

Lista figurilor

Figura 1: Exemplu sistem de repetiție (Sursă: Wikipedia)	3
Figura 2: Logo-ul aplicației	4
Figura 3: Diagrama funcțională a aplicației	8
Figura 4: Structura vocabularului limbii japoneze	9
Figura 5: Stadiile progresului fiecărui element din vocabular	11
Figura 6: Diagrama Use Case pentru utilizatorul autentificat	15
Figura 7: Relațiile folosite într-o diagrămă ERD	16
Figura 8: Diagrama bazei de date (ERD)	17
Figura 9: Tipurile enumerare folosite în baza de date	19
Figura 10: Structura soluției aplicației	22
Figura 11: Nivelele arhitecturii Onion	22
Figura 12: Schemă Model-View-Controller (Sursă: Uaic, FII, curs Introducere în .NET)	23
Figura 13: Structura proiectelor Data.Domain și Data.Persistance	24
Figura 14: Clasele POCO din cadrul proiectului Data.Domain	25
Figura 15: Clasele Wrappers din cadrul proiectului Data.Domain	27
Figura 16: Interfețele Repository din cadrul proiectului Data.Domain	27
Figura 17: Structura proiectului Business	28
Figura 18: Structura proiectului WebApplication	33
Figura 19: Documentație generată cu Swagger	34
Figura 20: Structura directorului Views	36
Figura 21: Structura directorului wwwroot	37
Figura 22: Directorul care conține informații despre popularea bazei de date	37
Figura 23: Secvență json pentru reprezentarea unui element Kanji	38
Figura 24: Exemplu toast	41
Figura 25: Diagramă Circliful	41
Figura 26: Captură de ecran a sesiunii de examinare	42
Figura 27: Captură ecran cu sesiunea de învățare	43
Figura 28: Captură ecran cu afișarea tuturor elementelor de vocabular în funcție de progresul lor... ..	43
Figura 29: Stările meniului principal	44
Figura 30: Etapă din crearea logo-ului	44
Figura 31: Schemă design pentru pagina principală	45
Figura 32: Ilustrații pentru stadiile itemilor vocabularului	45
Figura 33: Ilustrație din cadrul aplicației	47

Lista tabelelor

Tabelul 1: Valorile pentru distanța de timp dintre stadiile itemilor de vocabular	24
Tabelul 2: Proprietățile clasei DbContext	26
Tabelul 3: Configurarea cheii primare cu FluentAPI	26
Tabelul 4: Configurare SqlServer	26
Tabelul 5: Setarea șirului de conexiune în fișierul appsettings.json	27
Tabelul 6: Secvență de cod din clasa GenericRepo	29
Tabelul 7: Încărcarea datelor prin eager loading	29
Tabelul 8: Codul funcției ActiveForReview	31

Tabelul 9: Funcția de adăugare a unei entități în baza de date	31
Tabelul 10: Funcție pentru salvarea unui formular în baza de date	32
Tabelul 11: Fragment de cod responsabil pentru adăugarea middleware-ului MVC.....	33
Tabelul 12: Antetul metodei unui controller	35
Tabelul 13: Setarea layout-ului unui View	36
Tabelul 14: Secvență de cod Razor pentru afișarea traducerilor unui item	37
Tabelul 15: Setare autentificare bazată pe cookies	39

Cuprins

Lista figurilor.....	i
Lista tabelelor	i
Introducere.....	1
Context.....	1
Motivație	2
Semnificația titlului	3
Obiectivele generale ale lucrării.....	4
Descrierea sumară a soluției și structura lucrării.....	5
Contribuții	7
1. Proiectarea, structura și arhitectura aplicației.....	8
1.1. Structura aplicației	8
1.1.1. Vocabularul și sistemul spațiat de repetiție	9
1.1.2. Sesiunile de învățare și examinare	12
1.1.3. Gramatica și citirea.....	13
1.2. Scenarii de utilizare	14
1.3. Structura bazei de date	16
1.4. Concluzii	20
2. Implementare.....	22
2.1. Structura aplicației, arhitectura Onion și arhitectura web MVC.....	22
2.2. Entitățile domeniului, EF Core și SQL Server	24
2.3. Logica business.....	28
2.4. Nivelul prezentare.....	33
2.4.1. Punctul de pornire al aplicației	33
2.4.2. Swagger	34
2.4.3. Model-View-Controller	35
2.4.4. Sursele de informație pentru popularea bazei de date	38
2.4.5. Autentificarea bazată pe cookie-uri	39
2.5. Concluzii	40
3. Interfața web, elemente design și interacțiunea cu utilizatorul	41
Concluziile lucrării	46
Bibliografie	48

Introducere

Context

Fără îndoială o limbă străină reprezintă mereu oportunități și beneficii pentru dezvoltarea personală. Creierul se dezvoltă în urma procesului de învățare a unei limbi datorită asimilării unui sistem nou, vast și complex, ce constă din diverse reguli, specifice fiecărei limbi. Alt avantaj constă în îmbunătățirea memoriei, datorită însușirii noului vocabular. De altfel, procesul de luare a deciziilor al poligloților este mai facil și mai eficient. Acest lucru se datorează necesității de a analiza expresiile specifice unei limbi, ce uneori nu pot fi traduse sau conțin înțelesuri ascunse. Astfel, procesul de luare a deciziilor devine mai prudent și mai eficient. Contrar unei prime impresii, învățarea unei noi limbi duce la îmbunătățirea cunoștințelor asupra limbii materne. Anumite aspecte ale învățării limbii materne pot fi ignorate prin simplul fapt că limba maternă este o „limbă implicită”. Datorită comparațiilor frecvente, efectuate asupra noii limbi și a limbii materne, obținem o percepție mai bună asupra celei din urmă.

Mai mult sau mai puțin surprinzător este faptul că studiile arată o performanță sporită în cadrul academic a studenților multilingvi. Datorită supunerii creierului procesului de învățare a unei noi limbi sunt dezvoltate diverse abilități cognitive. Evident, un avantaj important este modul în care o limbă nouă influențează cultura unei persoane. O limbă nouă reprezintă poarta către cultura ei, astfel oferind posibilitatea lărgirii orizontului de înțelegere a culturii și a etniei respective. Fiind deschis unei noi culturi, o persoană poate deveni mult mai flexibilă în modul de gândire, având capacitatea de a vedea lucrurile din diferite puncte de vedere, ceea ce reprezintă o valoare importantă în globalizarea secolului curent. Desigur lista avantajelor poate continua cu multe alte elemente importante cum ar fi: extinderea potențialului în dezvoltarea carierei, consolidarea încrederii în sine, cunoașterea și dezvoltarea personală, etc. [1]

Odată cu stabilirea avantajelor, apare întrebarea: de ce anume limba japoneză?

Nu o să neg faptul că principala cauză în luarea acestei decizii a fost afecțiunea personală față de această limbă. Consider uimitoare această cultură, dar și limba în sine, împreună cu elementele ei complexe, dar atrăgătoare. Pe lângă asta, japoneza se numără printre cele mai interesante și utile limbi. Dacă anturajul și cultura japoneză nu te-au cucerit încă, putem enumera multe alte avantaje:

- Japonia ocupă locul 2 în economia mondială;

- Cunoașterea limbii japoneze oferă noi oportunități de afaceri și carieră, mai ales ținând cont de nivelul avansat economic al acestei țări;
- Japonia reprezintă o poartă către cultura și limbile asiatice;
- Japonia reprezintă o poartă pentru tehnologii, fiind creatoarea multor tehnologii renumite. (Shinkansen¹, calculatoarele de buzunar, roboții Android, etc)
- Japonezii și cultura lor se remarcă prin spiritul inovativ, oferind o perspectivă nouă și interesantă asupra lucrurilor;

și multe altele cel puțin la fel de importante. [2]

Motivație

Cei care au luat decizia să învețe această limbă au nevoie de un suport imens pentru că japoneza se poziționează printre cele mai dificile limbi. Pe lângă o cultură diversă și foarte diferită, japoneza se remarcă printr-un vocabular/alfabet foarte complex. În școli se învață 2.136 de cuvinte/caractere, deși în total există peste 50 mii.

FSI², o renumită agenție guvernamentală pentru învățarea limbilor străine, clasifică japoneza în ultima categorie, 8, ce reprezintă categoria celor mai dificile limbi. Ei afirmă că este nevoie de 88 săptămâni (2200 ore) de învățare intensă pentru a putea ajunge la un nivel de bază și a te simți confortabil în comunicarea cu japonezii nativi.

Să presupunem că cineva decide să învețe japoneza, fără să cunoască detalii despre această limbă. Mai întâi caută despre vocabularul acestei limbi, întrucât el este prima și cea mai anevoioasă barieră. Apoi află că există 3 alfabet: Kanji, Hiragana și Katakana. Mai apoi află că imensul alfabet „Kanji” este format și el din mai multe componente. Astfel, încă de la primele încercări, persoana în cauză poate fi ușor derutată iar decizia de a învăța japoneza devine incertă.

Prin urmare, scopul aplicației Foxy constă în eficientizarea și facilitarea procesului de învățare a limbii japoneze. Fiind printre pasionații de limbă și cultură japoneză am constatat de multe ori că există puține aplicații/platforme pentru limba japoneză, care să structureze într-un mod eficient toată informația (vocabularul, gramatica și citirea) astfel încât să ofere utilizatorului senzația de control asupra tuturor noțiunilor învățate, ci nu doar o impresie superficială, fără conexiuni logice asupra termenilor. Câteva aplicații care merită atenție atunci când vine vorba de învățarea limbii japoneze sunt: *Wanikani*, *Duolingo*, *Wasabi*, *Memorise*.

¹ Shinkansen - Termen folosit pentru o serie de trenuri de mare viteză japoneze (cu viteze de 210 km/h).

² FSI - Foreign Service Institute. Mai multe detalii aici: <http://www.effectivelanguagelearning.com>

Aflându-mă în situația unui „elev începător”, folosind anumite aplicații, am simțit de multe ore lipsa anumitor structuri și conexiuni ce mi-ar facilita procesul de învățare, cum ar fi: „Ce fac acum că am învățat câteva cuvinte?”, „Când pot începe lecțiile de gramatică?”, „La ce nivel de cunoaștere al limbii mă aflu?”, etc.

În urma acestor necesități, am decis să construiesc o aplicație pentru învățarea japonezei, cu un nou sistem de structurare a informațiilor, care să acopere majoritatea domeniilor limbii: vocabular, gramatică, citire și înțelegere. Avantajul de bază al acestei aplicații constă în utilizarea unui sistem spațiat de repetiție (SRS³), folosit pentru învățarea vocabularului. Acest sistem constă în memorarea unei cantități mari de informație prin coordonarea și creșterea intervalelor de timp ale memorării aceluiasi element. Figura 1 redă un exemplu clar și concis al acestui sistem.

Aplicația va fi structurată în trei componente: *Vocabular*, *Gramatică* și *Citire/Înțelegere*. Prototipul aplicației conține și un parser OCR (optical recognition character) pentru a permite utilizatorului să exerseze scrierea (de mână) a caracterelor sau căutarea simplă și eficientă a caracterelor rar întâlnite și necunoscute.

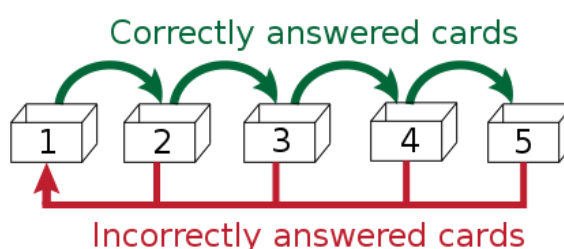


Figura 1: Exemplu sistem de repetiție (Sursă: Wikipedia)

Semnificația titlului

De ce „Foxy”? Pentru cei care nu cunosc cultura și mitologia japoneză acest titlu ar putea părea ciudat sau poate chiar nepotrivit. Foxy derivă din cuvântul englez „fox” (vulpe) și se referă la puiul de vulpe, căpătând o conotație jucăușă. În mitologia japoneză vulpea este considerat un personaj înțelept și imortal, cu abilități magice.

Vulpea – *kitsune* (狐) în japoneză – este o creatură divină, strâns legată de religia japoneză Șintoism. O treime din toate templurile existente sunt dedicate acestui personaj divin. Aceste temple sunt faimoase în Kyoto și reprezintă un important loc turistic, purtând numele

³ SRS – Spaced Repetition System. Mai multe detalii aici: https://en.wikipedia.org/wiki/Spaced_repetition

de „Fushimi Inari”. Specificul acestor temple constă în multe sculpturi ale vulpilor și predominarea culorii roșu. [3]

Mitologia acestui caracter este foarte vastă și extrem de interesantă, dar trăsătura principală ce m-a determinat să aleg acest titlu a fost „înțelepciunea”. Vulpile sunt considerate extrem de înțelepte. Având capacitatea de a se transforma în oameni (conform mitologiei), ele sunt considerate excelenți prieteni și chiar parteneri de viață.

Am ales denumirea engleză în favoarea celei japoneze întrucât aplicația este destinată începătorilor, iar un termen mai complicat ar putea fi derutant. În Figura 2 este prezentat logo-ul aplicației. Acesta conține simbolul unei vulpi. În centrul acestuia se află simbolul flagului Japoniei.



Figura 2: Logo-ul aplicației

Obiectivele generale ale lucrării

Foxy este o aplicație web pentru învățarea limbii japoneze, care are ca scop pregătirea utilizatorului pentru nivelul N5 – JLPT⁴. Acest nivel este considerat nivelul de bază. JLPT reprezintă un test internațional de cunoaștere a limbii japoneze pentru vorbitorii non-nativi. Acest test cuprinde cunoștințele generale asupra limbii, abilitățile de citire și ascultare/audiție. Se ține în Japonia de două ori pe an și constă din cinci nivele. Nivelul de bază este N5, iar nivelul expert este N1. Nivelul N5 are drept cerințe citirea și înțelegerea expresiilor tipice, cât

⁴ JLPT - Japanese-Language Proficiency Test. Test pentru stabilirea nivelului limbii japoneze pentru vorbitorii non-nativi.

și a subiectelor zilnice, discuțiilor specifice mediului academic primar (scrise în *hiragana*, *katakana* și *kanji* de bază).

Hiragana și *Katakana* reprezintă două alfabete/sisteme de scriere fonetică în limba japoneză. Fiecare caracter hiragana reprezintă un grup de unul sau două foneme (sunete), în general o consoană urmată de o vocală, ca de exemplu „*sa*”, „*to*”, „*mi*”, etc. Caracterele *katakana* se folosesc pentru scrierea cuvintelor de origine străină, intrate deja în limba curentă.

Aplicația *Foxy* prevede cunoașterea preventivă a acestor alfabete, întrucât obiectivul acesteia se focalizează pe învățarea celui de-al treilea alfabet: *Kanji*. Memorarea celor două alfabete, *Hiragana* și *Katakana*, nu prezintă un nivel de dificultate sporit, fiecare având în jur de 45 de caractere. Pe lângă asta, aplicația oferă acces imediat la aceste alfabete, fiind afișate mereu pe ecran în cadrul unui meniu vertical.

Un alt sistem de scriere al limbii japoneze este *Rōmaji*. Acesta constă în folosirea caracterelor din alfabetul latin pentru a transla un text scris în limba japoneză. El va fi utilizat des în aplicație pentru a reda citirea corectă a vocabularului.

Mult mai vast și mai dificil de memorat este alfabetul *Kanji*. El se consideră a fi derivat din alfabetul chinez și are peste 54.000 de caractere. Această aplicație prezintă un set de caractere al acestui alfabet, corespunzător nivelului N5 (menționat anterior), conform sistemului spațiat de repetiție pentru memorarea eficientă. De altfel, aplicația conține elementele de gramatică și citire, corespunzătoare nivelului de bază, structurate drept formulare și teste.

Acestea fiind spuse, obiectivul principal al aplicației este de a oferi utilizatorului o experiență cât mai bogată și mai plăcută în procesul de învățare al limbii japoneze, bazată pe: conexiuni logice între elemente; acces rapid către informația atât necunoscută, cât și cunoscută; informații relevante în legătură cu stadiul de învățare al elementelor.

Descrierea sumară a soluției și structura lucrării

Acest document conține prezentarea detaliată a aplicației *Foxy*. Pe lângă arhitectură și proiectare, este expusă implementarea pentru partea de back-end⁵ și partea de front-end⁶. Partea de back-end include modul de stocare, „best practice”⁷-urile folosite, detalii despre autorizare și fragmente de implementare pentru anumite funcționalități. Partea de front-end vine cu

⁵ Back-end – Parte logică a aplicației. De obicei este format din server și bază de date.

⁶ Front-end – Parte a aplicației care face legătura cu utilizatorul prin intermediul unei interfețe grafice. Include crearea design-ului și codul pentru dezvoltarea interfeței.

⁷ Best practice – Metodă sau tehnică considerată superioară altor alternative datorită rezultatelor obținute sau datorită apartenenței unui standard.

explicații care pornesc de la detalii tehnice de implementare și elemente de design grafic, până la explicația simbolurilor și motivarea alegerii anumitor termeni și denumiri.

Arhitectura aplicației oferă un suport bun pentru o mai bună înțelegere a părților de implementare, dar și pentru anumite funcționalități ale aplicației. Acest capitol va include diverse diagrame și scheme pentru a exemplifica structurarea componentelor logice, scenarii de utilizare și fluxuri ale aplicației.

Contribuții

Proiectul cuprinde ideile personale îmbinate cu elemente clasice specifice aplicațiilor pentru învățarea limbilor străine. Am conceput întreaga structură a aplicației ce conține elemente precum: modul de creare și structurare a entităților, scenariile de utilizare, mecanismele necesare pentru satisfacerea scenariilor de utilizare, ilustrații și elemente de design corelate cu tema și motivul aplicației.

Deși sistemul spațiat de repetiție este o tehnică des folosită în aplicațiile pentru învățarea limbilor străine, etapele și modul de implementare al sistemului folosit în aplicația Foxy au fost stabilite și create de către mine.

Întreaga aplicație a fost creată de la zero, folosind șabloane arhitecturale potrivite și librării open-source. Informațiile introduse în baza de date au fost selectate de către mine din surse oficiale, cărți și alte aplicații. Am luat această decizie pentru a putea personaliza în întregime caracterul datelor.

În procesul creării aplicației un suport bun și o sursă de idei a fost experiența personală. Fiind un fost utilizator al aplicațiilor de învățare a limbii japoneze, am încercat să introduc în aplicația Foxy elemente de care aș fi avut nevoie la rândul meu.

1. Proiectarea, structura și arhitectura aplicației

1.1. Structura aplicației

Proiectarea are unul dintre cele mai importante roluri în crearea unei aplicații. Ea reprezintă baza peste care urmează să se formeze aplicația. Deciziile pe care le-am luat atunci când am creat arhitectura *Foxy* au fost definite de câteva principii: structurarea elementelor cât mai clar și logic, respectarea tuturor condițiilor limbii și o experiență agreabilă pentru utilizator.

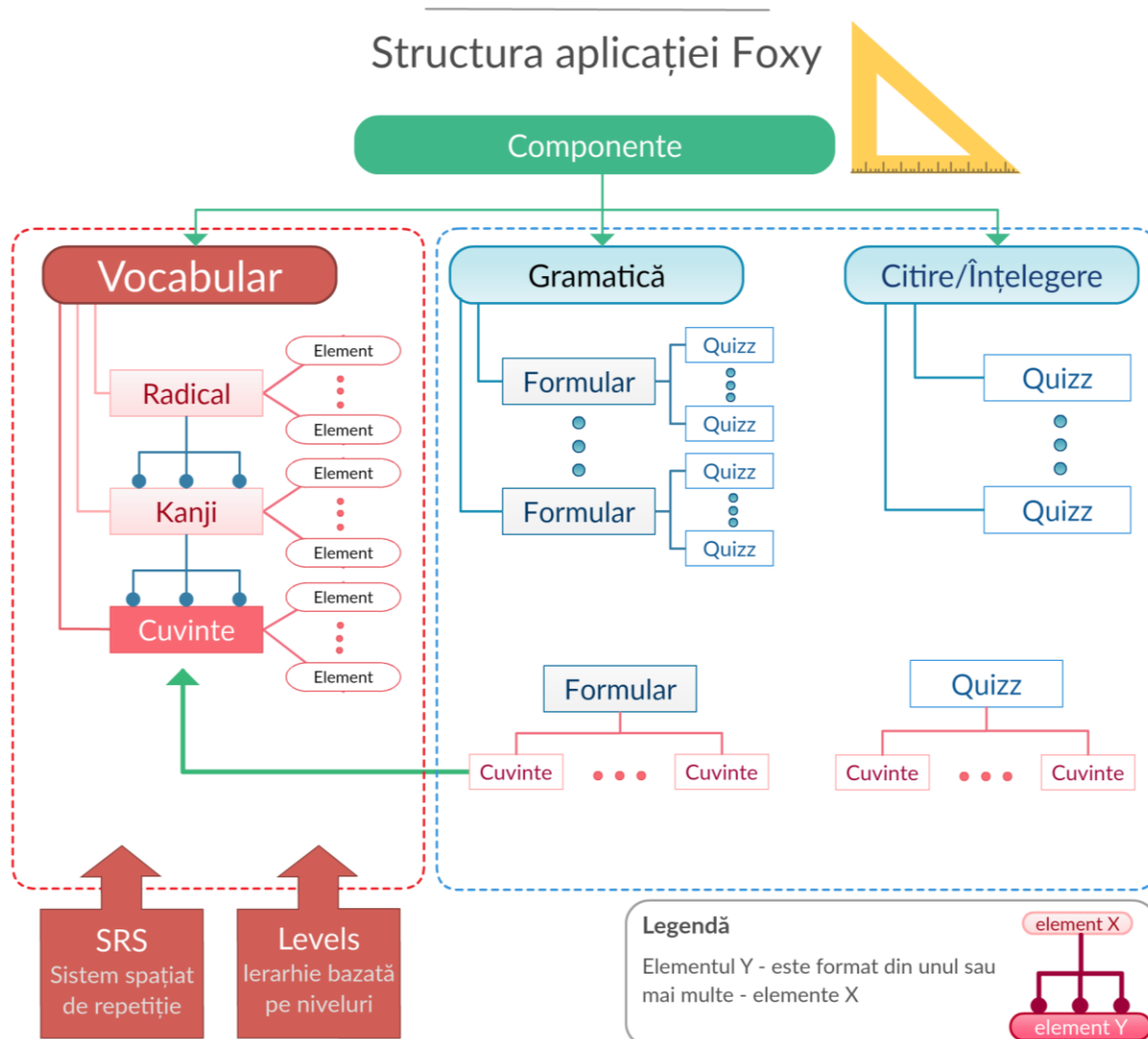


Figura 3: Diagrama funcțională a aplicației

Figura 3 reprezintă diagrama funcțională a aplicației. Am ales să structurez aplicația în trei mari componente – Vocabular, Gramatică și Citire/Înțelegere – întrucât consider acest mod de organizare printre cele mai eficiente în învățarea unei limbi. După cum se poate observa și în Figura 3, cea mai complexă parte este Vocabularul. Celelalte componente, Gramatica și Citirea, sunt legate strâns de Vocabular. Altfel spus, fără a cunoaște elementele vocabularului este mult mai greu de asimilat unități de gramatică și de a completa teste. Cu toate acestea, utilizatorul poate accesa aceste elemente și poate interacționa cu ele, pentru a nu-l limita în alegeri și a încuraja procesul de învățare. În plus, aplicația oferă suport pentru aceste cazuri, expunând mereu cuvintele folosite în formularele/testele de gramatică sau citire cu accentuarea cuvintelor ce nu au fost învățate încă.

1.1.1. Vocabularul și sistemul spațiat de repetiție

Vocabularul este componenta aplicației cu cea mai complexă structură. Un element al vocabularului reprezintă o unitate lexicală. În continuare voi folosi termenul *Item* pentru un element al vocabularului. Conform limbii japoneze, vocabularul poate fi structurat în mai multe părți:

- *Radicali* – Elemente ale limbii japoneze ce pot fi văzute ca niște piese ale unui puzzle în asamblarea unui Kanji. Radicalii au doar denumire și înțeles, nu posedă o citire propriu-zisă.
- *Kanji* – Elemente ce reprezintă caractere ale limbii japoneze. Este cel mai vast sistem de scriere și este derivat din alfabetul chinez. În unele surse poate fi referențiat cu denumirile de „pictograme” sau „ideograme”. Kanji reprezintă componente ale cuvintelor japoneze.
- *Cuvinte* – Unități lexicale ce formează frazele limbii japoneze. Un cuvânt este o combinație dintre Kanji și caracterele alfabetelor *hiragana* și *katakana*. Un cuvânt poate fi format doar dintr-un kanji. De altfel, un cuvânt poate să nu conțină niciun kanji (doar *hiragana* și *katakana*).

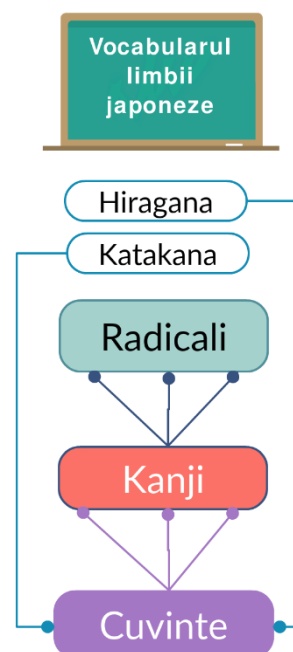


Figura 4: Structura vocabularului limbii japoneze

În Figura 4 am structurat schematic elementele enumerate anterior pentru o mai bună înțelegere. În aplicația *Foxy* vocabularul respectă structura respectivă și este format din trei

componente. Astfel, fiecare item poate avea unul dintre cele trei tipuri: Radical, Kanji, Cuvânt. Un item este reprezentat de un nume, unul sau mai multe înțelesuri (traduceri) și una sau mai multe citiri. Citirea unui item va fi reprezentată de o secvență de caractere hiragana sau katakana, întrucât acestea reprezintă niște silabe și se pliază perfect pentru a interpreta citirea unui item. Dacă itemul face parte din primul tip – Radical – atunci acesta nu are citire. Un item de tip *kanji* conține o colecție de elemente *radical*. Prin urmare, un item de tip *cuvânt* conține o colecție de elemente *kanji*.

După ce am stabilit care sunt caracteristicile unui item, e timpul pentru a descrie interacțiunea utilizatorului cu elementele vocabularului. Fiecărui item îi este asociat un progres al utilizatorului. Altfel spus, fiecare item înglobează informații precum: câte răspunsuri corecte sau greșite a dat utilizatorul, când a răspuns ultima dată, dacă se află pe lista itemilor favoriți, etc. După cum este reprezentat în Figura 3 aplicația folosește un sistem ierarhic de niveluri. Astfel, utilizatorul se va afla mereu la un anumit nivel. Un nivel este reprezentat de o colecție de itemi ai vocabularului.

După cum poate fi deja dedus, un item conține nivelul la care poate fi accesat. Itemii corespunzători unui nivel mai mare decât cel curent, sunt considerați blocați. Un item blocat împiedică înregistrarea progresului asupra lui. Totuși, utilizatorul are acces la toată informația despre itemul respectiv. Mai mult, sunt permise acțiuni precum: adăugarea unei notițe în legătură cu înțelesul sau citirea itemului, adăugarea itemului la lista de favorite, adăugarea sinonimelor.

Pentru a monitoriza progresul asupra unui item, am decis să organizez informația sub formă de stadii. Acest mod de organizare este reprezentat în Figura 5. În conformitate cu etapele prezentate în imagine voi enumera în ordine cronologică calea parcursă de un item:

- Inițial itemul este blocat, stare ce se menține până când utilizatorul ajunge la nivelul corespunzător itemului. Elementul se deblochează atunci când apare evenimentul de "Level Up", ceea ce în alte cuvinte înseamnă trecerea către următorul nivel (în acest caz nivelul corespunzător itemului).
- Odată deblocat, elementul trece în sesiunea de lecție. Despre această sesiune sunt prezentate mai multe detalii în subcapitolul următor. Aici, după cum ne sugerează și denumirea, utilizatorul face cunoștință cu noul element și încearcă să memoreze citirea și înțelesul itemului, acestea urmând să fie verificate în sesiunile de evaluare ce urmează.
- După sesiunea de învățare, urmează o sesiune de examinare. Despre sesiunea de examinare sunt prezentate detalii în subcapitolul următor. Pe scurt, scopul acestei sesiuni este de a evalua cunoștințele utilizatorului asupra itemului. Dacă

răspunsul oferit de utilizator a fost corect, elementul trece la următoarea etapă. Atât timp cât răspunsul utilizatorului este greșit, itemul rămâne în setul elementelor pentru sesiunea de învățare.

- După cum poate fi observat în Figura 5, prima sesiune de examinare care a avut loc cu succes este urmată de introducerea elementelor în ciclul stadiilor propriu-zise. Scopul utilizatorului este să treacă fiecare item prin cele patru stadii. Ajuns la ultimul stadiu, *Flourished*, itemul este considerat învățat și nu se mai întoarce la stadiile anterioare.

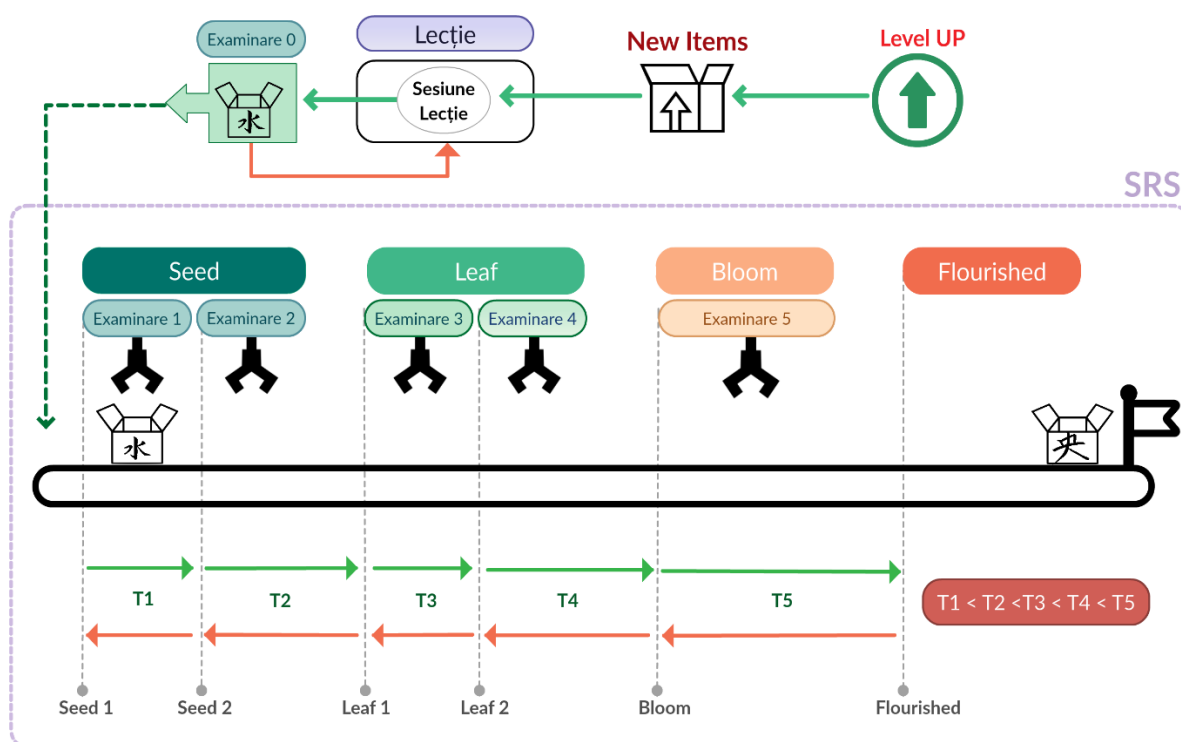


Figura 5: Stadiile progresului fiecărui element din vocabular

În continuare voi explica procesul prin care un element trece de la un stadiu la altul, dar și motivația alegerii denumirilor stadiilor. Cele patru stadii sunt denumite conform etapelor ciclului de dezvoltare al unei plante, pentru a crea o alegorie între o plantă/floare și un item. Inițial planta se află în stadiul de sămânță – *Seed* – asemănător cunoștințelor utilizatorului asupra itemului respectiv. După o anumită perioadă, utilizatorul începe să aibă o mai bună cunoaștere a itemului, astfel, plantei îi apare o frunzuliță – *Leaf*. Ulterior, itemul devine deja bine fixat în memorie, ajungând astfel la etapa de înmugurire – *Bloom*. Ultima etapă este înflorirea – *Flourished*. La această ultimă etapă itemul poate fi recunoscut instant de către utilizator. Am decis bifurcarea stadiilor *Seed* și *Leaf* pentru a avea în etapa inițială mai multe

evaluări la intervale mici de timp. După părerea mea, această structură stimulează memorarea itemului.

Modul de deplasare între stadii este bazat pe sistemul spațiat de repetiție. Aflându-se la un anumit stadiu, utilizatorul trebuie să aștepte o sesiune de examinare pentru a putea deplasa itemul către următoarea etapă. Timpul dintre examinări este strict crescător, după cum este prezentat și în Figura 5. Această este o condiție importantă a sistemului de repetiție, astfel încât de ea depinde memorarea efectivă. În timpul unei examinări, dacă răspunsul este corect, stadiul itemului avansează, în caz contrar, stadiul itemului revine la cel precedent. De exemplu, dacă itemul se află la stadiul *Bloom* și utilizatorul oferă un răspuns corect, stadiul itemului devine *Flourished*. Dacă răspunsul este greșit, stadiul itemului devine *Leaf 2*.

Acestea fiind spuse, mai rămâne de menționat condiția de trecere către un nou nivel pentru utilizator. Din setul de itemi corespunzător nivelului curent, se țin cont doar de elementele de tip *Radical* și *Kanji*. Pentru a trece la următorul nivel un anumit procentaj al „elementelor însușite” trebuie depășit. Prin „elemente însușite” se înțelege elementele ce au ajuns la stadiul *Leaf*, altfel spus, elementele ajunse în setul de învățare trebuie să treacă prin trei examinări cu succes realizate consecutiv. Procentajul acestor elemente în momentul curent este setat la 80%.

O să explic succint de ce condiția de trecere la următorul nivel nu ține cont de elementele de tip *cuvânt*. Itemii *cuvânt* se diferențiază prin mecanismul de deblocare. În cazul lor nu este suficientă condiția de a avea nivelul curent mai mare sau egal decât nivelul cerut de către item. Acest fapt este cauzat de structura itemilor. Itemii *cuvânt* pot fi formați din itemi *kanji*. Astfel, apare o nouă condiție ce trebuie satisfăcută pentru a debloca acești itemi. Pentru ca un item *cuvânt* să fie deblocat toți itemii *kanji* ce îl formează trebuie, la rândul lor, să fie deblocați.

1.1.2. Sesiunile de învățare și examinare

Pentru ca un item al vocabularului să se deplaseze de la un stadiu la altul și pentru a implementa cu succes sistemul de repetiție aceste sesiuni sunt esențiale.

Folosesc termenul „sesiune” pentru a oferi o imagine cât mai clară asupra implementării reale. O sesiune este reprezentată printr-o secvență de elemente ale vocabularului. Dimensiunea sesiunilor depinde de tipul lor. Tipurile posibile sunt *Învățare* și *Examinare*.

Dimensiunea prestabilită a sesiunii de învățare (*lesson session*) este cinci. Acest număr poate fi modificat la dorința utilizatorului, fără să depășească însă anumite limite, întrucât o sesiune de învățare cu un număr mare de elemente devine greoaie și inefficientă. În cadrul sesiunii de învățare utilizatorul face cunoștință cu noile elemente, însușește înțelesul și citirea,

dar și componentele ce formează acest item. Pe lângă asta, fiecare unitate ce trebuie memorată, cum ar fi înțelesul și citirea, conține un mnemonic care ajută utilizatorul să memoreze ușor elementul, prin conexiuni logice caracteristice itemului. Tot în cadrul acestei sesiuni, utilizatorul poate adăuga notițe specifice elementelor, poate adăuga sau șterge sinonime și poate audia cum se citesc elementele.

Pentru a forma o sesiune de învățare sunt extrase un număr de elemente din setul itemilor expuși pentru învățare conform nivelului curent. Modul în care se aleg itemii pentru o sesiune nu este aleatoriu. Elementele vor fi mereu aranjate astfel încât să nu apară probleme în dependențele itemilor. De exemplu, dacă un item *kanji* este format din doi itemi *radical*, atunci itemul *kanji* nu va apărea niciodată înaintea itemilor *radical*. Un scenariu ar fi ca elementele *radical* să fie selectate într-o sesiune anterioară sesiunii ce conține elementul *kanji*. Un alt scenariu ar fi ca toate cele trei elemente să apară în aceeași sesiune, dar aranjate în ordine, adică elementul *kanji* să fie poziționat după celelalte două elemente ce îl formează.

Sesiunea de examinare (*review session*) reprezintă o secvență de elemente formată din mulțimea tuturor elementelor active pentru evaluare. Un element este considerat activ pentru evaluare dacă a trecut suficient timp de la ultima evaluare, acest timp fiind stabilit conform stadiului elementului. În cadrul evaluării unui element utilizatorul trebuie să prezinte câte un răspuns pentru înțelesul și citirea elementului. Dacă răspunde corect la ambele, răspunsul final este considerat corect, în caz contrar, greșit. Dacă răspunsul este corect, stadiul elementului avansează, altfel – regresează. Foarte important este faptul că dacă unui item i s-a acordat un răspuns greșit în cadrul evaluării, chiar dacă acesta a trecut la un stadiu inferior, acesta este considerat în continuare activ pentru evaluare. Astfel, un item este permanent prezent în sesiunea de evaluare atât timp cât primește un răspuns greșit.

1.1.3. Gramatica și citirea

Partea de gramatică a aplicației este organizată în mai multe formulare. Fiecare formular are un status în legătură cu procentul vocabularului deja învățat, necesar pentru formularul respectiv. Acest fapt ajută utilizatorul în a înțelege cât de pregătit este pentru a parcurge formularul. Un formular conține informații despre un anumit subiect al gramaticii limbii japoneze. Fiecare dintre ele conține un set de întrebări. Aceste întrebări sunt structurate sub formă de teste formate din 5 întrebări cu mai multe variante de răspuns. Astfel, utilizatorul poate obține un punctaj pe o scară de la 0 la 5. Există și un punctaj referitor la formular, ce se actualizează odată cu punctajul obținut în urma completării unui test. Acest lucru permite

utilizatorului să observe ce formulare au un punctaj scăzut, pentru a le putea revizui și a-și consolida cunoștințele.

Partea de citire și înțelegere este formată din mai multe teste care, la fel, conțin mai multe variante de răspuns. Majoritatea acestor teste vor fi create pe baza unor surse oficiale cu scopul de a pregăti utilizatorul pentru testul JLPT, nivelul N5.

1.2. Scenarii de utilizare

Diagramele ce determină interacțiunea cu utilizatorul reprezintă un instrument important în definirea unui sistem. Am folosit diagrame UML⁸ pentru a reprezenta câteva scheme caracteristice aplicației. UML este un mod de a vizualiza programele software folosind o colecție de diagrame. Unul dintre avantajele utilizării UML este structurarea sistemului în subcomponente, sub forma unei ierarhii, oferind astfel o mai bună înțelegere.

În Figura 6 am reprezentat cazuri de utilizare ale aplicației pentru un utilizator autentificat. Această schemă este reprezentată printr-o diagramă UML de tipul *Use Case*. Diagramele *Use Case* modelează funcționalitatea sistemului folosind actori și scenarii. În diagrama din Figura 6 actorul este un utilizator autentificat. Am decis să reprezint în diagramă doar utilizatorul autentificat întrucât aici avem cele mai multe și mai complexe scenarii. Un utilizator vizitator nu are atât de multe scenarii de prezentat. Acesta poate face acțiuni elementare precum înregistrarea unui cont, autentificarea și vizualizarea descrierii aplicației.

Utilizatorului autentificat (care a efectuat cu succes acțiunea de *Login*) îi sunt puse la dispoziție toate funcționalitățile aplicației. Pentru a organiza mai bine informația am împărțit scenariile de utilizare în 3 secțiuni. Prima secțiune este *contul utilizatorului*. Aici utilizatorul poate efectua modificări asupra profilului său. Un scenariu specific pentru tema aplicației este modificarea dimensiunii sesiunii de învățare, despre care au fost prezentate detalii în subcapitolul 1.1.2 .

Secțiunea gramatică și citire conține două scenarii principale: accesarea unui formular și completarea unui test. Prin accesarea unui formular utilizatorul poate vedea conținutul acestuia și informații precum vocabularul necesar și punctajul obținut în urma completării testelor formularului respectiv. etc. Pe lângă asta, utilizatorul poate adăuga notițe pentru fiecare formular sau îl poate marca ca fiind favorit. După cum poate fi observat în diagramă, completarea unui test se ramifică în două categorii în funcție de tipul acestuia. Un test poate fi pentru gramatică sau pentru citire. Completarea unui test extinde scenariul de accesare a unui

⁸ UML - Unified Modeling Language. Mai multe detalii aici: <https://www.smartdraw.com/uml-diagram/>

formular, întrucât utilizatorului îi este oferită posibilitatea de a începe un test după ce a accesat un formular.

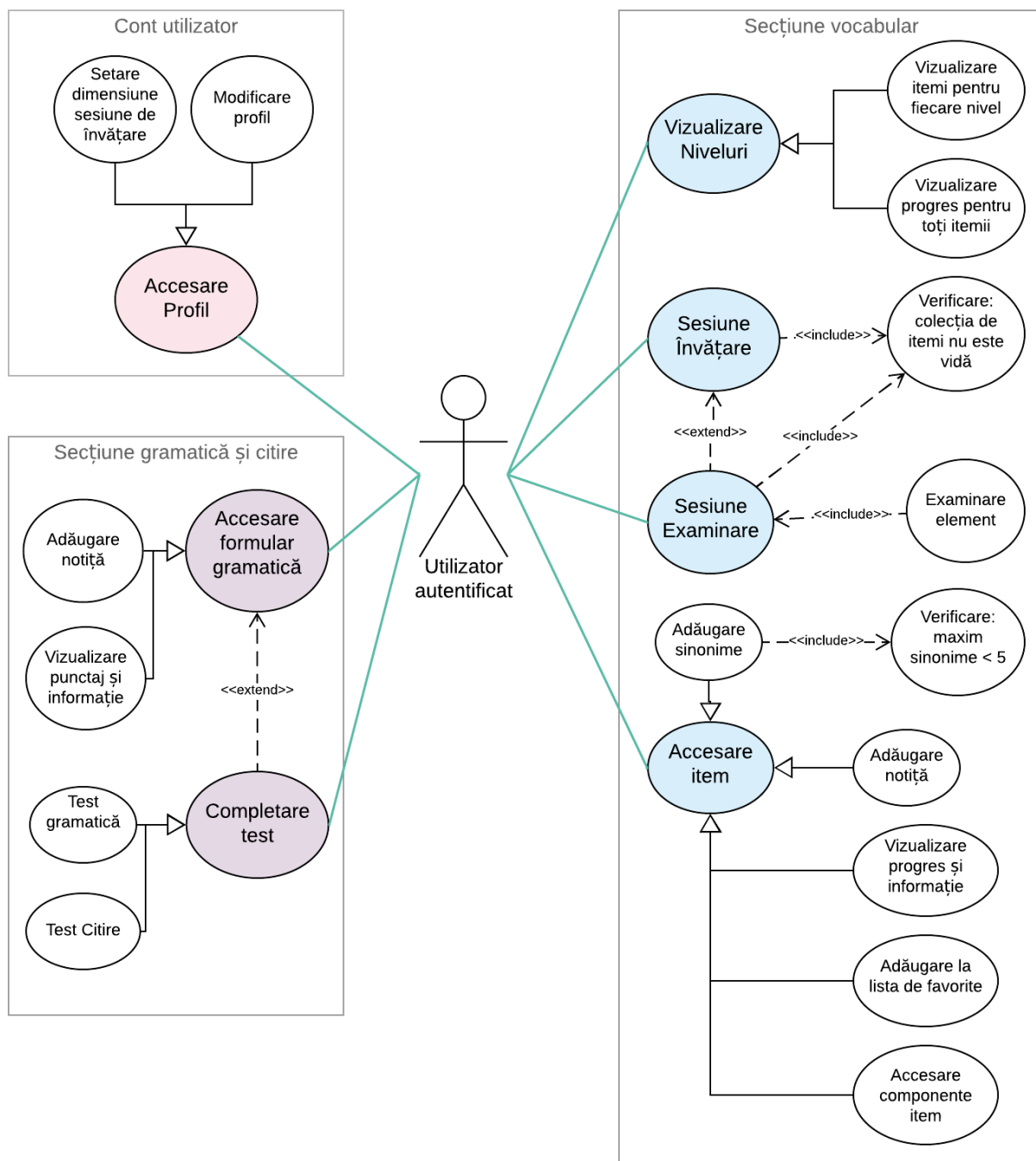


Figura 6: Diagrama Use Case pentru utilizatorul autentificat

Secțiunea vocabularului este un pic mai complexă decât celelalte datorită numărului de scenarii și a relațiilor de dependență. Am inclus în această secțiune patru scenarii generale, ce pot fi detaliate ulterior prin acțiuni specifice: vizualizare niveluri, sesiune învățare, sesiune examinare, accesare item. Utilizatorul poate vizualiza itemii vocabularului corespunzător fiecărui nivel, lucru extrem de important în crearea unei idei generale asupra ceea ce urmează

să fie însușit. Mai mult, utilizatorul poate vedea întreaga colecție de elemente ale vocabularului împreună cu stadiile la care se află. Un scenariu esențial este accesarea unui item, pentru că aici are loc expunerea informației itemul dar și interacțiunea utilizatorului cu elementul dat. Câteva dintre acțiunile utilizatorului incluse în acest scenariu sunt: adăugarea notițelor, adăugarea sau ștergerea sinonimelor, adăugarea la lista favoritelor, vizualizarea progresului itemului. Progresul unui item înglobează informații precum: denumirea stadiului, numărul de răspunsuri corecte și greșite, cea mai lungă secvență de răspunsuri corecte consecutiv. Trebuie menționat faptul că atunci când este adăugat un sinonim este necesară verificarea numărului total de sinonime. Acesta nu trebuie să depășească o anumită limită; aplicația permite în momentul actual maxim cinci sinonime. Această limită este introdusă pentru a menține în ordine informația ce trebuie asimilată pentru fiecare item, de altfel, scopul sinonimelor este de oferi utilizatorului libere în exprimare și eficientizarea procesului de memorare. Totuși, din motive evidente, sinonimele nu vor fi considerate răspunsuri corecte atunci când se examinează înțelesul(*meaning-ul*) unui item, întrucât acestea nu au niciun fel de limitare și se pot abate de la sesul real al elementului.

Celelalte două scenarii, *sesiune examinare* și *sesiune învățare*, au câteva aspecte în comun. În primul rând, înainte de a începe sesiunea, trebuie verificat dacă colecția din care urmează să fie extrase elementele nu este vidă. În cazul în care nu avem niciun element în colecție, sesiunea nu poate începe. După ce un element este vizitat în sesiunea de învățare, pentru a-l trece la următoarea etapă, acesta trebuie să treacă prin sesiunea de examinare. Prin urmare, sesiunea de examinare extinde sesiunea de învățare.

1.3. Structura bazei de date

Baza de date reprezintă scheletul aplicației. Aici are loc modelarea obiectelor. Procesul creării structurii bazei de date este foarte important pentru că o structură nepotrivită propagă erori în nivelele superioare. Am ales să reprezint structura bazei de date printr-o diagramă ERD⁹. O diagramă ERD (Entity Relationship Diagram) reprezintă relațiile dintre mai multe entități stocate în baza de date.

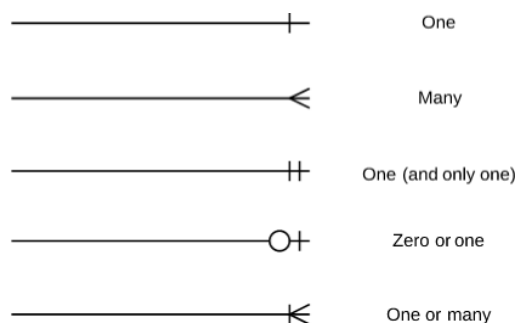
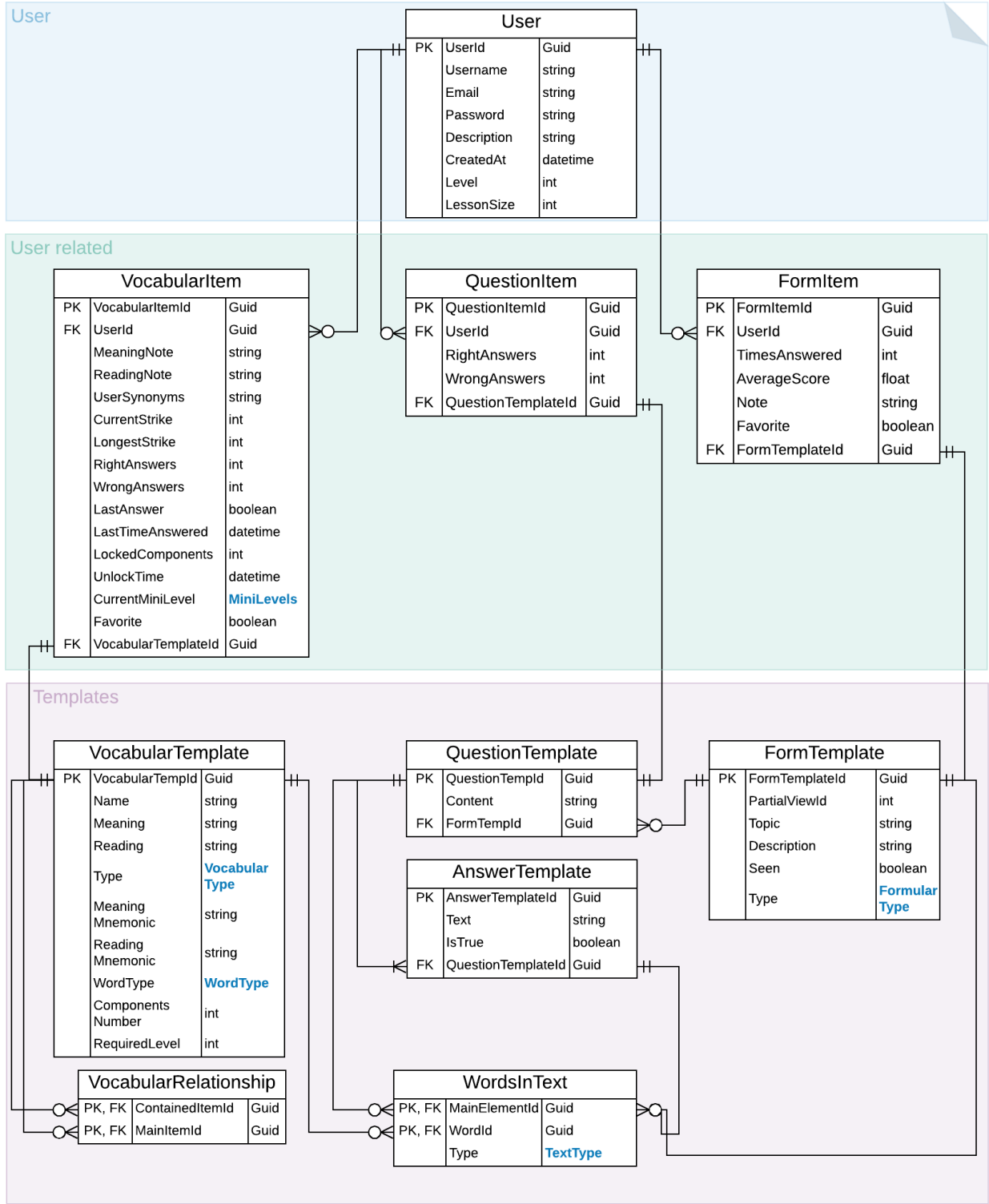


Figura 7: Relațiile folosite într-o diagramă ERD

⁹ ERD - Entity Relationship Diagram. Mai multe detalii aici: <https://www.smartdraw.com/entity-relationship-diagram/>

Figura 7 reprezintă notațiile efectuate în cadrul diagramelor ERD pentru a descrie relațiile dintre entități. Acestea vor fi utile în înțelegerea diagramei din Figura 8.



Remark:
BlueTypes represents enumerables and are represented in data base as BYTE

Figura 8: Diagrama bazei de date (ERD)

O să încep descrierea bazei de date, din Figura 8, prin prezentarea secțiunilor logice în care se încadrează tabelele. O entitate ce joacă rolul principal este utilizatorul, reprezentat de tabela *User*. Entitățile ce reprezintă elementele supuse procesului de învățare, cum ar fi vocabularul și gramatica, formează o secțiune a elementelor șablon (*template*). Am ales să le numesc așa pentru că un element poate fi văzut ca un tipar ce înglobează informațiile de bază: denumirea, citirea, înțelesul, etc. Aceste elemente șablon reprezintă un fundament pentru alte entități.

Entitatea *User* determină crearea unei noi secțiuni ce cuprinde elemente specifice unui anumit utilizator, dar care au la bază un element șablon. Această secțiune formează mecanismul de înregistrare a progresului utilizatorului pentru fiecare item ce trebuie însușit, iar elementele șablon permit evitarea repetării informației.

Voi continua prin detalierea secțiunii elementelor șablon, ce se regăsește în Figura 8 în partea de jos a diagramei, cu numele *Templates*. Această secțiune conține cea mai complexă structură. Aici are loc modelarea majorității entităților folosite în aplicație. După cum este redat în diagramă, în această secțiune întâlnim tabele caracteristice pentru: vocabular, formulare, întrebări și răspunsuri. În alte cuvinte, regăsim aici tot materialul de învățare folosit de aplicație, structurat în tabele ale bazei de date.

Tabelul *VocabularyTemplate* se referă la un șablon pentru un item al vocabularului. Observăm aici că tabela conține atribute precum: nume, înțeles, tip, mnemonici, nivelul necesar pentru a debloca elementul, etc. Atributul *Type* se referă la tipul itemului – Radical, Kanji sau Word. Am decis să creez tabele separate pentru aceste trei tipuri, întrucât majoritatea atributelor sunt comune. Conform structurii vocabularului, un item poate fi format din mai mulți itemi. Prin urmare, avem nevoie de o tabelă care să reprezinte relația respectivă. Această tabelă este *VocabularyRelationship* și este formată din id¹⁰-ul itemului principal și id-ul unei componente ce îl formează. De exemplu, itemul *cuvânt* 学生 este format din doi itemi *kanji*: 学 și 生. Astfel, pentru acest cuvânt vom avea în tabela *VocabularyRelationship* două intrări, ambele având setat pentru atributul *MainItemId* id-ul itemului *cuvânt*, iar pentru atributul *ContainedItemId* se va seta id-ul itemului *kanji* pentru fiecare intrare (rând al tablei). Relațiile dintre tabelele *VocabularyTemplate* și *VocabularyRelationship* sunt stabilite conform diagramei, în urma următorului raționament: un element din *VocabularyTemplate* nu trebuie să apară neapărat în tabela *VocabularyRelationship*, deci poate să apară de zero sau de mai multe ori; un atribut al

¹⁰ Id – unitate ce poate fi reprezentată printr-un număr sau printr-un șir de caractere și identifică în mod unic elementul căruia îi aparține.

tabelei *VocabularRelationship*, fiind și cheie străină, poate conține unul și doar unul dintre elementele tabelei *VocabularTemplate*.

În Figura 8 se pot observa anumite tipuri ale atributelor ce nu se încadrează în tipurile primitive. Aceste tipuri au de fapt la bază tipul enumerare, care în baza de date este interpretat ca fiind byte.

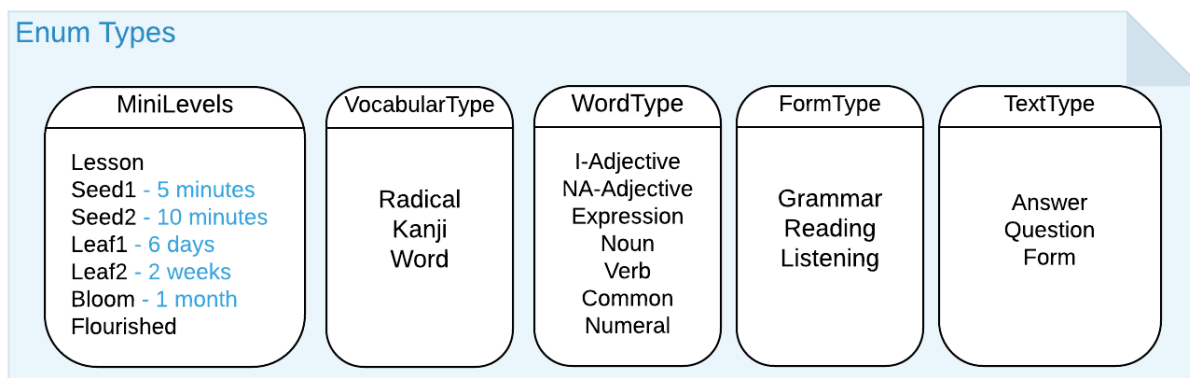


Figura 9: Tipurile enumerare folosite în baza de date

Tabela *VocabularTemplate* conține un atribut de tipul *WordType*, reprezentat în Figura 9. Acest tip reprezintă clasificarea în părți de vorbire a cuvintelor japoneze (substantive, verbe, etc).

În cadrul secțiunii elementelor șablon se găsesc alte 3 tabele interdependente. Pe scurt, ideea principală este: un formular (*FormTemplate*) conține mai multe întrebări (*QuestionTemplate*), care, la rândul lor, conțin mai multe răspunsuri (*AnswerTemplate*). Tabela *FormTemplate* conține attribute precum: temă, conținut, vizitat, și tip. Atributul *PartialViewId* va menține legătura către reprezentarea vizuală a formularului. Acest lucru se datorează faptului că un formular de gramatică are o structură complexă, cu imagini, tabele, schițe, dispuse dinamic în pagină, în funcție de formular; informații ce nu pot fi încadrate într-un simplu șir de caractere stocat în baza de date. Tipul *FormType*, ce descrie unul dintre attributele tabelului este descris în Figura 9. Tabela corespunzătoare unei întrebări, *QuestionTemplate*, conține informațiile șablon ale unei întrebări. Pe lângă conținut și cheie primară, observăm că această tabelă conține și o cheie străină – *FormTemplId*. Datorită acestui fapt putem crea relația de *one-to-many* (unu la mai multe) între tabelele corespunzătoare formularului și întrebărilor. Observăm că relația dintre aceste două tabele, din Figura 8, permite unui formular să nu conțină nicio întrebare (*zero-or-many*). Tabela *AnswerTemplate* reprezintă un răspuns și conține informații despre textul răspunsului, dacă este corect sau nu și desigur întrebarea căreia îi aparține, reprezentată prin cheia străină *QuestionTemplateId*. La fel ca în cazul precedent, avem relația de *one-to-many* (unu la mai multe) între tabelele *AnswerTemplate* și

QuestionTemplate. Totuși, trebuie menționat faptul că o întrebare nu poate avea zero răspunsuri, iar prin răspunsuri mă refer la entitatea *AnswerTemplate* ce poate fi un răspuns corect sau un răspuns greșit.

Din secțiunea șablonelor mai rămâne de explicat tabela *WordsInText*. După cum a fost explicat în capitolele precedente, aplicația trebuie să țină legătura de la elementele de gramatică și citire către elementele din vocabular. În alte cuvinte, trebuie să știm ce vocabular conține un formular, o întrebare sau un răspuns. Acest lucru nu poate fi obținut dinamic prin parsarea unui text din cauza structurii cuvintelor. De exemplu, textul 学生 poate reprezenta un cuvânt, dar poate reprezenta și două cuvinte (学 și 生). În tabela *WordsInText* se stochează legăturile dintre cuvintele vocabularului și textul entităților *QuestionTemplate*, *AnswerTemplate*, *FormTemplate*.

În diagrama din Figura 8 este reprezentată secțiunea din mijloc, ce conține trei entități. O entitatea a aceste secțiuni poate fi văzută ca un ambalaj peste un element șablon din secțiunea descrisă anterior. *VocabularyItem* are la bază un *VocabularyTemplate*, *QuestionItem* are la bază un *QuestionTemplate* și *FormItem* are la bază un *FormTemplate*. Pe lângă o cheie străină ce reprezintă elementul șablon, aceste entități conțin alte atribute referitoare la progresul utilizatorului. Aș vrea să detaliez puțin atributul *CurrentMiniLevel* din tabela *VocabularyTemplate* ce este de tipul *MiniLevels*. Acest tip este prezentat în Figura 9. Valorile enumerate reprezintă stadiile itemilor vocabularului prezentate în subcapitolele anterioare (*Seed1*, *Seed2*, *Leaf1*, *Leaf2*, *Bloom*, *Flourished*). În imagine am exemplificat prin câteva unități de timp distanța dintre aceste stadii. Aceste valori pot fi modificate dar important este să urmărească o creștere exponențială de la stadiul *Seed* către stadiul *Flourished*.

Între entitatea *User* și tabelele din secțiunea a doua sunt stabilite relații de tip una la mai multe (one-to-many) întrucât un utilizator are mai multe entități de tipul *VocabularyItem*, *QuestionItem* și *FormItem*. Pentru entitatea *AnswerTemplate* nu există o tabelă, în secțiunea a doua, pentru că nu este necesară înregistrarea progresului utilizatorului față de un răspuns al unei întrebări, întrucât nu are nicio utilitate practică.

1.4. Concluzii

Acest capitol este unul dintre cele mai mari și mai importante. Conținutul acestuia este complex și informativ. Aici se prezintă pentru prima dată aplicația, cu detalii arhitecturale și structurale, oferind o imagine clară asupra acesteia.

Proiectarea aplicației are un rol crucial în crearea acesteia. Fără un plan bine pus la punct ar putea apărea erori și complicații în procesul de implementare ce succedă proiectarea. Arhitectura unei aplicații software înglobează decizii referitoare la cerințe, performanță, organizarea sistemului, modul de comunicare a componentelor sistemului, riscurile ce pot apărea și multe altele.

În prim plan a fost stabilită structura și funcționalitățile aplicației. Au fost prezentați termeni specifici aplicației și a domeniului ei. A fost realizată divizarea în subcomponente funcționale împreună cu detalierea lor. După care, pentru a pune la punct tot ce oferă aplicația, au fost prezentate scenarii de utilizare.

În cele din urmă, pe baza cerințelor și a funcționalităților aplicației a fost creată structura bazei de date. Diagrama relațională pentru baza de date reprezintă un bun suport pentru o mai bună înțelegere a structurii dar și pentru implementarea acesteia.

2. Implementare

În implementarea celor propuse în capitolul anterior este esențială utilizarea tehnologiilor potrivite. Am ales să folosesc ASP.NET Core¹¹ (versiunea 2.0). ASP.Net Core reprezintă un framework open-source pentru crearea aplicațiilor. Principalele motive pentru care am ales această platformă sunt: caracterul open-source, mecanismul de dependency injection, suport pentru crearea aplicațiilor web.

ASP.NET Core MVC mi-a oferit posibilitatea de a dezvolta aplicația web conform șablonului arhitectural Model-View-Controller. Acest șablon permite respectarea principiului *separation of concern*, ce este un principiu cheie în modelarea aplicațiilor software. În linii generale, principiul *separation of concern* vizează distribuirea responsabilităților în cadrul unei aplicații. Voi reveni cu detalii în legătură cu noțiunile folosite aici în subcapitolele următoare.

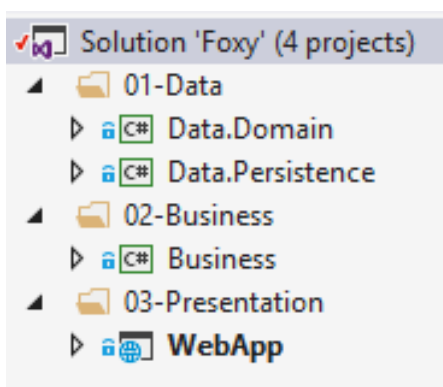


Figura 10: Structura soluției aplicației

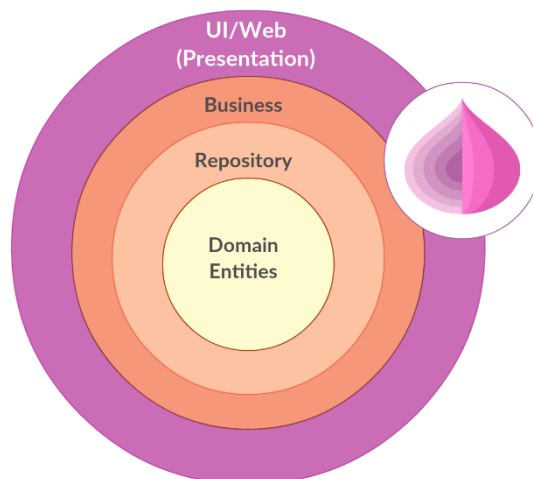


Figura 11: Nivelele arhitecturii Onion

2.1. Structura aplicației, arhitectura Onion și arhitectura web MVC

Soluția aplicației urmărește arhitectura *Onion*. Această arhitectură are la bază noțiunea de *layer* (strat). Datorită testabilității și mentenanței ea oferă un mod mai bun și mai eficient în crearea aplicațiilor. Principalul scop al acestei arhitecturi este de a satisface condițiile unei arhitecturi *n-tier*. O arhitectură *n-tier* se referă la o arhitectură client-server în care procesarea aplicației, managementul datelor și prezentarea aplicației sunt separate. Arhitectura Onion oferă

¹¹ ASP.NET Core – Mai multe detalii pot fi găsite pe site-ul oficial: <https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.0>

soluții pentru probleme des întâlnite cum ar fi cuplarea sau *separation of concern*. Se urmărește mereu o cuplare mică – loose coupling – ceea ce în alte cuvinte înseamnă că o clasă trebuie să fie cât mai puțin dependentă de alte clase.

În Figura 11 sunt reprezentate schematic straturile arhitecturii Onion. O mențiune importantă este că un strat nu va conține și nu va depinde de un strat exterior. În sens opus, un strat exterior folosește elemente ale straturilor interioare. După cum poate fi observat în Figura 10, soluția aplicației Foxy respectă straturile arhitecturii Onion. Singura abatere este faptul că stratul *Repository* este integrat în stratul *Business*. Acest lucru nu este o problemă întrucât repository-urile sunt injectate direct în proiectul *WebApplication*. În subcapitolele următoare vor fi prezentate detalii despre straturile arhitecturii Onion și maparea lor asupra proiectelor din cadrul soluției Foxy. [4]

Pentru a implementa arhitectura Onion am folosit MVC (Model-View-Controller). MVC este o arhitectură web ce soluționează probleme precum „separation of concern” datorită delimitării clare dintre interfață, logica business și entități.

View-ul este folosit pentru a crea interfața cu utilizatorul. *Model* este folosit pentru a transporta date între *View* și *Controller*. Un model poate fi de mai multe tipuri: model ce încapsulează datele logicii business, model pentru *View*, etc. *Controller*-ul este utilizat pentru a intercepta și a administra cererile web (web request) returnând un *View*. În esență, MVC rezolvă problema „separation of concern”, dar nu și a cuplajului (de acesta ocupându-se arhitectura Onion). În Figura 12 sunt reprezentate schematic componentele și relațiile dintre acestea în cadrul MVC. [5]

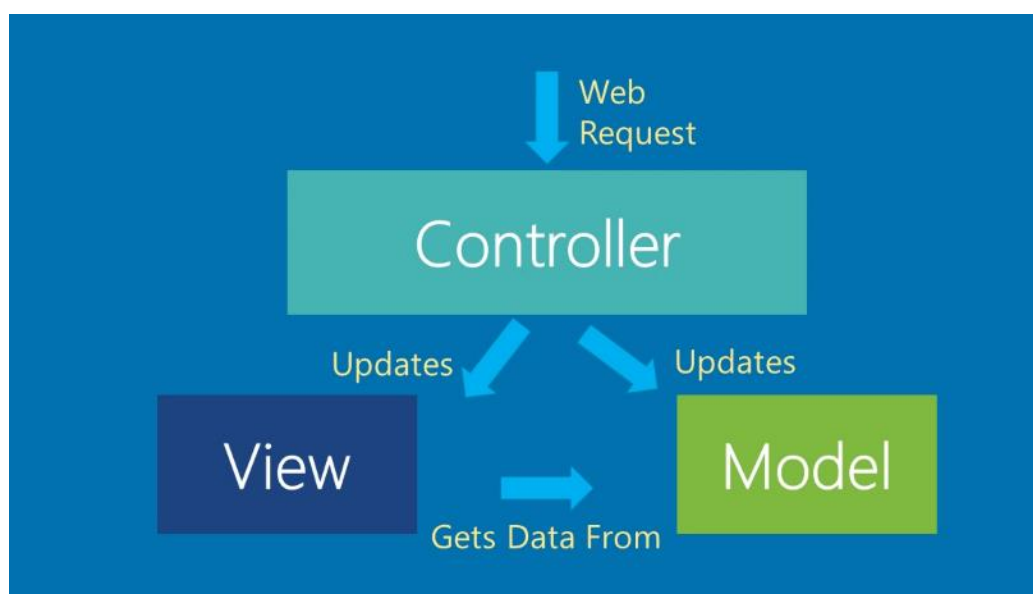


Figura 12: Schemă Model-View-Controller (Sursă: Uaic, FII, curs Introducere în .NET)

2.2. Entitățile domeniului, EF Core și SQL Server

Acest compartiment reprezintă partea centrală a arhitecturii Onion (vezi Figura 11). Datorită acestui fapt, proiectul *Data.Domain*, reprezentat în Figura 13, nu conține nicio referință către alte proiecte. Acest proiect conține trei fișiere. În aceeași ordine de idei, fișierul *Entities* este reprezentativ pentru stratul central al arhitecturii Onion.

Întrucât am ales strategia *Code First*¹², fișierul *Entities* conține clase POCO conform bazei de date prezentate în capitolul 1.3. Toate aceste clase sunt listate în Figura 14. Clasa *StaticInfo* reprezintă o excepție. Aceasta este o clasă statică, structura ei diferă și este folosită pentru a declara variabile globale pentru soluția aplicației dar și anumite tipuri de tip enumerare. Un exemplu este reprezentat în Tabelul 1.

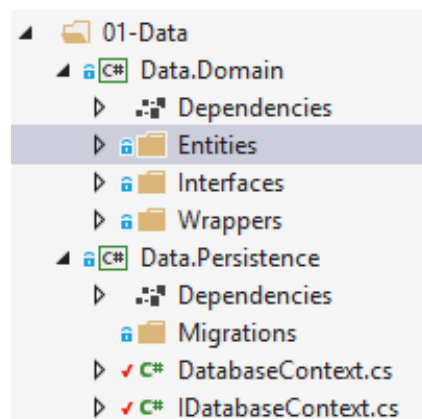


Figura 13: Structura proiectelor *Data.Domain* și *Data.Persistence*

```
public static List<int> minutesForLevel = new List<int>() {  
  
    0, //          lev0  
  
    3, //3 min     lev11  
  
    5, //5 min     lev12  
  
    10, //10min    lev21  
  
    15, //15min    lev22  
  
    20, //20min    lev3  
  
    0 // last level lev4  
  
};
```

Tabelul 1: Valorile pentru distanța de timp dintre stadiile itemilor de vocabular

Această listă reprezintă intervalele de tip între stadiile unui item al vocabularului. Observăm că valorile timpului sunt destul de mici. În momentul actual sunt setate în acest fel

¹² Code First – strategie prin care sunt scrise mai întâi clasele POCO (Plain old CLR objects) peste care se generează mai apoi baza de date

pentru a putea testa mai ușor aplicația și pentru a putea crea un demo fără să aștept luni de zile pentru a prezenta toate scenariile. Aceste valori pot fi ușor setate din acest loc, fără a fi nevoie de alte modificări. Valorile reprezintă numărul de minute.

Revenind la clasele POCO, rămâne de menționat că acelea urmăresc același șablon: au constructorul privat, proprietățile lor sunt publice pentru a fi citite dar private pentru a fi setate și au o metodă statică *Create*. Din cauza constructorului privat se forțează crearea unui obiect prin metoda *Create*. Aceste decizii au fost luate pentru a păstra încapsularea datelor dar și pentru a corespunde cu Entity Framework Core, despre care se va prezenta detalii în ceea ce urmează.

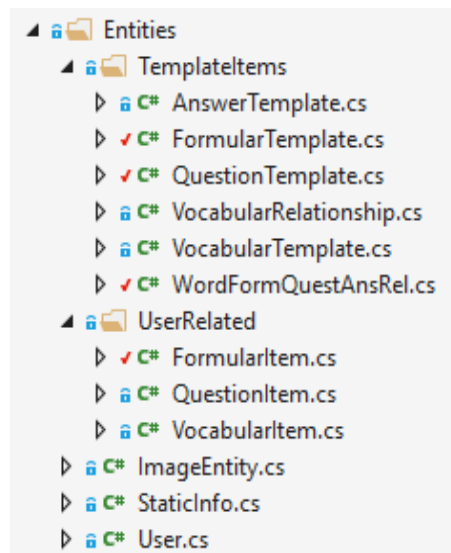


Figura 14: Clasele POCO din cadrul proiectului Data.Domain

EF Core este o platformă pentru accesarea datelor. Mai exact, EF mapează obiectele în așa fel încât developerii pot interacționa cu baza de date prin intermediul obiectelor .NET. Proiectul responsabil pentru reprezentarea EF este Data.Persistance, reprezentat în Figura 13. *DbContext* este clasa prin care se realizează accesul tabelelor din baza de date. EF folosește un set de convenții pentru accesarea tabelelor. Accesul unui tabel se realizează prin definirea unei entități care conține drept proprietăți attributele unui tabel din baza de date. Apoi, în clasa *DbContext* trebuie definită o proprietate prin care se va efectua accesul către datele din tabel. [6]

În Tabelul 2 sunt expuse proprietățile clasei *DbContext*. După cum poate fi observat, se respectă convenția denumirilor tabelelor: pentru a avea o listă de entități corespunzătoare unei tabele din baza de date, aceasta trebuie să respecte forma de plural a denumirii tabelului.

```
public DbSet<User> Users { get; set; }

public DbSet<ImageEntity> Images { get; set; }

public DbSet<VocabularTemplate> VocabularTemplates { get; set; }

public DbSet<VocabularItem> VocabularItems { get; set; }

public DbSet<VocabularRelationship> VocabularRelationships { get; set; }

public DbSet<FormTemplate> FormularTemplates { get; set; }

public DbSet<FormItem> FormularItems { get; set; }
```

```

public DbSet<QuestionTemplate> QuestionTemplates { get; set; }

public DbSet<QuestionItem> QuestionItems { get; set; }

public DbSet<AnswerTemplate> AnswerTemplates { get; set; }

public DbSet<WordsInText> WordFormQuestAnsRels { get; set; }

```

Tabelul 2: Proprietățile clasei DatabaseContext

EF Core poate fi configurat și prin intermediul FluentAPI. Acesta este cel mai puternic mod de a face anumite configurații. Configurațiile FluentAPI au cea mai mare prioritate, suprascriind convențiile și adnotările. Am folosit acest mod de configurare pentru a indica că o cheie primară este formată din două chei străine pentru clasele *VocabularRelationship* și *WordsInText*, după cum este prezentat în Tabelul 3.

```

protected override void OnModelCreating(ModelBuilder modelBuilder) {

    modelBuilder.Entity<VocabularRelationship>().HasKey (vocabularRelationship
=> new { vocabularRelationship.MainItemId, vocabularRelationship.ContainedItemId });

    modelBuilder.Entity<WordsInText>().HasKey(wordFormQuestAnsRel => new {
wordFormQuestAnsRel.MainElementId, wordFormQuestAnsRel.WordId });

}

```

Tabelul 3: Configurarea cheii primare cu FluentAPI

Alte detalii despre lucrul cu *DatabaseContext* vor fi prezentate în subcapitolul următor în contextul *Repository*.

În legătură cu baza de date mai rămâne de menționat faptul că am folosit *SQL Server Express LocalDB*. *LocalDB* este o versiune SQL Server Express Database Engine folosită pentru dezvoltarea aplicațiilor. Un avantaj este modul de configurare a bazei de date pentru că nu prezintă mari dificultăți. În fișierul *Startup.cs*, ce reprezintă punctul de pornire a aplicației, din proiectul *WebApplication* (vezi Figura 10), este injectat contextul bazei de date. Secvența de cod corespunzătoare acestei acțiuni este reprezentată în Tabelul 4.

```

services.AddDbContext<DatabaseContext>(options=>
options.UseSqlServer(Configuration.GetConnectionString("FoxyConnection")));

```

Tabelul 4: Configurare SqlServer

Serviul configurărilor ASP.NET Core citește un *ConnectionString* (șir de caractere ce reprezintă conexiunea cu baza de date). Acesta este citit din fișierul *appsettings.json* care se află, la fel, în proiectul *WebApplication*. Secvența de cod este prezentată în Tabelul 5.

```
"ConnectionStrings": {  
  
    "FoxyConnection": "Server = .\\SQLEXPRESS; Database = Foxy.Development;  
Trusted_Connection = true; MultipleActiveResultSets=True;"  
  
}
```

Tabelul 5: Setarea șirului de conexiune în fișierul *appsettings.json*

Conform structurii din Figura 13 mai rămâne de menționat rolul directoarelor *Interfaces* și *Wrappers*. În directorul *Wrappers* am plasat clase ce înglobează informații din entitățile POCO, folosite în baza de date. În Figura 15 sunt prezentate toate clasele din directorul respectiv. Aceste clase *wrapper* (ambalaj) sunt folosite în proiectele corespunzătoare straturilor exterioare pentru a expune mai ușor informația, dar și pentru a abstractiza entitățile. După cum poate fi observat și în imagine, acestea au denumiri sugestive. De exemplu, *LevelWrapper* reprezintă o clasă ce înglobează colecții de elemente ale vocabularului corespunzătoare unui anumit nivel. Mai mult, această clasă conține și o funcție pentru a sorta aceste elemente în funcție de tipul lor. Astfel, obțin un mecanism util de separare a responsabilităților, scutindu-mă de o sarcină în plus acolo unde logica codului se focusează pe aspecte mai importante.

Directorul *Interfaces*, prezentat în Figura 16, conține interfețele corespunzătoare claselor repository. Aceste clase sunt componente ale șablonului *Repository Pattern*.

Principalul avantaj al acestui șablon este utilizarea entităților fără a fi nevoie de a ști cum sunt stocate acestea și care sunt legăturile dintre ele. Toată informația persistentă, inclusiv maparea dintre tabele și obiecte, este stocată într-un repository specific. Alte avantaje ale acestui șablon sunt: existența unui singur loc în care pot fi efectuate modificări în legătură cu schimbările apărute în cadrul entităților, existența unui singur loc responsabil pentru un anumit set de tabele.

După cum a fost specificat în subcapitolul 2.1, în cadrul arhitecturii Onion, partea responsabilă de repository se află în cadrul componentei Business. Așa deci, aparte întrebarea

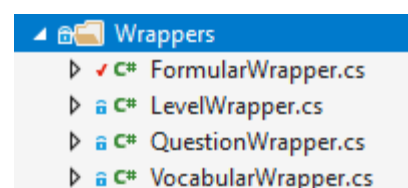


Figura 15: Clasele *Wrappers* din cadrul proiectului *Data.Domain*

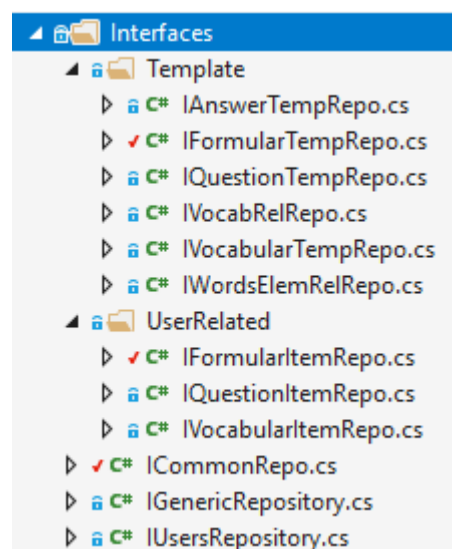


Figura 16: Interfețele Repository din cadrul proiectului *Data.Domain*

„De ce interfețele repository se află în compartimentul domeniului de date?”. Acest lucru este cauzat de necesitatea stabilirii unui contract pentru partea de business logic. Datorită acestei organizări, componenta responsabilă de logica business poate fi modificată sau schimbată în întregime, respectând totuși contractul stabilit de interfețele repository aflate în cadrul componentei domeniului de date. Astfel obținem versatilitate, o formă de polimorfism și extensibilitate. Mai multe detalii despre fiecare repository vor fi prezentate în subcapitolul următor.

2.3. Logica business

Stratul logicii business reprezintă o punte între domeniul de date al entităților și interfața Web. Layer-ul business conține logica asupra unei entități specifice aplicației. Datorită componentelor repository ce fac parte din acest domeniu, layer-ul business oferă o abstractizare asupra entităților aflate în domeniul datelor (prezentat în subcapitolul anterior). Repository pattern oferă o abordare ce conține cuplaj redus în accesarea datelor.

În Figura 17 este prezentată structura proiectului *Business*. Observăm că acesta conține o referință către proiectul *Data.Persistence*, fapt datorat necesității de a implementa interfețele repository, dar și de a avea acces către contextul bazei de date și entitățile acesteia.

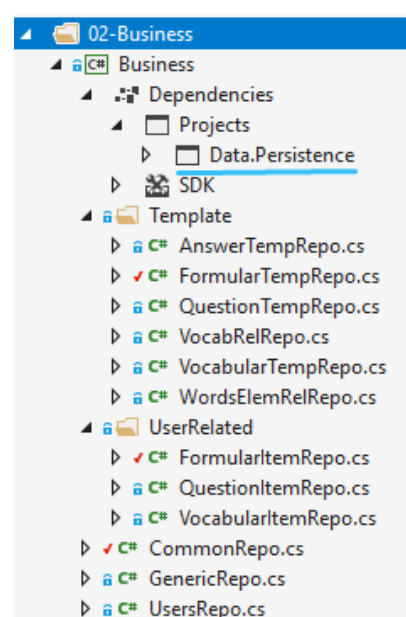


Figura 17: Structura proiectului Business

Pentru a respecta principiul DRY (Don't repeat yourself) am folosit șablonul *Generic Repository*. Principiul DRY este un principiu în crearea aplicațiilor software care urmărește reducerea duplicării/repetării informației de orice fel, util în special în arhitecturile multi-tier. *GenericRepo.cs* reprezintă o clasă abstractă ce implementează interfața *IGenericRepo*. Restul claselor de tip repository derivă din *GenericRepo*. Acest repository generic, după cum sugerează și denumirea sa, implementează niște metode de bază ce reprezintă acțiuni elementare asupra entităților bazei de date. Pentru a înțelege mai bine cum funcționează acest repository generic am prezentat o secvență din clasa *GenericRepo* în Tabelul 6.

```
public abstract class GenericRepo<T> : IGenericRepository<T> where T : class{

    private readonly IDatabaseContext _databaseContext;
```

```

private readonly DbSet<T> _entities;

protected GenericRepo(IDatabaseContext databaseContext) {

    _databaseContext = databaseContext;

    _entities = (databaseContext as DbContext)?.Set<T>();

}

/* Declararea metodelor elementare și implementare lor*/

}

```

Tabelul 6: Secvență de cod din clasa GenericRepo

După cum poate fi observat, DatabaseContext este injectat în constructorul clasei. Pe lângă asta, contează foarte mult faptul că GenericRepository folosește o entitate abstractă - <T> - ce va fi specificată atunci când se implementează această clasă. De exemplu, în secvența următoare această entitate este *User*:

```

public class UsersRepo : GenericRepo<User>, IUsersRepository

```

Un avantaj este faptul că metodele declarate și implementate în repository-ul generic pot fi suprascrise datorită faptului că au tipul virtual. Un exemplu de acest fel, în cadrul acestui proiect, sunt metodele *GetAll* și *FindById*, ce au fost suprascrise în clasele *QuestionTempRepo* și *FormularTempRepo*. A fost necesară suprascrierea lor din cauza relației one-to-many. Modul de încărcare implicit EF este *lazy loading*. Pentru relațiile one-to-many este necesar folosirea încărcării de tip *eager loading*. Acest mod de încărcare prevede încărcarea datelor atunci când se efectuează interogarea inițială, spre deosebire de *lazy loading*, care încarcă datele din baza de date atunci când proprietatea de navigare este accesată. În cazul de față, dacă nu aș fi folosit *eager loading*, colecția de *QuestionTemplates* din clasa *FormularTemplate* nu s-ar fi încărcat la accesarea unei entități de acest tip, prin urmare, ar fi fost nevoie de alte acțiuni pentru a încărca la rândul lor și obiectele din colecția respectivă. [7]

```

public override async Task<IEnumerable<FormTemplate>> GetAll() {

return          await          _databaseContext.FormularTemplates.Include(ft          =>
ft.QuestionTemplates).ThenInclude(at => at.AnswerTemplates).ToListAsync();

}

```

Tabelul 7: Încărcarea datelor prin eager loading

În codul prezentat în Tabelul 10Tabelul 7, *eager loading* se obține prin „Include” și „ThenInclude”. Observăm că în cod este prezentă o expresie *lambda* (=>). O expresie labda

este o funcție anonimă ce poate fi folosită pentru a crea un delegate¹³. Folosind *expresiile lambda* pot fi scrise funcții locale ce pot fi transmise ca argumente sau pot fi returnate drept rezultatul unui apel al unei funcții. Aceste expresii sunt în special utile în a scrie expresiile interogare LINQ. [8]

LINQ provine de la acronimul Language Integrated Query și reprezintă un set de tehnologii bazate pe integrarea capacităților interogarilor (query) direct în limbajul C#. Expresiile Query din cadrul LINQ sunt folosite pentru a interoga și a transforma diferite surse de date. De exemplu, o expresie de acest tip poate regăsi datele dintr-o bază de date producând un răspuns în formatul XML. Pe lângă asta, expresiile Query sunt ușor de utilizat întrucât utilizează multe elemente specifice modului de construcție a limbajului C#. [9]

Revenind la clasele repository, prezentate în Figura 17, putem afirma că acestea reprezintă întreaga logică din spatele entităților și cuplarea acestora în formarea unităților funcționale. Fiecare repository se ocupă în general de managementul entităților corespunzătoare. Entitățile specifice fiecărui repository sunt ușor de dedus din numele claselor. Totuși, anumite repository-uri pot fi injectate în altele, din cauza dependențelor dintre entități. De exemplu, după cum a fost specificat și în subcapitolul 1.3, entitatea *VocabularyItem* reprezintă un ambalaj pentru entitatea *VocabularyTemplate*, prin urmare, repository-ul *VocabularyItemRepo* trebuie să conțină un *VocabularyTempRepo*. Astfel, *VocabularyItemRepo* conține toate funcționalitățile pentru partea de vocabular. În așa mod, pentru a avea acces către toate funcționalitățile vocabularului, este suficientă folosirea acestui repository.

În cele ce urmează voi prezenta fluxul implementării unui scenariu în cadrul clasei *VocabularyItemRepo*. De exemplu, ce funcții sunt apelate atunci când se dorește extragerea elementelor active pentru sesiunea de evaluare. Metoda responsabilă de această funcționalitate este *GetVocabForReview* și trebuie să primească un parametru de tip Guid ce reprezintă id-ul utilizatorului. Guid este un tip de date ce reprezintă un identificator unic. Funcția *GetVocabForReview*, nu face altceva decât să itereze prin toate entitățile *VocabularyItem* ce aparțin utilizatorului și să apeleze funcția *ActiveForReview*, prezentată în Tabelul 8.

```
public bool ActiveForReview(VocabularyItem item){

    if (MiniIsGrand(item.CurrentMiniLevel, GrandLevels.Flourished))
        return false;

    if (MiniIsGrand(item.CurrentMiniLevel, GrandLevels.Lesson))
        return false;
```

¹³ Un delegate reprezintă un tip ce încapsulează o metodă, similar unui pointer la o funcție în C/C++.

```

        if (item.LastAnswer == false) return true;

        DateTime readyTime = item.LastTimeAnswered.AddMinutes
            (StaticInfo.minutesForLevel[(int)item.CurrentMiniLevel]);

        if (DateTime.Compare(readyTime, DateTime.Now) <= 0)
            return true;

        return false;
    }

```

Tabelul 8: Codul funcției ActiveForReview

Această funcție verifică mai întâi dacă un item nu se află în stadiu Flourished sau Lesson, pentru că în aceste cazuri el nu se poate regăsi în sesiunea de examinare. O altă condiție pentru care un element ar fi activ pentru evaluare este dacă ultimul răspuns a fost greșit (vezi subcapitolul 1.1.2). După aceste verificări, funcția calculează timpul la care elementul este activ: se adună la data în care a fost acordat ultimul răspuns numărul de minute conform stadiului curent. În final, dacă timpul actual (DateTime.Now) este mai mic, adică se află în trecut față de timpul la care elementul devine activ (readyTime), funcția returnează valoarea de adevăr. Acest repository conține multe alte metode specifice anumitor funcționalități: adăugarea itemilor pentru un nou utilizator, identificarea itemilor unui utilizator, deblocarea itemilor, trecerea la următorul nivel, identificarea itemilor activi pentru învățare, și multe altele.

În continuare voi prezenta câteva informații în legătură cu relațiile one-to-many dintre repository-urile *FormTemplate*, *QuestionTemplate* și *AnswerTemplate*. Mai întâi, e important să știm cum funcționează și cum arată metoda simplă de adăugare a unei entități în baza de date din *GenericRepo*. Conținutul acestei funcții este prezentat în Tabelul 9.

```

public virtual async Task Add(T entity) {

    await _entities.AddAsync(entity);

    await Save();

}

```

Tabelul 9: Funcția de adăugare a unei entități în baza de date

Proprietatea *_entities* are tipul *DbSet<T>*, ce reprezintă intrările unei table cu entități de tip *T* (*T* poate fi oricare dintre clasele POCO prezentate în proiectul *Data.Domain*). Funcția *AddAsync* adaugă asincron entitatea în tabela respectivă, iar modificările vor fi salvate doar după apelarea funcției *Save*. Atunci când lucrăm cu o relație one-to-many, EF permite salvarea

legăturilor dintre date doar dacă mai întâi au fost adăugate toate elementele (de exemplu prin *AddAsync*), funcția *Save* fiind apelată doar la final. Prin urmare metoda *Add* din *GenericRepo* nu va putea crea corect legăturile dintre claselor *FormTemplate*, *QuestionTemplate* și *AnswerTemplate*. Pentru a rezolva această problemă, am creat clasa repository *CommonRepo*. Această clasă nu implementează *GenericRepo* și conține o funcție, prezentată în Tabelul 10, pentru salvarea unei entități de tip *FormTemplate*. După cum poate fi observat, sunt adăugate pe rând toate entitățile conținute de un formular, după care se salvează modificările prin *SaveChangesAsync*.

```
public async Task SaveFormular(FormTemplate formular) {  
  
    foreach(var quest in formular.QuestionTemplates) {  
  
        foreach(var ans in quest.AnswerTemplates)  
  
            await _dbContext.AnswerTemplates.AddAsync(ans);  
  
        await _dbContext.QuestionTemplates.AddAsync(quest);  
  
    }  
  
    await _dbContext.FormularTemplates.AddAsync(formular);  
  
    await ((DbContext)_dbContext).SaveChangesAsync();  
  
}
```

Tabelul 10: Funcție pentru salvarea unui formular în baza de date

În concluzie, acest proiect conține mult prea multe metode pentru a putea fi explicate toate în detaliu. Totuși, este important să prezentăm aspectele principale și unele cazuri de excepție, pentru ca cititorul să-și poată forma o impresie generală asupra structurii acestui proiect. Clasele proiectului vor fi injectate în controller-ele proiectului ce urmează să fie descris în continuare.

2.4. Nivelul prezentare

Nivelul prezentare se referă la nivelul exterior din arhitectura Onion, ultimul nivel (vezi Figura 11). Proiectul *WebApplication* reprezintă punctul de start al aplicației, implementând arhitectura MVC.

2.4.1. Punctul de pornire al aplicației

În Figura 18 este prezentat structura proiectului *WebApplication*. Fișierul *Startup.cs* reprezintă punctul de start al proiectului. În această clasă sunt asamblate middleware-uri și sunt configurate servicii. Un middleware este, în linii generale, un pipeline (conductă) pentru a trata cereri și răspunsuri. Fiecare componentă de acest fel decide dacă transmite cererea către următoarea componentă din pipeline. Aceasta poate acționa înainte și după ce următoarea componentă este invocată. Middleware-urile sunt configurate folosind metodele: *Run*, *Map* și *Use*. Ordinea în care middleware-urile sunt adăugate în metoda *Configure* este esențială pentru securitate, performanță și funcționalitate. [10] Secvența de cod din Tabelul 11 reprezintă adăugarea middleware-ului MVC în care ruta implicită este „Home/Index”.

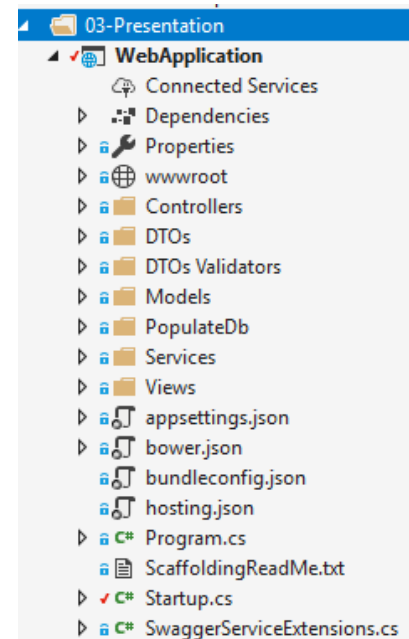


Figura 18: Structura proiectului *WebApplication*

```
app.UseMvc(routes => {  
  
    routes.MapRoute(  
  
        name: "default",  
  
        template: "{controller=Home}/{action=Index}/");  
});
```

Tabelul 11: Fragment de cod responsabil pentru adăugarea middleware-ului MVC

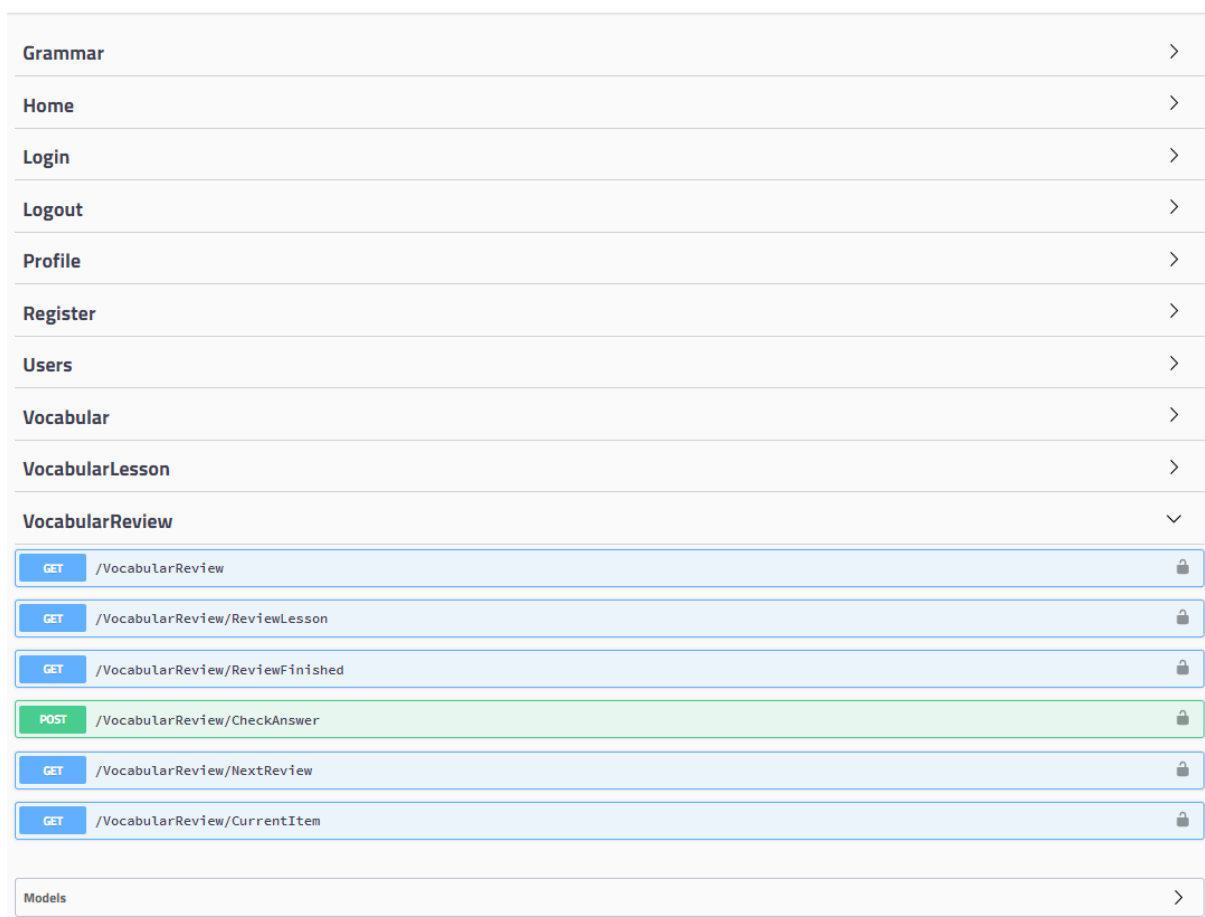
În metoda *ConfigureService* din *Startup.cs* pot fi adăugate servicii datorită mecanismului integrat de *dependency injection*. Pentru a adăuga un serviciu se folosește metoda *Add[Service]*. Pot fi adăugate servicii personalizate prin intermediul metodei *AddTransient*. Această metodă este folosită pentru a mapa tipuri de date abstracte la servicii/tipuri concrete, fiind instanțiate separat pentru fiecare obiect ce solicită acest serviciu. [11]

```
services.AddTransient<IUserRepo, UserRepo>();
```

Aceast fragment de cod exemplifică cele menționate mai sus. În urma acestei mapări, atunci când un constructor necesită un parametru de tipul *IUserRepo*, o instanță de tipul *UserRepo* îi va fi injectată.

2.4.2. Swagger

În procesul creării unei aplicații înțelegerea și verificarea tuturor funcționalităților reprezintă o provocare. Pentru a soluționa această problemă, fără a depinde de o interfață web finalizată, am folosit Swagger. Swagger, numit și Open API, generează o documentație interactivă utilă în explorarea tuturor funcționalităților aplicației. În cadrul proiectului, în faza inițială, am folosit *Swashbuckle.AspNetCore*, ce este un proiect open source pentru generarea documentației Swagger a aplicațiilor web ASP.NET Core. [12]



The image shows a Swagger UI interface with a sidebar on the left containing a list of API endpoints. The 'VocabularyReview' endpoint is expanded, showing a list of specific API methods. The methods are as follows:

Method	Endpoint	Visibility
GET	/VocabularyReview	Locked
GET	/VocabularyReview/ReviewLesson	Locked
GET	/VocabularyReview/ReviewFinished	Locked
POST	/VocabularyReview/CheckAnswer	Locked
GET	/VocabularyReview/NextReview	Locked
GET	/VocabularyReview/CurrentItem	Locked

At the bottom of the sidebar, there is a 'Models' section with a right-pointing arrow.

Figura 19: Documentație generată cu Swagger

În Figura 19 este reprezentată documentația interactivă generată de Swagger. Sunt listate în prim plan numele controller-elor. Secțiunile specifice fiecărui controller pot fi extinse pentru a vedea metodele conținute, după cum este prezentat în exemplu pentru

VocabuarReview. Putem vedea tipul verbul HTTP corespunzător metodei. Interfața Swagger pune la dispoziție, pentru orice metodă, tipul modelelor primite ca parametri împreună cu atributele ce trebuie completate. Prin urmare, pe lângă utilitate, Swagger oferă o experiență foarte plăcută pentru developer.

2.4.3. Model-View-Controller

După cum am specificat la începutul capitolului, proiectul WebApplication folosește arhitectura Model-View-Controller. În acest proiect locul în care se regăsesc controller-ele este directorul *Controllers* (vezi Figura 18). Controller-ele sunt clase ce tratează cererile navigatorului web. Acestea modifică modele și returnează obiecte View drept răspuns. Fiecare metodă publică din controller reprezintă un endpoint¹⁴ HTTP.

MVC invocă controller-ul corespunzător în dependență de URL¹⁵-ul primit, utilizând middleware-ul pentru rutare. Șablonul implicit folosit este:

```
/[Controller]/[ActionName]/[Parameters]
```

Controller reprezintă numele clasei controller, *ActionName* reprezintă metoda din cadrul controller-ului, iar *Parameters* reprezintă parametrii metodei. În cazul în care URL-ul indică doar numele controller-ului, metoda implicită este *Index*. Convenția de nume a claselor trebuie să conțină șablonul „[nume]Controller” (de exemplu „VocabularController”). Clasele controller trebuie să conțină adnotarea *[Route("[controller"])]*.

```
[HttpGet]
[Route("word/{name}")]
public async Task<IActionResult> Word([FromRoute]string name)
```

Tabelul 12: Antetul metodei unui controller

În Tabelul 12 este prezentat un fragment de cod ce reprezintă antetul unei metode împreună cu adnotările sale. Prima adnotare reprezintă verbul HTTP. A doua adnotare indică faptul că URL-ul va conține un parametru „name”.

Fiecare clasă controller va avea injectat unul sau mai multe repository-uri (din proiectul Business). Prin intermediul acestor repository-uri, controller-ul va crea și va face modificări asupra unor obiecte Model pentru a fi transmise unui View.

¹⁴ Endpoint – un capăt al canalului de comunicare.

¹⁵ URL – Uniform Resource Locator. Este o secvență de caractere standardizată, folosită pentru denumirea, localizarea și identificarea unor resurse de pe Internet.

Există mai multe tipuri de modele. De exemplu, directorul *DTOs*, din cadrul proiectului, se referă la modele de tip Data Transfer Object. În alte cuvinte, aceste modele reprezintă clase predefinite ce sunt transmise controller-ului. De exemplu, modelul *RegisterDto* conține toate informațiile necesare înregistrării unui utilizator precum nume, email, parolă. În linii generale, acest model se formează din informațiile introduse în navigatorul web și este transmis controller-ului responsabil pentru înregistrarea utilizatorului. Pentru aceste modele am creat anumite filtre cu scopul de a impune restricții cum ar fi lungimea, respectiv conținutul parolei. Aceste filtre sunt niște clase, localizate în directorul „*DTOs Validators*”, create cu ajutorul librării open source *FluentValidation*. Prin urmare, validarea modelului nu mai este responsabilitatea controller-ului. Serviciul *FluentValidation* este injectat în fișierul *Startup.cs*.

ViewModel este un alt tip de model în MVC. Aceste modele sunt niște clase, care nu conțin logică complexă. Sunt folosite pentru a fi pasate View-urilor, conținând informația necesară pentru a fi expusă în navigatorul web. Acest tip de modele se află în directorul *Models* (vezi Figura 18).

Un alt compartiment vast al acestui proiect este determinat de către componenta View. Un view poate fi considerat un fișier de tip *.cshtml* ce conține cod HTML mixat cu fragmente de cod *C#*, numit *Razor syntax*. Aceste componente se află în directorul *Views*. După cum poate fi observat în Figura 20, directorul *Views* conține câte un director pentru fiecare controller. Este foarte important să fie respectată convenția de nume, astfel încât controller-ul să poată identifica view-ul respectiv. Un director mai special este *Shared*. Aici sunt conținute fișiere ce pot fi accesate din orice controller. Un fișier important este *_Layout.cshtml*. Acesta reprezintă codul *cshtml* pentru elemente comune ale view-urilor. Desigur, pot fi create și layout-uri personalizate pentru anumite view-uri. Un view poate să nu conțină niciun layout. În Tabelul 13 este prezentat modul în care se setează un layout într-un View, în acest caz se setează ca View-ul să nu conțină niciun layout. În cadrul fișierului, simbolul „*@*” marchează începutul unei secvențe de cod *C#*.

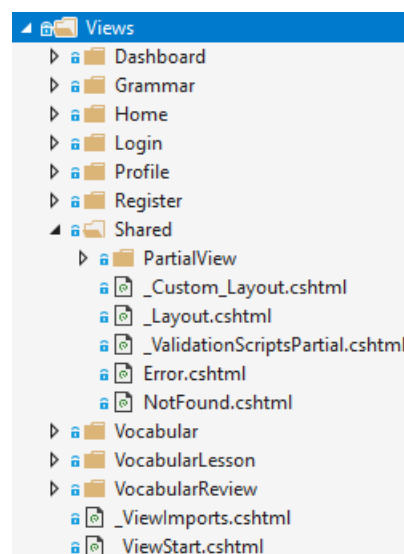


Figura 20: Structura directorului Views

```
@{
    Layout = null;
}
```

Tabelul 13: Setarea layout-ului unui View

Directorul *wwwroot*, prezentat în Figura 21, reprezintă un director static. Fișierele conținute de acesta sunt folosite de către View-uri. Aici se regăsesc fișiere: css, js, png, etc. Pentru a folosi un fișier din acest director, în secțiunea *head* a documentului html, trebuie inclusă o cale relativă către acesta.

Un view poate primi un model prin secvența de cod razor „@model NumeModel”. Fișierele de tip cshtml sunt ușor de utilizat. De exemplu, pentru a crea o listă cu elemente putem folosi *for* sau *foreach*, exact ca în C#.

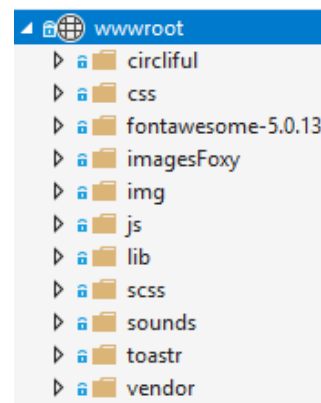


Figura 21: Structura directorului *wwwroot*

```
@for (int i = 0; i < Model.MeaningsList.Count(); ++i) {

    <span class="badge badge-info meaning-badge">

        @Model.MeaningsList[i]

    </span>

}
```

Tabelul 14: Secvență de cod Razor pentru afișarea traducerilor unui item

În Tabelul 14 este exemplificat cum se afișează toate înțelesurile unui cuvânt. Este foarte simplu datorită mixării codului c# și a elementelor html. Mai multe detalii despre implementarea unui view, deciziile de design, tehnicile și librăriile folosite vor fi prezentate în capitolul următor.

În proiectul *WebAppllication* mai rămân de analizat directoarele *Services* și *PopulateDb*. În directorul *Services* se găsește clasa *MainService* și interfața acesteia. Această clasă conține funcționalități necesare sesiunii de examinare. Aici este stocată starea fiecărui utilizator în legătură cu sesiunea de examinare. La începutul sesiunii, clasa primește colecția cu elementele active pentru evaluare. Clasa receptează răspunsurile utilizatorului, prin intermediul controller-ului, decide dacă acestea sunt corecte sau greșite și returnează un răspuns. Această clasă este responsabilă și pentru extragerea aleatorie a următoarului item pentru evaluare din colecția de elemente prezentată inițial. Clasa *MainService* este injectată în View-ul responsabil pentru sesiunea de examinare.

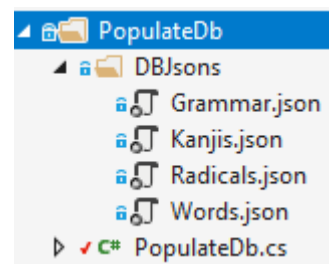


Figura 22: Directorul care conține informații despre popularea bazei de date

În Figura 22 este prezentat directorul care se ocupă cu popularea bazei de date. Mai exact, aici sunt salvate fișiere în format json și o clasă *PopulateDb* care se ocupă cu citirea acestor fișiere și popularea bazei de date prin intermediul repository-urilor.

```
{
  "name": "二",
  "meaning": "two",
  "reading": "o=二||k=ふた",
  "type": "1",
  "required_level": "1",
  "meaning_mnemonic": "The kanji is same as radical.",
  "reading_mnemonic": "Well, think about it.. you have TWO knee (に).",
  "word_type": "",
  "components": [ "二" ]
},
```

Figura 23: Secvență json pentru reprezentarea unui element Kanji

În Figura 22 este reprezentat un fragment din fișierul *Kanjis.json*. Acest fișier conține un json format dintr-o listă de elemente kanji. După cum poate fi observat și în figură, elementele *meaning* și *reading*, corespunzătoare traducerii și citirii, au un format special pentru reprezentarea unei liste într-un șir de caractere. De exemplu, pentru a avea mai multe traduceri acestea sunt despărțite prin delimitatorul „;”. În cazul citirii formatul este un pic mai complicat. Putem avea una sau două părți, separate prin delimitatorul „||”. Aceste părți trebuie să înceapă cu „o= ” sau „k= ”, urmate de caractere hiragana sau katakana despărțite la fel prin „;”. Formatul acesta este necesar pentru stocarea citirii onyomi¹⁶ și kunyomi¹⁷. Descifrarea acestor șiruri de caractere și plasarea lor în obiecte de tip listă este realizată automat de către clasa *VocabularyWrapper* aflată în proiectul *Data.Domain*. Ulterior, elementele de tip json sunt citite și prelucrate de către clasa *PopulateDb*, care le adaugă apoi în baza de date prin intermediul claselor repository.

2.4.4. Sursele de informație pentru popularea bazei de date

Pentru a popula baza de date am folosit diverse surse de informații. Pentru a crea fișierele *json* ce conțin elementele de vocabular am folosit site-ul oficial JLPT. De aici au fost selectate denumirile, înțelesurile și citirile elementelor [13]. Pe lângă câteva mnemonici create de mine, am selectat altele din aplicațiile disponibile de învățare a limbii japoneze Tofugu [14]

¹⁶ On'yomi (音読み) – citire derivată din pronunția chineză.

¹⁷ Kun'yomi (訓読み) – citirea de origine japoneză.

și Wanikani [15]. Pentru a oferi utilizatorului posibilitatea de a audia anumite citiri, am folosit site-ul SoundOfText ce transformă un text în format audio. Fișierele specifice cititorilor sunt salvate în directorul static *wwwroot*. [16] Imaginile folosite pentru ilustrarea formularelor de gramatică în secțiunea grilă au fost preluate de pe site-ul Wasabi. [17]

2.4.5. Autentificarea bazată pe cookie-uri

Un cookie este un text special, deseori codificat, trimis de un server unui navigator web și apoi trimis înapoi (nemodificat) de către navigator, de fiecare dată când accesează acel server. Folosirea autentificării bazate pe cookie-uri este o alegere potrivită pentru arhitectura MVC. Autentificarea bazată pe cookie, numită și autentificare bazată pe sesiuni, este setată din fișierul *Startup* conform secvenței de cod prezentată în Tabelul 15. După cum poate fi observat, un cookie poate conține mai multe configurări. De exemplu, în secvență sunt configurate următoarele setări: calea la care se face redirecționarea atunci când cererea nu este autorizată, calea implicită pentru autentificare, denumirea cookie-ului și timpul în care expiră un cookie de la crearea acestuia.

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>{

        options.AccessDeniedPath = new PathString("/Login");

        options.LoginPath = new PathString("/Login");

        options.CookieName = "FoxyCookie";

        options.ExpireTimeSpan = new TimeSpan(2, 0, 0);

    });
```

Tabelul 15: Setare autentificare bazată pe cookies

Pentru a autoriza un controller sau o metodă trebuie folosită adnotarea *[Authorize]*. În momentul autentificării, controller-ul creează un cookie ce conține o listă de credențiale precum emailul și numele utilizatorului. Acest cookie este setat prin intermediul clasei predefinite *HttpContext*. În cadrul unei metode autorizate informațiile pot fi extrase în felul următor:

```
string email = HttpContext.User.Claims.First().Value;
```

Această secvență de cod extrage email-ul utilizatorului din primul element al listei de credențiale (claims). În mod implicit, autentificarea bazată pe sesiuni în ASP.NET Core este securizată.

2.5. Concluzii

Scopul acestui capitol este de prezenta implementarea proiectului. Aceasta are la bază proiectarea și arhitectura aplicației, prezentată în capitolul precedent. Au fost descrise tehnici, arhitecturi, șabloane, *best practices*.

S-a urmărit prezentarea generală a proiectelor soluției Foxy. Fiind imposibil de descris fiecare metodă și funcționalitate, din cauza spațiului limitat, au fost menționate, totuși, reperele principale, exemplificate prin fragmente de cod pentru o mai bună înțelegere.

Au fost explicate noțiuni legate de șabloane arhitecturale și detalii despre autentificare. Au fost descrise elementele aplicației ce se pliază peste aceste arhitecturi și motivele în luarea deciziilor de a utiliza aceste șabloane.

3. Interfața web, elemente design și interacțiunea cu utilizatorul

Pentru o aplicație web este foarte important modul în care sunt expuse vizual elementele dar și interacțiunea cu utilizatorul. Principalul suport în crearea unui design *responsive*¹⁸ a fost Bootstrap. Acesta este un set de instrumente open-source care integrează html, css și js.

O experiență plăcută pentru utilizator este unul dintre cele mai importante scopuri. Din acest motiv am folosit diverse librării:

- Toastr [18]

Este o librărie Javascript pentru a afișa notificări. Un toast este o notificare ce apare pentru o scurtă perioadă de timp. În Figura 24 este prezentat un exemplu toast de tipul success. Un toast poate avea și alte tipuri: eroare, informare, avertizare.

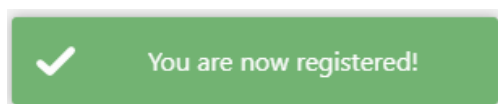


Figura 24: Exemplu toast

- Circliful [19]

Este o librărie JQuery folosită pentru a afișa statistici sub formă de cercuri. Am ales această librărie cu scopul de a oferi utilizatorului o experiență vizuală expresivă pentru a arăta procentajul de învățare a itemilor din vocabular. De asemenea, diagramele Circliful sunt complet responsive și permit configurarea multor opțiuni.

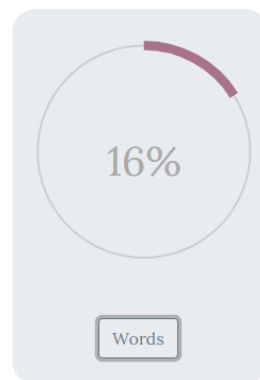


Figura 25: Diagramă
Circliful

- FontAwesome [20]

Este un set de instrumente ce pune la dispoziție o colecție vastă de pictograme. Folosirea pictogramelor potrivite în locul potrivit este foarte important pentru utilizator, exprimând mai bine scopul și funcționalitatea unei componente de pe ecran.

- Wanakana [21]

¹⁸ Design responsive – proprietatea interfeței web de a se adapta dimensiunii ecranului oricărui dispozitiv.

Este o librărie javascript ce permite transformarea unui text în caractere *hiragana* și *katakana*.

Pentru a crea sesiunea de examinare am folosit AJAX. Denumirea vine de la Asynchronous JavaScript And XML și se referă la o tehnică de încărcare a elementelor paginii în mod asincron. Acest lucru este foarte important pentru sesiunea de examinare. Este mult mai rapid și estetic încărcarea elementelor dinamic. Reîncărcarea paginii pentru fiecare element din cadrul sesiunii de examinare ar putea confuza utilizatorul, dar ar putea dura și mai mult timp.

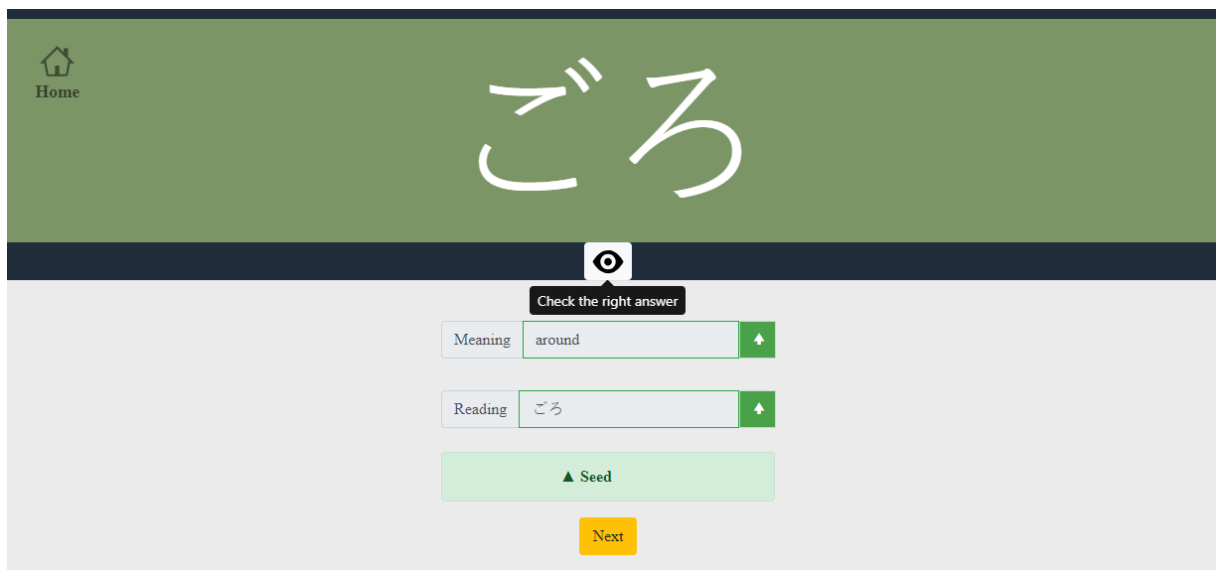


Figura 26: Captură de ecran a sesiunii de examinare

Figura 26 reprezintă un fragment din sesiunea de examinare. Toate elementele acestei pagini se încarcă dinamic. Utilizatorului îi sunt oferite informații în legătură cu stadiul itemului și corectitudinea răspunsurilor. Tot în cadrul acestei sesiuni, dar nu doar aici, am folosit *PartialView*. Acesta reprezintă o variantă în miniatură a unui View clasic ce poate fi încărcat oriunde în pagină. În acest caz, partialview-ul va fi arătat dacă utilizatorul va extinde secțiunea de vizualizare a răspunsului corect. Această secțiune poate fi accesată prin apăsarea butonului ce conține pictograma unui ochi, după cum poate fi observat și în Figura 26.

Un fragment din sesiunea de învățare este prezentat în Figura 27. În partea de jos putem observa secvența elementelor din sesiunea curentă. Elementul curent are o dimensiune mai mare comparativ cu celelalte. Sub fiecare element se află câte un cerculeț ce reprezintă dacă elementul a fost vizitat sau nu. Navigarea în cadrul sesiunii poate fi efectuată prin selectarea unui element din secțiunea de jos, prin folosirea săgeților dispuse în lateral, sau cu ajutorul săgeților de pe tastatură. Consider foarte util ultimul tip de navigare întrucât este foarte rapid și

comod pentru utilizator. Sesiunea de evaluare a elementelor poate fi accesată doar după ce toate elementele din cadrul secțiunii de învățare au fost vizitate. Pentru fiecare element sunt prezentate câte 3 secțiuni: structura, traducerea și citirea. Aici utilizatorul poate vizualiza, audia, dar și adăuga sinonime sau notițe.

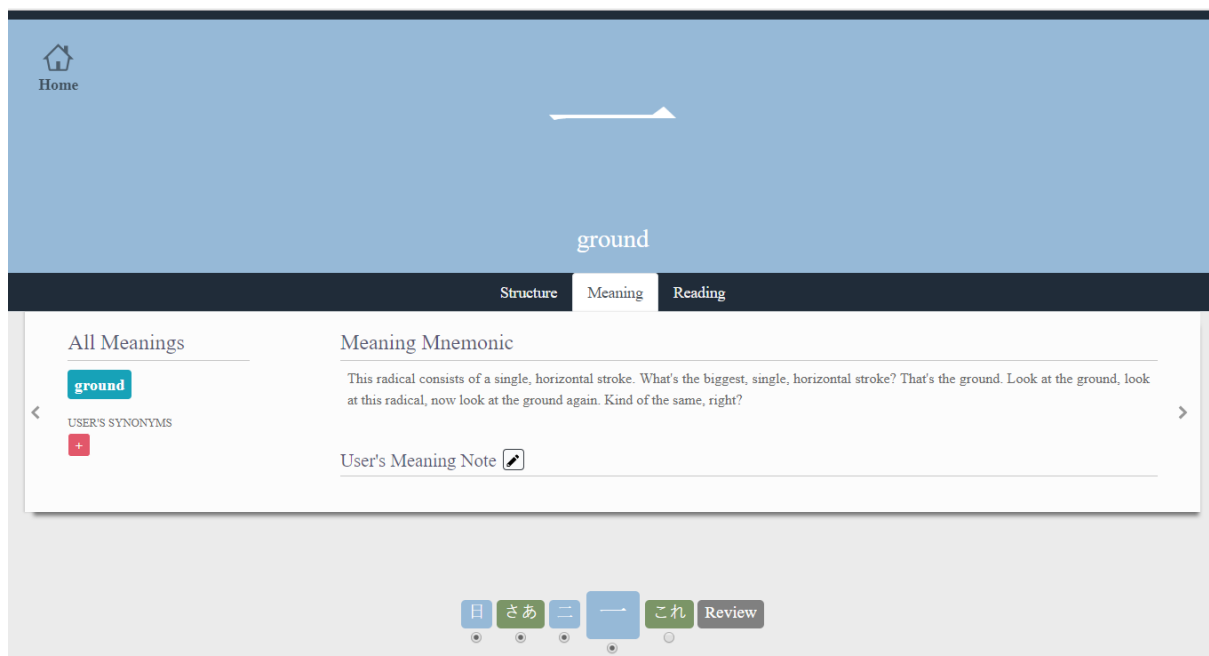


Figura 27: Captură ecran cu sesiunea de învățare

ALL ELEMENTS

三	人	女	女	うん	でも	上	二	丁	ト	一	人	日	イ	一	日	さあ	この
これ	～さん	二	ごろ	五	安い	時	一	一人	二	ナ	友達	五	うん	口			
一日	人	三	言	元気	試	一月	明	大	月	元	安	私	誰	学	弋	四	達
一つ																	
今	友	五	誰	気	大きい	気	イ	幸	月	子	験	ハ	明日	生	学生	月	又
大人	メ	ル	私	ム	今日	元	馬	寺	生	試験	工	四	禾	いつ	込	佳	大

RADICAL				KANJI				WORDS			
人	女	丁	ト	三	女	上	二	うん	でも	さあ	この
イ	一	日	二	一	人	日	五	これ	～さん		
ハ	ナ	五	口	時	試	明	大	ごろ	安い	一	一人
言	弋	気	イ	元	安	私	誰	二	友達	うん	
幸	月	子	ハ	学	達	今	友	一日	人	三	元気
又	メ	ル	ム	気	験	生	月	一月	月	四	一つ
元	馬	寺	生	四				五	誰	大きい	明日
工	禾	込	佳					学生	大人	私	
大								今日	試験	いつ	

Figura 28: Captură ecran cu afișarea tuturor elementelor de vocabular în funcție de progresul lor

În Figura 28 este prezentată afișarea grilă a tuturor elementelor de vocabular colorate corespunzător stadiilor în care se află. Atunci când utilizatorul poziționează cursorul peste un element, sunt afișate informații despre înțelesul și citirea acestuia. Acest mod de a prezenta toate elementele este foarte folositor pentru utilizator, care își poate crea o impresie generală despre stadiile tuturor elementelor, dar și vizualiza rapid traducerea și înțelesul fiecăruia.

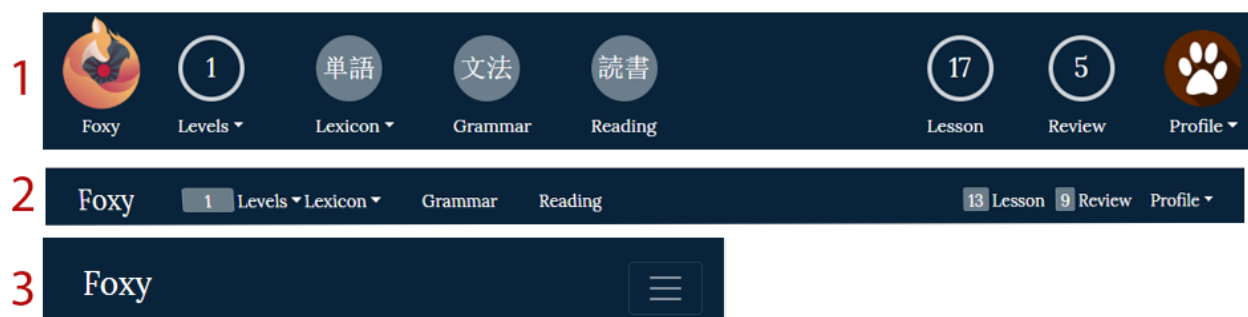


Figura 29: Stările meniului principal

Meniul principal al aplicației are trei stări. În Figura 29 sunt reprezentate aceste stări. Starea 3 reprezintă meniul pentru dispozitivele mobile. Pentru dispozitivele cu dimensiuni mai mari ale ecranului, starea 1 reprezintă meniul extins, care apare atunci când utilizatorul se află în partea de sus a paginii. Meniul trece în starea 2 atunci când utilizatorul navighează mai jos în pagină. Acest lucru l-am obținut folosind JQuery. Aceasta este o platformă de dezvoltare javascript și este foarte ușor de utilizat. Am obținut extinderea și restrângerea meniului prin adăugarea și eliminarea claselor corespunzătoare pentru elementele meniului.

Am creat logo-ul aplicației în Adobe Illustrator. În crearea acestuia am folosit „golden ratio” (secțiunea de aur). Acest șablon presupune crearea ilustrației bazate pe niște cercuri ce au diferența dintre diametru exact 1.618. În Figura 30 este prezentată o etapă din crearea logo-ului. Aici pot fi observate cercurile secțiunii de aur ce respectă proporția menționată.



Figura 30: Etapă din crearea logo-ului

Figura 31 conține o schemă pentru a reprezenta cum ar trebui să arate pagina principală a unui utilizator autentificat. Această schemă a fost creată înainte de a începe crearea interfeței web și a fost un suport bun. În Figura 33 este reprezentată o ilustrație tematică din cadrul aplicației, creată de către mine în Adobe Illustrator. Această ilustrație este folosită pentru pagina de autentificare, înregistrare, dar și pentru finisarea sesiunilor de

examinare. În Figura 32 sunt reprezentate ilustrațiile corespunzătoare stadiului fiecărui item al vocabularului.

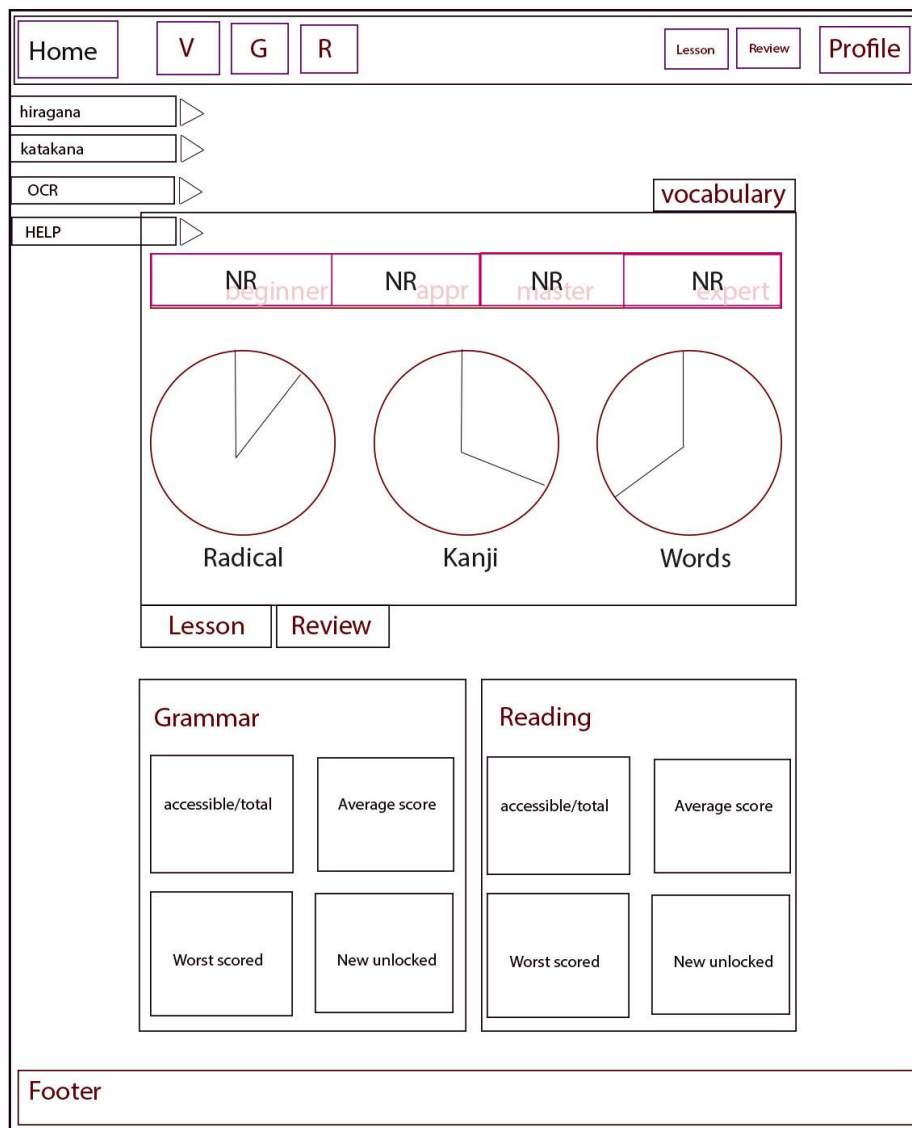


Figura 31: Schemă design pentru pagina principală

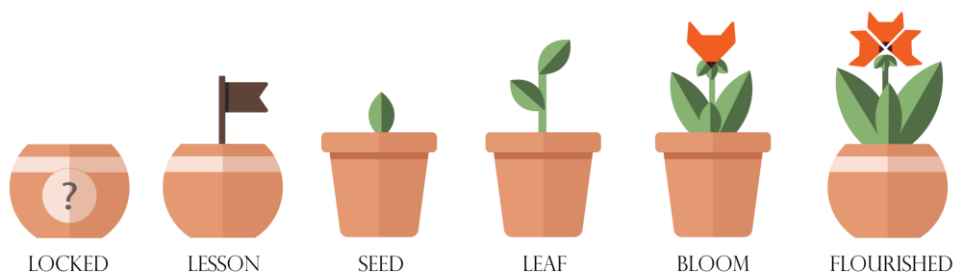


Figura 32: Ilustrații pentru stadiile itemilor vocabularului

Concluziile lucrării

Pentru învățarea limbii japoneze este nevoie de un suport foarte bun. Acesta este și scopul aplicației Foxy. Datorită structurii aplicației, sistemului de repetiție și interfeței web, aplicația oferă utilizatorului eficientizarea procesului de învățare și experiența plăcută în utilizarea aplicației. Tehnologia ASP.NET Core și arhitectura MVC mi-au oferit multe funcționalități, fapt ce a făcut procesul de implementare plăcut și fluent.

Informația din baza de date este consistentă și bine structurată. Au fost folosite surse diverse pentru a crea un conținut interactiv dar și demn de încredere. Din acest punct de vedere, aplicația este complet scalabilă. Acest lucru se poate obține ușor prin completarea fișierelor de tip *json* responsabile pentru menținerea informației și populării bazei de date, fără a fi necesare alte schimbări.

Consider foarte important aspectul vizual al interfeței web, dar și caracterul intuitiv. Consider că am reușit să ofer utilizatorului o experiență vizuală plăcută, datorită elementelor de design și a ilustrațiilor tematice. De asemenea, toate paginile și componentele aplicației sunt structurate cât mai clar și logic. Acest lucru este esențial pentru utilizator, care trebuie să se focuzeze asupra procesului de învățare, ci nu cel al descifrării modului de structurare și utilizare a aplicației.

Dept aspecte de viitor, aplicația poate evolua din mai multe puncte de vedere. Ar putea fi creată o versiune mult mai diversificată a sistemului spațiat de repetiție. De exemplu ar putea exista mai multe categorii referitoare la viteza procesului de învățare. Utilizatorul și-ar putea selecta singur ritmul în care dorește să învețe (lent, mediu, rapid). Un alt aspect ar fi pronunția cuvintelor japoneze. Întrucât aplicația nu vizează în niciun fel acest aspect, ar fi utilă implementarea unui mecanism de recunoaștere a vocii utilizatorului și de comparare a acesteia cu un rezultat considerat corect.

O altă idee ar fi introducerea unui compartiment pentru exersarea scrierii elementelor. Aceasta ar putea fi efectuată digital sau tradițional, pe foaie. Un alt element nou, ce consider că ar aduce un mare avantaj utilizatorului, ar fi un compartiment dedicat culturii și literaturii japoneze. Acesta ar alimenta constant interesul utilizatorului și l-ar aduce cât mai aproape de cultura limbii învățate.

Prin urmare, aplicația Foxy oferă utilizatorului funcționalitățile necesare pentru învățarea tuturor compartimentelor limbii japoneze. Principalul scop al acestei aplicații este să facă procesul de învățare cât mai ușor, eficient și plăcut. Scalabilitatea aplicației din punct de

vedere al informației expuse este asigurată și ușor de obținut. Designul, aceasta poate fi mereu îmbunătățită și înprospătată cu elemente noi.



Figura 33: Ilustrație din cadrul aplicației

Bibliografie

- [1] R. Szypulski, „5 unexpected benefits of learning another language,” 6 2 2016. [Interactiv]. Available: <https://examinedexistence.com/12-benefits-of-learning-a-foreign-language-2/>. [Accesat 8 6 2018].
- [2] L. Neuman, „Why study Japanese? Here are 8 reasons to start with!,” 21 1 2018. [Interactiv]. Available: <https://gogonihon.com/en/blog/why-study-japanese/>. [Accesat 10 5 2018].
- [3] L. Lombardi, „KITSUNE: THE DIVINE/EVIL FOX YOKAI,” 9 9 2014. [Interactiv]. Available: <https://www.tofugu.com/japan/kitsune-yokai-fox/>. [Accesat 16 6 2018].
- [4] S. S. Shekhawat, „Onion architecture In ASP.NET Core MVC,” 1 1 2017. [Interactiv]. Available: <https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-mvc/>. [Accesat 20 6 2018].
- [5] S. Smith, „Overview of ASP.NET Core MVC,” Microsoft, 8 1 2018. [Interactiv]. Available: <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1>. [Accesat 20 6 2018].
- [6] „Creating a Model,” Microsoft, 27 10 2016. [Interactiv]. Available: <https://docs.microsoft.com/en-us/ef/core/modeling/index>. [Accesat 21 6 2018].
- [7] T. Ugurlu, „Generic Repository Pattern - Entity Framework, ASP.NET MVC and Unit Testing Triangle,” 22 12 2011. [Interactiv]. Available: <http://www.tugberkugurlu.com/archive/generic-repository-pattern-entity-framework-asp-net-mvc-and-unit-testing-triangle>. [Accesat 21 6 2018].
- [8] M. Wenzel, „Lambda Expressions (C# Programming Guide),” Microsoft, 3 3 2017. [Interactiv]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>. [Accesat 21 6 2018].
- [9] B. Wagner, „Language Integrated Query (LINQ),” Microsoft, 2 2 2017. [Interactiv]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>. [Accesat 21 6 2018].
- [10] R. Anderson, „ASP.NET Core Middleware,” Microsoft, 22 1 2018. [Interactiv]. Available: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1&tabs=aspnetcore2x>. [Accesat 22 6 2018].
- [11] S. Addie, „Dependency injection in ASP.NET Core,” Microsoft, 14 10 2016. [Interactiv]. Available: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1>. [Accesat 22 6 2018].
- [12] C. Nienaber, „ASP.NET Core Web API help pages with Swagger / Open API,” Microsoft, 9 3 2018. [Interactiv]. Available: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-2.1>. [Accesat 22 6 2018].

- [13] „Japanese-Language proficiency test,” [Interactiv]. Available: <https://www.jlpt.jp/>.
- [14] „Tofugu,” [Interactiv]. Available: <https://www.tofugu.com/>.
- [15] „Wanikani,” [Interactiv]. Available: <https://www.wanikani.com/>.
- [16] „Sound to speech,” [Interactiv]. Available: <https://soundoftext.com/>.
- [17] „Wasabi JPN,” [Interactiv]. Available: <https://www.wasabi-jpn.com/>.
- [18] „Toastr git repository,” [Interactiv]. Available: <https://github.com/CodeSeven/toastr>.
- [19] „Circliful javascript plugin,” [Interactiv]. Available: <https://github.com/pguso/jquery-plugin-circliful>.
- [20] „FontAwesome,” FontAwesome, [Interactiv]. Available: <https://fontawesome.com/>.
- [21] „WanaKana,” [Interactiv]. Available: <https://wanakana.com/>.