CQRS, ES Tutorial

# Part 1: Design

Ez a tutorial azt mutatja be, hogyan lehet létrehozni egy DDD-re alapuló architektúrát Event Sourcing (ES) és CQRS segítségével. Megnézzük, hogyan lehet felépíteni a read modelleket majd a read modelleket felhasználjuk egy egyszerű ASP.NET MVC-s alkalmazásban.

A tutorialban lépésről-lépésre fogjuk létrehozni az alkalmazást.

A tutorial a cqrs.nu tutorialja alapján készült, azt bővíti ki, tartalmazza az edument által készített alap kódokat.

## A Domain

Az szálloda/étterem domainban fogunk dolgozni. Az első részekben az étterem domainban vesszük az alap dolgokat, később kibővítjük ezeket a szálloda domain-nel.

Az étteremben a **számlára** fókuszálunk, mint alap dologra, ami nyomon követi egy személy vagy csoport látogatását az étteremben. Amikor valaki beérkezik az étterembe, számlát **nyit**nak neki. A vendég utána **rendel**het italokat és ételeket. Az **italok**at azonnal **felszolgálja** a személyzet, az **étel**eket azonban el kell készítenie a séfnek. Miután a séf **elkészítette** az ételt, a pincér **felszolgálja**. Az étteremben tartózkodásuk alatt a vendégek bármikor **rendel**hetnek még ételt és italt is. **Módosít**hatják is a rendeléseiket, de csak addig, amíg azokat fel nem szolgálták. Végül a pincérek **lezár**ják a számlát, a vendégek pedig fizetnek és adhatnak borravalót is, ha elégedettek voltak a felszolgálással. A számlát teljesen, egy összegben, készpénzben ki kell egyenlítenie a vendégnek. Egy számla csak akkor zárható le, ha minden tételét felszolgálták vagy **lemond**ták.

## Az események

A fenti leírásban kiemeltem néhány igét és főnevet. Adatbázis centrikus módon dolgozunk, akkor inkább a főnevekre figyelünk, amiket táblákhoz kapcsolunk. Ebben az esetben az igék másodlagos jelentőségűek. Amikor parancsokat és eseményeket tervezünk, akkor az igék válnak fontossá. Az igékkel jobban le lehet írni egy üzleti folyamatot, mint a főnevekkel.

Események, amik a fentiek alapján szóba jönnek:

* BillOpened
* DrinksOrdered
* FoodOrdered
* DrinksCancelled
* FoodCancelled
* DrinksServed
* FoodPrepared
* FoodServed
* BillClosed

Az események nagyon domain központúak. Azért kell külön venni az italrendelést az ételrendeléstől, mert az italokat azonnal fel tudják szolgálni, míg az ételeket először el kell készíteni.

Lehetőleg ne használjunk általános kifejezéseket, mint például Created, Updated, Deleted. Szükség lehet ilyen kifejezésekre néha, de alapvetően a domain kifejezéseiből induljunk ki.

Az események múlt idejű igékkel vannak leírva.

## A parancsok

A parancsok segítségével tudunk kéréseket intézni a domain felé. Míg az esemény azt jelzi, hogy valami megtörtént, a parancsot végre lehet hajtani vagy el lehet utasítani. Egy végrehajtott parancs nulla vagy több eseményt generál, hogy a tényeket rögzítse a rendszerben. Egy elutasított parancs kivételt dob.

A parancsokat úgy ismerjük fel, hogy igéket keresünk. Arra kell fókuszálni, hogy a felhasználó mit tekint műveletnek. Például: az italokat és az ételeket különbözőképpen kezeljük a rendelés során, a vendégek nem akarják külön megrendelni az ételeket és külön az italokat. Ezért van egy parancs a rendelésre.

Parancsok, amik szóba jönnek:

* OpenBill
* PlaceOrder
* AmendOrder
* MarkDrinksSaved
* MarkFoodPrepared
* MarkFoodServed
* CloseBill

A parancsok felszólító módú igéket tartalmaznak.

## Kivételek

Fontos azon is elgondolkodni, hogy milyen dolgok miatt lehet visszautasítani egy parancsot. Kivételtípusokat definiálunk azokra az esetekre, amikor vissza kell utasítani egy parancsot. Ezek a kivételtípusok üzenetet hordozhatnak arra vonatkozóan, miért nem lehet a parancsot végrehajtani. Ez azért fontos, mert a domain logikának tudatnia kell a frontenddel, hogy mi a baj.

Esetünkben a következő kivételtípusok jönnek szóba:

* CannotCancelServedItem
* BillHasUnservedItems
* MustPayEnough

Itt a nevek eléggé beszédesek ahhoz, hogy tudjuk, mi a baj.

## Aggregátumok

Hogy el tudjuk dönteni, végrehajthatunk-e egy parancsot, tudnunk kell az aktuális állapotot. Ehhez minden szükséges információ megtalálható az eseményekben. Azonban minket nem általánosságban érdekel a teljes eseményfolyam, bennünket csak bizonyos események érdekelnek, például azok, amik egy bizonyos számlával vannak kapcsolatban, vagyis mintha minden számlának saját esemény folyama lenne.

Az aggregátumok jönnek a segítségünkre. Minden egyes aggregátumnak van saját esemény folyama. Ezeket az esemény folyamokat összesítve ki tudjuk számítani az aggregátum aktuális állapotát. Hogy egy parancs végrehajtható-e, az magától a parancstól és az aggregátum által összefogott eddigi eseményektől függ.

Egy aggregátum:

* vagy egy különálló objektum, ami nem hivatkozik másik objektumra
* vagy objektumok elkülönített gráfja, egy kiválasztott objektummal, ami az aggregátum gyökere (aggregate root). A külvilág csak a gyökeret ismeri.

A példaprogram úgy lett megválasztva, hogy csak egy aggregátum típust tartalmazzon. A legtöbb rendszerben azonban többet kell foglalkozni az aggregátumok azonosításával.

# Part 2: Domain logika

## Az első parancsunk és eseményünk

A számlanyitással kezdjük, mert az összes többi parancs csak nyitott számlával hajtható végre. Először definiáljuk a BillOpened eseményt, tartalmaz egy egyedi azonosítót, az asztal számát, ahol a vendégek elhelyezkedtek és a pincért, aki felszolgál nekik.

public class BillOpened

{

public Guid Id;

public int TableNumber;

public string Waiter;

}

Az egyetlen követelmény, hogy az esemény tartalmazzon egy Id mezőt, ami Guid típusú.

A parancs hasonlóan néz ki ebben az esetben:

public class OpenBill

{

public Guid Id;

public int TableNumber;

public string Waiter;

}

Ebben az esetben is az egyetlen követelmény, hogy az esemény tartalmazzon egy Id mezőt, ami Guid típusú és az aggregátum ez alapján csoportosítja a parancsokban található adatokat.

## Az első teszt

Készítünk egy teszt projektet is, amivel tesztelni fogjuk az aggregátumok működését. A projektben felhasználtam az edument által készített BDDTest osztályt, ami aggregátumok teszteléséhez nyújt nagy segítséget. Megadhatjuk, hogy milyen előfeltételek esetén (Given), amikor egy parancsot futtatunk (When), akkor milyen esemény létrejöttét várjuk (Then).

Az elő tesztünk a számlanyitás parancsot teszteli. Ehhez először létrehozzuk a számla aggregátumot, most még üresen:

public class BillAggregate : Aggregate

{

}

Az Aggregate osztály tartalmaz az aggregátumokkal kapcsolatban egy alap infrastruktúrát, mint például tárolja az aggregátum azonosítóját, és az események alkalmazásához szükséges alap kódot.

[TestFixture]

public class BillTests : BDDTest<BillAggregate>

{

private Guid testId;

private int testTable;

private string testWaiter;

[SetUp]

public void Setup()

{

testId = Guid.NewGuid();

testTable = 42;

testWaiter = "John";

}

[Test]

public void CanOpenANewBill()

{

Test(

Given(),

When(new OpenBill

{

Id = testId,

TableNumber = testTable,

Waiter = testWaiter,

}),

Then(new BillOpened

{

Id = testId,

TableNumber = testTable,

Waiter = testWaiter,

}));

}

}

Hozzá kell adni a projekthez a NUnit és az NUnit3TestAdapter NuGet csomagot.

Ha lefuttatjuk a tesztet, hibát kapunk, mivel még nem implementáltuk az aggregátumot, ezért nem jön létre a BillOpened esemény.

## Az első aggregátum

A BillAggregate a következőképpen néz ki:

public class BillAggregate : Aggregate,

IHandleCommand<OpenBill>

{

public IEnumerable Handle(OpenBill c)

{

yield return new BillOpened

{

Id = c.Id,

TableNumber = c.TableNumber,

Waiter = c.Waiter,

};

}

}

Ha újra futtatjuk a tesztet, akkor most már sikeresnek kell lennie.

A BillAggregate tartalmazza pillanatnyilag az összes parancsot. A Handle metódus kezeli a parancsokat, paraméterben kell azokat megadni. Az OpenBill parancsra egy BillOpened eseményt generálunk, aminek átadjuk a parancs paramétereit. Azért használunk yield utasítást, hogy akár nulla, akár több eseményt is tudjunk generálni.

## Rendelések felvétele

Következő feladat a rendelések felvétele. Először definiáljuk a DrinksOrdered és FoodOrdered eseményeket:

public class DrinksOrdered

{

public Guid Id;

public List<OrderedItem> Items;

}

public class FoodOrdered

{

public Guid Id;

public List<OrderedItem> Items;

}

A parancs sem meglepő:

public class PlaceOrder

{

public Guid Id;

public List<OrderedItem> Items;

}

Rendelés csak nyitott számlára vihető fel. Szükségünk van egy kivételre, ha a számla nem nyitott:

public class BillNotOpen : Exception

{

}

Megírhatjuk a hozzá tartozó tesztet is:

[Test]

public void CanNotOrderWithUnopenedBill()

{

Test(

Given(),

When(new PlaceOrder

{

Id = testId,

Items = new List<OrderedItem> { testDrink1 },

}),

ThenFailWith<BillNotOpen>());

}

A teszt hibát dob a következő üzenettel: Aggregate BillAggregate does not yet handle command PlaceOrder. Vagyis el kellene készíteni a parancs kezelőjét.

public IEnumerable Handle(PlaceOrder c)

{

throw new BillNotOpen();

}

Egyelőre csak ennyi. Nemsokára megírjuk a teljes metódust.

Kellene még teszt arra, hogy leteszteljünk három rendelési esetet: csak italt, csak ételt, vagy ételt és italt is rendeltek. Biztosak akarunk lenni abban, hogy a megfelelő eseményeket fogjuk kigenerálni. (ld. forráskódot: CanPlaceDrinksOrder, CanPlaceFoodOrder, CanPlaceFoodAndDrinkOrder metódusokat)

Mindhárom teszt ugyanazt a hibát adja: Aggregate BillAggregate does not know how to apply event BillOpened.

Az aggregátum fontos szerepet játszik abban, hogy az eseménytörténetet átalakítsa az aktuális állapot reprezentációjába. Ezt a generikus IApplyEvent implementálásával teszi meg, minden releváns eseményhez meg kell írni az Apply metódust.

public class BillAggregate : Aggregate,

IHandleCommand<OpenBill>,

IHandleCommand<PlaceOrder>,

IApplyEvent<BillOpened>

{

private bool open = false;

// …

public void Apply(BillOpened e)

{

open = true;

}

}

A parancsok sosem módosítják az állapotot, azt mindig csak az események teszik! A fenti példában is a parancs csak egy eseményt generál, és az eseményt paraméterül kapó Apply metódus végzi el az állapot módosítását. Ez nagyon fontos, hogy később újra tudjuk játszani a tárolt eseményeket és a végén ugyanabba az állapotba jusson a program.

Ez után már más kivétellel futnak hibára a tesztek: Expected events, but got exception BillNotOpen. Ezt a parancskezelő adja, amiben jelenleg még csak a kivételt dobó sor található. Írjuk meg ezt a metódust:

public IEnumerable Handle(PlaceOrder c)

{

if (!open)

{

throw new BillNotOpen();

}

var drink = c.Items.Where(w => w.IsDrink).ToList();

if (drink.Any())

{

yield return new DrinksOrdered

{

Id = c.Id,

Items = drink,

};

}

var food = c.Items.Where(w => !w.IsDrink).ToList();

if (food.Any())

{

yield return new FoodOrdered

{

Id = c.Id,

Items = food,

};

}

}

Ha nincs nyitott számla, akkor kivételt dobunk. A parancs dobhat kivételt, ha valamiért nem végrehajtható. Ezután létrehozzuk az eseményeket arról, hogy az ételek és/vagy italok rendelése megtörtént. Az aggregátum tudja a számla állapotát és arra csak neki van szüksége, hogy a parancsok végrehajtásakor tudja ellenőrizni azt. Az eseményeknek nem kell tudniuk a számla állapotáról, mert eseményt csak akkor hozunk létre, ha a számla nyitott. Ha visszajátsszuk az eseményeket, akkor lesz egy BillOpened esemény a DrinkOrdered és a FoodOrdered események előtt.

## Italok felszolgálása

Következő feladat az italok felszolgálása. A generált esemény az EventProduced, a parancs pedig MarkDrinksServed. Az esemény és a parancs definiálása egyértelmű:

public class DrinksServed

{

public Guid Id;

public List<int> MenuNumbers;

}

public class MarkDrinksServed

{

public Guid Id;

public List<int> MenuNumbers;

}

Ez után az esemény és parancs kezelőt kell megírni. Itt arra kell figyelni, hogy csak olyan italokat lehet felszolgálni, amit megrendeltek és nem lett még felszolgálva. Ehhez először letároljuk azt, hogy milyen italokat rendeltek meg. Ezért fel kell venni egy változót és módosítani kell a DrinksOrdered esemény kezelőjét:

private List<int> outstandingDrinks = new List<int>();

public void Apply(DrinksOrdered e)

{

outstandingDrinks.AddRange(e.Items.Select(i => i.MenuNumber));

}

Ellenőrizni kell, hogy minden felszolgálandó italt megrendeltek:

private bool AreDrinksOutstanding(List<int> menuNumbers)

{

var curOutstanding = new List<int>(outstandingDrinks);

foreach (var num in menuNumbers)

{

if (curOutstanding.Contains(num))

{

curOutstanding.Remove(num);

}

else

{

return false;

}

}

return true;

}

A parancskezelő ellenőrzi, hogy minden ital meg lett-e rendelve. Ha nem, hibát dobunk, ha igen, akkor eseményt generálunk:

public class DrinksNotOutstanding : Exception

{

}

public IEnumerable Handle(MarkDrinksServed c)

{

if (!AreDrinksOutstanding(c.MenuNumbers))

{

throw new DrinksNotOutstanding();

}

yield return new DrinksServed

{

Id = c.Id,

MenuNumbers = c.MenuNumbers,

};

}

Most már csak a felszolgált italokat kell levenni a listáról, ezt az eseménykezelőben tehetjük meg:

public void Apply(DrinksServed e)

{

foreach (var num in e.MenuNumbers)

{

outstandingDrinks.Remove(num);

}

}

## Ételek felszolgálása

Ételek felszolgálása hasonló az italok felszolgálásához. Kell egy lista a megrendelt étel számának tárolására és egy másik lista az elkészített étel számának tárolására:

private List<int> outstandingFood = new List<int>();

private List<int> preparedFood = new List<int>();

A rendelések felvételénél étel rendelése esetén egy FoodOrdered eseményt generáltunk, most annak a kezelőjét írjuk meg:

public void Apply(FoodOrdered e)

{

outstandingFood.AddRange(e.Items.Select(i => i.MenuNumber));

}

Az étel elkészültét egy FoodPrepared esemény létrejötte jelzi. A későbbiekben elkészítjük a szakács feladatlistáját, ott lehet majd bejelölni, ha egy étel elkészül. Most létrehozzuk a parancsot, amivel megjelöljük az elkészített ételeket:

public class MarkFoodPrepared

{

public Guid Id;

public List<int> MenuNumbers;

}

Kell egy kivételt dobni akkor, amikor olyan ételt akarnak elkészültnek jelölni, ami nem lett megrendelve:

public class FoodNotOutstanding : Exception

{

}

És készítünk egy segítő metódust, ami ellenőrzi, hogy minden elkészített étel meg lett rendelve:

private bool IsFoodOutstanding(List<int> menuNumbers)

{

return AreAllInList(want: menuNumbers, have: outstandingFood);

}

private static bool AreAllInList(List<int> want, List<int> have)

{

var curHave = new List<int>(have.Select(i => i));

foreach (var num in want)

if (curHave.Contains(num))

curHave.Remove(num);

else

return false;

return true;

}

(Módosíthatjuk az IsDrinksOutstanding metódust is, hogy az AreAllInList metódust használja.)

Most a FoodPrepared eseményt és annak kezelőjét írjuk meg:

public class FoodPrepared

{

public Guid Id;

public List<int> MenuNumbers;

}

illetve

public void Apply(FoodPrepared e)

{

foreach (var num in e.MenuNumbers)

{

var item = outstandingFood.First(f => f == num);

outstandingFood.Remove(item);

preparedFood.Add(item);

}

}

Megírjuk a MarkFoodServed parancsot és annak kezelőjét:

public class MarkFoodServed

{

public Guid Id;

public List<int> MenuNumbers;

}

private bool IsFoodPrepared(List<int> menuNumbers)

{

return AreAllInList(want: menuNumbers, have: preparedFood);

}

public class FoodNotPrepared : Exception

{

}

public IEnumerable Handle(MarkFoodServed c)

{

if (!IsFoodPrepared(c.MenuNumbers))

throw new FoodNotPrepared();

yield return new FoodServed

{

Id = c.Id,

MenuNumbers = c.MenuNumbers

};

}

Ez után lehet megírni a FoodServed eseményt és annak kezelőjét:

public class FoodServed

{

public Guid Id;

public List<int> MenuNumbers;

}

és

public void Apply(FoodServed e)

{

foreach (var num in e.MenuNumbers)

{

var item = preparedFood.First(f => f == num);

preparedFood.Remove(item);

}

}

Az esemény kezelőjében kivesszük az elkészült ételek listájából az ételt.

Elkészítjük a teszteket (ld. forráskód).

## Számla lezárása

Az események és a parancsok időnként megegyeznek, de gyakran előfordul, hogy az események több információt tartalmaznak, hogy az aggregátum fel tudja építeni az állapotot.

A számla zárása esemény a következő:

public class BillClosed

{

public Guid Id;

public decimal AmountPaid;

public decimal OrderValue;

public decimal TipValue;

}

Tartalmazza a fizetett összeget, a rendelés értékét, és a borravalót.

A parancs azonban csak a fizetett összeget tartalmazza:

public class CloseBill

{

public Guid Id;

public decimal AmountPaid;

}

Fontos megjegyezni, hogy az események a domain-nel vannak kapcsolatban, míg a parancsok a felhasználóval. Az eseménynek tartalmaznia kell minden adatot, ami szükséges az állapot újraépítéséhez.

Az aggregátum jelenleg nem tartalmazza a felszolgált tételek árát. Az árak a PlaceOrder paranccsal érkeznek, de csak akkor adjuk a számlához az összeget, amikor felszolgálják az egyes tételeket. Módosítani kell az OrderedItem osztályt:

public class OrderedItem

{

public int MenuNumber;

public string Description;

public bool IsDrink;

public decimal Price;

}

Jelenleg a megrendelt tételek menü számát tartjuk nyilván, de szükségük lesz az árra is, tegyük a listába a teljes OrderedItem osztályt:

private List<OrderedItem> outstandingDrinks = new List<OrderedItem>();

private List<OrderedItem> outstandingFood = new List<OrderedItem>();

private List<OrderedItem> preparedFood = new List<OrderedItem>();

Módosítani kell több helyen is a kódot emiatt:

public void Apply(DrinksOrdered e)

{

outstandingDrinks.AddRange(e.Items);

}

public void Apply(FoodOrdered e)

{

outstandingFood.AddRange(e.Items);

}

public void Apply(FoodPrepared e)

{

foreach (var num in e.MenuNumbers)

{

var item = outstandingFood.First(f => f.MenuNumber == num);

outstandingFood.Remove(item);

preparedFood.Add(item);

}

}

public void Apply(FoodServed e)

{

foreach (var num in e.MenuNumbers)

{

var item = preparedFood.First(f => f.MenuNumber == num);

preparedFood.Remove(item);

}

}

private static bool AreAllInList(List<int> want, List<OrderedItem> have)

{

var curHave = new List<int>(have.Select(i => i.MenuNumber));

foreach (var num in want)

if (curHave.Contains(num))

curHave.Remove(num);

else

return false;

return true;

}

public void Apply(DrinksServed e)

{

foreach (var num in e.MenuNumbers)

{

var item = outstandingDrinks.First(d => d.MenuNumber == num);

outstandingDrinks.Remove(item);

}

}

Szükségünk lesz egy változóra a felszolgált tételek árának tárolására:

private decimal servedItemsValue = 0M;

A DrinksServed és FoodServed események kezelőjében kell módosítani ezt a változót:

public void Apply(DrinksServed e)

{

foreach (var num in e.MenuNumbers)

{

var item = outstandingDrinks.First(d => d.MenuNumber == num);

outstandingDrinks.Remove(item);

servedItemsValue += item.Price;

}

}

public void Apply(FoodServed e)

{

foreach (var num in e.MenuNumbers)

{

var item = preparedFood.First(f => f.MenuNumber == num);

preparedFood.Remove(item);

servedItemsValue += item.Price;

}

}

Végül elkészítjük a számlazárás parancsot és az esemény kezelőjét, illetve néhány segítő metódust, kivételt:

public class TabHasUnservedItems : Exception

{

}

public class MustPayEnough : Exception

{

}

private bool HasUnservedItems()

{

return outstandingDrinks.Any() || outstandingFood.Any() || preparedFood.Any();

}

public IEnumerable Handle(CloseBill c)

{

if (!open)

throw new BillNotOpen();

if (HasUnservedItems())

throw new TabHasUnservedItems();

if (c.AmountPaid < servedItemsValue)

throw new MustPayEnough();

yield return new BillClosed

{

Id = c.Id,

AmountPaid = c.AmountPaid,

OrderValue = servedItemsValue,

TipValue = c.AmountPaid - servedItemsValue

};

}

public void Apply(BillClosed e)

{

open = false;

}

# Part 3: Olvasási modellek

Az olvasási modellek a lekérdezésekről szólnak. Az előző fejezetben épített domain logika parancsokat fogadott a külvilágból. A parancsok küldése void művelet: vagy végrehajtódik a parancs vagy visszautasításra kerül és kivétel dobódik.

Az olvasási modell egy olyan modell, ami az olvasásokra, vagyis lekérdezésekre specializálódik. A domain által generált eseményeket fogad és olyan modelleket épít belőlük, ami segít a kliens lekérdezéseinek megválaszolásában. Különböző aggregátumokból, de akár különböző típusú aggregátumokból érkező eseményeket is fogadhat. Felépíthet és fenntarthat például:

* Adatmodellt a memóriában
* Relációs adatmodellt RDBMS-ben tárolva
* Dokumentumokat, dokumentum adatbázisban tárolva
* Gráf adatmodellt gráf adatbázisban tárolva
* Keresési indexet

Olvasási modellenként kell eldönteni, hogy hogyan tároljuk ezeket az adatokat, lehet akár különbözőképpen minden olvasási modell számára. Akár több modell is felépíthető ugyanazokból az eseményekből. Az események a normalizált része a rendszenek, míg az olvasási modellek denormalizált formában tárolják az adatokat.

## Milyen olvasási modellekre van szükségünk?

A pincéreknek látniuk kell:

* A nyitott számlák listáját
* Adott asztalhoz tartozó számlát az egyes tételek állapotával
* A felszolgálandó italok listáját
* Az elkészített és felszolgálható ételek listáját

A szakácsoknak látniuk kell:

* Az elkészítendő ételek listáját

A pénztárosnak látnia kell:

* Az asztalhoz tartozó számlát és a fizetendő összeget

## A szakács todo listája

A szakács todo listájának kezeléséhez két eseményre kell reagálnunk:

* FoodOrdered, ami elkészítendő ételt ad a todo listához
* FoodPrepared, ami leveszi a listáról az elkészített ételt

Létrehozunk egy új projektet. Az olvasási modelleket osztályokként hozzuk létre, amik megvalósítják az ISubscribeTo generikus interfészt:

public class ChefTodoList :

ISubscribeTo<FoodOrdered>,

ISubscribeTo<FoodPrepared>

{

public void Handle(FoodOrdered e)

{

}

public void Handle(FoodPrepared e)

{

}

}

Felvehetnénk minden elkészítendő ételt egy listára, azonban az egyszerre megrendelt ételeket egyszerre szokás felszolgálni, ezért létrehozunk egy csoportot az egyszerre megrendelt ételeknek, ezért deklarálunk két DTO-t:

public class TodoListItem

{

public int MenuNumber;

public string Description;

}

public class TodoListGroup

{

public Guid Bill;

public List<TodoListItem> Items;

}

Az olvasási modell TodoListGroup-ok listáját fogja kezelni:

private List<TodoListGroup> todoList = new List<TodoListGroup>();

Mivel a todo listához érkezhetnek kérések konkurensen, figyelni kell, hogy thread safe legyen a megvalósítás. Tovább figyeljünk arra, hogy a belső listáinkat ne adjuk ki:

public List<TodoListGroup> GetTodoList()

{

lock (todoList)

{

return (from grp in todoList

select new TodoListGroup

{

Bill = grp.Bill,

Items = new List<TodoListItem>(grp.Items)

}).ToList();

}

}

Figyelni kell a lockolásra, mert a memóriában egy módosítható modellt tartunk. Ha adatbázisba írnánk és onnan olvasnánk, akkor nem kellene ezzel foglalkozni, megoldaná ezt helyettünk az adatbázis kezelő.

Fel kell építeni a todo listát. Ha új rendelés érkezik, fel kell venni a tételeket a listánkra:

public void Handle(FoodOrdered e)

{

var group = new TodoListGroup

{

Bill = e.Id,

Items = new List<TodoListItem>(

e.Items.Select(i => new TodoListItem

{

MenuNumber = i.MenuNumber,

Description = i.Description,

}))

};

lock(todoList)

{

todoList.Add(group);

}

}

Ha elkészül egy étel, levesszük az elkészítendő ételek listájáról:

public void Handle(FoodPrepared e)

{

lock(todoList)

{

var group = todoList.First(g => g.Bill == e.Id);

foreach (var num in e.MenuNumbers)

{

group.Items.Remove(group.Items.First(i => i.MenuNumber == num));

}

if (group.Items.Count == 0)

{

todoList.Remove(group);

}

}

}

## A pincérek todo listája

Fel kell iratkozni az összes eseményre, mivel minden esemény hatással van a listára. Létrehozunk néhány DTO-t is, amik szükségesek a lekérdezések megírásához.

public class BillItem

{

public int MenuNumber;

public string Description;

public decimal Price;

}

public class BillStatus

{

public Guid BillId;

public int TableNumber;

public List<BillItem> ToServe;

public List<BillItem> InPreparation;

public List<BillItem> Served;

}

public class BillInvoice

{

public Guid BillId;

public int TableNumber;

public List<BillItem> Items;

public decimal Total;

public bool HasUnservedItems;

}

private class Bill

{

public int TableNumber;

public string Waiter;

public List<BillItem> ToServe;

public List<BillItem> InPreparation;

public List<BillItem> Served;

}

Szükségünk van egy lekérdezésre, ami az olyan asztalok számának a listáját adja vissza, amelyekhez tartozó számlák meg vannak nyitva. Ehhez kell egy lista:

private Dictionary<Guid, Bill> todoByBill = new Dictionary<Guid, Bill>();

Ez után meg kell csinálni a BillOpened esemény kezelőjét. Minden megnyitott számlát felveszünk a listánkra:

public void Handle(BillOpened e)

{

lock (todoByBill)

todoByBill.Add(e.Id, new Bill

{

TableNumber = e.TableNumber,

Waiter = e.Waiter,

ToServe = new List<BillItem>(),

InPreparation = new List<BillItem>(),

Served = new List<BillItem>()

});

}

A számla zárásakor le kell venni a listáról a lezárt számlát:

public void Handle(BillClosed e)

{

lock (todoByBill)

todoByBill.Remove(e.Id);

}

Végül megírhatjuk a lekérdezést:

public List<int> ActiveTableNumbers()

{

lock (todoByBill)

return (from bill in todoByBill

select bill.Value.TableNumber

).OrderBy(i => i).ToList();

}

A következő lekérdezésünk egy pincér todo listáját adja vissza.

Ehhez készítünk egy segéd metódust, ami átmásolja egy lista elemeit egy másik listába. A segítő metódusokat nem írom a tutorialba, mert nem tartoznak szorosan a témához.

Ez után elkészítjük a lekérdezést.

public Dictionary<int, List<BillItem>> TodoListForWaiter(string waiter)

{

lock (todoByBill)

return (from tab in todoByBill

where tab.Value.Waiter == waiter

select new

{

TableNumber = tab.Value.TableNumber,

ToServe = CopyItems(tab.Value, t => t.ToServe)

})

.Where(t => t.ToServe.Count > 0)

.ToDictionary(k => k.TableNumber, v => v.ToServe);

}

Meg kell csinálni az italrendelés esemény kezelőjét. Ebben az esemény kezelőben kell letárolnunk a megrendelt italokat a megfelelő számlához.

public void Handle(DrinksOrdered e)

{

AddItems(e.Id,

e.Items.Select(drink => new BillItem

{

MenuNumber = drink.MenuNumber,

Description = drink.Description,

Price = drink.Price

}),

t => t.ToServe);

}

Étel rendelésekor hasonlóan kell eljárnunk, ott viszont az elkészítendő ételek listájához kell hozzáadni a megrendelt ételeket.

public void Handle(FoodOrdered e)

{

AddItems(e.Id,

e.Items.Select(drink => new BillItem

{

MenuNumber = drink.MenuNumber,

Description = drink.Description,

Price = drink.Price

}),

t => t.InPreparation);

}

Amikor az ételek elkészülnek, FoodPrepared esemény generálódik. Az esemény kezelőjében csak annyit kell tennünk, hogy átmozgatjuk az ételeket az egyik listáról a másikra:

public void Handle(FoodPrepared e)

{

MoveItems(e.Id, e.MenuNumbers, t => t.InPreparation, t => t.ToServe);

}

A DrinksServed és FoodServed események kezelője hasonló, át kell mozgatni a felszolgálandó tételek listájából a felszolgált tételek listájába.

public void Handle(DrinksServed e)

{

MoveItems(e.Id, e.MenuNumbers, t => t.ToServe, t => t.Served);

}

public void Handle(FoodServed e)

{

MoveItems(e.Id, e.MenuNumbers, t => t.ToServe, t => t.Served);

}

Már csak a lekérdezéseket kell megírni. Kíváncsiak vagyunk arra, hogy egy adott asztalhoz melyik számla tartozik. Csak meg kell keresni a listában az adott asztalt és visszaadni a kulcsot, ami a számla száma:

public Guid BillIdForTable(int table)

{

lock (todoByBill)

return (from tab in todoByBill

where tab.Value.TableNumber == table

select tab.Key

).First();

}

Érdekel még minket egy adott asztalhoz tartozó számla a különböző tételekkel:

public BillStatus BillForTable(int table)

{

lock (todoByBill)

{

return (from bill in todoByBill

where bill.Value.TableNumber == table

select new BillStatus

{

BillId = bill.Key,

TableNumber = bill.Value.TableNumber,

ToServe = CopyItems(bill.Value, t => t.ToServe),

InPreparation = CopyItems(bill.Value, t => t.InPreparation),

Served = CopyItems(bill.Value, t => t.Served)

})

.First();

}

}

Illetve még adott asztal számláját kell elkészítenünk, ami tartalmazza a felszolgált ételeket és italokat valamint a fizetendő összeget és azt, hogy mindent felszolgáltak-e:

public BillInvoice InvoiceForTable(int table)

{

KeyValuePair<Guid, Bill> bill;

lock (todoByBill)

bill = todoByBill.First(t => t.Value.TableNumber == table);

lock (bill.Value)

return new BillInvoice

{

BillId = bill.Key,

TableNumber = bill.Value.TableNumber,

Items = new List<BillItem>(bill.Value.Served),

Total = bill.Value.Served.Sum(i => i.Price),

HasUnservedItems = bill.Value.InPreparation.Any() || bill.Value.ToServe.Any()

};

}