# Test av IT-system

Hemtenta: Testverktyg

Luleå Tekniska Universitet
I0015N
VT25

Noah Alvandi

# Innehållsförteckning

**Innehållsförteckning** ..... **2**
**Inledning** ..... **3**
**Puppeteer – verktygsöversikt** ..... **4**
**Jämförelse med andra testverktyg** ..... **5**
Fördelar ..... 7
Nackdelar ..... 7
**Projektsammanhang** ..... **8**
**Testplan** ..... **9**
Syfte och Mål ..... 9
Omfattning och Avgränsning ..... 9
Testtyp och nivå ..... 9
Teststrategi ..... 9
Resurser och Tidsåtgång ..... 9
Leverabler ..... 10
**Testfall** ..... **11**
**Projektkonfiguration och körning av tester** ..... **12**
Systemkrav och beroenden ..... 12
Körning av tester ..... 12
Begränsningar och miljöberoende ..... 12
**Utvärdering av Puppeteer i praktiken** ..... **13**
**Slutsats** ..... **14**
**Referenser** ..... **15**

# Inledning

Syftet med denna hemtentamen är att identifiera, undersöka och praktiskt tillämpa ett testverktyg som inte har behandlats inom kursens ordinarie innehåll. Genom att utforska ett alternativt verktyg ges möjlighet att fördjupa förståelsen för mjukvarutestningens bredd, samt utveckla en kritisk förmåga att jämföra verktyg utifrån faktorer som användbarhet, begränsningar och kvalitetssäkring i mjukvaruprojekt.

Som en del av uppgiften har jag valt att arbeta med testverktyget **Puppeteer**, vilket inte använts i kursens tidigare moment. Verktyget har implementerats i ett verklighetsnära sammanhang — närmare bestämt i en applikation utvecklad under kursen Java II (D0024E): ett boklånesystem för universitetsbiblioteket. Detta ger möjlighet att utvärdera Puppeteer inte bara utifrån dess tekniska egenskaper, utan även i relation till ett konkret system med verkliga krav och användarflöden.

Målet med rapporten är att presentera en översikt av Puppeteer, jämföra det med andra etablerade testverktyg, redovisa en teststrategi med tillhörande testfall samt reflektera kring verktygets styrkor och svagheter i praktiken.

# Puppeteer – verktygsöversikt

Puppeteer är ett Node.js-bibliotek som tillhandahåller ett högnivå-API för att styra webbläsare som Chrome och Firefox via DevTools-protokollet eller WebDriver BiDi. Det används huvudsakligen för att automatisera webbläsarinteraktioner, vilket gör det särskilt användbart för end-to-end-testning av webbapplikationer. Puppeteer körs som standard i headless-läge, vilket innebär att tester kan utföras utan att en visuell webbläsare öppnas.

Verktyget möjliggör simulering av användarbeteende som att klicka på knappar, fylla i formulär och navigera mellan sidor. Det är därför särskilt kraftfullt när man vill testa hela användarflöden i en applikation. Med stöd för både synkron och asynkron interaktion kan man bygga stabila tester som väntar på att specifika element ska laddas eller visas på sidan innan verifiering sker.

Puppeteer används ofta tillsammans med Jest, ett populärt testbibliotek för JavaScript- och Node.js-applikationer. Kombinationen gör det möjligt att skriva modulära och lättlästa testfall, samt strukturera testsviter med tydliga körningsflöden och assertions. Det ger även stöd för parallell testkörning, vilket effektiviserar testningen vid större projekt.

# Jämförelse med andra testverktyg

Det finns flera populära verktyg för automatiserad testning av webbapplikationer. Under denna kurs används Cypress och Playwright som exempel, medan Puppeteer har valts i denna uppgift för att bredda perspektivet. Även Selenium, som är en mer etablerad standard, tas upp i jämförelsen. Dessa verktyg har olika styrkor beroende på testbehov, utvecklingsmiljö, samt hur avancerad och dynamisk applikationen är.

Tabell 1 jämför fyra vanliga verktyg inom webbutvecklingstestning utifrån centrala faktorer:

**Tabell 1**. Jämför fyra populära automatiserade testverktyg – Puppeteer, Playwright, Cypress och Selenium – utifrån aspekter såsom webbläsarstöd, teststrategi (headless eller visuell), språkstöd, parallell körning samt hantering av väntetider. Syftet är att ge en översikt av deras respektive styrkor och begränsningar för att underlätta valet av verktyg beroende på projektets krav.

| Verktyg<br><br>Egenskap | Puppeteer | Playwright | Cypress | Selenium |
|---|---|---|---|---|
| **Utvecklare** | Google | Microsoft | Cypress.io | Selenium Project (OSS) |
| **Stödda webbläsare** | Chrome, Firefox (begränsat stöd) | Chromium, Firefox, WebKit | Chrome, Firefox, Edge | Chrome, Firefox, Safari, Edge, Internet Explorer |
| **Stöder flera flikar/sessioner** | Begränsat | Ja | Nej | Ja |
| **API-design** | Enkel, låg nivå | Mer komplett, hög nivå | Mycket hög nivå och testvänlig | Lägre nivå, ofta via WebDriver |
| **Språkstöd** | JavaScript (Node.js) | Flera språk (Java, C#, Python, Ruby, m.fl.) | JavaScript (Node.js) | Flera språk (Java, C#, Python, Ruby, m.fl.) |
| **Testkörning i headless-läge** | Ja | Ja | Ja | Ja |
| **Snabbhet** | Snabb | Mycket snabb | Snabb, men beroende av DOM-insyn | Långsammare (använder WebDriver) |
| **Stöd för iframes** | Begränsat | Bra stöd | Begränsat stöd | Bra stöd |
| **Debuggerstöd** | Bra (via DevTools) | Bra | Inbyggd GUI för felsökning | Varierar beroende på språk |
| **Lämplig för E2E-testning** | Ja, men kräver mer manuell hantering | Ja | Ja, enkel att konfigurera | Ja, men kan vara komplext att sätta upp |
| **Community och support** | Mindre | Växande, starkt stöd av Microsoft | Stor och aktiv | Mycket stor och etablerad |

Vid valet av testverktyg är det avgörande att väga styrkor och svagheter mot projektets specifika behov. Puppeteer är ett Node-bibliotek utvecklat av Google som erbjuder en höggradig kontroll över Chromium-baserade webbläsare via DevTools-protokollet. Det lämpar sig särskilt väl för automatiserade UI-tester i moderna JavaScript-applikationer, såsom single-page applications (SPA).

# Fördelar

1. Direkt interaktion med DOM

Puppeteer tillåter programmeraren att interagera direkt med DOM-element, vilket möjliggör precisa tester och detaljkontroll över varje steg i användarflödet.

2. Stabil och snabb exekvering

Eftersom Puppeteer kommunicerar direkt med webbläsarens DevTools-protokoll, blir testkörningen både stabil och snabb. Detta minimerar latens jämfört med verktyg som använder externa webdrivrar.

3. Stöd för headless- och GUI-läge

Möjligheten att köra tester både headless och med synlig webbläsare (GUI) underlättar felsökning och förbättrar utvecklingsupplevelsen.

4. Enkel att integrera med moderna testverktyg

Puppeteer fungerar väl tillsammans med moderna JavaScript-ramverk och testmiljöer såsom Jest och TypeScript, vilket gör det smidigt att bygga upp en robust testsvit.

5. Flexibelt och skriptvänligt API

API:et är intuitivt och lågnivå, vilket ger stor flexibilitet vid implementation av icke-traditionella användarscenarier.

# Nackdelar

1. Begränsat webbläsarstöd

Puppeteer är huvudsakligen utformat för Chromium-baserade webbläsare. Även om visst stöd finns för Firefox, är det inte lika stabilt eller komplett som i exempelvis Playwright.

2. Brist på inbyggd parallellisering och cross-browser-testing

Funktioner såsom parallell testkörning och stöd för flera samtidiga användarsessioner kräver extern konfiguration eller tillägg, vilket kan begränsa skalbarheten i större projekt.

3. Manuell hantering av väntetider och synkronisering

Till skillnad från Playwright och Cypress, som automatiskt väntar på DOM-förändringar och nätverksstabilitet, kräver Puppeteer ofta manuell hantering. Det sker med metoder som **waitForSelector** eller genom egendefinierade timeout-lösningar som **await new Promise(resolve => setTimeout(resolve, ms))**. Detta ger flexibilitet men ställer också högre krav på noggrann synkronisering.

# Projektsammanhang

Det valda testverktyget, Puppeteer, har använts för att testa en inloggningsfunktion inom ett större projekt – ett boklånehanteringssystem som utvecklades under kursen Java II (D0024E) vid Luleå tekniska universitet. Systemet är en webbapplikation byggd med ett fullstack-upplägg, där backend är implementerad i Spring Boot (Java) och frontend i React med TypeScript. Systemet erbjuder funktioner som boklån, reservationer, hantering av användarroller samt autentisering via JWT (JSON Web Tokens).

I denna hemtentamen har fokus legat på att testa inloggningsflödet i applikationen. Syftet med testningen har varit att säkerställa att rätt meddelanden visas vid både lyckad och misslyckad inloggning, samt att valideringsfel hanteras korrekt när formulärfält lämnas tomma eller innehåller ogiltig data. Dessa är avgörande aspekter av en säker och användarvänlig inloggningsprocess.

Testmiljön har satts upp i ett separat Node.js-baserat projekt där Puppeteer användes för att automatisera interaktioner med webbgränssnittet. För att möjliggöra en strukturerad testkörning användes Jest som testlöpare, tillsammans med TypeScript för typkontroll och bättre utvecklarupplevelse. Projektets beroenden anges nedan:

Centrala beroenden:

- **puppeteer** (^24.9.0) – Automatiserar och kontrollerar en Chromium-instans via DevTools-protokollet.
- **jest** (^29.7.0) – Ett testramverk för JavaScript/TypeScript som hanterar testfall och assertions.
- **ts-jest** (^29.3.4) – Möjliggör användning av TypeScript tillsammans med Jest.
- **typescript** (^5.8.3) – Används för att skriva typade testskript.
- **@types/puppeteer**, **@types/jest**, **@types/node** – Ger typdefinitioner för utveckling i TypeScript.

Testfallen är skrivna för att efterlikna verkliga användarflöden, inklusive ifyllning av formulär, klick på inloggningsknapp, samt verifiering av respons i form av navigering och toast-meddelanden. Den testade inloggningsvyn är tillgänglig under **`/auth`** route i frontend-applikationen.

Genom att använda Puppeteer i detta sammanhang har jag fått möjlighet att tillämpa ett externt testverktyg på en verklig applikation, vilket både stärker förståelsen för testautomatisering och ger erfarenhet av integrering av tredjepartsverktyg i mjukvaruutvecklingsprojekt.

# Testplan

## Syfte och Mål

Syftet med denna testplan är att verifiera och säkerställa att inloggningsflödet i det webbaserade boklånesystem som utvecklats inom ramen för kursen Java II (D0024E) fungerar enligt kravspecifikation. Fokus ligger på att validera att användare kan logga in med giltiga inloggningsuppgifter, att systemet hanterar ogiltiga försök korrekt genom visning av felmeddelanden, samt att formulärvalideringen fungerar för tomma eller felaktigt ifyllda fält. Testningen sker automatiserat med hjälp av verktyget Puppeteer.

## Omfattning och Avgränsning

Testningen omfattar uteslutande funktionaliteten i inloggningsvyn. Funktioner såsom boklån, bokreservationer eller administrativa verktyg ingår inte i denna testomgång. De scenarier som testas inkluderar inloggning med korrekta uppgifter, inloggning med felaktig e-postadress respektive lösenord, försök till inloggning med tomma fält samt validering av inmatningar med ogiltigt format, exempelvis felaktig e-poststruktur eller för kort lösenord. Dessutom verifieras att rätt toast-meddelanden visas och att navigeringen sker korrekt vid lyckad inloggning.

## Testtyp och nivå

De genomförda testerna är funktionella tester på systemnivå, där hela användarflödet från inmatning till gränssnittets respons testas via simulerad användarinteraktion.

## Teststrategi

Testerna genomförs automatiserat i en Node.js-miljö med Puppeteer och Jest. Puppeteer används för att programmässigt interagera med webbläsarens DOM (Document Object Model), exempelvis för att skriva i fält eller klicka på knappar, medan Jest tillhandahåller struktur och assertioner för testutförandet. Varje testfall exekveras i en isolerad webbläsarsession för att förhindra påverkan mellan testerna. Resultaten loggas automatiskt i terminalen och kan vid behov kompletteras med felsökningsinformation.

## Resurser och Tidsåtgång

Testfallen har utvecklats av en student inom ramen för kursen "Test av IT-system". Utvecklingen av testmiljön, samt skapandet av testfall och struktur, har genomförts under cirka 1–1,5 timmar. Testverktyg som använts inkluderar:

- Puppeteer (^24.9.0) – för webbläsarautomatisering.
- Jest (^29.7.0) – som testkörningsmiljö.

- TypeScript (^5.8.3) – för typad och strukturerad utveckling.
- ts-jest och typer för Node.js och Puppeteer har använts som utvecklingsberoenden.

# Leverabler

Resultatet av detta arbete består av en komplett testplan som beskriver mål, strategi, omfattning, metodik och miljö. Till detta hör åtta automatiserade testfall som validerar inloggningsflödet under olika förutsättningar, tillsammans med tillhörande testkod implementerad i TypeScript med Jest och Puppeteer. Slutligen ingår en reflekterande analys av valet av Puppeteer som testverktyg i jämförelse med alternativa ramverk..

# Testfall

**Tabell 2**. Innehåller åtta strukturerade testfall som validerar funktionaliteten i systemets inloggningsformulär. Varje rad specificerar test-ID, kort beskrivning, steg att genomföra, parametervärden, förväntat samt faktiskt resultat och eventuella kommentarer. Testfallen omfattar både lyckade och felaktiga inloggningsförsök och möjliggör systematisk testning av inloggningsflödet.

| Testfall ID | Syfte | Teststeg | Testdata | Förväntat resultat | Faktiskt resultat | Status |
|---|---|---|---|---|---|---|
| TF1. | DOM-verifiering efter lyckad inloggning | 1. Navigera till **`/auth`**<br>2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | E-post: testuser@example.com<br>Lösenord: testpassword123 | Elementet "[data-testid='navbar-dashboard']" finns i DOM | Elementet "[data-testid='navbar-dashboard']" finns i DOM | Pass |
| TF2. | Lyckad inloggning med korrekta uppgifter | 1. Navigera till **`/auth`**<br>2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | E-post: testuser@example.com<br>Lösenord: testpassword123 | Navigering till Home, toast-meddelandet: 'Login successful!' visas på skärmen | Navigering till Home, toast-meddelandet: 'Login successful!' visas | Pass |
| TF3 | Fel e-post, korrekt lösenord | 1. Navigera till **`/auth`**<br>2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | E-post: nonexistent@example.com<br>Lösenord: testpassword123 | "Login failed" på grund av fel inloggningsuppgifter | Toast: 'Login failed: Invalid email or password' | Pass |
| TF4 | Korrekt e-post, fel lösenord | 1. Navigera till **`/auth`**<br>2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | E-post: testuser@example.com<br>Lösenord: wrongpassword | "Login failed" på grund av fel inloggningsuppgifter | Toast: 'Login failed: Invalid email or password' | Pass |
| TF | Tomt e-postfält | 1. Navigera till **`/auth`**<br>2. Fyll i lösenord<br>3. Klicka på “Login”-knappen | E-post: [tomt]<br>Lösenord: testpassword123 | Felmeddelande: 'Required' under e-postfält | Felmeddelande: 'Required' under e-postfält | Pass |
| TF6 | Tomt lösenordsfält | 1. Navigera till **`/auth`**<br>2. Fyll i e-post<br>4. Klicka på “Login”-knappen | E-post: testuser@example.com<br>Lösenord: [tomt] | Felmeddelande: 'Required' under lösenordsfält | Felmeddelande: 'Required' under lösenordsfält | Pass |
| TF7 | För kort | 1. Navigera till **`/auth`** | E-post: | Felmeddelande: 'Too | Felmeddelande | Pass |

| | | | | | | |
|---|---|---|---|---|---|---|
| | lösenord | 2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | testuser@example.com<br>Lösenord: 123 | short!' under lösenordsfält | : 'Too short!' under lösenordsfält | |
| TF8 | Ogiltigt e-postformat | 1. Navigera till `/auth`<br>2. Fyll i e-post<br>3. Fyll i lösenord<br>4. Klicka på “Login”-knappen | E-post: Invalid-email-address<br>Lösenord: validpassword | Felmeddelande: 'Invalid email' under e-postfält | Felmeddelande: 'Invalid email' under e-postfält | Pass |

# Projektkonfiguration och körning av tester

För att exekvera testsviten som utvecklats med Puppeteer och Jest krävs en korrekt konfigurerad utvecklingsmiljö. Nedanstående steg beskriver processen för att installera och köra testerna:

## Systemkrav och beroenden

Förutsatt att Node.js och npm är installerade på systemet, inleds konfigurationen med att projektet klonas från GitHub. Därefter installeras beroenden definierade i package.json:

```
git clone https://github.com/dev-alvandi/test-av-it-home-exam-puppeteer.git
cd test-av-it-home-exam-puppeteer
npm install
```

## Körning av tester

Efter installationen kan testsviten köras med kommandot:

```
npm run test
```

Tester körs i en kontrollerad instans av webbläsaren via Puppeteer och utvärderas med Jest. Resultatet återges direkt i terminalen, inklusive information om lyckade och misslyckade testfall.

## Begränsningar och miljöberoende

Det är viktigt att betona att testfallen i projektet är beroende av att systemet är tillgängligt lokalt via localhost. Backend- och frontenddelarna är inte uppladdade eller tillgängliga via nätet. Därför kommer samtliga testfall att misslyckas om de körs i en annan miljö än den lokala utvecklingsmiljön. Testerna förutsätter att systemet är aktivt och körs på utvecklarens maskin.

# Utvärdering av Puppeteer i praktiken

Testsviten består av åtta olika testfall som täcker både lyckade och misslyckade inloggningsförsök, valideringsfel samt DOM-verifiering. Puppeteer användes i kombination med Jest för att strukturera testfallen, hantera testlivscykeln (**beforeAll**, **beforeEach**, **afterEach**, **afterAll**) och formulera assertion-logik via **expect()** metoden.

En viktig erfarenhet under implementationen var att delad sidinstans (**page**) mellan flera testfall ledde till problem, trots att användaren loggades ut efter varje test. Detta berodde på tillståndsläckage mellan testerna, något som kunde lösas genom att skapa en ny instans av **page** i varje **beforeEach** och stänga den i **afterEach**. Denna förändring förbättrade testens stabilitet avsevärt och återspeglar vikten av att isolera tester i webbläsarbaserad automatisering.

Puppeteer visade sig vara särskilt effektivt för att:

- Identifiera och interagera med specifika DOM-element via **data-testid**.
- Vänta på att element skulle visas genom **waitForSelector**, vilket minskade behovet av statiska **setTimeout**.
- Utföra assertions mot visuella indikatorer som toast-meddelanden, vilket är centralt för att verifiera användarfeedback vid både framgång och fel.

Dock fanns vissa utmaningar, särskilt vid testning av element som renderades dynamiskt, såsom toast-meddelanden. Även om dessa visades korrekt i enskilda tester, uppstod initialt problem när testsviten kördes i sin helhet. Dessa problem kunde härledas till just delat tillstånd mellan testerna – något som visar på Puppeteers känslighet för asynkronitet och DOM-hantering i komplexa flöden.

Sammanfattningsvis har Puppeteer visat sig vara ett effektivt verktyg för automatiserad UI-testning i ett realistiskt scenario. I kombination med Jest och TypeScript har det varit möjligt att bygga en strukturerad, reproducerbar och fullt automatiserad testmiljö med tydliga och verifierbara resultat.

# Slutsats

Denna hemtentamen har gett möjlighet att tillämpa ett externt testverktyg utanför kursens ordinarie innehåll, med målet att fördjupa förståelsen för automatiserad testning inom moderna webbapplikationer. Genom att använda Puppeteer har testningen kunnat genomföras på ett sätt som simulerar faktiska användarinteraktioner i webbläsaren, vilket skapat en mer realistisk testmiljö jämfört med enklare enhetstestning.

Testerna fokuserade på inloggningsflödet i ett boklånesystem utvecklat inom ramen för Java II kursen (D0024E), med särskild betoning på både funktionella och validerande scenarier. Genom att kombinera Puppeteer med Jest och TypeScript kunde en testmiljö etableras som inte bara var kraftfull och återanvändbar, utan också tillräckligt flexibel för att hantera dynamiska gränssnittselement såsom toast-meddelanden och DOM-verifiering.

Under arbetets gång identifierades och hanterades praktiska utmaningar, bland annat relaterade till delade tillstånd mellan testfall, något som löses genom isolering av varje test i en egen webbläsarsession. Detta visar på vikten av testarkitektur och noggrann hantering av testmiljön vid automatiserad E2E-testning.

Puppeteer visade sig vara ett lämpligt verktyg för ändamålet och kan med fördel användas i framtida testprojekt, särskilt när det finns behov av att automatisera verkliga användarflöden i webbläsaren. Valet av verktyg bör dock alltid göras utifrån systemets komplexitet, krav på webbläsarstöd samt testteamets kompetens.

Sammanfattningsvis har denna uppgift bidragit till att bredda den praktiska kunskapen kring testautomatisering, verktygsval och implementation av strukturerade teststrategier i moderna webbsystem.

# Referenser


BrowserStack. (2023). Cypress vs Selenium vs Playwright vs Puppeteer: Core Differences. BrowserStack Guide. Retrieved May 30, 2025, from https://www.browserstack.com/guide/cypress-vs-selenium-vs-playwright-vs-puppeteer

Cypress.io. (2025). Cypress Documentation. Retrieved May 30, 2025, from https://docs.cypress.io/

García, B., del Alamo, J. M., Leotta, M., & Ricca, F. (2024). Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. In A. Bertolino, J. Pascoal Faria, P. Lago, & L. Semini (Eds.), Quality of Information and Communications Technology (pp. 142–149). Springer. https://doi.org/10.1007/978-3-031-70245-7_10

LambdaTest. (2024). Selenium Vs Cypress Vs Playwright: Key Differences at a Glance. LambdaTest Community. Retrieved May 30, 2025, from https://community.lambdatest.com/t/selenium-vs-cypress-vs-playwright-key-differences-at-a-glance-lambdatest/32040

Playwright Team. (2025). Playwright Documentation. Retrieved May 30, 2025, from https://playwright.dev/

Puppeteer Team. (2025). Puppeteer Documentation. Retrieved May 30, 2025, from https://pptr.dev/

Selenium Project. (2025). Selenium Documentation. Retrieved May 30, 2025, from https://www.selenium.dev/documentation/

Testim.io. (2020). Which Is Best: Puppeteer, Selenium, Playwright, Cypress? Testim Blog. Retrieved May 30, 2025, from https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-how-to-choose/