VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Aplikovaná informatika

PRÁCE S DATY V .NET/C#

Bakalářská práce

Autor práce: Adam Polívka

Vedoucí práce: Ing. Marek Musil

Jihlava 2026

Vysoká škola polytechnická Jihlava

Tolstého 16, 586 01 Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Adam Polívka**
Studijní program:	Aplikovaná informatika
Garant studijního programu:	Ing. Lenka Kuklišová Pavelková, Ph.D.
Název práce:	**Práce s daty v .NET/C#**
Vedoucí práce:	Ing. Marek Musil
Cíl práce:	Cílem práce je zmapovat a představit možnosti práce s daty v prostředí .NET/C#. Práce bude postavena na průzkumu standardních knihoven. V teoretické části práce budou představeny existující možnosti a doplněny vysvětlujícími ukázkami kódů. Budou diskutovány alternativy data setů a dalších komponent. Praktická část se dále zaměří na implementaci objednávkového systému, kde budou vybrané možnosti využity. Ukázka objednávkového systému bude realizována.

Abstrakt

Bakalářská práce se zabývá možnostmi práce s daty v prostředí .NET/C#. Součástí je jejich přehled podpořený praktickými ukázkami, které slouží i jako reference využití daných prostředků programovacího jazyka. Zaměřuje se na několik témat, jako jsou komponenty určené pro ukládání a práci s daty, přesněji pro komunikaci s databází nebo jiným zdrojem dat. Dále jsou uvedeny grafické komponenty určené pro zobrazení dat. Práce se soubory je také obsahem práce. Další část práce se věnuje tvorbě komplexní ukázky, objednávkového systému. Zde je kladen důraz na využití komponent, které byly vybrány na základě uvedených informací a ukázek. V této ukázce je také kladen důraz na využití tabulkových komponent.

Klíčová slova

DataAdapter; DataGrid; DataGridView; DataSet; DataTable; Objednávky; Práce se soubory; Zdroje dat

Abstract

The bachelor thesis deals with the possibilities of working with data in the .NET/C# environment. It includes an overview of these possibilities, supported by practical examples that also serve as programming language references. The thesis focuses on several topics, such as components designed for storing and working with data, specifically for communication with a database or other data sources. Additionally, graphical components intended for data visualization are presented. Working with files is also covered in the thesis. Second part of the work is dedicated to creating a comprehensive example, an order management system. Here, emphasis is placed on the use of components selected based on the previously provided information and examples. In this example, emphasis is also placed on the use of tabular components.

Keywords

DataAdapter; DataGrid; DataGridView; DataSet; Data sources; DataTable; Orders; Working with files

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Byl/a jsem seznámen/a s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Podle § 47b zákona o vysokých školách souhlasím se zveřejněním své práce podle Směrnice pro vedení, vypracování a zveřejňování závěrečných prací na VŠPJ, a to bez ohledu na výsledek obhajoby.

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mě požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutím licence.

V Jihlavě dne 8. dubna 2026

…….

Podpis studenta/ky

Poděkování

Děkuji zejména vedoucímu bakalářské práce Ing. Marku Musilovi za odborné vedení této práce a všem ostatním, kteří mě podporovali během jejího vypracování.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

ADO.NET	ActiveX Data Objects .NET
API	Application programming interface
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSV	Comma Separated Values
ERD	Entity Relationship Diagram
HTTP	HyperText Transfer Protocol
IPv4	Internetový protokol verze 4
JSON	JavaScript Object Notation
JWT	JSON Web Token
LINQ	Language Integrated Query
MS	Microsoft
ORM	Objektově relační mapování (Object Relational Mapping)
PDF	Portable Document Format
RBAC	Řízení přístupu na základě rolí (Role-Based Access Control)
SQL	Structured Query Language
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

1 Úvod

V dnešní době dochází často ke vzniku nových programovacích jazyků a technologií obecně. Platforma .NET je právě jedním z představitelů moderního vývoje, je určena především pro prostředí systému Windows, který se postupem času stal jedním z nejvíce využívaných operačních systémů na světě. Zároveň .NET nabízí i možnosti multiplatformního vývoje.

S důrazem k tomu, že naprostá většina dnešních aplikací pracuje s daty, která se musejí následně ukládat, zpracovávat a vyhodnocovat, má smysl a význam zabývat se tématem práce s daty v prostředí .NET. Platforma .NET nabízí různé formáty zpracování dat a různorodé komponenty a nástroje zprostředkovávající práci s daty.

1.1 Motivace a cíl

Primárním cílem této práce je vytvoření programových ukázek v jazyce C#/.NET, které mohou sloužit i jako příručka ke zkoumané problematice. Jde především o průzkum možností standardních knihoven jazyka s využitím dostupných zdrojů. Práce se mimo jiné zaměřuje na standardní komponenty pro práci s daty, jakou jsou *DataGrid*, *DataAdapter*, *DataSet* a *DataTable*. Možnosti využití těchto komponent jsou široké.

Na základě zpracovaných ukázek bude v druhé části práce vytvořena komplexní ukázka aplikace, čímž bude uvedeno využití zmíněných konstrukcí, knihoven a komponent jazyka. Motivací k vytvoření této aplikace je především praktická ukázka, která shrnuje vlastní podklad práce.

Objednávkový systém byl vybrán na základě možného rozšíření stávajících databázových aplikací společnosti Příhoda s. r. o. (FIS) modulem objednávek s doprovodnými funkcemi, jako je částečná evidence produktů a uživatelů. K jednotlivým schématům databáze je vhodné vytvořit aplikační rozhraní. Je zde snaha o vyšší využití databází pro práci s firemními daty a jejich následné zpracování.

1.2 Platforma .NET a její využití v současné době

Moderní aplikace vyžadují dostatečně efektivní technologie pro rychlé prototypování. Platforma .NET nabízí mnoho prostředků, které tento trend podporují. Pomocí programovacího jazyka platformy, jako je například programovací jazyk C#, lze vytvořit širokou škálu aplikací, co se týče oblasti nasazení aplikace a funkcionality aplikace. C# není jediným programovacím jazykem platformy .NET, který se může použít pro tvorbu aplikací a práci s daty. Dostupné jsou i další programovací jazyky v rámci platformy. Práce se však věnuje pouze jazyku C#, který je v současné praxi často využívaný, zároveň byl vyučován v rámci bakalářského studia, v předmětu Programování desktopových aplikací. V dostupných verzích .NET lze vytvářet webové aplikace a API, desktopové aplikace především pomocí Windows Presentation Foundation (WPF) či Windows Forms a aplikace určené pro mobilní platformy primárně v MAUI. Stejně jako operační systém Windows, je .NET udržován a vyvíjen společností Microsoft. Celkově lze říci, že platforma toho nabízí více než lze popsat v rámci jedné práce. Tato práce se zaměří pouze na vybrané partie, které souvisí s tématem práce s daty.

2 Zdroje dat a komponenty pro práci s daty

S postupem času se na základě často využívaných zdrojů dat rozvíjejí příslušné knihovny a komponenty jazyka C#, které mají za účel především usnadnění práce a zvýšení efektivity. Jazyk C# je moderní programovací jazyk se širokou komunitou a možnostmi v oblasti práce s daty. V současné době je umožněno využívat připravená řešení v podobě standardních knihoven jazyka, které pokrývají většinu běžných situací. V případě dalšího rozšíření lze jednotlivé knihovny instalovat přímo do projektu pomocí správce balíčků NuGet. Mezi běžné zdroje dat patří soubory, databázové systémy a webové či cloudové zdroje. Případně lze data čerpat z různých jiných systémů. Kromě toho je vždy možnost mít data uložena v paměti. Komponenty určené pro práci s daty, ať už jako prostředník pro práci s databází, či jako prostředník pro zpracování a zobrazení dat, jsou především *DataGrid*, *DataTable*, *DataSet* a *DataAdapter*.

2.1 Práce se soubory

Stejně jako u většiny ostatních programovacích jazyků je zde řešena komunikace se soubory pomocí streamů. Z užšího pohledu se jedná o proud dat (bajtů). Základem je abstraktní třída *Stream*, která se nachází v jmenném prostoru *System.IO* a nabízí především metody *Read* a *Write*, které umožňují čtení a zápis do proudu. Jako proud dat si lze v abstraktní rovině představit například data přenášená mezi dvěma zařízeními.

Tato část práce je zde uvedena také z důvodu představení exportu a importu dat, které jsou běžnou součástí většiny současných aplikací pracujících s daty. V těchto programech se často využívá komunikace, která může záviset na exportu dat z jednoho systému a jejich následném importu do systému druhého. Z těchto důvodů se zaměříme především na čtení a zápis dat do zvolených souborových formátů. S využitím uvedených ukázek bude možné zhotovit část systému určenou pro práci s exporty a importy dat. Na závěr je uvedeno zhodnocení užitečnosti pro účely importu a exportu. Budou probrány především formáty XML, JSON a CSV. Úplné zdrojové kódy programových ukázek jsou součástí přílohy 1 (adresář PraceSeSoubory).

2.1.1 Serializace a deserializace dat

Serializací se rozumí transformace zdrojových dat do podoby, ve které jsou snadno přenositelná, primárně z objektové reprezentace programovacího jazyka do formátu, jako je například JSON. Opačným procesem k serializaci je deserializace.

2.1.2 XML soubory

Formát XML slouží především k přenosu dat mezi systémy. Uplatnění lze z praktického hlediska nalézt ve webových službách, různých typech systémů nebo jen jako způsob uložení dat pro další zpracování. Může sloužit i k uložení konfigurací. Je to značkovací jazyk, což znamená, že svůj obsah vyjadřuje využitím tagů (značek). Nejedná se však o jazyk, ve kterém by se vytvářel zdrojový kód pro aplikace, nýbrž o jazyk uchovávající data se strukturovanou hierarchickou podobou. Na jeho základě je postaven XAML.

Obrázek 1: Příklad XML formátu

Zdroj: vlastní tvorba

Lze jednoznačně usoudit, že XML určitě najde své uplatnění. Platforma .NET má připravené komponenty pro práci s tímto formátem již ve vlastních standardních knihovnách, členěných do jmenných prostorů. *System.Xml* je jmenný prostor, který obsahuje většinu tříd, pomocí nichž lze především číst a zapisovat do souborů tohoto typu. Každý XML soubor se skládá z hlavičky a kořenového tagu, jemuž jsou podřízeny tagy ostatní, v ukázce je kořenovým (root) tagem People. Hlavička se skládá z čísla verze a specifikuje se zde i znaková sada. Třída *XmlReader* slouží pouze ke čtení ze souboru. Pro účely programové ukázky byla vybrána metoda *ReadToFollowing*. Tato metoda probíhá, dokud se nenarazí na element s daným názvem, pokud se nenajde, vrátí metoda false. Zároveň má i přetížené varianty, může přebírat parametr identifikátoru namespace. S použitím *XmlReaderu* není potřeba ukládat všechna data souboru do paměti. (Microsoft, 2025k)

Na níže uvedeném obrázku (Obrázek 2) je ukázka čtení dat a jejich následného vypsání do konzole. Je zde znázorněno využití třídy *XmlReader*. Metoda *ReadToFollowing* zajišťuje přesun aktuální polohy v souboru na daný element (tag). Metoda *ReadElementContentAsString* převede obsah elementu na řetězec, který se vypíše. Je možné se také setkat se serializací nebo mapováním na objekt (nastavení vlastností objektu načtenými hodnotami). Veškeré potřebné kroky pak zajistí objekt třídy *XmlSerializer*.

```
using (XmlReader readerXML = XmlReader.Create(Path.Combine(crdir, "data.xml"))) {
    readerXML.ReadToFollowing("People");
    while (readerXML.ReadToFollowing("Person")) {
        Console.WriteLine("==============");
        readerXML.ReadToFollowing("Name");
        Console.WriteLine($"Jmeno: {readerXML.ReadElementContentAsString()}");
    }
}
```

Obrázek 2: Ukázka XmlReader

Zdroj: vlastní tvorba

Třída *XmlWriter* je určena k zápisu do souborů XML. Stejně jako *XmlReader* nabízí i *XmlWriter* řadu metod. Při vytváření instance třídy lze využít cestu k souboru, který bude vytvořen či přepsán. Mezi důležité metody třídy patří *WriteStartElement* či její asynchronní ekvivalent *WriteStartElementAsync*, *WriteString* a *WriteEndElement*. Jednoduchá ukázka, která vytvoří obdobnou strukturu jako v předešlém příkladě, jednoduše demonstruje využití zmíněných metod. *WriteStartElement* vytvoří otevírací tag a *WriteEndElement* vytvoří uzavírací tag. Metoda *WriteString* zapíše řetězec předaný argumentem mezi tagy do elementu. (Microsoft, 2025l)

```
XmlWriter xmlWriter = XmlWriter.Create("user_write.xml");
xmlWriter.WriteStartElement("user");
xmlWriter.WriteStartElement("name");
xmlWriter.WriteString("Adam");
xmlWriter.WriteEndElement();
xmlWriter.WriteStartElement("age");
xmlWriter.WriteString("24");
xmlWriter.WriteEndElement();
xmlWriter.WriteEndElement();

xmlWriter.Close();
Console.WriteLine("Byl proveden zapis do souboru");
```

Obrázek 3: Ukázka XmlWriter

Zdroj: vlastní tvorba

Třída *XmlSerializer* je určena k převodu objektů do formátu XML a naopak. Konkrétní příklady již byly uvedeny pro *XmlReader*, kde se využívá deserializace pro načtení dat ze souboru. Pro usnadnění práce při serializaci a deserializaci slouží atribut *XmlElement*, který je potřeba uvést u vlastností třídy reprezentující tag v souboru, z ukázek tedy u třídy osoby či seznamu osob (lze využít i *XmlRoot* pro kořenový element). Dalšími možnostmi jsou například *XmlArray* a *XmlArrayItem,* související logicky s poli hodnot. Formát je vhodný pro přenos dat. Mezi jeho výhody se řadí jednoznačnost a čitelnost formátu uložených dat. Primární výhodou je dobrá podpora napříč různými systémy. Primární nevýhodou XML je jeho verbózní zápis, což logicky zpomaluje práci s tímto formátem především v porovnání s CSV a JSON. Kompletní kódy ukázek jsou součástí přílohy 1 (projekt xml v adresáři PraceSeSoubory).

2.1.3 JSON soubory

S příchodem webových technologií a programovacího jazyka JavaScript vznikl dnes často využívaný a oblíbený formát JSON. V současnosti je široce rozšířený a užitečný, uplatnění najde jako odlehčená alternativa k XML. JSON používají webové API servery jako hlavní formát komunikace. K práci s formátem JSON je určen jmenný prostor *System.Text.Json*. Při realizaci ukázky čtení ze souboru bylo využito načtení obsahu souboru do seznamu metodou *File.ReadAllText* a následné deserializovaní na seznam objektů typu *Person* zavoláním metody *JsonSerializer.Deserialize*, k níž je dostupná i její asynchronní varianta *DeserializeAsync* (Microsoft, 2025i). V neposlední řadě jsou volně dostupné knihovny, které jsou zaměřeny přímo na JSON. Ukázky jsou postaveny na serializaci a deserializaci dat.

```
// a) načtení řetězce s JSON obsahem
string jsonContent = File.ReadAllText(filePath);
// b) deserializace
var people = JsonSerializer.Deserialize<List<Person>>(jsonContent);
// c) výpis dat
if (people != null)
    foreach (var person in people)
    {
        Console.WriteLine($"Jmeno: {person.Name}");
        Console.WriteLine($"Vek: {person.Age}");
        Console.WriteLine($"Email: {person.Email}");
        Console.WriteLine($"Telefonni cislo: {person.PhoneNumber}\n");
    }
```

Obrázek 4: Ukázka čtení ze souboru JSON

Zdroj: vlastní tvorba

Zápis do JSON souboru probíhá z dat uložených v datové struktuře, obvykle objektu či seznamu objektů. Voláním metody *Serialize* vytvoříme textový řetězec reprezentující JSON pro výstup. Následně zapíšeme do zvoleného souboru řetězec. (Microsoft, 2025i)

```
var jsonString2 = JsonSerializer.Serialize(listperson);
File.WriteAllText(fileOutPath1, jsonString2);
Console.WriteLine("Zapis do souboru byl proveden");
```

Obrázek 5: Ukázka zápisu do souboru JSON

Zdroj: vlastní tvorba

Kromě výše zmíněného existují také třídy *Utf8JsonReader* a *Utf8JsonWriter*. Reprezentují obsah zdroje jako sekvenci elementů. Hodí se, pokud nechceme načítat celý obsah zdroje do paměti. Jedná se o velmi rychlostně optimální řešení, ale není nejsnazší na implementaci. S využitím *File.ReadAllBytes* načteme všechny bajty souboru do proměnné *jsonData*, která je datového typu *ReadOnlySpan<byte>*, ta uchovává data, ale nevytváří strukturu objektů. Následně na základě *JsonTokenType* rozlišíme, zda se jedná o řetězec, a provedeme patřičné kroky pro zobrazení výstupu. Ukázka je vytvořena pro *Utf8JsonReader*. JSON je rozšířený a vhodný formát pro import a export dat, protože je snadno čitelný, přenositelný mezi různými technologiemi a má jistě další výhody. Kompletní kód ukázek je součástí přílohy 1 (projekt json v adresáři PraceSeSoubory). (Griffiths, 2024)

```
ReadOnlySpan<byte> jsonData = File.ReadAllBytes(filePath);
Utf8JsonReader reader = new Utf8JsonReader(jsonData);
while (reader.Read())
{
    JsonTokenType tokenType = reader.TokenType;
    switch (tokenType)
    {
        case JsonTokenType.String:
            {
                string? text = reader.GetString();
                Console.Write(text);
                Console.WriteLine();
                break;
            }
    }
}
```

Obrázek 6: Ukázka Utf8JsonReader

Zdroj: Microsoft 2025h, vlastní zpracování

2.1.4 CSV soubory

CSV je formát určený zejména pro ukládání tabulkových dat. Skládá se z hlavičky (nepovinné) na prvním řádku, složené z názvu sloupců oddělených oddělovačem, což je ve většině případů středník. Následují samotná data, která jsou uložena na každém řádku pro všechny sloupce (oddělena stejným způsobem jako hlavička). Ukázky lze realizovat také pomocí knihovny CsvHelper (Close, 2025). Níže uvedená ukázka demonstruje využití knihovny CsvHelper pro načtení dat CSV souboru jako seznam objektů typu Person.

```
var config = new CsvConfiguration(CultureInfo.InvariantCulture)
{
    HasHeaderRecord = true, // HasHeaderRecord = true ... Soubor obsahuje hlavicku
    Delimiter = ";", // Delimiter = ";" ... Sloupce jsou oddeleny stredníkem
};

using (var reader = new StreamReader(csv_path))
using (var csv = new CsvReader(reader, config))
{
    var people = csv.GetRecords<Person>(); // ziskani List<Person>

    foreach (Person p in people)
    {
        Console.WriteLine($"\nJmeno: {p.Name}");
        Console.WriteLine($"Vek: {p.Age}");
    }
}
```

Obrázek 7: Ukázka čtení z CSV souboru pomocí knihovny CsvHelper

Zdroj: vlastní tvorba

V případě ručního zpracování při čtení se často využívá metoda *Split* pro práci s řetězci, která rozdělí řetězec na pole dle zadaného oddělovače. Následně lze s tímto polem dále pracovat, například ho použít pro naplnění objektu reprezentujícího soubor dat. V případě knihovny CsvHelper lze využít třídu *CsvReader*, ta zajistí naplnění seznamu daty. Kromě možnosti konfigurace, která nechybí ani zde, je tu třída, která se registruje pomocí metody *RegisterClassMap* a slouží k definici mapování mezi sloupci excelovského souboru a vlastnostmi třídy. Pro zápis do souboru se obdobně využije třída *CsvWriter*.

```
List<string> lns = new List<string>();
lns.Add("Name;Age;Email;City;Country;PhoneNumber;Hobby"); // hlavicka

foreach (Person person in data)
{
    lns.Add($"{person.Name};{person.Age};{person.Email};" +
        $"{person.City};{person.Country};" +
        $"{person.PhoneNumber};{person.Hobby}");
}

File.WriteAllLines(csv_path_2, lns);
```

Obrázek 8: Ukázka ručního zápisu do CSV souboru

Zdroj: vlastní tvorba

Vzhledem k tomu, že CSV formát je strukturně nekomplikovaný, jde o nenáročnou a rozšířenou možnost úschovy dat. Nespornou výhodou je rychlost zpracování, která vyplývá z jednoduchosti zápisu. Nevýhodou je nutnost ošetření dat, která by mohla obsahovat oddělovač, v takovém případě lze daný text uvést do dvojitých uvozovek, jinak může dojít k narušení formátu. Kompletní kód ukázek je součástí přílohy 1 (projekt csv v adresáři PraceSeSoubory).

2.1.5 Třída File a Directory

Pro práci se soubory jsou zde dále třídy *System.IO.File* a *System.IO.Directory*. Uplatnění najdou nejen v kombinaci s textovými soubory. Pokud ukládáme název cesty k souboru do řetězce, lze tuto cestu anotovat pomocí znaku zavináč, tím se zajistí escapování. Třídy *StreamWriter* a *StreamReader* jsou určeny pro čtení a zápis do proudu dat v určitém kódování.

Directory.CreateDirectory vytvoří adresář, pokud neexistuje.

Directory.EnumerateDirectories vrátí seznam adresářů v daném adresáři či podadresářích.

Directory.EnumerateFiles vrátí seznam souborů v daném adresáři a případně v podadresářích.

Directory.Exists obdobně jako stejně pojmenovaná metoda třídy File umožňuje zkontrolovat existenci adresáře.

Directory.GetCurrentDirectory vrátí cestu k aktuálnímu adresáři.

File.Copy se může využít při kopírování obsahu jednoho souboru do druhého.

File.Delete smaže existující soubor dle cesty zadané parametrem.

File.Exists určí, zda existuje soubor, jehož název je zadán parametrem.

File.ReadAllText jednoduše načte obsah souboru do řetězce.

File.WriteAllText zapíše řetězec do souboru, případný obsah přepíše.

Třída *FileMode* obsahuje konstanty určené pro práci se soubory, například *FileMode.Open*.

Path.Combine je metoda určená k sloučení dvou cest do jedné, používá se například pokud je nutné vytvářet různé cesty spjaté s jedním stejným adresářem.

Kompletní kód ukázek je součástí přílohy 1 (projekt filedir v adresáři PraceSeSoubory).

```
using (FileStream fs = new FileStream("Soubor.txt", FileMode.Append))
using (StreamWriter writer = new StreamWriter(fs, Encoding.UTF8))
{
    // Přidá >>RANDOM<< na konec souboru
    writer.Write(">>RANDOM<<");
}

// aktuální stav souborů
string aContent = File.ReadAllText("Soubor.txt");
Console.WriteLine($"Aktuální obsah souboru Soubor.txt: {aContent}");

// vyjmenuje všechny soubory v adresáři
string currDir = Directory.GetCurrentDirectory();
var filesTxt = Directory.EnumerateFiles(currDir, "*.txt", SearchOption.AllDirectories);
Console.WriteLine("Složka programu obsahuje txt soubory:");
foreach (var file in filesTxt)
{
    // vypíše pouze název souboru
    Console.WriteLine($"> {Path.GetFileName(file)}");
}
```

Obrázek 9: Ukázka práce s třídou File a Directory

Zdroj: vlastní tvorba

2.2 Komunikace s databází

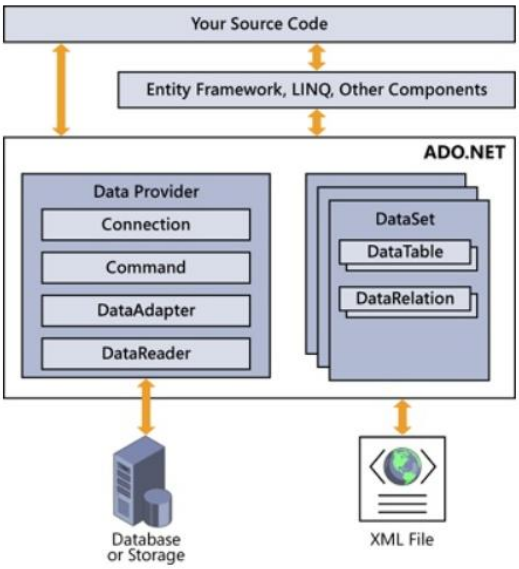
V současné době jsou databáze hlavní možností úschovy dat pro firemní aplikace. Systém řízení báze dat a báze dat tvoří dohromady databázový systém. V databázovém systému jsou v databázi uložena data, se kterými lze realizovat akce pomocí dotazovacího jazyka. Tento jazyk se nazývá SQL. Mezi komerční řešení lze zařadit Microsoft SQL Server. Oblíbenou alternativou je PostgreSQL.

2.2.1 ADO.NET

Jde o sadu komponent prostředí .NET, která umožňuje přístup k datům a jejich správu, kde hlavním zdrojem dat jsou databáze, ale může se jednat i o jiný zdroj. ADO.NET zahrnuje operace, jako je připojení ke zdroji a získání případných výsledků spuštěných dotazů, umožňuje také spouštět plnohodnotné CRUD dotazy. Na rozdíl od svých předchůdců nabízí možnosti odpojeného modelu, není tedy nutné mít aktivní spojení s databází po celou dobu interakce uživatele s aplikací. Základní možnosti připojení jsou dvě, připojené a odpojené. Pro odpojené řešení se využívá možností třídy *DataSet*, v druhém případě jsou data také získávána ze zdroje, ale každá změna se musí rovnou propsat. *DataSet* reprezentuje databázovou strukturu. Pro účely práce uvažujme primárně databázi jako zdroj dat. Knihovna ADO.NET komunikuje s databází na základě třídy *DbConnection*, ta umožňuje pomocí připojovacího řetězce (connection string) navázat spojení s databází. Tento řetězec se obvykle skládá z portu a názvu či IPv4 adresy serveru, kde běží databáze, názvu databáze a dalších parametrů, které se liší na základě zvoleného databázového systému. Jednotlivé dotazy jsou reprezentovány pomocí třídy *Command. Transaction* umožňuje spustit databázovou transakci. Datová vrstva dále zahrnuje třídy *DataTable*, *DataRow* či *DataColumn*, se kterými dále pracuje obalovací třída této vrstvy *DataSet*. *DataGrid* a *DataGridView* jsou komponenty určené pro reprezentaci tabulkových dat formou grafické komponenty (gridu). (Patrick, 2010)

2.2.2 Připojená a odpojená architektura

Princip činnosti obou možností přístupu je pro lepší představu nejvhodnější vyjádřit graficky, proto je zde uveden obrázek. Připojené řešení může být výhodou, co se týče paměťové náročnosti, protože se data čtou přímo z databáze. Naopak odpojené řešení má význam, pokud k databázi přistupuje hodně uživatelů najednou, šetří se zde otevřená připojení, což může pozitivně ovlivnit výkon. Důležitými třídami jsou především *DataTable, DataReader* a *DataAdapter*. Aktuálně jsou nejčastěji využívány objektově-relační knihovny jako je Entity Framework nebo Dapper.

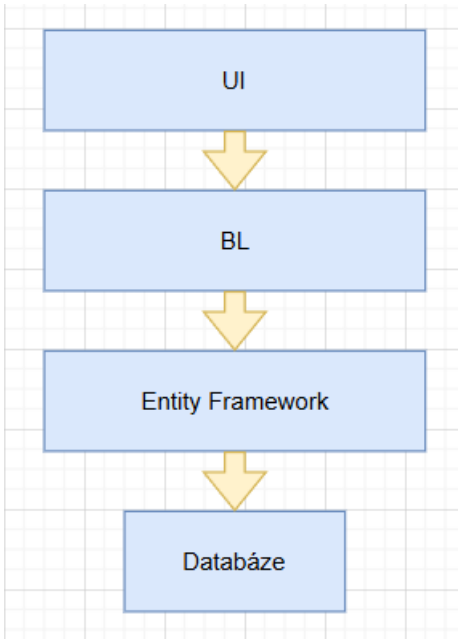


Obrázek 10: Klíčové ADO.NET komponenty

Zdroj: Patrick (2010)

2.2.3 Entity Framework a Dapper

Je na místě zmínit Entity Framework a Dapper. Jedná se o řešení, která jsou založena na principu objektově-relačního mapovaní a s jejichž pomocí není nutné řešit poměrně komplikované využití ručního mapování. Řádky z tabulky databáze jsou namapovány na objekty (entity), se kterými lze dále pracovat. Často se zde uplatňuje návrhový vzor repozitáře, který zapouzdřuje přístup k datům. Tento přístup zjednodušuje testování, jmenovitě testování metod. Diagram byl vytvořen pomocí softwaru Draw.io (JGraph Ltd., 2025) a je součástí přílohy 3 (EF_NTier.drawio).



Obrázek 11: Entity Framework z pohledu třívrstvé architektury
Zdroj: vlastní tvorba

Entity Framework je součástí ADO.NET a umožňuje částečné odstínění práce programátora od databáze. Je však nutné definovat objekty (entity) pro objektově-relační mapování. Entitní třídy jsou zpravidla veřejné třídy s veřejnými vlastnostmi opatřenými gettery a settery. Mapování vlastností tříd se děje automaticky, jinak je možné využít vlastnost *Column*, případně definice v *OnModelCreating* pomocí Fluent API. Z pohledu třívrstvé architektury spadá do již zmíněné datové vrstvy mezi databázi a Business Logic (BL). UI je uživatelské rozhraní aplikace. Pro připojení k databázi je potřeba definování databázového kontextu aplikace. To probíhá tak, že se podědí třída *DbContext*. Jednotlivé kolekce entit jsou pak dostupné z kontextu pomocí veřejných vlastností *DbSet<T>,* nad kterými lze využít LINQ. *DbSet<T>* je generická třída pracující s abstraktním datovým typem T. Přepsáním metody *OnModelCreating* lze nastavit konfiguraci. Následně lze vytvořit migraci dat pomocí příkazu *Add-Migration*. Pro zapsání změn je však ještě nutné zavolat *Update-Database*, což je příkaz, který zkontroluje, zda již byla migrace zapsána do databáze, či ne, a případně provede změny. Příkazy se volají v Package Manager Console nebo lze využít její alternativy ve standardní příkazové řádce, pro vytvoření migrace *dotnet ef migrations add* a pro uložení změn *dotnet ef database update*. Řetězec připojení k databázi se předává kontextu. Tímto způsobem lze vytvářet moderní aplikace s využitím ORM. Ukázka je součástí přílohy 1 (složka EntityFramework). Vytvořená ukázka využívá SQLite (The SQLite Consortium, 2025). Příkaz *Batteries.Init* je volán právě kvůli tomu, že se využívá SQLite. (Smith, 2021)

```
Console.WriteLine("=== Entity Framework ===");

using var ctx = new AppDbContext();

Batteries.Init();
string cmdTxt =
@"
INSERT INTO Products (Id, Name, Price, Description) VALUES
(1, 'Podlozka', 250, 'popis'),
(2, 'Notebook', 20000, ''),
(3, 'Mobil', 5000, ''),
(4, 'Lampa', 1200, ''),
(5, 'Baterka', 1000, '')
";
ctx.Database.ExecuteSqlRaw("Delete From Products"); // vyprázdní tabulku Products
ctx.Database.ExecuteSqlRaw(cmdTxt);
Console.WriteLine("Data byla úspěšně vložena do databáze");

// výpis
foreach (var p in ctx.Products.ToList())
{
    Console.WriteLine($"{p.Name} [{p.Price:C}]");
}
```

Obrázek 12: Ukázka práce s daty pomocí Entity Frameworku

Zdroj: vlastní tvorba

Dapper je framework vytvořený komunitou. V některých případech se vyplatí ho využít, především pokud se uvažuje nad psaním vlastních SQL dotazů, protože umožňuje snazší přizpůsobení včetně podpory tvorby vlastních dotazů, což umožňuje vetší kontrolu při práci s databází. Bývá často využíván místo Entity Frameworku při tvorbě webových API.

2.3 Komponenty ADO.NET

Součástí ADO.NET jsou především komponenty dostupné v podobě tříd. Jejich hlavním účelem je usnadnění připojení k databázi, provádění dotazů a následná reprezentace dat. To vše nabízí programové rozhraní datové vrstvy ADO.NET, která je jednou z komponent nativních knihoven .NET.

2.3.1 DataSet

Tato datová struktura je základní komponentou určenou pro úschovu dat. Její využití spočívá především v úschově dat v paměti. Je nezávislou komponentou. Může reprezentovat databázi a s využitím příslušných datových struktur přistupovat k datům, to vše bez nutnosti připojení k databázi, to je potřeba pouze při načítání a ukládání změn dat. Obecně lze k *DataSetu* přistupovat jako ke kolekci objektů třídy *DataTable*. *DataSet* zapouzdřuje jednotlivé objekty *DataTable* a umožňuje jejich souhrnnou správu. Po dokončení práce s tabulkami vložíme jednotlivé instance třídy *DataTable* do *DataSetu*, a to s využitím metody *Tables.Add* do *DataTableCollection*. Tímto způsobem naplníme *DataSet* daty. Celý proces není nijak komplikovaný, postup zachycuje následující programová ukázka. Samozřejmě lze pracovat přímo s třídou *DataSet*. Na objekty této třídy může být pohlíženo jako na relativně komplexní in-memory zdroj dat. Při větších objemech dat bude logicky program využívající *DataSet* paměťově náročnější, zároveň má nízký potenciál využití v real-time systémech. Například v případě časté úpravy dat s možností kumulovaného aplikovování změn do databáze se však může vyplatit.

```
dtProduct.Columns.Add(colProductId);
dtProduct.Columns.Add(colName);
dtProduct.Columns.Add(colPrice);
dtProduct.Columns.Add(colAvailableQty);
dtProduct.Columns.Add(colDescr);
dtProduct.Columns.Add(colCategoryId);
DataColumn? colPK = dtProduct.Columns["ProductId"];
if (colPK != null)
    dtProduct.PrimaryKey = new[]{ colPK }; // PK

// Vytvoření tabulky ProductCategory
DataTable dtProductCategory = new DataTable("ProductCategory");

DataColumn dcProductCategoryId = new DataColumn("ProductCategoryId", typeof(int));
DataColumn dcCategoryName = new DataColumn("Name", typeof(string));

dtProductCategory.Columns.Add(dcProductCategoryId);
dtProductCategory.Columns.Add(dcCategoryName);
DataColumn? colPK2 = dtProductCategory.Columns["ProductCategoryId"];
if (colPK2 != null)
    dtProductCategory.PrimaryKey = new[]{ colPK2 }; // PK
// přidání do ds
ds.Tables.Add(dtProduct);
ds.Tables.Add(dtProductCategory);
```

Obrázek 13: Ukázka komponenty DataSet a DataTable

Zdroj: vlastní tvorba

Kromě tabulek obsahuje *DataSet* i relace, ty lze chápat jako cizí klíče z pohledu databází. U sloupců lze definovat datový typ ukládaných informací s využitím funkce *typeof*. Nyní je vhodné rozebrat způsoby načtení dat do *DataSetů*. Prvním způsobem je serializace a deserializace dat ze souborů, daná tematika již byla probrána v předchozích částech práce, proto si uveďme jen obslužné metody. Pro zápis do formátu XML jazyk C# nabízí metody *WriteXml* nebo *WriteXmlSchema*. Ekvivalentem pro čtení dat jsou metody *ReadXml* či *ReadXmlSchema*. Nativně podporovaným formátem je v současné době XML. Dalším častým způsobem načtení dat do *DataSetu* je pomocí databázového připojení. K tomu slouží třída *DataAdapter*, která se liší na základě zvolené databáze, příkladem je *SqlDataAdapter* nebo *NpgsqlDataAdapter* (princip bude popsán). Rozšiřující informace lze ukládat do vlastnosti *ExtendedProperties*, patří sem metadata. Ukázky jsou součástí přílohy 1 (adresář DataSet). (Microsoft, 2025e)

```
using (XmlReader readerXML = XmlReader.Create(Path.Combine(crdir, "data.xml"))) {
    readerXML.ReadToFollowing("People");
    while (readerXML.ReadToFollowing("Person")) {
        Console.WriteLine("==============");
        readerXML.ReadToFollowing("Name");
        Console.WriteLine($"Jmeno: {readerXML.ReadElementContentAsString()}");
    }
}
```

Obrázek 14: Ukázka volání metody WriteXML

Zdroj: vlastní tvorba

2.3.2 DataTable

Často využívanou třídou pro reprezentaci dat, kterou zahrnuje ADO.NET, je především *DataTable*. Slouží jako reprezentace databázové tabulky, tj. obsahuje řádky a sloupce. Stejně jako DataSet lze i tuto datavou strukturu pomocí bindingu propojit s prvky grafického rozhraní. Řádky a sloupce jsou reprezentovány třídami, které se nazývají *DataRow* a *DataColumn*. *DataTable* je uchovává ve formě již zmíněných kolekcí *DataRowCollection* a *DataColumnCollection*. Pro získání konkrétních hodnot se používá zpravidla cyklus for s iterací přes řádky, takto lze přistupovat k sloupci přes jeho název.

Tabulka 1: Výběr vlastností DataTable

Název vlastnosti	**Význam**
Columns	Vrátí seznam sloupců tabulky.
Rows	Vrátí seznam řádků tabulky.
TableName	Umožňuje nastavit název tabulky.
PrimaryKey	Jde o primární klíč tabulky.
Constraints	Vrátí seznam relačních omezení tabulky.
DataSet	Vrátí přidružený DataSet.
DefaultView	Umožňuje získat základní DataView tabulky.
ExtendedProperties	Obdobně jako u komponenty DataSet, i zde umožňuje získání uživatelských komentářů.

Zdroj: Microsoft (2025f), vlastní zpracování

Jednotlivé sloupce lze vytvořit pomocí třídy *DataColumn* a následně je vložit do kolekce *Columns*, čímž lze vytvořit kompletní strukturu tabulky. Poté už stačí jen tabulku zaplnit konkrétními daty. V kombinaci s *DataSetem* lze definovat relace jednotlivých tabulek pomocí třídy *DataRelation*, čímž lze i specifikovat referenční integritu dat pomocí kaskádové akce (v SQL známé jako *cascade*). Po naplnění instance třídy *DataTable* daty je velmi často potřeba s těmito daty dále pracovat, k tomu slouží v první řadě metoda *Find* třídy *DataRowCollection*, umožňující vyhledání příslušného řádku dle primárního klíče. Možnosti filtrování jsou pak realizovány vlastností *DefaultView.RowFilter*, kde lze definovat podmínky, jako jsou menší, větší, rovná se, a je možné filtrovat podobně jako v SQL (pomocí LIKE). Jsou podporovány i logické operace, jako jsou AND a OR (Microsoft, 2025f).

Tabulka 2: Možné stavy řádků DataRow

DataRowState	Popis
Unchanged	Řádek nebyl změněn od posledního volání AcceptChanges.
Added	Řádek byl přidán, zatím neuložen pomocí AcceptChanges.
Modified	Řádek byl změněn, změna je zatím nepotvrzena pomocí AcceptChanges.
Deleted	Řádek byl smazán pomocí DataRow.Delete.
Detached	V tomto případě je řádek vytvořen, ale není zařazen do žádné DataRowCollection.

Zdroj: Microsoft (2025d) , vlastní zpracování

2.3.3 Třída DataView ve spolupráci s DataTable

Reprezentuje pohled na data. Třída umožňuje třídit a řadit data v *DataTable* bez přímé modifikace uložených dat. Ta jsou vrácena samostatně na základě aplikovaných pravidel. Nejedná se tak přímo o vizuální komponentu, nýbrž slouží jako pohled na data. *DataTable* má vlastnost *DefaultView*, která je datového typu *DataView* a umožňuje vracet upravená data voláním přímo přes objekt tabulky (již bylo diskutováno). Kromě výchozího view lze vytvořit i jiné *DataView*. Obecně se vyplatí obstarat si pro konkrétní činnost vlastní *DataView*. Nespornou výhodou je, že třída *DataView* je určena pro datový binding, takže ji lze propojit s vizuálními komponentami jako jejich *DataSource*. Z toho vyplývá, že jedna *DataTable* může být v aplikaci zobrazena na základě různých *DataView* vícekrát, což se může vyplatit zejména tehdy, pokud rozhraní aplikace vyžaduje grafické oddělení filtrovaných dat.

Hlavními způsoby práce s *DataView* jsou řazení a filtrování. Pro řazení lze využít řetězcovou vlastnost *Sort*, s níž lze určit, podle jakých sloupců se bude řadit. Může se použít sestupné *DESC* a vzestupné *ASC*. V případě řazení pomocí více sloupců jsou pak zápisy způsobů řazení odděleny čárkou. Filtrovaní je uskutečněno použitím již zmíněné vlastnosti *RowFilter*, která ukládá řetězec. Podmínky jsou obdobné těm z SQL, takže je lze řetězit pomocí *AND*. Porovnávat je umožněno pomocí znaků menší než či rovno. Na obrázku ukázky je názorný příklad.

```
// DataView
DataView dw = new DataView(dtProduct);
dw.AllowNew = true;
dw.AllowEdit = true;
dw.AllowDelete = true;
var nRow = dw.AddNew();
nRow["ProductId"] = 6;
nRow["Name"] = "LCD";
nRow["Price"] = 1100;
nRow["AvailableQty"] = 10;
nRow["Descr"] = "Stručný popis";
nRow["ProductCategoryId"] = 2;
nRow.EndEdit();
dw.Sort = "ProductId ASC";
dw.RowFilter = "Price > 1000";
dw[2].Row["Name"] = "PC";
for (int i = 0; i < dw.Count; i++)
{
    Console.WriteLine(dw[i].Row["Name"]);
}
dtProduct.RejectChanges();
Console.WriteLine();
for (int i = 0; i < dtProduct.Rows.Count; i++)
{
    if(i < dw.Count)
        Console.WriteLine($"DataView: {dw[i].Row["Name"]}");
    Console.WriteLine($"DataTable: {dtProduct.Rows[i]["Name"]}");
}
```

Obrázek 15: Příklad DataView s DataTable

Zdroj: vlastní tvorba

2.3.4 DataAdapter

Další již zmíněnou třídou je *DataAdapter*. Pro jednoduché vysvětlení jejího účelu lze využít obecně známý příklad síťového adaptéru napájející notebook, na obdobném principu je třída založena. Jejím hlavním účelem je propojení zdroje dat s komponentou jako je *DataTable*. *DataAdapter* je založen na návrhovém vzoru adaptér, poskytuje možnost propojit dvě nezávislá rozhraní a tím umožnit vzájemnou komunikaci. Možností realizace vzoru je více, obvykle se využívá interface s potřebnými metodami, který implementuje třída adaptéru, ta mimo to obsahuje objekty třídy, k níž je třeba vytvořit adaptér. Pomocí metod adaptéru se volají nad tímto objektem akce. Z toho je zřejmé, že se jedná o užitečný návrhový vzor, který lze využít při vzájemné komunikaci objektů v kontextu získávání dat z různých zdrojů pomocí společného rozhraní, které však původní třídy standardně nenabízejí. Třída *DataAdapteru* vlastně zapouzdřuje volání jednotlivých CRUD operací, obsahuje vlastnosti *InsertCommand*, *UpdateCommand* a *DeleteCommand*, je však nutné je specifikovat před samotným voláním například metody *Update*, jinak dojde k výjimce. Ta dokáže rozpoznat jednotlivé změny v *DataSetu* (nebo *DataTable*) a vybrat vhodnou třídu z výše uvedených, pro vložení dat tedy *InsertCommand*. V případě, že třída *DataTable* koresponduje s tabulkou databáze, lze využít možností třídy *CommandBuilder* k automatickému vytvoření těchto příkazů. Pro rozpoznání případných chyb slouží *DataRow.RowError*. Kromě *Update* je zde i metoda *AcceptChanges* sloužící k akceptaci provedených změn. Pro reset pak slouží *RejectChanges*, která vrátí vše do původního stavu po původním načtení dat. Pro naplnění *DataTable* daty je tu metoda adaptéru pojmenovaná *Fill*, která umožňuje naplnit například *DataTable* daty z výsledku SQL dotazu. Při plnění dat může dojít k situaci, kdy je potřeba mít sloupec v *DataTable* pojmenovaný jinak než

v databázi, k tomu se používá kolekce *DataTableMappings* adaptéru. V případě změny jména sloupců, což je během práce běžnější, lze použít *ColumnMappings*. Ukázka je převzata z přílohy 2. (Microsoft, 2025a)

```
using var conn2 = DbConnProvider.Instance.GetConn();
conn2.Open();
using var cmd = new NpgsqlCommand(sqlAP, conn2);
deliveriesAdapter.SelectCommand = cmd;

dtDeliveries.Clear();
deliveriesAdapter.Fill(dtDeliveries);
```

Obrázek 16: Ukázka DataAdapteru

Zdroj: vlastní tvorba

2.3.5 DataReader

Alternativou pro získání dat z datového zdroje k *DataAdapteru* je *DataReader*. Vrací proud dat, který se využívá pouze pro čtení. Data jsou obvykle získávána sekvenčně voláním metody *Read* v cyklu. Své využití najde při čtení velkých objemů dat, kde není nutné vše ukládat do paměti. Během čtení se drží aktivní připojení k databázi, dokud se vše nepřečte, nebo se čtení nepřeruší. *DataReader* se obvykle využívá s *using*. Data se zde neukládají do mezipaměti, jde tedy o výkonné řešení. Instanci třídy *DataReader* lze získat zavoláním *ExecuteReader* nad instancí třídy *Command*. *GetSchemaTable* je metoda umožňující získat schéma čtených dat. *DataReader* je komponenta, která může nalézt své využití v aplikacích, které přímo komunikují s databází a čtou data. Výhodou je právě rychlost čtení dat. Metoda *GetOrdinal* získá index sloupce z navrácených hodnot dotazu dle jeho názvu. Jsou dostupné i další metody (jako jsou *GetString*, *GetInt32* apod.), které získají konkrétní typ z výsledku dotazu. DataReader se využívá pro dotazy SELECT. Ukázka je převzatá z přílohy 2. (Microsoft, 2025j)

```
using var reader = cmd.ExecuteReader();

if (reader.Read())
{
    customerAccount = new CustomerAccount()
    {
        CustomerAccountId = reader.GetInt32(reader.GetOrdinal(
        UserId = reader.GetInt32(reader.GetOrdinal("user_id"))
```

Obrázek 17: Fragment ukázky DataReaderu

Zdroj: vlastní tvorba

2.3.6 DbCommand

Třída *DbCommand* je stěžejním prvkem aplikace při spouštění SQL dotazů nad databází. Připojení probíhá pomocí připojovacího řetězce zadaného do konstruktoru třídy *DbConnection*. Pro Microsoft SQL Server se jedná konkrétně o třídu *SqlConnection*, která je předána parametrem konstruktoru této třídy. Případně ji lze nastavit pomocí vlastnosti *Connection* přímo v objektu *Command*. Pro účely práce lze vybrat metody *ExecuteNonQuery*, *ExecuteReader, ExecuteScalar* a *CreateParameter*. *ExecuteNonQuery* se používá ke spuštění dotazů UPDATE,

INSERT a DELETE. Pro SELECT lze použít již zmíněnou metodu *ExecuteReader* nebo metodu *ExecuteScalar*, která přečte jen první navrácenou hodnotu (i v případě více hodnot). Jde o stále relevantní komponentu, na kterou se lze odkazovat v repository třídách.

2.4 DataGrid

Pod pojmem grid je myšlena komponenta umožňující zobrazení tabulkových dat. Jednou z klíčových komponent, které umožňují v dnešní době práci s tabulkovým zobrazením dat, je *DataGrid*. Je součástí WPF a v tomto směru najde široké uplatnění. Ve své základní podobě již podporuje změnu pořadí, měnění šířek a řazení sloupců. Vše lze však přenastavit s využitím vlastností *CanUserReorderColumns*, *CanUserResizeColumns* a *CanUserSortColumns*. Tato komponenta se dále skládá ze sloupců, jedná se o *DataGridTextColumn*, *DataGridHyperlinkColumn*, *DataGridCheckBoxColumn*, *DataGridComboBoxColumn, DataGridTemplateColumn* (vlastní styl). V případě *DataGridTextColumn*, kdy se hodnoty ve sloupci dále needitují, dojde k zobrazení *TextBlocku*, jinak se zobrazí TextBox (pro editaci). *DataGridHyperlinkColumn* transformuje text na hyperlinkový odkaz. Pro vlastní otevření odkazu prohlížečem je však nutné tuto funkcionalitu implementovat. *DataGridCheckBoxColumn* vykreslí *CheckBox* do sloupce. Hodnoty sloupce jsou true (políčko je zaškrtnuté) a false. *DataGridComboBoxColumn* se chová jako *ComboBox*.

```
// Načtení dat do WPF DataGridu
// Využití DataTable
private void Load_DataTable()
{
    DataTable dt = new DataTable();

    dt.Columns.Add("OrderId",  typeof(int));
    dt.Columns.Add("Description", typeof(string));
    dt.Columns.Add("Name", typeof(string));
    dt.Columns.Add("OrderedAt", typeof(DateTime));

    for (int i = 1; i <= 8; i++)
    {
        dt.Rows.Add(i, $"[DT] popis {i}", $"Objednávka {i}", DateTime.Now);
    }

    MainDataGrid.DataContext = dt;
}
```

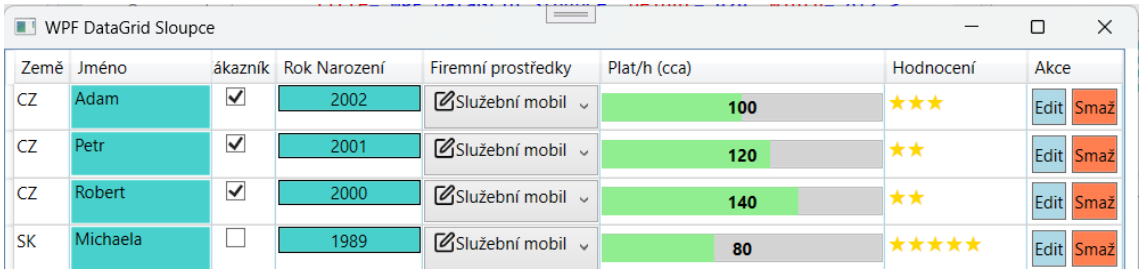
Obrázek 18: Ukázka výběru zdroje dat DataTable pro DataGrid

Zdroj: vlastní tvorba

Pokud je zdroj dat nastaven přes vlastnost *ItemsSource*, dojde k automatickému rozpoznání typu sloupce a následně se sloupce vytvoří a vloží samy. Pořadí odpovídá vlastnostem třídy. V *DataGridu* je *CurrentItem* vždy jen jeden aktuálně vybraný řádek v daný moment. Obdobně jsou dostupné i vlastnosti *CurrentCell* a *CurrentColumn*. Při možnosti výběru více řádků najednou je lze získat vlastností *SelectedItems*. Další jistě využitelnou vlastností je možnost zkopírování vybraných řádků do schránky zkratkou *<Ctrl> + <C>*. Vlastnost *FrozenColumnCount* nastaví počet sloupců, které zůstanou na svém místě při horizontálním scrollování, ty se počítají v tabulce zleva. *DataGrid* se skládá z *DataGridRow*. Ukázka je z přílohy 1 (adresář DataGrid). (Nathan, 2010)

DataColumnTemplate

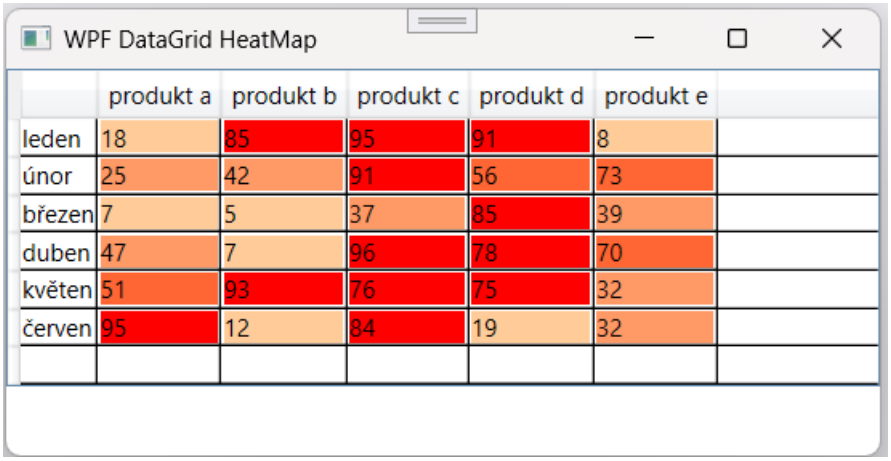
Je třeba vrátit se zpět ke sloupcům. Jde o zajímavou tematiku, kterou naplno využívá řada moderních knihoven pro práci s komponentami *DataGrid*. Jmenovitě *DataColumnTemplate* nabízí možnost specifikovat vlastní sloupec. Jistě lze vymyslet mnoho způsobů a kombinací, jak této možnosti naplno využít. Příklady definic lze nalézt v XAML souboru hlavního okna ukázkové aplikace. Pro vlastní úpravu lze využít třídy implementujících rozhraní *IValueConverter*. Tento přístup byl aplikován například na sloupec Hodnocení. Na základě ukázky lze *DataColumnTemplate* jednoznačně doporučit. Pro editaci bylo využito *INotifyPropertyChanged* na vlastnosti *Salary*.



Obrázek 19: Ukázka možností sloupců pro DataGrid

Zdroj: Nathan (2010), vlastní zpracování

DataGrid je v zde uveden jako zásadní komponenta pro práci s daty, jak již bylo uvedeno. Díky XAML se dá říci, že je zde mnoho možností, jak dále upravovat a vylepšovat vzhled této komponenty, a to již v základních WPF knihovnách bez použití balíčků. Avšak někdy je potřeba přistupovat pomocí kódu, tímto způsobem lze také nastavit různé styly. Možnosti demonstruje ukázka WPFHeatMap, která předvádí jednu ze základních technik vizualizace dat ve dvojrozměrném prostoru (Heat map). Na základě hodnoty v buňce je zvolena barva. Přístup je zde převážně bez XAML, což se v některých případech může využít.



Obrázek 20: Ukázka realizace Heat map v komponentě DataGrid

Zdroj: vlastní tvorba

Dále je umožněna validace dat. Příslušné chování lze specifikovat zděděním třídy *ValidationRule*. Validace souvisí s bindingy a jsou specifikovány u nich (v souborech XAML). V případě objektů ViewModel, což je doporučené, je tu *IDataErrorInfo*. Validace nastávají před samotným propsáním změn pomocí bindingu (datové vazby). Ukázky a jejich kompletní kód jsou součástí přílohy 1 (adresář DataGridColumns a WPFHeatMap).

2.5 DataGridView

Primární třídou určenou pro zobrazování dat pomocí tabulek je pro Windows Forms *DataGridView*. Ve WPF se využívá již zmíněná modernější alternativa *DataGrid*. Je poměrně běžnou možností propojení *DataTable* a vizuální komponenty nabízející širokou škálu přizpůsobení vzhledu a funkcionality. Tato komponenta umožňuje přizpůsobovat řádky, sloupce či konkrétní buňky gridu. Vlastnost *DataSource* slouží ke specifikaci zdroje dat. *DataGridView* obvykle využívá propojení s daty *TwoWay*. Obecný *DataGridView* může zobrazovat data v několika módech, tj. s datovou vazbou, bez datové vazby a virtuální. První z nich, mód bez datové vazby, je určen pro méně dat. Grid se zde nenapojuje přímo na zdroj dat pomocí *DataSource*, místo toho se musí daná data naplnit do gridu samostatně. Tato možnost může být výhodná, pokud chceme zobrazit data pouze pro čtení, jinak se vyplatí mód s datovou vazbou založený na propojení s datovým zdrojem pomocí *DataSource*. Pro virtuální mód musí být nastavena booleovská vlastnost *DataGridView.VirtualMode* na true. Primární využití najde při použití řešení s datovou vazbou, kde ho lze využít na doplňkové sloupce, které nejsou součástí zdroje dat napojeného pomocí *DataSource*. *DataGridView* umožňuje efektivně zachytávat události (eventy). Lze sem zařadit možnosti jako eventy reagující na uživatele (kliknutí, fokus myši), nastavení rozložení, vzhledu a podobně. Mezi ty běžně využívané zařaďme především *KeyDown*, *CellClick*, *CellContentClick*, *CellFormatting*, *CellPainting*, *CellValueChanged*, *SelectionChanged*. První z nich, *KeyDown*, je vyvolána při stisknutí klávesy za předpokladu, že grid má fokus. Následně lze provádět libovolné akce, např. nabídnout možnost exportu dat z gridu po stisknutí klávesy. Častá je také práce s buňkou, výše uvedené eventy umožňují zachytávat kliknutí (*CellClick*) a kliknutí na obsah buňky (*CellContentClick*). Pomocí těchto rozšíření lze korigovat chování gridu v různých situacích, což je užitečné a může nalézt uplatnění při vytváření sloupce zobrazujícího detail. *CellFormatting* se často využívá v případě zvýraznění textu buňky. *CellPainting* je vhodným místem pro provedení změn vlastního vykreslování buňky pomocí specifikovaných stylů, tj. lze přidat často využívané podmíněné zvýraznění a jiné. Data lze řadit pomocí metody *Sort* a vybrané řádky se získávají vlastností *SelectedRows,* jejíž ekvivalent pro sloupce se nazývá *SelectedColumns*. (Microsoft, 2025c)

Na pozadí této grafické komponenty je kód, kterým se definuje vzhled celého gridu, protože ten standardní je poměrně strohý. Většinu nastavení lze logicky odvodit s využitím vývojového prostředí a jeho zabudovaného našeptávače. *EnableHeaderVisualStyles* určuje, zda půjde použít vlastní styly pro vykreslení hlavičky. Pokud se nastaví na true, využijí se defaultní systémové styly, jinak se při nastavení na false mohou využít vlastní, jako je *BackgroundColor* apod. Ukázka je vcelku vysvětlující, proto autor usoudil, že není nutné detailně popisovat každý příkaz. Mezi ty zajímavé patří určitě *ForeColor*, vlastnost určující barvu písma, a *BackColor* pro nastavení barvy pozadí buňky. Dále lze přidat restrikci modifikace vzhledu gridu, jako je zamezení přidání řádku (*AllowUserToAddRows)* či možnost výběru více řádků zároveň (*MultiSelect*). Možností, co nastavit, je zde poměrně mnoho, proto se vyplatí vytvoření vlastní metody, která bude příkazy zapouzdřovat.

Níže je uvedena ukázka, která si klade za cíl demonstrovat využití obslužné metody k přidání sloupců do *DataGridView.* Vlastnost *HeaderText* umožňuje nastavit popis sloupce, který se zobrazí v hlavičce gridu. Samotné přidání a odebrání sloupce z kolekce *Columns* je realizováno využitím metod *Add* a *Remove*. Ukázky jsou opět součástí přílohy 1 (adresář DataGridView).

```
if (selectedIndex == 0)
{
    // typ 1. Sloupec TextBox
    DataGridViewTextBoxColumn column = new DataGridViewTextBoxColumn();
    column.Name = textBoxColumnAdd.Text;
    column.Width = 50;
    column.HeaderText = textBoxColumnAdd.Text;
    // přidání sloupce do datagridview
    dataGridViewAddColumns.Columns.Add(column);
}
```

Obrázek 21: Ukázka přidávání sloupců do DataGridView

Zdroj: vlastní tvorba

2.6 Další komponenty pro práci s daty

V prostředí C#/.NET lze vizualizovat data různými způsoby. Konkrétně pro WPF a Windows Forms jsou dostupné níže uvedené a popsané komponenty. Je správné sem zařadit třídy či komponenty, které obstarávají přípravu dat pro zobrazení na pozadí. Programovací jazyk C# dále umožňuje vytvořit vlastní znovupoužitelné komponenty pomocí *UserControl*. Účelem není představení každé možné komponenty, nýbrž jejich výběru s ukázkami praktického využití. Samozřejmě může čtenáře napadnout, že některé prezentované informace mohou být dále diskutovány, k čemuž jednotlivá témata vybízejí, protože bylo dle uvedených zdrojů čerpáno především z oficiální technické dokumentace, kde jsou některé informace týkající se možností dalšího využití zastoupeny spíše v omezenější míře.

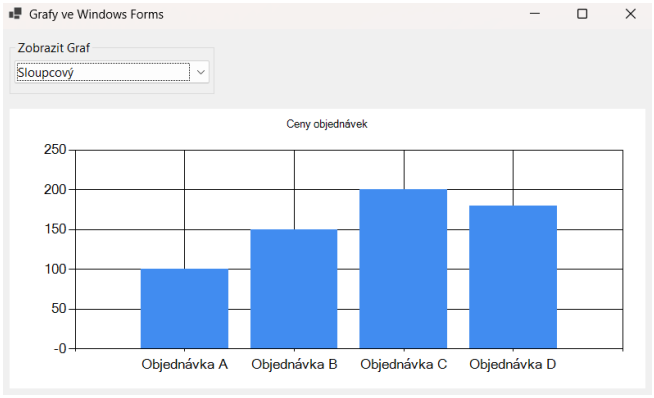
XAML je jazyk, ve kterém lze definovat vzhled a do určité míry chování komponent v rámci jazyka C# s použitím WPF. Struktura vychází z již diskutovaného XML. Ve WPF se jedná o stěžejní součást projektu. Definuje se obvykle ručně nebo s použitím vývojového prostředí. Lze ho chápat jako rozšíření XML o prvky programovacího jazyka. Ukázka je v části práce zabývající se datovými bindingy.

2.6.1 Chart

Pokud nastane situace, kdy je nutné vizualizovat data, jmenovitě v souhrnech, stojí za zvážení zobrazení dat v podobě grafů. Standardní knihovny jazyka C# mají balíček DataVisualization instalovatelný do projektu pomocí NuGet. Poté se může začít s vykreslováním i složitějších grafů. Knihovna je dostupná pro WPF i pro Windows Forms. Pokročilejší možnosti jsou realizovatelné volně dostupnými knihovnami. Celkově lze říci, že v programátorské praxi je výhodnější využít pokročilejších knihoven, jako je ScottPlot. Co se týče podpory složitějšího vykreslování, lze doporučit WPF a další dostupné knihovny.

Obecný graf se skládá z *ChartAreas* a *Series*. *ChartAreas*, jak z názvu vyplývá, umožňují rozdělit graf do více částí. Pomocí této třídy lze definovat popisy souřadnic X, Y a vlastnosti mřížky grafu. Jsou uloženy v *ChartAreaCollection*. Třída *Series* umožňuje reprezentovat hromadně body grafu. Pro vykreslení lze použít jednotlivé *DataPointy* uložené do *Series*, tj. je nutné dané body vytvořit a uložit. To vše je možné naprogramovat při vykreslování grafů, pro které se načítají data pouze před samotným vykreslením. V případě nutnosti překreslení se musí jednotlivé *DataPointy* opět vytvořit a znovu předat dle průzkumu autora práce. V případě grafů, které počítají s aktuálními

daty, je to však problém, je potřeba graf manuálně měnit při každé změně dat. To naznačuje možnost využití datových vazeb, které lze využít a ušetřit si práci. Třída *Chart* má vlastnost *DataSource*, která funguje obdobně jako u dalších komponent. V ukázce bylo do určité míry čerpáno ze zdroje C# Corner (Kala, 2025). Kompletní kód ukázky je uveden v příloze 1 (adresář WinformsChart).



Obrázek 22: Grafy ve Windows Forms

Zdroj: C# Corner 2025, vlastní zpracování

2.6.2 ComboBox

WPF ComboBox

S použitím knihovny WPF lze ušetřit nemálo práce, protože toho lze hodně definovat přímo v XAML. Níže uvedená ukázka využívá pouze možností XAML a dále není rozšířena o vlastní kód. Zobrazení aktuálně vybrané komponenty probíhá pomocí bindingu. Tímto způsobem se sem ukázka hodí, protože pouze pracuje s daty a vlastní C# kód není potřeba, vše se již děje na pozadí spuštěné ukázky.

```
<Window.Resources>
    <Style TargetType="{x:Type ComboBoxItem}" x:Key="DefaultComboBoxItemStyle">
        <Setter Property="Foreground" Value="Black"/> <!-- override -->
        <Setter Property="Background" Value="LightYellow"/>
        <Setter Property="Padding" Value="4,4"/>
    </Style>
    <Style x:Key="DefaultComboBoxStyle" TargetType="{x:Type ComboBox}">
        <Setter Property="Foreground" Value="MediumTurquoise"/>
        <!--<Setter Property="ItemContainerStyle" Value="{StaticResource DefaultComboBoxItemStyle}"/>-->
    </Style>
</Window.Resources>
<Grid Margin="20">
    <StackPanel VerticalAlignment="Top">
        <ComboBox x:Name="MainComboBox"
                  Height="31"
                  FontSize="16"
                  SelectedIndex="0"
                  Style="{StaticResource DefaultComboBoxStyle}"
                  ItemContainerStyle="{StaticResource DefaultComboBoxItemStyle}">
            <ComboBoxItem  Content="Objednávka A"/>
            <ComboBoxItem Content="Objednávka B"/>
            <ComboBoxItem Content="Objednávka C"/>
        </ComboBox>

        <Label Content="Vybraná možnost:"
               FontWeight="SemiBold"/>

        <TextBlock FontSize="14" Text="{Binding ElementName=MainComboBox,
                   Path=SelectedItem.Content}" x:Name="MainContent" Margin="5,5,5,5"/>
    </StackPanel>
</Grid>
```

Obrázek 23: Ukázka XAML pro WPF ComboBox

Zdroj: vlastní tvorba

Windows Forms ComboBox

Často skloňovaný *ComboBox*, tedy ve své podstatě rozbalovací výběr, najde své uplatnění v široké škále aplikací. Pomocí této komponenty lze dynamicky měnit obsah okna aplikace výběrem z dostupných možností, v případě načítání možností z databáze je možné mít výběr více dynamický. K tomu je opět výhodné využití datových vazeb. Vlastnost *DisplayMember* určuje, jaká vlastnost provázaného objektu se bude zobrazovat ve výběru této grafické komponenty, *ValueMember* slouží k uložení hodnoty výběru, kterou lze později získat vlastností *ComboBox.SelectedValue*. Umožňuje výběr pouze jedné hodnoty. Primárním obslužnou událostí, kterou lze upravit dle vlastního uvážení, je *SelectedIndexChanged* vyvolaný při změně vybrané položky seznamu.

Pro inicializaci a datový binding komponenty *ComboBox* lze využít kupříkladu způsob uvedený v ukázce níže. *DropDownStyle* se nastaví na *DropDownList*, poté se bude komponenta chovat tak, že nepůjde zapsat vlastní text do výběru. S využitím *DataSource* nastavíme zdroj dat na list objektů typu *ComboBoxItem*. *SelectedIndex* nastaví výběr na první element, takže na Objednávku A. Ukázky jsou uloženy v příloze 1 (adresář ComboBoxWPF a ComboBox).

```
private void InitComboBox()
{
    comboBox.DropDownStyle = ComboBoxStyle.DropDownList;
    _items = new List<ComboBoxItem>();
    _items.Add(new ComboBoxItem(1, "Objednávka A"));
    _items.Add(new ComboBoxItem(2, "Objednávka B"));
    _items.Add(new ComboBoxItem(3, "Objednávka C"));
    _items.Add(new ComboBoxItem(4, "Objednávka D"));

    comboBox.DataSource = _items;
    comboBox.DisplayMember = "Name";
    comboBox.ValueMember = "Id";
    comboBox.SelectedIndex = 0;
}
```

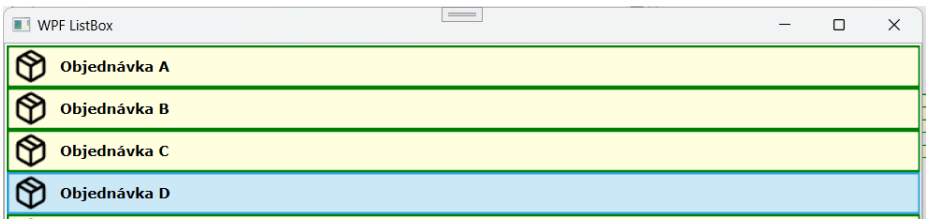
Obrázek 24: Ukázka inicializace ComboBoxu

Zdroj: vlastní tvorba

2.6.3 ListBox

WPF ListBox

Určitě stojí za to si projít ukázku ve WPF, jelikož se jedná o hojně využívaný přístup, je zde více možností. XAML se postará o moderní vzhled. Dostupné jsou módy *Single*, *Multiple*, *Extended*. V Single módu se chová element podobně jako *ComboBox. Multiple* mód uchovává vybrané položky v *SelectedItems*. *Extended* je kombinace *Single* módu s *Multiple*. Pro vybrání více položek je potřeba podržet *Shift* nebo *Control*.

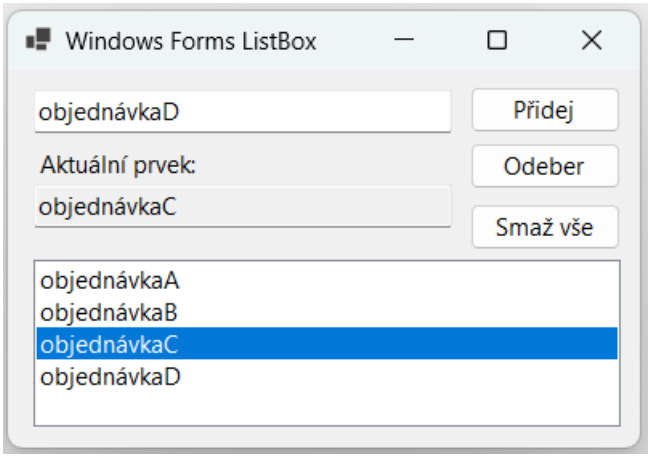


Obrázek 25: WPF ListBox

Zdroj: vlastní tvorba

Windows Forms ListBox

ListBox je v jazyce C# určený pro práci se seznamem hodnot. Jeho účel je v zobrazení všech možných hodnot najednou (na rozdíl od *ComboBoxu*). Vlastností *SelectionMode* lze specifikovat kolik hodnot ze seznamu lze označit. Pokud příjde slovo na výběr více hodnot, jsou tu dva typy, *SelectionMode.MultiExtended* a *SelectionMode.MultiSimple*. *MultiExtended* podporuje výběr pomocí kláves *Shift* a *Control*, naopak *MultiSimple* reaguje pouze na označení a odznačení prvku seznamu myší. *ListBox* umožňuje propojení se zdrojem dat použitím datových vazeb. Způsob implementace ukázky je na obrázku níže, je zde zdrojový kód pro tlačítko Přidej, Odeber a Odeber vše. *ListBox* se ovládá s využitím vlastnosti *Items* a metodami *Add*, *RemoveAt*, *Clear*. Z pohledu dat jde opět o nedocenitelnou komponentu. Ukázky jsou v příloze 1 (adresář ListBoxWPF a ListBox).



Obrázek 26: Grafický prvek ListBox
Zdroj: vlastní tvorba

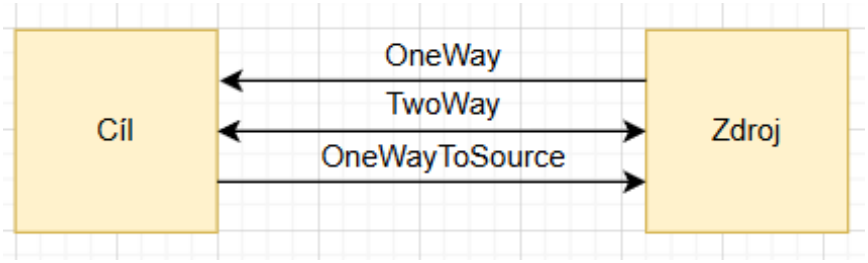
2.6.4 TextBox a TextBlock

Dobře známé komponenty *TextBox* a *TextBlock* zobrazují textová data. *TextBlock* je pouze součástí WPF. Vlastnost *Text* nastaví, co se má zobrazit uživateli na obrazovce, kromě toho se na příslušném objektu třídy *TextBox* dále nastavuje mimo jiné barva písma. Využití je různorodé, ať už pro zobrazení hlášek systému, či jen pro zobrazení daného textu. *TextBox* by měl sloužit pro text, který se dá změnit, na statické popisy je vhodné využít komponentu *Label*. V základním nastavení *TextBox* umožňuje přijímat vstup pouze z jednoho řádku, vlastnost *Multiline* umožní zadat víceřádkový vstup. Posuvníky se přidávají vlastností *ScrollBars*. Pokud je potřeba zadávat citlivé informace, je tu vlastnost *PasswordChar*. *TextBox* komponenty jsou součástí již předešlých ukázek. Ve WPF lze použít vlastnost *Text* pro data binding. Ve Windows Forms se používá kolekce *DataBindings*. Tu je možné využít ve spojení s ViewModely.

2.7 Data Bindings

Pro propojení vizuálních komponent se zdroji dat jsou určeny datové vazby, díky nimž je dosažitelné stavět snadněji udržovatelná uživatelská rozhraní, která nevyžadují složitou manuální správu jednotlivých propojení, tím pádem výrazně usnadňují práci. Vazby lze definovat programaticky a s využitím XAML (WPF). Základní metody propojení komponent se zdrojem dat

jsou zobrazeny na diagramu, zde byl použit program Draw.io (JGraph Ltd., 2025). Základní třídou pro tvorbu datových vazeb (Data Bindings) je *Binding* z jmenného prostoru *System.Windows.Data*, pro Windows Forms je to *System.Windows.Forms*. Ve Windows Forms je určena pro definování propojení s jednou vlastností, využití najde například ve spojení s *TextBoxem*. *BindingList* je naopak kolekce podporující automatické změny *TwoWay*. Existují dva základní typy datových vazeb, jednoduché a komplexní. Jednoduché se vážou na jednu vlastnost objektu a jeden prvek rozhraní. Komplexní se vážou na více prvků (například na sloupce gridu). Více o Windows Forms datových vazbách lze nalézt v dokumentaci (Microsoft, 2025b). Technologie WPF nabízí více možností, co se týče datových vazeb, proto má smysl si zde uvést tento odkaz. Zde se využívá definic ve formátu XAML. Pro detekování změn a zobrazení v grafickém rozhraní musí zdroj implementovat notifikační interface *INotifyPropertyChanged*. *OneWay* režim umožňuje přenos změn směrem ze zdroje do cíle, avšak opačný proces není umožněn. Ze své podstaty jde tedy o volbu vhodnou pro komponenty určené pouze ke čtení, například při zobrazování statického textu pomocí *TextBoxu*. Pro složitější vazby lze využít *TwoWay* přístup, při kterém jsou cíl a zdroj vzájemně propojeny a změny jsou propagovány do druhého v případě změny. Režim *OneWayToSource* je opakem *OneWay*. Alternativou k *OneWay* je *OneTime*, která pouze inicializuje cíl a následné další změny zdroje již nejsou propagovány do cíle. Diagram na obrázku 30 je v příloze 3 (DataFlows.drawio).



Obrázek 27: Schématické zobrazení základních datových toků

Zdroj: Microsoft (2025b), vlastní zpracování

Každý WPF binding se skládá ze čtyř hlavních částí, tj. z cílové komponenty, její vlastnosti, zdroje a cesty path (obvykle vlastnost zdroje). Vše lze definovat přímo v XAML. Zdroj je zpravidla reprezentován .NET objektem, ale není to restriktivní. Rozpoznávání vazeb je uskutečněno na základě vlastnosti *DataContext*. Klíčové slovo *Binding* obvykle určuje vlastnost, se kterou je provázána komponenta. Ukázka bindingů je v příloze 1 (adresář DataGridColumns). (*Microsoft, 2025b*)

```
<DataGridComboBoxColumn
    Header="Země"
    SelectedItemBinding="{Binding Country}"
    ItemsSource="{Binding Source={StaticResource countryEnum}}" />
<DataGridTextColumn Header="Jméno" Binding="{Binding FirstName}" Width="100">
    <DataGridTextColumn.CellStyle>
        <Style TargetType="DataGridCell">
            <Setter Property="Background" Value="MediumTurquoise"/>
        </Style>
    </DataGridTextColumn.CellStyle>
</DataGridTextColumn>
<DataGridTextColumn Header="Přijmení" Binding="{Binding LastName}"  Width="100"/>
<DataGridHyperlinkColumn Header="Info" Binding="{Binding InfoUrl}" IsReadOnly="False" Width="200"/>
```

Obrázek 28: Definovaní Bindingů v XAML

Zdroj: vlastní tvorba

2.8 Dotazovací jazyk LINQ

Užitečným nástrojem může být integrovaný dotazovací jazyk LINQ. Uveden byl již ve verzi C# 3.0 a byl vyvinut za účelem vyšší efektivity práce. Jeho účelem je redukce pracnosti aktivit spojených s filtrací dat, jejich vyhledáváním, transformováním a tříděním. Výhodou je, že není nutné instalovat balíčky s knihovnami, protože se jedná, jak již bylo psáno, o integrované rozšíření jazyka. LINQ je přizpůsoben pro práci s daty bez rozlišení jejich zdroje, takže je jedno, zda se filtruje kolekce či *DataTable*. Je však důležité, aby zdroj dat implementoval rozhraní *IEnumerable<T>* nebo *IQueryable<T>*. Lze tedy říci, že se dá využít prakticky vše, co lze iterovat. Pro data uložená v paměti se používá rozhraní *IEnumerable<T>*. *IQueryable<T>* využívá Entity Framework pro reprezentaci dotazů, a tím umožňuje odložit spuštění dotazu. (Griffiths, 2024)

Struktura dotazu vychází z jazyka SQL, ale v určitých směrech se liší. Její nespornou výhodou je, že všechny proměnné výrazu jsou silně typované. V první fázi je nutné obstarat data. Následuje vytvoření dotazu (query) a jeho spuštění pomocí cyklu. Základní klasifikace dotazů LINQ se dělí dle momentu získání dat na vyhodnocené okamžitě a s odloženým vyhodnocením. Výsledky volané pomocí metod Count či Average jsou vyhodnocovány ihned, zatímco dotazy s využitím cyklu jsou získány až při iteraci cyklem. Pro vynucení okamžitého navrácení výsledku lze využít metody ToArray nebo ToList. Kompilátor jazyka C# během překladu mění syntaxi dotazu na volání metod. Je tedy možné volat přímo metody místo vytváření dotazu. Obě varianty jsou sémanticky totožné, dotazy se dají kombinovat s voláním metod, v některých případech je to dokonce nutné. Metody zpravidla využívají lambda výrazy. Lambda výraz je anonymní funkce, kterou lze kompletně definovat přímo ve výrazu. Dotazy využívají klíčová slova jako select, from, where, orderby či group. Podobně jako v SQL i zde slouží select k získání konkrétních dat ze zdroje, from určuje, odkud se budou data brát (zdroj dat), where filtruje dle podmínek, které lze řetězit jako v SQL. Pro řazení je zde orderby a pro seskupení group. Pomocí LINQ to SQL lze dotazovat databáze bez znalosti SQL, což může být výhodné. Obdobně funguje i LINQ to XML pro práci s XML soubory. Pro použití je nutné přidat do hlavičky souboru příkaz using System.Linq. Je tu také možnost vzájemné integrace formátů, tj. načtená data z SQL databáze lze transformovat do jiných formátů. Ve většině případů LINQ usnadňuje čitelnost zápisu programového kódu. Celkově je více providerů LINQ. Dalšími jsou LINQ to Objects, LINQ to Entities, LINQ to Datasets. Jednotlivé dotazy lze použít vícekrát nad různými zdroji dat bez nutnosti psát dotaz pro každý zdroj znovu, čímž se lze vyhnout redundantnímu kódu. Jazyk LINQ kombinuje dobré programátorské návyky a usnadňuje práci s daty. Ukázka je v příloze 1 (LINQ). (Griffiths, 2025)

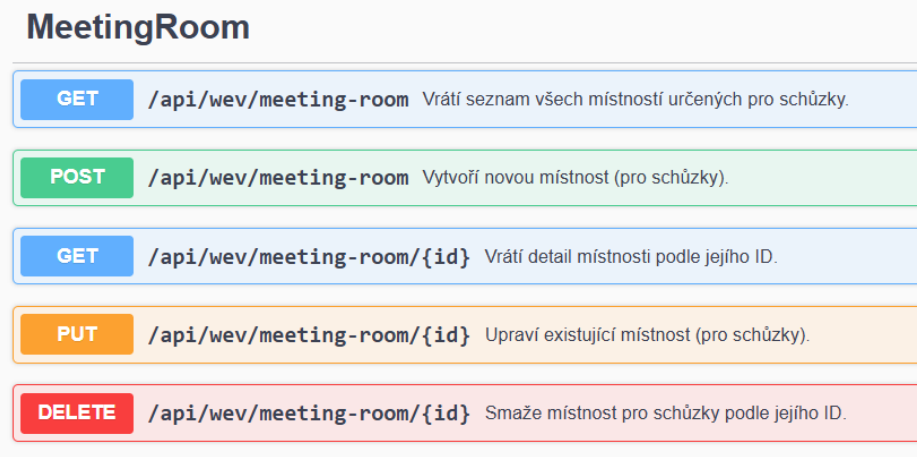
```
var queryData = from Order o in data2
                where o.ProductName.StartsWith("V")
                select o.Id;
Console.WriteLine("Id kde ProductName zacina na V: ");
foreach (var product in queryData) {
    Console.WriteLine($"Produkt: {product.ToString()}");
}
```

Obrázek 29: Dotazovací jazyk LINQ

Zdroj: vlastní tvorba

2.9 Webové API jako zdroj dat

Přístup ke zdrojům dat pomocí API se v dnešní době dostává čím dál tím více do popředí. Výhodou oproti ostatním přístupům je, že mohou nabízet přizpůsobené endpointy, které lze volat z klientských aplikací s danými parametry. Obvyklým formátem komunikace je již zmíněný JSON. Vlastní API obvykle dodržují standard REST (Representational State Transfer), taková API se pak označují jako RESTful. Mezi základní vlastnosti patří především oddělenost od klientských aplikací (stateless), jednotné rozhraní pro komunikaci a endpointy jednoznačně identifikující zdroj dat pro klienta.

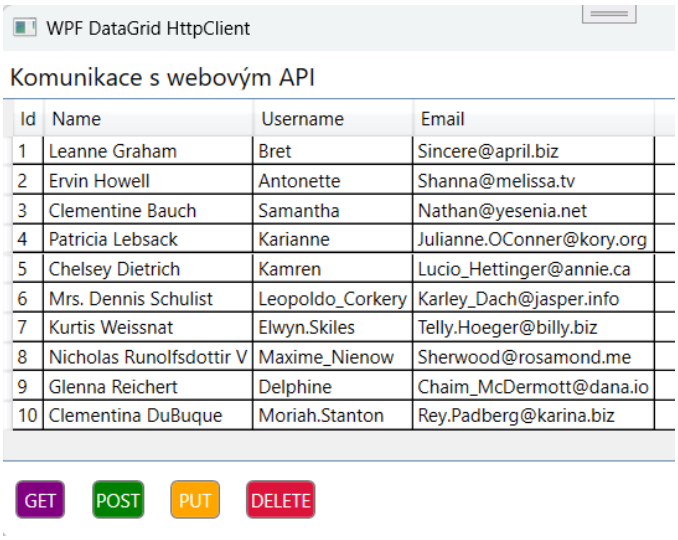


Obrázek 30: Příklad dokumentace k webovému API

Zdroj: vlastní tvorba

2.9.1 Komunikace s webovým API v jazyce C#

Aplikace často vyžadují propojení s webovým serverem. Komunikace zpravidla probíhá pomocí protokolu HTTP s využitím šifrování. Běžným formátem komunikace je JSON. Webové API je zpravidla napojené na databázi, tím pádem samotná aplikace může komunikovat pouze s API. Ověření desktopové aplikace je často řešeno pomocí tokenů JWT či API klíčů.



Id	Name	Username	Email
1	Leanne Graham	Bret	Sincere@april.biz
2	Ervin Howell	Antonette	Shanna@melissa.tv
3	Clementine Bauch	Samantha	Nathan@yesenia.net
4	Patricia Lebsack	Karianne	Julianne.OConner@kory.org
5	Chelsey Dietrich	Kamren	Lucio_Hettinger@annie.ca
6	Mrs. Dennis Schulist	Leopoldo_Corkery	Karley_Dach@jasper.info
7	Kurtis Weissnat	Elwyn.Skiles	Telly.Hoeger@billy.biz
8	Nicholas Runolfsdottir V	Maxime_Nienow	Sherwood@rosamond.me
9	Glenna Reichert	Delphine	Chaim_McDermott@dana.io
10	Clementina DuBuque	Moriah.Stanton	Rey.Padberg@karina.biz

Obrázek 31: Vizuální ukázka propojení DataGridu s webovým API

Zdroj: vlastní tvorba

Klient HTTP jazyka C# se nachází ve jmenném prostoru *System.Net.Http*. Umožňuje dotazování a posílání dat dle metod protokolu HTTP pro operace, jako jsou GET, POST a další. Po zpracování požadavku je vhodné JSON výsledek deserializovat, při posílání dat je naopak výhodné využít serializaci. Klient je uzpůsoben pro vytvoření pouze jedné instance pro celou aplikaci, v případě vytváření více instancí je nutné brát na vědomí, že se s každou další vytváří vlastní spojení, což může vést k nedostatku volných portů. Jednotlivé metody nabízejí své asynchronní alternativy. Ukázka využívá volně dostupné API na stránce jsonplaceholder.com, které je volně dostupné k použití (JSONPlaceholder, 2026). Je součástí přílohy 1 (DataGridHttpClient).

Na ukázce je znázorněna metoda *GetAllAsync*, která je uložena v obalovací třídě nad třídou *HttpClient*. Je pojmenována *HttpClientService*. Při vytváření objektu třídy *HttpClient* se rovnou přiřazuje vlastnost *BaseAddress*, jejíž typ je *Uri*. Právě zde se definuje základní adresa, vůči které bude ostatní volání endpointů relativní. Asynchronní metoda vrací *Task<HttpResponseMessage>*, pomocí klíčového slova *await* lze vyčkat na dokončení úlohy. Metoda *ReadAsStringAsync* pouze navrátí obsah odpovědi v podobě řetězce. Tato odpověď bývá zpravidla ve formátu JSON za předpokladu, že se provolává webové API. Následně dochází k již popisované deserializaci na seznam objektů. Další zajímavou třídou je HttpStatusCode, která reprezentuje navrácený status HTTP odpovědi.

```
/// <summary>
/// Provede get na všechny položky webového Api
/// </summary>
/// <returns></returns>
public async Task<List<UserSimple>> GetAllAsync()
{
    List<UserSimple> listData = new List<UserSimple>();

    var msg = await _httpClient.GetAsync("/users");

    string? strResponse = await msg.Content.ReadAsStringAsync();

    var ops = new JsonSerializerOptions
    {
        PropertyNameCaseInsensitive = true
    };

    if (strResponse != null)
    {
        List<UserSimple>? result = JsonSerializer.Deserialize<List<UserSimple>>(strResponse, ops);
        if (result != null)
            listData = result;
    }

    return listData;
}
```

Obrázek 32: Ukázka kódu realizace požadavku GET

Zdroj: vlastní tvorba

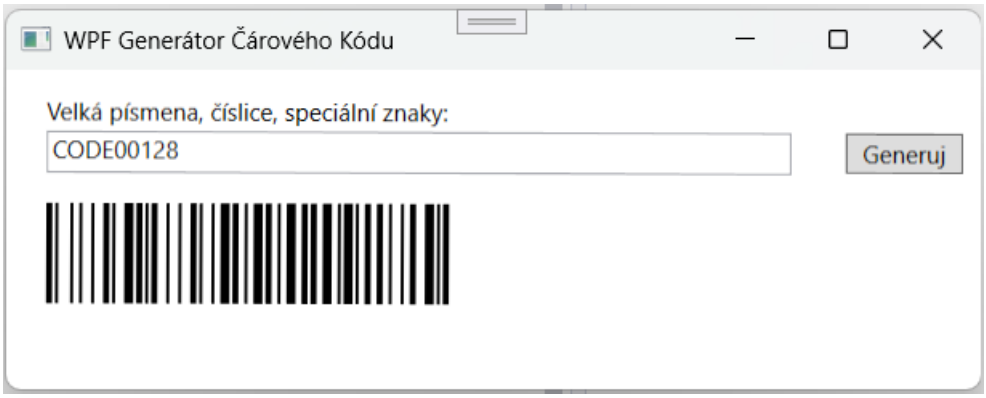
2.10 Asynchronní zpracování

Běžný přístup je blokující, to znamená, že pokud například načítáme data z velkého souboru, aplikace může zamrznout po dobu, dokud se data nenačtou, v případě, že někde nastane chyba, je nutné celý proces opakovat. Naštěstí jazyk C# nabízí postupy, jak se tomu vyvarovat. Jsou založeny na asynchronním přístupu.

Operace, která se má vykonat, je reprezentována instancí třídy *Task* či *Task<T>*. Tyto třídy se nacházejí ve jmenném prostoru *System.Threading.Tasks* a fungují na principu příslibu získání výsledku. Metoda *Run* třídy *Task* spouští danou operaci a vrací objekt *Task,* na kterém lze dále kontrolovat stav operace. Dalšími metodami jsou zejména *Wait* a *Delay*. *Wait* vyčká na dokončení operace, *Delay* umožňuje manuálně pozastavit vykonávání úlohy na určitý čas. Klíčové slovo *await*, oficiálně využívané od verze C# 5, umožňuje vyčkat na dokončení asynchronní operace, tedy zpracuje výsledek *Task* a případně vyhodí výjimku, pokud nastala během zpracování operace chyba. S *await* je svázané druhé klíčové slovo, *async*. To předává informaci, že následující část kódu, metoda, obsahuje alespoň jedno asynchronní volání. Může se využít pouze u metod vracejících void či *Task*, *Task<T>*. Metoda, která je označena jako *async,* může mít ve svém těle jedno a více volání *await*. Během asynchronního čekání pomocí *await* nedochází k blokování kódu, který *async* metodu zavolal. *Async* metody nevyžadují vícevkláknový přístup, protože *async* a *await* při svém vzniku nevytvářejí další vlákno, místo toho běží na synchronizačním kontextu. *Async* a *await* nesouvisí s *Task.Run*, které naopak vytváří nové vlákno pro úlohu. Využití *async* a *await* najde při vstupně-výstupních operacích, avšak v případě úloh závislých na CPU je vhodné využít *Task.Run*. Ukázka využití je součástí složky DataGridHttpClient v příloze 1. Pomocí async a await se řeší dostupnost dat.

2.11 Další grafická reprezentace dat

Součástí stávající práce může být rozšíření o zpracování grafických dat, což je aktuální problematika. Avšak vzhledem k povaze a obsahu práce se sem ukázky příliš nehodily. Čárový kód *CODE128* reprezentuje jednotlivé znaky jako kombinaci čar s určitou šířkou. Dle dostupných online informací existují tři typy *A*, *B*, *C*. Ukázka realizuje typ *A*, zde je každý znak reprezentován šesti šířkami. Čáry jsou vykresleny na okno jako instance třídy *Rectangle*, vzniknou přidáním na stanovenou pozici v *Canvasu*. Šířka čar je ve výchozím nastavení jeden a půl pixelu.



Obrázek 33: Ukázka čárový kód

Zdroj: vlastní tvorba

Podklady k teorii a realizaci návrhu algoritmu byly čerpány ze zdroje Wikipedie (Wikipedia contributors, 2025). Rozšíření programu by spočívalo v přidání ostatních typů (B, C). Otestovat generování je uskutečnitelné i volně dostupnými programy pro čtení čárových kódů. Projekt je přiložen v příloze 1 (adresář SimpleBarcode).

2.12 Dostupné zdroje a literatura

Čerpáno bylo z cizojazyčné literatury a oficiální dokumentace jazyka C# od společnosti Microsoft. Oficiální dokumentace je jedním z důvěryhodných zdrojů informací, které lze nalézt na internetu. Kromě již citovaných zdrojů jsou dostupné ještě další knihy a online zdroje. Zde je zastoupena v určité míře i česká literatura. K prozkoumání možností Windows Forms lze využít především oficiální dokumentaci Microsoftu. Pro tuto technologii je méně vhodné použít dostupné knihy, protože obvykle uvádějí pouze základní poznatky. Dalším zdrojem mohou být výuková videa či různé nástroje umělé inteligence. Autor během zpracování využil především nabyté zkušenosti z předmětů během studia a oficiální dokumentaci. Pro technologii WPF je dostupných informací více, včetně odborné literatury a různých dalších výukových materiálů. Vždy však záleží na tom, jakou informaci je potřeba vyhledat. Co se týče často skloňovaných komponent, jako jsou DataSet, DataTable, DataGrid a DataGridView, je literatury relativně dostatek, i když ne vždy je aktuální. Jednotlivé zdroje byly uvedeny v textu.

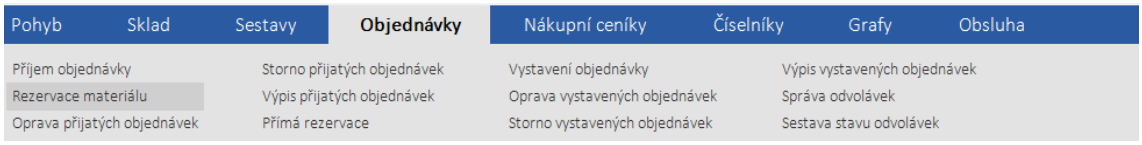
V oficiální dokumentaci od společnosti Microsoft je možné nalézt aktuální informace k daným verzím. Jednou z hlavních výhod je snadná dostupnost a právě ta aktuálnost, což může být stěžejní pro práci s konkrétní verzí .NET. V kombinaci s vhodně zvolenou literaturou může být dokumentace dobrým zdrojem informací. V tomto případě je vhodné mít literaturu jako kuchařku. Pro rychlé vyhledávání v dokumentaci lze využít již zmíněné nástroje umělé inteligence, ale pro tvorbu technických odpovědí jsou méně relevantní. Součástí dokumentace jsou obvykle třídy z daných jmenných prostorů s popisem jejich metod.

Ze zdrojů byly obvykle čerpány informace na základě vlastní úvahy, veškeré zdroje zpravidla obsahovaly více informací, než bylo vhodné pro použití v rámci zpracovávaného tématu, takže byla použita pouze část těchto informací.

3 Návrh a analýza objednávkového systému

V této kapitole jsou představeny návrh a analýza zamýšleného objednávkového systému demonstrující práci s daty.

Softwarové řešení zjednodušuje běžné činnosti a automatizuje je, zároveň přináší možnost využití a zpracování datových reportů. Systémy se často upravují na míru. Jedná se obvykle o samostatnou aplikaci, se kterou komunikuje odpovědná osoba ze strany dodavatele, nebo webovou aplikaci určenou pro všechny. Pomocí některých objednávkových systémů lze notifikovat zákazníka o stavu jeho objednávky. Moderní systémy samozřejmě mají i pokročilejší funkce, které se mohou prolínat do dalších odvětví. Případně lze tyto systémy propojit s více zdroji dat a tím umožnit integrovat objednávky z více aplikací do jednoho systému. Možnou integrací jsou účetní programy, skladové systémy a ostatní. Přínos tvorby zdrojového kódu aplikace, která pracuje s objednávkami, spočívá v odstínění obecného uživatele od manualní práce s objednávkami (například evidence pomocí excelovského souboru). Zároveň umožňuje spravovat databázi, kde jsou data uložena bezpečně a lépe strukturovaně. (MultiBuy, 2025)



Obrázek 34: Záložka Objednávky v programu Byznys

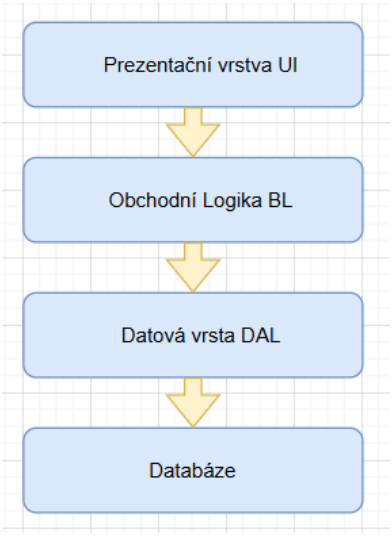
Zdroj: vlastní zpracování, (Seyfor, 2026)

3.1 Volba architektury aplikace

Projekt bude realizován jako desktopová aplikace v technologii .NET. Nebude se jednat o multiplatformní aplikaci, nýbrž bude určena výhradně pro prostředí MS Windows. Aplikace může sloužit pro potřeby firmy, která vyžaduje evidenci objednávek. Platforma .NET má silné prostředky pro tvorbu těchto aplikací. Alternativně je možné využít i jiné technologie a postupy, možností, jak udělat stejnou věc, je vždy více. Programovací jazyk C#/.NET je vhodný technologický základ.

Platforma .NET je vhodná pro tvorbu desktopových aplikací. Mezi stabilní a ozkoušené možnosti patří především WPF, Windows Forms nebo MAUI. Windows Forms je řešení architektury aplikací určených pro MS Windows a je založeno na umisťování vizuálních komponent na okno aplikace. Ve své podstatě zde není tolik možností, co se týče uživatelského rozhraní v porovnání s ostatními možnostmi .NET, ale to neznamená, že nelze na tomto základu postavit kompletní aplikaci. Hlavním vizuálním prvkem je formulář, který je určen pro interakci s uživatelem, právě zde dochází k zobrazování dat. Jednotlivé události, jako je kliknutí, jsou zachycovány pomocí eventů. Výběr vizuálních komponent je poměrně široký, pokud je však potřeba vytvořit vlastní, je tu *UserControl*. Definovat rozvržení komponent na okně aplikace lze pomocí *FlowLayoutu* nebo jsou zde vlastnosti *Dock* a *Anchor*. Windows Forms mohou být využity pro tvorbu formulářových aplikací. V případě WPF je tu Grid, se kterým se pracuje přímo v XAML. Výborně se hodí pro náročnější formulářové aplikace a využívá vektorový renderovací engine, což z něj činí vhodnou volbu pro aplikace pracující s grafikou. V porovnání s Windows Forms má i více možností v případě tvorby layoutů a animací.

Na základě dostupných informací byla vybrána technologie Windows Forms, protože je vhodná pro rychlé prototypování, zároveň se však počítá s rozšířením nebo případnou migrací. Většinu aplikací, se kterými se v dnešní době setkáváme, lze vnitřně rozdělit do několika logických vrstev, každá z těchto vrstev ohraničuje určitý celek aplikace. Členění má dobrý význam především ve vývoji, kde se tímto způsobem oddělí části aplikace do samostatných logických celků. Takovéto členění dále zjednodušuje proces refaktoringu a testování softwaru.



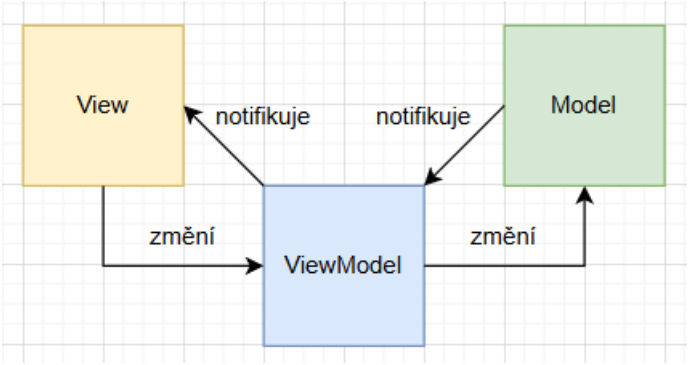
Obrázek 35: Vícevrstvá architektura aplikace
Zdroj: vlastní tvorba

Z výše uvedeného diagramu lze vyčíst, že v ideálním případě se aplikace dělí celkem do čtyř vrstev. První vrstva je prezentační. Zde se prezentují data získaná obvykle z vrstvy obchodní logiky. Můžeme sem zařadit komponenty, jako je například *DataGrid*. Obecně lze říci, že se jedná o část aplikace, se kterou interaguje uživatel. Další vrstvou je obchodní logika, ta bývá označována anglickým výrazem *business logic* (BL). Z pohledu vývoje se jedná o nejčastěji využívanou vrstvu, protože je zde implementována programová logika celé aplikace. Patří sem zejména servisní třídy. Diagram je součástí přílohy 3 (NTierArch.drawio).

Předposlední vrstvou je datová vrstva, sloužící pro předávání dat s předešlou vrstvou. Na rozdíl od ní zde však neprobíhají žádné logické operace s daty. Je možné i žádoucí data filtrovat či agregovat na základě dat předaných pomocí parametrů. Do vrstvy se zařazuje také Entity Framework.

Na závěr ještě uveďme čtvrtou vrstvu, databáze je hlavním zdrojem dat pro většinu aplikací. Na data se lze dotazovat pomocí standardizovaného jazyka SQL. V současné době jsou však dostupná i jiná řešení a alternativy klasických relačních databází.

Většina desktopových aplikací využívá v určité míře vzor *Model-View-ViewModel*. Svým způsobem je podobný známějšímu vzoru *Model-View-Controller*, který je mimo jiné v podstatě standardem. Model je zde určený pro reprezentaci doménových dat. View určuje, jak jsou data zobrazena uživateli. ViewModel je propojuje a umožňuje tím vzájemnou komunikaci, reprezentuje abstrakci stavu View z pohledu dat, komunikuje s Modelem a formátuje data pro View. V praxi se tento návrhový vzor používá především kvůli propojení ViewModelu s View pomocí datových vazeb. Obrázek znázorňuje diagram z přílohy 3 (MVVM.drawio).



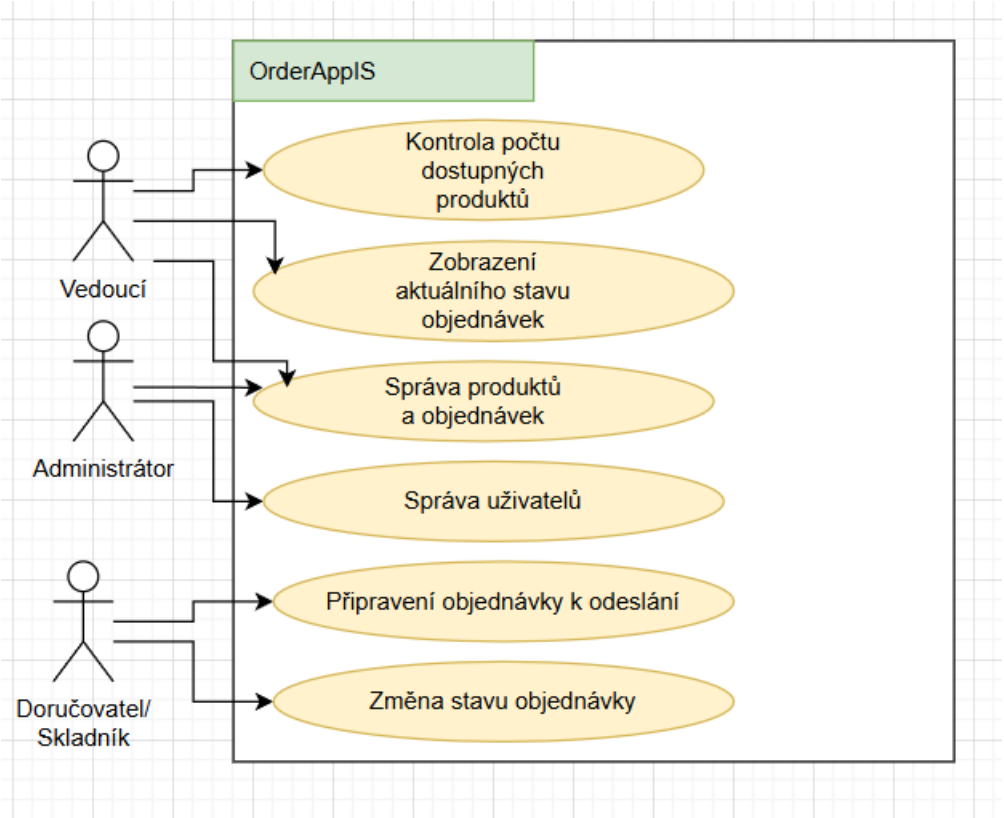
Obrázek 36: Schéma návrhového vzoru Model-View-ViewModel

Zdroj: vlastní tvorba

Dalším faktorem je volba, zda vytvářet webové API či ne, u jednodušších aplikací, ke kterým přistupuje méně uživatelů a mají propojení s databází, není přímo nutné. Využití API nabízí možnost vytvoření více typů klientských aplikací a více možností, co se týče propojování s ostatními systémy. Aplikace pro kterou se nyní vytváří analýza nevyžaduje další klientské aplikace, avšak je vhodné si zde uvést tuto možnost, protože tímto způsobem fungují větší systémy. Výhodou také je, že hlavní aplikační logika je nápsána právě v API, takže při změnách a správném provedení lze zjednodušit distribuci těchto změn všem klientským aplikacím najednou, zároveň je API jediný bod, který komunikuje s databází, což zlepšuje bezpečnost.

3.2 Model případů užití

Hlavními aktéry aplikace by měli být především vedoucí (manažer), skladník (doručovatel), administrátor a zákazník. Pro zákazníka by mělo vzniknout samostatné rozhraní v příštích verzích. Vedoucí schválí objednávku a vyzve zákazníka k úhradě. Skladník připraví a vyřídí zaplacenou objednávku. Cílem projektu je vytvoření systému, který bude tyto akce podporovat, včetně základní správy uživatelských účtů. Administrátorovi je umožněno spravovat uživatele systému a všechny objednávky, systém podporuje přístup na základě hesel a loginů uživatelů. Součástí je dále evidence plateb a informace o zákaznících a dodání. Pro účely manažera a skladníka by měla být součástí aplikace zjednodušená evidence skladových zásob, musí být zřejmé, zda se produkt nachází na skladě, či ne, a musí být uvedena jeho aktuální cena. Diagram je v příloze 3 (soubor UseCase.drawio). Navrhovaný systém bude fungovat formou zástupu zákazníků (objednávky vytvoří manažer). Bude navrženo vhodné rozšíření rozhraní pro zákazníky. Diagram slouží především k jasnému znázornění funkcionality navrhovaného systému z pohledu daných aktérů.



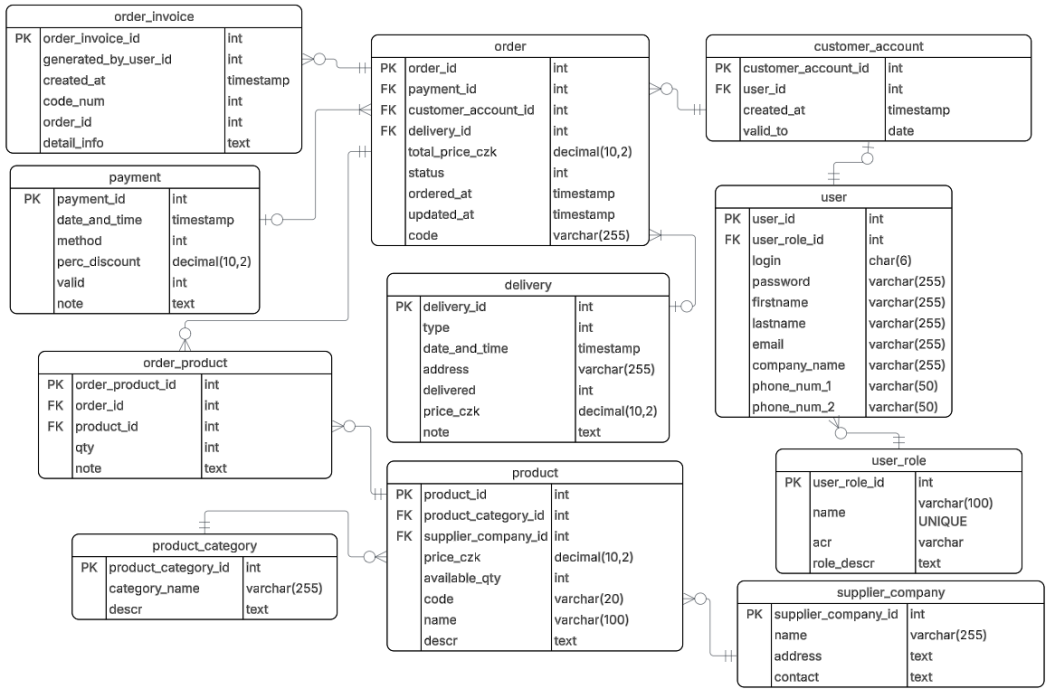
Obrázek 37: Model případů užití objednávkového systému OrderAppIS

Zdroj: vlastní tvorba

3.3 Volba databázového systému a zpracování ERD

Téměř každý systém musí ukládat data. Tradiční formou je využití relačních databázových systémů. Pro prostředí Windows se nabízejí různá řešení, včetně PostgreSQL. Relační databázové systémy jsou dobrou volbou ve většině případů, jmenovitě pro formulářové aplikace, protože umožňují strukturovat ukládaná data. Mimo jiné jsou již roky osvědčenými nástroji. Fungují na základě předem definovaného typovaného datového modelu. Základní komponentou je zde tabulka reprezentující jeden logický celek dat. Vztahy mezi tabulkami označujeme jako relace, odtud název relační. Tento typ databází nabízí atomicitu transakcí, konzistenci dat, izolaci transakcí a perzistenci dat v případě výpadku či jiné nestandardní situace. Dotazy nad databází lze realizovat pomocí SQL.

Nejvhodnější volbou může být PostgreSQL, protože se dobře hodí pro datově intenzivní aplikace a z pohledu práce s daty toho nabízí poměrně mnoho. Pro uchovávání lokálních dat je naopak výborným pomocníkem SQLite, který je přenositelný. Microsoft SQL Server se hodí pro enterprise level řešení s pokročilou analýzou dat, ale PostgreSQL toho také nabízí dostatek.



Obrázek 38: ER diagram

Zdroj: vlastní tvorba

Každá tabulka návrhu je opatřena primárním klíčem, který zaručuje jedinečnost uchovávaného záznamu. Uživatel aplikace je reprezentován tabulkou *user*, která eviduje jeho login, heslo, jméno a příjmení, e-mail, telefonní číslo a název firmy, pod kterou je evidován. Každý uživatel má svou roli, aplikace má celkem čtyři hlavní aktéry, vedoucího (manažera), skladníka (doručovatele), administrátora a zákazníka. Konečně se lze přesunout k objednávkám samotným, které jsou reprezentovány tabulkou *order*. Objednávka je navázána na platbu a doručení, má své položky. Každá objednávka by měla mít alespoň jednu položku, která je dále propojena s produktem a jeho kategorií. Produkt má sloupec *available_qty*, který určuje aktuální počet dostupných kusů na skladě. Objednávka se může nacházet ve stavech jako vytvořena, zpracovává se, připravena k odeslání, doručuje se, dokončena a zrušena. Platba obdobně může nabývat stavu platná, storno. Doručení je ve stavech nedoručeno, doručeno. Tabulka *supplier_company* je určená pro plnění daty o dodavatelích, ale zde je jen jako možné rozšíření informací o produktu. ER model databáze je znázorněn na výše uvedeném obrázku a ve větším provedení je v příloze A.6. K dispozici je také v elektronické verzi v souboru ER-model.png (příloha 3). Následuje popis důležitých tabulek *order* a *order_product*.

3.3.1 Tabulka order

Hlavní tabulkou celého schématu databáze je tabulka *order*. Obsahuje sloupce jako je celková cena objednávky (*total_price_czk*), *status*, *ordered_at*, *updated_at*, *code*. Řešení, která by vyhovovala zadání, je jistě více. Toto řešení využívá sloupec celková cena objednávky přímo v tabulce, takže se nebude muset dopočítávat přes položky. Řešení má své výhody a nevýhody. Výhodou je, že lze snadno sledovat přesnou cenu objednávky v daný moment (včetně historie). Nevýhodou je nutnost dopočtu cen. Alternativním řešením by mohlo být umístění ceny do položek objednávky. Ostatní sloupce by měly být vypovídající.

3.3.2 Tabulka order_product

Přidruženou tabulkou k *order* je *order_product*. Reprezentuje položku objednávky. Mezi její sloupce patří počet kusů a poznámka. Zároveň by zde mohla přibýt již zmíněná cena položky. Ta by mohla být uvedena jako cena za jednotku nebo celková cena položky objednávky. V tomto případě by se cena objednávky mohla dopočítávat přes uloženou proceduru. Teoreticky by však mohlo dojít ke ztrátě ceny objednávky, nesmělo by tedy docházet ke smazání těchto položek.

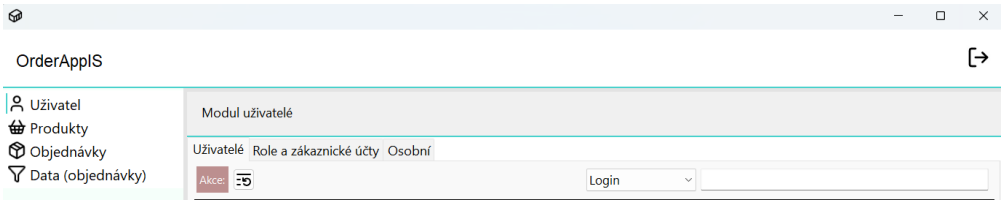
3.4 Komponenty a moduly systému

Aplikace bude rozdělena do samostatných celků (modulů). Má smysl aplikaci dělit na modul objednávek, uživatelů a produktů. Stanovme, co daná část aplikace musí obsahovat a umožňovat uživateli. Nezbytností je správa uživatelů. Zde bude možné měnit osobní údaje, jako je telefonní číslo či heslo k přihlášení. Pouze administrátor může spravovat účty. Jedná se o část aplikace, která se musí zobrazit každému uživateli, a zároveň je umožněno zobrazení informací o aktuálně přihlášeném uživateli. Pro správu produktů a jejich kategorií se také nabízí možnost vytvořit vlastní funkční celek. Součástí bude přehled sortimentu produktů včetně cen a kategorií. Zobrazení stavu množství produktu na skladě je dalším požadavkem. Produkty by mělo být možné hromadně importovat. Posledním modulem je modul Data (objednávky), slouží k zobrazení rozšiřujících informací k objednávkám.

3.5 Stručný popis uživatelského rozhraní

Okno aplikace bude rozdělené do dvou hlavních částí, hlavičky a těla. Hlavička bude obsahovat název aplikace a tlačítko pro ukončení. Jednotlivé moduly aplikace budou dostupné přes hlavní menu. To bude součástí těla v levém bočním panelu, pomocí této grafické komponenty bude umožněno přepínat mezi stránkami aplikace (někdy odkazovány jako layouty). Dále je nutné zobrazit, v jakém modulu či submodulu se aktuálně uživatel nachází, tím, že se vypíše textový popis do svrchní části těla aplikace, zbytek je pak určen pro zobrazení reprezentace programového obsahu aplikace. Uživatelské interakce bude každá stránka zachytávat a vyhodnocovat. Data načtená z databáze budou primárně zobrazována pomocí tabulek, které budou dále umožňovat řazení dle sloupců. Mezi často využívané komponenty dále patří *TextBox*, *Label*, *ComboBox*, *DataGridView* či vlastní *UserControl*.

Barvy aplikace jsou zvoleny tak, aby s aplikací šlo pracovat i delší čas a nedocházelo k přehlédnutí důležitých informací. Základem jsou systémové barvy *Window* a *ButtonFace*, pro zvýraznění je využita *MediumTurquoise* nebo *LightBlue*. Lze je změnit v nastaveních. Pro lepší uživatelský zážitek je využito ikon Lucide (Lucide Contributors, 2025). Lze říci, že aplikace vychází z rozumně pestré palety barev. Účelem je poskytnutí intuitivního uživatelského rozhraní.



Obrázek 39: Návrh vzhledu aplikace

Zdroj: vlastní tvorba

3.6 Zabezpečení aplikace

Aplikace má obsahovat správu uživatelských účtů. Pro zachování bezpečnosti se hesla šifrují (hashují) před uložením do databáze. Moduly mají různá práva na základě uživatelské role. V aplikaci bude implementován RBAC. Práva uživatelů vycházejí především z *Use Case* diagramu. Jedná se o jednodušší formu správy práv uživatele, která je dále rozšiřitelná na práva pro konkrétního uživatele. Pro hashování hesel je využito hojně podporované knihovny *Bcrypt.Net-Next* (BcryptNet Contributors, 2026). V ukázce je nastoveno jednotné heslo, což je pouze z testovacích důvodů a pro produkční chod aplikace by měl mít každý uživatel své vlastní.

3.7 Návrh testování a nasazení

Aplikace umožňuje testování dat. Testování bude provedeno vůči uživatelskému rozhraní a případně pomocí jednotkových testů. Nasazení bude probíhat prvotním spuštěním a nastavením databáze PostgreSQL. Aplikace musí mít přístup k databázi. Ke spustitelnému souboru bude následně vytvořen zástupce na ploše. Vydávání nových verzí bude podmíněno manuální změnou složky programu.

3.8 Zhodnocení

Hlavním zdrojem podkladů pro analýzu byly zmíněné požadavky a již fungující systémy. Na jejich základě byly uvedeny možnosti architektury a ukládání dat. K tvorbě diagramů byl využit specializovaný software Draw.io (JGraph Ltd., 2025) a služba Lucidchart (Lucid Software Inc., 2025), tímto se na ně autor odkazuje. Na základě diagramů a požadavků na systém bylo možné sestavit ER diagram určující strukturu databáze. Aplikace byla navržena tak, aby se mohla stát součástí již běžících systémů, se kterými by sdílela databázi. Kromě toho obsahuje vlastní správu uživatelů a jejich účtů, což z ní dělá samostatně spustitelný program. V současné době existuje mnoho řešení podobných systémů, jedná se spíše o webové systémy. Tento program se však staví k problematice jinak a řeší tak své využití. Výše uvedená analýza pouze rozvíjí základní myšlenky, které se týkají samotného systému. Je na místě počítat s rozšiřováním programu, což může být předmětem dalšího vývoje.

4 Tvorba aplikace

Následuje implementační část práce. Nabyté zkušenosti s programovacím jazykem C# spjaté s prací s daty bude nyní možné využít při tvorbě aplikace. Účelem je poskytnutí rozhraní k databázi objednávkového systému. Aplikace je pojmenována OrderAppIS, což lze interpretovat jako informační systém pro správu objednávek.

4.1 Programování aplikace

Po prvotním návrhu bylo třeba vyřešit, jak bude aplikace rozvržena z pohledu zdrojového kódu. Po zvážení více možností byla vybrána technologie Windows Forms, která zajistí rozumné možnosti vývoje a stabilitu aplikace, více informací o výběru a případné modernizaci je v analýze. Projekt se skládá celkem ze čtyř podprojektů *Orderappis*, *Orderappis.Data*, *Orderappis.Services* a *Orderappis.Test*, které jsou vloženy do společného řešení *OrderappisProj*. Tyto jednotlivé celky odpovídají stanoveným nárokům na aplikaci a její rozšiřitelnost, což může být i použití Entity Frameworku a případné napojení na API. Třídy reprezentující tabulky a poskytující přípojení k databázi jsou uloženy v projektu Data. Třída *DbConnProvider* slouží k distribuci připojení k databázi v rámci aplikace. V projektu *Services* jsou naopak uloženy servisní třídy, patří mezi ně i *AuthService*, jejímž účelem je správa přihlášeného uživatele. Při přihlašování se volá metoda *HandleLoginAction*.

```
public bool HandleLoginAction(string username, string password)
{
    string sql = @"SELECT user_id, user_role_id, login, ""password"",
        firstname, lastname, email,
        company_name, phone_num1, phone_num2
    FROM amain.""user""
    WHERE login = @login";

    using var conn = DbConnProvider.Instance.Conn;
    conn.Open();
    using var cmd = new NpgsqlCommand(sql, conn);
    cmd.Parameters.AddWithValue("@login", NpgsqlTypes.NpgsqlDbType.Varchar, username);

    using var reader = cmd.ExecuteReader();

    if (reader.Read())
    {
        string dbPassword = reader.GetString(3);
        if (VerifyPwd(password, dbPassword))
        {
            CurrentUser = new User(
                reader.GetInt32(0),
                reader.GetInt32(1),
                reader.GetString(2),
                dbPassword,
                reader.GetString(4),
                reader.GetString(5),
                reader.GetString(6),
                reader.GetString(7),
                reader.GetString(8),
                reader.IsDBNull(9) ? null : reader.GetString(9)
            );

            return true;
        }
    }

    return false;
}
```

Obrázek 40: Tělo metody HandleLoginAction

Zdroj: vlastní tvorba

Hlavním projektem řešení je *Orderappis*, obsahuje obecné grafické komponenty, znovupoužitelné komponenty (*UserControl*) a obslužné třídy. Program je složen z modulů objednávek, produktů a uživatelů. Každý modul či submodul je reprezentován pomocí vlastní komponenty *UserControl*, což umožňuje jeho použití i na jiných místech aplikace. V metodě *FormMain_Load* hlavního formuláře *FormMain* je volání formuláře pro přihlášení *AuthForm* a metody inicializace komponent metodou *InitMainForm*. Metoda je zavolána při události *Load* hlavního okna, která je určena pro tyto akce.

```
private void InitMainForm()
{
    InitLeftPanelMenu();
    string imgPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Images", "log-out.jpg");

    if (File.Exists(imgPath))
    {
        pictureBoxLogOut.Image = Image.FromFile(imgPath);
    }
    this.Icon = new Icon(@"Images\container.ico"); // nastavení hlavní ikony
}

private void FormMain_Load(object sender, EventArgs e)
{
    // Auth Form
    var authForm = new AuthForm();
    authForm.ShowDialog();

    // PRO PRODUKCI:
    if (!authForm.IsLoggedIn) // přihlášení selhalo
    {
        Close();
    }

    InitMainForm();
}
```

Obrázek 41: Metody InitMainForm a FormMain_Load hlavního formuláře

Zdroj: vlastní tvorba

V konstruktoru se vytváří instance třídy *MainLayoutSwitch* z jmenného prostoru *Dev*, která má za účel zobrazovat jednotlivé moduly na základě zvoleného prvku menu v levém bočním panelu obslužnou metodou *ChangeLayout*. Tímto způsobem funguje menu v projektu *Orderappis*.

```
public void ChangeLayout(ToolStripMenuItem mItemActivated, FormMain form)
{
    bool isMainLevel = false;
    UserControl? userControl = null;
    string PageLabelStr = "";

    switch (mItemActivated.Tag)
    {
        case "USERS":
            userControl = new MUsersMain();
            PageLabelStr = "Modul uživatelé";
            isMainLevel = true;
            break;
        case "ORDERS":
            userControl = new MOrdersMain();
            PageLabelStr = "Modul objednávky";
            isMainLevel = true;
            break;
```

Obrázek 42: Ukázka části metody ChangeLayout

Zdroj: vlastní tvorba

Moduly a submoduly mohou být členěny na jednotlivé stránky pomocí *TabControl*. Pro snazší rozvržení vlastního zobrazení modulů či submodulů je využito *TableLayoutPanelu* a *FlowLayoutPanelu*. *TableLayoutPanel* rozkládá nadřízený prvek na řádky a sloupce, ty jsou reprezentovány kolekcemi *Rows* a *Columns*. Každému řádku či sloupci lze přiřadit absolutní, procentuální či automatickou velikost.

Propojení se zdrojem dat je realizováno na základě práce s třídou *NpgsqlDataAdapter* a *DataGridView* je navázán vlastností *DataSource* na *DataTable*. Sloupečky *DataGridView* jsou definovány ručně, každý z nich má obvykle specifikovanou vlastnost *DataPropertyName*, *HeaderText* a *Name*. Dotazy SQL jsou specifikovány pro každou akci instance adaptéru zvlášť, aplikace obsahuje především CRUD operace. Jednou z hlavních funkcionalit tabulek (*DataGridView*) je stránkování, aplikace aktuálně nabízí možnosti zobrazení po třech, pěti, deseti, dvaceti, padesáti a sto řádcích. Výchozí volbou je deset záznamů na stránku. Stránkování se řídí vypočtenými vlastnostmi *currentPage*, *pageSize* a *totalPages*. Případné vyhledávání v gridech je řešeno pomocí vlastnosti *RowFilter*. Jednotlivé moduly jsou rozděleny na záložky, mezi nimiž se lze přepínat. Při načtení záložky dojde k načtení dat a aplikování stránkování na příslušný grid.

```
private void tabControlUsers_SelectedIndexChanged(object sender, EventArgs e)
{
    // reakce na změnu záložky
    int selectedTabIndex = tabControlUsers.SelectedIndex;
    if (selectedTabIndex == 0) // Tab1
    {
        Init();
    }
    if (selectedTabIndex == 1) // Tab2
    {
        InitTab2();
    }
    if (selectedTabIndex == 2) // Tab3
    {
        InitTab3();
    }
    if (selectedTabIndex == 3) // Tab4
    {
        PageCurrentUser();
    }
}
```

Obrázek 43: Ukázka přepínání záložek

Zdroj: vlastní tvorba

Většina komponent *UserControl*, které reprezentují daný modul či submodul, funguje ekvivalentně. Mezi výjimky patří Položky objednávky (*OrderProduct*), kde je využito dvou tabulek, přičemž do spodní se data načítají na základě aktuálního řádku v horní tabulce. Při změně aktuálního řádku horního (nadřízeného) gridu musí dojít i ke změně podřízeného gridu, to je zajištěno využitím eventu *SelectionChanged*. Metoda volaná v tomto eventu musí obsahovat identifikátor objednávky jako parametr. Ukázka je uvedena níže. Lze si všimnout, že byl zvolen přístup s adaptérem, obdobně je to řešeno napříč aplikací, tj. pomocí readeru či adaptéru se načtou data do tabulky *DataTable*, ta se následně nastaví jako *DataSource* pro komponenty, jako je především *DataGridView*.

```
private void LoadAllOrderProductsTab2Down(int orderId)
{
    string sql = pagHandler.OrdersQueryStoreMemory.GetByName(
        nameof(LoadAllOrderProductsTab2Down)
        );

    using var conn = DbConnProvider.Instance.GetConn();
    conn.Open();
    using var cmdDown = new NpgsqlCommand(sql, conn);
    cmdDown.Parameters.AddWithValue("@orderId", orderId);
    orderProductsAdapter.SelectCommand = cmdDown;

    dtOrderProducts.Clear();
    orderProductsAdapter.Fill(dtOrderProducts);

    // nastavení datasource
    dataGridViewDown.DataSource = dtOrderProducts;
}
```

Obrázek 44: Ukázka metody pro načtení položek objednávky

Zdroj: vlastní tvorba

Pro zajištění správně fungující autorizace bylo nutné získávat odkaz na aktuálně přihlášeného uživatele, který je uložen v paměti aplikace po přihlášení. Jednou z možností bylo ukládat pouze uživatelský identifikátor či e-mail a na základě těchto informací pokaždé načíst data z databáze, což je výhoda, pokud se například změní uživatelovy údaje během chodu aplikace, ať už ruční změnou či úpravou přímo v aplikaci. Nakonec bylo však využito načítání celého objektu z paměti aplikace bez případných změn jednou načtených dat po přihlášení. Obecně by to nemělo vadit, protože se stejně na uživatele odkazuje většinou prostřednictvím identifikátoru a mazat ho může pouze administrátor. Ten musí vědět, co přesně dělá, jinak může dostat aplikaci do méně konzistentního stavu, proto jsou v databázi tabulky s historií.

Metoda *InUserRole* přijímá jako parametr název role, která je uložena v databázi, a vyhodnocuje, zda uživatel spadá do role či ne. Na základě toho lze určit, komu se mají zobrazit daná tlačítka a komu ne. Obecně platí, že nejvíce práv má administrátor a poté i manažer. Každý uživatel může být pouze v jedné roli zároveň, nicméně administrátor může vytvořit i další uživatele pro jednu osobu s příslušnými právy. Metoda *GetCurrentUserRole* vrací roli aktuálně přihlášeného uživatele. Obě jsou součástí *AuthService*, což je třída, která náleží projektu Services. Dalším projektem společného řešení je projekt Data. Zde jsou uloženy třídy, které reprezentují tabulky v databázi, a jsou zde i jejich repozitáře, tj. třídy implementující návrhový vzor repozitáře.

Program obsahuje generování PDF faktury s využitím knihovny *PdfSharp* (empira Software GmbH, 2025a) a *HtmlRenderer* (ArthurHub, 2025). Tímto se na ně autor odkazuje, informace byly čerpány z příslušných dokumentací uvedených ve zdrojích. Třída vytvářející PDF je *PDFInvoice*.

PDFInvoiceHelper slouží k vytvoření exportu faktury. S využitím metody *buttonInvoice_Click* se otevře dialogové okno, ve kterém lze vytvořit fakturu. *PDFInvoice* má metodu *Create* s parametrem název souboru, objektem dat faktury, seznamem položek objednávky identifikátorem objednávky a příznakem dph. Pro zjednodušení byla využita definovaná šablona, která se načte jako řetězec pomocí *DefHTMLTemplate* a dále se řetězcovou funkcí *Replace* nahradí určené sekce daty z předaných parametrů. Třída *PdfGenerator* umožní vytvořit PDF dokument.

```
string orderProductsStr = "";
if (orderProducts != null)
{
    foreach (PDFInvoiceOrderProduct item in orderProducts)
    {
        string data = @$"
        <tr>
            <td>{item.ProductCode}</td>
            <td>{item.ProductName}</td>
            <td style=""text-align:right;"">{item.ProductQty}</td>
        </tr>";
        orderProductsStr += data + Environment.NewLine;
    }
}
html = html.Replace("[POLOZKY]", orderProductsStr);

PdfDocument pdf = PdfGenerator.GeneratePdf(
    html,
    PdfSharpCore.PageSize.A4
);

pdf.Save(filename);
```

Obrázek 45: Ukázka části metody Create

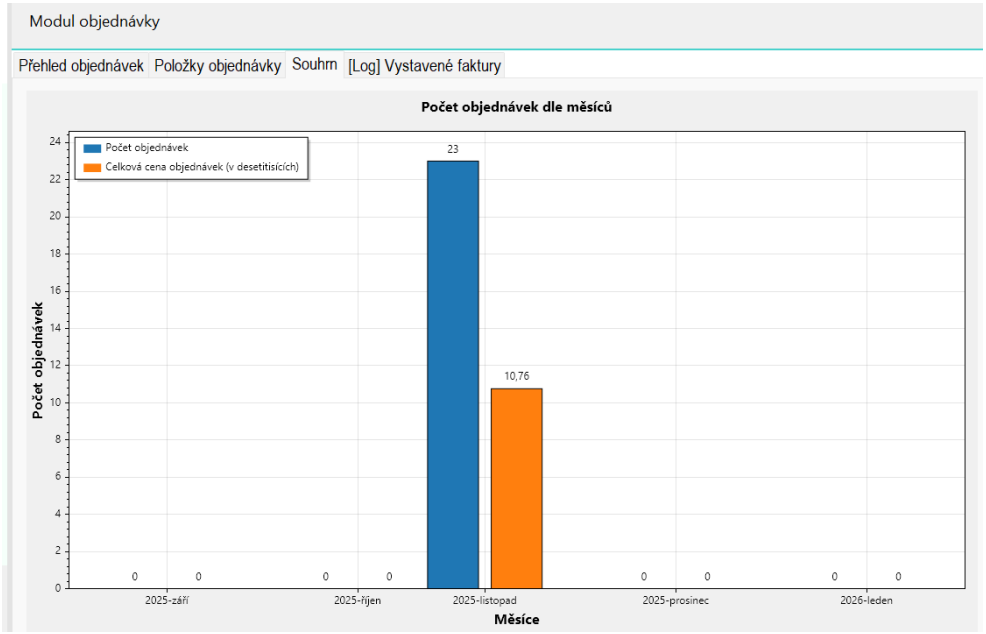
Zdroj: vlastní tvorba

4.2 Aplikace z pohledu práce s daty

Obsahem aplikace jsou kromě výše zmíněných exportů a funkcionalit i souhrny. Jedním z nich je souhrn objednávek dle měsíců. Graf znázorňuje počet objednávek a jejich celkovou cenu v desetitisících za posledních pět měsíců. Počet objednávek je modře a celková cena oranžově. Druhý graf se nachází v modulu produktů a znázorňuje počet objednávek dle zvolené kategorie. K jejich tvorbě autor použil knihovnu ScottPlot (Harden, 2025). Zároveň jsou vybrané akce logovány do souboru v adresáři aplikace (*mainLog.txt*), což se týká modulu objednávek. Pro manažery a administrátory je dostupné okno, kde lze listovat v historii změn objednávek, tímto způsobem lze vystopovat případné neshody v systému.

Z pohledu aplikačního kódu aplikace přistupuje k databázi pomocí vlastních SQL dotazů, které jsou poměrně časté. V některých případech byl použit vzor ViewModel pro usnadnění obsluhy formulářů. Dále jsou zavedeny vlastní kontroly (například při vytváření objednávky). Veškeré chybové hlášky jsou znázorněny uživateli v sekci Log, která je součástí většiny dialogových oken, nebo se používá *MessageBox*.

Většina oken je složena z části s tlačítky, které umožňují operace, jako jsou úpravy dat, z vlastního gridu s popsanými sloupci a ze stránkování, kde lze měnit počet záznamů na stránce a jednotlivé strany procházet. Práce s daty je v aplikaci všudypřítomná, ať už se jedná o export, import, generování PDF, přepočet cen, či jen samotnou komunikaci s databází.



Obrázek 46: Graf v modulu objednávek

Zdroj: vlastní tvorba

4.3 Přehled obsahu aplikace

Vzhledem k tomu, že se jedná o desktopovou aplikaci, sluší se uvést popis základních oken aplikace tak, jak byla naprogramována. Modul uživatelů je poměrně přímočarý. Po přihlášení administrátora (uživatel a00001, heslo hsl123) je umožněno spravovat uživatele. Pokud uživatel není administrátor, moc práv zde nemá, pouze refresh dat. Administrátor spravuje zákaznické účty, které se využívají k tvorbě objednávek. Mezi základní operace administrátora patří správa zákaznických účtů a uživatelů. Uživatelům může například změnit heslo nebo telefonní číslo.

Druhý je model produktů, zde manažer či vedoucí může nahrát nové produkty do databáze nebo exportovat ty stávající. Součástí je i zobrazení obrázku produktu. Záložka kategorie zobrazuje přehled kategorií používaných v aplikaci. Metoda načítání dat gridu hlavního okna pro modul objednávek je v příloze A.1. Jednotlivá důležitá okna jsou znázorněna v příloze A.2, A.3, A.4 a příloze A.5.

4.4 Zprovoznění databáze

Jako primární uložiště dat spojených s objednávkami byl vybrán PostgreSQL, určený na základě analýzy systému. Po instalaci databázového systému je nutné vytvořit příslušnou databázi a uživatele, který ji bude spravovat. Pro připojení k databázi lze využít specializovaná softwarová řešení, jako je volně dostupný DBeaver.

Pro připojení k databázi je nutné zadat adresu databázového serveru a port, na kterém databáze běží. Následně lze po připojení vytvořit databázi orderappis a uživatele, který ji bude spravovat. Vytvoření databáze je možné pomocí příkazu *CREATE DATABASE* a názvu databáze. Vytvoření uživatele lze provést pomocí *CREATE USER* a nastavení jeho práv pro práci s databází s využitím *GRANT*. Příkazy jsou v souboru *Db.sql*.

```
-- SPUSTIT PRIKAZEM: psql -U postgres -d orderappis -f Dump_h.sql
--
-- PostgreSQL database dump
--
-- Dumped from database version 14.9
-- Dumped by pg_dump version 14.9

SET statement_timeout = 0;
SET lock_timeout = 0;
```

Obrázek 47: Část obsahu souboru Dump_h.sql

Zdroj: vlastní zpracování

Nyní je vytvořena databáze, následuje schéma *amain*, do kterého se budou ukládat všechna aplikační data. Schéma není nic jiného než logický celek v databázi PostgreSQL. Může obsahovat tabulky, uložené procedury, sekvence, indexy a další. I zde lze přidělit práva ke schématu pomocí *GRANT* (příslušné skripty jsou součástí přiloženého souboru *Dump_h.sql*).

Pro vytvoření struktury tabulek slouží Data Definition Language. S využitím *CREATE* a *DROP* vytvoříme tabulky a případné procedury a další součásti schématu amain. Dalším pomocníkem může být příkaz *COMMENT* sloužící k vytvoření komentářů. Často se využívá i ke komentování jednotlivých sloupců tabulky, pokud například máme sloupec, který svou hodnotou reprezentuje určitou volbu (status objednávky). Pro vložení dat do databáze je určen Data Manipulation Language, přičemž pro vložení dat využijeme *INSERT*.

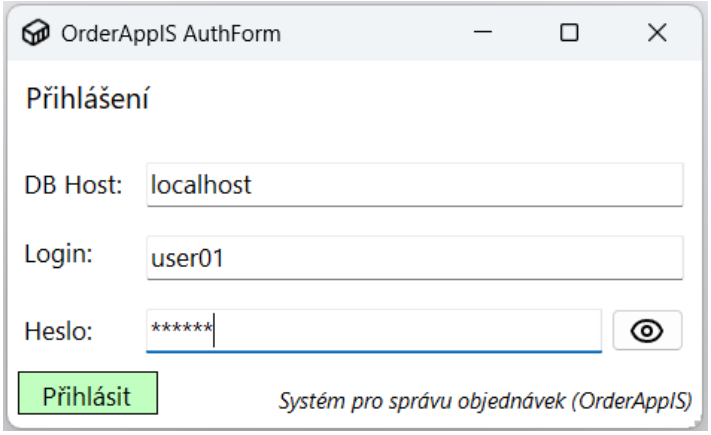
Pro zprovoznění stačí vytvořit uživatele *orderappis_client* a databázi *orderappis*, spustit příkaz ze svrchní části souboru Dump_h.sql (příkaz *psql*). Tímto krokem získáme aktuální databázi i s naplněnými daty a základně nastavenými právy. Podrobnější popis je v *README.md* (příloha 2).

4.5 Instalace

Kompletní popis zprovoznění je v soboru *README.md*. Celý program OrderAppIS se skládá ze složky, která obsahuje příslušné komponenty, jako jsou například dynamické knihovny, určené pro spuštění a správný chod programu. Manipulace s obsahem hlavní složky tedy není doporučena, protože by mohlo dojít k nestandardnímu chování programu. Aby se předešlo těmto chybám, lze na ploše vytvořit zástupce, který bude odkazovat na spustitelný soubor ve složce programu. Nejedná se tedy o nijak komplikovanou instalaci. Případné úpravy samotného programu lze aplikovat změnou aktuálního adresáře programu, tj. přehráním jeho obsahu novou verzí. Aplikace byla testována na systému Windows 11.

4.6 Ovládání

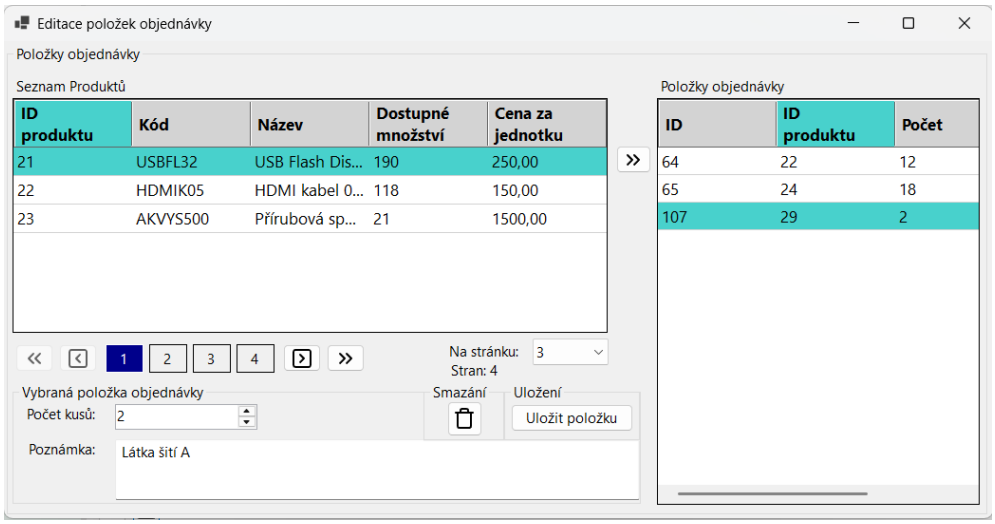
Nyní se přejde k popisu ovládání. V návrhu a analýze již bylo základně popsáno a navrženo rozhraní aplikace, proto lze rovnou přejít k ovládání. Při prvotním spuštění se zobrazí dialogové okno, které vyzývá uživatele k zadání přihlašovacích údajů, loginu a hesla. Bez zadání správné kombinace neproběhne přihlášení a uživatel se nedostane do hlavního okna aplikace. Údaj DB Host určuje IPv4 adresu databázového serveru. Za předpokladu, že dojde k úspěšnému přihlášení, dostaneme se do okna znázorněného na obrázku. Pomocí panelu na levé straně lze přepínat mezi moduly systému. Každý modul se skládá z jedné a více záložek přepínatelných v hlavním okně.



Obrázek 48: Přihlašovací okno

Zdroj: vlastní tvorba

V hlavičce programu je na pravé straně tlačítko odhlášení. První tlačítko nad gridem symbolizuje obnovu dat, přičemž po kliknutí dojde k přenačtení celého gridu. Následuje přidání, které otevře dialog. Takto fungují i tlačítka editace. Poslední tlačítko s ikonou koše je určeno ke smazání záznamu. Operace editace a smazání se vztahují ke konkrétnímu řádku, který je podbarven zeleně, stejně tak jako aktuální sloupec (pouze hlavička). Tato doprovodná okna obvykle obsahují i kontrolu validity formátu dat vkládaných uživatelem. Případné chyby se zpravidla zobrazují pomocí červeného textu, čímž i graficky upozorňují uživatele. Doprovodná okna i tlačítka, která je spouští, fungují ekvivalentně v rámci celé aplikace. V příkladu je uveden méně běžný přístup, který je součástí záložky Položky objednávky. Pomocí tlačítek a formuláře zde lze kontrolovat, které položky obsahuje daná objednávka. Kromě počtu kusů lze přidat i poznámku ke konkrétní položce. Cena položky vzhledem k množství se v odevzdané verzi automaticky přičítá k celkové ceně objednávky a naopak při odebrání prvků se od celkové ceny odečte. Funkcionalita byla otestována v běžných případech, pokud by nastala chyba, případně pokud by se měla změnit celková cena objednávky, lze ji nastavit v editovacím okně u příslušného gridu. Automatické dopočítávání ceny je i u dopravy k objednávce.

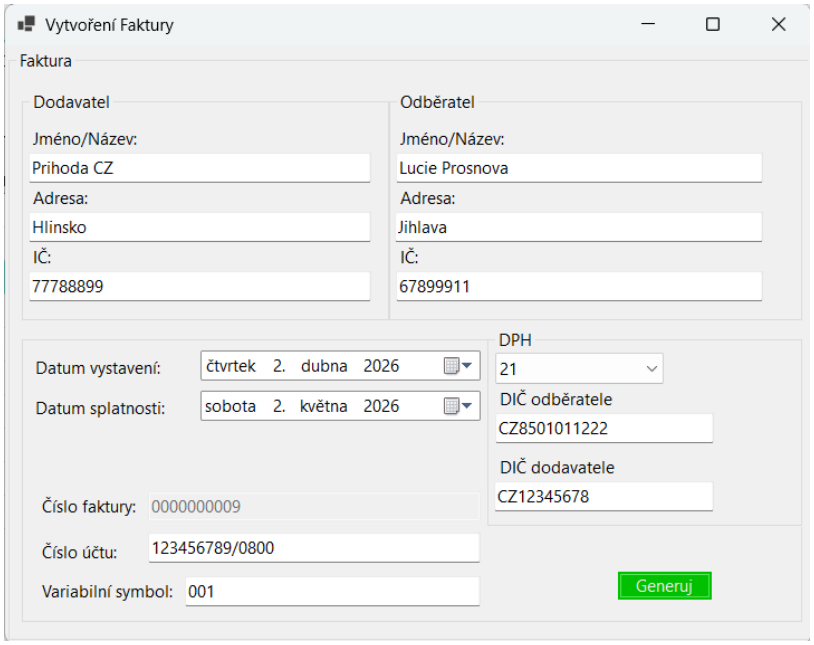


Obrázek 49: Příklad obslužného okna s použitím DataGridView

Zdroj: vlastní tvorba

Součástí programu jsou i další funkce, jako je import a export produktů, určené pomocí práv pro administrátory a manažery. Obslužná tlačítka se zpravidla nacházejí v pravé dolní části aktuálního okna aplikace. Po kliknutí se objeví dialogové okno, ve kterém bude možné provést akci či vybrat zdrojový soubor apod.

Ještě se sluší vysvětlit chování oken, ze kterých nemusí být na první pohled jasné, jak fungují. Dialogové okno Vytvoření faktury vyvolané tlačítkem Faktura z hlavního okna (při aktivním modulu objednávek) obsahuje formulář, ze kterého se následně doplní informace, jako je například jméno odběratele či firma dodavatele. Po stisknutí zeleného tlačítka Generuj dojde k vytvoření faktury, která se uloží do zvolené lokace a zaloguje se.

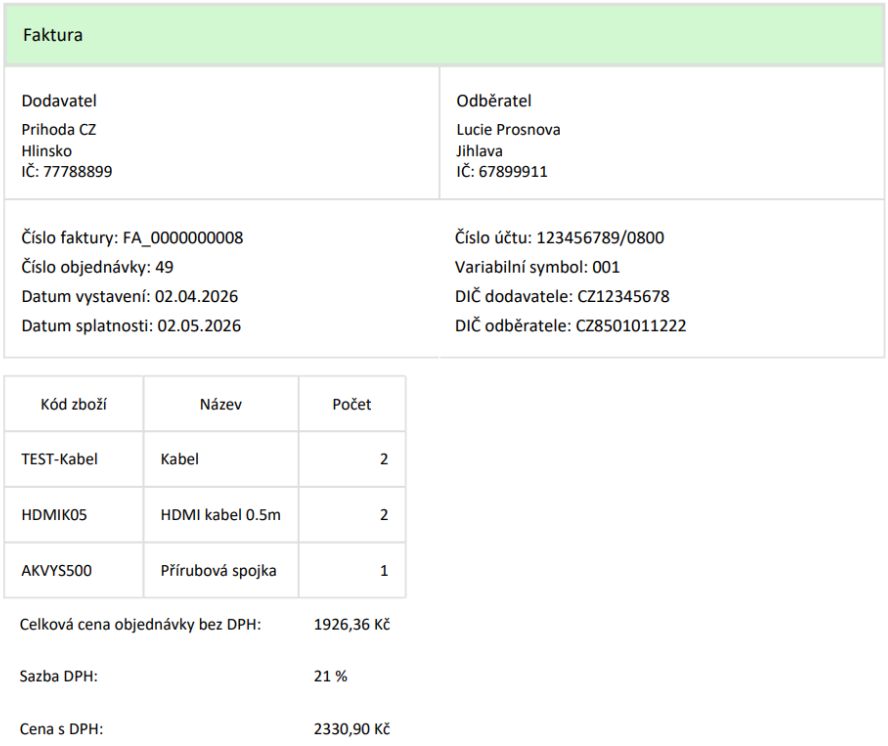


Obrázek 50: Vytvoření faktury

Zdroj: vlastní tvorba

Výsledná faktura se opět uloží ve formátu PDF a její vygenerování se zaloguje, takže poté lze vidět nový řádek s časovou známkou v modulu Objednávky na poslední záložce, což slouží

především jako kontrola. V současné době je aplikace rozšířena o možnost počítaní ceny s DPH. Dodatečná pole pro zadání DPH jsou implementována, a umožňují mimo jiné výběr sazby DPH. Faktura je vytvořena z HTML šablony a je rozdělena do několika hlavních sekcí (viz obrázek). Postup vytvoření uvedeného PDF výstupu byl již uveden v předchozí části práce. Veškeré zdrojové kódy lze nalézt v ukázkách v adresáři ObjednavkovySystem v příloze 2.



Faktura

Dodavatel	Odběratel
Prihoda CZ Hlinsko IČ: 77788899	Lucie Prosnova Jihlava IČ: 67899911

Číslo faktury: FA_0000000008	Číslo účtu: 123456789/0800
Číslo objednávky: 49	Variabilní symbol: 001
Datum vystavení: 02.04.2026	DIČ dodavatele: CZ12345678
Datum splatnosti: 02.05.2026	DIČ odběratele: CZ8501011222

Kód zboží	Název	Počet
TEST-Kabel	Kabel	2
HDMIK05	HDMI kabel 0.5m	2
AKVYS500	Přírubová spojka	1

Celková cena objednávky bez DPH:	1926,36 Kč
Sazba DPH:	21 %
Cena s DPH:	2330,90 Kč

Obrázek 51: Faktura vygenerovaná formulářem
Zdroj: vlastní tvorba

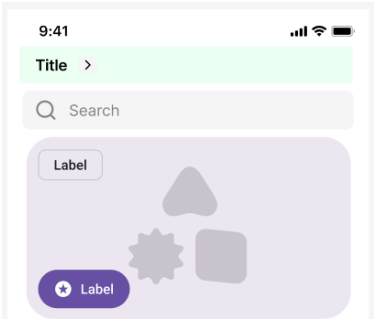
4.7 Testování aplikace

Aplikace byla interně testována ve společnosti Příhoda s. r. o., kde bylo na serveru v testovací databázi vytvořeno příslušné schéma dle uvedeného postupu v README.md. Aplikace byla spuštěna z počítače ve vývojovém oddělení, později byla testována s lokální databází. Aplikace se dokázala přihlásit k databázi a projít ověřením uživatele. Došlo ke správnému načtení dat z databáze a byla vyzkoušena základní funkcionalita aplikace. Bylo otestováno logování historických dat při změnách v objednávkách, přidávání položek do objednávky, generování PDF a zobrazení grafů. Zároveň byl proveden import zkušebních produktů. Byla vyzkoušena změna hesla uživatele. Výše uvedené akce se podařilo vyzkoušet úspěšně. Programování aplikace je proces, proto i testování musí být kontinuální. Testování provedl autor práce, zároveň své akce okomentoval a vysvětlil funkcionalitu a možnosti aplikace dalším pracovníkům vývojového oddělení. Přínos aplikace je pozitivní. Aplikace funguje i při připojení více zařízení současně k databázi, pro změnu dat v gridech je však nutné využít refresh. Pro zlepšení předvídatelnosti a celkové funkce databáze se vyplatí jí věnovat pozornost. Tímto tématem by se však mohla zabývat samostatná práce. Jednou z možností, jak snížit riziko nekonzistence dat, je využití API, protože má jednotný přístup do databáze.

Na základě výsledků testování lze jednoznačně usoudit, že aplikace hraje svou roli z pohledu objednávkového systému. V dalších verzích by bylo možné ji orientovat více směrem k více reportům a exportům či importům dat do databáze. Generování PDF je dobrá myšlenka, kterou je možné dále uplatňovat a rozvíjet. V případě rozšíření na webovou aplikaci by pak tato aplikace mohla mít především tento účel. Zároveň je i dobré, že je tu možnost spravovat uživatele a produkty.

4.8 Návrh rozšíření

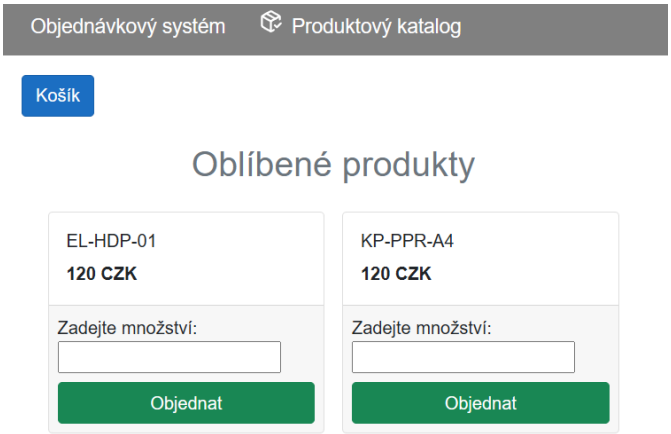
Aktuální aplikace najde uplatnění především v administrativě. Na základě zkušeností získaných během její tvorby se může vytvořit webové rozhraní, které bude zaměřeno na tvorbu objednávek z pohledu zákazníka a bude tak nabízet přívětivější uživatelský zážitek pro odběratele. Takové rozhraní je možné vytvořit v .NET C# s využitím ASP.NET. Kromě webové aplikace by samozřejmě bylo možné vytvořit i mobilní aplikaci, například migrací současné aplikace na MAUI, a vyvíjet rovnou i pro mobilní platformy. Mobilní aplikace ale vyžadují připojení k databázi přes API, takže je tu možnost nejdříve vytvořit API a potom napojit klientské aplikace. K vytvoření návrhu byl použit software Figma (Figma, 2026).



Obrázek 52: Grafické rozvržení svrchní části mobilní aplikace

Zdroj: vlastní tvorba

Prvním vylepšením vůči původnímu systému je zavedení vylepšeného produktového katalogu. To je stěžejní pro další rozvoj, protože by toto rozšíření bylo zaměřeno na zákazníky. Na základě katalogu lze pokračovat v objednávkách, což by však již bylo nad rámec celé práce. Pro návrh byla využita knihovna Bootstrap a dokumentace (Bootstrap documentation, 2025).



Obrázek 53: Grafické rozvržení webové aplikace

Zdroj: vlastní tvorba

4.9 Zhodnocení

Aplikace vznikla jako rozšiřující modul objednávek pro databázovou aplikaci firemního informačního systému (FIS), který vlastní společnost Příhoda s. r. o. Původní FIS je v současné době rozšiřován na webovou aplikaci a slouží především pro účely firmy, jako je například docházka a výroba. Tento program neobsahuje obchodní tajemství společnosti. Účel programu aktuálně spočívá nejen v možnostech práce s objednávkami, ale slouží primárně jako vzor pro práci s daty. Díky dostupným online zdrojům, jako je dokumentace jazyka C#, probíhala tvorba aplikace relativně dobře. Moderní vývojové prostředí Visual Studio nabízí také mnoho možností, a to především nabídku oken ze záložky Zobrazit, která usnadňují přehled o stavu projektu, využitých prvcích, instalovaných balíčcích apod.

Primární výhodou současného přístupu je relativně snadné prototypování a tvorba oken aplikace, což dále podporuje samotný vývoj. Pro vygenerování souboru Dump_h.sql bylo použito návodu k programu DBeaver (DBeaver.com, 2025). Zdrojový kód generování PDF je postaven na knihovně PdfSharp (empira Software GmbH, 2025a) včetně příkladů z jejich oficiálního účtu na GitHubu (empira Software GmbH, 2025b) a HTML Renderer (ArthurHub, 2025). K vytvoření sloupcového grafu v aplikaci byla použita knihovna ScottPlot (Harden, 2025). Jako databáze byla použita PostgreSQL (PostgreSQL Global Development Group, 2025). Pro inspiraci při tvorbě triggeru pro historická data byla využita webová stránka, která tento problém popisuje (Codegenes.net, 2025). Ikony jsou z již zmíněného zdroje od společnosti Lucide (Lucide Contributors, 2025). Kompletní ukázku zdrojového kódu objednávkového systému lze nalézt v příloze 2.

5 Další kroky

Přibližná představa o tom, jak by se mohlo dále pokračovat, byla již uvedena. Je vhodné věnovat se především WPF, protože na základě zjištěných poznatků práce toho nabízí více. Co se týče programování aplikací, jsou zde další možnosti rozšíření nejen v rámci možností platformy. Pro zvýšení robustnosti a kvality řešení se vyplatí mít naprogramované API. Zálohování databáze aplikací na cloud by bylo také vhodné. Aplikace by měla v dalších verzích přistupovat k databázi právě přes API. V případě zavedení modulu do provozu by tomu tak bylo. Aktuální projekt objednávkového systému mimo jiné řeší otestování možností databáze. Aplikace se zároveň může mírně upravit a v těchto verzích sloužit jako podpůrná aplikace, např. pro webové aplikace.

Z pohledu práce s daty je v dané technologii široká škála možností. Zajímavé by bylo širší využití předností technologie LINQ, která se výborně hodí na zpracování kolekcí dat (datasetů), což by mohlo být probráno v dalších verzích práce. Tato verze se však zaměřila pouze na konkrétní tematiku a pro ostatní nezbylo tolik prostoru. Dále by mohly být uvedeny i jiné programovací jazyky z platformy .NET. Prozkoumání možností cloudů, jako jsou AWS a Azure, z pohledu tématu práce by také mohlo být zajímavé.

V aktuální práci byl kladen důraz na grafické komponenty pro zobrazení tabulkových dat (v rámci práce označované jako grid). V tomto směru by šlo pokračovat a dále rozvíjet stávající práci. Výhodou technologie .NET a jazyka C# je relativně jednodušší práce s těmito komponentami v porovnání s ostatními programovacími jazyky a platformami. Díky tomu jazyk C# velmi zpopularizoval práci s komponentami pro zobrazení tabulkových dat. Jako moderní alternativa se nabízí možnost využití jazyka JavaScript a jeho dostupných knihoven.

Závěr

Práce se skládá ze tří hlavních částí, tj. ukázek, návrhu a analýzy a tvorby komplexní ukázky. V každé z nich se podařilo přednést výsledek, který by měl být vypovídající o zjištěných faktech a dosažených dílčích cílech. Každá část práce splňuje svůj záměr včetně návaznosti a vzájemné provázanosti tematických celků. Zároveň je zde prostor ke kontinuálnímu rozšiřování práce. Programovací jazyk by měl být chápán také jako prostředek, který umožní efektivní práci s daty (či soubory dat), což je jeden z výstupů práce.

Z části ukázek by mělo být možné sestavit si základní přehled o práci se soubory, jako jsou XML, CSV a JSON. Kromě toho jsou vysvětleny i další užitečné možnosti jazyka C#, jako je například LINQ a další. Část práce, která se zabývá tvorbou ukázek, má hned několik výstupů. Prvním z nich je sestavený stručný popis vizuálních a nevizuálních komponent pro práci s daty v .NET, jako jsou například DataGrid a DataTable. Dalším výstupem je uvedení možností, jak tyto komponenty využít, což opět přináší přínos. Obecně lze říci, že práce vnáší pohled na problematiku vizuálních komponent, které zobrazují tabulková data, což je další přínos. Data mohou být načítána z databáze nebo z jiného zdroje dat. Práce ukazuje, jak propojit vizuální komponentu se zdrojem dat. Výstupem je uvedení současných možností ve WPF a Windows Forms. Výstupem je souhrn i přehled, jak s těmito komponentami pracovat. Byla probrána témata, jako je připojení k webovému API a tvorba čárového kódu, nejen tyto ukázky mají přínos pro praxi. Ukázky k jednotlivým tématům se podařilo realizovat.

Komplexní ukázka ukazuje přístup pomocí desktopového rozhraní. V analytické části jsou vysvětleny možnosti volby technologií a návrhu aplikace. Následně byla dle návrhu naprogramována aplikace propojená s databází. Výstupem je aplikace, ve které lze spravovat objednávky a to, co s nimi souvisí. Je uveden postup, jak takovou aplikaci vytvořit. Zároveň jsou uvedeny také výhody a nevýhody daného řešení, což je jeden z výstupů této části. Ukázka aplikace uvádí možnosti využití komponent, jako jsou DataTable, DataAdapter a DataGridView, při komunikaci s databází. Výstupem je, že se takovéto aplikace vyplatí i dále směrovat k reportům dat z databáze a exportům či importům dat. Aplikaci se podařilo realizovat na základě analýzy, proběhlo i testování. Databáze byla také vytvořena a naplněna daty. Bylo navrženo další směřování aplikace.

Seznam použité literatury

ArthurHub. *High performance HTML Rendering library* [online]. 2025 [cit. 2025-12-07]. Dostupné z: https://github.com/ArthurHub/HTML-Renderer

Bootstrap documentation. *Get bootstrap* [online]. USA: get bootstrap, 2020 [cit. 2025-12-20]. Dostupné z: https://getbootstrap.com/docs/5.0/getting-started/introduction/

CLOSE, Josh. *CsvHelper – Getting Started* [online]. 2025 [cit. 2025-12-07]. Dostupné z: https://joshclose.github.io/CsvHelper/getting-started/

Codegenes.net. *How to Automatically Store Table History in PostgreSQL* [online]. 2025 [cit. 2025-12-20]. Dostupné z: https://www.codegenes.net/blog/how-to-store-table-history-in-postgresql/

DBeaver.com. *How to export data in DBeaver* [online]. 2025 [cit. 2025-12-02]. Dostupné z: https://dbeaver.com/2024/09/19/how-to-export-data-in-dbeaver/

BcryptNet Contributors. *Bcrypt.Net – Bringing updates to the original bcrypt package* [online]. 2026 [cit. 2026-01-26]. Dostupné z: https://github.com/BcryptNet/bcrypt.net

empira Software GmbH. *PDFsharp – A .NET library for processing PDF* [online]. 2025a [cit. 2025-12-07]. Dostupné z: https://docs.pdfsharp.net/PDFsharp/Overview/About.html

empira Software GmbH. *PDFsharp 6.0 and MigraDoc Foundation samples* [online]. 2025b [cit. 2025-12-10]. Dostupné z: https://github.com/empira/pdfsharp.Samples

FIGMA. *Figma: the collaborative interface design tool* [online]. San Francisco: Figma, Inc., ©2026 [cit. 2026-01-26]. Dostupné z: https://www.figma.com/

GRIFFITHS, Ian. *Programming C# 12.0: Build Cloud, Web, and Desktop Applications*. O'Reilly Media, 2024. ISBN 978-1-098-15836-1.

HARDEN, Scott W. ScottPlot. *ScottPlot: grafická knihovna* [online]. 2025 [cit. 2025-12-07]. Dostupné z: https://scottplot.net/

KALA, Manoj. *Create Graph Windows Form – WinForm using .Net 8. C#* [online]. C# Corner, 2025 [cit. 2025-12-31]. Dostupné z: https://www.c-sharpcorner.com/article/create-graph-windows-form-winform-using-net-8/

JGraph Ltd. *drawio-desktop* [online]. [cit. 2025-12-20]. Dostupné z: https://github.com/jgraph/drawio-desktop

Lucide Contributors. *Lucide: knihovna ikon* [online]. 2025 [cit. 2025-12-02]. Dostupné z: https://lucide.dev/

Lucid Software Inc. *Lucidchart: Web-based visual collaboration software* [online]. 2025 [cit. 2025-12-02]. Dostupné z: https://www.lucidchart.com

Microsoft. *DataAdapter Class* [online]. USA: Microsoft Corporation, 2025a [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.data.common.dataadapter?view=net-8.0

Microsoft. *Data Binding Overview* [online]. USA: Microsoft Corporation, 2025b [cit. 2025-12-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/

Microsoft. *DataGridView Class* [online]. USA: Microsoft Corporation, 2025c [cit. 2025-12-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.datagridview?view=windowsdesktop-10.0

Microsoft. *DataRowState Enum* [online]. USA: Microsoft Corporation, 2025d [cit. 2025-12-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.data.datarowstate?view=net-8.0

Microsoft. *DataSet Class* [online]. USA: Microsoft Corporation, 2025e [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.data.dataset?view=net-10.0

Microsoft. *DataTable Class* [online]. USA: Microsoft Corporation, 2025f [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.data.datatable?view=net-8.0

Microsoft. *DataView Class* [online]. USA: Microsoft Corporation, 2025g [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.data.dataview?view=net-8.0

Microsoft. *How to use Utf8JsonReader in System.Text.Json* [online]. USA: Microsoft Corporation, 2025h [cit. 2025-12-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/use-utf8jsonreader

Microsoft. *JsonSerializer Class* [online]. USA: Microsoft Corporation, 2025i [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializer?view=net-8.0

Microsoft. *Retrieve data using a DataReader* [online]. USA: Microsoft Corporation, 2025j [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/retrieving-data-using-a-datareader

Microsoft. *XmlReader Class* [online]. USA: Microsoft Corporation, 2025k [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmlreader?view=net-8.0

Microsoft. *XmlWriter Class* [online]. USA: Microsoft Corporation, 2025l [cit. 2025-12-03]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmlwriter?view=net-8.0

MultiBuy, s.r.o. *Objednávkový systém* [online]. 2025 [cit. 2025-12-10]. Dostupné z: https://www.objednavkovy-system.cz/

NATHAN, Adam. *WPF 4 Unleashed*. Pearson Education, 2010. ISBN 978-0-672-33697-3.

PATRICK, Tim. *Microsoft ADO.NET 4 Step by Step*. Microsoft Press, 2010. ISBN 978-0735638884.

PostgreSQL Global Development Group. *PostgreSQL* [online]. 2025 [cit. 2025-12-07]. Dostupné z: https://www.postgresql.org

JSONPlaceholder. *JSONPlaceholder – Free fake REST API* [online]. 2026 [cit. 2026-01-31]. Dostupné z: https://jsonplaceholder.typicode.com/

Seyfor. ERP systém Byznys. *Byznys* [online]. [cit. 7. 3. 2026]. Dostupné z: https://www.byznys.eu/cs-cz/erp-system-byznys

SMITH, Jon P. *Entity Framework Core in action*. Second edition. Shelter Island: Manning, [2021]. ISBN 9781617298363.

The SQLite Consortium. *SQLite* [online]. 2025 [cit. 2025-12-07]. Dostupné z: https://www.sqlite.org/

Wikipedia contributors. *Code 128* [online]. 2025 [cit. 2025-12-03]. Dostupné z: https://en.wikipedia.org/wiki/Code_128

Přílohy

Příloha 1: Ukázky

Obsahuje složky projektů ComboBox, ComboBoxWPF, DataGrid, DataGridColumns, DataGridHttpClient, DataGridView, DataSet, EntityFramework, LINQ, ListBox, ListBoxWPF, PraceSeSoubory, SimpleBarcode, WinformsChart, WPFHeatMap.

Příloha 2: Objednávkový systém

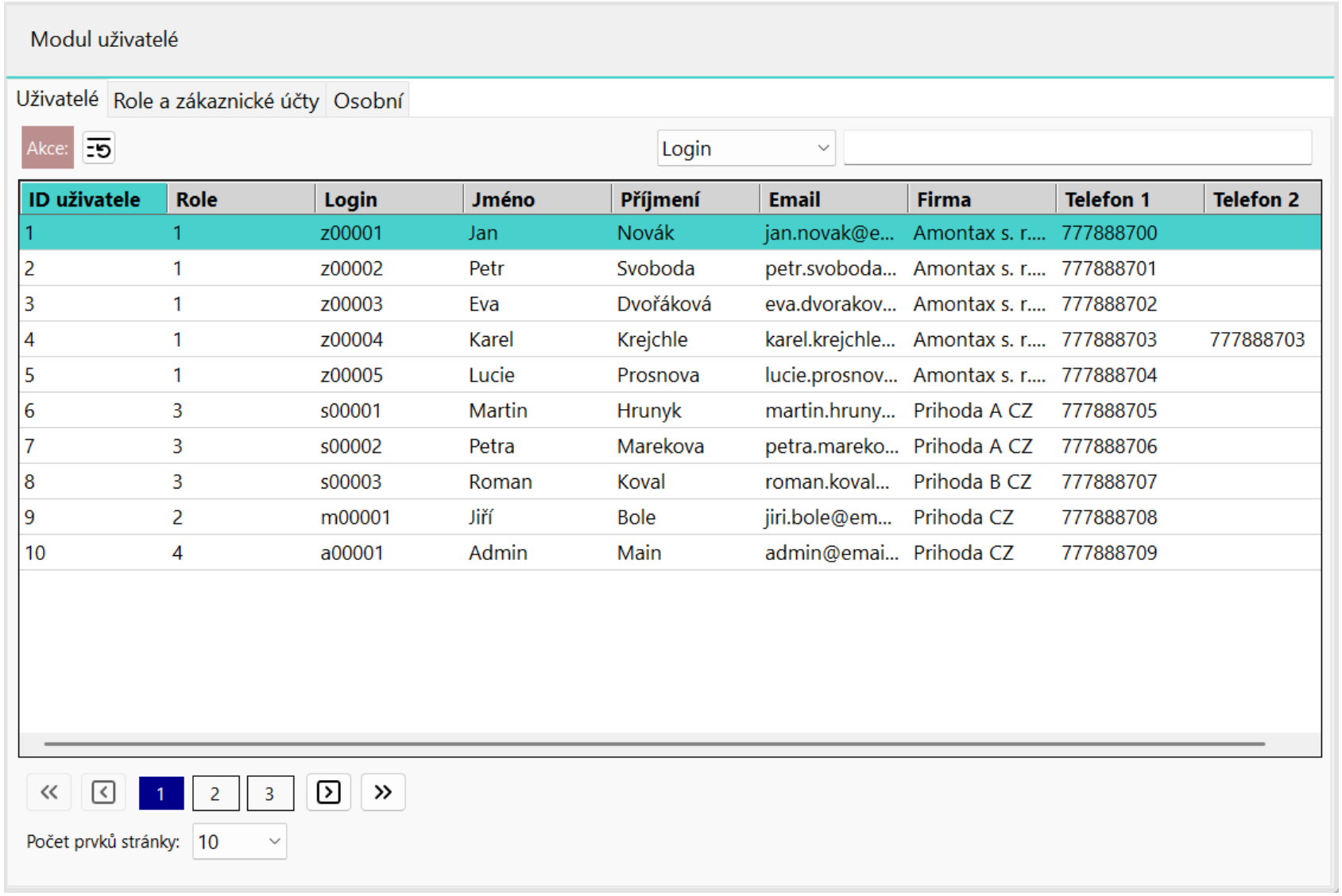
Jedná se o složku ObjednavkovySystem.

Příloha 3: Podklady

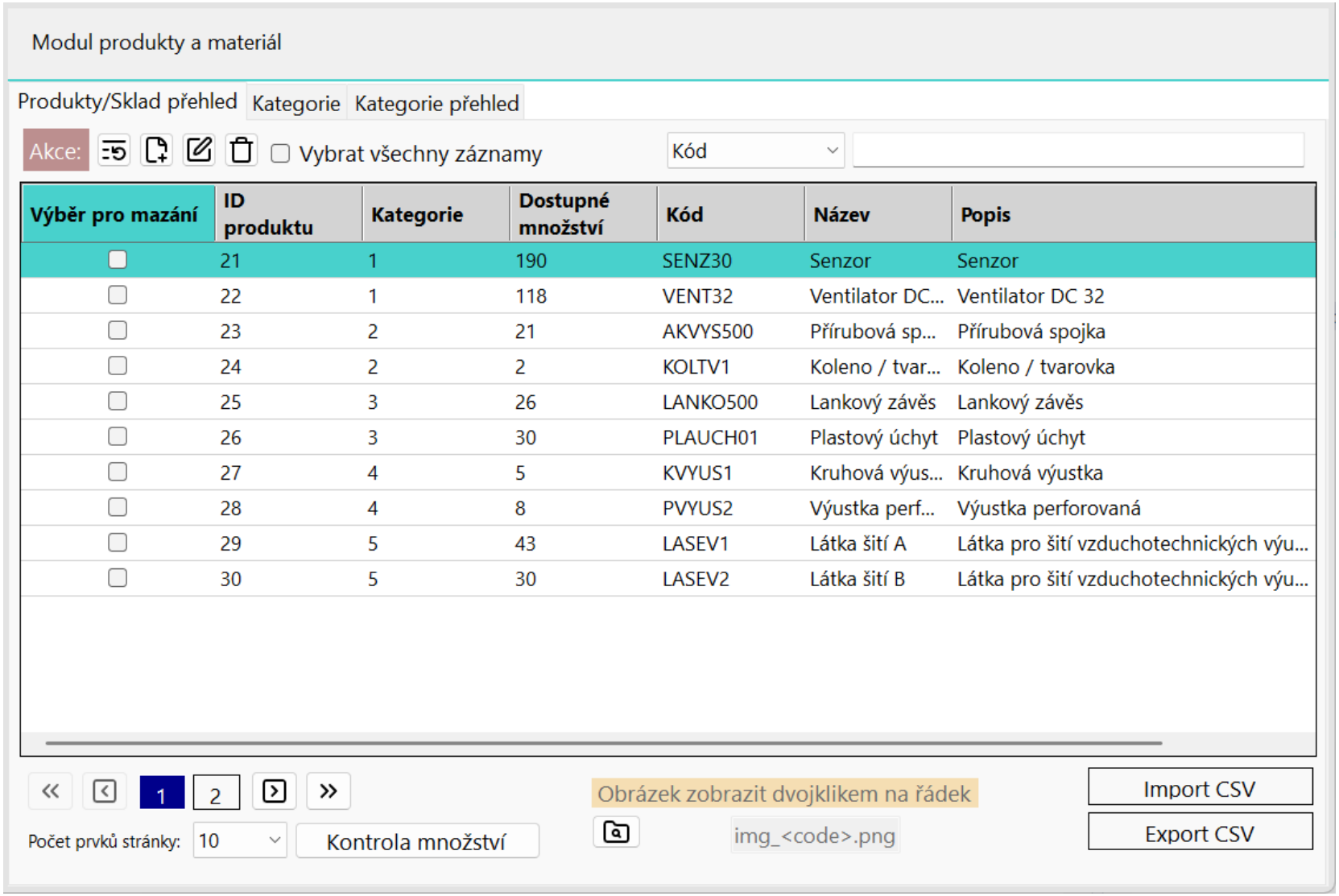
Jedná se o složku Podklady.

```
private void LoadPageOrders()
{
    pagHandler.PaginateOrders(ordersAdapter,
        dtOrders,
        nameof(LoadPageOrders),
        ref pageSize,
        ref tPages,
        ref cPage);

    pgTotalNum.Text = tPages.ToString();

    dataGridViewOrders.DataSource = dtOrders;

    List<Button> btns = LPButtons1(btnFirst,
        btnPrev,
        btnNext,
        btnLast,
        cPage,
        tPages,
        PaginationButton_Click
        );

    Control[] controls = btns.ToArray();
    pagButtons.Controls.Clear();
    pagButtons.Controls.AddRange(controls);
}
```

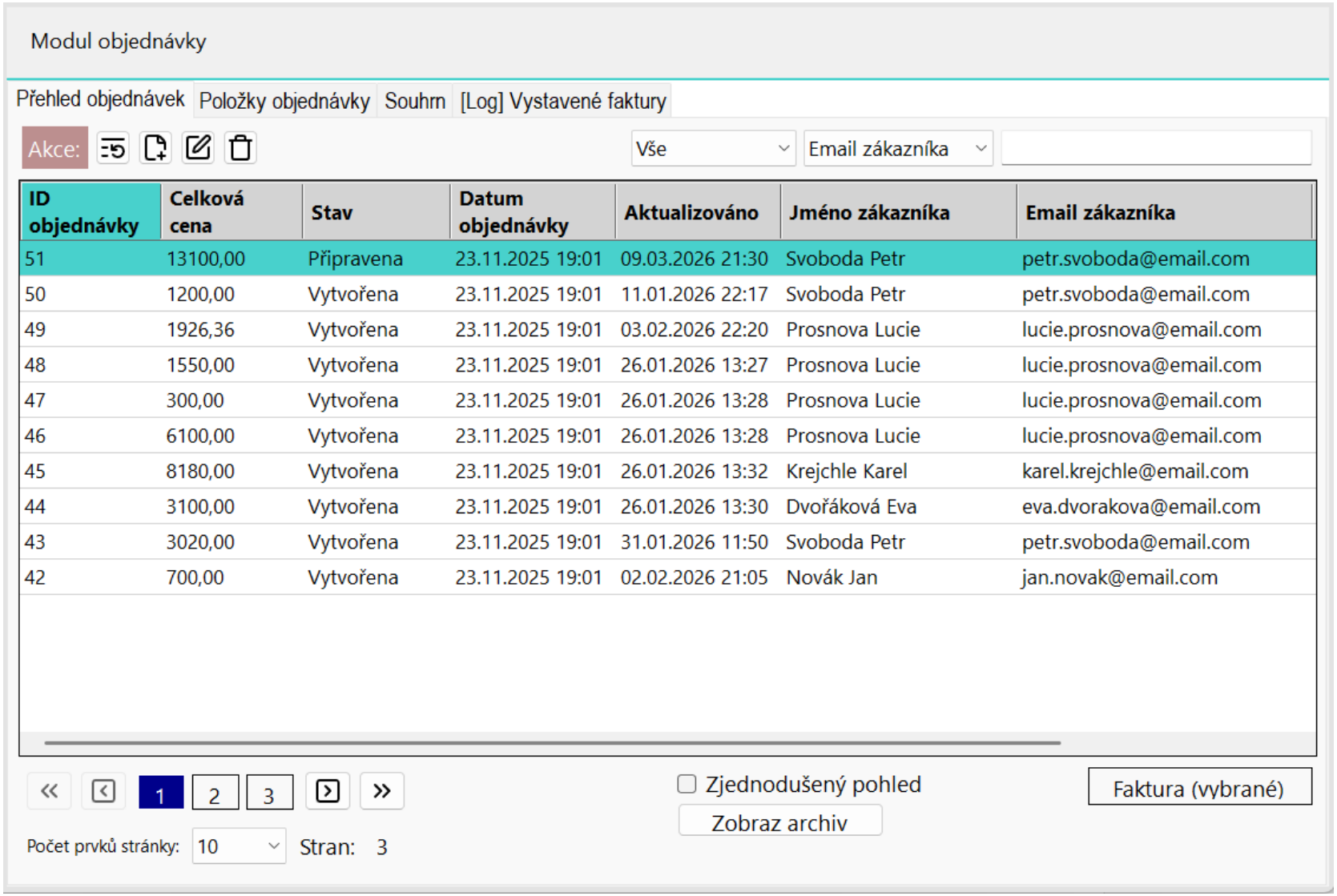
Příloha A.1: Obrázek metody LoadOrders



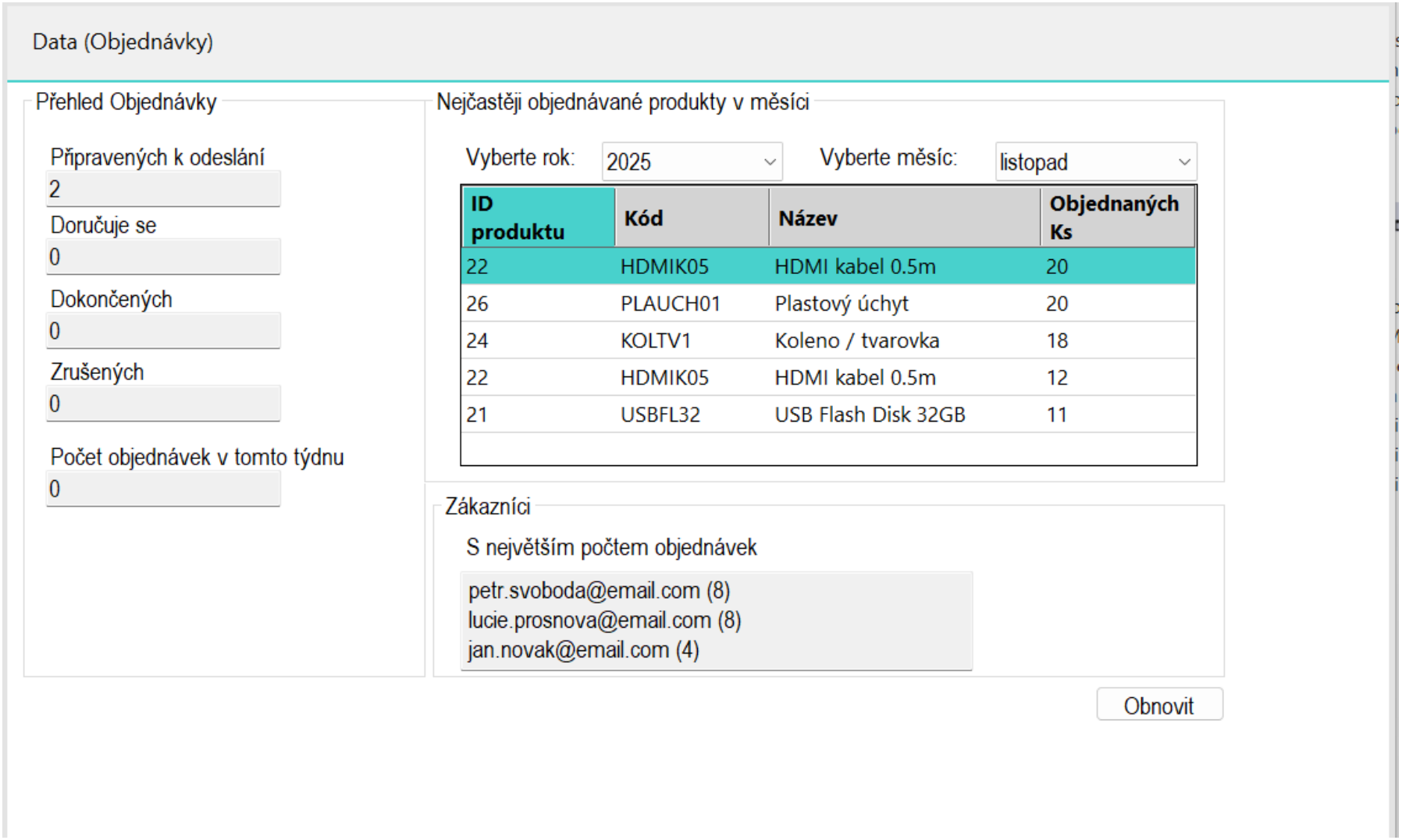
Příloha A.2: Obrázek modulu uživatelé



Příloha A.3: Obrázek modulu produkty



Příloha A.4: Obrázek modulu objednávek



Příloha A.5: Obrázek modulu Data (objednávky)

Příloha A.6: Obrázek ER diagramu