platformy podporujuce asp .net core

s cim pracuje user prihlasenie - user identity, userdbcontext ...

ako zaradit controller do arei

asp .net core je - open source

co znamena skratka mvc - model, view, controller

na co sluzi model

na co sluzi view

co znamena ked je classa static

na co nam sluzia vrstvy

ako sa oznacuje viac riadkovy kod v razor pages

pomocou akych helperov funguje attributy na validaciu - dataannotations - asi

akym priakzom vytvarame migracie

co je to dependency injection - uz implementacia

otazka na typ dependeny injections - myslim ze spravan odpoved bola singleton, ale otazka bola tak divne polozena

ake typy validacii je mozne pouzit v asp .net core

ako restrictnut controllery pre pristup od uzivatela

nieco aj s autentizaciou

NET Framework

• Jen Windows

• Více funkcionality, více NuGet packages, odladěnější

• Použít, pokud už ve firmě existují aplikace pro .NET Framework

.NET Core

• Multiplatformní: Windows, Linux, Mac

• Open-source

• V ideálním scénáři je více výkonný a škálovatelný

• Použít při vyvíjení Microservices

• Použít při využívání Docker containers

**NET standard**

Není moc vhodné mít více knihoven se třídami, které obsahují to samé,

přičemž rozdíl je jen v platformě, na kterou cílí.

• Toto způsobuje problémy obzvláště, pokud je napsán kód tak, že na jiných platformách

nefunguje správně jen v důsledku chybějících funkcí.

• K vyřešení problémů byl vytvořen .NET Standard. Jedná se o množinu

specifikací, které udávají, jaké funkce lze využít na všech platformách, které

využívají .NET.

• Účel je tedy jednoduchý: sdílet stejný kód mezi platformami .NET.

• .NET Standard je zpětně kompatibilní

• Lze využít pro vrstvu Model

**Architektonické vzory** se používají jako vzory pro rozdělení návrhu aplikace do logických

skupin softwarových komponent, neboli vrstev.

Jednotlivé vrstvy rozdělují odlišnou funkcionalitu komponent.

Návrh architektury aplikace by měl být tedy proveden tak, aby byla podpořena

znovupoužitelnost vrstev.

Vyšší vrstva hraje roli klienta nižším vrstvám. (používá je; něco od nich potřebuje)

Nižší vrstvy poskytují služby vyšším vrstvám. (nabízí funkcionalitu, kterou mohou využít)

Třídy by neměly být závislé na třídě z vyšší vrstvy

Nejrozšířenější architektonické vzory jsou:

• MVC – Model-View-Controller (to je ten pravý vzor, který nás zajímá pro ASP.NET)

• MVP – Model-View-Presenter (to jsou ty slavné Windows Forms)

• MVVM – Model-View-ViewModel (to je ten Xamarin a WPF, které někteří z vás opakují)

Všechny tyto vzory usnadňují vývoj tím, že jsou volně kombinovatelné, relativně snadno

testovatelné a snadno udržovatelné.

**MVC**

Umožňuje vývoj založený na 3 základních prvcích: Model, View, Controller.

• Model obsahuje data pro View

• Controller zpracovává interakce, aktualizuje Model a následně předává Model do View

• View přijímá Model z Controlleru a renderuje UI

• Obsahuje komponenty pro vývoj webových aplikací jako route engine, view engine atd.

• (HTTP = Hypertext Transfer Protocol)

**Model**

Model reprezentuje množinu tříd, které popisují hlavní výpočetní logiku aplikace a definují

způsob práce s daty.

Také definuje pravidla jak mohou být data měněna a jak s nimi lze manipulovat a zpracovávat je.

Data mohou být odsud předána a následně zobrazena ve vrstvě View.

• Udržuje stav systému

• Obsahuje logiku aplikace

• Jedná se o rozhraní pro přístup k datům

• Obsahují informace (data) pro view

• Běžné je mít modely v separátním

assembly/knihovně (zjednodušení sdílení)

• O jejich vytvoření a naplnění daty

se starají controllery s využitím dalších vrstev

**ViewModel v Modelu**

• ViewModel svou podstatou spadá pod Model

• Umožňuje sloučit více Entit Modelu do jedné třídy a zobrazit je ve View

• Manipuluje s ním opět Controller

**View** obsahuje to, co vidí uživatel a s čím může interagovat (formuláře, tlačítka, stránka na webu, apod.). Reprezentuje tedy uživatelské rozhraní, které může být napsáno např. v HTML + C# (Razor pages), XAML (Xamarin, WPF), C# (Windows Forms) aj. Zobrazuje data, která obdrží z nižších vrstev (Model). Neobsahuje výpočetní logiku aplikace

**Controller** přijímá vstup od uživatele (příchozí dotazy, HTTP požadavky) • Manipuluje Modelem, spouští aplikační logiku a volí vhodné View pro výstup. • Controller je prostředníkem mezi vrstvami View a Model/ViewModel -> získává Model/ViewModel naplněný daty a předává jej do View. • Dochází v nich k validaci dat, k požadavkům na získání dat, správě chyb a volání view pro vytvoření výstupu • Více View může používat stejný Controller. • Controller by neměl znát prvky vrstvy View. • Public metody Controlleru se nazývají akční metody (action) obvykle jsou mapovány 1:1 s prvky v UI • Výchozí HTTP metoda pro každou akční metodu je Get (atribut HttpGet)

**Routing**

• Provádí na serveru mapování requestu na odpovídající Controller a Action (popřípadě i Area)

• Jednoduché tvoření ”friendly URL”

**Areas**

• Menší funkční jednotky sdružené do jednoho modulu

• Umožňují organizaci velkých projektů do malých celků

• Umožňují vytváření vlastních Controllerů, View, Modelů v

rámci daných celků

• Využití např.: u e-commerce aplikací pro oddělení business

jednotek – checkout, platby, vyhledávání

• V routingu musí být vždy specifikována area, ve které se má

hledat Controller a Action

• Area musí být uvedena vždy před default

• Controller musí mít atribut [Area]:

**View v MVC**

View obsahuje to, co vidí uživatel a s čím může interagovat (stránka na webu).

Reprezentuje tedy uživatelské rozhraní

Zobrazuje data, která obdrží z nižších vrstev (z Modelu).

View přijímá Model z Controlleru a renderuje UI.

Neobsahuje výpočetní logiku aplikace.

**Razor Pages = View v ASP.NET Core MVC**

• Razor Pages jsou ukládány do souboru s příponou .cshtml

• Razor syntaxe se skládá z:

• Razor kódu

• C# kódu

• HTML tagů

• Razor umožňuje psaní server-side renderovaného kódu do HTML

Stránek

**Blok kódu**

• Bloky kódu v Razoru začínají @ a jsou následovány { … }

• Tyto výrazy nejsou renderovány

• Bloky spolu sdílí proměnné v rámci celé Razor Page

**Typy View v ASP.NET Core MVC**

• Layout View obsahuje společnou část UI

• View obsahuje konkrétní obsah jedné stránky

• Partial View Obsahuje jeden HTML prvek (např.

\_CookieConsentPartial.cshtml)

• HTML definice v samostatném souboru (ve složce Shared)

• Ve View/LayoutView lze Partial View vyrenderovat pomocí Razor

nebo Tagu

**Data ve View**

• Data lze předat do View použitím Modelu (Strongly typed),

ViewData (Weakly typed), ViewBag (Weakly typed).

• Pokud je celá stránka složena z více View, jsou data sdílena

• Model

• Controller vytváří a naplňuje model a předává jej do View

• return View(new ProductViewModel());

• ViewData

• Jedná se o Dictionary, který v sobě obsahuje dynamic objekty

• ViewData["SuperClass"] = new Product();

• ViewBag

• Dynamická property, která nevyžaduje test na null ani přetypování

• ViewBag.SuperClass = "jmeno";

**Tag Helper**

• Kód, který pomáhá přetransformovat HTML tagy při renderování

• Redukují explicitní přechod mezi HTML a C#

• Helpery se zaměřují na rozšíření standardních HTML tagů –

poskytují server-side atributy

Např.: asp-controller=„“ asp-for=””

**Databázový systém**

• Databázový systém se dělí na samotná data (báze dat/databáze)

a na jejich řízení (systém řízení báze dat)

**Systém řízení báze dat**

• DBMS - DataBase Management System je rozhraní mezi aplikačními

programy a uloženými daty

• MS SQL Server, MySQL, MariaDB, SQLite, Oracle, PostgreSQL atd.

• Vlastnosti

• Správa klíčů (primární, cizí apod.)

• Jazyk vyšší úrovně pro práci s daty (SQL, QBE, ...)

• Autentizace a autorizace uživatelů

• Správa transakcí, atomicita jednotlivých příkazů

• Robustnost a zotavitelnost po chybách bez ztráty dat

• Kanály pro hlášení zpráv po úspěšně vykonaných dotazech, chybových hlášek, varování

• Procedury

• Triggery

• Indexy

• atd.

**Jazyk SQL**

• Jazyk pro práci dat v relačních databázích

• Relační databáze – založená na tabulkách a vazbách mezi nimi

• Datové typy – smallint, integer, decimal, numeric, float, char, date, varchar…

Základní typy příkazů:

• Manipulace s daty (Select, Insert, Update, Delete, …)

• Definice dat (Create, Alter, Drop)

• Řízení práv (Grant, Revoke) (GRANT privileges ON object TO user;

REVOKE privileges ON object FROM user;)

• Řízení transakcí (Begin Transaction, Commit, Rollback)

**LINQ – Language INtegrated Query**

• Integrovaný jazyk .NET frameworků od verze 3,5

• Jednotná syntaxe pro přístup k datům bez ohledu na jejich

zdroj (databáze, objekt v paměti, XML)

• Stejný dotaz tedy funguje pro databázi, XML soubor a např. List<T>

• Podobný zápis jako u SQL dotazů

• LINQ dotaz se převádí na lambda výrazy

**Lambda výrazy**

o Jedná se o formu anonymních metod (.NET 3.5 a výše)

o Lze je použít na místě, kde je vstupním parametrem delegát

o Např. v rozšířených metodách kolekce List. (using System.Linq;)

o Zapisují se ve tvaru:

o ( vstupní\_parametry\_oddělené\_čárkou ) => { lambda\_výraz }

o Pozn. závorky je možné vynechat, pokud je výraz jednoduchý.

o Např. (a, b) => a + b;

**Šablona delegáta:** Action

o Action obsahuje referenci na metodu, která provádí akce/operace.

o NEVRACÍ nic a MŮŽE mít více vstupních parametrů (nebo žádný).

o Action pro jeden parametr [6]: public delegate void Action<in T>(T obj)

**Šablona delegáta:** Func

o Func obsahuje funkci, která vrací výsledek operace/výpočtu.

o Měla by mít min. 1 vstup (označeno "in") a přesně jeden výstup ("out")

(nicméně existuje i varianta bez vstupu).

o Func pro dva vstupní parametry:

**Entity Framework Core**

• Framework (NuGet Package)

• Object-Relational Mapper (ORM) – objektově relační mapování

• Umožňuje pracovat s databází za pomoci .NET objektů

• Eliminuje nutnost psaní kódu pro přístup k datům (obzvlášť SQL dotazů)

• Zrychluje tvorbu základních CRUD (create, read, update, delete) operací

• Podporuje hned několik databázových providerů (Database Providers):

• SQL Server

• MySQL

• SQLite (s omezeními)

• PostgreSQL

**Fluent API**

• Výchozím chováním je mapovat třídy na tabulky za použití konvencí v EF

• V případě, že je nutné tyto konvence obejít a mapovat třídy jiným

způsobem.

• Způsoby jako toto provést:

• Pomocí atributů (anotací)

• Pomocí Fluent API.

• Atributy pokrývají jen omezenou část funkcionality Fluent API, takže

existují scénáře, které není možné provést pomocí atributů.

• Fluent API je nejčastěji využíváno v překryté metodě OnModelCreating v

odvozené třídě od DbContext.

**Zahnutí/ignorování typů**

• Zahrnutí typu znamená, že EF má metadata o daném typu a může provádět

čtení/zápis s daným typem

• Všechny typy, které jsou dostupné pomocí DbSet property jsou

automaticky zahrnuty

• Zahrnuty jsou dále typy uvedeny v OnModelCreating

• Zahrnuty jsou typy, které jsou objeveny pomocí rekurze typů již

zahrnutých v modelu

**Migrace**

o Slouží pro synchronizaci dat modelu (entit) a databáze

o Slouží k aktualizování schémata databáze bez ztráty dat (v produkční verzi důležité!)

o K použití je potřeba

• Package Manager Console tools (Microsoft.EntityFrameworkCore.Tools)

• .NET Core CLI tools

o Pro použití migrace je nutné provést následující příkazy:

o dotnet tool install --global dotnet-ef (jen jednou, pokud je třeba)

o Vytvoření migrace:

o Add-Migration <jmeno>

anebo

dotnet ef migrations add InitialCreate

o Update databáze:

o Update-Database

anebo

dotnet ef database update

**SOLID**

• 5 pravidel pro psaní čistého a testovatelného kódu

• Single Responsibility Principle

• Open Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

• Výhody dodržování

• Lepší udržovatelnost kódu

• Lepší škálovatelnost kódu

• Méně chyb v kódu

• Snazší hledání a oprava chyb v kódu

**Single Responsibility Principle**

• SRP říká, že třída má mít odpovědnost za pouze jednu věc a tím jen jeden

důvod ke změně.

• O co se třída vpravo může starat:

• Správa věcí v košíku

• Validace dat (lze řešit i ve validátoru)

• Volat DAL (Data Access Layer)

• Co třída dělá navíc:

• Logování

• Problém v kódu vzniká ve chvíli, kdy je zaveden nový Logger

• Musí dojít k úpravě třídy Cart

**Liskov Substitution Principle**

• LSP říká, že zděděná třída může upravovat, ale nesmí omezovat ani

zakazovat funkcionalitu rodiče

• Zděděná třída by nikdy neměla skrýt implementaci rodiče

• Předek je vždy zobecněním potomka

• Metoda využívající referenci na rodiče může použít metody odvozené třídy bez toho,

aniž by to změnilo její funkci.

• Např. Čtverec nemůže dědit ze třídy Obdélník, protože dochází k porušení

pravidel (omezuje funkce obdélníku), obdélník tak není jeho zobecněním.

(a neplatí to ani naopak!)

• Problém se neprojevuje ve chvíli kompilace, ale za běhu aplikace (při realizaci)

• Čtverec by musel být použitelný v situacích, kdy je použitelný i obdélník

**Dependency Inversion Principle**

• DIP říká, že “vysokoúrovňové moduly” (High Level Modules) by nikdy neměly být

závislé na “nízkoúrovňových modulech” (Low Level Modules), ale na abstrakci

• Všechny služby, které potřebujeme by měla třída dostat zvenčí

• Nevytváříme abstrakci podle implementace, ale implementaci podle abstrakce

• Např. problém v kódu níže vzniká ve chvíli, kdy zavedeme nový logger

**Client vs. Server-Side Validation**

• Server Side Validation

• Validace uživatelských vstupů na straně serveru po odeslání (submit) stránky ke

zpracování

• Využívá se pokud request vyžaduje serverové zdroje (resources) pro validaci.

• Client Side Validation

• Validace uživatelských vstupů na straně klienta ve webovém prohlížeči (Firefox,

Chrome apod.).

• Využívá se pokud request nepotřebuje serverové zdroje (resources) pro validaci.

• Nevyžaduje odeslání stránky ke zpracování.

**Validace na serveru**

• Vstup od uživatele je odeslán na server

• Vstup je validován na serveru za pomocí ASP.NET a

validačních pravidel

• Po validačním procesu na serveru je odeslána odpověď zpátky

klientovi.

• Je naprosto nutné provádět validaci na straně serveru,

protože tím zabráníme situacím, kdy uživatel obejde klientskou

validaci a odešle nežádoucí vstupy, čímž ohrozí data na serveru

• Vhodné pro ověření správnosti dat před uložením do databáze

**Responzivní design**

• Responzivní design znamená, že při návrhu stránky počítáme s

více velikostmi/rozlišeními obrazovky.

• Návrh začíná od nejmenší (mobilní verze) – Mobile-first design

• Malá obrazovka by NEměla mít méně obsahu.

• Nesmíme za uživatele rozhodovat, který obsah nebo funkcionalitu

smí/nesmí použít.

• Pravidlo scrollování ve webové aplikaci:

• Vertikální scrollování je důležité pro prohlížení obsahu

• Horizontální scrollování znamená špatný návrh

• Pro otestování Responzivity lze změnit velikost okna prohlížeče

• Občas může být nutné i obnovení stránky

**Unit (jednotkové) testy**

• Unit testy zjišťují, zda jednotlivé části kódu fungují správně.

• Testované případy by měly být nezávislé na ostatních.

• Izolují se části aplikace od infrastruktury a závislostí.

• Při testování Controlleru, nebo modelu je testována pouze 1 metoda.

• Netestuje se chování závislostí nebo frameworku.

• Unit testy neřeší interakci mezi komponentami (to řeší integrační testy).

• Snaží se zamezit efektu “oprava jedné chyby přinese tři další”

• Umožňují provést změny v kódu bez strachu, že nějaká část přestane

fungovat

• Nikdy se neopravují testy – opravuje se špatný nefunkční kód

**Integrační testy**

• Integrační testy se od unit testů liší v tom, že testují 2 nebo více

komponent dohromady (tzn. jejich integraci).

• Testují v mnohem širší spektrum systému.

• Testují komponentu, která je závislá na dalších komponentách

• Zjišťují, zda interakce komponent pracuje správně

• Často zahrnují i infrastrukturu.

• interakce s databází, se soubory, se sítí, test API, UI, logování atd.

• Testují reálné případy užití (use case)

• Integrační test je např. Celý proces přidání předmětu do košíku,

pokračování v nákupu, volba způsobu dopravy/platby, založení

objednávky …

**Refactoring**

• Změna v systému, která mění interní strukturu kódu, ale

nemá (nesmí) mít vliv na jeho chování.

• Uspořádání písmen v polévce nemá vliv na chuť

• Důvod: Zajištění dodržení SOLID principů, čistý kód

• čistý kód je jednodušší, testovatelnější a levnější na údržbu

**Cloudové platformy a služby**

• Nabízí vytvoření IT infrastruktury spravovanou třetím subjektem.

• Jedná se o platformy pro vývoj, správu a nasazení aplikací.

• Cloudové platformy:

• Microsoft Azure

• Google Cloud Platform

**Cloud Computing**

• Cloud computing znamená poskytnutí výpočetních služeb, včetně serverů

(VM), úložišť, databází, sítí, softwaru, analytických nástrojů a inteligentních

funkcí atd. přes Internet.

• Obvykle se platí jen za služby, které jsou využity

**Výhody Cloud Computingu 1/2**

• Nižší náklady

• eliminuje investiční náklady na nákup hardwaru, softwaru, na vytvoření a

provoz datových center a IT pracovníků spravujících infrastrukturu.

• Rychlost

• i ohromná množství výpočetních prostředků jde zajistit během minut,

• Výkon

• datová centra jsou pravidelně upgradované na nejmodernější generaci

rychlého a efektivního výpočetního hardwaru.

• Globální rozměr

**Nevýhody Cloud Computingu**

**8**

• Data nejsou fyzicky součástí infrastruktury firmy a může tak

teoreticky dojít k úniku osobních a jiných citlivých údajů.

• Závislost na podpoře služeb poskytovatelem cloudu.

• neexistuje možnost plné kontroly

• Náročná změna poskytovatele služeb.

• Vysoké nároky na stabilitu a rychlost Internetového připojení.

• Při případném výpadku připojení to znamená, že se ke službám

nedostaneme a nemůžeme s nimi tedy pracovat.

• Pro velké firmy neznamená finanční úsporu nákladů