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: TBD

SOURCE CODE: HTTPS://GITHUB.COM/ANDERSBJERREGAARD/BETTERBOARD-BA

INDHOLDSFORTEGNELSE

FORORD

Kode eksempler i denne rapport kan forekomme med følgende snippet: ’{…}’. Dette fremviser en kollapset metode, constructor, klasse eller lignende. Hvori 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

- Gevinsterne ved automatisk kvalitetssikring i forskellige testniveauer.

ANTI-PATTERN

- Det dårlige ved manuel release.

EKSISTERENDE LØSNING

BetterBoard har en automatisk kvalitetssikringsproces på dets softwareprodukter. Bestående udelukkende af en suite af unit tests, på systemets web api monolit.

- Diagram over infrastrukturen.

PROBLEMSTILLING

BetterBoard har en ambition om at kunne *deploy* 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 kvantificerbare accept kriterier for en forbedret automatisk software kvalitetssikringsproces. Der kan garantere firmaets tillid til en automatisk *continuous deployment*.

Herunder 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 snævret specifikation. Til formål for 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).

Og som problemformuleringen beskriver. Så er det specifikt acceptance tests der menes der kan automatiseres, der vil blive bearbejdet.

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

I samme anordning som foroven: Førnævnte firma har en del eksisterende software, skrevet i en bestemt tech stack. Vil denne opgave, tilnærmelsesvis, 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

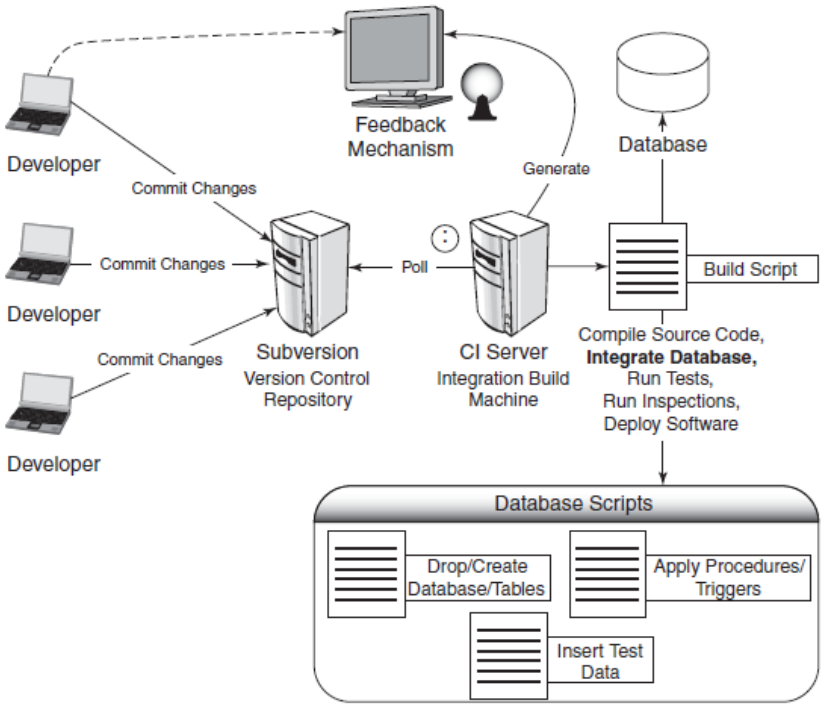
Den fundamentale teori bag de væsentligste værktøjer og emner bliver gennemgået i dette afsnit. Til formål for at bringe læseren bedst muligt ind i analyse-, refleksion- og konklusionerne i denne rapport.

CI / CD PIPELINES

Continuous integration, delivery & deployment eller CI / CD pipelines. Er et værktøj der bruges til at integrere kildekode ændringer, kompilere system artefakter 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. Til gavn for at reducere risici og skabe 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 1: CI / CD eksempel (Duvall, Matyas & Glover, 2007)

SECRET / KEY MANAGEMENT

Som svar på et anti-pattern at skrive kritiske oplysninger blottet i kildekode. Findes der værktøjer til at styre diverse nøgler, som systemer kan i et omfang er afhængig af. Dette giver lejlighed til at kunne open-source kildekode. Men stadig have implementeringsspecifikt logik baseret på key management for et system.

Er Azure den valgte 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.

```yaml
trigger:
  - main

stages:
  - stage: Deploy
    displayName: 'Deploy Resources'
    jobs:
      - job: AzureCLI
        pool:
          name: Default
        steps:
          - task: AzureKeyVault@2
            inputs:
              KeyVaultName: 'keyvault_name'
              SecretsFilter: '*'
              RunAsPreJob: false
            displayName: 'Retrieve Key Vault Secrets'
          - task: Bash@3
            inputs:
              filepath: './deploy.sh'
            env:
              APP_ID: $(AppId)
              APP_SECRET: $(AppSecret)
              TENANT_ID: $(TenantId)
            displayName: 'Deploy script'
```

Figur 2: 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 – som 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.

SYSTEMDOKUMENTATION

SYSTEM INFRASTRUKTUR DIAGRAMMER

SEKVENSDIAGRAMMER

AZURE CLOUD & TOOLING

TESTNIVEAUER

ACCEPTANCE TEST

UNIT TEST

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.

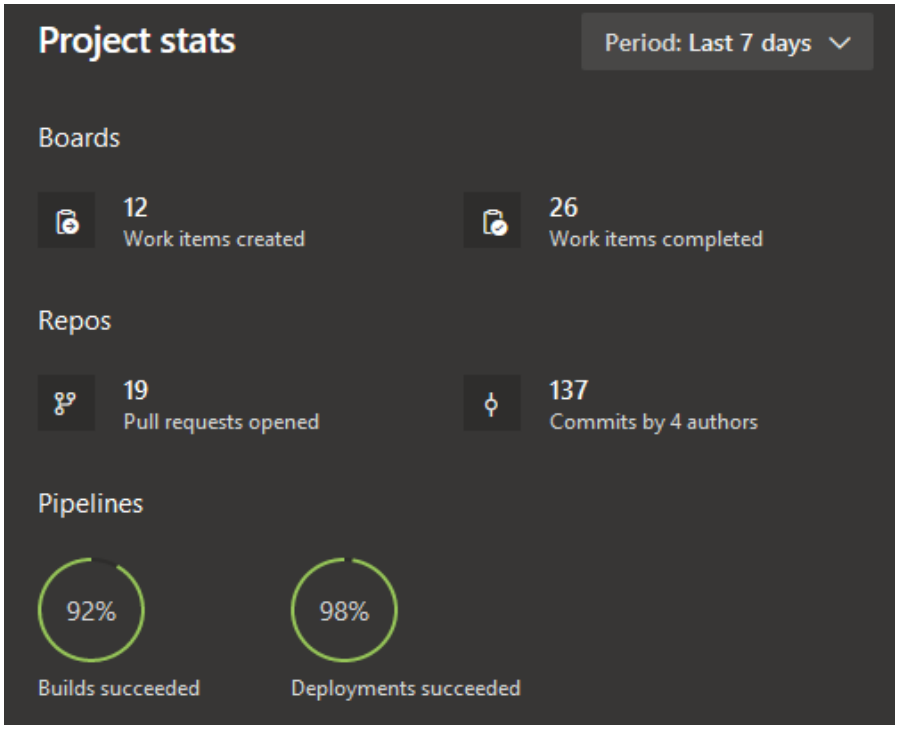
Ydermere, har valget af source control provider indflydelse på tilgængelighed.

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 3: Azure Repos Project stats eksempel

Ydermere til projektstyring er Azure DevOps Boards. Som er et gennemført projektstyringsværktøj, direkte integreret med kildekoden. 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 – som vil blive uddybt yderligere under afsnittet ”CI / CD” i denne rapport.

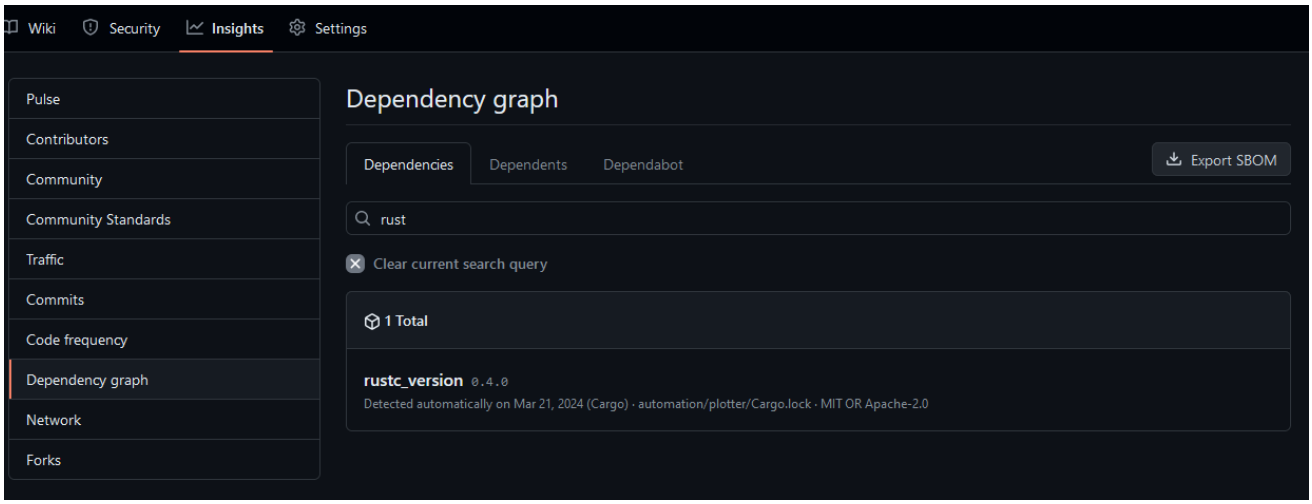
Er Azure den valgte cloud provider, er der gevinster ved også at holde sig til selvsamme tech stack, og vælge Azure Repos som source control provider.

Dog er Azure Repos, efter standard, privat 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 4: 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 distruberede Azure ressourcer gennem GitHub – Som vil blive gennemgået under afsnittet ”CI / CD” i denne rapport.

Pragmatisk set, er Azure Repos en mere 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 gement 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 it business requirements*." (Humble & Farley, 2011).

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

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

Tests op mod applikationslaget vil kræve at den individuelle test skal holde stadie 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 idempotente og 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 stadieopsætningen 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:

```
        [HttpPost]
        [Route(@"testtransaction/start")]
        [ApiKeyAuthorization]
        public async Task<IHttpActionResult> Start([FromBody] TestRunIdentifier runIden-
tifier)
        {
            try
            {
                await _transactionService.Seed(runIdentifier);
            }
            catch (Exception e)
            {
                string returnMessage = ExceptionMessageHelper.ConcatenateException-
Message(e);

                if (e.GetType() == typeof(InvalidTransactionException))
                {
                    return BadRequest(returnMessage);
                }

                return InternalServerError(e);
            }
            return StatusCode(HttpStatusCode.Created);
        }

        [HttpDelete]
```

```
        [Route(@"testtransaction/cleanup")]
        [ApiKeyAuthorization]
        public async Task<IHttpActionResult> CleanUp([FromBody] TestRunIdentifier runI-
dentifier)
        {
            try
            {
                await _transactionService.CleanUp(runIdentifier);
            }
            catch (Exception e)
            {
                string returnMessage = ExceptionMessageHelper.ConcatenateException-
Message(e);

                if (e.GetType() == typeof(InvalidTransactionException))
                {
                    return BadRequest(returnMessage);
                }

                return InternalServerError(e);
            }
            return StatusCode(HttpStatusCode.NoContent);
        }
```

Figur 5: 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 endpointed 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.

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 AuhorizationFilterAttribute, 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(HttpActionContext actionContext)
        {
            IEnumerable<string> apiKeyHeaders = Enumerable.Empty<string>();
            if (!actionContext.Request.Headers.TryGetValues("X-Api-Key", out apiKeyHead-
ers))
            {
                actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized, "Miss-
ing 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(HttpActionContext actionContext, Can-
cellationToken cancellationToken) {...}

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

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

        private async Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync-
Core(HttpActionContext actionContext, CancellationToken cancellationToken,
Func<Task<HttpResponseMessage>> continuation) {...}
    }
```

Figur 6: 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 7: Test Transaction Endpoint HTTP body domæne klasse

Klassen foroven fremviser det eneste data påkrævet at opsætte stadie til acceptance tests. Id propertien 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 propertien 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 endpointed 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.

```
        /// <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 '{COM-
PANY_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 - " + identi-
fier.ToString("N") };

            board = await _boardService.CreateBoard(companyId.ToString(), board, Ob-
jectId.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 8: 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 sikkerhedsmekaniske 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.

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:

```yaml
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 9: Playwright tests i GitHub Actions (playwright.dev, 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.dev, 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.

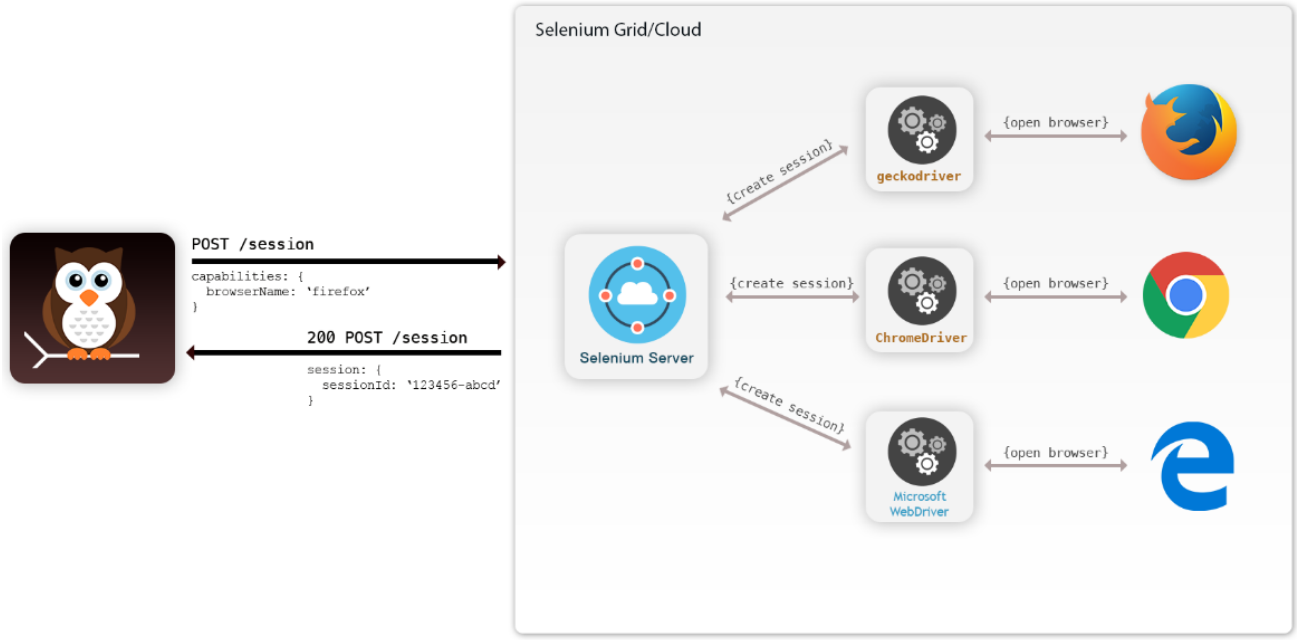
Et positivt aspekt af ovenstående er dog at der er minimal opsætning påkrævet, før der kan fås udbytte af acceptance tests.

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 (nightwatchjs.org, 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 10: Nightwatch tests op mod distribuerede browser sessions (nightwatchjs.org, 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 kompileres 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. Hvori et 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.

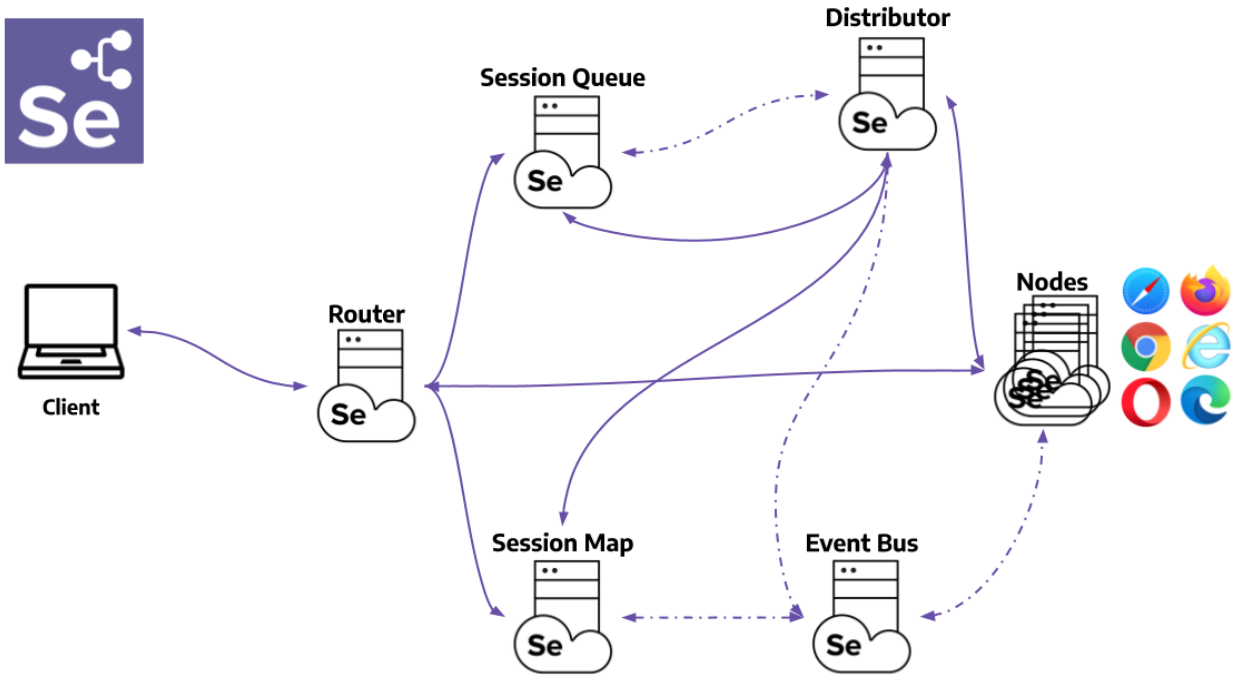
På baggrund af arkitekturen bag Cypress betyder dog også at det skalerer på samme vis som Playwright. Som resulterer i at CI serveren står for skaleringen, sæt at tests skal eksekveres parallelt. Hvordan dét kan implementeres, kan læses under 'CI / CD' afsnittet længere nede i denne rapport.

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.dev, 2024). Som bliver fremhævet af dets systemdiagram:



Figur 11: Selenium Grid infrastruktur komponenter (selenium.dev, 2024)

Cypress vurderes til at være det mest sofistikeret 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 bunder 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.

UDRULNING AF SELENIUM GRID

DOCKER

.NET ASPIRE & BICEP

AZURE CONTAINER APPS

VIRTUALISERING

GENSKABELIGE MILJØER

LOKAL & CLOUD MILJØ

END-TO-END TESTS

USE CASES

VALIDERING AF WEBDRIVER RESULTATER

TEST EKSEKVERINGS MILJØ

TEST MILJØ

CI / CD

GITHUB ACTIONS

AZURE PIPELINES

SECRET / KEY MANAGEMENT

SELF-HOSTED RUNNERS / AGENTS VS. CLOUD PROVIDED RUNNERS / AGENTS

UDRULNING TIL AZURE

AZURE WEB GUI

INFRASTRUCTURE-AS-CODE

.NET ASPIRE

KONKLUSION

LOKALT SYSTEM INFRASTRUKTUR DIAGRAM

DISTRIBUERET SYSTEM INFRASTRUKTUR DIAGRAM

MONOLIT ENDPOINTS SEKVENSDIAGRAMMER

PROBLEMFORMULERING BESVARELSE

LITTERATURLISTE

Crispin, L. & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Upper Saddle River, Nj: Addison-Wesley.

Humble, J. & Farley, D. (2011). *Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation*. Upper Saddle River, Nj: Addison-Wesley.

Duvall, P.M., Matyas, S. & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk* . Pearson Education.

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].

IBM (2024). *Containerization Explained | IBM*. [online] www.ibm.com. Tilgængelig her: https://www.ibm.com/topics/containerization [Tilgået 3. Maj 2024].

Service Connections (2024). *Service connections in Azure Pipelines - Azure Pipelines*. [online] learn.microsoft.com. Tilgængelig her: https://learn.microsoft.com/en-us/azure/devops/pipelines/library/service-endpoints?view=azure-devops&tabs=yaml [Tilgået 3. Maj 2024].

UseDotNet@*2* (2024). *Use dotnet v2 task*. [online] learn.microsoft.com. Tilgængelig her: https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/use-dotnet-v2?view=azure-pipelines [Tilgået 3. 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].

playwright.dev. (2024). *Fast and reliable end-to-end testing for modern web apps | Playwright*. [online] Tilgængelig her: https://playwright.dev/ [Tilgået 14. Maj 2024].

nightwatchjs.org. (2024). *Nightwatch.js*. [online] Tilgængelig her: https://nightwatchjs.org/ [Tilgået 15. Maj 2024].

cypress.io. (2024). *Cypress Docs | System Requirements*. [online] Tilgængelig her: https://docs.cypress.io/guides/getting-started/installing-cypress#Hardware [Tilgået 15. Maj 2024].

selenium.dev. (2024). *Selenium Grid Components*. [online] Tilgængelig her: https://www.selenium.dev/documentation/grid/components/ [Tilgået 15. Maj 2024].

BILAG