Şteică Mălina-Alexa

Facultatea de Matematica şi Informatică

Design Patterns

Modelele de design joacă un rol esențial în dezvoltarea software, deoarece oferă soluții testate pentru problemele de design frecvente. În cadrul codului SortMicroserviceExample furnizat, putem identifica mai multe modele de design utilizate în mod eficient. Această lucrare va explora trei dintre aceste modele: Singleton, Factory și Asynchronous Programming. Vom prezenta exemple concrete din cod pentru a ilustra implementarea și beneficiile acestor modele.Singleton Design Pattern: Modelul de design Singleton se asigură că este creată doar o singură instanță a unei clase și furnizează un punct global de acces la acea instanță. În codul SortMicroserviceExample, clasa Program utilizează modelul Singleton. Deoarece metodele test_asc și test_desc sunt definite ca statice, acestea pot fi accesate fără a crea o instanță a clasei Program. Acest lucru garantează că este creată și utilizată doar o singură instanță a clasei Program pe întreaga durată de execuție. Modelul Singleton contribuie la menținerea consistenței și furnizează un punct centralizat de acces pentru funcționalitatea înrudita.

Modelul Singleton este evident în metoda Main, unde metodele test_asc și test_desc sunt apelate direct fără a instantia clasa Program. Acest lucru asigură că există doar o singură instanță a clasei Program care este partajată între metode.

Factory Design Pattern: Modelul de design Factory furnizează o interfață pentru crearea de obiecte, dar permite subclaselor să decidă ce clasă să instantieze. În codul SortMicroserviceExample, clasa HttpClient reprezintă un exemplu al modelului Factory. Clasa HttpClient oferă o metodă factory numită SendAsync, care creează și returnează o instanță a clasei HttpClient pentru a trimite cereri HTTP asincron. Modelul Factory permite flexibilitate în crearea diferitelor instanțe ale claselor înrudite în funcție de condiții sau configurații specifice.

Utilizarea clasei HttpClient demonstrează modelul Factory. Codul creează o instanță a clasei HttpClient utilizând cuvântul cheie new, iar apoi utilizează metoda factory SendAsync pentru a crea și returna o instanță a clasei HttpRequestMessage. Acest lucru permite codului să trimită cereri HTTP asincrone utilizând instanța de HttpClient creată.

Asynchronous Programming Pattern: Modelul de programare asincronă permite executarea operațiilor non-blocante, îmbunătățind performanța și reactivitatea. În codul SortMicroserviceExample, programarea asincronă este utilizată în mod intens prin utilizarea cuvintelor cheie async și await. Aceste cuvinte cheie permit codului să efectueze operații asincrone fără a bloca firul de execuție.

Metodele test_asc, test_desc și Main sunt marcate cu cuvântul cheie async, indicând faptul că acestea conțin operații asincrone. Cuvântul cheie await este utilizat pentru a aștepta finalizarea operațiilor asincrone, cum ar fi httpClient.SendAsync și response.Content.ReadAsStringAsync. Prin utilizarea modelului de programare asincronă, codul realizează executarea concurentă a sarcinilor, îmbunătățind eficiența și reactivitatea generală.

Codul SortMicroserviceExample integrează în mod eficient modelele de design Singleton, Factory și Asynchronous Programming. Modelul Singleton asigură o singură instanță a clasei Program, promovând consistența și accesul centralizat. Modelul Factory este ilustrat de clasa HttpClient, permițând crearea dinamică a obiectelor înrudite. În cele din urmă, modelul de programare asincronă îmbunătățește performanța și reactivitatea prin permiterea operațiilor non-blocante. Aceste modele de design contribuie la modularitatea, ușurința de întreținere și execuția eficientă a codului, evidențiind importanța lor în dezvoltarea software.

Avantajele și dezavantajele utilizării serviciilor sau microserviciilor

Arhitectura bazată pe microservicii a câștigat o popularitate tot mai mare în dezvoltarea software în ultimele decenii. Această abordare implică dezmembrarea aplicațiilor complexe în componente mai mici și mai puțin dependente între ele, care pot fi dezvoltate, implementate și întreținute independent.

Adoptarea microserviciilor în dezvoltarea software aduce numeroase avantaje:

- Rezistență și izolare a defectelor: Arhitectura microserviciilor adaugă o rezistență sporită la defecte. Deoarece fiecare serviciu este independent și are interfețe bine definite cu celelalte servicii, în cazul în care un serviciu întâmpină probleme sau eșuează, celelalte servicii pot continua să funcționeze normal. Nu există dependențe stricte între servicii, ceea ce permite o izolare mai bună a defectelor și reduce impactul acestora asupra întregului sistem.
- Scalabilitate și performanță optimizate: O caracteristică cheie a arhitecturii microserviciilor este scalabilitatea orizontală. Aceasta înseamnă că fiecare componentă poate fi scalată individual în funcție de necesități. Astfel, resursele pot fi utilizate în mod eficient, iar serviciile specifice pot fi scalate în mod independent, fără a necesita scalarea întregii aplicații. Această flexibilitate permite obținerea unor performanțe ridicate și gestionarea eficientă a creșterii volumului de trafic sau a cerințelor sporite.
- Flexibilitate și agilitate în dezvoltare: Arhitectura microserviciilor încurajează flexibilitatea și agilitatea în dezvoltarea software. Componentele individuale pot fi dezvoltate și implementate independent, fără a afecta celelalte servicii. Acest lucru facilitează ciclurile de lansare mai rapide, integrarea și implementarea continuă și adoptarea ușoară a noilor tehnologii. Dezvoltatorii pot lucra pe module separate, ceea ce duce la o mai mare eficiență și productivitate în procesul de dezvoltare.

Cu toate acestea, există și câteva dezavantaje asociate cu utilizarea microserviciilor:

- Complexitatea managementului: Utilizarea microserviciilor aduce cu sine o creștere a complexității managementului. Fiind componente independente, acestea necesită coordonare și monitorizare adecvată. Este esențial să dispunem de o infrastructură și de instrumente adecvate pentru a gestiona eficient această complexitate.

- Necesitatea testării și monitorizării avansate: Datorită naturii distribuite a microserviciilor, este necesară o testare și monitorizare avansate pentru a asigura funcționarea corectă și performanța sistemului. Aceasta poate implica eforturi suplimentare și cerințe mai mari de resurse.

- Comunicare și latență: Comunicarea între microservicii poate introduce întârzieri și poate crește complexitatea sistemului. O abordare inadecvată sau ineficientă a comunicării poate afecta performanța generală a sistemului.

În concluzie, arhitectura bazată pe microservicii reprezintă o abordare tot mai populară în dezvoltarea de software, oferind numeroase avantaje precum scalabilitatea, flexibilitatea și rezistența la defecte. Prin împărțirea aplicațiilor complexe în componente mai mici și mai puțin dependente între ele, microserviciile permit dezvoltarea, implementarea și întreținerea independentă a acestora. Cu toate acestea, există și dezavantaje asociate, cum ar fi complexitatea managementului și cerințele sporite de testare și monitorizare. Evaluarea atentă a nevoilor și contextului specific al proiectului este crucială în luarea unei decizii informate cu privire la adoptarea arhitecturii microserviciilor. Prin înțelegerea profundă a avantajelor și dezavantajelor, dezvoltatorii pot profita la maximum de potențialul acestei abordări arhitecturale și pot crea sisteme software eficiente și flexibile.