**Funcționalități**

Aplicația pe care o voi dezvolta este un sistem de bug tracking menit să țină evidența bug-urilor ce pot apărea în timpul dezvoltării unei aplicații software. Aplicația va oferi caracteristicile de bază pe care le deține orice bug tracker însă va încerca să ofere noi funcționalități care vor ușura munca dezvoltatorului.

Utilizatorii vor avea posibilitatea de a se înregistra pentru a putea folosi aplicația, putându-se mai apoi autentifica folosind numele de utilizator și parola alese. De asemenea, manager-ul proiectului va putea adăuga noi membrii la echipa sa. Pentru aceștia se va crea un cont de utilizator cu o parolă generată automat. Această parolă va fi trimisă prin e-mail noilor utilizatori la adresa specificată la adăugare, ei având ulterior posibilitatea de a o modifica.

Aplicația va putea urmări mai multe proiecte la care lucrează un dezvoltator, oferindu-i acestuia posibilitatea de a administra bug-urile apărute în cadrul tuturor proiectelor sale. Acesta va putea adăuga fie bug-uri, fie noi funcționalități la care trebuie să se lucreze în viitor, permițând astfel să se facă o distincție clară între acestea. Toate acestea vor putea fi organizate în milestone-uri (etape importante din ciclul de viață al proiectelor care trebuie îndeplinite până la o anumită dată prestabilită), care vor fi adăugate de către manager-ul de proiect.

În formularul care va trebui completat la adăugarea unui nou task, se va putea specifica, pe lângă un titlu sugestiv, o descriere amănunțită, și va exista posibilitatea de a adăuga fișiere sau fotografii auxiliare. De asemenea, acum se va putea selecta gradul de severitate pe care îl are respectiva problemă (low, medium, high, critical). Aceasta va putea fi atribuită unui anumit milestone, specificându-se astfel data la care va trebui să fie rezolvată. Va exista posibilitatea de a alege un responsabil pentru rezolvarea bug-ului, însă acest lucru nu va fi obligatoriu. Pentru o grupare a task-urilor în funcție de anumite categorii relevante pentru echipa de dezvoltatori, se vor putea adăuga diferite etichete ce constau din cuvinte cheie sugestive.

O altă modalitate de a adăuga bug-uri va fi prin trimiterea de e-mail-uri de către utilizatori, fie aceștia dezvoltatori sau clienți. Astfel, subiectul e-mail-ului va reprezenta titlul noului task, iar conținutul va fi descrierea adiacentă. Prin această funcționalitate se va putea primi feedback din partea clienților mult mai rapid și mai eficient.

O dată ce un bug este asignat unui dezvoltator, acesta va trece din starea “open”, în starea “in progress”, sugerând astfel că s-a început rezolvarea lui. La finalul dezvoltării, acesta va trece în starea “in testing” și abia după ce testarea s-a finalizat cu succes bug-ul va putea fi considerat “closed”.

Pentru a putea fi la curent cu toate modificările aduse bug-urilor, dezvoltatorii implicați vor primi notificări privind aceste schimbări. Cei care vor primi notificări vor fi persoanele care au creat respectivul bug, sunt asignate lui sau au scris comentarii în legătură cu acesta. Următoarele schimbări vor declanșa trimiterea de notificări: noi comentarii sau fișiere atașate task-urilor, un nou task asignat dezvoltatorului în cauză, modificarea stării unui task sau închiderea unui task în care dezvoltatorul este implicat.

Dezvoltatorii vor avea posibilitatea de a adăuga notițe pentru fiecare din task-urile proprii, putând astfel să consemneze idei pentru studii ulterioare. De asemenea, se va pune la dispoziție accesul la un motor de căutare (Bing) pentru a căuta mai facil răspunsuri la întrebări tehnice sau de altă natură. Acestea vor putea fi salvate în detaliile adiacente fiecărui task, nefiind nevoie de o recăutare următoare sau de adăugarea unui bookmark în browser-ul personal.

În pagina de listare a task-urilor, se vor putea aplica diferite filtre pentru a ușura căutarea în funcție de o categorie. Vor exista filtre predefinite, însă dezvoltatorul va avea de asemenea posibilitatea de a-și crea propriile filtre personalizate. Task-urile vor putea fi afișate în funcție de milestone-ul la care sunt asignate, etichetele pe care le dețin, categoria din care fac parte, gradul de severitate, starea etc.

Aplicația va oferi de asemenea posibilitatea de a genera rapoarte ce evidențiază volumul de muncă parcurs sau viitor. Se va putea vizualiza și lista de activități care au fost întreprinse în fiecare zi, iar manager-ul va putea primi prin e-mail un sumar al activităților la o perioadă de timp selectată de el: zilnic, săptămânal sau lunar.

**Platforma de dezvoltare**

Aplicația de față este o aplicație web-based, fiind construită folosind framework-ul ASP.NET MVC 5. Am ales să folosesc ASP.NET deoarece acesta este susținut de puternica platformă .NET care oferă deosebit de multe facilități. Server-ul ce va găzdui aplicația va fi Internet Information Services (IIS).

MVC este un model arhitectural compus din trei componente interconectate: Model, View, Controller. Model-ul conține reprezentarea informației cu care utiizatorii interacționează. Aceasta poate fi constituită de view model, ce reprezintă datele transmise între view și controller, sau de domain model, care conține datele de bază ale aplicației, precum și operațiile de manipulare ale acestora. View-ul ajută la redarea informației într-o interfață grafică, iar controller-ul, care procesează cererile primite, efectuează operațiile necesare pe model și selectează view-ul care trebuie redat utilizatorului.

Principalul motiv pentru care am ales să folosesc modelul arhitectural MVC este faptul că acesta respectă principiul separării intereselor (separation of concerns). Fiecare componentă este bine definită și independentă: logica de manipulare a datelor este conținută numai în model, logica de afișare a datelor numai în view, iar codul care se ocupă cu cererile utilizatorului este conținut doar în controller. Astfel, aplicația devine mult mai ușor de menținut și extins, oferind și posibilitatea de a scrie cod curat și organizat. De asemenea, MVC înlesnește realizarea de unit teste și îmbunătățește search engine optimization. Modelul arhitectural MVC oferă posibilitatea dezvoltatorului să aleagă șabloanele URL pe care link-urile aplicației sale să le respecte, făcându-le astfel mult mai ușor de căutat și înțeles.

Pentru a asigura persistența datelor, am ales să folosesc Microsoft SQL Server 2012. În ceea ce privește manipularea datelor din baza de date, platforma .NET pune la dispoziție ca instrument de Object Relațional Mapping (ORM), Entity Framework. Versiunea folosită este Entity Framework 6, iar modalitatea de creare a tabelelor și de manipulare a datelor este Code First. Pentru a utliza Entity Framework Code First, este necesar doar să creezi clasele ce reprezintă entitățile de bază ale aplicației, creându-se mai apoi automat tabelele bazei de date. Pentru a manipula datele, se folosesc instanțe ale claselor în cauză.

În ceea ce privește interfața cu utilizatorul, pe lângă HTML, CSS, Javascript și jQuery, voi folosi și Ajax și Bootstrap, care adaugă un plus de interactivitate și atractivitate aplicației.

**Șabloanele repository și unit of work**

Șabloanele repository și unit of work sunt șabloane des întâlnite în dezvoltarea de aplicații, ele creând un nivel de abstractizare între nivelul de acces la date și nivelul de logică de business al aplicației. Implementând aceste șabloane, mentenabilitatea aplicației sporește considerabil, aceasta putând fi izolată de modificările realizate în depozitul de date, refolosirea codului este sporită și testarea unitară (unit testing) este facilitată, precum și dezvoltarea pornind de la teste (test-driven development).

Următoarea imagine prezintă relațiile dintre clasele controller și cele de context în cazul în care nu sunt folosite șabloanele repository și unit of work și în cazul în care acestea sunt folosite. Se poate observa cum acestea se plasează între clasele controller și baza de date. Componenta unit of work înglobează depozitele (repositories) existente în aplicație, precum și contextul bazei de date.

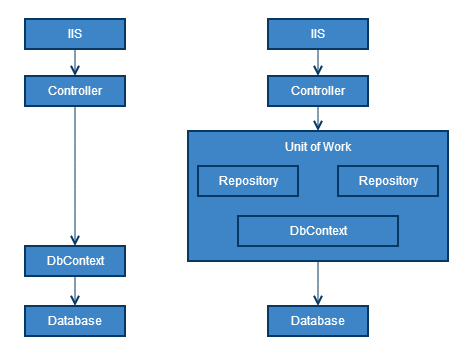


Figura X.y: Relațiile dintre clasele Controller și DbContext

Șablonul repository este creat pentru a administra persistența datelor de la și către Entity Framework. Există două abordări de implementare a acestui șablon: strict pentru a asigura persistența datelor sau pentru asigurarea persistenței datelor și, de asemenea, a funcțiilor necesare logicii de business.

În aplicația de față, am ales să folosesc cea de-a doua abordare. În acest sens, am realizat o clasă generică care oferă toate operațiile CRUD (Create, Read, Update, Delete) necesare manipulării datelor. Introdu aici code snippet number

În implementarea clasei GenericRepository am ales să folosesc sarcinile (task-urile) asincrone întrucât aceste apeluri non-blocante către nivelul bazei de date îmbunătățesc scalabilitatea aplicației. Efectuarea unui task în mod asincron duce la eliberarea firului de execuție principal care poate efectua alte operații sau poate accepta alte cereri cât timp datele necesare sunt aduse din baza de date.

În ceea ce privește șablonul unit of work, scopul său este de a asigura folosirea unui singur context de baze de date atunci când există mai multe depozite în aplicație. Astfel, la finalizarea mai multor operații, se poate apela o singură dată metoda care va efectua toate modificările ce trebuie comise. Folosind șablonul unit of work este asigurată integritatea datelor. Introdu aici code snippet number

Clasa UnitOfWork conține o metodă de salvare a modificărilor necesare și câte o proprietate pentru fiecare depozit în parte. De asemenea, clasa implementează interfața IDisposable. Acest lucru este necesar pentru fiecare clasă care instanțiază o resursă (în cazul de față, contextul bazei de date) pentru a o elibera în momentul în care aceasta nu mai este folosită.

În Introdu aici code snippet number se poate observa modul în care aceste șabloane sunt folosite în clasele de tip controller.

**Dependency injection**

În proiectarea unei aplicații orientate-obiect, un principiu important de design este cuplajul slab (loose coupling). Acesta presupune ca obiectele să nu aibă mai multe dependențe decât este necesar, iar numărul acestor dependențe să fie cât mai redus. Mai mult decât atât, obiectele ar trebui să fie dependente de interfețe și nu de obiecte concrete. Cu ajutorul cuplajului slab, reutilizarea codului este crescută, iar întreținerea aplicației devine mai facilă.

Una din modalitățile de a aplica loose coupling într-o aplicație este dependency injection (injecția de dependențe). Injecția unei dependențe (serviciu) presupune transmiterea acesteia către obiectul care depinde de ea (clientul). Aceasta este cerința fundamentală a modelului de design software, clientului nefiindu-i permis să construiască sau să găsească singur serviciul de care are nevoie. Astfel, crearea dependențelor clientului este separată de comportamentul acestuia și se respectă principiul cuplajului slab.

Dependency injection este un șablon de design care implementează inversion of control (inversarea controlului) și permite aplicației să respecte principiul dependency injection. Acest principiu presupune ca modulele de nivel înalt să nu depindă de module de nivel scăzut și ambele să depindă de entități abstracte. De asemenea, aceste entități nu ar trebui să depindă de detalii, acestea din urmă depinzând la rândul lor de entități abstracte. Cu alte cuvinte, modulele de nivel înalt dintr-o aplicație nu ar trebui să depindă de implementări concrete ale modulelor de nivel scăzut, ci doar de abstractizări. Inversion of Control reprezintă o implementare a acestui principiu.

Inversion of Control descrie un design în care porțiuni personalizate ale unei aplicații primesc fluxul de comandă de la o bibliotecă generică, reutilizabilă. O arhitectură software care respectă un astfel de design inversează controlul în comparație cu programarea procedurală tradițională: în programarea tradițională, codul personalizat care exprimă scopul programului cheamă biblioteci reutilizabile pentru a rezolva sarcini generice, dar, cu inversion of control, codul reutilizabil este cel care cheamă codul personalizat. Dependency Injection reprezintă o versiune specializată a șablonului Inversion of Control. Imaginea următoare reliefează legătura dintre principiul Dependency Injection, șablonul Inversion of Control și Dependency Injection.

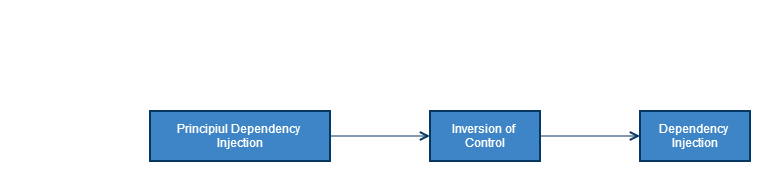


Figura x.y: Legătura dintre principiul Dependency Injection, șablonul Inversion of Control și Dependency Injection

Dependency Injection presupune existența a patru elemente: un obiect ce reprezintă serviciul, clientul care este dependent de serviciul respectiv, interfața pe care clientul o folosește pentru a comunica cu serviciul și container-ul, care este responsabil pentru injectarea serviciului în client.

Schema următoare prezintă modul în care cele patru elemente enunțate mai sus interacționează.

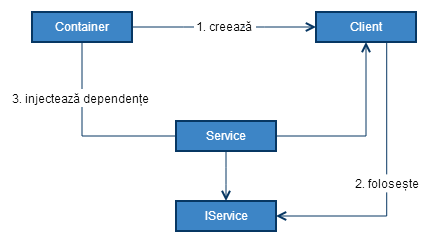


Figura x.y: Modul în care componentele implicate în Dependency Injection interacționează

Container-ul este cel ce creează dependențele și le injectează automat atunci când este necesar, creând automat obiectele cerute. Acesta ajută dezvoltatorul să gestioneze dependențele din aplicație într-un mod simplu și ușor.

Există trei modalități prin care container-ul poate injecta serviciul în client: prin intermediul constructorului (dependențele sunt furnizate cu ajutorul constructorului de clasă), prin intermediul unei proprietăți (clientul expune o proprietate pe care container-ul o folosește pentru injectarea dependenței), prin intermediul unei interfețe (serviciul conține o metodă de injecție pentru a transmite dependența clientului; clienții trebuie să implementeze o interfață care expune o proprietate ce acceptă dependența).

Container-ul pe care l-am ales în dezvoltarea aplicației este Ninject. Acesta este un dependency injector open source create special pentru platforma .NET, acesta fiind și motivul principal pentru care am ales să îl folosesc. În ceea ce privește modalitatea prin care container-ul injectează dependențele în client, am ales prima variantă, și anume cea prin intermediul constructorului. Fiecare controller conține un constructor ce așteaptă o implementare a interfeței IUnitOfWork. Ninject este responsabil cu furnizarea implementării. Introdu aici code snippet number

**MVC (Model-View-Controller)**

Model-View-Controller este un șablon arhitectural folosit cu preponderență pentru a crea interfețe grafice. Modelul se bazează pe modularitate și premisa sa majoră este de a separa trei aspecte diferite ale interfeței grafice: datele (model-ul), reprezentarea vizuală a datelor (view-ul) și interfața dintre view și model (controller-ul). Ideea de bază din spatele păstrării acestor trei componente separate este că fiecare trebuie să fie cât mai independente posibil, iar modificările aduse uneia dintre ele nu trebuie să le afecteze pe celelalte.

Model-ul reprezintă nucleul aplicației. Acesta conține datele și logica de business, fiind responsabil cu manipularea datelor stocate, de cele mai multe ori, într-o bază de date. El conține metode care permit inserarea, ștergerea, actualizarea și preluarea datelor. În funcție de datele cerute de utilizator, model-ul oferă controller-ului o reprezentare a acestora, acestea urmând a fi transmise către view, care le poate reda în orice format se dorește. Model-ul nu aplică nicio formatare asupra datelor. Astfel, un singur model poate transmite date către oricâte view-uri. De asemenea, este redusă cantitatea de cod duplicat întrucât model-ul este scris o singură data și poate fi refolosit de toate view-urile.

View-ul este componenta care se ocupă cu afișarea interfeței grafice, fiind cea pe care utilizatorul o poate vizualiza și cu care poate interacționa. View-ul preia datele din model , date care îi sunt transmise de către controller, și le transpune într-un layout care este prezentat utilizatorului. View-ul nu produce modificări datelor, ci doar le afișează. Toate cererile de a modifica datele sunt generate de controller și realizate de către model.

Controller-ul este componenta care se ocupă de interacțiunea cu utilizatorul, de lucrul cu model-ul și, în cele din urmă, de selectarea view-ului care trebuie să redea interfața grafică. Acesta interpretează cererile de la utilizator și ia măsurile necesare pentru a le îndeplini. El determină ce acțiune a fost solicitată și, bazându-se pe cererea primită, selectează model-ul corespunzător și îi transmite acestuia eventualele modificări necesare, creează un view și asociază unul sau mai multe model-e cu view-ul. Controller-ul nu manipulează datele, ci apelează doar anumite metode din model. De asemenea, el nu afișează datele cerute, ci creează un view care este responsabil cu acest lucru.

În figura de mai jos sunt prezentate componentele șablonului arhitectural MVC și relațiile dintre acestea. Este reliefat faptul că controller-ul este cel ce primește cererile utilizatorului, preia datele din modelul corespunzător și apoi le transmite view-ului. Atât controller-ul, cât și view-ul depind de model, însă modelul nu cunoaște existența model-ului sau a controller-ului.

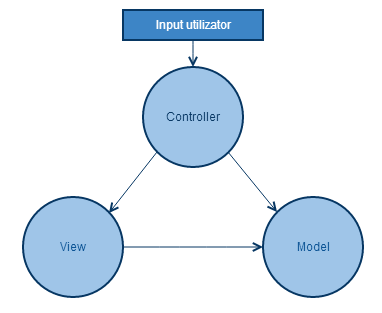


Figura x.y: Componentele șablonului MVC

Pentru dezvoltarea aplicației am ales să folosesc framework-ul ASP.NET MVC. Fiind bazat pe platforma .NET, dezvoltatorul are acces la o mulțime de API-uri atât din platforma .NET, cât și dezvoltate de terțe, ceea ce facilitează dezvoltarea mai facilă a unei aplicații cu numeroase funcționalități.

Modelul MVC este transpus în ASP.NET MVC astfel: model-ul, care conține clasele din domeniul aplicației, încapsulează datele stocate în baza de date, precum și codul necesar pentru manipularea lor. În ASP.NET MVC el reprezintă nivelul de acces la date și utilizează un instrument precum Entity Framework, împreună cu cod particularizat pentru logica de business specifică domeniului. În ceea ce privește view-ul, ASP.NET MVC oferă un view engine pentru a genera dinamic cod HTML. View engine-ul ales este Răzor. Acesta oferă o sintaxă optimizată pentru generarea codului HTML și dorește simplificarea dezvoltării de view-uri. Controller-ele sunt clasele ce administrează relația dintre model și view. În ASP.NET MVC, numele lor conține în mod convențional sufixul „Controller”.

**Entity Framework**

Entity Framework este un instrument de Object Relațional Mapping existent în platforma .NET. Acesta permite dezvoltatorilor să creeze aplicații care au acces la date programând pe un model de aplicație conceptual, în loc de a programa direct pe o schemă de stocare relațională. Astfel, dezvoltatorii pot lucra cu datele sub formă de obiecte și proprietăți specifice domeniului, fără a fi nevoie să se preocupe de tabelele și coloanele aferente din baza de date. Cu ajutorul Entity Framework, dezvoltatorii pot lucra la un nivel de abstractizare mai ridicat atunci când se ocupă de date și pot crea și întreține aplcatii orientate pe date folosind mai puțin cod decât în aplicațiile tradiționale.

Elementul central al Entity Framework îl reprezintă entitatea. O entitate reprezintă datele asociate cu un anumit obiect din domeniul de business al aplicației, din perspectiva aplicației. Cu alte cuvinte, reprezintă acele date care au relevanță pentru aplicația respectivă, chiar dacă obiectul real poate avea mai multe date asociate. Din perspectiva unui administrator de date, entitatea este un singur rând dintr-un view care conține toate informațiile ce au legătură cu acel obiect, indiferent de tabela în care sunt ele păstrate.

În ceea ce privește entitățile, ele pot fi privite din mai multe puncte de vedere. Din punct de vedere fizic, ele reprezintă tabelele, cheile, indecșii, view-urile din baza de date care conțin datele asociate obiectului real. Aceste elemente sunt optimizate pentru a facilita stocarea și manipularea fără erori de către sistemul de gestiune al bazei de date (SGBD). Deși stocarea fizică a datelor este eficientă pentru SGBD, este dificil de înțeles petru dezvoltator. Din punct de vedere logic, entitatea reprezintă datele combinate necesare care definesc un obiect. Din perspectiva bazei de date, aceasta reprezintă view-ul, care combină datele din mai multe tabele prin intermediul cheilor și a altor elemente. Acest punct de vedere logic este destul de abstract pentru dezvoltator, poate duce la erori și la mult cod de gestionat. Din punct de vedere conceptual, entitatea reprezintă perspectiva reală a datelor, așa cum se aplică obiectului. Acest punct de vedere prezintă informații într-un mod ușor de înțeles, ca obiecte, în care accentul se pune pe date și nu pe structura bazei de date.

Nefolosind un ORM, dezvoltatorul trebuie să ia în considerare toate cele trei perspective ale datelor: cea fizică, cea logică și cea conceptuală. Trebuie să cunoască cu precizie care tabelă stochează anumite date, cum respectiva tabelă se asociază cu alte tabele din baza de date și cum să grupeze datele astfel încât să creeze o imagine completă a unei anumite entități. De asemenea, dezvoltatorul este responsabil cu crearea manuală a conexiunii dintre aplicație și baza de date. Entity Framework vine în întâmpinarea devoltatorului și realizează astfel de sarcini automat, acesta din urmă concentrându-se asupra entității, și nu asupra structurii fizice sau logice a bazei de date.

Este posibil să se definească relații între diferite entități prin asocieri. O astfel de asociere este similară cu un join realizat la nivelul bazei de date. Una sau mai multe proprietăți din fiecare entitate definesc relația dintre cele două entități. Multiplicitatea acestor proprietăți de asociere determină dacă aceasta este o conexiune una-la-unu, unu-la-mai-mulți sau mulți-la-mai-mulți. Asocierea este bidirecțională, entitățile având acces reciproc deplin.

Pentru a permite unei entități să vizualizeze datele furnizate de către o altă entitate, acestea au o proprietate de navigare. Această funcționalitate este asemănătoare unei chei externe dintr-o bază de date, dar este mai ușor de folosit și implementat. De asemenea, Entity Framework pune la dispoziție posibilitatea de a folosi lazy loading. Acest lucru presupune ca o entitate sau o colecție de entități să fie automat încărcată din baza de date prima dată când o proprietate ce face referire la acestea este accesată. Astfel, nu sunt aduse date din baza de date decât atunci când este nevoie de ele. Exemplul din Introdu aici code snippet number reliefează modul în care acest lucru este realizat. La adăugarea cuvântului cheie *virtual*, lazy loading este activat, iar respectivele date sunt aduse numai la momentul în care respectiva proprietate este activată. La polul opus se găsește eager loading, care presupune ca toate entitățile care fac parte dintr-o anumită entitate din cadrul unei interogări să fie încărcate o dată cu aceasta. Eager loading este dobândită prin folosirea metodei *Include*. În Introdu aici code snippet number se poate vedea un exemplu de utilizare.

Pentru a oferi posibilitatea de a deține toate informațiile referitoare la entități împreună, Entity Framework pune la dispoziție un container. Înainte de a interacționa cu entitățile, un astfel de container este creat, acesta fiind cunoscut sub numele de context sau DbContext. Acesta facilitează interogarea bazei de date, convertind valorle din baza de date în obiecte entitate și viceversa. El ține evidența modificărilor ce au avut loc în cadrul entităților după realizarea interogărilor din bază de date. De asemenea, menține persistența datelor, efectuând inserări, actualizări sau stergeri în baza de date, ținând cont de starea entităților. Clasa DbContext conține câte un DbSet<T> pentru a defini relația dintre fiecare tabelă și model. De asemenea, în această clasă se regăsesc și opțiunile de configurare ale acestor relații în baza de date. Clasa DbContext poate fi vizualizată în Introdu aici code snippet number.

Entity Framework pune la dispoziție trei modalități de a interacționa cu baza de date: Model First, Database First și Code First. În abordarea Model First, dezvoltatorul definește modelul bazei de date. O dată ce entitățile și relațiile dintre acestea sunt proiectate, baza de date fizică este generată de către Entity Framework. Database First presupune că Entity Framework să genereze clase care reflectă design-ul unei baze de date existente. În ultima abordare, Code First, sunt create inițial clasele care reprezintă tabelele bazei de date și apoi este generată pe baza acestora baza de date.

În mod tradițional, aplicațiile proiectate și dezvoltate sunt orientate pe date (Data Driven Design). Acest lucru înseamnă că dezvoltatorul este nevoit să se gândească la datele necesare pentru a îndeplini cerințele de business și apoi să dezvolte aplicația de jos în sus (bottom up), pornind de la baza de date. Pentru o astfel de abordare se pretează folosirea metodei Database First oferită de Entity Framework.

O alternativă de proiectare și dezvoltare a aplicațiilor software este prin utilizarea abordării orientate pe domeniu (Domain Driven Design). În această abordare dezvoltatorul se gândește în termeni de entități și modele necesare pentru îndeplinirea cerințelor de business. În cazul în care unele dintre aceste modele trebuie să fie persistente, ele pot fi menținute într-o bază de date sau alt mediu de stocare. O astfel de abordare presupune ca modelele proiectate să poată fi stocate oriunde. Astfel, modelele pot ignora modalitatea prin care se va asigura persistența datelor, iar logică de persistență poate fi menținută separat. Entity Framework Code First este recomandată pentru crearea acestor tipuri de aplicație orientate pe domeniu, aceasta fiind și abordarea aleasă de mine în dezvoltarea aplicației.

Entity Framework Code First permite dezvoltatorului să folosească obiecte de tip POCO (Plain Old CLR Objects) pentru definirea modelelor. Acestea sunt obiecte care nu moștenesc nicio altă clasă și nici nu implementează nicio interfață. De asemenea, ele nu cunosc niciun detaliu despre modul în care este asigurată persistența datelor în aplicație și nici despre Entity Framework. Clasa care se ocupă de relația cu Entity Framework este cea care extinde DbContext.

Pentru a putea manipula datele într-o manieră facilă, platforma .NET pune la dispoziție LINQ to Entities. Language-Integrated Query (LINQ) este un set de extensii care cuprinde operații de interogare, setare și transformare integrate în limbajul de programare C#, fiind o punte de legătură între lumea obiectelor și lumea datelor. Acesta extinde limbajul prin adăugarea unor expresii de interogare asemănătoare operațiilor SQL și poate fi folosit pentru a extrage și procesa date din vectori, clase enumerabile, documente XML sau baze de date relaționale.

LINQ to Entities oferă suportul LINQ în scrierea interogărilor pentru Entity Framework. Acestea sunt reprezentate de interogări sub forma unor arbori de comenzi, care sunt executate asupra contextului. LINQ to Entities convertește interogările LINQ în interogări sub formă de arbori de comenzi, le execută asupra Entity Framework și returnează obiecte care pot fi utilizate atât de LINQ, cât și de Entity Framework. LINQ to Entities permite dezvoltatorilor să creeze interogări flexibile, puternic tipate, prin utilizarea expresiilor LINQ și a operatorilor standard de interogare ai acestuia. Un exemplu de utilizare a LINQ to Entities se găsește în Introdu aici code snippet number.

Pentru o mai facilă mentenabilitate a aplicației, am ales să folosesc dependency injection (injectarea dependențelor). Aceasta presupune ca fiecare componentă să nu cunoască detalii despre o alta cu care trebuie să comunice, ci să comunice cu aceasta prin interfețe. Acest lucru este realizat printr-un dependency injection container care asignează clasele necesare fiecărei interfețe specificate. Container-ul ales este Ninject.

