Ministerul Educaţiei, Culturii şi Cercetarii al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Ingineria Software si Automatica

RAPORT

Lucrare de laborator nr.2

Disciplina: Programarea aplicațiilor distribuite.
Tema: Web Proxy: realizarea transparenței în distribuire.

A efectuat: st. Prodan Marcel, Ceban Vitalie, Zavorot Daniel
gr.TI-194

A verificat: asist. univ. Cristofor Fiștic

Chişinău 2022

Scopul lucrării

Studiul protocolului HTTP în contextul distribuirii datelor. Utilizarea metodelor HTTP în implementarea interacțiunilor dintre client și un server de aplicații. Studiul metodelor de caching și load-balancing aplicate la crearea unui serviciu proxy.

Obiectivele lucrări

Elaborarea unui server cu procesare concurentă a cererilor HTTP. Realizarea unei aplicații de intermediere a accesului la nodurile warehouse.

Considerații generale

Etapa 1

În această etapă clientul va obține datele semi-structurate distribuite prin intermediul unui nou nod special creat. Acest nod informativ va juca rolul de data-warehouse (DW). Serverul informativ va păstra toate datele necesare clientului. Orice interacțiune cu acest server se va face prin protocolul HTTP:Metoda GET se va utiliza de client pentru a cere date trimise anterior de cumpărare (e.g. GET/employee/?id=1 sau GET /employees/?offset=1&limit=4). Metoda PUT se va utiliza de nodurile de stocare pentru a trimite la nodul informativ datele necesare clientului. Metoda POST poate fi utiliza pentru transmiterea datelor de modificat spre DW, iar nodul sursă prin interogări repetate să ceară eventualele modificări (e.g. GET update/employees/ - metoda pull-request). Opțional, se acceptă metodele PUSH, cînd la modificarea unei entități nodul DW va iniția sau cerere spre nodul sursă.

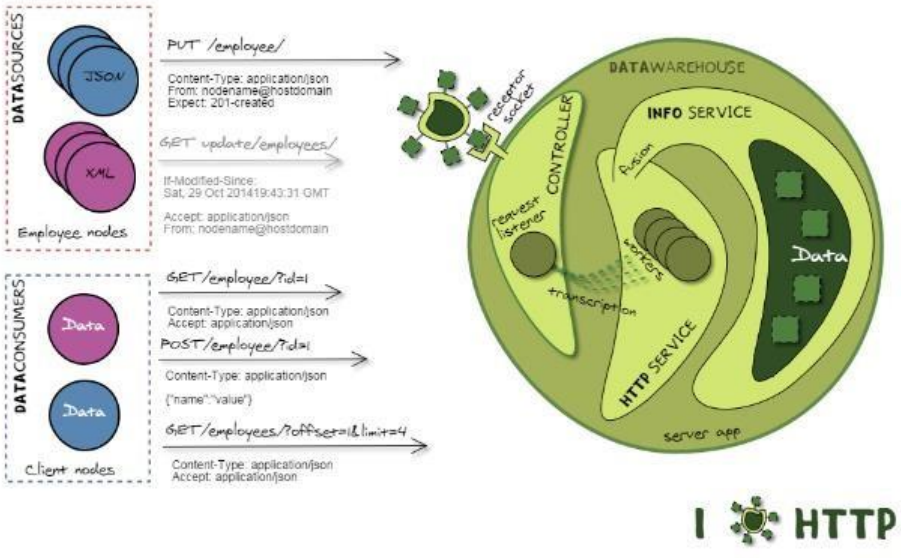


Figura 1.1 – Fluxul de cereri HTTP.

În vederea utilizării eficiente a resurselor de calcul se va impune procesarea concurentă a cererilor. Modelul multi-threading va corespunde schemei thread-per-request. În condițiile există acces simultan la colecția de date (adăugare/citire) se cere folosi unele colecții thread-safe sau implementarea unor mecanisme de acces reciproc exclusiv. „Maparea" cererilor HTTP pe operațiunile de servicii se va face în componente numite generice Controlor.

Etapa 2

În această etapă, toate cererile de achiziții vor fi preluate de un reverse proxy (mai departe proxy). Un proxy este un program care acționează ca un intermediar între o aplicație client și un end-server. Astfel, în loc de interacțiune directă cu serverul final pentru a obține datele necesare, clientul accesează aplicația proxy care și transmite cererea spre serverul final. Cînd end-server (în contextul DW) trimite răspunsul spre proxy, anume proxy și va trimite răspunsul final spre client. Printre beneficiile utilizării proxy se numără: load balancing și caching.

Caching este memorarea temporară a răspunsurilor trimise de serverele finale la nivelul nodului proxy. Aceasta se face pentru a accelera comunicarea între client și server dar și pentru a micșora numărul de cereri către end-servers. Dacă nodul proxy detectează o cerere, răspunsul la care a fost memorat, acesta îl va returnainstantaneu clientului. Natura temporară a procesului de caching permite ca proxy să nu transmită răspunsuri învechite care între timp au putut deveni incorecte.

Load balancing este procesul prin care un proxy distribuie sarcina între mai multe servere finale. Aceasta asigură nici unul dintre end-servers , la un moment dat, nu este suprasolicitat, iar cererile clienții sunt procesate rapid și într-o manieră eficientă. Unul dintre algoritmii des folosiți pentru load balancing este algoritmul Round-Robin.

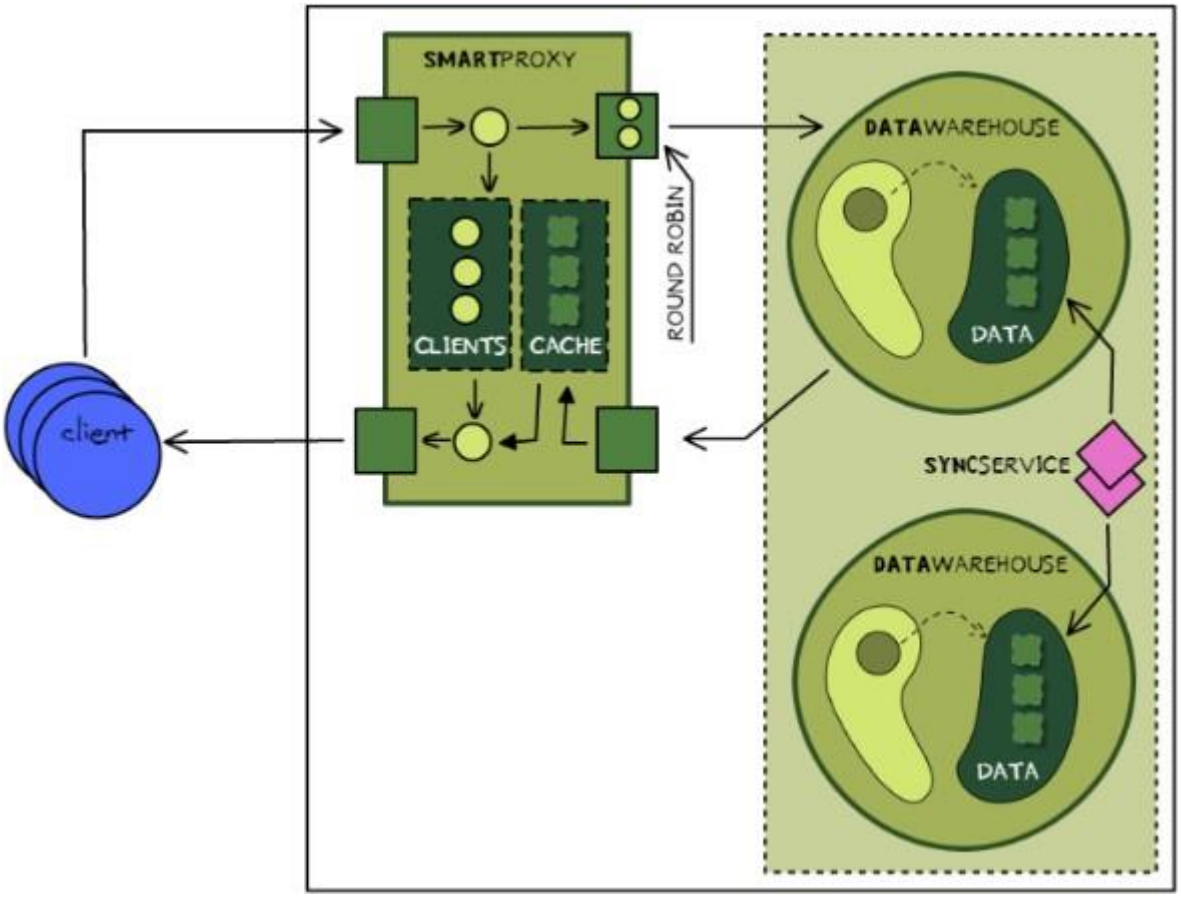


Figura 2 – Distribuția sistemului pe mai multe noduri.

1. Elaborarea lucrării, etapa 1.

Pentru elaborarea lucrării de laborator numărul 2 s-a ales cadrul de dezvoltare .NET și pentru stocarea datelor s-a ales o baza de date NoSQL și anume mongoDB. În prima etapă s-a creat un API care ar face legătura dintre un client și baza de date. În acesta s-au creat:

1.1. Modele aplicației (entitățile)

```
14 references
public class Movie: MongoDocument
{
    0 references
    public string Name { get; set; }
    0 references
    public List<string> Actors { get; set; }
    0 references
    public decimal? Budget { get; set; }
    0 references
    public string Description { get; set; }
}
```

Figura 3 – Modelul pentru filme.

1.2. Controllerul (end-pointul) cu cele 4 methode get, put, post, delete.

```
[HttpGet]
0 references
public List<Movie> GetAllMovies()
{
    var records = _movieRepository.GetAllRecords();

    return records;
}
```

Figura 4 – Endpointul de returnare a listei cu toate filemele.

```
[HttpGet("{id}")]
0 references
public Movie GetMovieById(Guid id)
{
    var result = _movieRepository.GetRecordById(id);

    return result;
}
```

Figura 5 – Endpointul de returnarea a liste cu un film concret.

```
[HttpPost]
0 references
public IActionResult Create(Movie movie)
{
    movie.LastChangedAt = DateTime.UtcNow;
    var result = _movieRepository.InsertRecord(movie);

    _movieSyncService.Upsert(movie);

    return Ok(result);
}
```

Figura 6 – Endpointul de crearea a unui film.

```
[HttpPut]
0 references
public IActionResult Upsert(Movie movie)
{
    if (movie.Id == Guid.Empty)
    {
        return BadRequest("Empty Id");
    }

    movie.LastChangedAt = DateTime.UtcNow;
    _movieRepository.UpsertRecord(movie);
    _movieSyncService.Upsert(movie);

    return Ok(movie);
}
```

Figura 7 – Endpointul de modificarea a unui film concret.

1.3. Repositoriul generic.

Un repositoriu generic este o clasa carea nu are un tip definit concret si este initializat cu indicarea tipului care ar trebui să fie această instanță. Această abordare ne permite să apelăm acest repositoriu nu doar pentru entitatea de filme dar și pentru altele care vor fi create. Acest repositoriu contine o instanta a ORM folosit pentru crearea interogarilor care permite aplicarea cererilor cum ar fi get, update, delete.

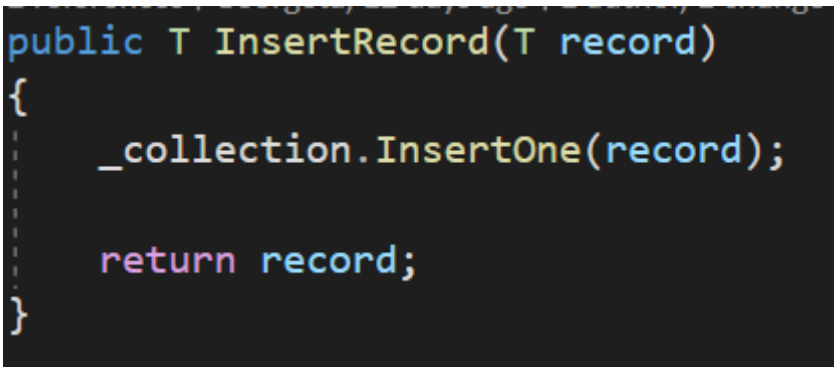
```
public void DeleteRecord(Guid id)
{
    _collection.DeleteOne(doc => doc.Id == id);
}
```

Figura 7 – Metoda de stergere a unui record în baza la id.

```
public T GetRecordById(Guid id)
{
    var record = _collection.Find(doc => doc.Id == id).FirstOrDefault();

    return record;
}
```

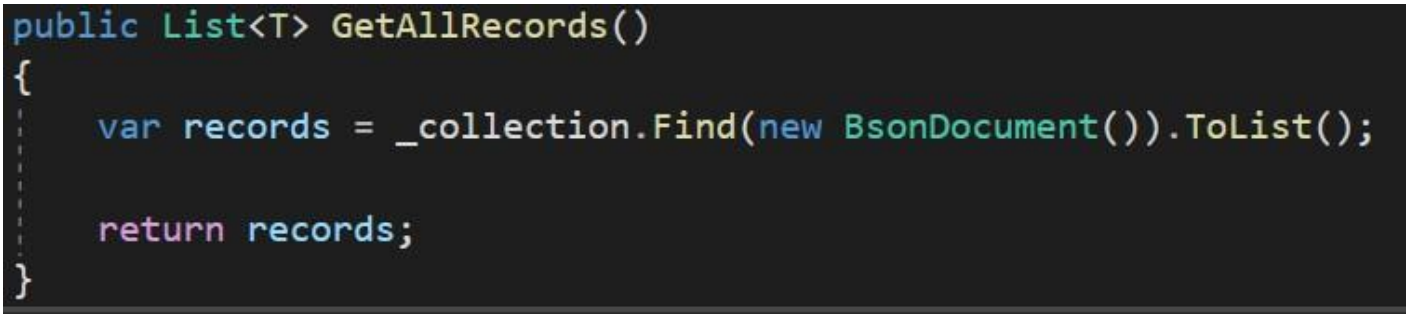
Figura 8 – Metoda de obținere a unui record în baza la id.



```
public T InsertRecord(T record)
{
    _collection.InsertOne(record);

    return record;
}
```

Figura 9 – Metoda de inserarea a unui record nou.



```
public List<T> GetAllRecords()
{
    var records = _collection.Find(new BsonDocument()).ToList();

    return records;
}
```

Figura 10 – Metoda de obținerea a tuturor recordurilor.

2. Elaborarea lucrării, etapa 2:

Pentru această etapa s-a apelat la folosirea librăriei Ocelot, cu ajutorul căruia a fost implementat Load Balancing-ul și cache-ul. Acesta ar fi un punct prin care trec toate cererile noastre si apoi sunt rutate la nodul principal. Ocelot ne-a permis o implimentare usoară a round robin in proiectul nostru, round robin este înparțirea egală a ceririlor către nodurile disponibile. Figura 11 relevă fisierul de configurare ocelot.json, în acesta am declarat porturile celor 2 api în cazul nostru 9001 și 9002, la fel s-a declarat și metoda de împărțire a cererilor, în cazul nostru Round Robin.

```
"UpstreamPathTemplate": "/{url}",
"UpstreamHttpMethod": [ "Get", "Put", "Post", "Delete" ],
"LoadBalancerOptions": {
  "Type": "RoundRobin"
},
"FileCacheOptions": {
  "TtlSeconds": 10,
  "Region": "movieCacheRegion"
}
```

Figura 11 – Fișierul ocelot.json

În fișier se poate evidenția și secțiunea de configurarea a cache-ului „FileCacheOptions” acesta este setat pentru pastrarea cache-ului timp de 10 secunde in sectiunea cu denumirea „moviecacheRegion”.

3. Elaborarea lucrării, etapa 3:

În această etapă sa implementat synchronizarea a două noduri. Pentru aceasta sa creat un proiect nou ASP.NET Core Web API în care s-a configurat procesul de sincronizare, aceasta va fi o sincronizare manuala și lucrează dupa tipul când într-un nod nou a fost modificat ceva, acesta este notificat, după care el face sincronizarea pe celelalte noduri. Acesta este un microserviciu care este apelat de fiecare dată cand s-au făcut modificari pe unul din nodurile serverului. Pentru aceasta s-au creat:

3.1. Entitate de sincronizare:

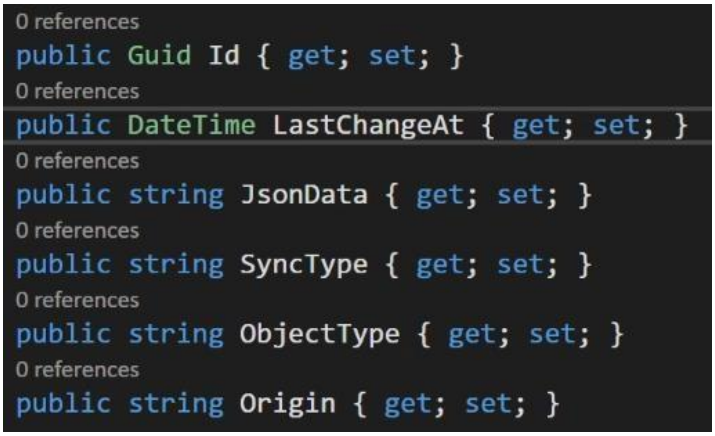


Figura 12 – Entitatea de sincronizare.

În figura 12 este prezentată entitatea de sinchronizare, aceasta este modelul care va fi acceptat de serviciul de sincronizare de fiecare data când este nevoie de sincronizare. Se poate observa ca proprietatea SyncType este tipul sincronizarii, poate fi delete sau upsert, delete cand ceva sa sters de pe un node si este nevoie să se stearga si de pe restu iar upsert cand ceva s-a adaugat s-au sa modificat de pe un nod si trebuie sincronizat si pe celelalte noduri.

3.2. Serviciul de sincronizare:

```
public HttpResponseMessage Delete(T record)
{
    var syncType = _settings.DeleteHttpMethod;
    var json = ToSyncEntityJson(record, syncType);

    var response = HttpClientUtility.SendJson(json, _settings.Host, "POST");

    return response;
}
```

Figura 13 – Metoda de delete pentru sincronizarea nodurilor.

```
public HttpResponseMessage Upsert(T record)
{
    var syncType = _settings.UpsertHttpMethod;
    var json = ToSyncEntityJson(record, syncType);

    var response = HttpClientUtility.SendJson(json, _settings.Host, "POST");

    return response;
}
```

Figura 14 – Metoda upsert pentru sincronizarea nodurilor.

3.3. Serviciul de sincronizare atomat la fiecare 20 secunde.

```
public Task StartAsync(CancellationToken cancellationToken)
{
    _timer = new Timer(DoSendWork, null, TimeSpan.Zero, TimeSpan.FromSeconds(20));

    return Task.CompletedTask;
}
```

Figura 15 – Metoda de apelare a procesului de sincronizare.

```
private void DoSendWork(object state)
{
    foreach(var doc in documents)
    {
        SyncEntity entity = null;
        var isPresent = documents.TryRemove(doc.Key, out entity);

        if (isPresent)
        {
            var receivers = _settings.Hosts.Where(x => !x.Contains(entity.Origin));

            foreach(var receiver in receivers)
            {

                var url = $"{receiver}/{entity.ObjectType}/sync";

                var result = HttpClientUtility.SendJson(entity.JsonData, url, entity.SyncType);
            }
        }
    }
}
```

Figura 16 – Metoda de apelare a sincronizării.

4. Elaborarea lucrării, etapa suplimentară:

La această etapă s-a realizat interfața clientului, echipa a ales pentru crearea aplicației client framework-ul Angular. Clientul este un proiect simplu care implementeaza cele 4 cruduri create în back și afisarea datelor corespunzător.

4.1. Interfața client:

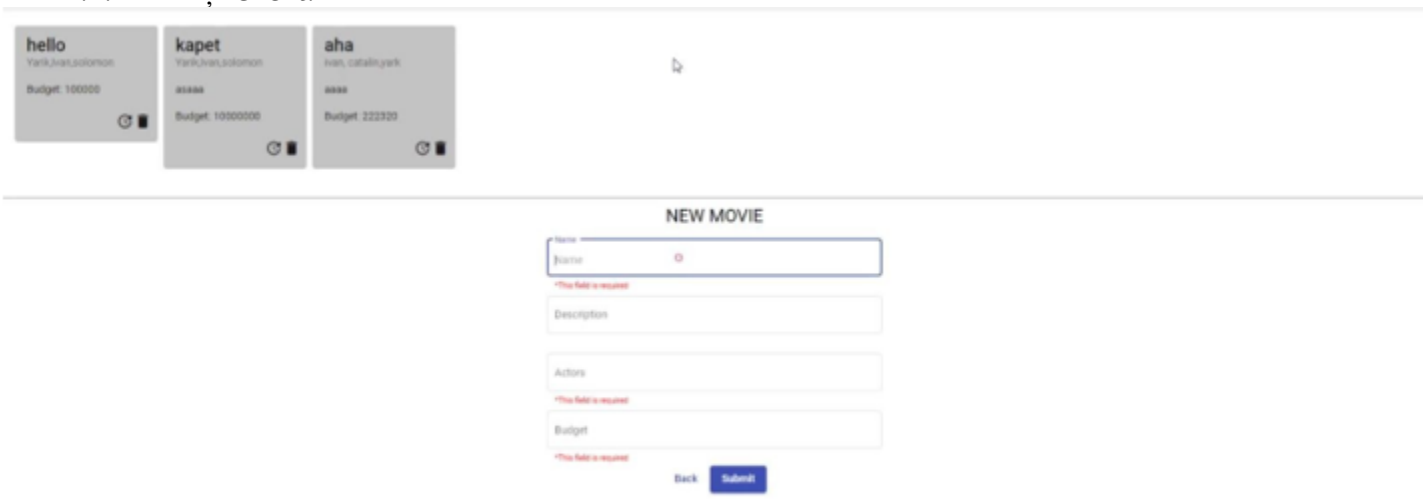


Figura 17 – Interfața client.

4.2. Serviciul de exploatare a end-pointurilor:

```
getMovies(): Observable<any> {
  return this.http.get<any>(this.apiUrl + 'movie');
}

getMovie(id: string): Observable<any> {
  return this.http.get<any>(this.apiUrl + 'movie/' + id);
}

deleteMovie(id: string): Observable<any> {
  return this.http.delete<any>(this.apiUrl + 'movie/' + id);
}

updateMovie(movie: MovieModel): Observable<any> {
  return this.http.put<any>(this.apiUrl + 'movie/', movie);
}

addMovie(movie: MovieModel): Observable<any> {
  return this.http.post<any>(this.apiUrl + 'movie/', movie);
}
```

Figura 18 – Metodele de exploatare a end-pointurilor.

```
getMovies() {
  this.movieService.getMovies().subscribe(
    (response) => {
      this.movies = response;
    },
    (error) => {
      console.log(error);
    }
  );
}
```

Figura 19 – Obtinerea filmelor in client.

```
deleteMovie(id: string) {
  this.movieService.deleteMovie(id).subscribe(
    (response) => {
      this.getMovies();
      console.log('Succes', response);
    }, error => {
      this.getMovies();
      console.log('error', error);
    }
  );
}
```

Figura 20 – Stergerea unui film concrete din client.

```
var item = new MovieModel();
item.name = this.movieForm.get('name')?.value;
item.description = this.movieForm.get('description')?.value;
let actors = this.movieForm.get('actors')?.value;
item.actors = actors.split(',');
item.budget = parseInt(this.movieForm.get('budget')?.value);
this.movieService.addMovie(item).subscribe(() => {
  console.log('Movie added!');
  this.getMovies();
  this.showForm = 0;
}, () => {
  console.log('Something went wrong!');
})
```

Figura 21 – Adaugarea unui film din client.

```
var item = new MovieModel();
item.name = this.movieForm.get('name')?.value;
item.id = this.movieForm.get('id')?.value;
item.description = this.movieForm.get('description')?.value;
let actors = this.movieForm.get('actors')?.value.toString();
item.actors = actors.split(',');
item.budget = parseInt(this.movieForm.get('budget')?.value);
this.movieService.updateMovie(item).subscribe(() => {
  console.log('Movie updated!');
  this.getMovies();
  this.showForm = 0;
}, () => {
  console.log('Something went wrong!');
})
```

Figura 22 – Modificarea datelor prin intermediul clientului.

Concluzie

La lucrarea de laborator numărul 2 s-a făcut cunoștințele cu metodele de caching a requesturilor și impărțirea lor prin intermediul librăriei Ocelot, s-a implementat sincronizarea a 2-a noduri și s-a lucrat cu o bază de date noSQL și anume mongoDB. Aceasta este o experiență foarte importantă acumulată de echipă deoarece s-a înteles metodele de lucru a sincronizarii nodurilor și cum un sistem mare distribuit își poate impărți cererile pe mai multe noduri cu ajutorul Round Robin. La fel pentru echipa a avut o primă experiență de lucrul cu o bază non-relațională. Celelalte tehnologii nu au fost ceva inovativ pentru echipă însă la fel experiența acumulată pe parcursul realizării acestui laborator a adus noi cunoștințe care în viitorul apropiat vor putea fi puse în practică. Cu siguranță nu o să ne oprim aici, dar o să ne aprofundăm în tema sincronizarii, pentru a afla și implementarea altor metode de sincronizare și caching.