1. Domain driven design چیست؟
   1. تعریف ساده اش اینه: شیوه ای برای توسعه سیستم های پیچیده. منظور اینه که ما یه سری مسئله سمت طراحی و ایجاد سیستم مورد نظرمون داریم که میخوایم اینا رو برطرف کنیم. DDD میاد و یه سری مفاهیم، راهنما و دستورالعمل معرفی می کنه که بر اساس اونا  
      code base مونو ایجاد کنیم.
2. منظور از domain اصلا چیه؟
   1. کلیتش اینه: یه موضوعی وجود داره که حول محور اون میخوایم یه برنامه بنویسیم. Domain میاد و میگه که اطلاعات و دانش مرتبط با اون موضوع و منطق برنامه (application logic) به چه صورته؛ اینکه چه چیز هایی داریم، چه اتفاقاتی میفته، چه اتفاقاتی ممکنه بیفته، در مواجهه با هر اتفاق چه تصمیمی اتخاذ کنیم و...
   2. یه مثال میزنیم: فرض کنیم یه برنامه meet داریم. یه سری افراد دارن داخل این برنامه یه جلسه یا میتینگ برگزار می کنن. هر کس که حضور داره میتونه یه وضعیتی داشته باشه مثل online, disconnect, early left, …. توی هر کدوم از این وضعیت ها ما میتونیم دقیق مشخص کنیم که اگه شخص قرار بگیره، چه اتفاقاتی باید بیفته. حالا مثلا early left رو در نظر بگیریم؛ طرف تا چه مدت زمانی از میتینگ بیرون بره، ما توی این وضعیت نگه اش می داریم؟ یه دیقه، دو دیقه، ده دیقه یا هر مقدار دیگه ای. این ویژگی و تصمیمی که براش می گیریم، همه در حیطه domain هستن.
3. مزایای استفاده از DDD چیه؟
   1. یه متدولوژی ثابت شده اس در طراحی سیستم های پیچیده به خاطر دستور العمل ها و راهنمایی های گام به گام اش در طراحی و پیاده سازی سیستم ها.
   2. در طولانی مدت، سیستم ما انعطاف پذیری بیشتری داره؛ راحتتر میتونیم تغییرات رو بهش اضافه یا اعمال کنیم.
   3. از معماری هایی مثل میکروسرویس یا سیستم های توزیع شده پشتیبانی می کنه (microservice and distributed systems)
   4. تست نویسی به مراتب راحتتره (نداشتن وابستگی های infrastructure و مشخص بودن مرز بین قسمت های مختلف)
   5. چون قسمت های مختلف domain رو جدا و هر بخش رو مجزا هندل می کنه، نگهداری (maintenance) پروژه راحتتره.
   6. گرچه در ابتدا به خاطر پیچیدگی و نداشتن پایه اولیه پروژه، پیاده سازی و توسعه طولانی تر و سختتره اما در طولانی مدت ساده تر و کم هزینه تر میشه.
4. 2 بخش اصلی DDD چی هستن؟
   1. Strategic design: این بخش گرچه به نسبت بخش بعدی کمتر به چشم میاد، اما به شدت مهمه. تو این بخش ما domain رو بررسی می کنیم و سعی میکنیم بفهمیم subDomain های ما چیا هستن، چه ارتباطی با هم دیگه دارن، چرا اصلا وجود دارن (در واقع میخوایم بدونیم هر subDomain چه نقش کلیدی ای رو داره بازی میکنه که بودنش رو واجب میکنه)، هر subDomain رو کجا قرار میدیم (ممکنه چندتاشون با هم داخل یه برنامه باشن و یک یا چندتا داخل یه برنامه دیگه)، میزان اهمیت و هزینه توسعه هر subDomain و... . به طور کلی، میپرسیم که چه چیزهایی داخل domain ما هستن و چرایی وجودشون؛ اینکه چه مشکلی رو دارن برای ما حل می کنن که به خاطرش بخوایم اینا رو داشته باشیم. در نهایت، هر subDomain باید به یه logical self-contained component تبدیل بشه که در درون خودش، داره مسائل مربوط به اون subDomain رو حل می کنه و با subDomain های دیگه هم در ارتباطه.
   2. Tactical design: اینجا حالا با "چطور" سر و کار داریم. چطور یه subDomain کار میکنه؟ چطور با subDomain ها ارتباط برقرار میکنه؟ چطور میخواد مسائل رو حل کنه؟ جواب این سوال ها رو توی این بخش میدیم که بیشتر با مفاهیم کدنویسی و فریمورک هاش سروکار داره.
5. منطور از ubiquitous language در DDD چیه؟
   1. یه زبان مشترک درباره domain که توسط همه افراد تیم توسعه دهنده استفاده میشه تا بتونن به صورت واضح و پایدار (consistent) بر طبق اون با هم دیگه در ارتباط باشن. منظور اینه که یه سری تعاریف و کلیات هست که وقتی کسی درباره اونا صحبت میکنه، نیازی به رمزگشایی و توضیح نیست و همه متوجه میشن منظور چیه.
   2. این زبان بسته به domain یا subDomain میتونه فرق کنه؛ ممکنه یه مفهوم داخل یه subDomain به یه شکل تعریف بشه و معنی بده و در یه subDomain دیگه تعریف و معنی دیگه ای داشته باشه. اما در هر حال، وقتی مشخص باشه که اون مفهوم کجا مطرح داره میشه، طرف مقابل متوجه میشه که تعریف و معنی مورد نظر، کدوم هست.
   3. این زبان فقط به کدنویس ها محدود نمیشه و افرادی مثل تحلیلگر هم با این زبان مشترک دارن درباره domain صحبت می کنن.
6. Invariant ها چی هستن؟
   1. اینا Business rule ها هستن که همیشه باید برقرار باشن. مثلا توی همون اپ میتینگ، قرار میذاریم که تعداد افراد شرکت کننده داخل یه میت نباید بیشتر از 30 تا باشه (حالا به هر دلیلی) یا هر میت نباید بیشتر از 60 دقیقه طول بکشه. اینا یه سری قواعد هستن که قرار نیست شکسته بشن و همیشه برقرار هستن (یکم شبیه به validation هایی هستن که به طور عادی روی انتیتی ها میذاریم).
7. منظور از Enforcing invariants چیه؟
   1. ما داخل domain مون یه سری انتیتی (domain model) داریم. Invariant ها هم که business rule ها هستن. Enforcing invariant یعنی ما بیایم این business rule ها رو روی انتیتی هامون اجبار کنیم؛ در واقع همیشه انتیتی ها تحت هر شرایطی از اون invariant ها تبعیت کنن. حالا این کار رو به دو صورت میشه انجام داد: یا یه جایی خارج از خود انتیتی این اجبار انجام بشه یا اینکه داخل خود انتیتی این اجبار اتفاق بیفته.   
      در حالت اول این invariant ها (که domain logic هم بهشون میگیم) مستقیما داخل انتیتی نمیشینن بلکه جایی خارج از کلاس اون انتیتی داره این domain logic پیاده سازی و enforce میشه.  
      در حالت دوم (که DDD میگه این رو باید لحاظ کنیم) ما داخل کلاس انتیتی این  
      domain logic ها رو پیاده سازی می کنیم. در واقع domain logic میره داخل domain model و نه خارج از اون.
8. Result pattern چیه؟
   1. ما یه زمانی هست که میخوایم اعلام کنیم که چیزی موفقیت آمیز بوده یا نبوده. اگه بوده که صرفا اعلام اوکی بودن می کنیم اما اگه نبوده، باید اینم بگیم که چرا نبوده.  
      با این pattern، ما میتونیم هر دو نوع موفقیت آمیز بودن و نبودن رو به عنوان یه نوع تعریف کنیم در حالی که خود موفقیت (Success) و عدم موفقیت (Error) هر کدوم شئ جدا باشن.  
      اینطوری انگار همیشه داریم برای مثلا یه متد، یه مدل خروجی میدیم.
   2. این pattern برای خروجی API ها و تست نویسی خیلی مناسبه.
9. Tactical pattern؟
   1. قبلا درباره tactical design توضیح دادیم، اینم ادامه همون میشه. به این صورت که پایه و اساس نحوه انجام و پیاده سازی پروژه رو با tactical pattern انجام میدیم. بخش های مختلف tactical pattern که پایه های نحوه پیاده سازی ان (درباره هر کدوم مفصل صحبت می کنیم) میشن:
      1. Entities
      2. Value objects
      3. Aggregates
      4. Domain services
      5. Factories
      6. Repositories
      7. Domain events
10. Entity چیست؟
    1. کلاسی که ویژگی های زیر رو داشته باشه میتونه یه entity باشه:
       1. Identifier: چیزی که باهاش بشه هر نمونه از اون کلاس رو شناسایی کرد (معمولا فیلد ID توی کلاس اگه باشه، identifier ما اون به حساب میاد)
       2. Equality: دو شئ از اون کلاس بر چه اساسی با هم برابر میشن؟ چه فیلد هایی هستن که با مقایسه اونا دو شئ برابر میشن یا نه و یا چه منطقی به کار میره برای سنجش برابری دو شئ از یک کلاس؟ این میشه بحث equality که کلاسی که میخواد entity باشه، باید بتونه برابری دو شئ از خودش رو تعیین کنه.  
          به طور عام، برابری با فیلد ID سنجیده میشه و حتی اگه دو شئ از اون کلاس باقی فیلد هاشون فرق کنه، صرف اینکه ID های برابر داشته باشن از نطر ما برابر به حساب میان. از این رو معمولا base class ای که برای entity میسازیم، داخلش متد Equals به طور پیش فرض override شده و داره ID دو شئ از کلاس رو با هم مقایسه میکنه. اگه لازم باشه، توی خود کلاس میتونیم این متد رو مجددا override کنیم.  
          مشابه این کار رو برای متد GetHashCode هم انجام میدیم.
11. Value object چیست؟
    1. اینجا ما یک یا مجموعه ای از پراپرتی ها رو داریم که در کنار هم یه معنی بزرگ تری رو میدن. مثلا کلاس Email با یه پراپرتی Address که صرفا داره مشخص میکنه این پراپرتی منظورش آدرس ایمیله؛ حالا چرا به جای اینکه یه پراپرتی string به اسم Email داشته باشیم، خود Email رو یه کلاس جدا تعریف میکنیم؟ دلیل اینه که ما یه سری کار ها داریم که مخصوص مفهوم Email هست (مثلا اینکه باید از یه فرمت خاصی پیروی کنه). به جای اینکه بیایم این کار ها رو ببریم داخل کلاسی که میخواد یه پراپرتی برای Email داشته باشه، کلا مفهوم Email رو تبدیل به کلاس می کنیم و رفتار های مختص خودشو داخل خودش تعریف می کنیم.  
       یا مثلا مفهوم TimeRange که میتونه از دو پراپرتی از نوع DateTime به اسم های StartDate و EndDate تشکیل بشه. حالا رفتار هایی مثل همیشه بزرگتر بودن EndDate از StartDate یا بدست آوردن فاصله بین شروع و انتهای یه TimeRange، همه و همه میتونن داخل یه کلاس باشن که نوع TimeRange رو با این دو پراپرتی و تمام رفتار های مربوط به خودش، هندل میکنه
    2. احتمالا با این اوصاف سوال پیش میاد که تفاوتش با Entity چیه که جواب اینه که ValueObject دیگه Identifier نداره. از طرفی، یه ValueObject مستقلا استفاده نمیشه و خودش جزئی از یه Entity هست.
    3. یه ویژگی دیگه شون اینه که Immutable هستن؛ یعنی مثلا ساختن یه شئ از یه ValueObject با مثلا new کردن اش اتفاق نمی افته بلکه هر گونه تغییر یا تاثیری که قراره روی اون ValueObject گذاشته بشه، از طریق متد هایی هست که داخلش می نویسیم و اجازه دسترسی از بیرون میدیم. مثلا آپدیت کردن یه شئ از یه ValueObject دیگه با تغییر دادن پراپرتی هاش امکان پذیر نیست و باید متد Update ای براش وجود داشته باشه و اونو کال کنیم.
12. Aggregate چیست؟
    1. یه تعریف انگلیسی هس که هرچی زور میزنم فارسیش در نمیاد خودشو میذارم:  
       One or more domain objects that always need to stay consistent as a whole  
       منظور چیه؟ ما دو مدل domain object داریم: Entity , Value object. اما یه سری وقتا هست که یک یا چندتا از اینا رو یه جا لازم داریم اما این یه جا، خودش مستقیما یه Entity یا value object نیست.
    2. با یه مثال توضیح میدیم: فرض کنیم داریم یه میتینگ ست می کنیم. اینجا entity ای به اسم Reservation رو داریم. حالا در ظاهر باید یه انتیتی دیگه به اسم Session بسازیم که داخل خودش داره لیستی از Reservation رو نگه میداره اما در اصل Session بیشتر از اینا داره کار انجام میده؛ داخلش علاوه بر List<Reservation>، برگزار کننده و حداکثر تعداد شرکت کنندگان رو هم نگه میداره. پس در اصل خودش از انتیتی های دیگه تشکیل میشه.
    3. از طرفی رفتاری که Session بروز میده هم مهمه. اگه یه Session داریم، همیشه در یه وضعیت valid هست. مهم نیست که عملیات اضافه کردن یه نفر به میتینگ موفقیت آمیزه یا نه، خود میتینگ قبل و بعد از این عملیات بوده و همچنان پابرجاس (وضعیت valid) و این داره اون بخش consistent توی تعریف رو توضیح میده.
    4. برای اینکه مشخص باشه که چه انتیتی هایی از نوع Aggragate هستن، یه base class برای Aggregate ها می سازیم که هر Aggregate ای ازش ارث بری کنه. این کلاس معمولا خودش داره از Entity ارث بری می کنه اما برای ما مشخص می کنه که کلاسی که داره ازش ارث بری می کنه، یه Aggregate هست.
13. Domain service ها چی هستن؟
    1. یه سری سرویس هایی هستن که برای domain مون می نویسیم. اینا قراره که  
        business logic و business rule ها رو پوشش بدن منتهی سوال پیش میاد که مگه ما اینا رو داخل domain object هامون نذاشتیم؟ پس چرا باید به همچین چیزی نیاز داشته باشیم؟  
       جواب اینه که در اصل فایده domain service ها اینه که گرچه برای پوشش دادن business rule & logic به کار میرن اما در واقع اون هایی رو پوشش میدن که مستقیما نمیشه داخل domain object ها گذاشت. در اصل وقتی که داریم قواعدمون رو پیاده سازی می کنیم، احتمالش هست که یه سری هاشون رو نشه مستقیما داخل domain object ها نوشت؛ به جاش میایم داخل domain service ها می نویسیم
    2. به ندرت پیش میاد که به domain service ها نیاز پیدا کنیم. معمولا اینطوریه که اگه حس کردیم بهشون نیاز داریم، یه مقدار لازمه فکر کنیم که آیا واقعا نیاز هست یه domain service تعریف کنیم یا با کمی تغییر توی domain object ها میشه اونجا هندلش کرد؟  
       اما به این معنی نیس که اصلا بهشون نیاز پیدا نمیکنیم.
14. Factory ها چی هستن؟
    1. به طور ساده، مسئول ایجاد اشیا پیچیده هستن. وقتی یه کلاس رو داریم که ایجاد کردن شئ ازش، پیچیده میشه و کار های زیادی باید قبل از ساختش انجام بشه یا validation هایی پاس بشه، میتونیم بیایم یه کلاس یا متد Factory تعریف کنیم و ساخت شئ از اون کلاس رو ببریم داخل این متد.
    2. اینکار علاوه بر اینکه منطق ساخت شئ رو ایزوله میکنه (و دیگه هیچجوره نمیشه غیر از اون متد، شئ از کلاس مورد نظر ساخت)، به سادگی و خوانایی کد هم اضافه میکنه؛ ما دیگه لازم نیست داخل خود کلاس اصلی به خاطر کد زیاد، سردرگم بشیم. در ضمن به ما امکان استفاده از Result pattern رو هم میده که به این درد میخوره که اگه در حین ساخت شئ از کلاس مورد نظر، به هر دلیلی خطایی پیش اومد، کلا شئ ای ساخته نشه و یه Error پاس داده بشه به جای Exception raising. اگر هم شئ ساخته شد، خودش پاس داده میشه.
    3. معمولا یه static method داخل همون کلاس هستن که بهتره constructor کلاس رو private تعریف کنیم تا کلا کسی نتونه به غیر از این متد، جور دیگه ای شئ از کلاس بسازه.
15. Repository ها چی هستن؟
    1. یه پکیج از سرویس هایی هست که از بیرون میشه روی domain object ها (توی این مورد، بیشتر انتیتی ها و Aggragate ها) اعمالشون کرد. یکم گیج کننده اس نه؟  
       ما اگه لازم داشته باشیم مثلا یه رکورد از کلاس رو بر اساس Id اش پیدا کنیم و بیاریم، میتونیم کدی که قراره اینکارو انجام بده ببریم داخل repository و بذاریم داخل یه متد مثل GetById. یا اگه نیاز باشه که یه رکورد ما آپدیت بشه، میتونیم کدشو مثل GetById ببریم داخل repository و یه متدش کنیم.
    2. این پکیج که به بیرون از domain object اجازه دسترسی و تغییر و... رو روش میده، میگیم repository.  
       به طور معمول repository ها عملیات های CRUD رو روی domain object ها هندل میکنن.
    3. مشابه factory، اینکار تغییر دادن روی domain object ما رو به متد ها و شیوه ای که ما صلاح میدونیم، محدود میکنه و دیگه کسی نمیتونه خارج از این چارچوب روی domain object عمل کنه.
16. Disconnected domain model ها چی هستن؟
    1. وقتی که داخل یه aggregate نیازه که با یه انتیتی یا aggregate دیگه ای کار کنیم و از repository اون انتیتی یا aggregate برای ارتباطش با aggregate مورد نظر استفاده می کنیم، بهش گفته میشه disconnected domain model. چرا که دیگه خود اون انتیتی یا aggregate داخل domain استفاده نشده. چنین اتفاقی یه زنگ خطره به این معنی که احتمالا طراحی ما مشکل و نیاز به بازبینی داره.
17. Eventual consistency چیه؟
    1. در حالت عادی، توی یه برنامه وقتی یه درخواست سمت ما میاد، کل مراحلش توی یه بخش انجام میشه. در eventual consistency این مراحل پخش میشن و هر کدوم دست domain مخصوص خودشون میرن که انجام بشه و این پخش شدن از طریق event هاس. یه مثال بزنیم: تو همون بحث میتینگ که بالاتر گفتیم، اگه شخصی بخواد به میتینگ اضافه بشه، باید domain مربوط به میتینگ (مثلا Meet) چک کنه که آیا ظرفیت تکمیله یا نه، آیا این شخص اجازه اضافه شدن به میت رو داره یا نه و در کل مواردی که سمت domain مربوط به میتینگ باید چک بشه. از اونور باید بررسی بشه که مثلا ID که از شخص گرفته شده درسته یا نه، شخص آنلاین هست یا نه و مواردی از این دست که مربوط به domain کاربر میشه. حالا اتفاقی که میفته اینه که هر بخش رو به domain خودش پاس میدیم که هندلش کنه و برای اینکار از event ها استفاده می کنیم.
18. Domain event ها چی هستن؟
    1. یه تعریف باحال براش داریم:  
       یه اتفاق قابل اهمیت (از دید بیزنس) که در سیستم به وقوع ***پیوسته است.***
    2. خب اینکه یه تعریف خیلی کلی هست. اما در واقعیت این تعریف داره این ماجرا رو پوشش میده. داستان اینه که یه سری کار ها هستن که برای ما مهمه وقتی که انجام میشن و به اتمام میرسن، بعدش یه کار دیگه ای انجام بشه. در واقع یه سری عملیات داریم که اتفاق افتادنشون به شرط اینه که یه عملیات دیگه ای قبلا به اتمام رسیده باشه. پس هر وقت که عملیات اصلی اتفاق افتاد و تموم شد، یه event رو raise می کنه که خبر بده آقا من کارم تموم شده حالا اونی که میخواسته بدونه، خبردار بشه تا کار خودشو انجام بده.
    3. عملیاتی که میخواد بعد از اصلیه انجام بشه اما داخل domain مورد نظر هست معمولا. یعنی یه domain ای در واقع هست که اونه که میخواد خبردار بشه و یه کار دیگه ای رو انجام بده در نتیجه تعریف این event و پیاده سازیش میفته داخل domain مورد نظر و از این رو میشه یه domain event.
19. یه نکته مهم اینه که زمانی ما میایم یه property اضافه می کنیم که متوجه میشیم نیازش داریم. اینکار از این جهت مهم و نسبتا بدون مشکل هست که همیشه داریم هر آنچه که یه domain نیاز داره رو داخل خودش هندل می کنیم. پس زمانی که یه پراپرتی جدید اضافه میشه، هر چیزی که نیازه به خاطر اون ایجاد بشه همون موقع ایجاد یا اضافه میشه (چه برای همون domain object چه domain object هایی که به طریقی دارن باهاش کار میکنن) و نگرانی بابت اینکه چیزی به مشکل بخوره نداریم.
20. منظور از Analysis Paralysis چیه؟
    1. تعریف domain model ها در DDD بدون تعارف بخوایم بگیم، کار سختی هست. دلیلش اینه که دائما مدل ها در حال تغییر هستن و هرچیزی رو همون لحظه اضافه میکنیم. طراحی مدل ها به صورتی که دقیقا همون چیزی که لازم داریم باشن، به شدت سخت و زمان بر هست چون باید تمام سناریو های لازم رو براشون بررسی کنیم و طبق اون ها طراحی مدل رو انجام بدیم. مواقع زیادی پیش میاد که یه طراحی رو فکر می کنیم که درسته اما کمی جلوتر که میریم میبینیم که ایراداتی داره و باید طور دیگه ای طراحی صورت بگیره. پس ما همیشه در یه دور تکرارشونده و طاقت فرسای آنالیز و بررسی domain هستیم که بهترین خروجی ممکن رو در اون زمان به ما بده.
21. منظور از Identity uniqueness چیه؟
    1. توی هر Aggregate، ما یه Entity ای رو داریم. هر دوی اینا هم یه Identifier مجزا دارن. این Identifier ها باید یکتا باشن منتهی مهمه که این یکتا بودن نسبت به چی هست. برای Aggregate، اینطوریه که باید در کل برنامه، Identifier یه Aggregate یکتا باشه. برای entity اما اینطوریه که اگه نسبت به Aggregate ای که این Entity داخلش قرار داره، یکتا باشه هم کفایت میکنه (البته entity هم میتونه مثل Aggregate در سطح کل برنامه دارای Identifier یکتا باشه). پس ما میتونیم داخل 2 تا Aggregate با Identifier های متفاوت، Entity های با identifier یکسان داشته باشیم.
22. قاعده delete کردن یه domain object در DDD چیه؟
    1. فرض کنیم یه domain object ای داریم که با یک یا چندتا domain object دیگه رابطه (relation) داره. در حالت عادی اینطوریه که وقتی یه رکورد پاک میشه، به خاطر relation موجود در دیتابیس، رکورد وابسته هم پاک میشه. توی DDD اما همچین کاری رو نمی کنیم؛ چرا که نمیخوایم بین domain object ها وابستگی داشته باشیم (درسته که relation بینشون برقراره اما قرار نیست که قواعد دیتابیس به جای قواعد DDD قرار بگیرن). پس به جاش میایم از طریق event ها این وابستگی رو هندل می کنیم؛ وقتی یه domain object رو delete می کنیم، یه event ای raise میشه و هر کسی که باید خبردار بشه (مثلا domain object های وابسته) اون event رو consume میکنن و طبق اون کار delete سمت خودشون رو انجام میدن. پس دیگه وابستگی دیتابیسی ای در کار نیست. (البته لازمه که زمان config کردن domain object ها سمت دیتابیس، توسط ORM این مورد رو لحاظ کنیم)
23. چرا aggregate ها رو بهتره تا حد امکان کوچیک طراحی کنیم؟
    1. دیتای کمتری سمت دیتابیس واکشی میشه
    2. احتمال اینکه 2 تا درخواست روی یه aggregate همزمان بخوان کاری انجام بدن کمتر میشه (race condition نداریم)
24. چرا توی هر transaction بهتره فقط یه aggregate باشه؟
    1. این یکم توضیح میخواد. فرض کنیم توی مثال میتینگ، یه میتینگ برای یه روز و ساعت خاص ست شده باشه. اگه قرار باشه که این میتینگ کنسل بشه، اول باید از لیست میتینگ ها حذف بشه، بعدش از لیست رزرو ها حذف بشه و در نهایت از schedule افرادی که رزرو اش کردن هم حذف بشه. خب در این حالت ما داریم روی 3 تا domain model مختلف اثر میذاریم با اینکه در اصل درخواست سمت یه domain model اومده و هر سه این کار ها یا کامل باید انجام بشن یا هیچکدوم پس با یه transaction طرف هستیم. منتهی به جای اینکه دونه دونه این کار ها رو توی یه بدنه انجام بدیم، با کمک از event ها میایم عمل مربوط به هر domain model رو سمت خودش میفرستیم تا هندلش کنه اما یه wrapper هم دور درخواستمون میذاریم که اگه هر جایی از این مراحل (چه delete سمت domain model اصلی چه اونایی که دارن تاثیر میگیرن) به خطا خوردیم، کل این کار ها رو به حالت قبل برگردونیم و ارور سمت کاربر بفرستیم. اینطوری عملا مثل این میمونه که توی هر درخواست، انگار با یه aggregate کار می کنیم اما در واقعیت با اون هایی که وابستگی بهش دارن هم کار میکنیم.
25. نحوه تعریف وابستگی بین aggregate ها چطوریه؟
    1. Aggregate ها از طریق Identifier هاشون به همدیگه relation میزنن.
    2. داخل یه entity نباید مستقیم به entity یا aggregate ای رفرنس بدیم (مستقیما پراپرتی از نوع اون entity یا aggregate برای ایجاد relation تعریف کنیم). اگه لازم شد اینکارو کنیم، یا باید اون entity تبدیل به یه aggregate بشه یا اینکه با Identifier entity یا aggregate مورد نظر ارتباط رو ایجاد کنیم.
26. تفاوت application services و domain services چیه؟
    1. همون طور که از قبل داشتیم، domain service ها یه سری domain logic ان که مستقیما توی domain object ها نتونستیم پیاده شون کنیم.  
       از اونور اما application service ها در واقع مسئول هماهنگ سازی (orchestration) بین domain object ها و انواع tactical pattern ها (مثل repository, factory و...) هست.
    2. یه تعریف از یه بنده خدایی به نام vladimir khorikov هست که دونستنش جالبه چون به نوعی تفاوت این دوتا رو نشون میده:  
       domain logic ها هرچیزی هستن که به تصمیمات بیزنسی مریوط میشن. از این رو، domain service ها در روند تصمیم گیری به همون صورتی که domain object ها انجامش میدن، مشارکت میکنن. Application service ها هم وظیفه هماهنگ کردن (orchestration) اون تصمیم ها رو دارن به همون صورتی که domain service داره تصمیم های domain object ها رو هماهنگ میکنه.
27. تفاوت transactional consistency و eventual consistency چیه؟
    1. transactional consistency همون طور که قبلا هم گفتیم، وضعیت all or nothing هست. یعنی کل روند یه درخواست از ابتدا تا انتها باید کامل و درست انجام بشه و اگه کوچکترین خطایی اتفاق بیفته، کل کار های انجام شده هم باید برگشت بخوره. یه گیر که داره اینه که کاربر ممکنه زمان زیادی برای دریافت جواب منتطر بمونه چون کل روند باید تا انتها بره و این میتونه زمان بر باشه. اما با این وجود، مزیتش اینه که مطمئن میشیم در چه وضعیتی قرار داریم و موقعیت برامون همیشه مشخص هست در نتیجه اینکار،پیاده سازی این حرکت ساده تره به نسبت eventual consistency.
    2. eventual consistency این هم بالاتر تعریف کردیم و یه جورایی بر عکس transactional consistency هست؛ یعنی ما با اینکه یه سری کار ها باید هنوز انجام بشه، اما با این حال کاربر رو زودتر خبردار میکنیم که این کار کامل انجام شده. پس این کار به نسبت مدل قبلی، سریعتر هست و کاربر متنظر نمیمونه. اما از اون ور ما رو ممکنه توی موقعیت های غیر مطمئن بذاره. مثلا اگه توی میتینگ، کاربر ایجاد کننده میتینگ بیاد و اونو لغو (حذف) کنه، ما بعد از اینکه میتینگ رو حذف کردیم بهش پیام موفقیت آمیز بودن میدیم اما هنوز باید این میتینگ رو از لیست میتینگ های باقی افرادی که داخلش میخواستن شرکت کنن پاک کنیم و بهشون یه نوتیف هم بدیم. در این مورد، با اینکه کاربر سریعا از نتیجه مطلع شده، اما امکان داره به هر دلیلی، قسمت های بعدی به مشکل بخورن و درست اجرا نشن در حالی که ما خبر دادیم کار به درستی صورت گرفته. اینجاس که یه سیستم error handling مناسب باید داشته باشیم تا این اتفاق به کمترین حد خودش برسه.
    3. به صورت کلی، از transactional consistency واسه خود domain object ها استفاده میشه؛ یعنی کاری روی مثلا aggregate ها که میخواد انجام بشه (که خود اون aggregate یا یه domain service داره هندل میکنه)، این به صورت transactional اجرا بشه اما ما بین aggregate ها رو به صورت eventual هندل بشه. اینطوری خود aggregate ها همیشه در وضعیت های مشخص قرار دارن و کافیه که ارتباط بین شون رو به درستی هندل کنیم.
28. تفاوت rich model و anemic model چیه؟
    1. جفتشون برای طراحی domain model ها به کار میرن منتهی نحوه طراحیشون هست که فرق داره. توی anemic، بیشتر اوقات فیلد ها و پراپرتی ها از بیرون قابل دسترسی ان (public getter and setter) و به همین خاطر مستقیما میشه روی خود پراپرتی ها تاثیر گذاشت. در حالی که rich model کاملا برعکسه. شاخصه های یه rich model در ادامه لیست شده:
       1. فیلد ها به صورت پیش فرض private readonly تعریف میشن و فقط در صورت نیاز این سطح دسترسی تغییر میکنه.
       2. فقط اون چیزهایی که واقعا نیاز هست روی model مون انجام بشه رو از بیرون قابل دسترسی می کنیم و وقتی که نیاز باشه.
       3. حواسمون به primitive obsession باشه. منظور اینه که فیلد هایی رو به صورت primitive (مثل int, double, char, bool, datetime) تعریف می کنیم که چیزی بیشتر از اون نیاز ندارن. مثلا اگه لازم باشه مختصات جغرافیایی ذخیره کنیم، به جای گذاشتن 3 تا فیلد double داخل model مون، بهتره که این 3 تا رو داخل یه ValueObject تعریف کنیم و اونو توی model بذاریم تا همیشه از valid بودن مختصات مطمئن باشیم.
       4. باید حواسمون باشه که گاهی ممکنه کاری که داخل application service ها انجام میدیم، در سطح خود domain object مون باشه. پس لازمه که اونو به domain service یا خود domain object منتقل کنیم.
29. منظور از always valid و not always valid چیه؟
    1. این تعریف ها در اصل روی domain model ها اعمال میشه.  
       Always valid domain model یعنی domain model ای که در هر زمان از اجرای برنامه، تمام دیتایی که داخلش هست، valid ئه.  
       در مقابل، not always valid domain model یعنی domain model ای که در زمان هایی از اجرای برنامه، بهش اجازه داده میشه که در وضعیت valid نباشه. مثلا در زمان ساخت شئ از یه domain model، ممکنه همه دیتای مورد نیاز موجود نباشه و این برای ما invalid تلقی بشه. چنین domain model ای معمولا به یه پراپرتی IsValid نیاز داره که در هر زمان بشه مشخص کرد وضعیتش valid ئه یا invalid. موقع ساخت domain model به خودمون بستگی داره که کدوم رو انتخاب می کنیم اما به طور معمول، always valid به not always valid ارجحیت داره.
30. Persistence ignorance یعنی چی؟
    1. زمانی که میخوایم domain model مون رو بسازیم، احتمالا به این فکر میفتیم که سمت دیتابیس اش چیکار کنیم. اینکه دیتابیس اش چی باشه، چطوری دیتا رو داخلش وارد کنیم یا ازش واکشی کنیم و ... . این اصل میگه که کلا این بحث persist کردن دیتا توی دیتابیس و اینا رو در زمان طراحی domain model ات بنداز دور. فقط و فقط روی طراحی خود domain model تمرکز کن و بعدا به این که سمت دیتابیس چیکار کنی فکر کن.
    2. یکی از دلایل استفاده از repository pattern همینه. چرا که وقتی از repository استفاده می کنیم، انگار که برامون مهم نیست که دیتا چطوری و از کجا میاد و چطوری ذخیره، آپدیت یا پاک میشه. فقط میدونیم که دسترسی مون وجود داره و دیگه درگیر مسائل دیتابیسی نمیشیم. خیلی راحت و با تعریف یه سری متد که مشخصه چیکار میکنن، خودمونو از فکر به دیتابیس بی نیاز می کنیم.
31. Strategic design چیه؟
    1. به پروسه ای میگن که در اون یک سیستم با ابعاد و ساختار بزرگ رو تعریف و تبدیل به business domain مون کنیم.  
       بیا مسخره بازی رو بذاریم کنار؛ این یه تعریف آکادمیک و از این خوشگل بازیای کتاب ها و کورس هاس. ساده بخوایم بگیم، میشه فهمیدن و تعریف کردن business domain. اینکه بدونی چی رو میخوای، چرا میخوای، به کجای سیستم ربط داره و موارد این چنینی.
    2. اما چطور انجامش میدیم؟
       1. اول اینکه از متخصص های اون domain کمک بگیر. اصطلاحات، نیازمندی ها، چالش ها، تعاریف و ... رو یه متخصص اون domain هست که میدونه و با کمک اون میتونی ubiquitous language ات رو بسازی.
       2. event storming. این هستش که تمام تصمیم گیرنده ها رو یه جا جمع کنی و یه timeline باهاشون بسازی؛ اینکه هر چیزی مشخص باشه که کی انجام میشه و چه چالشی ممکنه بخوره و ... . در واقع داریم همه افراد تصمیم گیرنده و ذی نفع رو یه جا جمع می کنیم و سعی می کنیم به یه جمع بندی همه جانبه از سوی تمامی این افراد برسیم.
       3. اینکار رفت و برگشتی هست. به دلایلی مثل تجربه، تغییرات در حین زمان، عدم دانش کافی یا اشراف به تمامی جنبه ها و ... این اتفاق میفته که ما با یه بار انجام Strategic design نمیتونیم کار این بخش رو تموم کنیم و در طول زمان مجددا به این بخش برمیگردیم. این یه چیز طبیعی توی DDD هستش و باید همیشه در نظر داشته باشیم که نیازه بین strategic و tactical design دائما سوییچ کنیم.
32. Sub-domain ها چی هستن؟
    1. زمانی که داریم مون domain رو تعریف و طراحی می کنیم، در ظاهر فقط یه تک مفهوم هست. اما در واقعیت وقتی بیشتر دقت می کنیم میبینیم که درون اون domain های دیگری هم موجوده. مثلا اگه domain میتینگ رو در نظر بگیریم، در وهله اول به نظر میاد که صرفا یه رزرو و شرکت در میتینگ هست اما در واقعیت علاوه بر اون ما بحث پرداخت هزینه شرکت در میتینگ، پنل مدیریت کاربر، مدیریت میتینگ و... رو هم داریم. اینا هر کدوم به تنهایی یه domain مجزا هستن و کارهای متفاوت از domain اصلی رو انجام میدن اما با این حال برای کاری که میخوایم روی domain اصلی انجام بدیم لازم هستن.
    2. این sub-domain ها خودشون یه domain به حساب میان و تمام قواعدی که قبلا گفتیم رو به صورت جداگانه برای هر کدوم اونا باید پیاده سازی کنیم یا در نظر بگیریم.
33. Sub-domain ها چند نوع هستن؟
    1. Core sub domain: این sub domain، همون مشکلی هست که ما با پروژه مون میخوایم حلش کنیم. در واقع اصل مسئله ای که باهاش طرف هستیم رو اینجا باهاش سروکار داریم.  
       تشخیص این sub domain در زمان strategic design مهمه چون اولویت ها بر اساس این sub domain مشخص میشن. (مثلا برگزاری میتینگ)
    2. Supporting sub domain: این sub domain قسمتی از مشکل بیزنس اصلی رو حل می کنه. یه جورایی قسمتی از خود بیزنس اصلی به حساب میاد اما اصلیه نیست. (مثلا پنل کاربری یا مدیریت میتینگ)
    3. Generic sub domain: این یکی یه مشکل کلی رو حل میکنه. راه تشخیص اش اینه که میتونیم حتی خودمون پیاده اش نکنیم و از راه حل ها یا محصولات موجود که از قبل دیگران توسعه دادن استفاده کنیم. مزیتش اینه که انرژی و وقتمون رو صرف دو sub domain دیگه می کنیم و کمتر درگیر این یکی میشیم. (مثلا پرداخت هزینه شرکت در میتینگ)
34. چرا اصلا sub domain داریم؟
    1. یه بیزنس رو راه انداختن، هزینه هایی داره. چه زمانی چه مالی ما باید سعی کنیم که کمترین هزینه رو بدیم. تعریف sub domain ها باعث میشه که ما بتونیم domain ها و بخش های مختلف بیزنس رو شناسایی و در نتیجه هزینه انجام هر کدوم رو ارزیابی و پیش بینی و در نهایت بین اون ها بر اساس نوع هزینه ها، اولویت بندی کنیم.
35. منظور از context mapping چیه؟
    1. توی domain مون یه سری context داریم که تقریبا همون sub domain ها میشن و خب طبق قبل، اینا از همدیگه کاملا جداسازی شدن و مرزبندی هاشون مشخصه.  
       Domain object های ما ممکنه توی هر کدوم از این context ها حضور داشته باشن اما معنی و مفهومشون توی هر کدوم متفاوته. مثلا کاربر (user) توی مدیریت کاربری میشه عملا خود تعریفی که برای یه کاربر داریم و با تمام تعاریفش. همین user توی پرداخت هم حضور داره اما تعریفش در اونجا اینه که یه چیزی هست که دارای یه سری پرداختی، فیش، سبد خرید و ... هست.
    2. Context mapping میشه ایجاد ارتباط بین context هامون که هر domain object چه معنی و مفهومی توی هر کدوم داره و چطوری بینشون جابجا میشه.
    3. <https://github.com/ddd-crew/context-mapping>
36. انواع team relationship ها در رابطه با context ها چیا هستن؟
    1. یه توضیح ریز قبل از اینکه بریم سراغ انواع. معمولا اینطوریه که روی هر context یه تیم جدا کار میکنه (ممکنه اینطور هم نباشه و یه تیم روی بیشتر از یه context کار کنه). این وسط لازمه بدونیم که تغییرات یه context چه تاثیری روی یه context دیگه میذاره که در واقع داره نحوه ارتباط بین تیم ها رو هم مشخص میکنه. انواع شون ایناس:
    2. Mutually dependent: توی این حالت هر دو context ما به هم دیگه وابسته ان و هر کدوم یه تغییری کنه، اون یکی هم باید تغییراتی رو به نسبت اون اعمال کنه.
    3. Free: توی این حالت دو context ما هیچ وابستگی ای ندارن و تغییر هیچ کدوم اون یکی رو تغییر نمیده
    4. Upstream-downstream: اینجا یکی از context ها وابسته اس و تغییر اونی که مستقله روی وابسته تاثیر میذاره اما برعکسش اتفاق نمی افته. اونی که وابسته اس رو بهش downstream و اونی که مستقله رو بهش upstream میگیم.
37. Pattern های مختلف برای context mapping؟
    1. Open-host service: context یه اینترفیس جامع تعریف میکنه که همه از اون استفاده میکنن. مثلا پرداخت که نحوه سرویس دهی "پرداخت" اش به همه به یه صورته و یه api کلی داره.
    2. Conformist: یه downstream context داریم که خودشو با upstream context اش بدون اینکه روی اون تاثیر بذاره، تطبیق میده .
    3. Anti-corruption layer: downstream context یه لایه ترجمه تعریف میکنه تا از model داخل خودش در برابر تغییر از بیرون محافظت کنه.
    4. Shared kernel: دو context از یه زیر مجموعه ی مشترک از domain model ها استفاده میکنن و هر دو صاحب اش و در قبال اش مسئولن.
    5. Partnership: برای وضعیت mutually dependent به کار میره که هر دو context به هم وابسته ان.
    6. Customer-supplier: مثل همون upstream-downstream هست که اینجا میگیم مستقل مون supplier و وابسته مون customer ئه. هر تغییری از سوی supplier روی domain model به وجود بیاد که customer بهش وابسته اس، supplier موظفه که customer رو هم در جریان بذاره.
    7. Published language: context ها از طریق یه سری قرارداد که دقیق و با جزئیات تعریف شده، با هم ارتباط می گیرن. (مثلا message broker ای که با فرمت json دیتا جابجا میکنه)
    8. Separate ways: دو context ما عملا با هم ارتباطی ندارن و راه خودشونو میرن بدون تاثیر روی اون یکی.