НАВЧАЛЬНО-МЕТОДИЧНА ЛІТЕРАТУРА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ
УНІВЕРСИТЕТ ІМЕНІ ІВАНА ПУЛЮЯ

Кафедра приладів і контрольно-вимірювальних системи

К О Н С П Е К Т Л Е К Ц І Й

Основи інформаційних систем

для студентів спеціальності
176 – «Мікро- та наносистемна техніка»

ТЕРНОПІЛЬ, 2023

Конспект лекцій розроблений у відповідності з навчальним планом за спеціальністю 176 – Мікро-та наносистемна техніка.

Укладачі: д.т.н., Паламар М.І., к.т.н., Стрембіцький М.О.

Відповідальний за випуск: завідувач кафедри приладів і контрольно-вимірювальних систем Паламар М.І.

Розглянуто та затверджено на засіданні приладів і контрольно-вимірювальних систем Тернопільського національного технічного університету імені Івана Пулюя, протокол № 7 від «1» травня 2023 р.

Схвалено та рекомендовано до друку науково-методичною комісією факультету прикладних інформаційних технологій та електроінженерії ТНТУ, протокол № 10 від «5» травня 2023 р.

Конспект лекцій складений з врахуванням методичних розробок інших вищих закладів освіти, а також матеріалів літературних джерел, перелічених в списку.

ЗМІСТ

ЛЕКЦІЯ 1 АЛГОРИТМИ В ПРОГРАМУВАННІ. ASP.NET.

1.1 Вступ до алгоритму в програмуванні

Сьогодні весь світ оцифровується. Почуття інтелекту, відчуття спілкування в кожному традиційному пристрої, що робить наше життя таким легким, таким швидким. Всі ці технологічні досягнення просуваються за допомогою програмного забезпечення, що представляє собою купу програм, призначених для вирішення проблеми. І кожна програма будується на логіці / рішенні, яке називається алгоритмом. Алгоритм імені названий на честь розумної людини з Багдаду, Аль Хварізмі. Він був першою людиною, яка представила алгоритми світу, які були механічними, точними та однозначними.

Що таке алгоритм?

Стандартним визначенням підручника було б - алгоритм - це чітко визначене покрокове рішення або ряд інструкцій для вирішення проблеми. Алгоритм може бути методом пошуку найменш поширеного кратного двох чисел або рецептом приготування Вега Маньчжурського.

Що таке алгоритм з точки зору програмування?

Розумієте, комп’ютер в основному робить багато математики, а значить, має вирішити багато проблем. Саме тому алгоритми складають серце інформатики. Комп'ютерний алгоритм - це обчислювальна процедура, яка приймає набір кінцевих вхідних даних і перетворює їх на вихід, застосовуючи деяку математику та логіку. Алгоритм програмування матиме кілька кроків наступним чином -

1. Визначення проблеми - Що робити?
2. Збір даних - Що ми маємо для вирішення проблеми? Або входи.
3. Обробка даних - розуміння того, що ми маємо, або перетворення їх у зручну форму.
4. Логічний підхід - використання зібраних та створених даних проти логіки для вирішення.
5. Рішення - презентуйте рішення так, як вам потрібно, в графічному інтерфейсі або терміналі, на діаграмі чи діаграмі.

Щоб сказати це в двох словах, з урахуванням кінцевого вхідного значення для x, алгоритм перетворює його на ефективне вихідне значення y, де y f (x) для деякої чітко визначеної функції f.

Один важливий аспект, який потрібно знати, полягає в тому, що алгоритми не строго пов'язані з будь-якою мовою програмування. Вони є загальними рішеннями як такими.

Як алгоритм програмування робить роботу такою простою?

Тематичне поле алгоритмів стало настільки глибоким і широким, що закладені теорії та основи допоможуть нам атакувати будь-яку обчислювальну проблему. Існує так багато ефективних алгоритмів, які вже публікуються, як двійковий пошук, сортування міхурів, сортування вставки, сортування злиття, швидке сортування, алгоритми Евкліда для пошуку GCM, алгоритми Прима, щоб знайти найкоротший шлях у графіку тощо.

Існує так багато видів алгоритмів, як -

Алгоритми грубої сили. Які прямий пробний та помилковий підхід до вирішення проблем? Так само, як ви повторюєте додавання, щоб знайти результат проблеми множення.

Алгоритми ділення та перемоги. Які розбивають проблему на невеликі підпроблеми, а потім поєднують результат кожної підпрограми, щоб отримати кінцевий результат. Так само, як спочатку ви розділите монети різних номіналів на різні відра, а потім порахуйте кількість монет у кожному відрі, щоб знайти, скільки монет окремих номіналів є.

Жадібні алгоритми Які слідують за евристикою вирішення проблем, щоб досягти наступного найкращого стану, щоб знайти як результат остаточний кращий стан. Так само, як ви знайдете менш крутий район, який легко піднімається на гору.

Динамічне програмування. Підхід, який є тим самим, як ділити і перемагає, але розділяє проблему на підпрограми, таким чином, щоб їх результати були повторно використані для інших підпроблем.

Такі методології допомагають нам створити хороший алгоритм, який має такі визначальні характеристики. Хороший алгоритм -

1. Точність - вона знає точні та правильні кроки для виконання.
2. Унікальний - вхід для поточних інструкцій походить лише з попередньої інструкції.
3. Кінцеве - Алгоритм закінчується даванням результату після виконання кінцевої кількості інструкцій.
4. Загальність - Алгоритм має добрий набір входів, а не суворо один вхід.

Переваги алгоритму і чому ми повинні використовувати алгоритм у програмуванні?

Більше, ніж широкий горизонт застосувань у реальному світі, алгоритми виконують роль потужної лінзи, яку можна побачити через проблему. Алгоритм допомагає нам вирішити, проблема вирішувана чи ні. Якщо так, то як, наскільки швидко і наскільки точно? Якщо ні, то алгоритм знову допомагає нам вирішити, чи зможемо ми вирішити його частину.

Говорячи про те, чому нам слід використовувати алгоритми в програмуванні, ми повинні розуміти, що комп'ютерні програми приймають різні алгоритми, що працюють на комп'ютерному обладнанні, яке має процесор і пам'ять, і ці компоненти мають обмеження. Процесор не є нескінченно швидким, і пам'ять у нас не вільна. Вони обмежені ресурси. Вони повинні використовуватися з розумом, і хороший алгоритм, ефективний з точки зору часових складностей і просторових складностей, допоможе вам це зробити.

Як ця технологія допоможе вам у вашому кар’єрному зростанні?

Як і будь-які інші технології, дизайн алгоритмів в програмуванні також постійно змінюється, оскільки апаратне забезпечення комп'ютера постійно розвивається. Починаючи від традиційних машин x86 до суперкомп'ютерів до квантових комп’ютерів, відбулися революційні зміни в способі вирішення проблем. Маючи сильні знання про алгоритм, це відрізняє кваліфікованого програміста від решти. Сучасні ресурси насправді не вимагають вивчення алгоритмів з такою кількістю розроблених програмних систем та бібліотек, але глибоке розуміння цього питання допоможе вам набагато більше.

Алгоритм програмування

Незважаючи на те, що коли-небудь у нас є неймовірно швидкий процесор і пам'ять, яка є безперервною, ми все одно повинні вивчити алгоритм, спроектувати їх так, щоб побачити, чи рішення припиняється і чинить це з правильним результатом. Нехай це будуть комерційні програми, наукові обчислення, інженерія, оперативні дослідження чи штучний інтелект у кожній галузі, що формулює проблеми, з'ясування ефективних алгоритмів для вирішення та структури даних для вирішення залишаться неминучими назавжди.

Так само, як це важливий план перед роботою. Важливо визначити алгоритм перед кодуванням.

1.2 ASP.NET

Активні серверні сторінки (ASP) - це програма веб-додатків, призначена для архітектури динамічних веб-сторінок, веб-додатків та веб-служб. У 1998 році Microsoft вперше представила свою мову сценаріїв на стороні сервера як класичний ASP, написаний у VBScript із розширенням файлу ".asp". ASP.NET є спадкоємцем класичного ASP, який вийшов та відкрив Sourced Microsoft у 2002 році з розширенням файлу .aspx, що зазвичай пишеться на C # (C Sharp). Пізніше в 2016 році Microsoft випустила ASP.NET Core, повноцінну систему застосувань, яка об'єднує ASP.NET MV, WEB API та веб-сторінки.

Зазвичай називається ASP +, ASP.NET - це функція Інтернет-сервера інформації Інтернету Microsoft (IIS), яка дозволяє веб-технократам та веб-розробникам динамічно створювати приплив веб-сторінок, просто вставляючи запити у реляційну базу даних на веб-сторінці. ASP.NET абсолютно інший та унікальний у порівнянні зі своїм попередником двома основними способами: -

- Він підтримує код, написаний на компільованій мові, такі як visual basic, C #, C ++ & Perl.
- Дозволяючи WYSIWYG (те, що ви бачите, те, що ви отримуєте) редагування сторінок, його функції серверних елементів управління можуть відокремлювати код від вмісту.

Розуміння ASP.NET

- ASP.NET виконує веб-сторінку, використовуючи об'єктно-орієнтований підхід до програмування. Кожен елемент asp.net - це об'єкт, який працює на сервері. Компілятор, сумісний з виконанням .NET, сумісний з виконанням часу, відповідає веб-сторінці .NET в проміжній мові, тоді компілятор JIT перетворює проміжний код у вихідний машинний код, і цей машинний код з часом запускається на процесор.
- Для розробника ASP.net дуже важливо зрозуміти життєвий цикл сторінки ASP.NET та життєвий цикл програми ASP.NET. Дуже важливо знати, як запит обробляється за допомогою IIS та як веб-сторінка обробляється та обслуговується користувачеві.
- Оскільки розробнику необхідно купувати дороге програмне забезпечення (зване Інтегровані середовища розвитку або IDE) для створення додатків, вартість розробки та програмного забезпечення є важливим фактором, який слід враховувати при створенні веб-сайту чи веб-додатків. Окрім цього, до програм ASP.NET можна отримати доступ до всіх популярних баз даних, MySQL, MariaDB, Postgres, Microsoft SQL Server, CouchDB та MongoDB. Низька вартість,

висока швидкість та широка підтримка мови є одними з найбільш значущих переваг використання Asp.NET.

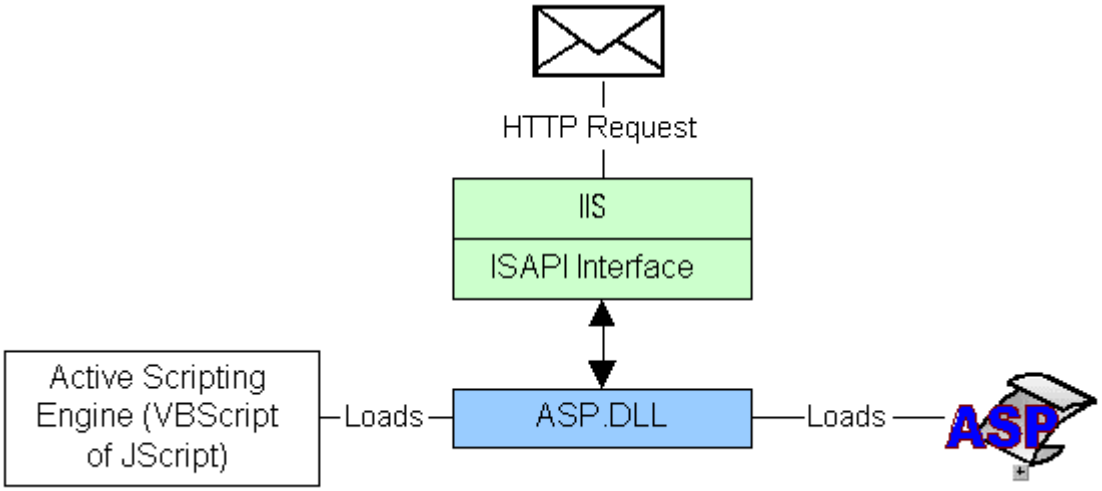
Як ASP.NET робить роботу такою простою?

ASP.net дає нам можливість виконувати кілька завдань, використовуючи єдину рамку, що робить її кращою порівняно з іншими рамками веб-розробки. Деякі його особливості такі:

- Відповідність стандарту
- Виявлення можливостей браузера
- Властивості стилю на контролі
- Адаптери управління
- Зворотні сторінки та керування викликами
- Надійність та продуктивність
- Довгі запити (більше 110 секунд)
- SqlMembershipProvider
- Асинхронні події сторінки з веб-формами
- Запити орган юридичної особи
- Пожежно-незабутня робота
- УвімкнутиViewState та ViewStateMode
- Перенаправлення та відповідь. Закінч
- Безпека
- Налаштування додатків
- EnableViewStateMac
- Середня довіра
- Попросити перевірку
- Аутентифікація форм та сесії без cookie

1.3 Робота ASP.NET

ASP.NET-Workflow можна пояснити, використовуючи наступну схему з детальним поясненням.



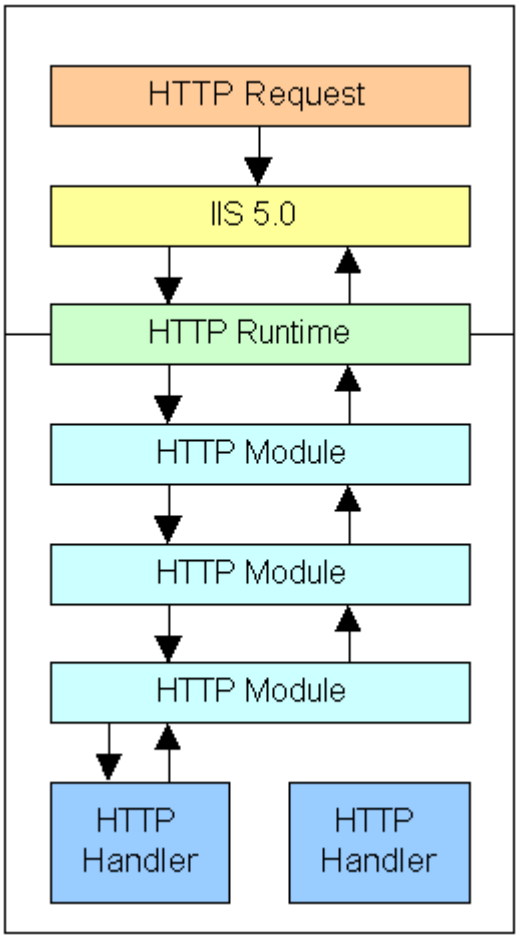
1. Через URL-адресу веб-браузер надсилає запит на файл asp.net на веб-сервер.
2. Як процес, запит отримує веб-сервер asp.net, тобто IIS, який взамін дає відповідний файл з пам'яті.

3. Тепер для обробки файлу asp.net він перенаправляється до двигуна сценарію asp.net для подальшого завдання веб-сервером.

4. Двигун сценарію запускає скрипт на стороні сервера, на який він стикається зверху вниз файлу.

5. Після того, як механізм сценаріїв завершить сценарій на стороні сервера, він надсилає веб-сервер HTML-сторінку.

6. Це пояснюється за допомогою наведеної нижче моделі.



Переваги ASP.NET

З точки зору розробника asp.net, деякими найбільшими перевагами використання asp.net є:

- Час кодування набагато менше в рамках технології asp.net.
- Це краще, ніж поза коробкою.
- Додатки, вбудовані в систему asp.net, занадто безпечні.
- Visual Studio, основа для asp.net, має додатковий дизайн та багатий набір інструментів.
- ASP.NET забезпечує постійний моніторинг.
- З asp.net розгортання простіше, ніж будь-коли.
- Простіше писати та підтримувати сторінки.
- Виконання ASP.NET ретельно керує та контролює всі процеси

Навички, необхідні для ASP.NET

Основними навичками, необхідними для asp.net є:

- Повинен мати знання asp.net MVC.
- Додаток бази даних - повинен мати знання бази даних SQL.

- Повинен знати технологій веб-розробки на стороні клієнта.
- Розуміння ООП (об'єктно-орієнтоване програмування).
- Досвід роботи в VB.Net, C #, MVC та ін.

1.4 Сфера застосування ASP.NET

Майбутнє технології ASP.NET дуже яскраве та велике відповідно до майбутніх 20-25 років, оскільки це стало провідною платформою сьогодні для веб-розробок. Оскільки ядро ASP.net є відкритим кодом і воно дуже швидко зростає, можна сказати, що в майбутньому він має дуже широку сферу розвитку. Мало того, що компанії, але організації та технократи також використовують технологію asp.net для розробки веб-сторінок, оскільки Microsoft має дійсно гарну стратегію розвитку .net в майбутньому.

Хто є потрібною аудиторією для вивчення ASP.NET технологій?

Правий кандидат у галузі asp.net повинен володіти такими навичками:

- Він повинен глибоко знати основні рамки asp.net.
- Він повинен мати можливість розрізняти основні навички .net та .net.
- Він повинен знати такі мови, як C #, візуальні основи, F #, C ++ тощо.
- Він повинен знати рамки зв'язку, такі як WCF, ASP.NET Web API, 1-2, Web Services.

Як ця технологія допоможе вам у кар’єрному зростанні?

З новим прогресом та розвитком технологій asp.net, якими керує Microsoft, це робить вічнозеленим кандидатом продовження своєї кар'єри в галузі asp.net-технологій. Майбутнє цієї дванадцятирічної компанії здається дуже освіченим у майбутньому, тому можна використовувати цю мову як свій варіант кар’єри в IT-сфері, оскільки вона допомагає Microsoft, тому немає шансів, що технологія asp.net буде поза ринком у майбутньому.

Висновок

ASP.NET робить Інтернет як програмну платформу набагато простіше в управлінні веб-програмуванням. Технологія Asp.net дає нам повну свободу контролювати наш розвиток і може бути використана в будь-якому місці, будь то невеликий чи великий проект. Оскільки ним управляє Microsoft, це ніколи не може бути вчорашня технологія, оскільки Microsoft вклала багато коштів у свою розробку та співтовариство. Таким чином, в цілому, можна сказати, що ASP.NET, безсумнівно, стане інструментом вибору для більшості веб-розробників протягом наступних п'яти-десяти років.

ЛЕКЦІЯ 2 WEB-РОЗРОБКИ

2.1 Коротка історія WEB-розробки

Для того щоб зрозуміти різні аспекти та дизайнерські завдання ASP.NET MVC, варто розглянути історію WEB-розробки хоча б коротко. Протягом багатьох років платформи для WEB-розробки від Microsoft демонстрували зростаючу потужність і, на жаль, зростаючу складність. Як показано в табл. 1.1, кожна нова платформа видаляла конкретні недоліки свого попередника.

Таблиця 2.1 – Хронологія розвитку технологій WEB-розробки від Microsoft

Період	**Технологія**	**Сильні сторони**	**Слабкі сторони**
Юрський період	Common Gateway Interface («загальний інтернет шлюзу»), CGI	Простота, гнучкість та єдиний варіант рішення на цей час	Працює поза WEB-сервером, таким чином, є ресурсомістким (використовує окремий процес операційної системи для кожного запиту)
Бронзова епоха	Microsoft Internet Database Connector, IDC (конектор без даних для Інтернету)	Робота всередині WEB-сервера	Просто оболонка для SQL запитів та шаблонів для форматування множини результатів
1996	Active Server Pages, ASP (активні серверні сторінки)	Загальна мета	Інтерпретується під час виконання; підтримує "спагетті-код"
2002/03	ASP.NET Web Forms 1.0/1.1	Скомпільований; UI, що зберігає стан (stateful); велика інфраструктура; підтримує об'єктно орієнтоване програмування	Важкий по пропускній здібності; некрасивий HTML; етестований
2005	ASP.NET Web Forms 2.0		
2007	ASP.NET AJAX		
2008	ASP.NET Web Forms 3.5		
2009	ASP.NET MVC 1.0		
2010	ASP.NET MVC 2.0, ASP.NET Web Forms 4.0		
2011	ASP.NET MVC 3.0		
2012	ASP.NET MVC 4.0, ASP.NET Web Forms 4.5		

Примітка

CGI є стандартним засобом підключення веб-сервера до довільно виконуваної програма, яка повертає динамічний контент. Специфікація підтримується Національним центром суперкомп'ютерних програм (NCSA)

2.2 Традиційна ASP.NET WEB форми

ASP.NET був величезним проривом, коли вперше з'явився у 2002 році. На рис. 2.1 показаний стек технологій Microsoft, як ми тепер їх спостерігаємо

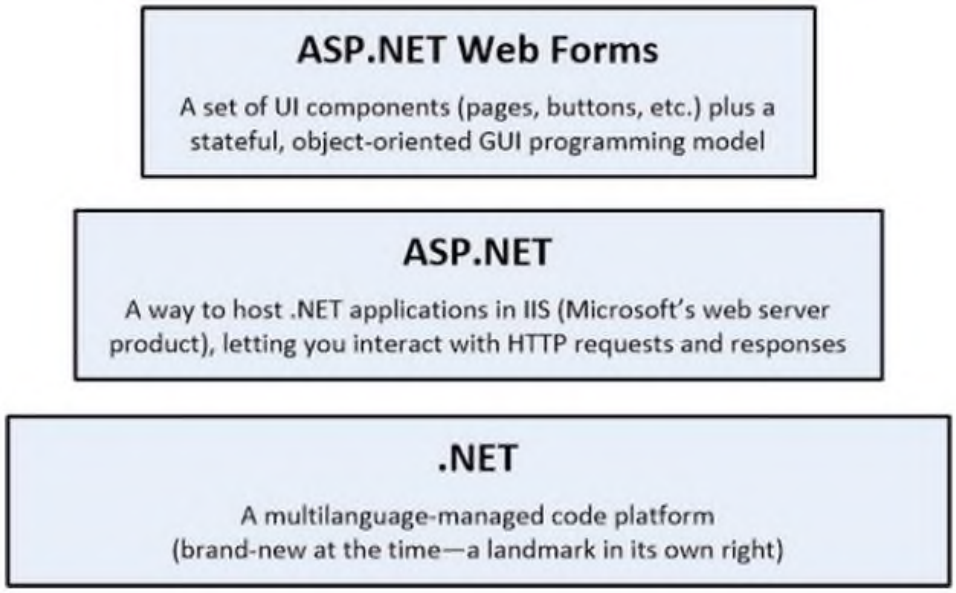


Рисунок 2.1 – Стек технологій **ASP.NET Web Forms**

У Web Froms Microsoft спробував сховати роботу як з HTTP, так і з HTML (які на той час були незнайомі для багатьох розробників) за допомогою моделювання користувача інтерфейсу (UI) як ієрархії об'єктів, керованих із боку сервера. Кожен елемент управління стежив за своїм станом за всіх запитів (за допомогою можливості View State), якщо потрібно, обробляючи себе як HTML і автоматично підключаючись до подій на стороні клієнта (наприклад, для натискання кнопки) з відповідним кодом для обробника подій на серверної сторони. По суті, веб форми є гігантським шаром абстракції, призначений для доставки класичного графічного інтерфейсу користувача (GUI) через Інтернет.

Ідея полягала в тому, щоб зробити веб-розробку на кшталт розробки Windows Forms. Розробникам більше не потрібно було б працювати із серією незалежних HTTP запитів та відповідей; ми змогли б працювати за умов UI, що зберігають стан (stateful). Ми могли б забути про Мережу та її природу "незбереження стану" і натомість вибудовувати інтерфейси за допомогою конструктора «drag-and-drop», і уявіть собі, або принаймні вдайте, що все це відбувається на сервері.

2.3 ASP.NET Web Forms

В принципі, традиційна розробка ASP.NET Web Forms була дуже класною, але реальність виявилася більш вимогливою. Згодом використання веб форм у реальних проектах показало деякі їхні недоліки:

- *Вага View State*: В результаті використання актуального механізму підтримки стан між запитами (відомого як View State) ми отримали великі блоки даних, що передаються між клієнтом та сервером. Ці дані можуть досягати сотень кілобайт навіть для скромних веб-додатків, і вони йдуть туди і назад при кожному запит, що призводить до збільшення часу відгуку та підвищення вимог до пропускну здатність сервера.

- *Життєвий цикл сторінки*: Механізм для об'єднання події з боку клієнта з кодом серверного обробника події – частина життєвого циклу сторінки – можливо надзвичайно складним та делікатним. Небагато розробників досягли успіху в маніпуляції з елементами управління під час виконання коду, не отримавши помилок View State або не виявивши, що деякі обробники подій таємниче не виконувалися.
- *Неправильний розподіл завдань*: Модель виділеного коду (code-behind) ASP.NET надає можливість для того, щоб винести код програми за рамки HTML розмітки на окремий клас виділеного коду. Це широко віталося через поділу логіки та уявлення, але, насправді, розробники змушені змішувати код подання (наприклад, маніпуляції з деревом серверних елементів) управління) з логікою програми (наприклад, управлінням базами даних) у цих класи виділеного коду, які стають просто жахливими. Кінцевий результат може бути недовговічним та незрозумілим.
- *Обмежені можливості з HTML*: Серверні елементи керування відображають себе як HTML, але не обов'язково так, як ви хочете. До версії ASP.NET 4 вихідних даних HTML не вдавалося відповідати веб стандартам або добре працювати з каскадними таблиці стилів (CSS). Також серверні елементи керування генерували непередбачувані та складні значення атрибуту ID, до яких важко отримати доступ при допомоги JavaScript. Ці проблеми багато в чому вирішилися в ASP.NET 4 і ASP.NET 4.5, але у вас все ще можуть виникнути складнощі в отриманні HTML, який ви очікуєте.
- *Абстракції з прогалиною*: Web Forms намагається сховати HTML та HTTP, де це тільки можливо. Коли ви намагаєтеся реалізувати власні механізми поведінки, ви часто можете випасти з абстракцій, які змушують вас переробляти механізм зворотної передачі подій або виконувати дурні дії, щоб він згенерував бажаний HTML. Крім того, всі ці абстракції можуть стати неприємним бар'єром для компетентних веб-розробників.
- *Слабке тестування*: Розробники ASP.NET не могли припустити, що автоматизоване тестування стане важливим компонентом розробки програмного забезпечення. Не дивно, що жорстка архітектура, яку вони розробили, не підходить для модульного тестування (юніт-тестування). З інтеграційним Тестуванням також можуть виникнути проблеми.

ASP.NET продовжував розвиватись. У версії 2.0 було додано набір стандартних компонентів для додатків і вони можуть зменшити обсяг коду, який вам потрібно писати самостійно. Реліз AJAX у 2007 році був відповіддю Microsoft на безумство Web 2.0/AJAX, він підтримував гарна взаємодія зі стороною клієнта, не ускладнюючи життя розробникам. Все набагато покращилося з релізом ASP.NET 4, який вперше найсерйознішим чином прийняв веб стандарт. Останній реліз ASP.NET 4.5, насправді, має деякі риси ASP.NET MVC і застосовує їх до світу Web Forms, що допомагає вирішити деякі досить значущі проблеми, але, незважаючи на це, багато внутрішні обмеження все ж таки присутні.

2.4 Веб стандарти та REST

Останніми роками зросла потреба відповідати веб стандартам. Веб сайти призначаються для різних пристроїв і браузерів, ніж будь-коли раніше, і веб

стандарти (*HTML, CSS, JavaScript* тощо) залишаються нашою однією великою надією насолоджуватися пристойним переглядом сайтів та додатків скрізь, навіть по холодильниках з інтернету підтримкою. Сучасні Web платформи не можуть дозволити собі ігнорувати ідеї бізнесу та бажання розробників відповідати веб стандартам.

Повсюдно починає використовуватись HTML5. Він надає веб розробникам багаті можливості, що дозволяють виконувати роботу, яка раніше була винятковою прерогативою серверів на стороні клієнта. Ці нові можливості і зрілість, що настає бібліотек *JavaScript*, таких як *JQuery*, *JQuery UI*, та *JQuery Mobile*, позначають, що стандарти стають все більш важливими і складають основу для багатших веб-додатків.

Примітка

Розглянемо HTML5, jQuery без подробиці, тому що ці технології самі по собі заслуговують на особливу увагу.

У той самий час REST (Representational State Transfer, «передача уявлень станів») став домінуючою архітектурою для взаємодії програм через HTTP, повністю затьмарюючи SOAP (технологія, що лежить в основі оригінального підходу ASP.NET до веб-сервісів). REST описує додаток в умовах ресурсів (URI), представляючи реальні об'єкти та стандартні операції (HTTP методи), відображаючи доступні операції на цих ресурсах. Наприклад, ви можете використовувати PUT з новим http://www.example.com/Products/Lawnmower або DELETE з http://www.example.com/Customers/Arnold-Smith.

Сучасні веб-програми обслуговують не тільки HTML. Найчастіше вони також повинні обслуговувати дані JSON або XML для різних технологій клієнта, включаючи AJAX, Silverlight, та рідних додатків для смартфонів. Це відбувається, звичайно, з REST, який усуває історичні відмінності між веб-сервісами та веб-додатками, але вимагає такого підходи до обробки HTTP та URL, які не так легко підтримуються ASP.NET Web Forms.

2.5 Екстремальне програмування та розробка через тестування (test-driven development, TDD)

Не тільки веб-розробка розвивалася в останнє десятиліття - розробка програмного забезпечення, загалом, також змістилася на шлях екстремального програмування. Це може означати багато різних речей, але в основному йдеться про створення програмних проектів як адаптаційних процесів відкриття та розробки без обтяжливого та обмежуючого перспективи планування Бажання екстремального програмування, як правило, йде рука об руку з певним набором програмістських навичок та використовуваних засобів (як правило, з відкритим вихідним кодом), які сприяють і допомагають вивченню та використання таких технологій.

TDD (розробка через тестування) та його останнє втілення, **BDD**, є двома очевидними прикладами. Ідея полягає в розробці програмного забезпечення з початкового опису прикладів бажаної поведінки (відомого як тести або специфікації), так що у будь-який момент ви можете перевірити стабільність та правильність вашої програми, виконуючи набір тестів щодо реалізації. .NET

інструментів вистачає для підтримки **TDD/BDD**, але вони, як правило, не дуже добре працюють із Web Forms:

- *Інструменти для модульного тестування* дозволяють визначити поведінку окремих класи або інші невеликі частини коду в ізоляції. Вони можуть бути ефективно використані тільки для програмного забезпечення, яке було спроектовано як набір незалежних модулів, тому кожен тест може бути запущений окремо. До На жаль, деякі програми Web Forms можуть бути протестовані таким чином. Наслідуючи керівництво фреймворку про те, щоб вставити логіку в обробники подій або навіть використовувати серверні елементи керування, які безпосередньо надсилають запити до баз даних, розробники зазвичай, зрештою, щільно пов'язують логіку програми із середовищем виконання Web Forms. Це занепад для модульного тестування.
- *Інструменти автоматизованого тестування інтерфейсу* користувача дозволяють моделювати ряд взаємодій користувача із запущеним екземпляром програми. Теоретично ці тести можуть бути використані з Web Forms, але вони можуть не спрацювати, коли ви зробите невеликі зміни у макеті сторінки. Не звертаючи на це увага, Web Forms починає генерувати зовсім інші HTML структури та ID елементів, так що наявний набір тестів стає марним.

Спільнота розробників .NET програм з відкритим вихідним кодом та незалежний постачальник програмного забезпечення (ISV) створили високоякісні фреймворки для модульного тестування (NUnit та XUnit), фіктивні фреймворки (Moq та Rhino Mocks), інверсійні контейнери (Ninject та Autofac), сервери безперервної інтеграції (Cruise Control та TeamCity), об'єктно-реляційні мапери (NHibernate та Subsonic) тощо. Прихильники цих інструментів та методів подали свій голос, створюючи публікації та організовуючи конференції під спільним брендом ALT.NET. Традиційна технологія ASP.NET Web Forms не підтримує ці інструменти та методи через свій монолітний дизайн, тому у цієї групи експертів Web Forms не користується особливою повагою.

2.6 Ruby on Rails

У 2004 році **Ruby on Rails** був не дуже відомою технологією з відкритим вихідним кодом від невідомий гравець. І раптом настала слава, яка перевернула правила веб-розробки. Не те, щоб Ruby на Rails містив революційні технології, але ця концепція взяла існуючі інгредієнти і змішала їх таким переконливим та привабливим чином, що існуючі платформи почали горіти для сорому.

Ruby on Rails (або просто Rails, як його зазвичай називають) прийняв архітектуру MVC (яку ми опишемо трохи нижче). Застосовуючи MVC і працюючи в гармонії з HTTP-протоколом, а не проти нього, сприяючи конвенціям замість потреби у конфігурації та інтегруючи інструментарій об'єктно-реляційного мапінгу (ORM) у своїй основі, програми Rails зайняли своє місце без особливих зусиль. Це виглядало так, як мала б виглядати веб розробка весь час, як якби ми раптом зрозуміли, що наші інструменти всі ці роки воювали, а тепер війна закінчилася.

Rails показує, що відповідність веб стандартам і REST не повинна бути жорсткою. він також показує, що екстремальне програмування та TDD працюють

найкраще з фреймворками, які їх підтримують. Решта світу розробки досі до цього підтягується.

2.7 Sinatra

Завдяки Rails незабаром багато веб-розробників стали використовувати Ruby як основну мови програмування. Але в такому суспільстві, що інтенсивно розвивається, це було тільки питанням часу, коли з'явиться альтернатива Rails. Найвідоміша, Sinatra, з'явилася у 2007 році.

Sinatra відкидає майже всю стандартну інфраструктуру Rails (маршрутизацію, контролери, представлення і т.д.) і просто картує URL шаблони для блоків коду Ruby. Відвідувач запитує URL, що викликає блок коду Ruby, який буде виконано, і дані передаються назад браузеру - ось і все. Це неймовірно простий вид веб-розробки, але він знайшов свою нішу у двох основних напрямках. По-перше, для веб-сервісів, що підтримують REST, він просто швидко виконує свою роботу. По-друге, оскільки Sinatra може бути підключена до широкого спектру HTML шаблонів з відкритим вихідним кодом ORM технологіям, вона часто використовується як основа, на якій збирається користувальницький Web фреймворк відповідно до архітектурних потреб будь-якого проекту, що є під рукою.

Sinatra ще має відвоювати свою частку ринку у серйозних MVC платформ, таких як Rails (або ASP.NET MVC). Ми згадуємо її лише для ілюстрації поточних тенденцій індустрії веб розробки в бік спрощення, а також тому що Sinatra виступає як протилежної сили щодо інших фреймворків, дедалі більше накопичуючи у собі основний функціонал.

2.8 Node.js

Іншою важливою тенденцією є рух у бік використання JavaScript як основної мови програмування AJAX вперше показав нам, що JavaScript дуже важливий, JQuery показав нам, що він може бути потужним та елегантним, а драйвер Google JavaScript V8 з відкритим вихідним кодом показав нам, що може бути неймовірно швидким. Сьогодні JavaScript стає серйозною серверною мовою програмування. Він служить сховищем даних та є мовою запитів для кількох нереляційних баз даних, у тому числі CouchDB та Mongo. Також він використовується як універсальна мова на серверних платформи, такі як Node.js.

Node.js з'явився приблизно у 2009 році і здобув широке визнання дуже швидко. Архітектурно він схожий на Синатру в тому, що він не застосовує MVC патерн. Це більше низькорівневий спосіб підключення запитів HTTP до коду. Його основні нововведення полягають у наступному:

- *Використання JavaScript*: Розробникам потрібно працювати тільки з однією мовою клієнтського коду до логіки на стороні сервера і навіть до логіки запитів даних з допомогою CouchDB тощо.
- *Абсолютна асинхронність*: Базовий API Node.js просто не блокує потоки під час очікування введення/виводу (I/O) або під час будь-якої іншої операції. Всі I/O здійснюються з початком операції і потім отриманням зворотного виклику, коли I/O завершується. Це означає, що Node.js робить надзвичайно ефективним використання системних ресурсів і може обробляти десятки тисяч одночасних запитів CPU (альтернативні платформи, як правило, обмежуються близько сотні одночасних запитів для CPU).

Як і Sinatra, Node.js є нішевою технологією. Зазвичай більшість бізнес-проектів, створюючи реальні додатки в обмежених часових рамках, потребують повних фреймворки, такі як Ruby on Rails і ASP.NET MVC. Node.js згадується тут тільки для щоб ми могли розглянути інші сучасні технології, а не тільки ASP.NET MVC.

Наприклад, ASP.NET MVC включає асинхронні контролери. Це спосіб обробки HTTP-запитів з неблокованим I/O та обробки більшого числа запитів для CPU Ви побачите, що ASP.NET MVC дуже добре інтегрується з складним JavaScript кодом, що працює у браузері.

ЛЕКЦІЯ 3 ASP.NET MVC. MVC ПАТЕРН

3.1 Основні переваги ASP.NET MVC

ASP.NET став великим комерційним успіхом, але, як говорилося, решта світу веб розробки також розвивався, і хоча Microsoft здував пил з Web Forms, її основні конструкції почали виглядати дуже застарілими.

У жовтні 2007 року на першій конференції з ALT.NET в Остіні, штат Техас, віце-президент Microsoft Скотт Гатрі оголосив та продемонстрував нову платформу з розробки MVC, побудовану на базовій платформі ASP.NET, явно задуману як пряму відповідь на еволюцію технологій, таких як Rails та як реакцію на критику Web Forms. У наступних розділах описується, як ця нова платформа подолала обмеження Web Forms і знову збільшила ASP.NET.

3.2 Архітектура MVC

Важливо розрізняти архітектурний патерн MVC та ASP.NET MVC Framework. MVC патерн не є новим, його коріння сягає 1978 року і проекту Smalltalk в Xerox PARC, але він завоювала величезну популярність сьогодні як патерн для веб додатків за наступними причин:

- Взаємодія користувача з MVC додатком слідує природному циклу: користувач робить дію, у відповідь на цю програму змінює свою модель даних і надає користувачеві оновлений вигляд. А потім цикл повторюється. Це дуже зручно для веб-програм, що надаються у вигляді серії HTTP запитів та відповідей.
- Необхідність веб-програмі об'єднувати кілька технологій (наприклад, бази даних, HTML і виконуваний код), як правило, розбивається на безліч рівнів або шарів. Моделей, які випливають із цих комбінацій, природні для концепції MVC.

ASP.NET MVC Framework реалізує MVC патерн і тим самим забезпечує значно покращений поділ концепцій. Насправді, ASP.NET MVC реалізує сучасний варіант MVC патерну, який особливо добре підходить для веб-додатків.

Застосовуючи та адаптуючи MVC патерн, ASP.NET MVC Framework сильно конкурує з Ruby on Rails і аналогічні платформи, і переносить MVC патерн в основне русло світу .NET. Підсумовуючи досвід та кращу практику розробників, які використовують інші платформи, можна сказати, що ASP.NET MVC може запропонувати навіть більше, ніж Rails.

3.3 Розширюваність

Внутрішні компоненти настільного ПК є незалежними частинами, які взаємодіють лише через стандартні, публічно документовані інтерфейси. Ви можете легко вийняти відеокарту або жорсткий диск та замінити його іншим від іншого виробника і будете впевнені, що він впишеться в слот і працюватиме. MVC Framework також побудований як ряд незалежних компонентів, що задовольняють .NET інтерфейс або побудовані на абстрактному базовому класі, тому ви можете легко замінити компоненти, такі як система маршрутизації, двигун для перегляду тощо іншими.

ASP.NET MVC дизайнери побудували його таким чином, щоб дати вам три варіанти вибору для кожного компонента MVC Framework:

- Використовувати реалізацію за умовчанням компонента у його нинішньому вигляді бути достатньо більшості додатків).
- Вивести підклас реалізації за умовчанням для настроювання її поведінки.
- Замінити компонент повністю за допомогою нової реалізації інтерфейсу або абстрактного базового класу.

3.4 Жорсткий контроль над HTML та HTTP

ASP.NET MVC визнає важливість отримання чистої, що відповідає стандартам розмітки. Його вбудовані методи HTML помічника надають відповідні стандартам вихідні дані, але існують і більш значні філософські зміни в порівнянні з Web Forms. Замість щоб плодити величезні ділянки HTML, яким нам складно керувати, MVC Framework рекомендується створити простий, елегантний стиль розмітки за допомогою CSS.

Звичайно, якщо ви хочете використовувати деякі готові віджети для складних елементів інтерфейсу користувача, такі як вибір дати або каскадне меню, то вам варто знати, що підхід ASP.NET MVC до розмітки спрощує використання найкращих у своєму роді UI бібліотек, таких, як JQuery UI або бібліотеки Yahoo YUI. Розробники JavaScript будуть раді дізнатися, що ASP.NET MVC так добре спрацював із популярною бібліотекою JQuery, що Microsoft зробив JQuery є вбудованою частиною шаблону проектів ASP.NET MVC і навіть дозволяє безпосередньо посилатися на .js файл jQuery на власних CDN серверах Microsoft.

Сторінки, що згенеровані ASP.NET MVC, не містять жодних даних View State, тому вони можуть бути в сотні кілобайт менше, ніж звичайні сторінки, створені за допомогою ASP.NET Web Forms. Незважаючи на сучасний широкосмуговий зв'язок та швидкі підключення, ця економія пропускної спроможності досі надзвичайно приваблива для кінцевих користувачів.

Як Ruby on Rails, ASP.NET MVC працює в гармонії з HTTP. Ви повністю контролюєте запити, що проходять між браузером і сервером, тому ви можете підігнати налаштування під наскільки вам це подобається. AJAX зроблено просто, і немає ніякого автоматичного зворотного втручання у стані за клієнта. Будь-який розробник, який насамперед черга фокусується на веб програмуванні, майже напевно вважає це звільненням і буде насолоджуватися робочим процесом.

3.5 Тестування

Архітектура MVC дає вам чудову можливість створювати вашу програму таким, щоб його можна було легко супроводжувати та тестувати, тому що ви, звичайно, захочете розділити логічні блоки програми з незалежних частин програмного забезпечення. Проте, Автори ASP.NET MVC на цьому не зупинилися. Для підтримки модульного тестування вони прийняли компонентно-орієнтований дизайн фреймворку та переконалися, що кожна окрема частина побудована те щоб відповідати вимогам модульного тестування.

Вони додали майстри (візарди) Visual Studio для створення початкових модульних проектів тестування від вашого імені, які інтегровані з інструментами модульного тестування з відкритим вихідним кодом, такими як NUnit та XUnit, а також власним MSTest Microsoft. Навіть якщо ви ніколи не писали модульних тестів, вам буде легко у цьому розібратися.

Розглянемо, як писати чисті, прості модульні тести ASP.NET MVC контролерів та дій, з підтримкою фальшивих (fake) та фіктивних (mock) реалізацій фреймворк компонентів для моделювання будь-яких сценаріїв, використовуючи різні стратегія тестування.

Тестування – це не тільки питання модульного тестування. ASP.NET MVC програми також добре працюють із інструментами автоматичного тестування. Ви можете написати тестові скрипти, які імітують взаємодію користувача, без необхідності гадати, які структури HTML елементів, CSS класи або ID буде генерувати фреймворк, та вам не доведеться турбуватися про структуру, якщо вона раптом несподівано зміниться.

3.6 Потужна система маршрутизації (роутингу)

Стиль посилань змінився, оскільки технологія веб-додатків покращилася. Такі посилання, як ця:

/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742

можна зустріти досить рідко. Тепер вони замінені більш простим та чистим форматом:

/to-rent/chicago/2303-silver-street

Є кілька вагомих причин для турботи про структуру URL. По-перше, пошукові системи надають значну вагу ключовим словам, що знаходяться в URL. Пошук "оренда в Чикаго" (rent in Chicago) має набагато більше шансів із простим URL. По-друге, багатьом користувачам Інтернету тепер вистачить навичок та знань, щоб зрозуміти URL, та оцінити можливості Навігація, набравши його в адресному рядку свого браузера. По-третє, коли хтось розуміє структуру URL, він, швидше за все, посилатиметься саме на нього, поділиться цим посиланням з другом або навіть продиктує її вголос телефоном. По-четверте, таке посилання не надає технічні подробиці, папки, імена файлів та структуру програми на весь громадський Інтернет, так що ви можете змінити внутрішню реалізацію, не порушуючи посилання.

У ранніх фреймворках було складно реалізувати чисті посилання, але ASP.NET MVC використовує можливість System.Web.Routing, яка за промовчанням створює чисті URL-адреси. Тепер ви можете контролювати схему посилань та її зв'язок та ставлення до додатка, тобто ви вільні у створенні шаблону URL-адрес, які є важливими та корисними для користувачів, без необхідності відповідати визначеним шаблоном. І, звісно, це означає, що ви можете легко визначити сучасну URL-схему в стилі REST, якщо в цьому є потреба.

Можливість розробляти у найкращих сегментах платформи ASP.NET

Існуюча платформа Microsoft ASP.NET надає зрілий, що добре зарекомендував себе набір компонентів та засоби для розробки ефективних та дієвих веб додатків.

По-перше, і що найбільш очевидно, оскільки ASP.NET MVC базується на. NET платформи, у вас є можливість писати код будь-якою .NET мовою і мати

доступ до тих же API функцій, не тільки до MVC, але й до великої .NET бібліотеки класів та величезної екосистеми сторонніх .NET бібліотек.

По-друге, готові можливості платформи ASP.NET, такі як майстер-сторінки, автентифікація, ролі, профілі та інтернаціоналізація, можуть зменшити кількість коду, який потрібно писати і підтримувати для будь-яких веб додатків, і ці функції так само ефективні при використанні MVC Framework, як і в класичних проектах Web Forms. Виможете знову використовувати деякі вбудовані серверні елементи керування Web Forms, а також свої власні елементи управління з попередніх проектів ASP.NET у додатках ASP.NET MVC (якщо вони не залежать від деяких конкретних можливостей Web Forms, таких як View State).

Сучасний API

З моменту свого створення в 2002 році .NET платформа Microsoft невблаганно розвивалася, підтримуючи та навіть визначаючи аспекти сучасного програмування.

ASP.NET MVC 4 був створений для .NET 4.5, тому його API повною мірою прийняв переваги найсучасніших мов та технологій, у тому числі ключове слово await, методи розширень, лямбда-вирази, анонімні та динамічні типи та LINQ (Language Integrated Query). Багато з методів API MVC Framework і патерни кодування слід чистішим, більш виразним композиціям, ніж це було можливо на ранніх платформах.

ASP.NET MVC має відкритий вихідний код

На відміну від попередніх платформ веб-розробки Microsoft, ви можете завантажити вихідний код для ASP.NET MVC і навіть змінити та скомпілювати власну версію. Це має неоціненне значення, коли ваше налагодження стосується системи компонентів, і ви хочете зайти в код (і навіть прочитати коментарі програмістів-торців). Це також корисно, якщо ви створюєте «просунуті» компоненти та хочете подивитися, які можливості існують для їх розвитку або як працюють вбудовані компоненти.

Крім того, ця можливість хороша у тому випадку, якщо вам не подобається, як щось працює, якщо ви знайшли помилку або якщо ви просто хочете отримати доступ до чогось такого, що в іншому випадку недоступно, тому що ви можете просто змінити це самостійно. Тим не менш, вам потрібно буде відслідковувати зміни і повторювати їх, якщо ви перейдете на новішу версію платформи. ASP.NET MVC ліцензований Microsoft Public License (Ms-PL, http://www.opensource.org/licenses/ms-pl.html), а Open Source Initiative (OSI)-затвердив відкриту ліцензію. Це означає, що ви можете змінювати вихідний код, розгортати його і навіть розповсюджувати ваші зміни публічно, як вами створений проект.

Порівняння з ASP.NET Web Forms

Ми вже докладно описали слабкі сторони та обмеження ASP.NET Web Forms та те, як ASP.NET MVC долає багато з цих складнощів. Однак, це не означає, що технологія Web Forms мертва. Microsoft неодноразово заявляв, що обидві технології активно розвиваються та активно підтримуються, і що немає жодних планів відмовитися від Web Forms. У певному сенсі, вибір між цими двома технологіями є вашою філософією програміста. Давайте їх порівняємо:

- *Web Forms* працює з поданням, як з тим, що зберігає стан (що є stateful), додаючи абстрактний шар для HTTP та HTML. Використовуючи View State та функції зворотного виклику (postback) створення ефекту збереження стану (statefulness). Завдяки цьому можлива розробка drag-and-drop у стилі Windows Forms, тобто ви вставляєте UI віджети в шаблони та заповнюєте кодом обробники подій.
- *MVC* ж включає істинну природу HTTP, працюючи з ним, а не борючись проти. MVC Framework вимагає розуміння, як веб-програми працюють насправді. Завдяки цьому ви відчуєте, що є простим, потужним, сучасним підхід до написання веб-додатків з чистим кодом, який легше розширити і підтримувати протягом тривалого часу без дивних ускладнень та болючих обмежень.

Є, звичайно, випадки, коли Web Forms принаймні можна так само добре, і, напевно, навіть краще використовувати, ніж MVC. Очевидним прикладом є невеликі додатки для інтранет, які переважно прив'язані безпосередньо до таблиць бази даних. Сильні сторони drag-and-drop програмування Web Forms можуть переважити його слабкі сторони, коли вам не потрібно турбуватися про пропускну здатність або пошукову оптимізацію.

Якщо ж ви пишете програми для Інтернету або великі інтранет програми, вас привабить хороша пропускна здатність, краща сумісність із браузерами та більш продумана підтримка автоматизованого тестування, що пропонує MVC.

Перехід від Web Forms до MVC

Якщо у вас є ASP.NET Web Forms проект, який ви плануєте перекласти на MVC, вам буде приємно дізнатися, що ці дві технології можуть співіснувати в одному додатку. Це дає можливість для поступового перенесення існуючих програм, особливо якщо програма розбивається на шари з моделлю предметної області або бізнес логіки, які обмежені від сторінки Web Forms. У деяких випадках, можливо, стоїть навіть навмисно розробити додаток, який буде гібрид двох технологій.

Порівняння з Ruby on Rails

Rails став стандартом, з яким порівнюються інші веб платформи. Для розробників та компаній, які живуть у світі Microsoft .NET, набагато легше прийняти та навчитися ASP.NET MVC, в той час як розробники та компанії, які працюють з Python або Ruby на Linux або Mac OS X знайдуть більш легкий шлях до Rails. Малоймовірно, що ви захочете перейти від Rails на ASP.NET MVC або навпаки. Між цими двома технологіями існують деякі реальні відмінності.

Rails є цілісна платформа для розробки, що позначає, що віна працює з повним стеком, починаючи від управління базою даних, через ORM, до обробки запитів з допомогою контролерів та дій та з вбудованими засобами автоматизованого тестування.

ASP.NET MVC Framework фокусується на обробці веб запитів у MVC патерні за допомогою контролерів та дій. Він не має вбудованого ORM інструменту, вбудованих засобів автоматизованого тестування або системою керування перенесенням баз даних. Це тому що що у .NET платформи вже є величезний вибір

можливостей для виконання цих функцій, та ви можете використати будь-яку з них. Наприклад, якщо ви шукаєте інструмент ORM, ви можете використовувати NHibernate, Subsonic, Microsoft Entity Framework або один з багатьох інших зрілих існуючих рішень. Така розкіш .NET платформи, хоча це не означає, що дані компоненти не так тісно інтегровані в ASP.NET MVC, як еквіваленти в Rails.

Порівняння з MonoRail

MonoRail є більш ранньою платформою для веб-додатків .NET MVC, створеною в рамках проекту з відкритим кодом Castle, і ця платформа розвивається з 2003 року. У багатьох Відношеннях MonoRail виступав як прототип для ASP.NET MVC. MonoRail продемонстрували, як подібна Rails MVC архітектура може бути побудована для ASP.NET та надав моделі, методи та термінологію, що використовуються в реалізації Microsoft.

Ми не бачимо у MonoRail серйозного конкурента. Це, мабуть, найпопулярніша. NET платформа для веб-додатків, створена за межами Redmond, і свого часу вона отримала широке розповсюдження. Тим не менш, з моменту запуску ASP.NET MVC, про проект MonoRail чути нечасто. Імпульс ентузіазму та новаторства у світі .NET веб розробки в даний час час зосереджено на ASP.NET MVC.

3.7 MVC патерн

Хороше розуміння того, що лежить в основі MVC, може допомогти вам розглянути всі можливості фреймворку.

Історія MVC

Термін *model-view-controller* використовувався з кінця 1970-х і виник із проекту Smalltalk у Xerox PARC, де він був задуманий як спосіб організації деяких ранніх додатків GUI. Деякі дрібні функції вихідного **MVC** патерну були прив'язані до конкретних понять Smalltalk, таким як екрани та інструменти (*screens i tools*), але ширші поняття як і раніше застосовні до додатків, і вони особливо добре підходять для веб-додатків.

Взаємодія з **MVC** додатком слідує за природним циклом дій користувача та оновлень уявлень, де уявлення вважається таким, що не зберігає стан (*stateless*). Це добре узгоджується з HTTP запитами та відповідями, що лежать в основі веб-додатків.

Крім того, **MVC** змушує *розділяти поняття*: доменна модель та логіка контролера відокремлені від інтерфейсу користувача. У веб-додатках це означає, що HTML зберігається окремо від решти програми, що робить технічну підтримку та тестування простіше та легше. Ruby on Rails привів до відновлення інтересу до **MVC**, і він залишається зразковою «дитиною» **MVC**. З того часу з'явилося багато інших **MVC** фреймворків, які продемонстрували переваги **MVC**, зокрема, звичайно, ASP.NET **MVC**.

За великим рахунком, MVC патерн позначає, що MVC додаток буде розділено як мінімум на три частини:

- *Моделі*, які містять або подають дані, з якими працюють користувачі. Це можуть бути прості моделі уявлення, які лише представляють дані, передані від контролера уявленню, або можуть бути доменними моделями, які містять дані домену, а також операції, перетворення та правила роботи з цими даними.

- *Представлення*, які використовуються для того, щоб обробити деякі частини моделі як інтерфейс користувача.
- *Контролери*, які обробляють запити, виконують операції для моделі і вибирають уявлення для показу користувачеві.

Моделі є визначенням всесвіту, в якому працює ваш додаток. У банківському додаток, наприклад, модель являє собою все в банку, що підтримує додаток: то є рахунки, головну книгу та кредитні ліміти для клієнтів, а також операції, які можуть бути використані для маніпулювання даними в моделі, такі як депонування засобів та здійснення вилучення з рахунків. Модель також відповідає за збереження загального стану та узгодженості даних, наприклад, вона гарантує, що всі угоди будуть додані у книгу, і що клієнт не зможе зняти більше грошей, ніж він може, або більше грошей, ніж має банк.

Моделі також визначаються тим, за що вони не відповідають: моделі не займаються UI і обробкою запитів – це обов'язки уявлень та контролерів. Представлення містять логіку, необхідну для відображення елементів моделі користувача, і більше нічого. Вони не мають прямого розуміння моделі та жодним чином безпосередньо не повідомляються з моделлю.

Контролери є мостом між уявленнями та моделлю: запити надходять від клієнта та обслуговуються контролером, який вибирає відповідне подання для показу користувачеві та, при необхідності, відповідні дії, які потрібно виконати з моделлю.

Кожен елемент MVC архітектури є чітко визначеним та автономним – це називається поділом понять. Логіка, яка маніпулює даними в моделі, міститься лише в моделі; логіка, яка відображає дані, знаходиться лише у поданні, а код, який обробляє запити користувачів та вступні дані, міститься тільки в контролері. З чітким поділом обов'язків між кожною з частин ваша програма буде легшою підтримувати і розширювати, незалежно від того, наскільки більшим воно ставатиме.

3.8 Доменна модель

Найбільш важливою частиною програми MVC є доменна модель. Ми створюємо моделі шляхом виявлення реальних осіб, операцій та правил, що існують у галузі чи діяльності. Вони повинні підтримуватись нашим додатком, і вони відомі як домен.

Потім ми створюємо програмне представлення домену – *доменну модель*. Для наших цілей доменна модель є набір типів **C#** (класи, структури і т.д.), відомих під загальною назвою *доменні типи*. Операції з домену представлені методами, визначеними в доменних типах, а доменні правила виражаються в логіці всередині цих методів, у застосуванні **C#** атрибутів до методів. Коли ми створюємо екземпляр доменного типу, щоб надати певний фрагмент даних, ми створюємо *доменний об'єкт*. Доменні моделі, як правило, постійні - є багато різних способів досягнення цього, але реляційні бази даних залишаються найпоширенішим вибором

Доменна модель є окремим авторитетним визначенням бізнес даних та процесів всередині вашої програми. *Постійна доменна модель* також є значним визначенням стану доменного уявлення.

Порада

*Поширений спосіб забезпечення поділу доменної моделі та решти частини **ASP.NET MVC** програми полягає в тому, щоб помістити модель у окреме складання **C#**. Таким чином, ви можете створити посилання на доменну моделі з інших частин програми, але потрібно переконатися, що у зворотному напрямі посилань немає. Це особливо корисно у великомасштабних проектах.*

3.9 ASP.NET реалізація MVC

У MVC контролери є C# класами, як правило, похідними від класу *System.Web.Mvc.Controller*. Кожен відкритий (*public*) метод у класі, похідний від *Controller* називається **методом дії**, який пов'язаний з налаштовуваним URL через систему маршрутизації (роутинг) ASP.NET. При надсиланні запиту на URL, пов'язаний з методом дії, виконуються вирази в класі контролера для того, щоб виконати деякі операції над доменною моделлю, а потім вибрати подання для відображення клієнта. На рис. 3.1 показано взаємодію між контролером, моделлю та представленням.

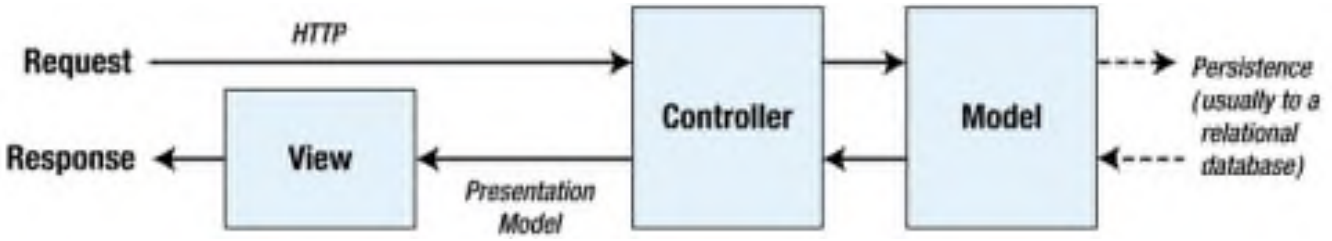


Рисунок 3.1 - Взаємодія основних компонентів у програмі MVC

ASP.NET MVC Framework забезпечує підтримку вибору движків представлення. Більш ранні версії MVC використовували стандартний ASP.NET движок представлення, який обробляв ASPX сторінки за допомогою версії синтаксису розмітки Web Forms. MVC 3 увів движок представлення Razor, який був вдосконалений в MVC 4 і який використовує повністю інший синтаксис. Visual Studio забезпечує підтримку *IntelliSense* для обох движків представлення, що дуже спрощує роботу з даними представленями, надісланими контролером.

ASP.NET MVC не застосовує жодних обмежень на реалізацію вашої доменної моделі. Ви можете створити модель за допомогою звичайних C# об'єктів та здійснювати зберігання за допомогою будь-який з баз даних, об'єктно-реляційних фреймворків чи інших інструментів зберігання даних, що підтримуються .NET. Visual Studio створює папку */Models* у шаблоні MVC проекту. Це підходить для простих проектів, але в більш складних програмах, як правило, доменні моделі визначаються в окремому проекті Visual Studio.

Порівняння MVC з іншими патернами

MVC - це не єдиний архітектурний патерн програмного забезпечення, звичайно, є багато інших, і деякі з них (або, принаймні, були) надзвичайно популярні. Ми можемо багато дізнатися про MVC, розглянувши інші патерни. У наступних розділах коротко опишемо різні підходи до структурування програми та зіставимо їх з MVC. Деякі з патернів є близькими варіаціями на тему MVC, тоді як інші абсолютно відрізняються.

Не варто стверджувати, що MVC є ідеальним патерном для всіх ситуацій. Бувають випадки, що деякі конкуруючі патерни так само корисні або навіть краще, ніж MVC. Рекомендовано робити обґрунтований та усвідомлений вибір при вибір патерну.

Паттерн Smart UI

Один з найпоширеніших патернів відомий як **smart UI**. Більшість програмістів у якийсь момент своєї кар'єри створювали **smart UI** програми.

Щоб створити smart UI додаток, розробники вибудовують інтерфейс користувача, зазвичай шляхом перетягування набору компонентів або елементів керування на дизайнерську поверхню. Елементи управління повідомляють про взаємодію з користувачем, представляючи події для натискання кнопок, натискання клавіш, рух миші і так далі. Розробник додає код у відповідь на ці події у серії обробників подій – невеликих блоків коду, які викликаються коли відбувається певна подія для певного компонента. І Тут ми закінчуємо з монолітним додатком, як показано рис. 3.2, 3.3 – код, який обробляє інтерфейс користувача і логіка змішуються разом, поняття взагалі не розділені. Код, який визначає допустимі значення для даних, запитів даних або змінює обліковий запис користувача, ділиться на маленькі шматочки, з'єднані разом по порядку, у якому очікуються події.

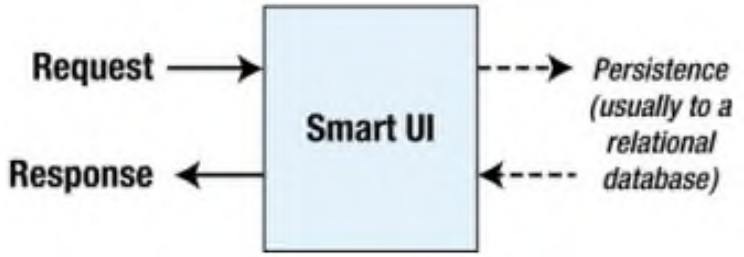


Рисунок 3.2 - Smart UI патерн

Найбільшим недоліком цієї конструкції є те, що її важко підтримувати та розширювати: змішування доменної моделі та коду бізнес логіки з кодом користувача інтерфейсу призводить до дублювання, де той же фрагмент бізнес логіки копіюється і вставляється для підтримки новоствореного компонента. Пошук усіх дублюючих частин та внесення правок може бути важким, а в складному smart UI додатку іноді майже неможливо додати нову функцію, не ламаючи існуючу. Тестування smart UI додатків також може бути важким, єдиний спосіб - це імітації взаємодій з користувачем, яка далека від ідеалу і забезпечує основу для повного тестування.

У світі MVC Smart UI часто називають *анти-патерном*, якого слід уникати за всяку ціну. Ця антипатія виникла принаймні частково тому, що люди прийшли до MVC у пошуках альтернативи, провівши частину своєї кар'єри, намагаючись розвивати та підтримувати Smart UI програми. Не все гладко в smart UI патерні, але є і позитивні аспекти у такому підході. Smart UI програми швидко і легко розробляти: розробники інструментарію доклали багато зусиль для того, щоб розробка була приємною, та навіть самий недосвідчений програміст може створити щось професійне і досить функціональне протягом кількох годин.

Найбільша слабка сторона Smart UI додатків – супровід та підтримка – яка не спостерігається при розробці невеликих додатків. Якщо ви створюєте простий інструмент для невеликого аудиторії, то Smart UI програма може бути ідеальним рішенням. Додаткова складність MVC програм просто не виправдана.

Нарешті, Smart UI ідеально підходить для створення прототипів інтерфейсу користувача, ці інструменти конструювання справді хороші. Якщо ви сидите з клієнтом та хочете зрозуміти вимоги до інтерфейсу, інструменти Smart UI можуть бути швидкими та чуйними способом створення та тестування різних ідей.

3.10 Архітектура Model-View

Область, в якій зазвичай виникають проблеми з підтримкою в Smart UI додатках - це бізнес-логіка і внесення змін може стати дуже неприємним процесом. Поліпшення в цій галузі пропонує архітектура *model-view* (модель-представлення), яка витягує бізнес-логіку в окрему доменну модель При цьому дані, процеси та правила всі зосереджені в одній частині програми, як показано на рис. 3.

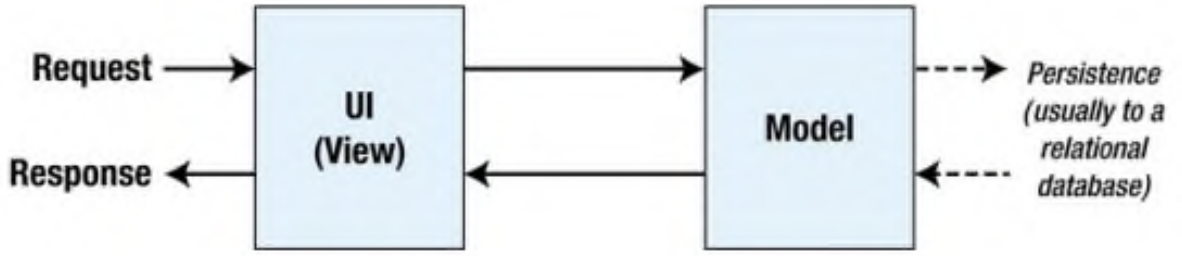


Рисунок 3.3 – model-view патерн

Архітектура model-view є значним покращенням у порівнянні зі Smart UI - наприклад, її набагато легше підтримувати. Проте виникають дві проблеми:

Перше, оскільки інтерфейс користувача і доменна модель настільки тісно інтегровані, утруднюється виконання юніт тестування. Друга проблема випливає з практики, а не з визначення патерну. Ця модель зазвичай містить масу коду доступу до даних, а це означає, що модель даних містить не тільки бізнес дані, операції та правила.

Друга проблема випливає з практики, а не з визначення патерну. Ця модель зазвичай містить масу коду доступу до даних, а це означає, що модель даних містить не тільки бізнес дані, операції та правила.

Класична трирівнева (three-tier) архітектура

Для вирішення проблеми з архітектурою model-view трирівнева або тришарова (three-tier або three-layer) модель відокремлює код від доменної моделі і поміщає його в новий компонент під назвою DAL. Це показано рис. 3.4.

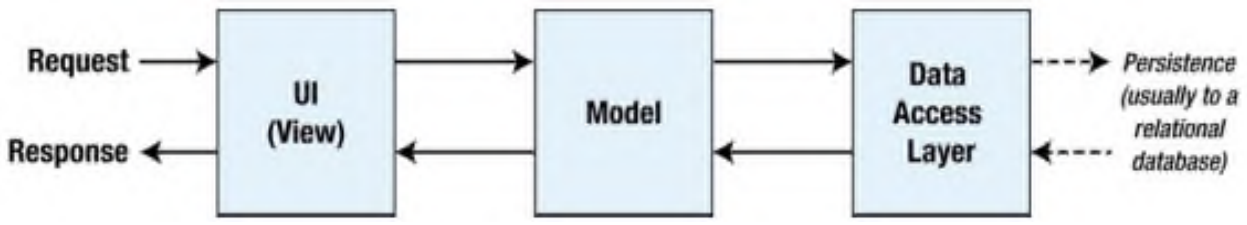


Рисунок 3.4 - Трирівневий патерн

Це великий крок уперед. Трирівнева архітектура є найбільш широко використовуваним патерном для бізнес-додатків. Вона не має обмежень щодо того, як реалізований користувальницький інтерфейс, і забезпечує хороший поділ понять, не будучи занадто складною. Також DAL може бути створений таким чином, що юніт тестування буде щодо легким. Можна помітити очевидну подібність між класичним трирівневим додатком та MVC патерном. Різниця полягає в тому, що оскільки UI шар безпосередньо з'єднаний з "click-and-event" GUI фреймворком (таким як Windows Forms та ASP.NET Web Forms), стає майже неможливим виконання автоматизованих юніт тестів. І оскільки UI частина трирівневої програми може бути дуже складною, ми маємо багато коду, який не може бути ретельно протестований.

У гіршому випадку, відсутність у трирівневій моделі управління над шаром UI означає, що багато таких програм в кінцевому підсумку виглядають як погано замасковані Smart UI програми, без реального поділу понять. І в результаті ми отримуємо нетестоване, не підтримуваний додаток, який також є дуже складним.

Варіації MVC

Ми ознайомилися з основними принципами створення MVC-додатків, і особливо те, як вони відносяться до реалізації ASP.NET MVC. Інші інтерпретують аспекти патерну по-іншому та коректують чи іншим чином адаптують MVC для своїх проектів. У наступних розділах ми представимо короткий огляд двох найпоширеніших варіантів на тему MVC. Розуміння цих варіантів не є необхідним для роботи з ASP.NET MVC. Ми увімкнули цю інформацію для повноти.

Model-View-Presenter паттерн

MVP є варіацією MVC, який розроблений, щоб відповідати GUI платформам, що зберігає стан (stateful), таким як Windows Forms або ASP.NET Web Forms, і це стоїть спроба представити кращі аспекти Smart UI без тих проблем, які він зазвичай приносить.

У цьому патерні презентер (*presenter*) має ті ж обов'язки, що і MVC контролер, але він також має більш безпосереднє відношення до уявлень, безпосередньо керуючи значеннями, що відображаються в UI компонентах, залежно від дій та даних, введених користувачем. Існують дві реалізації цього патерну:

- Реалізація *пасивного уявлення*, в якому уявлення не містить логіки – це контейнер для елементів керування UI, які безпосередньо управляються презентером.
- Реалізація *наглядуючого контролера*, в якому уявлення може бути відповідальним за деякі елементи логіки уявлення, такі як пов'язані дані, і джерела даних можна посилатися з доменної моделі.

Різниця між цими двома підходами стосується того, наскільки розумним є уявлення. У будь-якому випадку, презентер відокремлюється від GUI фреймворку, що робить логіку презентера простою та підходить для модульного тестування.

Model-View-View Model патерн

MVVM патерн є найостаннішою варіацією MVC, рік виникнення - 2005 у команді Microsoft, яка розробляла технологію, що стала Windows Presentation Foundation (WPF) та Silverlight.

У MVVM патерні моделі та уявлення грають ті ж ролі, що й у MVC. Різниця полягає в MVVM концепція моделі подання, яка є абстрактним поданням інтерфейсу користувача. Як правило, це C# клас, який розкриває обидві властивості для даних, які будуть відображатися в інтерфейсі, та операції за даними, які можуть бути викликані з інтерфейсу користувача. На відміну від MVC контролера, модель представлення MVVM не має поняття, що існує уявлення (або будь-яка конкретна технологія UI). MVVM представлення використовує зв'язувальну функцію WPF/Silverlight для двонаправленого зв'язування властивостей, що надаються елементами управління в поданні (пункти в меню або результат натискання кнопки), з властивостями, пропонованими моделлю уявлення.

MVVM тісно співпрацює зі зв'язуючими функціями WPF і тому це не той патерн, який легко застосувати і до інших платформ.

Порада

***MVC** також використовує термін модель подання, але він відноситься до простому класу моделі, який використовуються тільки для передачі даних з контролера у виставу. Ми маємо чітко розрізняти модель уявлення та доменну модель, яка є складним уявленням даних, операцій та правил.*

ЛЕКЦІЯ 4 ПРОБЛЕМНО-ОРІЄНТОВНЕ ПРОГРАМУВАННЯ (DDD).ПОБУДОВА СЛАБОЗВ'ЯЗАНИХ КОМПОНЕНТІВ

ASP.NET MVC не диктує того, яка технологія має використовуватись для доменної моделі. Ви вільні вибрати будь-яку технологію, яка буде взаємодіяти з .NET Framework, та тут є багато варіантів. Тим не менш, ASP.NET MVC пропонує інфраструктуру та угоди, щоб допомогти підключити класи доменної моделі до контролерів та уявлень, а також до самого MVC Framework. Є три ключові функціональні можливості:

- *Зв'язування даних моделі* є функцією, яка автоматично заповнює об'єкти моделі, використовуючи вхідні дані, як правило, надіслані з HTML форми.
- *Метадані моделі* дозволяють описати фреймворку зміст класів моделі. Наприклад, ви можете надати читабельний опис їх властивостей або підказати про те, як вони повинні відображатись. MVC Framework може автоматично представити зображення або редактор UI для класів моделі в уявленнях.
- *Валідація*, яка виконується під час зв'язування даних та застосовує правила, які можуть бути визначені як метадані.

Ми коротко торкнулися зв'язування даних та валідації, коли ми будували наше перше MVC. На даний момент, ми збираємося відкласти реалізацію ASP.NET MVC убік і подумати про доменному моделюванні як самостійної діяльності. Ми збираємося створити просту доменну модель за допомогою .NET і SQL Server, використовуючи деякі основні технологічні прийоми зі світу DDD.

4.1 Побудова доменної моделі

У вас, напевно, вже був досвід мозкового штурму створення доменної моделі. Зазвичай сюди включені розробники, бізнес експерти, велика кількість кави, печива та ручки з маркерами. Через деякий час люди в кімнаті приходять до початкового спільного знаменника, і тоді з'являється перший проект доменної моделі. (Опустимо розповідь про багатогодинні розбіжності. Досить сказати, що розробники будуть витрачати перші години, вимагаючи від бізнесу експертів реальних завдань, а не взятих з наукової фантастики, у той час як бізнес-експерти висловлюватимуть подив і стурбованість, що час та кошторис витрат аналогічні тим, що NASA вимагає досягти Марса. Кава має важливе значення у вирішенні таких протиріч і в такому протистоянні, зрештою, сечовий міхур у всіх буде настільки повним, що всі підуть на компроміс, і намітиться прогрес у вирішенні завдання).

Ви могли закінчувати чимось схожим, що зображено на рис.4.1 і що є відправною точкою для цього прикладу – проста доменна модель для аукціонного додатка.

Рисунок 4.1 - Перший начерк моделі для аукціонного додатка

Ця модель містить набір елементів *Members*, кожен із яких містить набір елементів *Bids*. Кожен *Bid* призначений для одного *Item*, а кожен *Item* може містити декілька *Bid* від різних *Members*.

Повсюдно використовувана мова

Ключовою перевагою реалізації вашої доменної моделі як окремого компонента є те, що ви можете встановити на свій вибір мову і термінологію. Вам варто спробувати знайти термінологію для її об'єктів, операцій та відносин, які будуть зрозумілі не лише розробникам, а й бізнес-експертам. Ми рекомендуємо вам адаптувати доменну термінологію, коли модель вже існує. Наприклад, якщо те, що розробник буде називати користувачами та ролями (*users* та *roles*), відомо в домені як агенти та дозволу (*agents* і *clearances*), то ми рекомендуємо вам прийняти останній варіант у вашому доменної моделі.

При моделюванні таких понять, для яких експерти не мають потрібної термінології, вам потрібно дійти спільної угоди про те, як ви будете їх називати, створюючи повсюдно використовується мова, яка проходитиме через доменну модель.

Розробники, як правило, говорять мовою коду - іменами класів, назвами таблиць баз даних і таке інше. Бізнес експерти не розуміють ці терміни, та вони й не повинні. Бізнес експерт із невеликими технічними знаннями – це небезпечна річ, бо він буде постійно фільтрувати вимоги через розуміння того, на що здатна технологія. І якщо так станеться, ви не отримаєте істинного розуміння того, що потрібне бізнесу.

Цей підхід також допомагає уникнути у додатку надузагальнення. Програмісти, як правило, часто хочуть змоделювати всі можливі реалії бізнесу, а не саме те, що потребує бізнес. У моделі аукціону таке могло б статися, якби ми змінили *members* та *Items* загальними поняттями *resources* і *relationships*. Коли ми створюємо доменну модель, яка не зовсім відповідає моделюваному домену, ми втрачаємо можливість отримати будь-яку реальну картину бізнес-процесу і, в майбутньому, уявлення бізнес-процесу може виявитися неправильним. Ці обмеження не є забороною, вони є маяком, який направляє вашу роботу у правильне русло.

Зв'язок між мовою, що повсюдно використовується, і доменною моделлю не повинен бути поверхневий: DDD експерти припускають, що будь-які зміни в повсюдно використовуваному мові повинні призводити до зміни моделі. Якщо ви дозволите моделі не бути синхронізовані з доменом, ви створите проміжну мову, яка буде відрізнятися у моделі та домену, і це може надалі призвести до серйозних проблем. Ви створите спеціальний клас людей, які можуть говорити обома мовами, і тоді вони почнуть фільтрувати вимоги, ґрунтуючись на своєму неповному розумінні обох мов.

4.2 Агрегати та спрощення

На рис. 4.1 представлена хороша відправна точка для нашої доменної моделі, але тут немає ніяких корисних вказівок щодо реалізації моделі з використанням C# та SQL Server. Якщо ми завантажимо в пам'ять елемент *Member*, чи ми також повинні завантажити *Bids* і *Items*, пов'язані з ним? І якщо це так, чи потрібно нам

завантажити всі інші *Bids* для цих *Items*, а також *Members*, які пов'язані з цими *Bids*? Коли ми видаляємо об'єкт, чи повинні ми видалити пов'язані з ним об'єкти, і якщо так, то які? Якщо ми виберемо для реалізації сховище документів замість реляційної бази даних, яка колекція об'єктів буде єдиним документом? Ми цього не знаємо, а наша доменна модель не дає жодних відповідей на ці запитання.

У цьому випадку DDD пропонує організувати доменні об'єкти в групи, які називаються агрегатами. На рис. 4.2 показано, як ми можемо поєднати об'єкти в нашій доменній аукціон моделі.

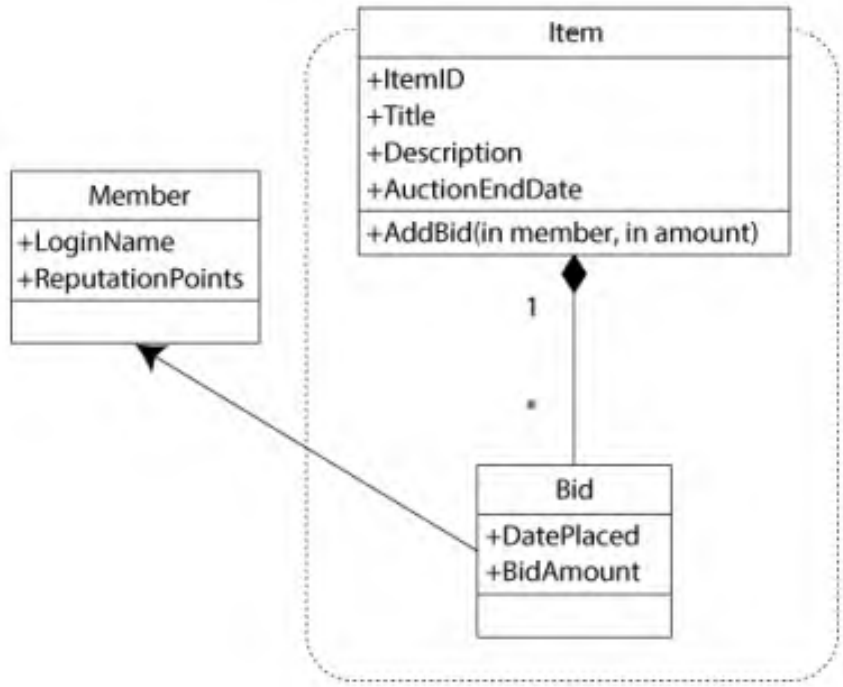


Рисунок 4.2 - Аукціонна доменна модель із агрегатами

Суть агрегату полягає у групуванні кількох об'єктів доменної моделі: є ключова сутність (*root entity*), яка використовується для визначення цільного агрегату, і вона діє як «бос» для валідації та збереження операцій. Агрегат розглядається як єдине ціле з точки зору зміни даних, тому ми повинні створити агрегати, які представляють собою відносини, що мають сенс у контексті доменної моделі, та створити операції, які логічно відповідають реальним бізнес-процесам: іншими словами, ми повинні створювати агрегати, групуючи об'єкти, що змінюються як група.

Ключове правило **DDD** говорить, що об'єкти за межами конкретного екземпляра агрегату можуть бути пов'язані тільки з root entity, а не з будь-яким іншим об'єктом усередині агрегату (насправді, ідентичність не кореневого об'єкта має бути унікальною тільки в межах свого агрегату). Це правило підтверджує уявлення про роботу з об'єктами всередині агрегату як із єдиним. цілим.

У нашому прикладі *Members* та *Items* є ключовими сутностями агрегату, тоді як Bids можуть бути доступні тільки в контексті елемента *Item*, який є кореневим у своєму агрегаті. Елементи *Bids* можуть бути пов'язані з елементами *Members* (які є основними сутностями, root entity), але *Members* не можуть безпосередньо посилатися на *Bids* (оскільки вони не є кореневими).

Одна з переваг агрегатів полягає в тому, що вони спрощують систему відносин між об'єктами в доменній моделі, і часто вони можуть дати додаткове розуміння природи домену, який моделюється. По суті, створення агрегатів обмежує відносини між об'єктами доменної моделі, так що вони стають більш схожими на відносини, які існують у реальному домені. У лістингу 1 показано, як виглядає наша доменна модель, виражена C#.

Листинг 1: Аукційна домена модель, виразу в C#

```
public class Member
{
        public string LoginName { get; set; } // Уникальный ключ
        public int ReputationPoints { get; set; }
}
public class Item
{
        public int ItemID { get; private set; } // Уникальный ключ
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime AuctionEndDate { get; set; }
        public IList<Bid> Bids { get; set; }
}
public class Bid
{
        public Member Member { get; set; }
        public DateTime DatePlaced { get; set; }
        public decimal BidAmount { get; set; }
}
```

Зверніть увагу, як ми легко змогли зрозуміти односпрямований характер відносин між *Bids* та *Members*. Ми також змогли змоделювати деякі інші обмеження, наприклад, елементи *Bids* залишаються незмінними (це являє собою загальну стратегію аукціону, коли ставки не можуть бути змінені, як тільки вони зроблено). Застосування агрегації дозволило нам створити більш корисну та точну доменну модель, яку ми змогли з легкістю уявити у C#.

Загалом, агрегати додають структуру та точність доменної моделі. Вони спрощують застосування валідації (ключова сутність стає відповідальною за валідацію стану всіх об'єктів у агрегаті), і очевидні юніти, які потрібно зберігати. І оскільки агрегати, по суті, є атомарними одиницями нашої доменної моделі, вони також є відповідними одиницями для керування транзакціями та каскадного видалення з бази даних.

З іншого боку, вони накладають обмеження, які іноді можуть з'явитися штучними, бо найчастіше вони є штучні. Агрегати зазвичай з'являються в бази даних документа, але вони не є рідним поняттям в SQL Server і в більшості

інструментів ORM. Тому якщо ви бажаєте їх добре реалізувати, вашій команді знадобиться дисципліна та ефективний обмін інформацією.

4.3 Визначення репозиторіїв

Якоїсь миті нам потрібно буде забезпечити сталість нашої доменної моделі: це, як правило, здійснюється за допомогою реляційних, об'єктних чи документальних баз даних. Постійність не є частиною нашої доменної моделі – це самостійне або ортогональне поняття у нашому принципі поділу понять. Це означає, що ми не хочемо змішувати код, що обробляє сталість, з кодом, який визначає доменну модель.

Звичайним способом забезпечити поділ між доменною моделлю та системою сталості є визначення *репозиторіїв* - це об'єкти представлення основної бази даних (або сховища файлів, які ви обрали). Замість того, щоб працювати безпосередньо з базою даних, доменна модель викликає методи, визначені в репозиторії, який у свою черга робить запити до бази даних для зберігання та вилучення даних моделі. Це дозволяє ізолювати модель від сталості.

Угода полягає у визначенні окремої моделі даних для кожного агрегату, тому що агрегати є природними одиницями задля збереження сталості. У разі нашого укціону, наприклад, ми можемо створити два репозиторії: один для *Members* і один для *Items* (зверніть увагу, що нам не потрібен репозиторій для *Bids*, тому що вони будуть збережені як частина агрегату *Items*). У лістингу 2 показано, як можна визначити ці репозиторії.

Лістинг 2: C# класи репозиторіїв для доменних класів *Member* та *Item*

```
public class MembersRepository
{
        public void AddMember(Member member)
        {
        /* Реалізуй мене */
        }
        public Member FetchByLoginName(string loginName)
        {
        /* Реалізуй мене */
        return null;
        }
        public void SubmitChanges()
        {
        /* Реалізуй мене */
        }
}
public class ItemsRepository
{
        public void AddItem(Item item)
        {
        /* Реалізуй мене */
```

```
        }
        public Item FetchByID(int itemID)
        {
        /* Реалізуй мене */
        return null;
        }
        public IList<Item> ListItems(int pageSize, int pageIndex)
        {
        /* Реалізуй мене */
        return null;
        }
        public void SubmitChanges()
        {
        /* Реалізуй мене */
        }
}
```

Зверніть увагу, що репозиторії стосуються лише завантаження та збереження даних: вони не містять доменну логіку взагалі. Ми можемо завершити класи репозиторіїв шляхом додавання виразів для кожного методу, який виконує операції зі збереження та отримання для відповідного механізму збереження.

4.4 Побудова слабозв'язаних компонентів

Як ми вже говорили, одна з найважливіших особливостей шаблону MVC полягає в тому, що дозволяє розділяти поняття. Ми хочемо, щоб компоненти у нашому додатку були настільки незалежними, наскільки це можливо, і були такі взаємопов'язані, щоб ми могли ними керувати.

У нашій ідеальній ситуації кожен компонент нічого не знає про інші компоненти та працює з іншими областями програми через абстрактні інтерфейси. Це відомо як слабкий зв'язок компонентів, і вона спрощує тестування та можливість змін нашого додатку.

Простий приклад допоможе все це зрозуміти. Якщо ми пишемо компонент *MyEmailSender*, який буде надсилати електронну пошту, ми хотіли б реалізувати інтерфейс, який визначає всі відкриті функції, необхідні для надсилання електронної пошти, яку ми б назвали *IEmailSender*.

Будь-який інший компонент нашої програми, якому потрібно надіслати e-mail скажімо, наприклад, допоміжний метод скидання пароля *PasswordResetHelper*, може надіслати електронну пошту лише посилаючись на методи інтерфейсу. Як показано на рис. 4.3, між *PasswordResetHelper* та *MyEmailSender* не існує прямої залежності.

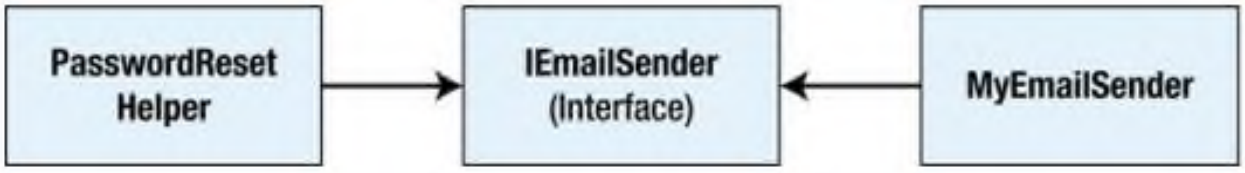


Рисунок 4.3 - Використання інтерфейсів для розділення компонентів

Вводячи *IEmailSender*, ми гарантуємо, що не існує прямої залежності між *PasswordResetHelp* та *MyEmailSender*. Ми могли б замінити *MyEmailSender* іншим компонентом для відправки електронної пошти або навіть використовувати для тестування mock-об'єкти.

Примітка

Не всі відносини мають бути розділені за допомогою інтерфейсу. Рішення про це ґрунтується на наступних тезах: наскільки складним є додаток, який вид тестування потрібно і яка можлива довгострокова підтримка. Наприклад, в невеликому та простому ***ASP.NET MVC*** *додатку ми могли б не розділяти контролери від доменної моделі.*

Використання впровадження залежності (Dependency Injection)

Інтерфейси допомагають нам розділяти компоненти, але у нас, як і раніше, є проблема: C# не надає вбудований спосіб для легкого створення об'єктів, що реалізують інтерфейси, виключення створення інтерфейсів конкретного компонента. Зрештою, у нас є код, показаний у лістингу 1.

Лістинг 1: Створення екземпляра конкретного класу для реалізації інтерфейсу

```
public class PasswordResetHelper {
        public void ResetPassword() {
                IEmailSender mySender = new MyEmailSender();
                //виклик методу інтерфейса для конфігурації інформації по e-mail
        mySender.SendEmail(); }}
```

Ми лише частково на шляху до слабко зв'язаних компонентів: клас *PasswordResetHelper* налаштовує та надсилає електронну пошту через інтерфейс *IEmailSender*, але для створення об'єкта, який реалізує цей інтерфейс, повинен був створити екземпляр *MyEmailSender*.

Ми зробили ще гірше, тепер *PasswordResetHelper* залежить від *IEmailSender* та *MyEmailSender*, як показано рис. 4.4.

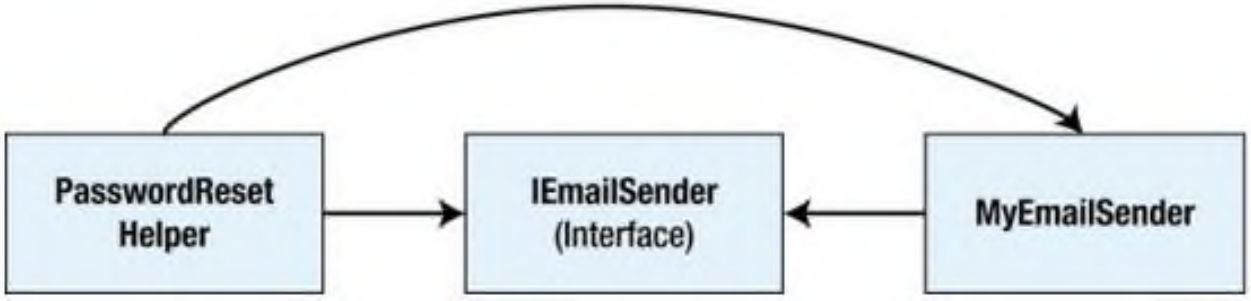


Рисунок 4.4 - Компоненти, які все одно тісно пов'язані

Що нам потрібно, так це спосіб отримати об'єкти, які реалізують цей інтерфейс без необхідності створення об'єкта безпосередньо. Вирішення цієї

проблеми називається *впровадженням залежності* (Dependency injection, DI), відоме також як *інверсія управління* (Inversion of Control, IoC).

DI є патерном, що завершує слабкий зв'язок компонентів, який ми почали з додавання інтерфейсу IEmailSender у наш простий приклад. Ось ми описуємо DI, і ви могли б поставити питання, до чого ця метушня, але зрозумійте нас правильно: це важлива концепція, яка займає ключову позицію у ефективній MVC розробці.

Є два варіанти DI патерну. Перший полягає в тому, що ми видаляємо всі залежності від конкретні класи з наших компонентів, в даному випадку PasswordResetHelper. Це робиться шляхом передачі реалізації необхідних інтерфейсів у конструктор класу, як показано в лістингу 2.

Лістинг 2: Видалення залежностей із класу *PasswordResetHelper*

```
public class PasswordResetHelper
{
        private IEmailSender emailSender;
        public PasswordResetHelper(IEmailSender emailSenderParam)
        {
                emailSender = emailSenderParam;
        }
        public void ResetPassword()
        {
                // виклик методу інтерфейса для конфігурації інформації по e-mail
                emailSender.SendEmail();
        }
}
```

Ми розірвали залежність між *PasswordResetHelper* та *MyEmailSender*: конструктор *PasswordResetHelper* вимагає об'єкт, який реалізує інтерфейс *IEmailSender*, але він не знає, або його не хвилює, що це за об'єкт і він більше не відповідає за його створення.

Залежності впроваджуються в *PasswordResetHelper* під час виконання, тобто буде створено екземпляр класу, що реалізує інтерфейс *IEmailSender*, та переданий конструктору *PasswordResetHelper* під час створення екземпляра. Тут немає залежності від часу компіляції між *PasswordResetHelper* і будь-яким класом, який реалізує інтерфейси від яких він залежить.

Примітка

Клас ***PasswordResetHelper*** *вимагає, щоб його залежності були впроваджено за допомогою його конструктора – це відомо як* ***constructor injection*** *(впровадження через конструктор). Ми могли б також зробити так, щоб залежності були впроваджені через відкриті властивості, що відоме як* ***setter injection*** *(впровадження через властивості).*

Оскільки із залежностями працюють під час виконання, ми можемо вирішити, які реалізації інтерфейсу будуть використовуватися при запуску програми: ми можемо вибирати між різними компонентами надсилання електронної пошти або вставляти mock-об'єкти для тестування.

4.5 DI приклад з MVC

Повернімося до аукціонної доменної моделі, яку ми створили раніше, і застосуємо до неї DI. Метою є створення класу контролера, ми назвемо його *AdminController*. Він використовує репозиторій *MembersRepository*, але без безпосереднього зв'язку *AdminController* та *MembersRepository*. Ми почнемо з визначення інтерфейсу, який розділить наші два класи - ми назвемо його *IMembersRepository* – і змінимо клас *MembersRepository* для реалізації інтерфейсу, як показано у лістингу 3.

Лістинг 3:- Інтерфейс *IMembersRepository*

```
public interface IMembersRepository
{
        void AddMember(Member member);
        Member FetchByLoginName(string loginName);
        void SubmitChanges();
}
public class MembersRepository : IMembersRepository
{
        public void AddMember(Member member)
        {
                /* Реалізуй мене */
        }
        public Member FetchByLoginName(string loginName)
        {
                /* Реалізуй мене */
        }
        public void SubmitChanges()
        {
                /* Реалізуй мене */
        }
}
```

Тепер ми можемо написати клас контролера, який залежить від інтерфейсу *IMembersRepository*, як показано в лістингу 4.

Лістинг 4: Клас *AdminController*

```
public class AdminController : Controller
{
        IMembersRepository membersRepository;
```

```
        public AdminController(IMembersRepository repositoryParam)
        {
            membersRepository = repositoryParam;
        }
        public ActionResult ChangeLoginName(string oldLoginParam, string
newLoginParam)
        {
            Member member =
        membersRepository.FetchByLoginName(oldLoginParam);
            member.LoginName = newLoginParam;
        membersRepository.SubmitChanges();
            // будемо показувати представлення
 return View();
        }
}
```

Клас *AdminController* вимагає реалізації інтерфейсу *IMembersRepository* як параметра конструктора: його буде впроваджено під час виконання, що дозволяє *AdminController* працювати на екземпляр класу, що реалізує інтерфейс, не будучи пов'язаним із цією реалізацією.

4.6 Використання контейнера застосування залежності

Ми вирішили питання із залежністю: ми збираємося впровадити наші залежності у конструктори наших класів під час виконання. Але нам все одно потрібно вирішити ще одне питання – як ми створимо екземпляр конкретної реалізації інтерфейсу без створення залежностей ще десь у нашому додатку?

Відповіддю є контейнер впровадження залежностей, також відомий як IoC контейнер. Це компонент, який діє як посередник між залежностями, які вимагає клас типу *PasswordResetHelper*, і конкретної реалізації цих залежностей, як *MyEmailSender*.

Ми реєструємо набір інтерфейсів чи абстрактних типів, які використовує наше додаток, з DI контейнером, і говоримо йому, для яких конкретно класів повинні бути створено екземпляри для задоволення залежностей. Таким чином, ми реєструємо інтерфейс *IEmailSender* з контейнером та вказуємо, що має бути створений екземпляр *MyEmailSender* щоразу, коли потрібна реалізація *IEmailSender*. Коли б нам не потрібен *IEmailSender*, наприклад, для створення екземпляра *PasswordResetHelper*, ми йдемо до DI контейнер, і нам дається реалізація класу, який ми зареєстрували як конкретної реалізації за промовчанням цього інтерфейсу – в даному випадку, *MyEmailSender*.

Нам не потрібно створювати DI контейнери самим: є кілька відмінних версій із відкритим вихідним кодом та вільною ліцензією. Один із контейнерів, який нам подобається, називається Ninject і ви можете отримати інформацію про нього на www.ninject.org.

Порада

__Microsoft__ створила свій власний __DI__ контейнер, який називається __Unity__. Однак ми збираємося використовувати __Ninject__, тому що нам він подобається і він демонструє здатність змішувати і поєднувати інструменти при використання MVC. Якщо ви хочете отримати більше інформації про __Unity__, відвідайте unity.codeplex.com.

Роль контейнера DI може здатися простою і тривіальною, але це не той випадок. Хороший DI контейнер, такий як Ninject, має деякі дуже розумні функції:

- **Ланцюжок залежностей**: Якщо ви запитуєте компонент, який має свої власні залежності (наприклад, параметри конструктора), контейнер також буде задовольняти ці залежності. Таким чином, якщо конструктор для класу *MyEmailSender* вимагає застосування інтерфейсу *INetworkTransport*, DI контейнер підтвердить реалізацію за умовчанням цього інтерфейсу, передасть його конструктору *MyEmailSender* і поверне результат у вигляді за промовчанням *IEmailSender*.
- **Управління життєвим циклом об'єкта**: Якщо ви запитуєте компонент більш ніж один раз, чи повинні ви щоразу отримувати той самий екземпляр чи новий? Хороший DI контейнер дозволить вам налаштувати життєвий цикл компонентів, дозволяючи вибрати один з визначених варіантів, включаючи *singleton* (той самий екземпляр щоразу), transient (новий екземпляр щоразу), *instance-per-thread, instance-per-HTTP-request, instance from-a-pool* та багато інших.
- **Конфігурація значень параметрів конструктора**: Наприклад, якщо конструктор для нашої реалізації інтерфейсу *INetworkTransport* вимагає рядка *serverName*, ви в стані встановити значення для неї у конфігурації DI контейнера. Це груба, але проста система конфігурації, яка видаляє будь-яку необхідність для вашого коду обходити рядки підключення, адреси серверів тощо.

Можливо, ви спробуєте написати власний DI контейнер. Ми вважаємо, що це великий експериментальний проект, якщо у вас є час, який ви можете вбити, і ви хочете дізнатися багато нового про C# і .NET відображення (reflection). Якщо ж ви хочете використовувати DI контейнер у MVC додатку для виробничих цілей, ми рекомендуємо вам скористатися одним з DI контейнерів, що зарекомендували себе, таким як *Ninject*.

ЛЕКЦІЯ 5 ПОНЯТТЯ ПРО МОВУ РОЗМІТКИ. НУМЕРОВАНІ ТА МАРКОВАНІ СПИСКИ

5.1 Базові конструкції мови HTML

Гіпертекст - це текст, у який вбудовані спеціальні коди, що задають форматування тексту, наявність у ньому ілюстрацій, мультимедійних вставок та гіперпосилань.

HTML (Hyper Text Markup Language - мова гіпертекстової розмітки) – мова тегів, якою пишуться гіпертекстові документи для мережі Інтернет.

5.2 Поняття тегу

Вся інформація про форматування документу міститься в фрагментах розташованих між знаками < > — такий фрагмент називають ***тегом***.

Відкриваюча дужка **< ім'я тега, атрибут>** закриваюча дужка

Приклади тегів HTML:

<TITLE>, <BODY>, <TABLE>, </A>, <IMG>, </CENTER>.

Теги бувають:

- контейнерні (парні);
- поодинокі (одинарні).

Приклади **парних тегів**:

<HTML> </HTML>,

<B> </B>,

<HEAD></HEAD>, <H3></H3>, <ADDRESS></ADDRESS>, <LI></LI>.

Приклади **одинарних тегів**:

***
,***

<HR>,

<META>, <BASEFONT>,

<FRAME>, <INPUT>.

Відкривальні теги (вказують програмі на початок об'єкту) можуть містити атрибути, які впливають на ефект, створюваний тегом.

Атрибути — задають значення властивостей об'єкту, поміщеного у контейнер.

Приклад тегу з атрибутами:

*<**BODY** BGCOLOR="#000000">*

5.3 Структура HTML-документа

HTML-документ складається з ***основного тексту*** і ***тегів розмітки***.

HTML-документи містяться у файлах із розширенням ***.htm*** або ***.html***.

Структура:

<HTML>

<HEAD>

<TITLE> Заголовок документа ***</TITLE>***

</HEAD>

<BODY>

Текст, що відображається на екрані

</BODY>

</HTML>

<HTML> </HTML> **-** починає та закінчує документ
<HEAD> </HEAD> **-** починає та закінчує заголовок документу
<TITLE> </TITLE> **-** задає назву документу
<BODY></BODY> **-** починає та закінчує тіло документу

Заголовки

Для того щоб розбити текст на логічні частини, використовують ***заголовки.***
(Існує 6 рівнів заголовків документів).
Вони позначені тегами від
<H1>...</H1> до ***<H6>...</H6>.***

Абзаци та рядки

Для визначення звичайних ***абзаців*** у мові HTML використовують теги ***<P>*** і ***</P>***.

Теги <H> та <P> можуть додатково містити ***атрибут ALIGN,*** який додатково використовується для того, щоб вирівняти текст заголовку або абзацу по центру, за правим або лівим краєм.

Наприклад:
<H2 ALIGN=center>
<P ALIGN=right>.

Одинарні теги

**
** - одинарний тег, що використовується для переходу на новий рядок без створення абзацу

<HR> - одинарний тег, що використовується для створення горизонтальної лінії, яка візуально відділяє різні частини документа.

Кольори та ***зображення*** для всього документа і його тла задають за допомогою тегу ***<BODY>***. Він може мати такі атрибути:

• ***BACKGROUND="URL"*** — замість URL вказують адресу малюнка, який має бути тлом для сторінки;
• ***BGC0L0R=значення*** — задає колір, який має бути фоновим для документа;
• ***TEXT=значення*** — задає колір тексту;
• ***LINK=значення*** — визначає колір гіперпосилань у документі;
• ***ALINK=значення*** — задає колір гіперпосилань під час клацання;
• ***VLINK=значення*** — задає колір переглянутих гіперпосилань.

Гарнітуру, розмір та колір шрифту для фрагмента тексту задають за допомогою тегу

<FONT АТРИБУТИ> Текст </FONT>.

Наприклад:
 текст
…..
…..
…..
 текст
 текст - означає, що літери слова комп'ютер матимуть темно-синій колір.

5.4 Теги форматування символів тексту

Теги	*Ефект*	*Приклад*
<B> та </B>	**Напівжирне накреслення**	**bold**
<I> та </I>	**Курсивне накреслення**	**italic**
<U> та </U>	**Підкреслене**	**underlined**
<S> та </S>	**Перекреслене**	**striken**
^{та}	**Верхній індекс**	**superscrtpt**
_{та}	**Нижній індекс**	**subscript**

Є й інші теги, які можна застосовувати для зміни параметрів тексту.

• ***<BIG>*** Текст ***</BIG>*** — збільшення розміру шрифту.

• ***<SMALL>*** Текст ***</SMALL>*** — зменшення розміру шрифту.

• ***<EM>*** Текст ***</EM>*** — виділення важливих фрагментів тексту. (Текст відображатиметься курсивом)

• ***<DEL>*** Текст ***</DEL>*** — створення перекресленого тексту. (Текст буде перекреслено горизонтальною лінією).

5.5 Нумеровані та марковані списки

У HTML-документах використовують три види списків:

• невпорядкований (маркований);

• упорядкований (нумерований);

• список визначень.

У *невпорядкованому списку* для виділення елементів використовуються певні позначки (крапки, квадрати тощо). Починається такий список із тегу **<UL>**, а закінчується — **</UL>**.

Кожний елемент списку починається з тегу **<LI>**. Тег <UL> може мати атрибут TYPE, який визначає форму позначки. Цей атрибут набуває таких значень:

• **disk** — зафарбоване коло;

• **circle** — коло;

• **square** — маленький чорний квадрат.

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<UL TYPE=disk>
<LI> I-кур </LI>
<LI>II-курс </LI>
<LI>III-курс </LI>
</BODY>
</HTML>
```

В *упорядкованому списку* всі елементи пронумеровані. починається список із тегу **<0L>**, а закінчується — **</OL>**. Кожний елемент такого списку також

починається з тегу **<LI>.** Тег **<OL>** може мати атрибут **TYPE**, який визначає тип нумерації. Цей атрибут набуває таких значень:

• A — велика літера;
• a — мала літера;
• I — велика римська цифра;
• i — мала римська цифра;
• 1 — арабська цифра.

У разі потреби за допомогою атрибута START можна задати відмінний від одиниці початковий номер елемента, наприклад <0L TYPE="I« START="5">, Нумерований список, початковий елемент якого позначено латинською літерою E, яка є п'ятою в алфавіті, описують так:

```
<OL TYPE="A" START="5">
<LI>Перший елемент списку</LI>
<LI>Другий елемент списку</LI>
<LI>Третій елемент списку</LI>
<LI>Четвертий елемент списку</LI>
</OL>
```

<BASEFONT> для всього документа призначається розмір шрифту.
Наприклад: <BASEFONT SIZE=6>
<FONT> можна визначити колір шрифту для кожного елемента списку.
Наприклад:

```
<HTML>
<TITLE>Райдуга</TITLE>
<BODY>
<BASEFONT SIZE=6>
<OL TYPE=i>
<LI><FONT COLOR=red>Червоний</FONT></LI>
<LI><FONT COLOR=orange>Оранжевий</FONT></LI>
<LI><FONT COLOR=yellow>Жовтий</FONT></LI>
<LI><FONT COLOR=green>Зелений</FONT></LI>
<LI><FONT COLOR=lightblue>Блакитний</FONT></LI>
<LI><FONT COLOR=blue>Синій</FONT></LI>
<LI><FONT COLOR=darkmagenta>Фіолетовий</FONT></LI>
</OL>
</BODY>
</HTML>
```

5.6 Текстові гіперпосилання

Для створення гіперпосилання необхідно використовувати *теги* **<A>** та **</A>**, визначивши для тегу <A> *атрибут* **HREF**. Його значенням має бути адреса URL, на яку вказує посилання. Текст посилання розташовують між тегами <A> і </A>.

Якщо веб-сторінка, на яку вказує посилання, розміщена на іншому веб-сайті, то значенням атрибута HREF має бути повна URL-адреса з назвою протоколу включно; такі посилання називають *зовнішніми. Якщо ж гіперпосилання вказує на*

іншу сторінку того самого сайту, то для пошуку документа достатньо задати лише відносний шлях; таке посилання називають *внутрішнім.*

Гіперпосилання може вказувати на певне місце всередині сторінки, якщо туди попередньо вбудувати якір-мітку.

Для визначення якоря також використовують теги **<A> і </A>**, але замість атрибута **HREF** задають атрибут **NAME**, значенням якого має бути ім‘я якоря. Воно може складатися з літер та цифр, але не повинно містити символів пробілу. Якщо на сторінці є кілька міток, то всі вони повинні мати різні назви.

Для створення посилання на встановлений якір потрібно у тегу **<A>** зазначити його ім'я в кінці адреси URL після імені документа, відокремивши його символом #. Символ # вказує на те, що після нього записано назву мітки, а не ім'я файлу. Посилання на мітку всередині поточного документа задають так:

<A HREF="#назва_мітки">Текст посилання</A>.

Якщо в атрибуті HREF задати адресу електронної пошти зі словом **mailto**: перед нею, то після вибору такого посилання можна надіслати електронний лист, де в полі Кому буде записано цю адресу.

```
...
<BODY>
Тернопільський національний технічний університет імені Івана Пулюя
<A HREF="http://www.vpedliceum.blogspot.com">блог</A>.
<P> А тепер можна перейти на
<A HREF="II-m.html">
іншу сторінку</A>.
<P>Про те, як зв'язатися з автором, розповідається
в <A HREF="#olenap">
кінці цієї сторінки</A>
<BR><BR>
Можна використати матеріали, що розміщені
<A HREF="text.doc"> в цьому текстовому документі.</A>
<BR><BR>
<A NAME="olenap"
HREF="mailto:pedliceum@mail.ru">pedliceum@mail.ru</A>
</BODY>
…
<HTML>
<HEAD>
<TITLE></TITLE>
<HTML>
<BODY LINK=magenta ALINK=yellow>
<H1><A HREF="1.html">Наша продукція</A></H1>
<H1><A HREF="1.html">Наші замовники</A></H1>
</BODY>
</HTML>
```

ЛЕКЦІЯ 6 ВИКОРИСТАННЯ ТАБЛИЦЬ У HTML-ДОКУМЕНТАХ

6.1 Елементи таблиці

Для створення таблиці використовують чотири елементи. Таблицю описують за допомогою тегів **<TABLE> ... </TABLE>;** вона повинна мати один або кілька рядків **<TR>...</TR>**, у кожному з яких може міститися заголовок **<TH>...</TH>** або дані **<TD>...</TD>.**

За умовчанням таблиця має невидимі межі комірок. Для того щоб зробити їх видимими, використовують атрибут **BORDER**, який має цілочислове значення, що визначає товщину рамки в пікселах, наприклад <TABLE BORDER=5>.

Таблиця може мати заголовок, який задається тегами **<CAPTION> та </CAPTION>.** Тег <CAPTION> може *містити атрибут ALIGN з одним із значень top або bottom*, які визначають розташування заголовка відповідно перед таблицею або після неї (за умовчанням встановлено значення top). Наприклад, <CAPTION ALIGN=bottom> Розклад уроків </CAPTION>.

Кожний рядок таблиці починається з тегу **<TR>** і закінчується тегом **</TR>.** Якщо рядок містить *заголовки стовпців таблиці, то використовують теги* **<TH> і </TH>,** якщо ж *дані* — то **<TD> і </TD>.**

Кілька комірок можуть бути об'єднані в одну як по горизонталі, так і по вертикалі. Об'єднання першого типу застосовують тоді, коли потрібно створити для кількох стовпців спільний заголовок.

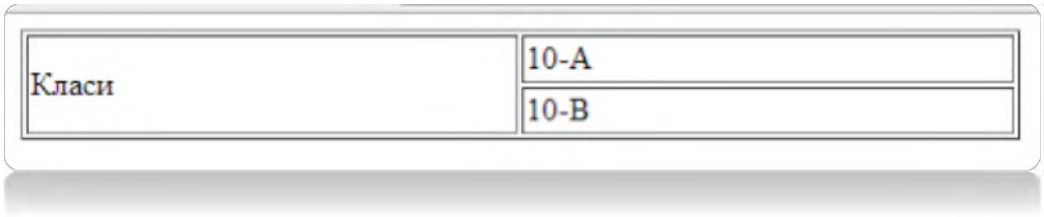
Коли вміст кількох комірок поспіль у стовпці однаковий, їх об'єднують по вертикалі.

Для об'єднання комірок використовують такі атрибути: **COLSPAN** *(об'єднання по горизонталі, у рядку)* і **ROWSPAN** *(по вертикалі, у стовпці)* тегу <TD>. Значеннями цих атрибутів є кількість об'єднуваних стовпців або рядків. Наприклад, **C0LSPAN=3** *означає, що комірка розтягнута на 3 стовпці*, а **R0WSPAN=2** *— що комірка займає 2 рядки.*

Розглянемо далі, як настроювати ширину всієї таблиці, а також окремих комірок. Зауважимо, що ширина всіх комірок в одному стовпці однакова, а в одному рядку може бути різною. *Ширину таблиці задають атрибутом* **WIDTH** *у тегу <TABLE>,* Ширину таблиці можн*а комірки — за допомогою того самого атрибута у тегу <TD> або <TH>.* а визначати як у пікселах, так і у відсотках від ширини вікна, а комірки — у пікселах або у відсотках від ширини таблиці.

```
<HTML>
<TITLE>Приклад встановлення ширини таблиці</TITLE>
<BODY>
<TABLE BORDER="1" WIDTH=500>
<TR>
<TD ROWSPAN="2" WIDTH=50%>Класи</TD>
<TD WIDTH=250>10-A</TD>
</TR>
<TR>
<TD>10-B</TD>
```

```
</TR>
</TABLE>
</BODY>
</HTML>
```



Вся таблиця має ширину 500 пікселів. Ширина комірки з текстом ≪Класи≫ становить 50 % від ширини всієї таблиці, а комірки з текстом ≪10-А≫ — 250 пікселів. Комірка з текстом ≪10-Б≫ теж завширшки 250 пікселів (хоча для неї ширину не було задано), оскільки вона розташована в одному стовпці з попередньою.

Текст всередині комірок можна вирівнювати як по горизонталі, так і по вертикалі. Горизонтальне вирівнювання вмісту всього рядка задають за допомогою атрибута **ALIGN** *у тегу* **<TR>;** цей самий атрибут у тегах **<TD>** або **<TH>** в окремих комірках виконує таку саму роль. *Атрибут* ***ALIGN*** *може набувати значень center, right, left або justify , які задають вирівнювання вмісту таблиці відповідно по центру, за правим чи лівим краєм або за шириною.*

Приклади тегів **<TR> та <TD>** із використанням вирівнювання: **<TR ALIGN=right>.** Вертикальне вирівнювання задають у тегах <TR>, <TD> або <TH> за допомогою атрибута **VALIGN**, який може набувати значень *top, bottom та middle,* що визначають вирівнювання відповідно за верхнім та нижнім краєм і по центру. Для оформлення зовнішньої рамки таблиці можна використовувати атрибут **FRAME тегу <TABLE>.** Його значення визначають, що браузер відображатиме:

• box — всі чотири сторони рамки;
• above — лише верхню межу рамки;
• below — лише нижню межу;
• lhs — лише ліву межу;
• rhs — лише праву межу;
• hsides — верхню й нижню межі рамки;
• vsides — ліву й праву межі рамки;
• void — зовнішня рамка не відображатиметься.

Відображення розділювальних ліній між стовпцями та рядкам таблиці визначають за допомогою атрибута **RULES** тегу **<TABLE>.**

Він може набувати таких значень:

- a l l — відображати всі вертикальні та горизонтальні лінії;
- rows — лише горизонтальні лінії між рядками;
- cols — лише вертикальні лінії між стовпцями;
- none — не показувати розділювальних ліній.

6.2 Колірне оформлення таблиць

Для оформлення комірок таблиць за допомогою кольорів застосовують атрибут **BGCOLOR**. Колір можна визначати, вказуючи його назву англійською мовою або символ # та шістнадцяткове число.

Якщо потрібно задати колір для всієї таблиці, то атрибут BGCOLOR задають у тегу <TABLE>, для зміни кольору лише в одному рядку — в тегу <TR>, а настроювання кольору окремої комірки забезпечує наявність цього атрибута всередині тегу <TD> або <TH>.

Колір рамки таблиці можна задати за допомогою таких атрибутів:

- **BORDERCOLOR —** *колір ВСІЄЇ рамки;*
- **BORDERCOLORLIGHT —** *колір світлої частини рамки;*
- **BORDERCOLORDARK —** *колір темноїЇ частини рамки.*

Ці атрибути вставляють у тег <TABLE>. Щоб вони діяли, необхідна наявність ще й атрибута **BORDER**, який задає товщину рамки.

Веб-сторінка сайту може містити кілька блоків або вікон, які називають ***фреймами,*** *або* ***кадрами.*** *У кожному з них відображається* свій HTML-документ. В одному фреймі може міститися навігаційне меню, а в іншому відкриватися веб-сторінки, на які вказують його пункти.

Для того щоб створити веб-сторінку з фреймами, потрібно кілька HTML-документів. В одному з них задають розмітку екрана, тобто розташовують у вікні браузера фрейми, кожному з яких призначають свої документи.

Сторінка з розміткою, як і звичайна, починається з тегу <HTML> і закінчується тегом </HTML>. Для поділу екрана на кілька фреймів використовують теги ***<FRAMESET> і </FRAMESET>.*** Перший має бути розташований після тегу заголовка, але перед тегом <BODY>.

Іноді в таких документах зовсім не використовують тег <BODY>.

Два фрейми можна розташовувати поруч по горизонталі або один над одним. У першому випадку використовують атрибут COLS, а в другому — атрибут ROWS тегу <FRAMESET>. Для поділу вікна на фрейми через кому записують два числа, які визначають розміри фреймів. Для трьох фреймів потрібно три числа. Розміри фреймів вимірюють у пікселах або відсотках від розміру екрана.

Якщо потрібно зазначити, що фрейм займає те місце, яке залишилося, використовують символ *.

Наприклад, тег <FRAMESET ROWS="150, *"> задає поділ вікна на два горизонтальні фрейми, один з яких має висоту 150 пікселів. а другий займає те місце, що залишилося. Тег <FRAMESET COLS="20%, 55%, *"> задає поділ вікна на три вертикальні фрейми, один з яких займає 20 % від ширини екрана, другий — 55 %, а третій займає те місце, що залишилося. Можна використовувати одночасно і горизонтальний, і вертикальний поділ вікна на фрейми — це роблять за допомогою вкладення тегів <FRAMESET> один в один.

Після поділу екрана на вікна для кожного фрейму слід задати HTML-документ, який відображатиметься в ньому. Для цього використовують тег <FRAME> з атрибутами, що керують властивостями

фреймів:

• **SRC** — задає ім'я файлу, що відображатиметься у фреймі;

• **NAME** — задає ім'я фрейму;

• **SCROLLING** — визначає наявність (значення yes) або відсутність (значення по) смуг прокручування у вікні фрейму (за умовчанням — yes);

• **NORESIZE** — забороняє користувачу змінювати розміри фрейму;

• **BORDER** — визначає ширину розділювальної смуги між фреймами в пікселах;

• **BORDERCOLOR** — визначає колір розділювальної смуги між фреймами;

• **MARGINHEIGHT** — додає порожнє поле, висота якого визначена в пікселах, між верхньою межею фрейму і початком тексту або графіки;

• **MARGINWIDTH** — додає порожнє поле, ширина якого визначена в пікселах, між боковими межами фрейму і початком тексту або графіки.

Оскільки фрейми підтримують не всі браузери, необхідно помістити тег <NOFRAME> перед тегом <BODY>, а між <BODY> I </BODY> записати повідомлення, яке з'являтиметься у вікні, якщо браузер не підтримує фрейми. Після </BODY> має йти </NOFRAME>.

ЛЕКЦІЯ 7 ФРЕЙМИ, ЇХНІ ТЕГИ Й АТРИБУТИ

Веб-сторінка сайту може містити кілька блоків або вікон, які називають ***фреймами,*** *або* ***кадрами.*** *У кожному з них відображається* свій HTML-документ. В одному фреймі може міститися навігаційне меню, а в іншому відкриватися веб-сторінки, на які вказують його пункти.

Для того щоб створити веб-сторінку з фреймами, потрібно кілька HTML-документів. В одному з них задають розмітку екрана, тобто розташовують у вікні браузера фрейми, кожному з яких призначають свої документи.

Сторінка з розміткою, як і звичайна, починається з тегу <HTML> і закінчується тегом </HTML>. Для поділу екрана на кілька фреймів використовують теги ***<FRAMESET> і </FRAMESET>.*** Перший має бути розташований після тегу заголовка, але перед тегом <BODY>.

Іноді в таких документах зовсім не використовують тег <BODY>.

Два фрейми можна розташовувати поруч по горизонталі або один над одним. У першому випадку використовують атрибут COLS, а в другому — атрибут ROWS тегу <FRAMESET>. Для поділу вікна на фрейми через кому записують два числа, які визначають розміри фреймів. Для трьох фреймів потрібно три числа. Розміри фреймів вимірюють у пікселах або відсотках від розміру екрана.

Якщо потрібно зазначити, що фрейм займає те місце, яке залишилося, використовують символ *.

Наприклад, тег <FRAMESET ROWS="150, *"> задає поділ вікна на два горизонтальні фрейми, один з яких має висоту 150 пікселів. а другий займає те місце, що залишилося. Тег <FRAMESET COLS="20%, 55%, *"> задає поділ вікна на три вертикальні фрейми, один з яких займає 20 % від ширини екрана, другий — 55 %, а третій займає те місце, що залишилося. Можна використовувати одночасно і горизонтальний, і вертикальний поділ вікна на фрейми — це роблять за допомогою вкладення тегів <FRAMESET> один в один.

Після поділу екрана на вікна для кожного фрейму слід задати HTML-документ, який відображатиметься в ньому. Для цього використовують тег <FRAME> з атрибутами, що керують властивостями

фреймів:

• **SRC** — задає ім'я файлу, що відображатиметься у фреймі;

• **NAME** — задає ім'я фрейму;

• **SCROLLING** — визначає наявність (значення yes) або відсутність (значення по) смуг прокручування у вікні фрейму (за умовчанням — yes);

• **NORESIZE** — забороняє користувачу змінювати розміри фрейму;

• **BORDER** — визначає ширину розділювальної смуги між фреймами в пікселах;

• **BORDERCOLOR** — визначає колір розділювальної смуги між фреймами;

• **MARGINHEIGHT** — додає порожнє поле, висота якого визначена в пікселах, між верхньою межею фрейму і початком тексту або графіки;

• **MARGINWIDTH** — додає порожнє поле, ширина якого визначена в пікселах, між боковими межами фрейму і початком тексту або графіки.

Оскільки фрейми підтримують не всі браузери, необхідно помістити тег <NOFRAME> перед тегом <BODY>, а між <BODY> І </BODY> записати повідомлення, яке з'являтиметься у вікні, якщо браузер не підтримує фрейми. Після </BODY> має йти </NOFRAME>.

7.1 Формат GIF (.gif)

Формат GIF (Graphics Interchange Format — формат обміну графічними даними) почали використовувати з 1987 року для обміну малюнками через канали зв'язку глобальної мережі. Він зберігає зображення, що можуть містити не більш ніж 256 кольорів, і не залежить від апаратного забезпечення комп'ютера. Окрім цього, під час завантаження таких файлів на веб-сторінку може бути використаний режим interlaced (рядки малюнка відображаються через один). Ця технологія дає змогу побачити приблизний вміст картинки до її повного відтворення і в разі необхідності скасувати завантаження.

У GIF-файлах можна зробити один чи більше кольорів прозорими: вони будуть невидимими у вікні браузера та деяких інших програм. Також є можливість зберігати в одному файлі кілька картинок, задавши час показу кожної, тобто застосовувати анімацію. На сьогодні фахівцями розроблено чимало готових анімаційних GIF-файлів, які можна використовувати на своїх веб-сторінках.

7.2 Формат PNG (.png)

Формат PNG (Portable Network Graphic — переносні мережні графічні дані) є одним із перспективних форматів графіки для Інтернету, який створено з метою заміни GIF.

Формат PNG забезпечує високу якість графіки та прийнятні розміри файлів. Зображення може зберігатися як у реальних кольорах, так і в 256-колірній GIF-палітрі.

7.3 Формат JPEG (.jpg)

Формат JPEG (Joint Photographic Experts Group — об'єднана група експертів у галузі фотографії) був створений для того, щоб позбутись обмежень, властивих формату GIF.

Два попередніх формати орієнтовані на рисовану і креслену графіку. Формат JPEG призначений для збереження повноколірних реалістичних фотозображень. Він має потужні засоби для стиснення зображень, щоправда шляхом зниження їхньої якості. JPEG-зображення зберігаються у файлах із розширенням .jpg.

Оскільки цей формат спеціально розробляли для збереження ілюстрацій, що містять велику кількість кольорів, він є найприйнятнішим для деяких типів графічних даних. Це кольорові фотографії, графічні дані з градієнтним заповненням частин зображення, фотознімки з відтінками одного кольору тощо.

7.4 Створення тла веб-сторінки

Зазвичай для тла вибирають зображення невеликого розміру та неяскравих кольорів, а його файл роблять маленького розміру, що дає змогу швидко завантажувати сторінки. Текст на тлі повинен легко читатись.

Для формування тла у вигляді малюнка у тегу <BODY> використовують атрибут BACKGROUND, значенням якого є URL-адреса файлу зображення. Наприклад,

Тег **<BODY BACKGROUND="1. jpg“>**

визначає, що для фонового заповнення веб-сторінки буде використано файл **1.jpg.**

Далі наведено приклад HTML-коду сторінки, в якої тло оформлене у вигляді малюнка, що міститься у файлі **mone.jpg.**

```
<HTML>
<TITLE></TITLE>
<BODY BACKGROUND="mone.jpg">
<H1><A HREF="l.html">Наша продукція</A></H1>
<H1><A HREF="2.html">Наші замовники</A></H1>
</BODY>
</HTML>
```

7.5 Вставлення зображень на веб-сторінку

Для розміщення малюнків у HTML-документі використовують одинарний тег **<IMG>**. Він має обов'язковий атрибут **SRC**, значенням якого є URL-адреса файлу зображення, записана в абсолютній (повністю) або відносній формі (починаючи від поточного каталогу; при цьому ≪батьківський≫ каталог позначають за допомогою двох крапок і слешу (. . /)). Під час відкривання документа браузер завантажує малюнок і відображає його в тому місці документа, де розташований тег <IMG>.

Графічний об'єкт буде показаний на веб-сторінці у своїх реальних розмірах. Якщо є потреба у їх зміні (масштабуванні об'єкта), нові розміри можна задати за допомогою атрибутів WIDTH та HEIGHT, значення яких визначають відповідно ширину та висоту зображення в пікселах. При цьому необхідно зберегти пропорції малюнка, інакше він матиме вигляд розтягнутого або сплюснутого. Слід також пам'ятати, що іноді користувачі відключають відображення графіки у вікні браузера, щоб прискорити завантаження документів. Тому, на випадок, коли малюнка на сторінці не буде, бажано повідомити, що на ньому зображено. Для цього використовують альтернативний текст — більш-менш докладний опис зображення, який задають у тегу **<IMG>** як значення спеціального атрибута **ALT**. Якщо браузер не може показати малюнок, він замість нього виводить цей текст.

Розглянемо HTML-документ, в якому використаємо малюнок flamingo.jpg, де зображено фламінго. У тегу <IMG> задамо атрибути WIDTH,HEIGHT та ALT.

```
<HTML>
<TITLE>Малюнок</TITLE>
<BOOУ BACKGROUND="flamingo.jpg">
<IMG SRC="2.jpg" WIDTH=300 HEIGHT=200 ALT="Фламінго">
</BODY>
</HTML>
```

7.6 Розміщення зображень у тексті

Зображення можна розмістити у тексті, але при цьому слід визначити, в який спосіб текст обтікатиме його. Для взаємного розміщення тексту і зображень призначений атрибут ALIGN у тегу <IMG>, який може набувати, зокрема, таких значень:

• left — зображення розміщене в лівій частині сторінки, текст обтікає його з правого боку;

• right — зображення розміщене у правій частині сторінки, текст обтікає його з лівого боку;

• top — обтікання немає, зображення розміщене в тексті, відповідний рядок якого вирівняно за верхньою межею малюнка;

• bottom — обтікання немає, зображення розміщене в тексті, відповідний рядок якого вирівняно за нижньою межею малюнка;

• middle — обтікання немає, зображення розміщене в тексті, відповідний рядок якого вирівняно по середній лінії малюнка

```
<HTML>
<TITLE>МАЛЮНОК</TITLE>
<BODY BACKGROUND="1.jpg">
<IMG SRC="flamingo.jpg" ALIGN=left HSPACE=10 VSPASE=10>
Блакитні води океану та коралові рифи,
ласкаве сонце і білосніжний прибережний пісок,
фантастичний підводний світ і казкові птахи,
буяння барв сходів і заходів, непрохідні ліси
і п'янкий запах орхідей...
Сприятливий клімат Острова Свободи дає можливість
туристам відвідувати його цілий рік. Тут немає зайвої
вологості, а спека не така страшна завдяки постійним
подувам бризів із океану. Середньорічна температура
повітря й води становить + 26—28°С.
</BODY>
</HTML>
```

7.7 Графічні гіперпосилання

Зображення, як і текст, можна використовувати як посилання. Для цього тег **<IMG> необхідно помістити між тегами <A> і </A>.** Зображення-посилання має синю рамку, а після наведення на нього вказівника миші той набуває такої самої форми, що й у випадку текстового посилання.

Приклад

```
<HTML>
<TITLE>Птахи</TITLE>
<BODY BACKGROUND="1.jpg">
<A HREF="gorobci.html"><IMG SRC="gorobci.jpg"></A>
<A HREF="flamingo.html"><IMG SRC="flamingo.jpg"></A>
</BODY>
</HTML>
```

7.8 Карти посилань

Якщо тег <IMG> з ім'ям файлу рисунка розмістити між тегами <A> і </A>, то з цим рисунком буде пов'язане одне гіперпосилання. Можна зробити і по-іншому: поділити рисунок на області, і кожну з них перетворити на гіперпосилання. Такі області називають *гарячими, а повне зображення — картою посилань. Вказівник*

миші після наведення на гарячу область набуває форми руки, як і у випадку текстового посилання. Гарячі області можуть бути прямокутними, багатокутними або круглими. Вибираючи зображення, яке заплановано зробити картою

посилань, слід подбати про те, щоб гарячі області не перетиналися. Для того щоб відвідувач сайту зрозумів, що це карта гіперпосилань, а не просто гарна картинка, необхідно дати пояснювальні тексти.

7.9 Формування карти гіперпосилань

Графічну карту посилань створюють за допомогою кількох тегів. У тегу **<IMG>** визначають атрибут SRC, щоб задати зображення, і атрибут USEMAP, значенням якого є ім'я карти (має починатися символом #). Власне карту створюють за допомогою тегу **<MAP>** з атрибутом NAME, який містить ім'я карти (визначене в атрибуті USEMAP, але без символу #) та тегу **</MAP>.** Між ними записують теги **<AREA>,** які задають параметри гарячих областей. При цьому використовують такі атрибути:

• HREF — визначає гіперпосилання, пов'язане з областю;

• SHAPE — визначає форму області, його значеннями можуть

бути:

• rect — прямокутник;

• polygone або poly — багатокутник;

• circle — коло;

• COORDS — містить координати області у вигляді взятого в лапки

списку чисел, розділених комою. Для прямокутника задають чотири числа — координати верхнього лівого і правого нижнього кутів, для багатокутника — послідовні координати кожного кута, для кола — координати центра і радіус.

```
<IMG SRC="map.bmp" USEMAP="#karta">
<MAP NAME="karta">
<AREA HREF="l.html" SHAPE="rect" COORDS="70,250,160,270">
<AREA HREF="2.html" SHAPE="circle" COORDS="260,140,60">
</MAP>
```

Слід попрацювати над малюнком карти у графічному редакторі, щоб окреслити необхідні області, які пізніше стануть гіперпосиланнями, та визначити координати, що будуть записані як значенняатрибута COORDS.

Місце розташування будь-якої точки на малюнку визначають два числа — її горизонтальна та вертикальна координати. Верхня ліва точка має координати (0;0). Більшим горизонтальним координатам відповідають правіші точки, більшим вертикальним — нижчі.

Наприклад, якщо малюнок має розмір 400x300 пікселів, і необхідно визначити прямокутну область, що займає його верхню ліву частину та має вдвічі меншу ширину і висоту, запис тегу <AREA> буде таким:

<AREA SHAPE="rect" COORDS="0,0,200,150">

У цьому записі атрибут COORDS визначає прямокутник за допомогою двох точок: лівого верхнього кута з координатами (0;0) і правого нижнього з координатами (200; 150) (центральна точка малюнка, що має розмір 400x300).

Якщо гаряча область є колом, то слід зазначити координати лише однієї точки — центра кола, а також задати його радіус, наприклад:

<AREA SHAPE="circle" COORDS="200,150,40">

Такий запис визначає круглу гарячу область, розміщену в центрі малюнка розміром 400x300. Координати центральної точки (200;150), радіус кола — 40 пікселів. Найскладніший випадок — багатокутна гаряча область, для якої потрібно послідовно зазначати координати всіх кутів. ПрипустиМО, що на малюнку розміром 400x300 пікселів потрібно визначитигарячу область у формі рівнобедреного трикутника, основа якогопроходить точно посередині малюнка, займаючи всю його ширину, а вершина міститься посередині верхньої межі. Цю область визначають так:

<AREA SHAPE="poly" COORDS="0,150,400,150,200,0">

ШІСТЬ координат задають три точки — кути трикутника. У цьому випадку їх послідовність неважлива, і ви могли б написати, наприклад, 400;150;0;150;200;0, проте коли кількість кутів є більшою, важливим є порядок з'єднання точок. Наприклад, записи

<AREA SHAPE="poly" COORDS="0,0,200,0,200,150,100,75,0,150">

<AREA SHAPE="poly" COORDS="0,0,200,0,100,75,200,150,0,150">

дадуть різні результати. У першому випадку це прямокутник із вирізаним трикутником знизу, а в другому — з вирізаним трикутником праворуч.

Визначення гарячих областей є дуже клопіткою роботою, оскільки доводиться задавати координати кожної вузлової точки. На реальному малюнку це неможливо зробити точно, тому за допомогою графічних редакторів знаходять усі координати, а потім переносять їх у HTML-документ.

СПИСОК ЛІТЕРАТУРИ

1. Зубенко В.В., Омельчук Л.Л.. Програмування : навчальний посібник (гриф МОН України) / - К. : ВПЦ "Київський університет", 2011. - 623 с.

2. Ставровский А.Б, Карнаух Т.О. Перші кроки програмування.- К.:Диалектика.-2005, с.389.

3. Кравець П.О. Об'єктно-орієнтоване програмування. – Видавництво Львівської політехніки, 2012. – 624 с.

4. Daniel Solis. Illustrated C# 7. The C# Language Presented Clearly, Concisely, and Visually. – APress, 2018. – 799 p.

5. Herbert Schildt. C# 4.0: The Complete Reference. – McGraw-Hill/Osborne Media, 2010. – 949 p.