleringsstrategi på det de distribuerede browser sessions.

Dog, at implementere en arkitektur, som vist på figur 10, vil resultere i en høj netværkstrafik mellem Nightwatch testene og de distribuerede browser sessions. Sammenlignet med strategien fremhævet med Playwright.

Nightwatch supporterer kun at skrive tests i JavaScript. Der kan dog kompiles til JavaScript fra andre sprog. Men på tidspunktet at dette bliver skrevet, findes der for eksempel ingen værktøjer til .NET der kan kompilere Nightwatch tests skrevet i C# til JavaScript.

CYPRESS

Cypress er unik fra mange andre lignende webapplikation test værktøjer. I det omfang at det ikke bruger webdriver protokollen til at udføre sine tests. I stedet kører det direkte i en browser. Hvilket resulterer i test eksekveringen ofte er hurtigere, sammenlignet med værktøjer der afhænger af webdriver. Ydermere, betyder det at Cypress kan interagere med netværkskald mellem browseren og en api. Hvoret webdriver afhængig test værktøj kun kan se det renderede html, kan Cypress opfange netværkskald til- og fra browseren. Det kan udnyttes til for eksempel at mocke netværks fejlkoder til applikationen under test. Til formål for at se hvorledes applikationen håndterer disse udfald.

På baggrund af arkitekturen bag Cypress betyder dog også at det skalerer på samme vis som Playwright. Som resulterer i at CI serveren står for skaleringen, sæt at tests skal eksekveres parallelt.

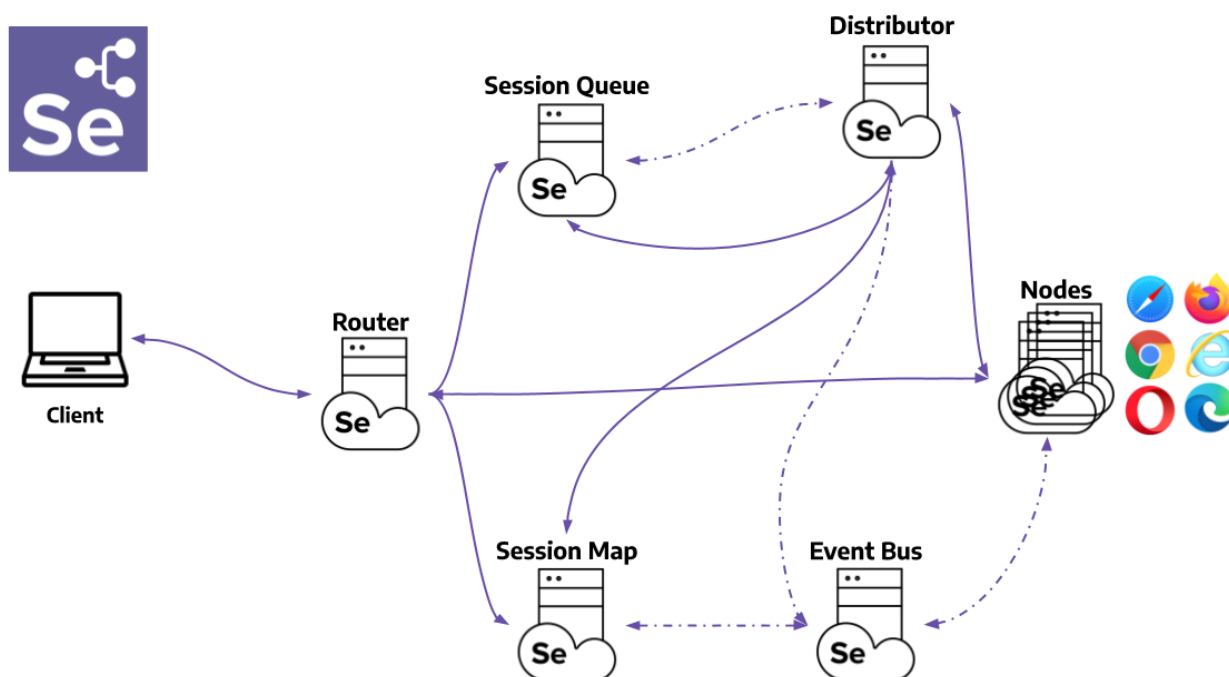
Cypress skiller sig også ud ved at tilbyde en enterprise cloud løsning, der tager sig af mange udfordringer ved selv at skulle sætte end-to-end test infrastruktur op.

SELENIUM

Selenium er den ældste, men også mest udbredte, af disse webapplikation test værktøjer. Selenium tests kan skrive i stort set alle programmeringssprog, især fordi hele Selenium projektet er open-source.

Selenium arkitekturen bygger på at eksekvere scripts over http op mod et Selenium Grid. Som står for at håndtere interaktion mellem test scripts og browser sessions. Ydermere tager det ansvaret for at skalere infrastruktur og parallelisere test eksekvering.

Selenium Grid er formålsbygget til containerisering og cloud-distribueret skalerbarhed (Selenium, 2024). Som bliver fremhævet af dets systemdiagram:



Figur 17: Selenium Grid infrastruktur komponenter (Selenium, 2024)

Cypress vurderes til at være det mest sofistikerede værktøj, især til at se metrikker for sine tests. Dets unikke måde at interagere med en webapplikation på – direkte i browseren – er en stor fordel. Alligevel, vurderes Selenium til at være det oplagte værktøjsvalg til dette projekt. På baggrund af dets ekstensive dokumentation og det faktum at det er så udbredt. For ikke at nævne Selenium Grid's tiltag til skalerbarhed og parallelisering. Det bundet ud i en langt mere strømlinet udviklingsproces. Fordi størstedelen af ressourcekraften påkrævet at gennemføre ens acceptance tests, er distribueret i forvejen med Selenium Grid.

Det skal understreges, at der menes en self-hosted Selenium løsning med dette værktøjsvalg. Og ikke en af de cloud-baseret løsninger. Fordi et væsentligt delproblem for dette projekt er at dokumentere den automatiseret drift.

AZURE SERVICE TYPE FOR SELENIUM GRID

Med valget om at distribuere et Selenium Grid i en self-hosted facon. Skal det analyseres, hvilken Azure Service type er den mest passende til at distribuere det. For bedst at udnytte Selenium Grid's skalering og parallelisering, er det væsentligt at kunne udrulle det som en cloud-native arkitektur.

Cloud-native kan defineres ud fra følgende kvote: *"Cloud-native architecture and technologies are an approach to designing, constructing, and operating workloads that are built in the cloud and take full advantage of the cloud computing model."* (Cloud Native, 2023).

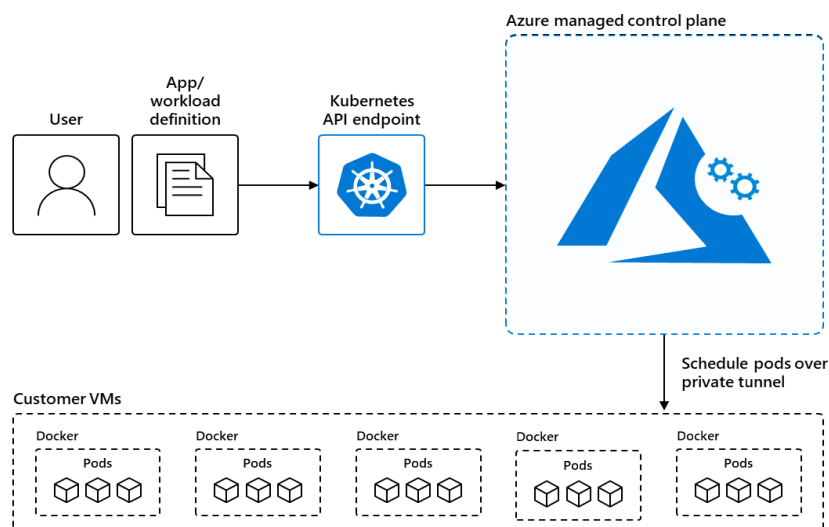
Dermed er det vigtigt at den valgte Azure Service kan udnytte 8. princip i Den Tolvte-Faktor App (Twelve Factor App, 2017): En industristandard for at skalere forskellige processer.

Fordi det vurderes, ud fra Selenium Grid's arkitektur på figur 11, at Nodes komponenterne skal være i stand til skalere baseret på mængden af acceptance tests.

Baseret på principperne for cloud-native, vurderes det at der er 3 muligheder for distribuering i Azure: Azure Kubernetes Service og Azure Container Apps.

AZURE KUBERNETES SERVICE

Azure Kubernetes Service eller AKS som forkortelse, er en managed Kubernetes service. Servicen reducerer kompleksiteten der kan være ved at konfigurere Kubernetes. Det gør den for eksempel ved automatisk at kreere og konfigurere control planen i Kubernetes (Azure Kubernetes, 2024):



Figur 18: Overblik over AKS (Azure Kubernetes, 2024)

AKS kan også blive brugt med Helm charts, som ofte er brugt til at manage applikationerne i et cluster. Denne integration fungerer dog bedst hvis Azure Container Registry bruges som repository for ens Helm charts.

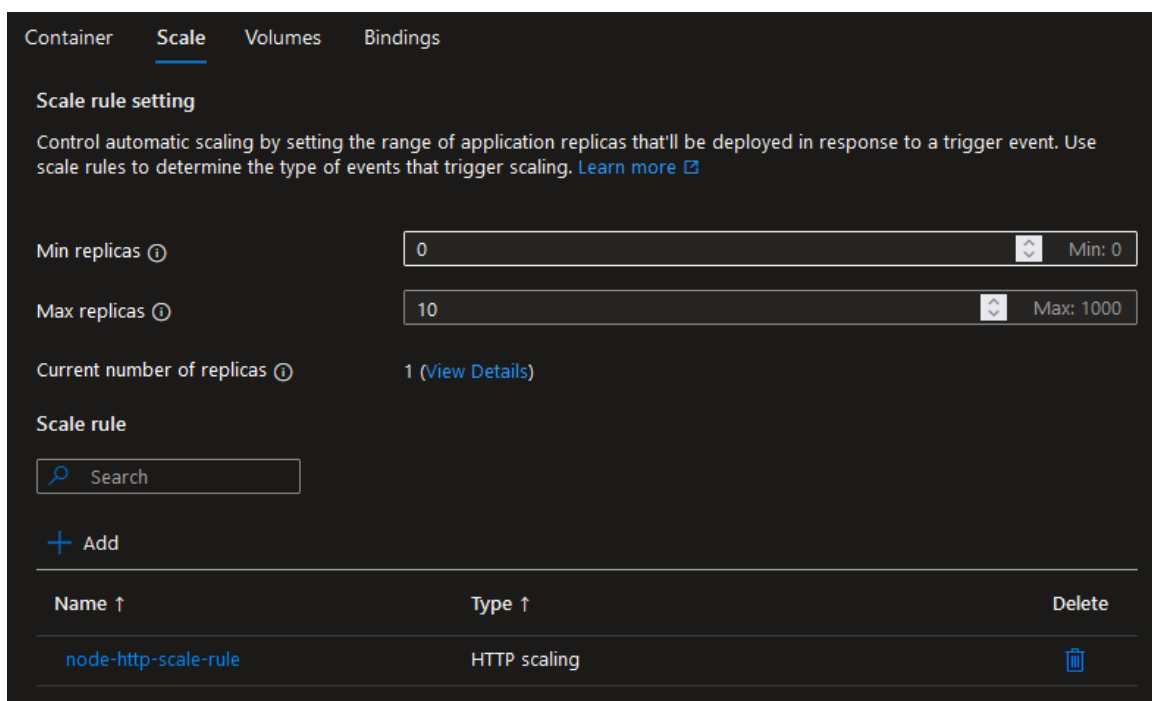
Med AKS får man kernegevinsterne der er ved et orkestreret container miljø: Fejltolerance og høj tilgængelighed. Dog kræver det stadig en hel del opsætning at få udrullet, på baggrund af alle de AKS features som skal tages hånd om. Samt den implicitte kompleksitet der er ved Kubernetes clustre.

Opsummeret set er AKS en god strategi, hvis en løsning kræver en skræddersyet konfiguration og skalering.

AZURE CONTAINER APPS

Azure Container Apps, eller ACA som forkortelse, abstraherer meget af den underliggende infrastruktur management. Det er stadig en managed Kubernetes service. Men påkræver betydeligt mindre opsætning i forhold til AKS. Ulempen er selvfølgelig at der mistes en del kontrol over den underliggende orkestrering.

ACA udsteder dog nogle simple auto-scaling regelopsætninger. Som for eksempel at skalere en specifik service, baseret på indgående http trafik eller events. Dette betyder der fås mange af de samme gevinster der er ved et konfigureret Kubernetes cluster, i henhold til dynamisk skalering.



Figur 19: ACA GUI konfiguration af skalering

På baggrund af at Selenium Grid ikke har så mange komponenter, vurderes det at Azure Container Apps er en tilstrækkelig service type til at distribuere det. Det vurderes også at en ACA løsning vil kræve mindre udviklingstid med hensyn til drift. Ydermere bliver ACA promoveret som service type af Microsoft, ved at udstede nogle værktøjer der letter udrulningsprocessen – Som vil blive berørt i afsnittet forneden.

UDRULNING AF SELENIUM GRID

Overordnet set vurderes det at der er 3 forskellige måder at udrulle et Selenium Grid til ACA på:

- Gennem Azure Portalens brugergrænseflade.
- Gennem en CI / CD pipeline med Infrastructure-as-Code.

- Gennem et CLI værktøj der tager udgangspunkt i en lokal orkestrering.

Det forkastes at analysere udrulning gennem brugergrænseflade. Fordi det går stridt imod denne opgaves formål at dokumentere den automatiseret drift.

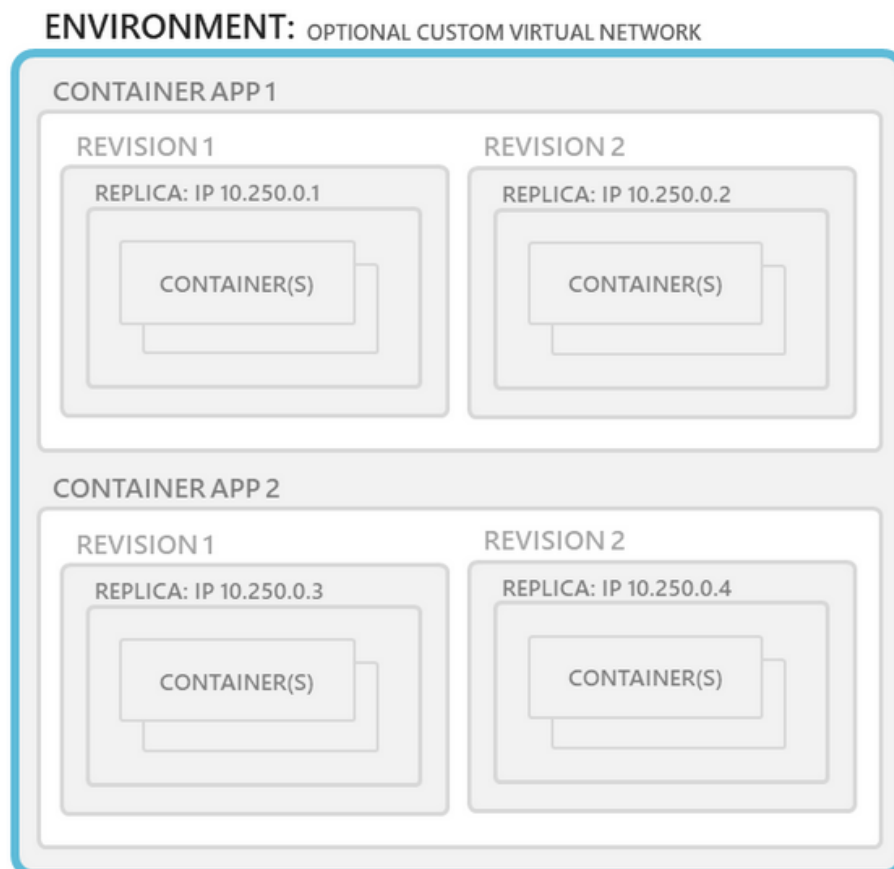
Der er andre udrulningsmetoder inden for Azure. Men vurderes til ikke at opfylde opgavens afgrænsning eller firmaets tech stack: Som for eksempel udrulning gennem Python.

INFRASTRUCTURE AS CODE

Bicep vurderes til at være det mest passende IaC værktøj til dette projekt. Baseret på det faktum at BetterBoard så småt er begyndt at bruge det som IaC værktøj, i kontrast til andre som Pulumi eller Terraform.

Bicep kan udnyttes til at beskrive et ACA environment indeholdende de nødvendige Selenium Grid komponenter. Selenium udsteder nemlig de forskellige Grid komponenter som images på DockerHub, der da kan blive brugt i en container kontekst.

Dog kan det være komplekst at få opsat en multi-container applikation op i ACA, baseret på blot dokumentation og kildekode. Ydermere er en udfordring blandt andet de netværksregler som der er i ACA. Container applikationer i ACA kan ikke, uden konfiguration, kommunikere med hinanden på tværs af et internt virtuelt netværk. For eksempel, med udgangspunkt i figuren forneden, kan Container App 1 ikke kommunikere med Container App 2 inden for samme virtuelle netværk, uden konfiguration.



Figur 20d: ACA arkitektur (Container Apps, 2023)

Dette er stik modsat af et traditionelt Docker bridge netværk, som tillader fuld kommunikation mellem containere på samme netværk. Ydermere, er Selenium Grid's dokumentation baseret på Docker.

Det vurderes at det er en kompleks opgave, med risiko for meget 'trial and error', i manuelt at beskrive en Selenium Grid infrastruktur i ACA med Bicep. Dog, vurderes det at Bicep templates er en god strategi til at dokumentere den automatiseret drift.

Derfor undersøges om dokumentationen af infrastrukturen kan automatiseres.

.NET ASPIRE

.NET Aspire er, på tidspunktet at dette skrives, et nyt værktøj udstedet til .NET værktøjskassen. Aspire kan opsummeret set beskrives ved Microsoft's egen kvote: *" .NET Aspire is an opinionated, cloud ready stack for building observable, production ready, distributed applications."* (Aspire, 2024).

Det er et værktøj der kan bruges til at orkestrere lokale applikationer og containere. Dette kan udnyttes yderligere, til at benytte Aspire's værktøjskasse til at udrulle en orkestrering op mod ACA. Aspire er nemlig bygget på det fundament at udrulle til ACA.

Eksempelvis kan der tages udgangspunkt i en Docker compose fil der beskriver et Selenium Grid:

```
services:

  selenium-hub:
    image: selenium/hub:4.18.0-20240220
    container_name: selenium-hub
    ports:
      - "4444:4444"
    networks:
      - selenium-grid

  chrome:
    image: selenium/node-chrome:4.18.0-20240220
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_OVERRIDE_MAX_SESSIONS=true
      - SE_NODE_MAX_SESSIONS=3
    networks:
      - selenium-grid

  edge: {...}

  firefox: {...}

networks:
  selenium-grid:
    name: selenium-grid
    driver: bridge
```

Figur 21: Selenium Grid docker compose

Selenium/hub imaget er en samling af alle Selenium komponenterne bortset fra Selenium Nodes.

Resultatet af den viste Docker compose kan reproduceres i Aspire. Aspire kan nemlig konfigureres til at starte container applikationer, baseret på en Container Engine.

Hernæst kan Aspire frameworket udvides til at beskrive container applikationer:

```
public class SeleniumHubResource(string name, string? endpoint = null) : ContainerResource(name, endpoint), IResourceWithServiceDiscovery
{
    internal const string Image = "selenium/hub";
    internal const string Tag = "4.18.0-20240220";
    internal const string HttpEndpointName = "http";

    private EndpointReference? _endpointReference;

    public EndpointReference HttpEndpoint => _endpointReference ??= new(this, HttpEndpointName);

    public ReferenceExpression ConnectionStringExpression => ReferenceExpression.Create($"SE_EVENT_BUS_HOST={HttpEndpoint.Property(EndpointProperty.Host)}");
}
```

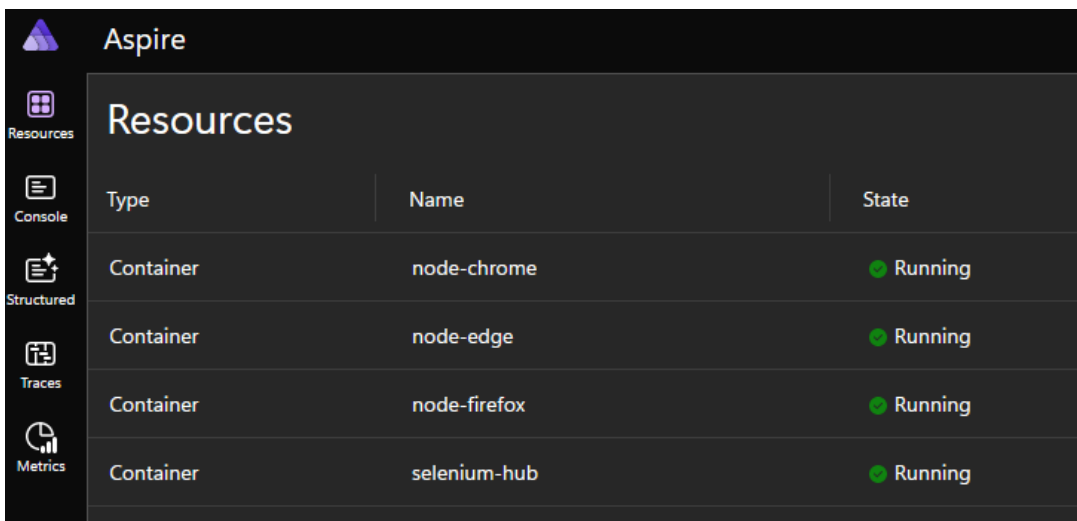
Figur 22: Selenium Hub container beskrevet i .NET Aspire

Klassen kan dernæst blive refereret til i en extension metode til Aspire builderen:

```
var resource = builder.AddResource<SeleniumHubResource>(seleniumHub)
    .WithEndpoint(4442, 4442, "tcp", "publish")
    .WithEndpoint(4443, 4443, "tcp", "subscribe")
    .WithHttpEndpoint(4444, 4444, "webdriver")
    .WithExternalHttpEndpoints()
    .WithImage(SeleniumHubResource.Image, SeleniumHubResource.Tag);
```

Figur 23: Selenium Hub container orkestreret i .NET Aspire

Med alle ressourcerne beskrevet, kan Aspire løsningen køres. Og resultatet er det samme som Docker compose filen vist tidligere:

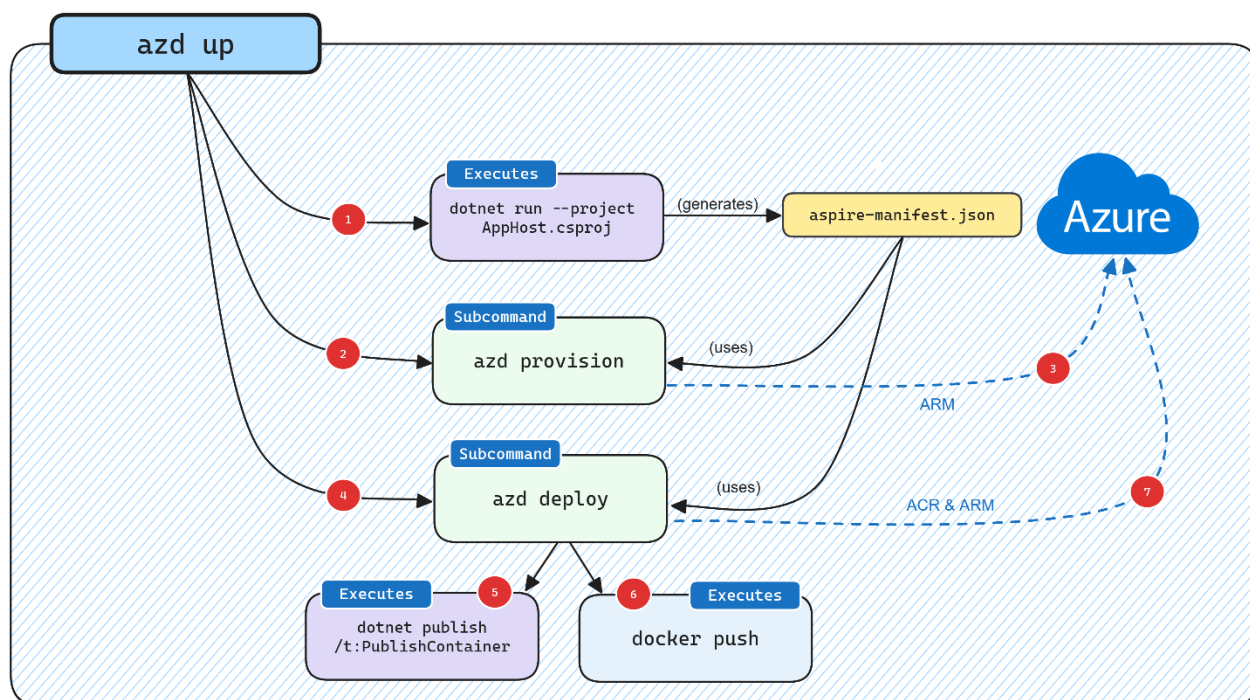


Type	Name	State
Container	node-chrome	Running
Container	node-edge	Running
Container	node-firefox	Running
Container	selenium-hub	Running

Figur 24: Selenium Grid i containere på Aspire dashboard

Bare det at orkestrere containere lokalt er ikke noget substantielt. Men Aspire kommer med indbygget værktøjer til at konfigurere OpenTelemetry for vedrørte applikationer. Således at der kan fås dybdegående indsigt i ens applikationer lokalt, såvel som distribueret på Azure.

Ydermere, udsteder Aspire værktøjer til at genskabe den samme orkestrering og konfigurerings på ACA såvel som lokalt. For eksempel ved brug af Azure Developer CLI:



Figur 25: Azure Developer CLI integration til .NET Aspire projekter (Aspire Deployment, 2024)

Opsummeret set, viser figuren foroven en række CLI kommandoer, der udruller et lokalt Aspire projekt til Azure Container Apps. Den største gevinst ved dette er at dokumentationen af infrastrukturen bliver automatiseret.

Ultimativt betyder det at det er en lettere overgang at gå fra et lokalt miljø til et produktionsmiljø. For ikke at nævne at der er færre elementer der skal driftes. Baseret på det faktum at orkestreringen og konfigureringen i et Aspire projekt, vil blive udtrykt ligeledes i ACA. I kontrast til et scenarie, hvor man ville skulle drifte et lokalt miljø – eksempelvis en Docker compose med referencer til Azure Container Registry – og noget IaC i form af Bicep op mod Azure.

Dog vurderes det at kommandoen 'azd up' abstraherer meget af kontrollen væk fra udvikleren. Derfor benyttes følgende CLI kommandoer til at generere Bicep filer baseret på Aspire projektet:

```
> azd config set alpha.infraSynth on
> azd infra synth
```

Figur 26: Azure Developer CLI provision subcommands

En af ressourcerne genereret er for eksempel Selenium Hub komponenten, nu dokumenteret i en Bicep fil:

```
resource seleniumHub 'Microsoft.App/containerApps@2023-05-02-preview' = {
  name: 'selenium-hub'
  location: location
}
```

```

properties: {
  environmentId: containerAppEnvironment.id
  configuration: {
    activeRevisionsMode: 'Single'
    ingress: {
      external: true
      targetPort: 4444
      transport: 'http'
      allowInsecure: false
      additionalPortMappings: [
        {
          external: false
          exposedPort: 4443
          targetPort: 4443
        }
        {
          external: false
          exposedPort: 4442
          targetPort: 4442
        }
      ]
    }
  }
}
template: {
  containers: [
    {
      image: 'selenium/hub:4.18.0-20240220'
      name: 'selenium-hub'
    }
  ]
  scale: {
    minReplicas: 1
    maxReplicas: 1
  }
}
tags: union(tags, {'aspire-resource-name': 'selenium-hub'})
}

```

Figur 27: Selenium Hub container beskrevet som en Azure Container App i Bicep

Resultatet af dette er at Bicep dokumentationen kan bruges som udgangspunkt til udrulning i en CI / CD pipeline. Ydermere giver det rådighed for at kunne supportere state management, idet der nu fås en historik over konfigurationer til denne multi-container applikation. På baggrund af at kildekode ændringer til Aspire projektet kan kobles til en automatisk proces om at generere Bicep kode.

Opsummeret set, henter Aspire værktøjskassen tid fra drift og infrastruktur opsætning til gavn for udvikling.

Bicep koden genereret foroven – i sin helhed – beskriver følgende ressource gruppe og dets ressourcer i Azure:



Figur 28: Selenium Grid udrullet i Azure Container Apps

Hvordan infrastrukturen fra figur 16 blev udrullet, vil blive uddybet under afsnittet 'CI / CD'.

TESTS

Det skal undersøges hvad der bør testes.

Derfor er der blevet samarbejdet med BetterBoard's CEO, for at identificere funktioner og brugerscenarier der er værdifulde at teste automatisk.

Kernefunktionerne der vurderes at skulle testes, blev dokumenteret som use cases – som kan findes i bilag 2 (use cases).

En af disse use cases vil der blive taget udgangspunkt i, i denne rapport:

UC 03 – Møde Oprettelse

- Ud fra forsiden på en portal: Klik på **Opret møde**, i kolonnen til venstre.
- Verificer at man **ikke** kan oprette mødet før formularen er udfyldt med de nødvendige informationer.
 - F.eks. test med ukorrekt tidspunkt: Et uangivet tidspunkt, eller starttidspunkt **efter** sluttidspunkt.
- Udfyld formularen tilstrækkeligt.
 - Vælg én eller flere test brugere at sende notifikation til.
- Verificer at mailnotifikationen bliver afsendt.
 - Her tænkes at verificere om [sparkpost](https://sparkpost.com/) responderer med success HTTP statuskode, på vores api-kald til deres service.

Figur 29: Use Case for møde oprettelse på bestyrelsesportalen

Sparkpost, som benævnt i use casen foroven, er den valgte smtp-service til at levere e-mails på vegne af BetterBoard's software systemer. Det forventes at en e-mail bliver korrekt afsendt, hvis http-request'en til Sparkpost returnerer en success http statuskode.

IMPLEMENTERING

Implementeringen af disse tests blev skrevet som C# kode ved udnyttelse af Selenium WebDriver nuget pakken. Det blev vurderet at XUnit var en passende projekttipe til at indeholde disse tests. En faktor var at det gav mulighed for at udnytte 'dotnet' cli værktøjet i en CI / CD pipeline. Med kommandoer som 'dotnet test', kan det give et dybdegående indblik i enhver test metodes flow.

Fordi systemet under test kræver en bruger der er logget ind, til at udføre de identificeret use cases, blev der udarbejdet en cookie baseret login funktionalitet. Det vurderes nemlig at det kun er nødvendigt at teste login funktionaliteten gennem brugergrænsefladen én gang for en hel test suite.

```
public static bool Login(RemoteWebDriver driver, Uri targetUri, UserCredentials userCredentials)
{
    ILoginWindow loginWindow = new LoginWindow(driver, targetUri);

    loginWindow.Navigate();

    if (_cookies is not null)
    {
        driver.Manage().Cookies.DeleteCookieNamed("ai_session");
        driver.Manage().Cookies.DeleteCookieNamed("ai_user");

        driver.Manage().Cookies.AddCookie(_cookies["ai_session"]);
        driver.Manage().Cookies.AddCookie(_cookies["ai_user"]);

        return true;
    }

    loginWindow.AssertNavigation();

    loginWindow.Login(userCredentials);
    loginWindow.AssertLogin(userCredentials);

    ICookieJar jar = driver.Manage().Cookies;
    Cookie session = jar.GetCookieNamed("ai_session");
    Cookie user = jar.GetCookieNamed("ai_user");
    _cookies = new Dictionary<string, Cookie>() {...};

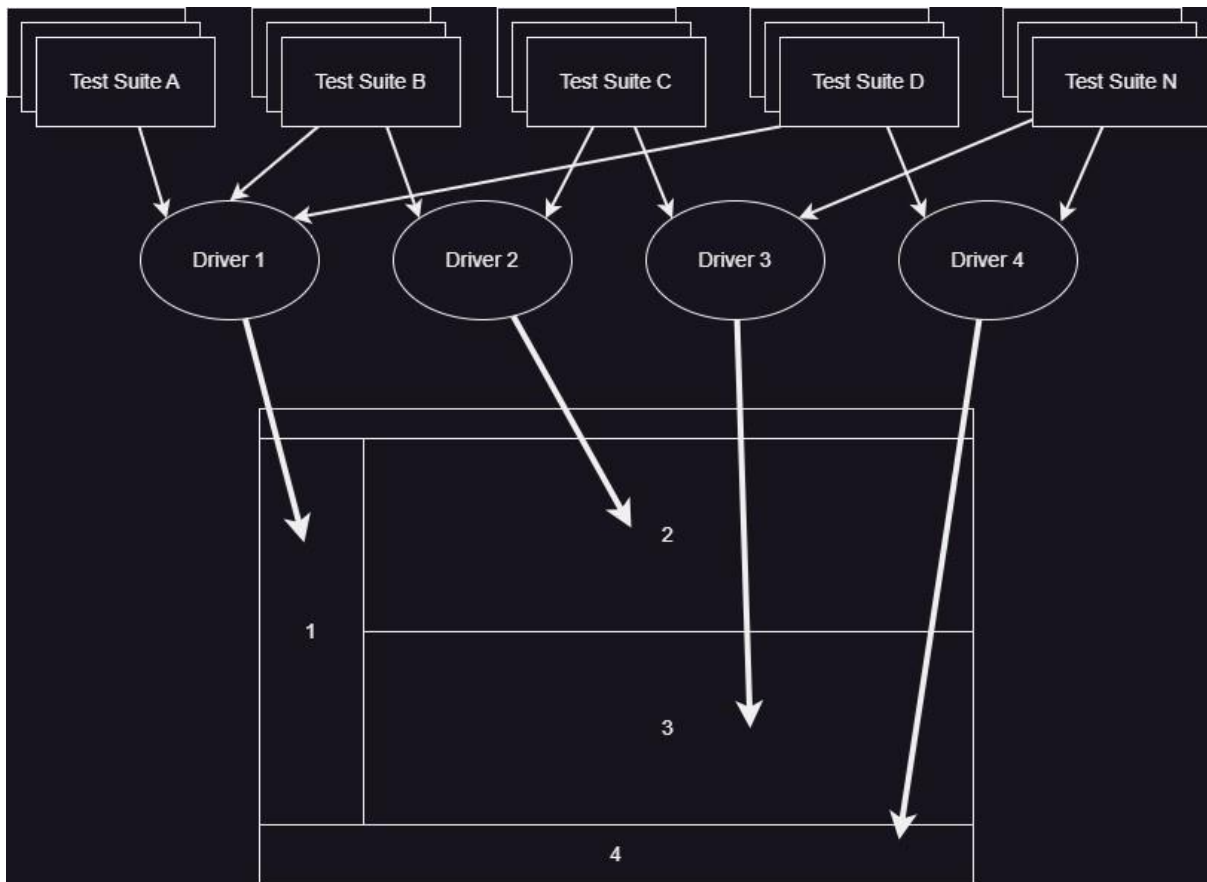
    return false;
}
```

Figur 30: Login implementering

Metoden foroven forventes at kaldes i hver test case. Men brugergrænseflade interaktionen for at logge ind, vil kun eksekvere én gang – på baggrund af at 'static' keywordet bliver udnyttet. Efterfølgende kald vil få de nødvendige cookies, i deres browser sessions, for et succesfuldt login fra første kald.

Denne implementering sparer værdifuld tids eksekvering for test suiten, især i takt med test suiten vokser med flere test cases.

Som det kan ses på kodes eksemplet foroven, bliver det gjort brug af en 'LoginWindow' klasse. Logikken for at interagere med brugergrænsefladen er samlet i en række klasser af denne art, som følge designmønsteret 'Winow Driver Pattern' – som bliver uddybet i bogen *Continuous Delivery* (Humble & Farley, 2011).



Figur 31: Brugen af Web Driver mønsteret i acceptance testing

Opsummeret set, giver designmønsteret et abstraktionslag mellem ens acceptance tests og brugergrænsefladen for systemet under test. Dette resulterer i en lavere kobling og mindre drift hvis brugergrænsefladen skulle ændre sig.

Koden for en test case ender da med at se således ud:

```
driver = new RemoteWebDriver(_gridUri, options);

_ = LoginSession.Login(driver, _targetUri, _testUserCredentials);

IBoardsWindow boardsWindow = new BoardsWindow(driver, _targetUri);

boardsWindow.Navigate();
boardsWindow.AssertNavigation();

boardsWindow.GoToBoard("Test Board", ref _testOutputHelper);
boardsWindow.AssertGotoBoard("Test Board");

INavigationMenuWindow navMenuWindow = new NavigationMenuWindow(driver, _targetUri);
```



```
navMenuWindow.CreateMeeting();
navMenuWindow.AssertMeetingPopup();

IMeetingWindow meetingWindow = new MeetingWindow(driver, _targetUri);

meetingWindow.FillAndConfirmMeeting();
meetingWindow.AssertMeetingConfirmed();
```

Figur 32: Snippet af UC 03 test case implementering

Hver test case kan benytte sig af forskellige 'window drivers'. Og design mønsteret sikrer sig, at der kun skal ændres ét sted, skulle der ske 'breaking changes' til brugergrænsefladen for en given test case.

Logikken for at interagere med brugergrænsefladen gennem en Selenium WebDriver kan ses for neden:

```
public class BoardsWindow(RemoteWebDriver driver, Uri baseUri) : IBoardsWindow
{
    public void GoToBoard(string boardName, ref ITestOutputHelper testOutput)
    {
        // Find board name's index
        var wait = new WebDriverWait(driver, TimeSpan.FromSeconds(20));
        wait.IgnoreExceptionTypes([typeof(NoSuchElementException), typeof(StaleElementReferenceException)]);
        var headers = wait.Until(d =>
        {
            var e = d.FindElements(By.TagName("h5"));
            return e.Any() ? e : null;
        });
        Assert.NotNull(headers);

        {...}
    }
}
```

Figur 33: Snippet af Selenium WebDriver logik i C#

'WebDriverWait' klassen er en essentiel brug af Selenium WebDriver. Da det giver fundament for at kunne udøve eksplicitte ventetider på en given handling. Dette er vigtigt da html elementerne på brugergrænsefladen ikke nødvendigvis er rendered efter en tidligere handling – som at load en url, eller lignende.

'Until' metoden for 'WebDriverWait' klassen udfører lambda metode blokken indtil den ikke returnerer 'false', 'null' eller en exception som ikke er angivet i 'IgnoreExceptionTypes'.

Alle implementeret test cases er indpakket i en 'Task' – C#'s wrapper klasse for en asynkron operation – for at parallelisere den samme test logik for tre forskellige browsere: Firefox, Chrome og Edge:

```
DriverOptions[] driverOptions = AvailableDriverOptions.Get();
Task[] parallelTests = new Task[driverOptions.Length];
for (int i = 0; i < driverOptions.Length; i++) {
    DriverOptions options = driverOptions[i];
    ApplyOptionArguments(options);
    Task task = Task.Run(() => {...}); // Test logic
    parallelTests[i] = task;
}
await Task.WhenAll(parallelTests);
```

Figur 34: Snippet af test parallelisering af browsere

Men for at eksekvere testene kontinuerligt i takt med systemet under tests udvikling og drift. Er det altafgørende at sætte det i kontekst af en CI / CD pipeline.

CI / CD

GitHub Actions er det valgte CI / CD værktøj på baggrund af source control valget. Det betyder også at skrevne GitHub Action Workflows skal følge et specifikt yaml skema.

En af styrkerne der er ved GitHub Actions, er at der er udstedt software til at køre self-hosted runners. Så man kan hoste sin CI server på egen valgte hardware. Ultimativt betyder det ofte at det bundet ud i hurtigere pipeline eksekveringstid end GitHub's hosted runners. Især ved at udnytte pre-installeret software eller caching.

Self-hosted runners blev også udnyttet i dette projekt på baggrund af gevinsterne som nævnt foroven. I takt med software kravene bliver identificeret for de skrevne GitHub Actions, kan der driftes en Dockerfile der beskriver et image for en GitHub Actions runner med det nødvendige software pre-installeret:

```
FROM ubuntu:22.04

ARG ARCH="x64"
# Prevents prompts
ARG DEBIAN_FRONTEND=noninteractive
ARG RUNNER_VERSION="2.316.1"

RUN apt update -y && apt upgrade -y

RUN useradd -m docker

# Uuid generation package
RUN apt install uuid-runtime

# Packages for .net
RUN apt install -y --no-install-recommends \
    curl jq ca-certificates dotnet-sdk-8.0

RUN curl -sL https://aka.ms/InstallAzureCLIDeb | bash

# GH actions runner boilerplate
RUN mkdir -v /home/docker/actions-runner

WORKDIR /home/docker/actions-runner

RUN curl -o actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz -L
https://github.com/actions/runner/releases/download/v${RUNNER_VERSION}/actions-run-
ner-linux-x64-${RUNNER_VERSION}.tar.gz

RUN tar xzf ./actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz

RUN chown -R docker ~docker && /home/docker/actions-runner/bin/installdependen-
cies.sh
```

```

COPY start.sh start.sh

RUN chmod +x start.sh

USER docker

ENTRYPOINT [ "./start.sh" ]

```

Figur 35: GitHub Actions runner dockerfile

Workflows kan beriges af andre open-source actions. En af disse der er brugt i dette projekt er Azure Login Action (Azure Login, 2024). På en GitHub hosted runner ville der da skulle installeres den pågældende Azure CLI version midlertidigt, for at kunne udføre operationen, som tager værdifuld eksekveringstid for pipelinen.

Azure Login Action udsteder funktionalitet til at authenticate op mod Azure. Dette giver lejlighed til at kunne udnytte GitHub Secrets til at skjule kritisk data i et givent workflow og udrulle Selenium Grid til Azure Container Apps gennem GitHub Actions:

```

name: Deploy Aspire
on: workflow_dispatch

permissions:
  id-token: write
  contents: read

jobs:
  deploy:
    runs-on: self-hosted
    env:
      AZURE_CLIENT_ID: ${ secrets.AZURE_CLIENT_ID }
      AZURE_TENANT_ID: ${ secrets.AZURE_TENANT_ID }
      AZURE_SUBSCRIPTION_ID: ${ secrets.AZURE_SUBSCRIPTION_ID }
      AZURE_ENV_NAME: selenium-env
      AZURE_LOCATION: westeurope
      AZURE_RESOURCE_GROUP: seleniumRG
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Install azd
        uses: Azure/setup-azd@v1.0.0

      - name: Log in with Azure
        uses: azure/login@v2
        with:
          creds: '${ secrets.AZURE_CREDENTIALS }'

      - name: Deploy bicep
        run: |
          az deployment group create --resource-group ${ env.AZURE_RESOURCE_GROUP }
          --name selenium-deployment --template-file ./src/IaC/selenium-hub/main.bicep --
          parameters ./src/IaC/selenium-hub/main.parameters.json

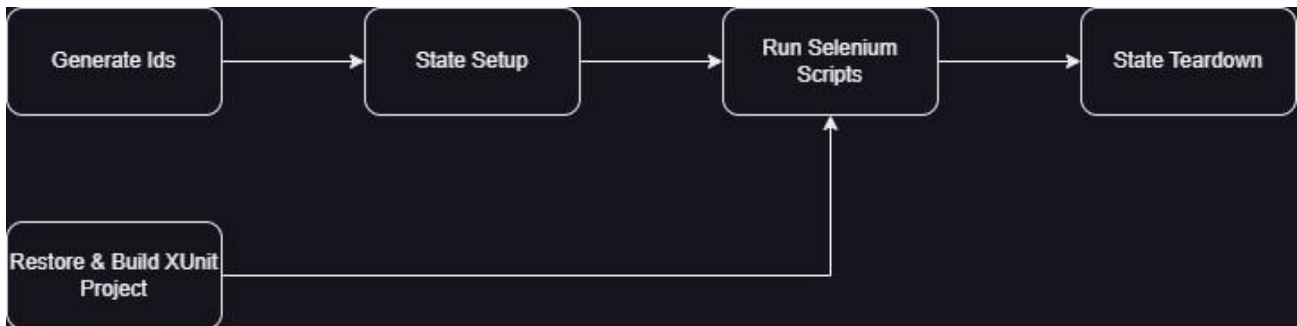
```

Figur 36: Authentication mod Azure og udrulning med Bicep some IaC i GitHub Actions

AUTOMATISERET ACCEPTANCE TESTS

Med infrastrukturen bag Selenium Grid udrullet på Azure Container Apps og webdriver scripts skrevet i et XUnit projekt. Giver det rede for at eksekvere acceptance tests på en automatiseret facon i en CI / CD pipeline.

Workflowet skal gøre brug af test stadie opsætningen – diskuteret under afsnittet 'Test Miljø & Stadie' – og testene i XUnit projektet. Fra et overbliksperspektiv skal et gennemført workflow udføre følgende:

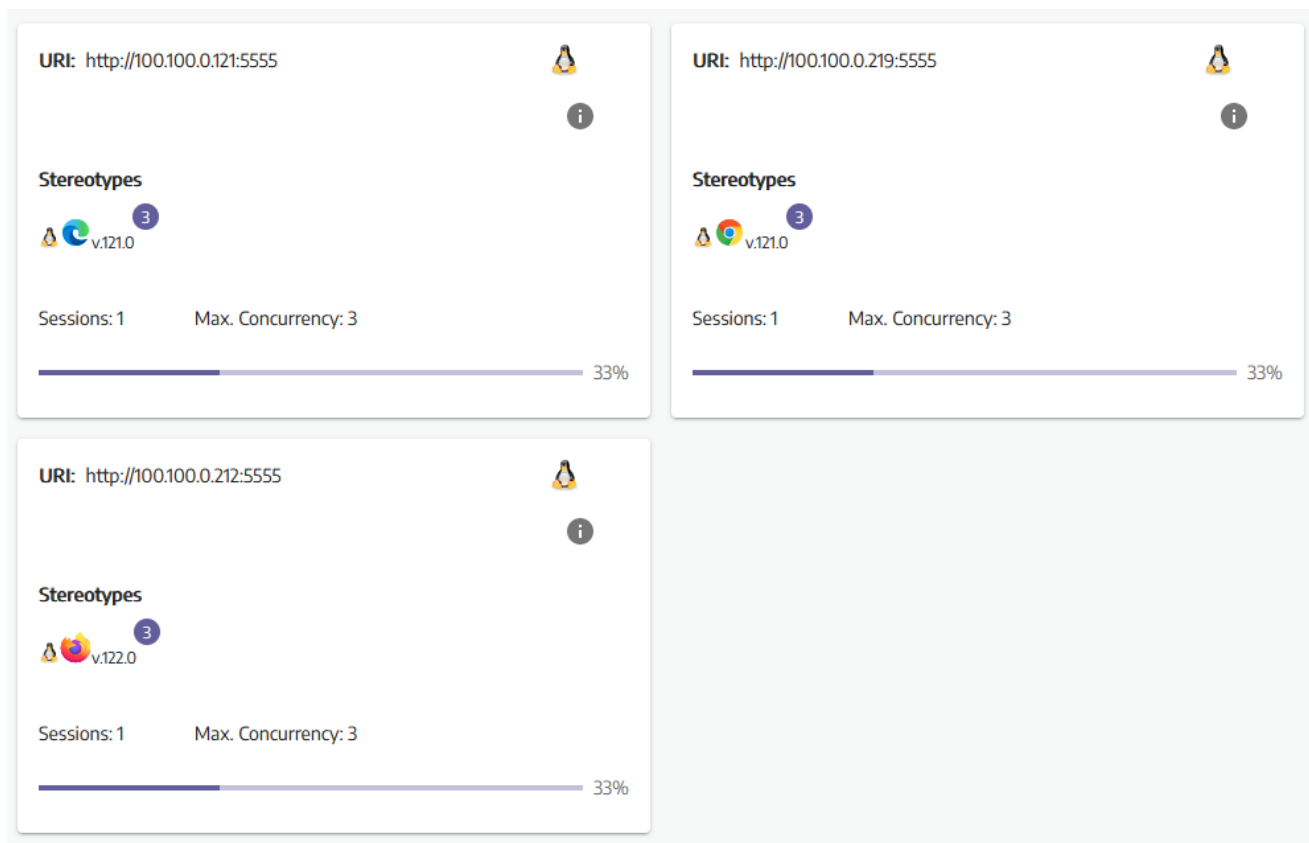


Figur 37: Acceptance tests workflow

Den fulde workflow fil kan findes i bilag 3 (Acceptance Tests Workflow). Af årsag til open-source er workflow triggeren sat til 'workflow_dispatch', for at det ikke kan udnyttes af fremmede til unødigt at stresse infrastrukturen. Sæt triggeren var sat til 'pull_request', ville workflowet eksekvere når en fremmede åbner en pull request mod repositoret.

Det er væsentligt for workflowet at 'State Teardown' jobbet eksekverer hvis 'State Setup' har kørt. Da det tager hånd om at fjerne data efter det ikke længere har funktion. I GitHub Actions kan der sættes 'continue-on-error' på et givent job, for at sikre at hele workflowet ikke stopper ved et u hensigtsmæssigt job. Dette kan udnyttes til at sikre sig at 'State Teardown' altid vil eksekvere hvis 'State Setup' også har eksekveret.

Det distribueret Selenium Grid udsteder en brugergrænseflade – som kan tilgås på url'et: gridguardian.betterboard.dk – der giver indsigt i statussen på aktive webdriver sessions. Blandt andet verificeringen om at forskellige browser sessions kører parallelt:



Figur 38: Selenium Grid brugergrænseflade

Et overblik over en tidlig iteration af acceptance test systemet i et lokalt miljø kan ses i bilag 4 (System Diagram).

KONKLUSION

BetterBoard havde en ambition om at udrulle software ændringer som minimum 8 gange om dagen, med tillid til den automatiske kvalitetssikring.

Dette projekt har ikke direkte testet denne KPI. Men har faciliteret en mere omfattende automatisk kvalitetssikring for BetterBoard's softwareprodukter.

Implementeringen af dette test system har lagt stor vægt på eksekveringstid og parallelisering, til formål for ikke at lade den automatiske kvalitetssikringsproces være flaskehalsen for at opnå at udrulle 8 gange om dagen. Da gennemsnittet for workflow kørslerne for dette projekts system har været på et minut, vurderes det at kvalitetssikringen ikke vil være en flaskehals for automatisk udrulning.

For at garantere firmaets tillid til den automatiske kvalitetssikring, der som minimum også dækker de manuelle tests der blev foretaget før udrulning, blev der udarbejdet nogle accept kriterier – i form af use cases – i samarbejde med BetterBoard's CEO. Med acceptance tests implementeret som selenium webdriver scripts, der simulerer disse use cases, vurderes det at den automatiske kvalitetssikring er tilstrækkelig til at garantere firmaets tillid til en automatisk udrulning.

Ydermere er den distribueret softwareinfrastruktur – Selenium Grid – udrullet i dette projekt, blevet dokumenteret, i form af BetterBoard's favorit IaC værktøj: Bicep. Derudover er CI / CD pipelines blevet udnyttet til at automatisere al drift i så høj grad som muligt. Derfor vurderes det at den automatiseret drift, berørt i dette projekt, er tilstrækkelig dokumenteret.

I takt med at acceptance testene er blevet automatiseret igennem en CI / CD pipeline, vurderes det at BetterBoard kan kvalitetssikre dets softwareprodukter løbende og kontinuerligt i takt med opdateringer før det når ud til slutbrugeren.

Opsummeret set har dette projekt faciliteret rammerne for at kunne give BetterBoard tillid til en automatisk kvalitetssikringsproces, til gavn for at kunne udrulle dets softwareprodukter som minimum 8 gange om dagen.

LITTERATURLISTE

- Aspire (2024). *.NET Aspire overview - .NET Aspire*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/dotnet/aspire/get-started/aspire-overview> [Tilgået 24. Maj 2024].
- Aspire Deployment (2024). *Deploy a .NET Aspire app to Azure Container Apps using `azd` (in-depth guide) - .NET Aspire*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/dotnet/aspire/deployment/azure/aca-deployment-azd-in-depth> [Tilgået 24. Maj 2024].
- Azure Project Visibility (2023). *Change project to public or private - Azure DevOps Services Public and Private Projects*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/azure/devops/organizations/projects/make-project-public?view=azure-devops> [Tilgået 6. Maj 2024].
- Azure Kubernetes (2024). *What is Azure Kubernetes Service (AKS)? - Azure Kubernetes Service*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/azure/aks/what-is-aks> [Tilgået 17. Maj 2024].
- Azure Login (2024). *GitHub Actions for deploying to Azure*. [online] Tilgængelig her: <https://github.com/Azure/login> [Tilgået 27. Maj 2024].
- Azure Resource Manager (2024). *Templates overview - Azure Resource Manager*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/overview> [Tilgået 24. Maj 2024].
- Baresi, L., Quattrocchi, G. & Rasi, N. (2023). *A Qualitative and Quantitative Analysis of Container Engines*. [online] Tilgængelig her: <https://arxiv.org/pdf/2303.04080> [Tilgået 6. Maj 2024].
- BetterBoard (2024). *Om BetterBoard | Bestyrelsesportal og software | Personer bag*. [online] Tilgængelig her: <https://www.betterboard.dk/om-betterboard/> [Tilgået 28. Maj 2024].
- Cloud Native (2023). *What is Cloud Native?* [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition> [Tilgået 17. Maj 2024].
- Container Apps (2023). *Networking in Azure Container Apps environment*. [online] learn.microsoft.com. Tilgængelig her: <https://learn.microsoft.com/en-us/azure/container-apps/networking?tabs=workload-profiles-env%2Cazure-cli#custom-vnet-configuration> [Tilgået 24. Maj 2024].
- Crispin, L. & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Upper Saddle River, Nj: Addison-Wesley.
- Cypress (2024). *Cypress Docs | System Requirements*. [online] Tilgængelig her: <https://docs.cypress.io/guides/getting-started/installing-cypress#Hardware> [Tilgået 15. Maj 2024].
- Duvall, P.M., Matyas, S. & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.

- Humble, J. & Farley, D. (2011). *Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation*. Upper Saddle River, Nj: Addison-Wesley.
- IBM (2024). *Containerization Explained | IBM*. [online] [www.ibm.com](https://www.ibm.com/topics/containerization). Tilgængelig her: <https://www.ibm.com/topics/containerization> [Tilgået 3. Maj 2024].
- Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud*. Beijing; Boston; Farnham; Sebastopol; Tokyo O'reilly.
- Nightwatch (2024). *Nightwatch.js*. [online] Tilgængelig her: <https://nightwatchjs.org/> [Tilgået 15. Maj 2024].
- Playwright (2024). *Fast and reliable end-to-end testing for modern web apps | Playwright*. [online] Tilgængelig her: <https://playwright.dev/> [Tilgået 14. Maj 2024].
- Unmesh Gundecha (2012). *Selenium Testing Tools Cookbook*. Packt Publishing Ltd.
- Selenium (2024). *Selenium Grid Components*. [online] Tilgængelig her: <https://www.selenium.dev/documentation/grid/components/> [Tilgået 15. Maj 2024].
- Service Connections (2024). *Service connections in Azure Pipelines - Azure Pipelines*. [online] [learn.microsoft.com](https://learn.microsoft.com/en-us/azure/devops/pipelines/library/service-endpoints?view=azure-devops&tabs=yaml). Tilgængelig her: <https://learn.microsoft.com/en-us/azure/devops/pipelines/library/service-endpoints?view=azure-devops&tabs=yaml> [Tilgået 3. Maj 2024].
- Twelve Factor App (2017). *The Twelve-Factor App*. [online] Tilgængelig her: <https://12factor.net/concurrency> [Tilgået 17. Maj 2024].
- UseDotNet @2 (2024). *Use dotnet v2 task*. [online] [learn.microsoft.com](https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/use-dotnet-v2?view=azure-pipelines). Tilgængelig her: <https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/use-dotnet-v2?view=azure-pipelines> [Tilgået 3. Maj 2024].

BILAG

- Bilag 1: Test State Setup Sekvensdiagram. [online] Tilgængelig her: <https://raw.githubusercontent.com/AndersBjerregaard/BetterBoard-BA/master/documentation/closed-source-sequence-diagram.svg>
- Bilag 2: Use Cases. [online] Tilgængelig her: <https://github.com/AndersBjerregaard/BetterBoard-BA/blob/master/documentation/ba-usecases.md>
- Bilag 3: Acceptance Tests Workflow. [online] Tilgængelig her: <https://github.com/AndersBjerregaard/BetterBoard-BA/blob/master/.github/workflows/selenium-tests.yml>
- Bilag 4: System Diagram. [online] Tilgængelig her: <https://raw.githubusercontent.com/AndersBjerregaard/BetterBoard-BA/master/documentation/betterboard-e2e.png>