Denni Wernersson - dw222fy
Jesper Lindström - jl255en

1dv609 - Software Testing
Assignment 2
Kladd & Dokumentation

1 Projektbeskrivning

“A short description of the project.”

Validator är ett projekt som används för att validera strängar. Projektet består av olika “validatorer” (funktioner) som kan testa och verifiera ifall strängen i argumentet är “korrekt” enligt verktygets regler. Till exempel så finns det validator för e-mail adresser, personnummer, telefonnummer, kredit- och bank-kortnummer, URL:er, datum, osv.

Projektet innehåller också verktyg för att sanera strängar, alltså ändra på den införda strängen enligt sanerings-verktygets regler, eller för att läsa ut någon särskilt information. Till exempel finns det sanerare som enbart behåller karaktärer i strängen enligt en whitelist, också en som eliminerar karaktärer enligt en blacklist. Det finns också sanerare som kan läsa ut saker som datum och booleska värden från strängar.

Projektet är skrivet helt och hållet i Javascript, idag är det ~11 år gammalt. Första “fullständiga” (1.0.0) release hände för 9 år sedan. Projektet utvecklas öppet på GitHub. Det finns två maintainers (“ägare”), Chris O’hara och Anthony Nandaa. Vem som helst är tillåten att göra bidrag till projektet, idag finns strax över 380 contributors.

1.1 Syfte och mål

“What is its purpose? What does it aim to accomplish?”

Syftet med verktyget är att kunna validera användare-indata. Verktyget riktar sig till användningsområden inom både back-end och front-end. Användningen av verktyget i ett back-end sammanhang hade till exempel kunnat vara validering av formatet på email-adresser innan en ny användare registrerar sig med den adressen till en databas. Verktygets användning inom front-end området används vanligtvis för att placera ett validerings-ansvar på användarens klient, så att det blir svårare för en användare att “bombardera” back-end tjänsten med information att validera. Användningen av verktyget inuti klientsidan ger också möjlighet att “snällare” vägleda användaren till att ge rätt sorts input, istället för att tvinga användaren att vänta på ett svar (vägledning) ifrån serversidan.

2 Projektspecifikation och krav

“Provide an overview of the requirements and specifications.”

Projektet har inga krav som är “skarpt” specifierade, alltså ingen formell kravspecifikation. Däremot finns det beskrivning av de olika verktygens förväntade betéenden.
Inga skarpa krav specifierade, men det “självklara” är väl att verifiera input som är “korrekt”, och att inte råka släppa igenom input som är “fel”. Alltså att validera rätt och inte fel :)

Validatorernas förväntade betéendet finns specificerade i projektets README, detta är nog det närmsta man kommer till en formell kravspecifikation. Betéendespecifikationen beskriver och förklarar hur man som utvecklare använder projektets verktyg, samt funktionaliteten av de olika verktygen.

2.1 Intressenter och risker

“Stakeholders, risks, evaluation, etc.”

Projektet och dess verktyg används främst av utvecklare som behöver validera data/information som de tar emot från sina användare. Det finns väldigt många projekt som är beroende av validator, enligt npmjs har ungefär 5500 projekt på plattformen listat validator bland sina beroenden.

Några exempel på “större” projekt som använder validator:

- Express-validator (~360000 nedladdningar i veckan), en express middleware för att lättare använda validator i samband med Express.
- Sequelize (~1,1 miljoner nedladdningar i veckan), ett databashanteringsverktyg

Projektet är väldigt populärt; runtomkring 6 miljoner nedladdningar från npmjs varje vecka och populariteten ser ut att växa stabilt. På grund av validators breda användning inom utveckling i javascript är det viktigt att projektets verktyg “gör rätt”. Ifall verktyget skulle brista och resultera i felaktig informationsvalidering så kan med all sannolikhet leda till negativa konsekvenser för projektets användare. Alltså är det viktigt att projektet underhålls för att säkerställa kvalitén av sina verktyg. Det läggs därför relativt stor vikt på testningen i projektet.

3 Tidigare, nuvarande och framtida utveckling

“The past, present, and future development of the project.”

3.1 Tidigare

Projektet har varit utveckling sedan länge; för 11 år sedan startades projektet, 2 år senare (9 år sedan) sattes projektet i det första “färdiga” läget (v1.0.0). Den första versionen som finns versionshanterad på GitHub är 4.0.0. Mycket har hänt sedan projektets början, omstrukturering av kod, tillägg av nya verktyg, utökad testning, osv. Utvecklingen över åren har varit relativt “jämn”, alltså inga perioder som projektet har stått stilla/legat i ide. Det finns ett fåtal “spikar” av aktivitet inuti projektet om man tittar på GitHubs aktivitetsspårningsverktyg, men de beror främst på ett större antal småförändringar inuti projektdokumentationen.

3.2 Nuvarande

Den aktuella versionen av projektet är idag v13.7. Projektet utvecklas aktivt; utvecklare committar ofta, ett flertal PR:s som ligger i väntan på code review och det finns väldigt många forks där folk jobbar på bitar av projektet. Just nu har projektet två maintainers (ägare) och runtomkring 380 contributors. Idag är utvecklingstakten/hastigheten ungefär likadan som den varit under hela projektets livstid.

3.3 Framtida

Det finns ingen formell plan över vad som är planerat för projektets framtid, men projektets GitHub-repo har ganska många issues upplagda som berör den framtida utvecklingen. Till exempel issues som handlar om buggar och issues som berör begäran/efterfrågan av nya features.

Det finns ingenting inom projektets diskussion eller dokumentation som tyder på att projektet är på väg att läggas ned (deprecate) eller någonting i den stilen, allt som syns utifrån tyder bara på ett hälsosamt projekt som är välunderhållet och har all möjlighet att förbli på det viset.

4 Nuvarande testningsstrategi

“The current testing strategy, the kinds of tests being performed, and how the testing is reported on.”

Validator genomför testning av sin kod genom användandet av automatiska enhetstester.

Utvecklarna mäter kod-basens code coverage, i nuläget täcks 100% av koden av automatiska enhetstester. Projektets code coverage mäts varje gång de automatiska testerna exekveras. Ingen form av mocking/stubbing/spying/faking i de automatiska testerna. Eftersom projektets kod inte är objektorienterad är det svårt att implementera någon form av mocking.

Projektet har CI/CD-pipelines (GitHub Workflow) som exekverar alla automatiska enhetstester varje gång man pushar mot Master-branchen, samt varje gång en PR (Merge request) begärs. Den automatiska pipelinen sparar också resultatet av code coverage mätningen.

4.1 Verktyg och bugghantering

“What tools are used? How do they handle bugs?”

4.1.1 Verktyg

Den automatiska enhetstestningen genomförs med test-ramverket “Mocha”. Detta verktyget är väldigt vanligt att kombinera med assertion-biblioteket “Chai”, men detta projektet använder istället enbart Mocha i samband med egenskrivna helper-funktioner som gör test-koden mer läsbar och lättare att utveckla/utöka.

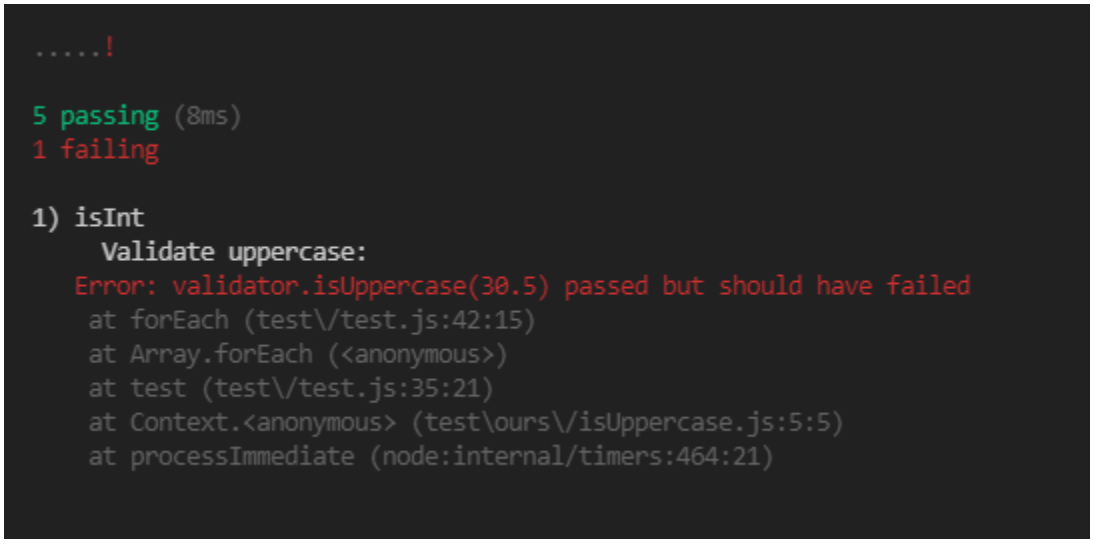
Code coverage mäts med hjälp av verktyget “Istanbul”. Verktyget mäter fyra sorters code coverage: (1) Statements, (2) Branches, (3) Functions och (4) Lines. Det verkar inte finnas något stöd för att mäta loop coverage i Istanbul. När code coverage mäts så sparas resultatet i en XML-fil.

GitHub Workflow, som är projektets automatiska pipeline, ansvarar för att exekvera de automatiska enhetstesterna och mäta kodbasens code-coverage varje gång någon pushar mot- eller gör en PR mot master-branchen.

XML-filen som innehåller code-coverage resultaten filen skickas med hjälp av projektets pipeline vidare till plattformen “codecov.io”, som är ett verktyg för att hantera code coverage och de krav och önskemål däromkring. Codecov.io är ej åtkomligt för oss studenter som utomstående, därför kan vi inte se de code-coverage krav som kan stå specificerade där. Vi antar att utvecklarna siktar på 100% code coverage, baserat på att det är deras nuvarande coverage.

4.1.1 Bugghantering

Buggar som fångas av de automatiska enhetstesterna presenteras i konsollen när det testas lokalt. För att projektets CI/CD pipeline ska lyckas måste alla tester passera, om ett test misslyckas resulterar det alltså i en misslyckad pipeline.



En bugg som fångats av ett automatiskt enhetstest. Rapporterar funktionsnamn, indata, faktiska betéende och förväntade betéende.

Buggar som upptäcks på annat vis kan rapporteras som en bug-report issue inuti projektets issues på GitHub. Det finns en mall att följa för hur en buggrapport ska se ut. Vem som helst har fritt fram att jobba på att laga buggen. När en utvecklare är klar med sitt buggfixande ska hen göra en PR mot master-branchen, när PR:en har gått igenom en code review kan den beviljas av en maintainer och mergas mot master.

5 Vår testning

"Document your testing performed."

5.1 Variant av testning

"Exploratory testing is optional, what kind of structured testing will you do?"

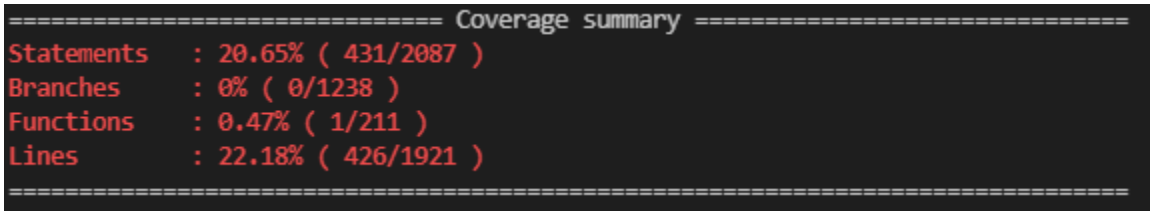
Vi planerar att utföra testning av projektet genom att utnyttja samma verktyg som utvecklarna. Vi skriver egna tester i Mocha, tillsammans med de test-helpers som utvecklarna har skrivit. Vi baserar testernas indata och förväntat utdata på betéendespecifikationen som finns i projektets README.

Ifall det finns tid för det så ska vi försöka bygga ett eget testverktyg för att simulera ett användarregistrering-scenario, och då utnyttja/testa projektets "isEmail()" och "isStrongPassword()" validatorer.

5.2 Metod och Resultat

"What did you do and what did you find?"

Vi har genomfört (hittills) tester mot 17 st funktioner som är implementerade i validator projektet. Vi skrev egna testfiler för varje funktion, istället för att ha allt i en jätte-fil som projektet har just nu. Sedan skrev vi tester i en liknande stil av de tester som redan är specificerade i projektet. Vi valde vår indata och baserade våra förväntningar på verktygens/funktionernas betéendespecifikation. Vi strävade efter att nå 100% coverage på de funktioner vi testar. Något vi la märkte till direkt var att ingen av istanbuls code coverage kriterier faktiskt börjar på 0%.



Coverage Summary utan någon testning alls

Statement coverage började på 20,65%, Branches på 0, Functions på 0,47%, och Lines på 22,18%. Istanbul väljer att räkna code-coverage för alla "import" och variabeldeklarationer utan att det sker någon testning, alltså blir det ganska mycket "gratis" coverage inuti sammanfattningen. Det finns också en fil med namnet "alpha" som enbart exporterar en massa konstanter, i projektet används de i princip som enums, iallafall så räknas hela den filen också till code-coverage sammanfattningen utan någon testning.

Efter implementationen av våra tester (17 test-suites hittills) sammanfattade Istanbul vår code-coverage såhär: *Coverage summary efter implementering av 17 olika tester*

```
============================= Coverage summary =====
Statements   : 25.35% ( 529/2087 )
Branches     : 7.43% ( 92/1238 )
Functions    : 10.9% ( 23/211 )
Lines        : 26.97% ( 518/1921 )
================================================================
```

Code coverage efter vår testning (hittills)

Projektet har några util funktioner (i stil med typ toString() och sånt) som används inuti validator-funktionerna, vilket resulterar i lite "extra" coverage för samtliga kriterier.
(Exempelvis så står det att vi har täckt 23 funktioner med tester, men vi har egentligen bara skrivit tester för 17 st)

Våra test-suites täcker våra utvalda funktioner-under-test med 100% coverage.
Hundra procent coverage.

File		Statements		Branches		Functions		Lines
alpha.js		100%	40/40	100%	0/0	100%	0/0	100%
blacklist.js		100%	3/3	100%	0/0	100%	1/1	100%
contains.js		100%	9/9	100%	2/2	100%	1/1	100%
equals.js		100%	3/3	100%	0/0	100%	1/1	100%

Exempel på vår coverage rapport genom användning av istanbul-html-reporter

5.3 Upptäckter

Verktyget är lätt/simpelt att testa med hjälp av utvecklarnas test-helpers. Testningen sker i princip bara genom att specificera indata och förväntat utdata. Man kan göra det väldigt lätt för sig att skriva många tester om man bara använder rätt verktyg för det (eller skapar ett eget).

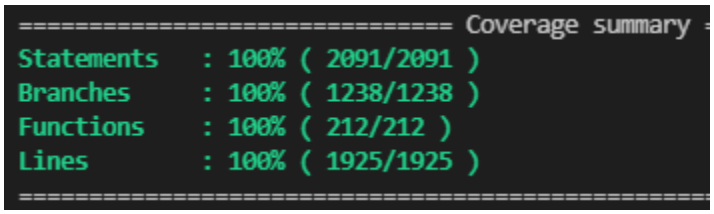
Utan mocking/stubbing/liknande kan det vara svårt i vissa fall att lokalisera och vara säker på vart en bugg verkligen ligger. I projektet finns ett flertal funktioner som internt använder sig utav andra validator-funktioner för att lösa ett problem. Ifall en funktion är buggig så kan det då vara svårt att vara säker på var källan till problemet ligger.

6 Deras testning

“Document the pre-existing tests and their results.”

Idag finns 224 tester som kan exekveras, alla passerar och inga misslyckas. Det tar ~271ms att köra själva testerna, men det är ganska mycket som ska ske innan några faktiska tester exekveras när man kör test-skriptet. Innan testningen startar så ska projektet lintas och kompileras genom transkompilatorn “Babel”, vilket tar ~17 sekunder. Detta är relativt snabbt, men man hade kunnat bryta bort lintningen och transkompilationen ifrån testningen för att göra test-exekveringen lite mer smärtfri. Kombinationen av ansvar i testskriptet kan tyda på att testningen riktar sig till att exekveras i projektets pipeline, snarare än lokalt.

Deras testning täcker kodbasen med 100% code coverage. Istanbul mäter coverage för statements, branches, functions och lines, men verktyget kan inte spåra loop coverage. Av den anledningen vet vi inte hur projektets loop coverage ser ut.



Summary av deras code coverage