@RobertHaken

Software & Cloud Architect | Microsoft MVP: Development | HAVIT, s.r.o.

Havit.Blazor stack

architektura aplikací s Blazor UI, gRPC komunikací a pseudo-DDD aplikační logikou

Havit.Blazor stack

Blazor WebAssembly SPA UI

gRPC komunikace se ASP.NET Core hostem (sdílení .NET interfaces client+server)

Entity Framework Core ORM nad SQL

abstrahovaná datová vrstva (Repositories, Queries, UnitOfWork), psedo-DDD uspořádání

async-first

Azure/on-prem podpora (např. abstrakce nad file-system / Azure Blob Storage)

https://github.com/havit/NewProjectTemplate-Blazor

Kontext

zakázkový vývoj, aplikace na míru

převážně line-of-business/enterprise/e-commerce aplikace

- "databázové aplikace" – obvykle logika kolem jednoho centrálního datového kontextu
- webové UI
- příklady: https://www.havit.cz/reference-webove-aplikace/

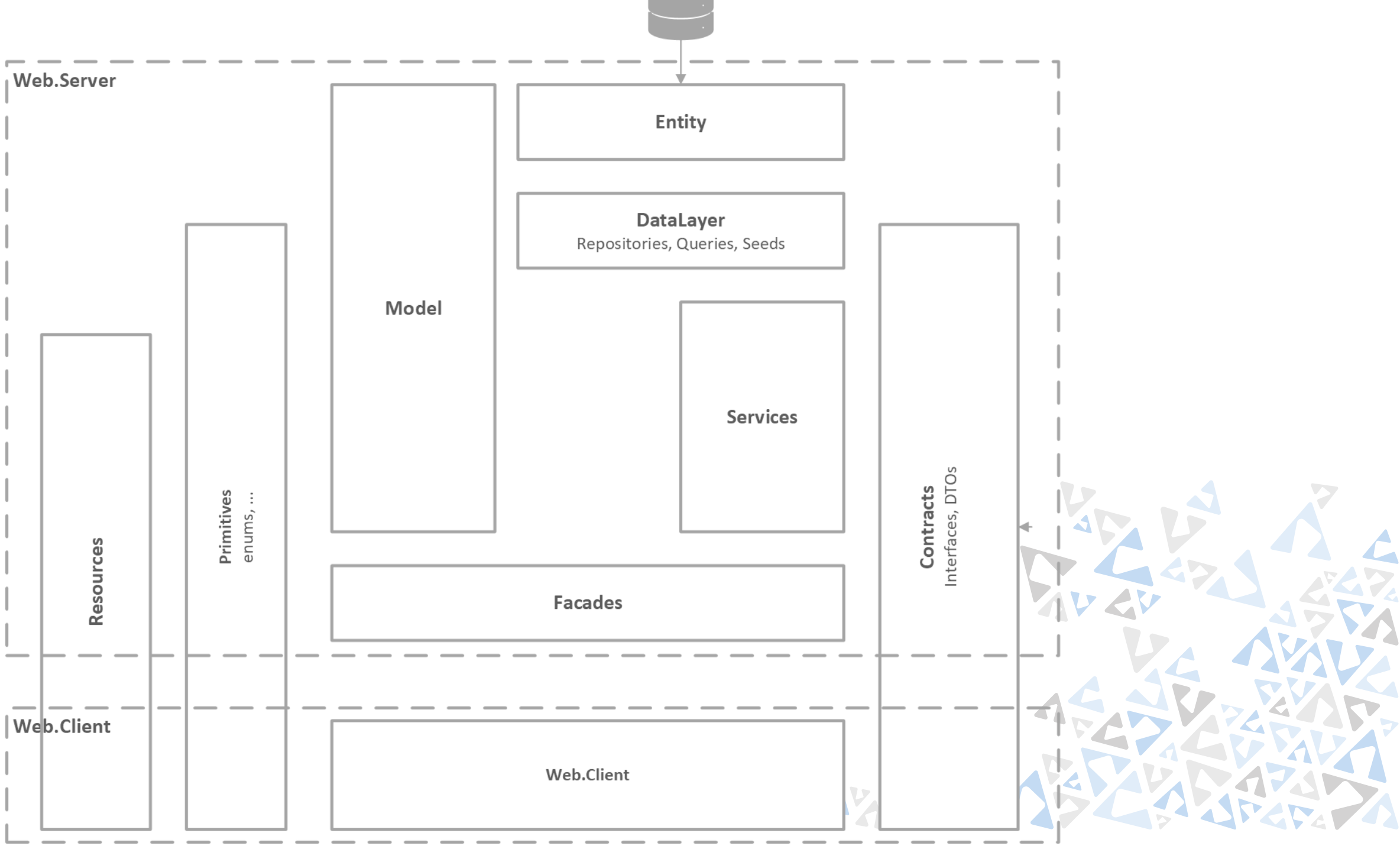
obvykle rozsah 100 – 4000 MDs, ale i více

tým 1-4 vývojáři, tester, project-manager/analytik

agilní principy vývoje (přírůstky funkční i technologické, častý deployment)

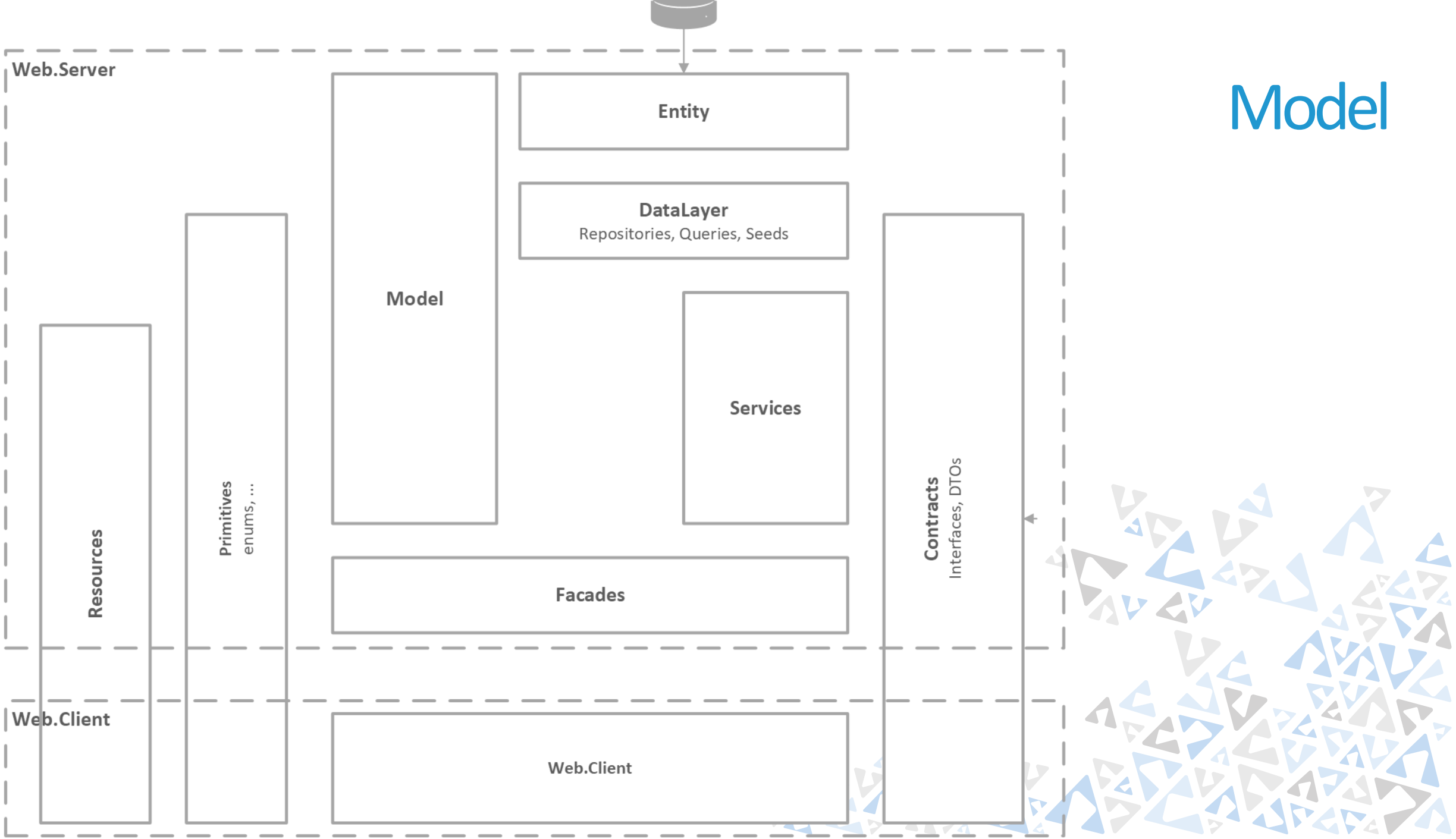
jeden vývojář pracuje na více projektech, důležité sdílené know-how

deployment ze 75% do Azure PaaS (App Service), 25% jinam (on-prem)


Web.Server
Entity
DataLayer
Repositories, Queries, Seeds
Model
Services
Facades
Resources
Primitives
enums, ...
Contracts
Interfaces, DTOs
Web.Client
Web.Client

Model

Model


Web.Server
Entity
DataLayer
Repositories, Queries, Seeds
Model
Services
Resources
Primitives
enums, ...
Facades
Contracts
Interfaces, DTOs
Web.Client
Web.Client

Model

POCO (Plain Old CLR Object, Plain Old Class Object)

- nepoužíváme layer supertype (`EntityBase`)

převážně „anemický“

- nechceme řešit dependency-injection do Modelu, v modelu obsažená logika je tímto limitovaná

plní role Domain Model i Data Model (pro EF Core, viz dále)

- nezastírá, že je modelem pro relační uspořádání a bude 1:1 mapován do tabulek SQL serveru
- EF Core anotace však směřujeme do `IEntityTypeConfiguration<T>` v projektu `Entity`
- necháváme jen atributy, které anotují model samotný (`Required`, `MaxLength`, ...),
  ne jeho perzistenci

Model

konvence pro běžné situace

- `int ID` klíč
- `DateTime Created`
- `DateTime? Deleted` (volitelný soft-delete)
- ale i několik interfaces např. `ILocalized<XyLocalization>, ILocalization<Xy>`

obvykle se vyhýbáme dědičnosti

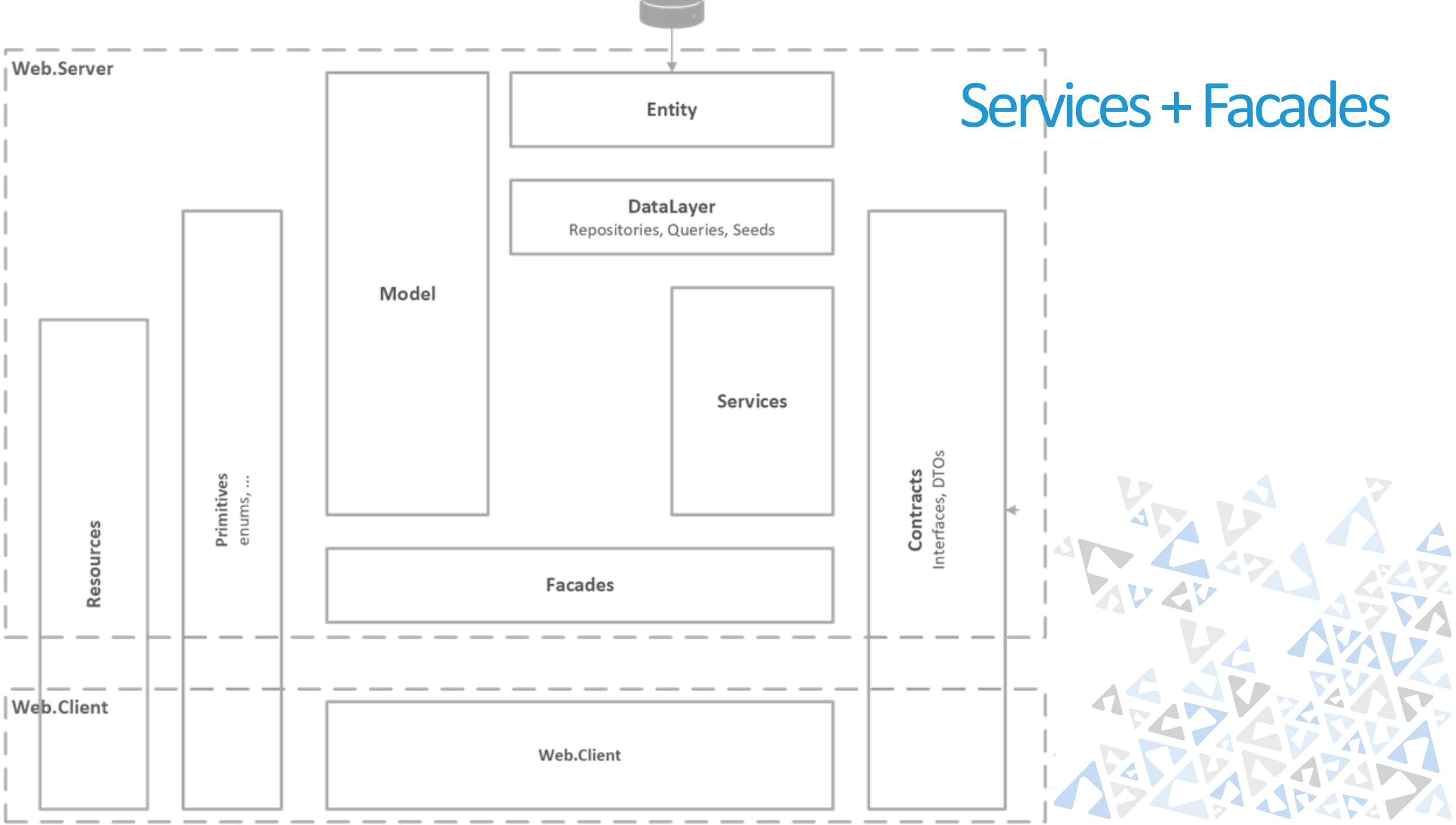
- ve prospěch asociací (FK v DB)

nemá žádné "heavy" závislosti

- ne EF Core, jen lehké anotace

Services + Facades

Services + Facades



Facades

Facades vystavují služby-operace pro vnější vrstvy aplikace (UI, API pro třetí strany, jobs, ...)

- není dovoleno volat přímo Services, Model, DataLayer atp., vše musí jít přes Facades
- Facades se nevolají navzájem (-> extrakce do Services, viz dále)

zpravidla atomické business operace/transakce

- `CreateInvoice()`, `AddPayment()`, `ApproveAbsence()`, `DisableUser()`
  - obvykle jeden `unitOfWork.Commit()`
- `GetInvoiceList()`, `GetContactDetail()`, `GetAbsencesStatistics()`

vstupem a výstupem Facades jsou zásadně DTO

- nikdy nepřijímají ani nevrací objekty z Modelu, vždy se mapuje na DTO

nezaměňovat s *Facade* design pattern (GoF)

Services

doménová a aplikační logika

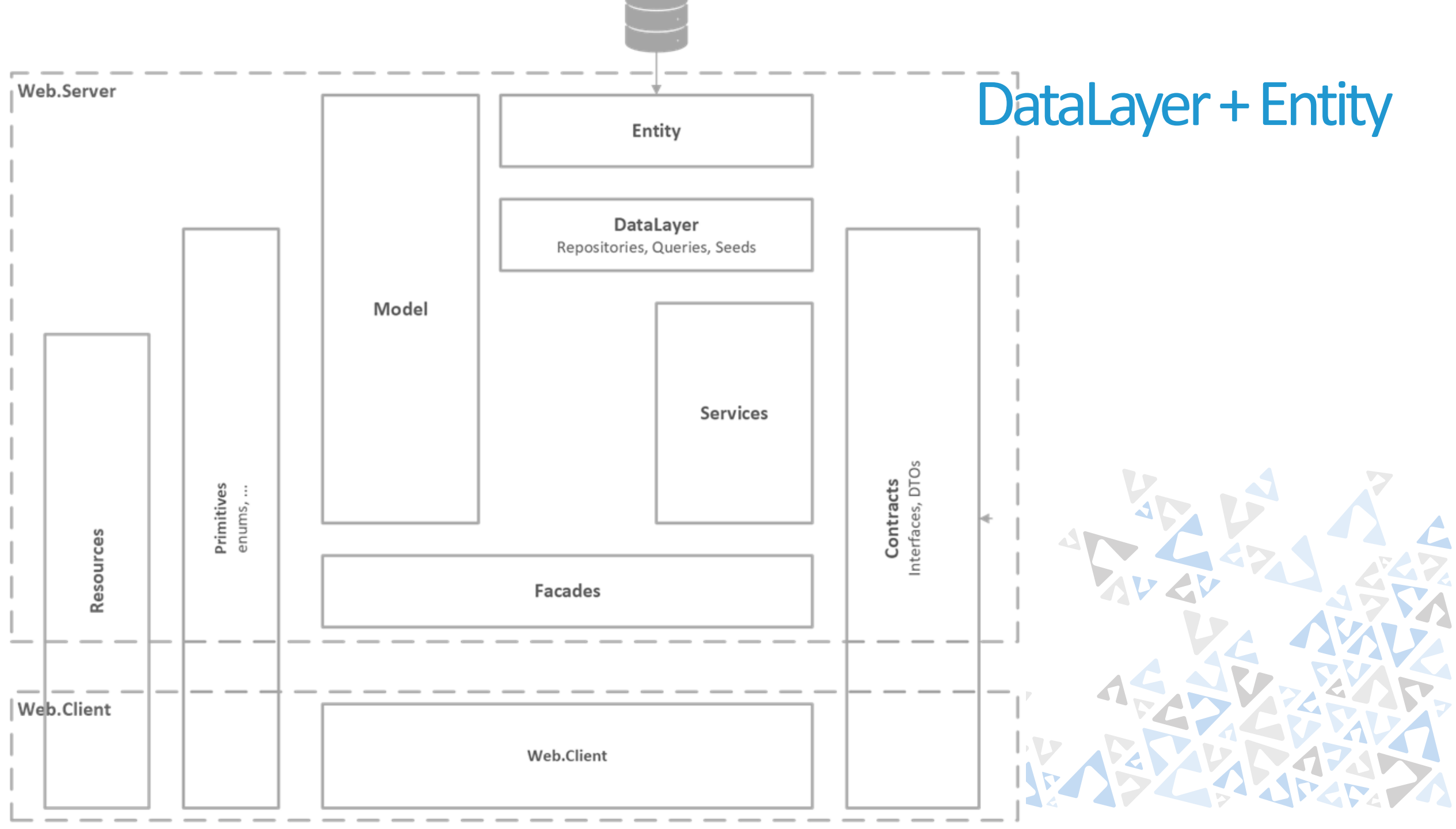
- co není dost triviální (CRUD) pro přímé volání z Facade
- co má být sdíleno mezi více Facade
- co je samostatně testovatelnou unitou

například

- doménové kalkulace všeho druhu (ceny, dovolené, úroky, ...)
- workflow či interpretační logika (schvalování, mailing, ...)
- integrační logika (napojení na externí služby a logika kolem, ...)
- mappery (plain, nepoužíváme AutoMapper ani nic podobného)

DataLayer + Entity

DataLayer + Entity



Entity

mapuje Model na DB pomocí `EntityFrameworkCore` ORM

nikdy se nepoužívá přímo, ale výhradně skrze DataLayer

obsahuje

- jediný aplikační DbContext
- konfigurace entit (`IEntityTypeConfiguration<T>`)
- migrace schématu DB

DataLayer

poskytuje datové služby pro Services a Facades

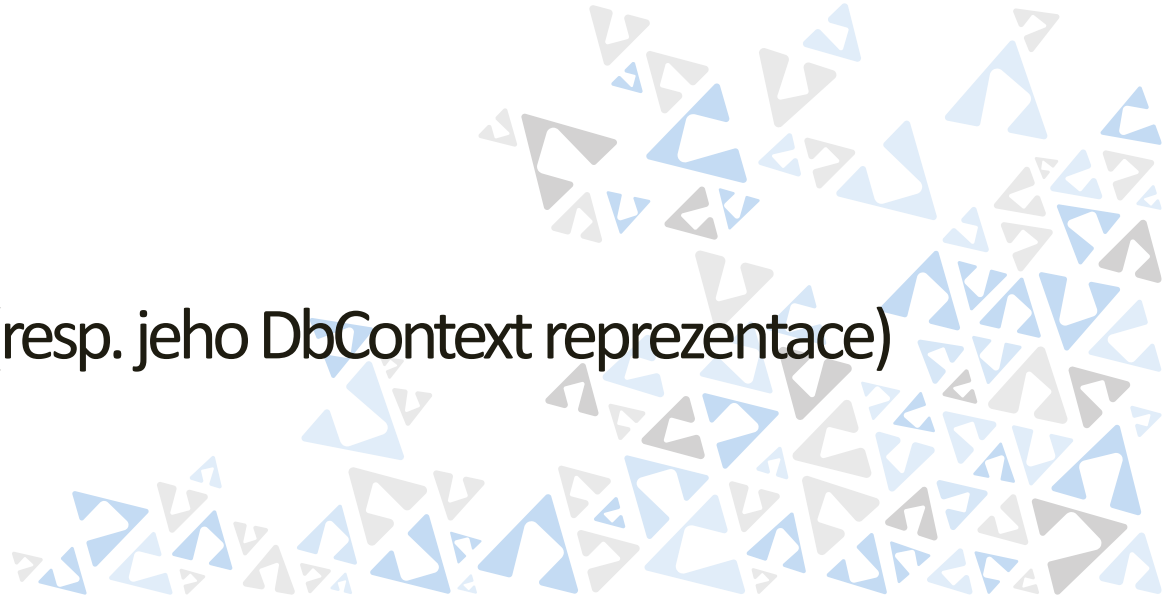
- nikdo jiný nesmí s DataLayerem přímo pracovat
- odstiňuje doménovou a aplikační logiku od fyzického uložení dat (Entity)

obsahuje

- repositories
- queries
- seeds*

hodně generovaného kódu na základě Modelu (resp. jeho DbContext reprezentace)

- repositories, data sources, entries, ...



DataLayer – Repositories

generovaný základ `IXyRepository`

- `GetAll()`, `GetObject(int id)`, `GetObjects(int[] id)`

přes partial interface/class lze rozšiřovat, např.

- `GetByXy(...kritéria...)`
- triviální read-operace nad jednou entitou nebo aggregate-root (složitější viz Queries)

výstupem je obvykle entita z Model (může být i DTO)

podporuje „includes“ (`GetLoadReferences()`)

podporuje soft-deletes (`Data`, `DataIncludingDeleted`)

podporuje cachování (atribut `[Cache]` na entitě v Model)

DataLayer – Queries

bázová třída `QueryBase<TResultItem>`

- `protected abstract IQueryable<TQueryResultItem> Query();`
- `protected Select(), SelectPage(), SelectDataFragment(), Count(), First(), Single(), Any(), ...`

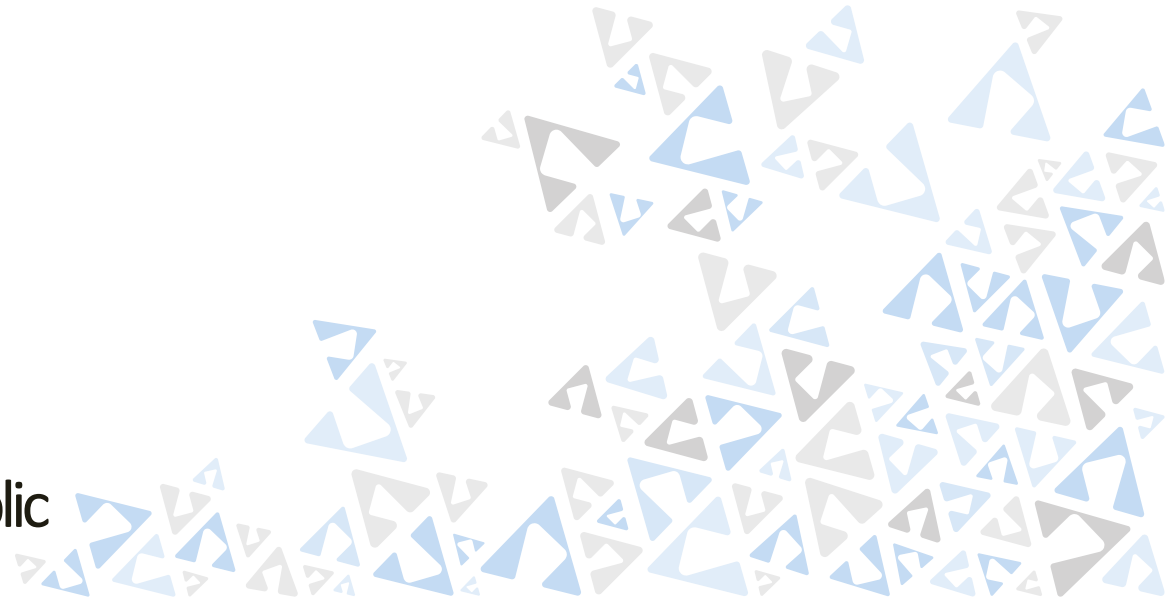
abstrakce `IXyDataSource` namísto `DbSet<TXy>` pro podporu testovatelnosti

- constructor injection

používá se pro netriviální dotazy do DB

- joiny přes více entit (více IXyDataSources)
- komplexnější filtry, nastavitelné řazení
- stránkování dat atp.

programátor sám volí interface, nic není předem public



DataLayer - DataSeeds

vlastní mechanizmus údržby statických dat v DB (číselníky, počáteční konfigurace atp.)

per entity + celé grafy

podporuje

- nastavení párovacích klíčů
- částečné aktualizace, popř. inkrementální seedy
- vzájemnou závislost seedů (`GetPrerequisiteDataSeeds()`)
- seedovací profily (Core, Test, Demo, ...)
- vzájemnou závislost profilů (`GetPrerequisiteProfiles()`)
- historie spuštění seedů a eliminaci opakovaných spuštění (`__DataSeeds` tabulka)

lze spouštět při startu aplikace, při deploymentu, kdykoliv z kódu

Web.Server + Web.Client

Web.Server

ASP.NET Core host (= Startup project)

hostuje

- Web.Client převážně v podobě statických assets – Blazor WebAssembly client
  - host-page v podobě Razor Page (verzované odkazy na JS a CSS, pre-konfigurace)
  - volitelný server prerendering
- Facades v podobě gRPC-Web služeb (a tím celé jádro aplikace – Model, Entity, DataLayer, Services, ...)
- Hangfire dashboard
- podpůrné „serverovřiny“
  - endpointy pro upload a download souborů
  - endpoint pro claims-enrichment Blazor clienta
  - health-checks
  - API pro třetí strany, ...

Web.Client

Blazor WebAssembly SPA

“transparentní” volání serverových Facades

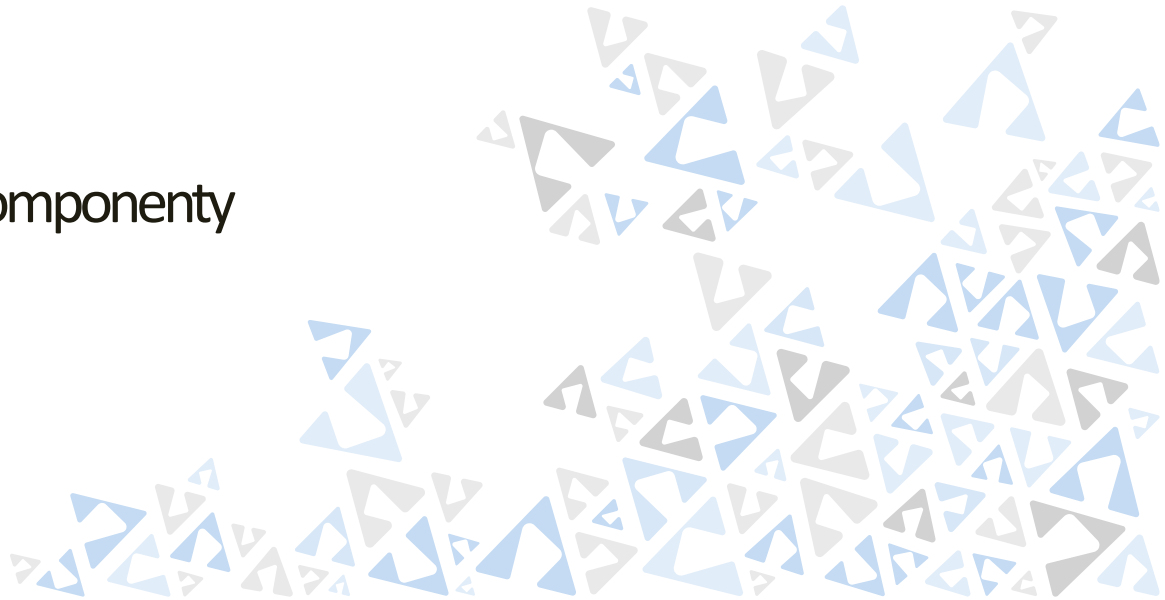
- stačí injectovat IXyFacade přes dependency injection
- gRPC-Web runtime-generated clients (HttpClientFactory)

vlastní komponenty Havit.Blazor.Components.Web.Bootstrap

- https://havit.blazor.eu
- MIT licence, open-source
- Bootstrap 5.2 componenty + vlastní higher-level komponenty

nevylučuje využití dalších knihoven (Telerik atp.)

- pokud se nepoperou s Bootstrapem



Contracts, Primitives, Resources, MigrationTool, Utility, *.Tests

Podpůrné projekty

Contracts, Primitives, Resources

sdílené mezi serverovou a klientskou částí

Contracts

- interfaces a DTOs mezi vrtvami aplikace
- zejména definice Facades a používané in/out DTOs
- přímo konzumuje Web.Client
- DTOs včetně validátorů

Primitives

- enumy sdílené mezi Model a DTOs

Resources

- lokalizace sdílené mezi serverem a klientem
- source-generator pro strong-type dekorátory `IStringLocalizer`

MigrationTool, Utility, *.Tests

MigrationTool

- samostatně spustitelné jako upgrade DB - migrace a seedy dat
- vhodné pro deployment-time upgrade (zejm. víceinstanční aplikace)

Utility

- Hangfire host (scheduled jobs)
- Azure App Service Continuous WebJob

*.Tests

- unit-tests
- IntegrationTests
- TestForLocalDebugging

Vybraná specifika

Vytvoření nové solution z šablony

https://github.com/havit/NewProjectTemplate-Blazor – README.md

vyklonování šablony

- https://github.com/havit/NewProjectTemplate-Blazor/generate (new GitHub repository)
- nebo do nové složky: `git checkout-index --prefix=git-export-dir/ -a`

edit SetupSolution.ps1 and set variables

- Namespace, SolutionName, ports, Mailing options, ...

run SetupSolution.ps1

- plain text replace in files + rename files

delete SetupSolution.ps1

done

Validace

`FluentValidation` library

- na rozdíl od System.ComponentModel.DataAnnotations dobře funguje s dependency-injection

`XyDtoValidator : AbstractValidator<XyDto>` jako nested-class v DTO

Web.Client: `Blazored.FluentValidation` pro Blazor

Authentication, authorization

standardní mechanizmy ASP.NET Core

- fungují transparentně i pro gRPC služby, i pro Blazor klienta
- ClaimsPrincipal, [Authorize], roles, authorization-policies, ...

OpenID Connect (OIDC) pro Blazor SPA

- Azure AD B2B jako výchozí volba
- Azure AD B2C pro public-facing aplikace
- generické OIDC pro 3rd party identity providery (BYOI)
- Duende IdentityServer, kde je vyžadováno "vlastní" řešení (až poslední možnost)

Web.Server: `IClaimsTransformation` enrichment o další claims (role, ...)

Web.Client: `AccountClaimsPrincipalFactory` enrichment o další claims (ze serveru)

Errors, health-monitoring

používáme standardní mechanizmus výjimek

- nepoužíváme `ProblemDetails`, ani podobné "obohacení" všech výstupních DTO (dto.Messages, ...)
- individuální výjimky možné (

`OperationFailedException` (+ potomci)

- recoverable errors
- speciální postavení, specifický handling v UI, nelogují se do chybových stavů aplikace

Azure Application Insights pro kompletní telemetrii

- včetně on-premise deployment
- všechny složky (Web.Server, Web.Client Blazor, Utilities, ...)
- Requests, PageViews, Exceptions, Dependencies, Traces, Custom Events, Performance Counters, ...

Resources

Šablona projektu

https://github.com/havit/NewProjectTemplate-Blazor

Havit.Blazor komponenty pro Bootstrap

https://havit.blazor.eu/

HAVIT YouTube channel

https://www.youtube.com/user/HAVITcz