Högskolan i Gävle

Cleaning Manager

Bästa Brancharna

Norton Åbom, Emre Özata, Brandon Adams Nkata, Elias Wåhlin, Linus Renström

2025-06-06

Programvaruteknik

Lärare: Åke Wallin
Lärare/handledare: Åke Wallin

Innehållsförteckning

Innehållsförteckning

Inledning

I detta projekt skulle ett system utvecklas för att underlätta hantering av städningar på ett företag. För detta har Spring Boot-ramverket använts för att underlätta och effektivisera utvecklingen. Projektet syftar främst på gruppens arbetssätt och utveckling enligt den agila metoden Scrum.

1.1 Bakgrund

För att effektivt kunna nå ett mål med agil utveckling har den agila metoden Scrum använts i teamet. Samt har det använts det populära ramverket Spring Boot.

1.1.1 Scrum

Scrum är en iterativ och inkrementell metodik som utgör ett flexibelt ramverk för team att effektivt samarbeta mot ett gemensamt mål. Arbetet delas upp i korta cyklar, så kallade sprints där varje sprint kan beskrivas som ett inkrement, vilket är en ytterligare del tillagd i systemet. En sprint varar vanligtvis mellan 2–4 veckor [1]. I scrum finns en backlogg, så kallad product backlog, som är uppdelade uppgifter som utvecklingsteamet ska utveckla. Denna backlog uppdateras kontinuerligt, vilket är en stor faktor till det som gör Scrum agilt.

Vanligtvis brukar följande roller ingå i Scrum:

Product Owner – Ansvarar för prioriteringar i backloggen och ser till att rätt sak byggs.

Scrum Master – Coachar teamet och röjer undan onödiga hinder.

Utvecklingsteamet - självorganiserat team som utvecklar systemet.

Scrum involverar även delar som sprintbacklog, vilket är prioriterade uppgifter utifrån productbacklog som teamet tror kan utföras under en sprint. Dessa backlogs som hamnar i sprintbacklog brukar vanligt vis pokras med poäng där poängen är ett mått på prioritet och tidsåtgång. Dessa poäng kan sedan beskrivas i ett diagram, så kallat för burnt-down-chart, vilket blir som ett mått för arbetseffektivitet. För varje dag brukar man även sätta upp ett mål för vad som ska bli klart under dagen, så kallad för daily sprint. Uppgifternas status brukar visuellt representeras i ett visuellt bräde med vanliga förekommande delar som To-do, testing, done.

I början av varje arbetsdag utförs ett daily standup, vilket är en chans för Scrum Master och utvecklingsteamet att resonera om hinder, vad som gått bra och vad som ska göras. I slutet av varje sprint engageras en Sprint Review, vilket är ett moment där alla tre roller samt övriga intressenter kan delta i, där en genomgång av den genomförda sprinten redovisas. Utifrån denna review kan man säkerställa att arbetet går åt rätt håll. Utöver Sprint review sker även en Sprint Retrospective vilket är en refkletion över arbetssättet under sprinten. I en Retrospective deltar vanligtvis endast Scrum Master och utvecklingsteamet, där de reflekterar över hur deras arbete kan förbättras.[1]

1.1.2 Spring Boot

Inom programvaruteknik har utvecklingen av webbaserade system blivit en viktig del. För att på ett effektivt sätt kunna bygga applikationer som är skalbara, modulära och robusta så används vanligt vis ramverket Spring Boot.

I artikeln *Design and Implementation of Tourism System Based on Spring Boot* använder de Spring Boot som ramverk för att skapa ett informationssystem för turister. I systemet hanteras användarregistreringar, behörigheter och bokningar. Det skapade turistsystemet visar på hur Spring Boot kan fungera på en tydligt skiktad arkitektur som har enkelriktade beroenden mellan lager där man kan dra paralleller till de aktuella städ-system som utvecklats. Likt de turistsystem som skapats så använder de olika typer av behörighetskontroller, hantering av persistenta domänobjekt samt identifiering.

Turistsystemet utvecklades enligt gedigna arbetsmetoder, där det fanns en tydlig kravspecificering, skiktad arkitektur samt mycket omfattande testning. Dessa krav kan man se som en liknelse till de krav som ställts på de städ-system som skall utvecklas där det ställts krav på testbarhet, hantera olika roller och skiktad arkitektur. Detta leder till att denna artikel har varit till en god grund för motivering kring val av ramverk som skall användas i detta projekt. [2]

1.2 Syfte

Syftet med projektet var att skaffa en större förståelse över hur det är att arbeta i ett utvecklingsteam enligt ett agilt ramverk för systemutveckling. Denna Rapport kommer att redovisa de arbetsmetoder, arbetssätt och verktyg som använts för att implementera systemet.

2 Metod

För att effektivt och agilt arbeta bestämdes det att införa ett Scrum-aktigt arbetssätt. Detta innebör att Scrums arbetsmetoder adopterades under de olika faserna. Nämligen planering, utveckling, testning och granskning och integration.

2.1 Planering

Under planeringsfasen använde vi Scrum Poker som planeringstillvägagångssätt för att skatta och prioritera backlogg-ärenden, vilka sedan togs in i sprintbacklogen.

Samtliga team-medlemmar hade ett antal poker kort som var markerade enligt Fibonaccis talföljdvariation på värden. En backlog issue introducerades där vardera teammedlem fick skatta de antal poäng som de ansåg issue:n skulle tillsättas. Rätt så ofta varierade värden på skattningen som teammedlemmarna uppskattade. Som följd väcktes det diskussioner och motiveringar kring hur alla tänkte med deras skattningar. Slutligen kom teamet till en konsensus om de slutliga poängen som issue:n skulle få.

2.2 Utveckling

Under utvecklingen införde vi en del principer kring Scrum arbetssättet för att uppnå en agilt arbetsgång.
Dessa principer var bland annat:

- Par och mob-programmering för en kontinuerlig kodgranskning och kunskapsdelning.
- Daily standup och en kontinuerlig dialog i teamet om strategier och uniforma lösningar.
- Strukturellt mässigt infördes ett Scrum Board på GitLab för att visualisera teamets arbetsflöde
- Code Reviews utfördes också på slutet av sprintarna där hela gruppen fick granska lösningarna innan integration till main branchen. Detta gjordes i form av mob-programmering där teamet satt i grupp och diskuterade vid samma dator.

2.3 Testning och granskning

Funktionalitettestning och kodgranskning utfördes i form av par respektive mob-programmering.
Testning och granskning utfördes kontinuerligt under sprinten i dem respektive par som utvecklat funktionaliteten. Slutet av sprinten utfördes det granskning av alla avklarade backlogs på ett mob tillväggångsätt. Denna fas med en slutgiltig kodgranskning och funktionalitettestning var i enlighet likt en code review som i vanliga fall skulle syfta på att redovisa utvecklingen till product owners och stakeholders.

2.4 Integration

Efter kodgranskningen och testning integrerades den nya funktionaliteten i main branchen, detta gjordes med hjälp av version hanteringssystemet Gitlab. Det tänkta tillvägagångssättet som främjades var att alltid hämta det senaste från repositoryt från molnet och sedan lokalt mergea in den nya lokala feature branchen i lokala main. Efter testades allt återigen för att vara säker kring att systemet fungerar som förväntat och slutligen pusha den nya funktionaliteten till repo:t i molnet för en gemensam åtkomst.

2.5 Verktyg och arkitektur

Systemet byggdes upp med samma fyrskikts-princip i varje domänpaket oavsett om det gällde Person, Room, Cleaning eller någon annan modul. Överst låg ett REST-lager som exponerade JSON-endpoints, tog emot Caller-Uuid och Target-Uuid i headers och konverterade data till och från DTO-objekt med MapStruct (se Figur 1).

```
@GetMapping(path = "/get")
public ResponseEntity<? extends BasePersonDto> getPersonByUuid(
        @RequestHeader(name = "targetUuid") String targetUuid,
        @RequestHeader("callerUuid") String callerUuid) {
    Person person = service.getPersonByUuid(targetUuid);
    if (service.isAdmin(callerUuid)) {
        return ResponseEntity.ok(personMapper.toFullDto(person));
    }
    return ResponseEntity.ok(personMapper.toLimitedDto(person));
}
```

Figur 1 visar en kod snut på en endpoint i en av controllerklass

Under detta befinner sig ett Service-lager där affärsregler, rättighetskontroller och transaktioner placerades, där all loggning sker via en gemensam SLF4J-adapter. Varje serviceinstans injicerades via konstruktörer, vilket gjorde koden lätt att enhetstesta med Mockito (se Figur 2).

```
@Autowired
public PersonService(PersonRepository personRepository) {
    this.personRepository = personRepository;
    this.logger=Logger.get();
}
```

Figur 2 visar hur annoteringen @Autowired används för att injicera

Service-klasserna anropade sedan Repository-lagret som oftast bestod av gränssnitt som ärvde från JpaRepository. Alla standard-CRUD-operationer genererades därför automatiskt, och specialfrågor som findByUuid deklarerades med metodnamn eller JPQL. Hibernate stod för SQL-genereringen, så ingen manuell databaskod behövde underhållas (se Figur 3).

```
public interface PersonRepository extends JpaRepository<Person, Long> {

    public Optional<Person> findByUuid(String uuid);

    Person readById(Long id);
}
```

Figur 3 visar en kodsnutt av hur repository interface konstruerades där dem fick ärva av JpaRepository

Längst ned fanns respektive domän-entitet – exempelvis Person, Room eller Cleaning. Samtliga entiteter var annoterade med `@Entity` och kapslade sina egna invarianter: set-metoder kastade domänspecifika undantag vid felaktiga värden, och varje entitet bar på ett unikt UUID-fält som genererades i en `createUuid()`-metod innan objektet sparades. Relationskopplingar definierades med `@ManyToOne` och `@OneToMany`, vilket gjorde att Hibernate automatiskt hanterade främmande nycklar.

Felhanteringen samlades i en gemensam `@ControllerAdvice` noterad klass som i detta projekt kallades för `RestExceptionHandler`. Där översattes alla domän- och service-exceptions till konsekventa HTTP-svar med rätt statuskod och felpayload, så att samma felkontrakt gällde för samtliga endpoints (se Figur 4).

```
@ExceptionHandler(RoomNotFoundException.class)
public ResponseEntity<Object> handleInvalidRoom(RuntimeException ex, WebRequest request) {
    return handleExceptionInternal(
            ex,
            ErrorResponse.of("0x69", ex, HttpStatus.NOT_FOUND),
            HttpHeaders.EMPTY,
            HttpStatus.NOT_FOUND,
            request);
}
```

Figur 4 visar exempel på hur ett definerat exception omvandlas till ett Http-svar

Bean-Validation-annotationer på indata-DTO:er stoppade ogiltiga värden redan innan kontrollernas metodkroppar kördes, medan "fail-fast"-kontroller i entiteternas set-metoder fångade logiska fel djupare ned i stacken (se Figur 5).

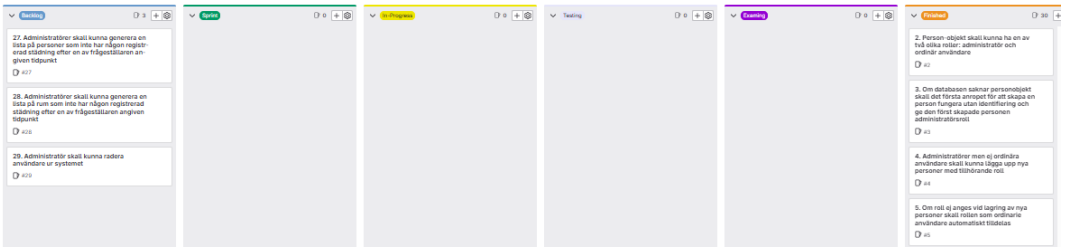
```
public record PersonCreateDtoIn(
        @NotBlank @NotNull
        String firstName,

        @NotBlank @NotNull
        String lastName,

        @NotNull @NotBlank
        LocalDate birthDate,

        Role role
) implements BasePersonDto {

}
```

Figur 5 visar Bean-validation på inkommande data via Dto:n

Denna återkommande struktur REST → Service → Repository → Entity tillämpades konsekvent på varje domänområde. Resultatet blev tydliga ansvarsgränser, enkelriktade beroenden, hög testbarhet och ett kodmönster som uppfyllde samtliga funktionella och icke-funktionella krav i projektets specifikation.

2.6 Dokumentation och spårbarhet

Ett spring/kanban-board och digitaldagbokslogg nyttjades för att visualisera och överblicka om det färdiga och kommande arbete. Dessutom infördes en policy för att skriva beskrivande commit-meddelande för att öka spårbarheten kring versionshantering (se Figur 6).



Figur 6 visar dent digitala spring/kanban board på GitLab som användes för att spåra och övervaka projektets arbetsflöde

3 Genomförande

I följande avsnitt presenteras genomförandet i form av de sprintar som pågick under projektet. En sprint varade i ungefär tre arbetsdagar, med vissa undantag. Vissa sprintar var uppdelade i fler än tre dagar då halvdagar kunde förekomma.

3.1 Sprint 1

2025-04-09

Eftersom projektet utgick från gamla workshops så hade vi alla en gemensam förståelse över domänerna, ramverket spring samt vilket arbetssätt som ska användas. Det enda som utfördes var en snabb överblick och sedan delade vi upp oss i grupper om två respektive tre personer eftersom vi var 5 medlemmar. Vi började med Scrum-poker där vi bestämde att poängen estimerar endast hur lång tid vi trodde uppgiften skulle ta och inte hur viktiga dem var. Vi hade även en diskussion kring backloggen om gemensamma idéer angående hur man ska tolka samt lösa de första uppgifterna. Vi skulle beskriva de som var tre i gruppen för par programmering fast med två navigators i stället för en. Efter första arbetsdagen hade vi bränt av de fem första uppgifterna från backloggen då de nästan redan var givna från tidigare workshop.

2025-04-10

Andra arbetsdagen började vi med en daily standup och förståelsen för Git började komma, vilket ledde till att vi effektivt kunde använda branches och merge. Eftersom vi satt tillsammans hela gruppen kunde vi på så sätt undvika three-way-merges med konflikter då vi hade koll på vilka filer vi jobba i. Vi märkte dock att upprepat kod fanns på flera ställen och börja tänka på refaktorering.

Issues som blev klar: 6, 7

3.2 Sprint 2

2025-04-14

Dagen började med daily standup och vi tog upp problem som refaktorering samt märkte vi att vi använde default-konstruktor i tester. Vi tänkte inte mer på en stor refaktorering då vi ansåg att vi kunde utföra en stor refaktorering senare då vi vet mer om hur systemet kommer se ut. Dock bestämde vi att testerna skulle ändras så vi använde args konstruktorer i stället för default konstruktoren som hibernate kräver. Detta utfördes i branches som vi sen raderade eftersom vi ansåg att vi inte behövde ha någon historik över detta. Vi gjorde även default konstruktorer protected för att minska fler dåliga tester.Vi började senare även resonera kring om serviceklasserna skulle injicera andra service klasser eller om vi skulle injicera repositories. Fördelen med service klasser var att vi fick med logik från metoder och bättre separation mellan lagerna. Nackdelarna är dock att det blir ett extra anrop samt risken för cirkulärt beroende. Dock så valde vi serviceklasserna då vi ansåg att metoden `isAdmin` från `personService` skulle användas mycket.

Issues som blev klar: 8,9

2025-04-17

Dagen startades med Daily standup. Denna dag lades mycket tid på dtoer då vi börja tänka på säkerhet och vad man vill exponera om systemets bakomliggande struktur för en vanlig användare. Vi gjorde dtoer för användare och admin. I våra dtoer kunde vi även införa validering så att man inte kunde mata in tomma namn och nullvärden med `@NotBlank` och `@Notnull`. Detta gjorde att vi följde principen om fail-fast då det smäller redan i API-lagret. Vi började även på issue 11 där vi kunde använda streams som vi lärt oss. Vi skulle hämta städningar med specifikt datum kopplat till ett rum. Då hämtar vi listan med alla cleanings och streamar den.

Issues som blev klar: 10

2025-04-18

Daily standup. Denna dag blev en halvdag där mycket fokus lade på att tolka uppgift 14. Vi införde en domän som kallades feehistory så vi kunde få historik över löner som städare haft och aktuellt har. Eftersom ett till domän skapades så var alla tvungen att hänga med och förstå tanken. Vi fixade även till att korrekt lön betalas ut om man skulle ha haft en förändrad lön under en viss löneperiod.

Issues som blev klar: 11,12,14

3.3 Sprint 3

2025-04-23

Under sprint-planeringen kom vi fram till att mob-programmera issue 15-20. Anledningen var att dessa uppgifter var starkt sammankopplade och vi ansåg att alla i teamet skulle få en gemensam förståelse för hur vi skulle lösa problemet.

Denna dag gick arbetet trögt framåt då det uppstod förvirring i teamet kring hur en avgiftsändring för en person skulle kunna ske inom ett visst intervall, utan att påverka den dåvarande avgiften. Lösningen blev att skapa en metod som tar hand om avgiftsposter i avgiftshistoriken. I metoden kontrolleras om den anropande har rättighet till att ändra, men också kontrollerar att inga avgifter sätts före den senaste löneutbetalningen. Som parametrar tas då ett datumintervall emot, och det är posterna inom detta intervall som uppdateras till den nya avgiften. Eftersom vi valde att lägga ner mycket tankeverksamhet för dessa uppgifter lyckades även issue 19 täckas omedvetet. Alltså blev 15, 16 och 19 klara denna dag.

2025-04-24 (halvdag)

Denna dag inleddes med att tillsammans skriva tester för de omfattande uppgifterna 15, 16 och 19. Eftersom lösningarna på dessa issues var förhållandevis komplexa, tog även testerna lite längre tid att få till. Detta var en halvdag och testerna hann inte bli helt klara.

2025-04.25 (halvdag)

Testerna för uppgift 15, 16 och 19 skrevs färdigt och ansågs robusta. Uppgift 17 inleddes vilket skulle innebära en ny entitet i domän-lagret, `PaymentHistory`. Vi skrev även om lite i koden, då vi flyttade `getAccumalatedSalaryByPersonUuid` och `getAccumalatedSalaryByPeriod` från `CleaningService` till `PaymentHistoryService`. Detta gjordes för att få en bättre logisk struktur och ansvarsfördelning (Separations Of Concerns).

Vi skapade funktionalitet för att beräkna och returnera en specifik DTO med intjänade pengar från senaste löneutbetalningen, fram till och med dagens datum. Den specifika DTO:n har att göra med om man är administratör eller vanlig användare. Alltså att returvärdet blir olika beroende på vad man har för rättigheter.

Dagen avslutades med att planera inför sista sprintdagen, där behovet av att kunna hämta en lista med samtliga utbetalningar för specifika användare diskuterades. Därefter skulle tester implementeras.

2025-04-29

Issue 17 färdigställdes genom att man nu kunde hämta en lista med alla utbetalningar för både enskilda användare, och för samtliga användare. Tester skrevs även för denna uppgift och slutfördes samma dag.

För resterande uppgifter, 18 och 20, valde vi att dela upp oss och parprogrammera igen. Detta på grund av att vi nu hade fått et gemensam förståelse för hur denna del av systemet skulle se ut.

Uppgift 18 färdigställdes och innefattade funktionalitet för hur administratörer skulle kunna ändra löneutbetalningsdatum för enskilda användare. Dock stod vi i valet om funktionen skulle vara att ändra alla personers löneutbetalning, eller bara en enskild. Anledningen till att vi valde just en enskild var för att det inte kändes realistiskt att ändra alla personers poster, då det kan finnas flera olika faktorer till varför den är satt till ett visst datum.

Funktionaliteten för att hantera obetalda städningar, uppgift 20, blev klar. Här introducerades återigen nya DTO:er beroende på rättighet. Testerna hann inte bli klara för denna uppgift, på grund av att de första uppgifterna i sprinten mob-programmerades och tog lång tid i form av tankeverksamhet. Till nästa sprint skall alltså tester för 20 skrivas.

3.4 Sprint 4

2025-05-05

Fortsättning för uppgift 20 togs upp denna dag. Vi valde att mob-programmera testerna för denna eftersom detta egentligen tillhörde den tidigare sprinten och skulle bli klart.

När dessa var klara pokrade vi nya poäng för kommande print, vilket ledde till att vi återigen redan täcker vissa issues i backlogen. I detta fall var det uppgift 21 och 23. I sprint backlogen lades nu till uppgift 22, 24, 25 och 26. För denna sprint par-programmerade vi.

2025-05-09

Till en början använde vi springs Specifications, och implementerade en filtreringslösning för uppgift 22. Mycket betänketid gick åt att hitta en vettig lösning i repository-lagret. Här lades metoderna `lessThan` och `greaterThan` till. I service sammanställdes specifikationerna och rest-lagret tar emot fler filterparametrar i form av `@RequestParam`. Denna lösning skrotades dock då vi valde att i stället använda streams på det hämtade datat. Dock behöll vi `@RequestParam` för att ta emot de olika vilkoren.

Samtidigt utvecklades issue 24,25 och 26 med tester. Dessa var tydliga och tog inte lång tid, möjligtvis att 24 tog lite längre tid då en ny entitet behövde skapas i form av `DeviationReport`. I denna kunde en admin hämta avvikelserapporter från ett givet datum och framåt. En avvikelserapport kunde vara av anledningarna exempelvis att ett rum inte blivit städat på två dagar. 25 och 26 gick fort då det enbart var filtrerings-tillägg för administratörer.

Inför nästa tillfälle behövde tester för uppgift 22 skapas, därefter skulle refaktorering börjas se över.

2025-05-12

Denna dag insåg vi att uppgift 22 behövde göras om. Den behövde göras mer generell och läsbar. Eftersom attributen är olika för domän-objekten gjordes specifika filtreringsmetoder. Till slut blev den färdigställd och test skapades.

Vi började även refaktorera koden, med början i domain-lagret. Här lades bland annat kontroll-logik och javadoc till. I domain lades även `hash()`, `equals()` och `toString()` till enkelt med snabbkommando. I service-lagret städade vi upp beroenden så att det bara fanns koppling mellan klasserna i samma lager. Alltså exempelvis att `PersonService` kan använda funktionalitet i en annan repository-klass, men var då tvungen att gå genom respektive service-klass. Rest-lagret behövde många justeringar, främst när det kommer till path-namn. Inget känsligt data skulle skickas med `@PathVariable`, utan i stället via headern eller bodyn.

Genomgående ändrades namn på metoder och variabler för att hålla en god konsistens genom hela arbetet.

2024-05-14

Sista dagen på projektet blev en lite längre arbetsdag då projektet skulle finslipas och refaktoreras. Något som även lades till denna dag var logger i service-lagret. För denna använde vi adapter-mönstret för att kunna använda Javas logger med ett eget implementerat interface. Varje metod i service loggas för initialisering och för exekvering. Alltså när metoden kallas skrivs en loggning, och när allt gått igenom skrivs en till.

Vi valde även vid detta tillfälle att skriva fem integrationstester. För detta användes annotationer som `@SpringBootTest @AutoConfigureMockMvc @Transactional och @ActiveProfiles`.

4 Resultat

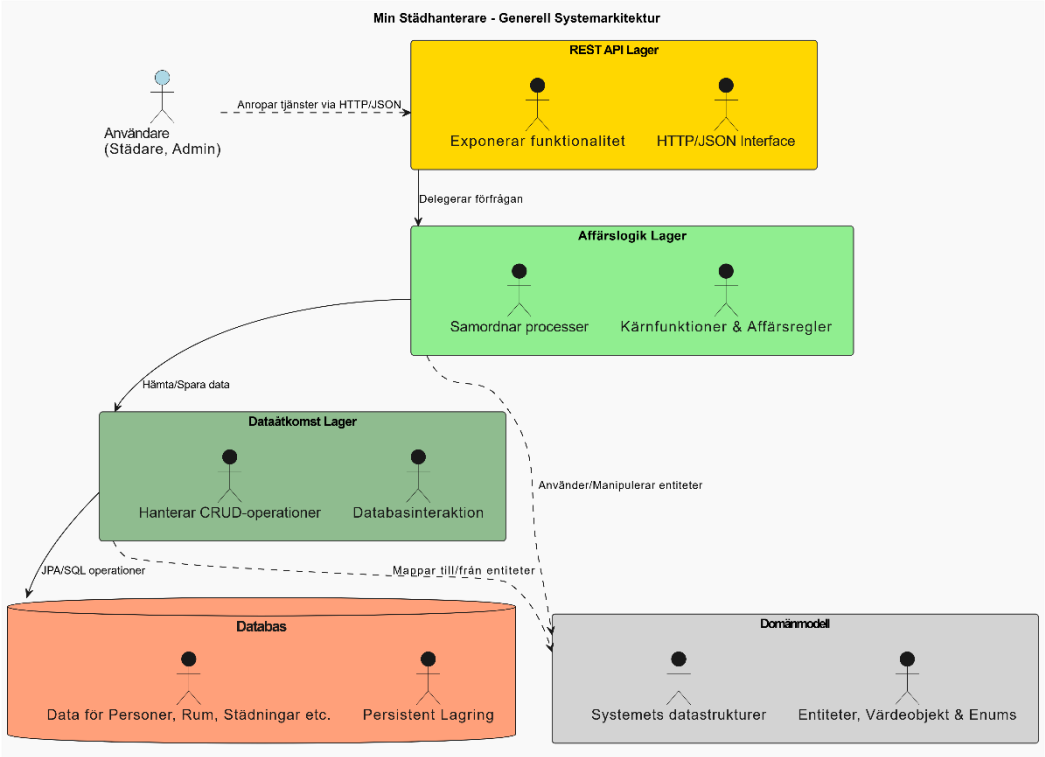
Resultatet av detta utvecklingsprojekt är ett skapat springboot-baserat städsystem som är uppbyggt i fyra olika lager, Domain, Repository, Service och Rest. Lagerna är uppdelade enligt ”Single Responsobility”-principen där som exempel Service-lagret har hand om all kontrollogik av indata, Rest-lagret har enstaka kontroller av indata i DTO:rna med Springboots inbyggda funktioner som `@Valid` och `@NotNull` och all kommunikation mellan databasen och systemet sker via Repository-lagret med hjälp av Java Persistence API som förenklade kommunikationen ordentligt med databasen för att slippa skapa egna DAO-klasser med specificerad SQL-kod. I DomänLagret finns alla klasser som är till för skapande av objekt, till exempel Person och Cleaning.

Systemet kan hantera olika användare som kan städa, få utbetald lön specifikt insatta datum, personer kan ha olika roller (Admin och User) och har därmed olika rättigheter i applikationen, man kan registrera städningar med 15 minuters mellanrum samt övrig funktionalitet. Till utvecklingen av mjukvaran togs också i beaktning till de non functional requierments som satte de icke funktionella kraven till exempel att inga UUID:n får lagras i sökvägen och att ett UUID endast får returneras till anroparen om det är en administratör.

Resultatet av arbetssättet som användes gjorde att 26 av 29 backlog-issues genonfördes. Dessa issues hanterades metodisk i en SCRUM-board som sattes upp via funktionen ”Issue boards” som finns i Gitlab som är till för just detta. Där valdes kolumnerna ”Backlog”, ”Sprint”, ”In-Progress”, ”Testing”, ”Examining” och ”Finished”. För varje sprint kördes ”Planning Poker” på de befintliga backlogsen för att bestämma hur många som skulle göras under varje sprint. Detta medförde att arbetet genomfördes metodiskt med struktur i iterationer vilket gav resultatet att systemet har många fungerande funktioner

4.1 Generell illustrering av systemet

Städssystemet kan på ett generellt sätt beskrivas enligt komponentdiagrammet i figur 7, där diagrammet visar systemets grundläggande komponenter och deras beroenden.



Figur 77 Visualiserar en högnivå vy över stadssystemet som presenteras med ett komponentdiagram.

Då en användare interagerar med systemet kommer den att anropa restlagret som tar emot inkommande HTTP-förfrågningar. Där restlagrets primära funktion är att utföra validering av inkommande data, översätta förfrågningarna samt returnera ett svar.

Kärnan i applikationen är då servicelagret som kapslar in och bearbetar av förfrågningarna. Där den säkerställer att all logik utförs enligt väl definierade krav.

Dataåtkomstlagret kommer då att hantera all kommunikation till systemets persistenta lagring där den tillhandhåller logik för att utföra CRUD-operationer.

I domänmodellen så hanteras det systemets centrala datastrukturer, entiteter samt enumeringar.

Systemet kan utföra operationer på inkommande och utgående data tack vare dataåtkomstlagret i systemet och det ske på ett kontrollerat sätt.

5 Diskussion

Det städsystem som har utvecklats under detta projekt har gjort att vi behövt ta hänsyn till både tekniska krav samt metodologiska krav. Där vi tekniskt har använt oss av Spring Boot och metodiskt har vi strävat efter att arbeta efter den agila utvecklingsmetoden Scrum.

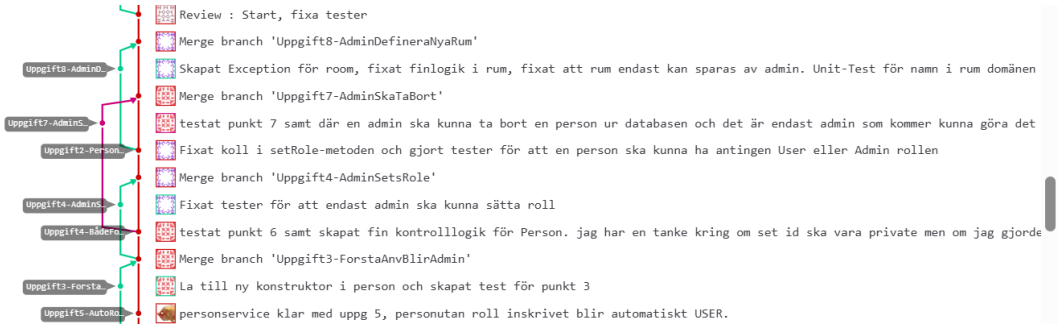
5.1 Scrum

I förhållandet till artikeln *Why and how is Scrum being adapted in practice: A systematic review* har vårt arbetssätt avvikit på flera punkter från det rekommenderade sättet att arbeta med scrum. Vi har dock inkluderat delar som daily standup, sprintar (med kortare perioder), productbacklog, sprint backlog, pokring, sprint retrospective, scrum board och viss kontakt med scrum Master.

Vi har försökt förhålla oss till den artikelns syn på Scrum så mycket som möjligt. Utan strikta arbetsdagar, aktiva intressenter, sprint reviews, dagliga sprintar och en uppdaterad backlog har det dock blivit svårt att fullt tillämpa det agila arbetssättet som brukar vara vanligt i arbetslivet. Dock så har vi ändå fått en överblick över hur det kan se ut att jobba utifrån Scrum, vilket är en fördel.

5.2 Arbetssätt

En viktig lärdom från projektet är hur man arbetar effektivt i team. Vi har blivit väl insatta i versionshanteringssystemet Git, vilket möjliggjorde effektivt parallellt arbete. Vi har lärt oss att brancher kan representera features och man får då historik över projektets arbete (se figur 8).



Figur 88 - git-trädets struktur under arbetet i gitlab.

Vi har även fått med oss riktlinjer i att arbeta med Git som exempelvis, commita ofta, gör regelbundna pulls och ha välbeskrivna commits.

Utöver git lärde vi oss styrkan av par-programmering och mob-programmering, då vi märkte att navigatorn hade större roll än förväntat. Att sitta och kolla när någon kodar ledde till mycket intressanta diskussioner och olika sätt att lösa en uppgift. Samt var det lättare som navigator att tänka på designen och slarvfel som inte den som kodade märkte. När vi körde mob-programmering på vissa uppgifter märkte vi att alla fick en gemensam förståelse och att alla blev överens över ett tankesätt.

5.3 Kod

Genom att använda oss av det tekniska ramverket Spring Boot som vi i gruppen tidigare inte hade arbetat med, vilket ledde till att vi stötte på problem vid integrationstester samt problematik med JUnit. Vi var exempelvis tvungna att flytta på delar av vår filtreringslogik från repositoryt till service klassen för att kunna skapa isolerade och mockade tester.

Under projektets gång blev det också en utmaning kring då vi bestämde oss för att serviceklasserna endast får tala med andra serviceklasser och inte deras andra repository klasser förutom deras egna. Detta ökade i sin tur den skiktade arkitekturen som vi eftersträvat men där vi då riskerade cirkulära beroenden men som enkelt kunde lösas genom att använda oss av lazy loading.

Den vetenskapliga artikeln *Design and Implementation of Tourism System Based* gav oss en bra inblick i funktionalitet hos springboot som skulle gynna oss inom detta utvecklingsprojekt, då de har utvecklat ett liknande system. Genom att använda oss av Spring Boot gav det oss en snabbare start då det minskade vårt behov till att skriva onödig boiler plate code, samt på ett enkelt och effetkivt sätt använda oss av den inbyggda funktionaliteten kring dependency injection som underlättade vårt arbete.

Referenser

[1] M. Hron och N. Obwegeser, ”Why and how is Scrum being adapted in practice: A systematic review”, *J. Syst. Softw.*, vol. 183, s. 111110, jan. 2022, doi: 10.1016/j.jss.2021.111110.

[2] H. Zhao, H. Li, Q. Yu, H. Zhang, och Q. Liu, ”Design and Implementation of Tourism System Based on Spring Boot”, *2024 3rd Int. Conf. Artif. Intell. Comput. Inf. Technol. AICIT Artif. Intell. Comput. Inf. Technol. AICIT 2024 3rd Int. Conf. On*, s. 1–5, sep. 2024, doi: 10.1109/AICIT62434.2024.10729996.