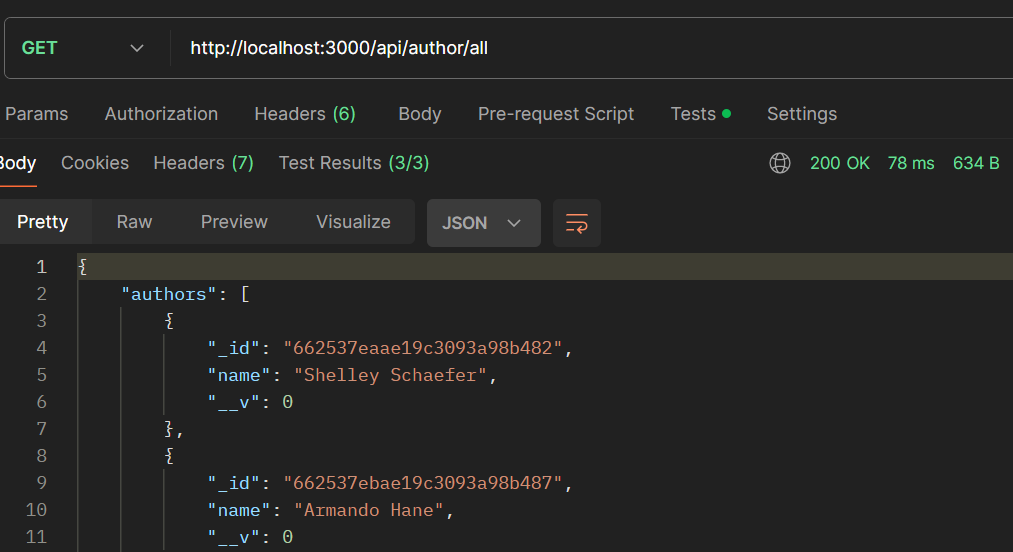
Min testprocess

Det var en väldigt givande uppgift som jag till en början hade svårt för att komma igång med. Måndagen och tisdagen då vi fick uppgiften var jag mycket osäker på vad jag skulle göra och hur. Det kändes som jag missat något grundläggande om hur man ”kopplar ihop” Visual Studio Code och postman. Därför förlorade jag mycket tid åt att kolla igenom lektioner de dagarna för att hitta bitarna som saknades. Men när jag väl kom igång på onsdagen gick det fort. Jag tänker inte förneka att vi samarbetade en del i Ronneby som grupp men alla gör såklart inte exakt likadant och vi alla lärde oss mycket av vårt samarbete.

Uppgifterna var klara på mindre än en vecka för att kunna lägga mer tid åt detta dokumentet och README.md filen.   
  
Det som var svårt med den här uppgiften var att flera av punkterna var svårlästa och otydliga med vad man ville ha exakt och många var även svåra att säga vad som är manuellt och automatiskt.

Jag är nöjd med det jag lämnar in. Kunde det varit bättre? Säkerligen, men mina kunskaper sträcker sig inte längre än det jag visat här. Uppgiften var väldigt bra och jag vet att jag kommer dra nytta av den i framtiden då min handledare på min LIA plats till våren-25 blev helt till sig när de hörde att jag hade ett projekt i MongoDB.

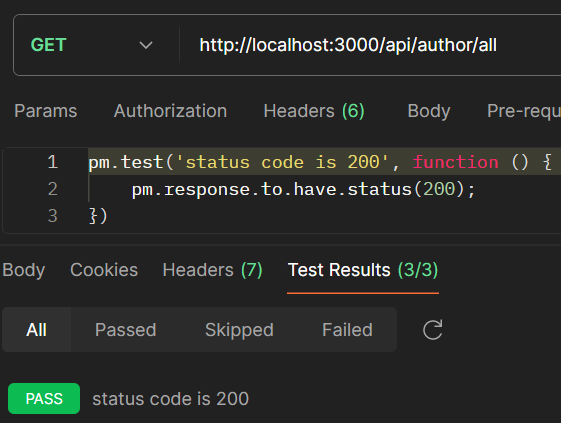
Med det sagt hälsar jag trevlig läsning!

Punkt 1:1

Det första manuella testet innebär att skapa en get-request och få fram den information man söker.

I mitt exempel valde jag att göra en get-request på alla författare i min databas. Jag valde därför api/author/all i js för att inte krocka med andra tester som har bara api/author. Testet gjordes för att kontrollera att requesten hämtades på korrekt sätt. Resultatet blev att man kan se 200 OK efter att requesten skickats.

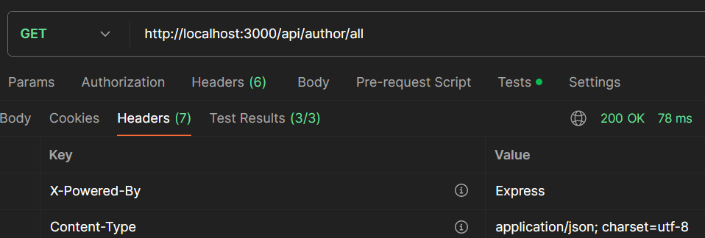
Figur 1: Man kan se statusmeddelandet 200 OK med grön text till höger

Punkt 1:2

Det första automatiska testet innebär att kontrollera att rätt statuskod registrerats.

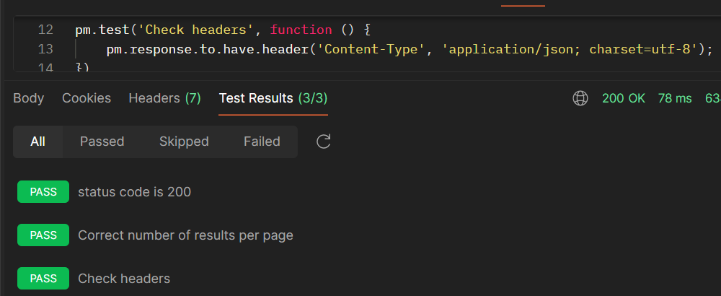
Jag använder samma exempel som ovan och skrev då ett automatiskt test: som är markerat på bilden där man förväntar sig statuskod 200. Testet gjordes för att man på ett överskådligt sätt ska kunna se vilka requests som ger rätt statuskoder för att snabbt kunna åtgärda fel som uppstår.

Figur 2 Längst ned står det PASS vilket betyder att testet godkändes

Punkt 2:1

Det andra manuella testet innebär att kontrollera att headers content-type är av rätt format (application/json). I samma get- request som i tidigare exempel kan vi här även se att Content-type är av rätt format.

Figur 3   
Längst ned till höger ser vi att content-type är av rätt format

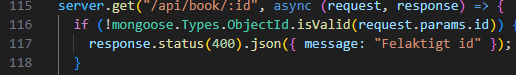


Punkt 2:2

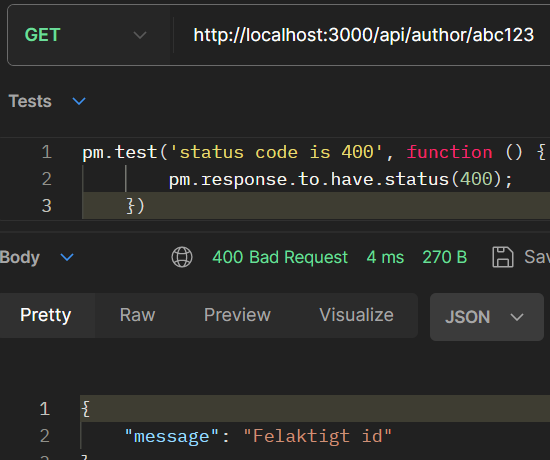
Det andra automatiska testet innebär att kontrollera att content-type är av rätt format. Det kan man göra med **pm.response.to.have.header()**funktionen där man förväntar sig ett visst svar.

Figur 4 På bilden kan vi se att testet Check headers är godkänt och att content-type är av rätt format

Punkt 3:1

Det tredje manuella testet innebär att kontrollera att API ger statuskod 400 vid felaktig användning. Jag valde att lösa det genom att lägga koden på bilden i en av mina get- requests i js. Koden betyder att om man fyller i ett felaktigt mongoose ObjectId i url fältet ska ett 400 meddelande visas. Testet gjordes för att försäkra oss om att rätt kod syns vid rätt tillfälle för att underlätta för användare att se över vad de gjorde fel då ett 400 meddelande innebär felinmatning.

Figur 5 Eftersom koden står innanför en get-request som används i flera fall kommer inte den är delen av koden att utlösas om man inte ändrar på id:t som är sparat som en variabel.

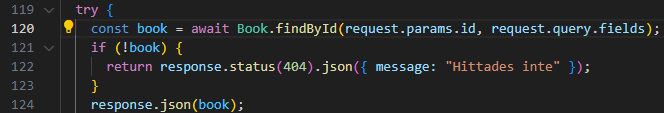
Punkt 3:2

Det tredje automatiska testet innebär att kontrollera att API ger statuskod 400 om man exempelvis fyller i fel id i en get-request. Jag gjorde en ny get-request men kunde använda mig av js koden från 3:1 för att skapa ett 400 fel. På bilden ser vi testkoden i postman och responsen jag fick.

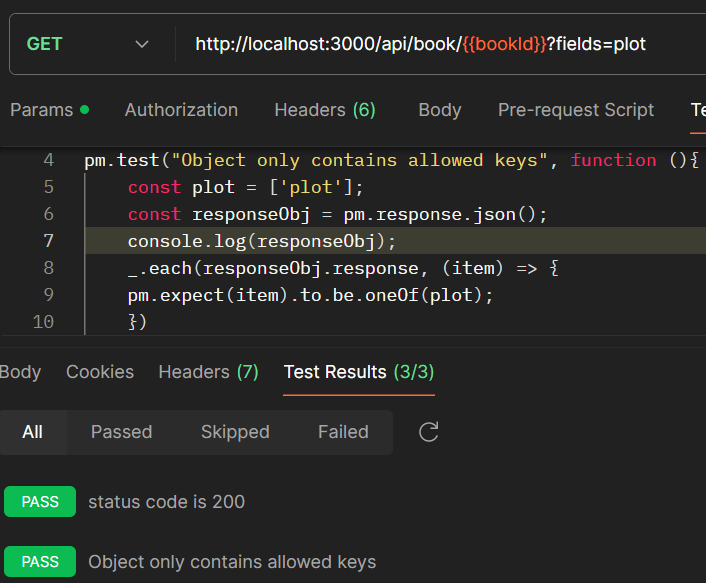
Testet gjordes för att säkerställa att rätt meddelande syns vid felinmatning.

Punkt 4:1

Det fjärde manuella testet innebär att kontrollera att API kan visa delar av ett objekt med query parametrar. Jag använde **request.query.fields** ihop med **findById()**.

I url fältet i postman kunde jag sedan specificera vilket fält jag ville få utskrivet ur den valda boken. Resultatet blev att den valda bokens plot (handling) visades. Testet gjordes för att kunna plocka ut specifika delar från Json objektet.

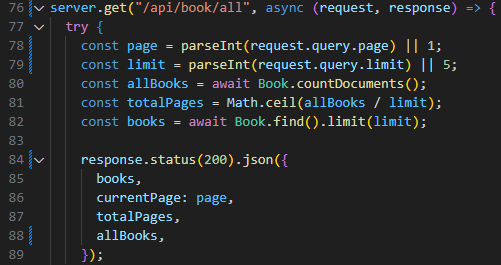
Punkt 4:2

Det fjärde automatiska testet innebar att kontrollera att enbart de tillåtna delarna visades i ett filtrerat resultat. Som jag nämnde i 4:1 valde jag att testa visa Plot.  
Jag sparar ner plot i en array och jämför om det som visas i response är samma som det jag sparat i arrayen. Anledningen till att jag sparade ner det i en array var för att jag ville kunna lägga till fler filter men det fungerade inte. Men jag lät det ligga kvar som en array i hopp om att lista ut det om tid fanns kvar mot slutet.

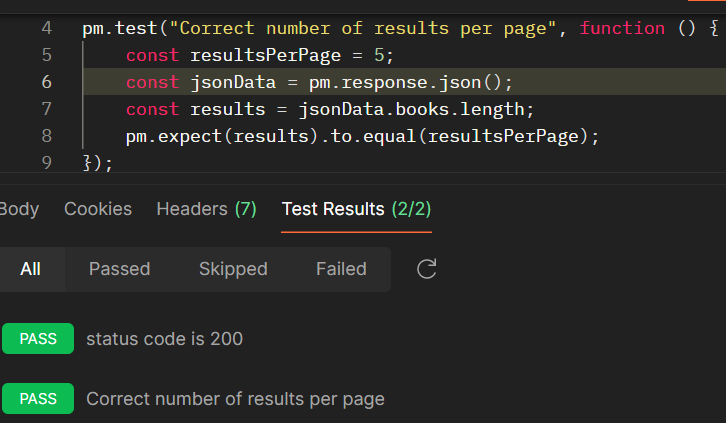
Testet gjordes för att kunna kontrollera vad som syns. Det hade kunnat vara sekretessbelagd information i övriga objektet som exempelvis sjukvårds journaler där man skulle behöva gå djupare för viss information.

Punkt 5:1

Det femte manuella testet innebär att paginera och se till att enbart ett antal objekt syns på varje sida i resultatet när man förväntar sig ett stort antal objekt.

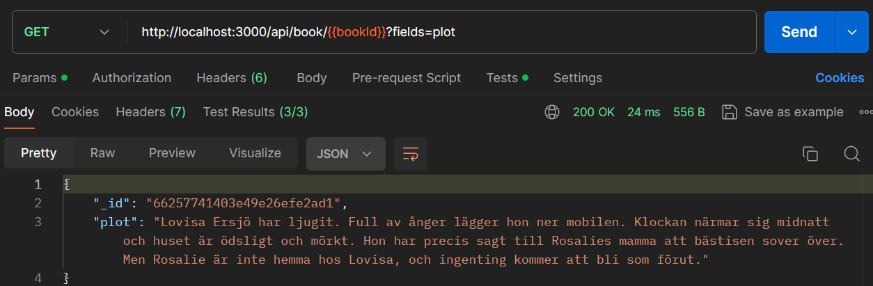
Jag sparade ner vilken sidan jag är på, satte en limit på 5 objekt/sida och räknade alla böckerna och bad vsc räkna ut antaletböcker /5 för att få fram hur många sidor totalt resultatet är på. Sedan sparar jag ner alla böcker tillsammans med limit i books och förväntar mig en statuskod 200 och ett resultat som visar alla böckerna med max 5 på varje sida samt nuvarande sida av totala antalet sidor och antalet böcker visas. Detta var precis det resultatet jag fick förutsatt att jag i url fältet i postman har **api/book/all?page=1?limit=5**. Utan sista biten av url fungerar det inte.

Testet gjordes för att kontrollera att man kan paginera och välja antal visningar per sida.

****Punkt 5:2

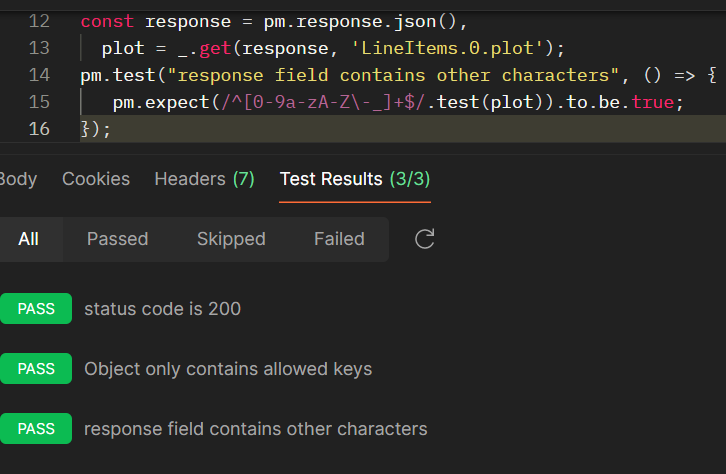
Figur 6 På bilden ser man att testet ”Correct number of results per page” är ett godkänt test.

Det femte automatiska testet innebär att kontrollera att pagineringen fungerar korrekt. För att fortsätta på punkt 5:1 använde jag en gräns på 5 objekt per sida. Jag spara ner siffran fem, min response och räknar antalet objekt i min response som visas och förväntar mig att det ska vara 5. Testet är till för att spara tiden det skulle ta att räkna objekten manuellt samt ge en tydlig överblick om det fungerar eller ej.



Punkt 6:1

Det sjätte manuella testet innebär att kontrollera specialtecken visas. På bilden ser vi att å, ä och ö visas i response då texten är skriven på Svenska. Syftet med testet är att säkerställa att alla typer av tecken går att visa, även de utanför engelska tangentbordet.

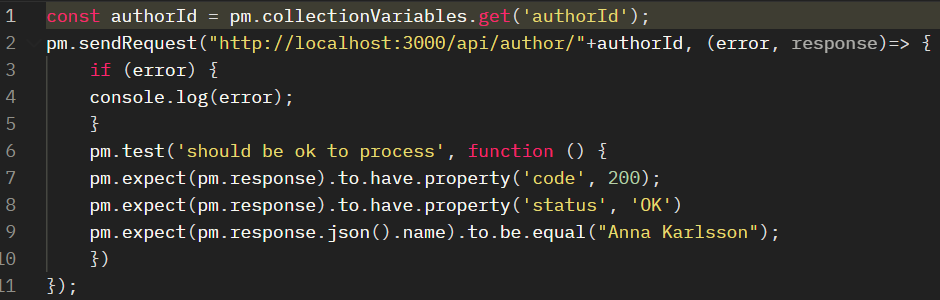
Punkt 6:2

Det sjätte automatiska testet innebär att kontrollera att specialtecken visas. Jag testar specialtecken med testet bredvid som finns i samma get-request där jag filtrerar ut plot som jag tidigare nämnt.

I testet utgår jag från att andra tecken än /^[0-9a-zA-Z\-\_]+$/ kan finnas med, vilket det gör eftersom det är en text på Svenska med å, ä och ö.

Syftet med testet är att kontrollera att special tecken skrivs ut och är läsbara.

Punkt 7:1-2

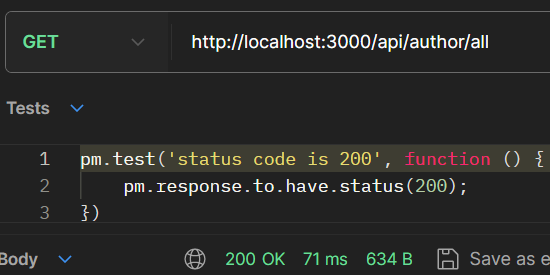
Det sjunde manuella och automatiska testet hänger ihop då det innebär att köra flera test samtidigt och se hur API hanterar det. Jag gjorde ett automatiskt test för detta där jag testar att koden är 200, status är OK och namnet på författaren är Anna Karlsson. Syftet med testet är att kunna skicka flera requests samtidigt för att se att API går igenom alla punkterna för ett godkänt test. Testet är godkänt förutsatt att man har kört Put-requesten update author först då testet fallerar för att författaren har kvar sitt första efternamn, Jansson annars.

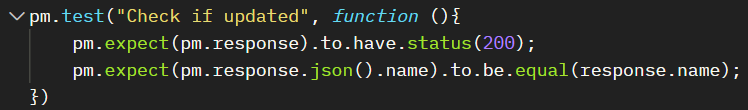
Punkt 8:1

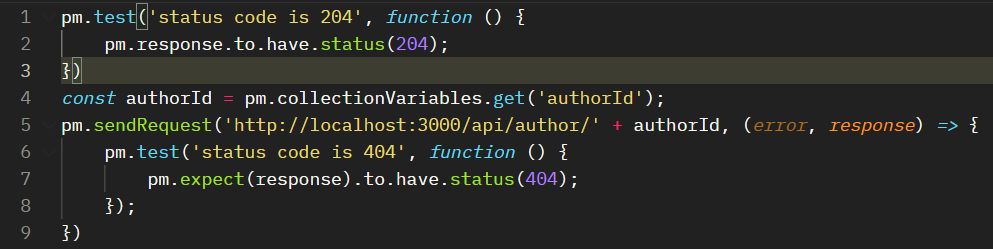
Det åttonde manuella testet innebär att kontrollera att post, get, put och delete requests visar rätt statuskoder samt når sina endpoints. För att manuellt testa dem körde jag igenom dem en efter en och kunde se att min post fick medddelande 201 och syntes i mongoDB compass. Min get fick meddelande 200 och hämtade objektet i fråga. Min put ändrade på objektet i MongoDB compass och min delete tog bort objektet från MongoDB compass.

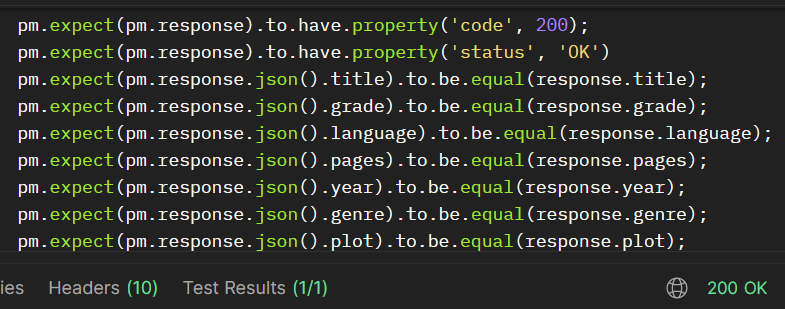


Punkt 8:2

Det åttonde automatiska testet innebär att kontrollera statuskoder och endpoints. Jag gjorde det genom att skriva olika tester. Alla fyra har en *pm.response.to.have.status().*

I put-requesten har jag även ett test för att kontrollera att namnet blev uppdaterat.

I delete-requesten gjorde jag ett kontrolltest för att se om objektet verkligen var borta och fick då rätt statuskoder. Testerna gjordes för att säkerställa att rätt data och statuskoder sparas eller visas.

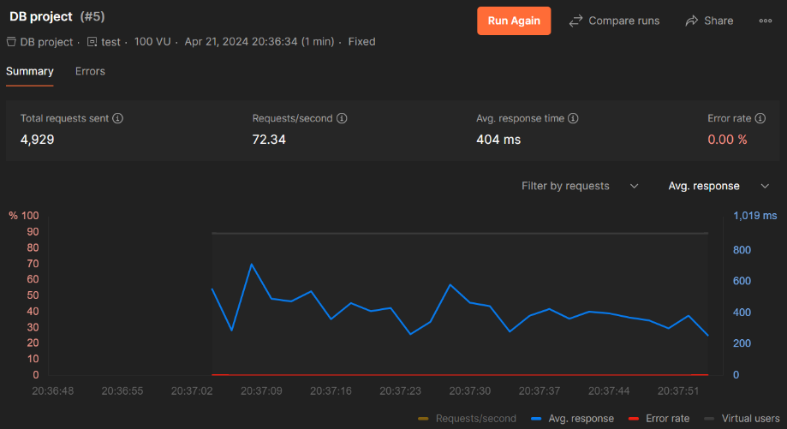
Punkt 9:1-2

Det nionde manuella och automatiska testet innebär att kontrollera att en put -request sparar ner rätt information med hjälp av flera tester i samma test. Här kollar jag att testet fick kod 200 och status ok. Jag testar att de olika fälten är sparade som det inmatade.

Testet gjordes för att säkerställa att förändringar sparades korrekt.

Punkt 10:1-2

I postman finns det ett verktyg för att simulera användare som gör request under en utsatt tid för att se hur API hanterar det. Det otydligt hur man skulle göra testet manuellt.

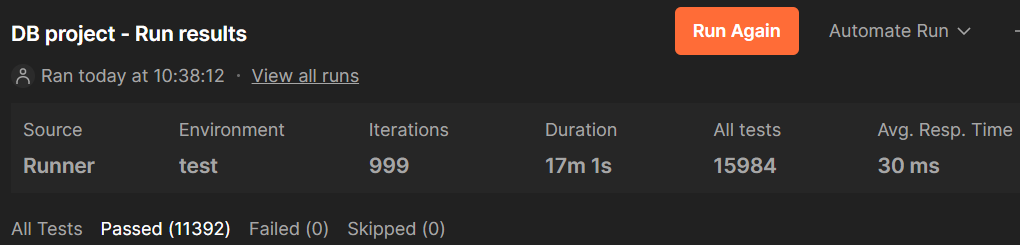
Testet gjordes för att pressa API till max för att upptäcka brister.

På bilden ser man att testet kördes under en minut med 100 användare. Tillsammans sände de 4929 requests med en genomsnittlig responsetid på 404 millisekunder utan errors.   
Resultatet är dock beroende av att requests görs i rätt ordning då de är beroende av de satta variablerna som är bookId och authorId.

Punkt 11:1

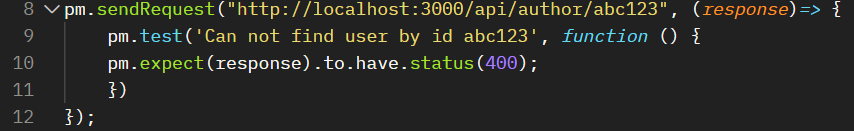
Det elfte manuella testet innebär att kontrollera att API bibehåller data efter exempelvis bortkoppling mot servern. Jag testade detta genom att koppla bort anslutningen i mongoDB Compass och återansluta. Resultatet av testet var all data fanns kvar. Testet gjordes för att kontrollera att data inte försvinner vid bortkoppling.

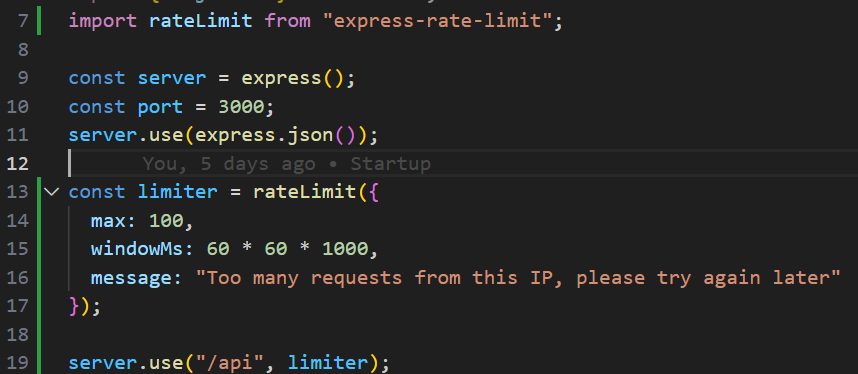
Punkt 11:2

Det elfte automatiska testet innebär att kontrollera att API bibehåller data efter exempelvis bortkoppling mot servern. Jag körde run collection med 1000 iterations och bröt connection i mongoDB compass och förväntade mig fallerade tester. Detta va inte var fallet.

Testerna rullade på som vanligt och alla var godkända med goda responstider. Efter rådfrågan av klasskamrater insåg jag att det var fler som upplevde samma **problem** och mindes då hur Thomas brukade säga om Cypress *”It´s not a bug, it´s a function”*. Så är troligtvis fallet här också. Postman måste spara ner en backup av servern och därför går testen igenom oavsett om servern är bortkopplad.

12:1-2

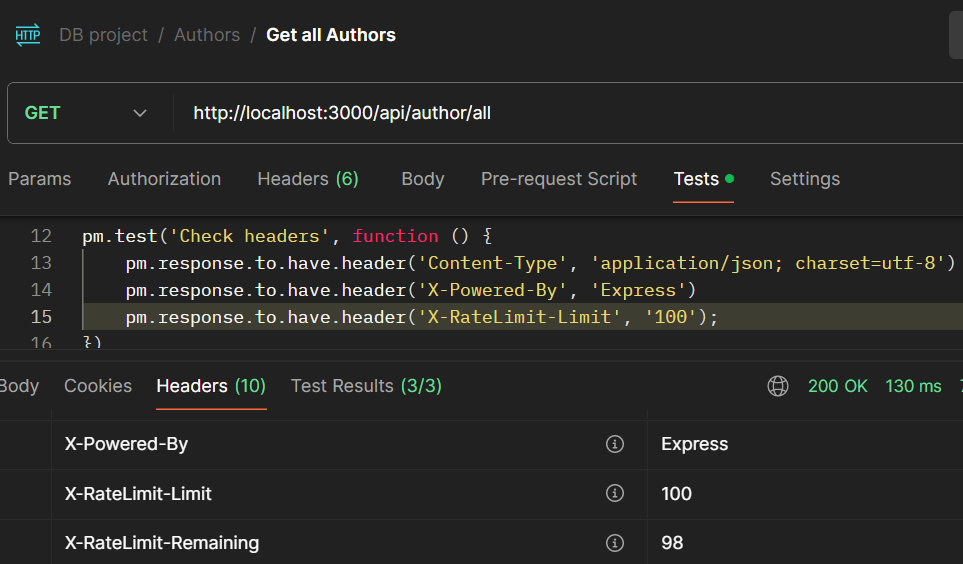
Jag var osäker på hur man skulle testa 12:1 på ett manuellt sätt och har därför inte gjort den. Men det tolfte automatiska testet innebär att kontrollera att rätt statuskod visas om felaktiga parametrar används. Med en **sendRequest** i *get an author by id* och *get a book by id* testar jag url med felaktigt id och om det resulterar i ett 400 meddelande. Testet gjordes för att säkerställa att rätt statuskoder visades om man använder fel parameterar.



Punkt 13:1

Det trettonde manuella testet innebär att implementera logik för ett max antal requests. I samråd med klasskamrater insåg vi att vi behövde använda ett verktyg som heter express-rate-limiter. Vi installerade först verktygen sen importerade vi det till server.js där vi sedan kunde ange ett maxvärde samt en tidsperiod och ett felmeddelande. Med den nya logiken får man nu upp tre nya headers i response.

Punkt 13:2

Det trettonde automatiska testet innebär att kontrollera att rate limit fungerar korrekt. Detta gjorde jag genom att baka in de nya headers i mitt ”check headers” (2:2) test som går igenom med godkänt resultat.

Syftet med testet är att kontrollera att rate limit är implementerat.