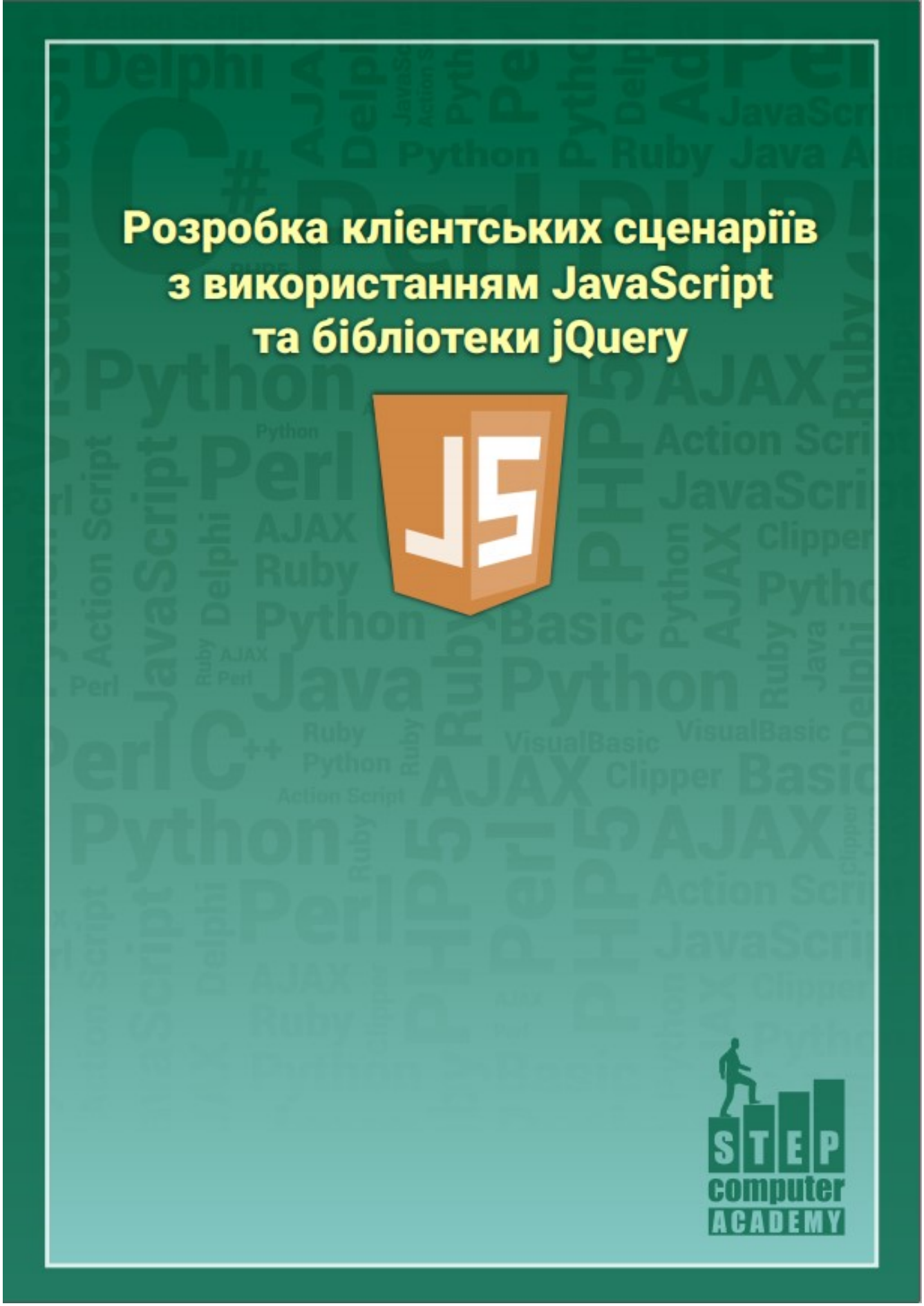
Розробка клієнтських сценаріїв з використанням JavaScript і бібліотеки jQuery

Уроки

Додаткові матеріали

js12_1.7z


Розробка клієнтських сценаріїв
з використанням JavaScript
та бібліотеки jQuery
JS
STEP
computer
ACADEMY

Патерн MVC

Продовжимо знайомство з патернами проєктування і розглянемо детальніше один із найпоширеніших патернів — MVC.

1. Що таке патерн MVC?

Патерн MVC це один зі структурних шаблонів проєктування. Пригадайте, які патерни належать до структурних? Правильно, це шаблони, що стосуються структури програми або програмного комплексу, схеми розподілу складових частин, а також взаємодії (зв'язків) між ними.

Назва MVC є абревіатурою слів Model-View-Controller, що визначають три основні складові частини програми: Модель, Презентацію та Контролер. Одразу уточнимо, що в тексті уроку ми будемо з великої літери називати складові частини, наприклад, «Модель», водночас для позначення конкретних блоків, наприклад, «модель товарів» використовуватимемо звичайне написання.

2. Цілі та завдання патерну Model-View-Controller

Завданням будь-якого з шаблонів є передати позитивний досвід розроблення програмних продуктів і підказати розробнику найбільш ефективні схеми для їхніх власних проєктів.

Одна з глобальних цілей патерну MVC полягає в тому, щоб максимально розділити завдання з опрацювання даних і завдання відображення отриманих результатів. У цьому ключі, як протилежний підхід, можна навести антипатерн «змішування коду і розмітки», в якому дані обробляються і результати виводяться в одному і тому ж блоці коду. Далі наведено

умовні приклади, у яких ілюструється змішування та розділення коду і розмітки:

Таблиця 1

Змішування коду та розмітки	Поділ коду та розмітки
Price: {{Math.round(price,2)}} Discount: {{Math.round(price* (1-percent),2)}}	{{fullPrice = Math.round(price,2); disc = Math.round(price*(1-percent),2)}} Price: {{fullPrice}} Discount: {{disc}}

Змішування значно ускладнює спільну роботу дизайнерів і програмістів, а також має низку інших недоліків, але наше завдання обговорювати не антипатерн, а правильні підходи. Однак, розглядаючи протилежності, часто стають набагато зрозумілішими завдання, які вирішуються тим чи іншим інструментом.

Уявімо собі, що перед нами поставлено завдання реалізувати різні форми відображення даних: у вигляді списку або у вигляді таблиці, причому мати можливість перемикатися між цими формами за вибором користувача.

У змішаному варіанті для створення нового подання доведеться копіювати всі присутні обчислення в нову розмітку. При цьому, якщо з часом алгоритми розрахунків зміняться, то доведеться вносити одні й ті самі зміни в усі створені подання.

У прикладі з розділеним кодом ми вже можемо помітити, що в програмі чітко проглядаються дві частини — обробка даних (математичні формули) та їхнє представлення (розмітка). Такий поділ значно спрощує нам вирішення поставленого зав-

дання. Усі розрахунки як для списку, так і для таблиці, будуть однаковими, відрізнятися будуть тільки теги їхньої розмітки. Якщо алгоритми розрахунків зміняться, то зміни треба буде внести тільки в одному місці.

У термінах MVC, ми вже неявно познайомилися з двома структурними частинами «Model» і «View». Модель являє собою розрахункову частину, а Подання (*View*) — розмічальну. У цьому простому прикладі не дуже добре видно роль Контролера, але саме його завданням буде перемикання різних форм відображення даних залежно від вибору користувача. Розберемося з цими частинами більш детально.

3. Model

Що таке Model?

Модель — це структурна частина програми, що відповідає за роботу з даними. На цьому визначенні можна було б зупинитися, але, оскільки вся програма, так чи інакше, працює з даними (інакше в комп'ютерах не буває), у визначення моделі часто додають уточнення: «за роботу з бізнес-даними» або «бізнес-логіку». Цим підкреслюється той факт, що модель служить для роботи з основними даними, заради яких створюється програма, а не просто для перенесення будь-яких математичних обчислень в окрему структурну частину.

У вужчому сенсі, модель являє собою деякий об'єкт, який самостійно заповнює себе актуальними даними. У найпростішому випадку ми можемо порівняти модель з об'єктом JavaScript «Date», який при кожному конструюванні «new Date()» міститиме однакові поля — день, місяць, рік тощо, але значення цих полів будуть різними залежно від моменту створення об'єкта. Сама процедура заповнення полів від нас

прихована, ми лише знаємо, що створення об'єкта заповнить їх актуальними даними.

Подібним чином і моделі мають фіксовані набори полів (інтерфейси), а значення для цих полів формуються внутрішніми алгоритмами моделі. Нам достатньо знати, що створення або запит моделі заповнить її потрібним чином.

Як уже зазначалося, це визначення того, що модель є об'єктом, не є повним. Таке визначення популярне для ООП-мов і, дійсно, дуже поширене. Проте, у більш загальному сенсі модель — це структурна частина програми із зазначеними вище властивостями. Це може бути, як об'єкт, так і модуль, пакет, функція або інша сутність. Суть Моделі не у формі її побудови, а в завданнях, які на неї покладено.

Цілі та завдання Model

Моделі створюються для обробки бізнес-даних

Головною метою Моделі є обробка бізнес-даних програми або бізнес-логіка. Трохи повторимося, бізнес-даними ми називаємо ті дані, заради яких створюється програма. Це можуть бути дійсно дані з бізнесу, наприклад, відомості про продажі мережі магазинів. Однак, не варто цим обмежуватися. Бізнес-даними ми називатимемо будь-які дані, що обслуговуються програмою, наприклад, публікації на форумі або розміщені оголошення, відео-трансляції або довідник із медицини, набір стандартів із веб-безпеки або колекцію фрактальних зображень — усе те, чому присвячена наша програма.

При цьому також скажемо про винятки. До бізнес-даних ми не належатимемо відомості про розміри екрана або його орієнтацію, розрядність кольору монітора, доступні шрифти, тип процесора або операційної системи, про кількість доступних потоків, тип бази даних або діалект SQL для неї. Це дуже ва-

жливі дані, що безпосередньо впливають на роботу програми, але ці дані важливі для будь-якої програми й обробляються однаково (або дуже схожим чином) у всіх них. Бізнес-дані — це відмінні дані, основний контент для конкретного застосунку, робота з цими даними унікальна: обробка даних про продажі сильно відрізняється від обробки відео-контенту. Саме для відокремлення, локалізації унікальних алгоритмів обробки даних створюються моделі.

Моделі приховують алгоритми обробки даних

Алгоритми роботи з бізнес-даними (або бізнес-логіка) має низку особливостей, які й призводять до появи моделей. Про унікальність уже було сказано, але потрібно додати, що навіть для схожих даних алгоритми можуть сильно відрізнятися. Дві мережі магазинів можуть абсолютно по-різному розраховувати показники динаміки, преміювання або депреміювання співробітників, плани розвитку. Ба більше, ці алгоритми можуть мати високу інтелектуальну цінність і вимагати захисту від конкурентів. Тому одним із завдань Моделі можна вважати приховування алгоритмів обробки даних наскільки це можливо.

Модель забезпечує гнучкість змін алгоритмів

Якщо хороші алгоритми унікальні і приховуються від широкого доступу, то програмісти стикаються із завданням розробки цих алгоритмів для кожної нової програми. У процесі створення хороших алгоритмів доводиться стикатися з частою необхідністю вносити зміни в уже впроваджені коди. Причому іноді потрібно також і скасовувати внесені раніше зміни, якщо вони показали гірші результати під час випробувань. Це означає, що модель повинна мати високу гнучкість, давати змогу впроваджувати і «відкочувати» зміни.

Модель забезпечує централізацію змін

Очевидно, що всі зміни мають відбуватися в одному місці програми, а не по безлічі точок впровадження коду. Такі вимоги називають централізацією — створення «центру», зміни в якому відображатимуться на всій решті програми, навіть якщо дані використовуються в кількох інших її частинах.

Модель підтримує цілісність даних

Окрім опрацювання наявних даних, найчастіше, програма повинна мати можливість накопичення нових даних. Користувачі пишуть нові пости, копірайтери додають новий контент, адміністратори завантажують нові товари на сайт. У більшості випадків, нові дані — це насправді комплекс даних, який має приходити як єдине ціле. Наприклад, для товару — це назва, ціна, фотографія, опис, знижка, доступна кількість тощо. Якщо якихось даних не буде, можливі збої в роботі системи, як мінімум, у плані неправильного відображення даних. Для запобігання подібних ситуацій, Модель може містити в собі перелік необхідних полів, які обов'язкові до заповнення. Це підтримує цілісність даних — гарантію того, що всі необхідні складові дані будуть присутні, нічого не буде забуто.

Підбиваючи деякі підсумки щодо завдань Моделі можемо зазначити, що основною метою Моделі є робота з даними, яка не обмежується їх передачею до користувача або від користувача. Модель також забезпечує аналіз і обробку даних.

4. View

Що таке View?

Другою структурною частиною патерну MVC є подання (*View*). Ця частина програми відповідальна за те, яким чином

відбувається взаємодія з користувачем — за інтерфейс користувача.

Подання отримує дані, розраховані моделлю, і формує з них видиму частину програми. Коли ми дивимося на сайт, мобільний або віконний застосунок, ми бачимо саме Подання програми. Перехід на різні сторінки сайту (або вікна додатків) — це перемикання між різними Поданнями.

У найпростішому випадку можна сказати, що Поданням є HTML розмітка сайту та її CSS стилізація. Однак, не зовсім правильно буде вважати, що Подання — це виключно пасивна частина програми, яка лише вставляє конкретні дані в розмічальні шаблони. У разі якщо на сторінці потрібне введення будь-яких даних, Подання має провести первинну перевірку (валідацію) введених даних і забезпечити їхнє передання для подальшого опрацювання.

У складніших випадках Подання перебуває в постійному зв'язку з іншими частинами програми і реагує на зміни «в реальному часі». Напевно кожен із вас бачив, як у месенджерах з'являється напис на кшталт «*typing...*» коли хтось починає набирати для вас повідомлення. Відстеження подібних подій також є частиною Подання.

Цілі та завдання View

Подання реалізує взаємодію з користувачем

Головним завданням Подання є взаємодія з користувачем. З одного боку, Подання має відобразити необхідну інформацію для цієї конкретної сторінки сайту. Найчастіше, різні сторінки — це різні Подання.

Подання забезпечує макетування дизайну

Якщо сайт дотримується деякого єдиного стилю, то різні Подання містять як однакові частини (*Header*, *Footer*, *Menu*), так

і відмінні. Відповідно, до завдань Подання можна віднести організацію макетів, за якими будуються різні сторінки з однаковими частинами. Причому це не просто завдання копіювання спільних частин, а реальна інтеграція — зміни в заголовку сайту мають одразу відобразитися на всіх його сторінках.

Подання оновлює динамічні дані

Інформація, відображена на сторінці сайту, може мати обмежений час актуальності. Можна уявити собі сайт деякої біржі, на якій курси акцій постійно змінюються. Їхній дизайн, розмітка, стилі — все залишається постійним, але самі дані мінливі. Відповідно, завдання відстеження змін даних та їх оновлення також покладається на Подання. У цьому ж контексті можна навести приклад згадуваних вище месенджерів, у яких інформація не тільки може змінитися, а й доповнитися новими повідомленнями.

Подання забезпечує валідацію даних, що вводяться

Потік даних у програмі може відбуватися і в інший бік. Не тільки відображатися, а й створюватися. Напевно кожен проходив процедуру реєстрації на будь-якому сайті. При цьому користувач вводить певні персональні дані, а система перевіряє як сам факт їхнього введення, так і коректність даних, наприклад, номера телефону або адресу електронної пошти. У низці випадків подібна перевірка може відбуватися і не в рамках Подання, але сучасні тенденції розроблення програмного забезпечення дедалі частіше реалізують перевірку одразу для того, щоб унеможливити передачу свідомо неправильних даних. Тому завдання валідації даних, що вводяться користувачем, здебільшого належить до Подання.

Подання готує дані для надсилання

Аналогічні перевірки повинні відбуватися не тільки під час реєстрації, а й при будь-якому введенні даних з боку користувача — розміщення оголошень, додавання повідомлень або публікації новини, адміністрування контенту тощо.

Після перевірки введені дані, вочевидь, мають бути передані системі для внесення в базу даних. Процес передачі даних насправді також є комплексним завданням з безліччю налаштувань і особливостей. По-перше, потрібно перетворити дані в певний формат. Для цього зазвичай використовують стандартні формати: JSON, XML, CSV тощо, але може бути використано й інший формат. По- друге, необхідно провести транспортне кодування — перетворення даних перед надсиланням. У веб-завданнях зазвичай використовується HTTP, в електронній пошті — base64, можливо, навіть знадобиться шифрування даних, якщо використовуються канали підвищеної надійності. Усі ці завдання назвемо попередньою підготовкою даних для надсилання. Звісно, це завдання також відноситься до Подання.

Подання обробляє відповіді/помилки сервера

Передані дані обробляються сервером програми і повертається результат їх обробки. Це може бути як успішний відгук, так і повідомлення про помилки, а, можливо, запит узагалі не зможе бути обслужений унаслідок збоїв роботи будь-якої з частин програми. На подання при цьому покладається завдання аналізу цих результатів — деякі з них можна показувати користувачеві, наприклад, успішний результат. Інші результати слід приховувати, особливо ті, в яких міститься службова інформація про помилки. Наприклад, отримавши повідомлення «*Error "cannot connect to mysql server (10060)"*» Подання має приховати отриманий текст і повідомити користувача нейтральним повідомленням «Виникли проблеми, повторіть запит пізніше».

Подання керує зберіганням локальних даних

Ще одним із завдань Подання є «запам'ятовування» попередніх виборів користувача. Якщо сайт підтримує кілька мов інтерфейсу, то бажано запам'ятати обрану користувачем мову під час першого відвідування сайту і потім спочатку показувати інтерфейс цією мовою. Аналогічна ситуація відбувається і з авторизацією користувача — введення логіна і пароля не має супроводжувати кожен перехід на нову сторінку. Однак, у випадку з авторизацією слід зазначити і зворотне завдання — якщо користувач довго не виявляв активності, то доступ слід скасувати. За аналогією із запам'ятовуванням даних сформулюємо це завдання як «забування» даних. У загальному випадку можна об'єднати завдання запам'ятовування і забування як завдання управління збереженням локальних даних. Локальними ми будемо називати ті дані, які залишаються в поданні і не передаються на зберігання в базу даних системи.

Як бачимо, завдань у подання досить багато. Напевно, в окремих випадках можна говорити і про інші завдання. Хочеться додатково підкреслити той факт, що Подання — це не просто розмітка, а значна частина програми зі своїми завданнями і функціями.

5. Controller

Що таке Controller

Третьою структурною частиною MVC є Контролер (*Controller*). Цю частину можна уявити собі як проміжну ланку між Моделлю та Поданням.

Як ми вже зазначали вище, у програмі може бути досить багато моделей для різних даних, а також безліч подань, у ко-

жному з яких може бути задіяно кілька моделей одразу. Наприклад, на сторінці представлені дані про користувача (посилання на профіль) і водночас дані про товари. Серед цих даних відображаються відгуки користувачів про товари. При цьому на сторінці присутні курси валют та актуальні знижки.

Ми вже знаємо, що всі ці дані — про користувача, товари, відгуки, акції, курси валют — це все різні моделі. Водночас дані від усіх цих моделей потрібні для формування одного подання. Інші подання можуть використовувати інші моделі, в іншій кількості або в іншій комбінації.

В іншому місці сторінки можна навести приклад складання відгуку про товар. Дані про його оцінку (зірки) можуть бути передані в одну модель, що визначає рейтинг товарів, текст відгуку — в іншу модель, питання щодо обслуговування — в третю. Причому відгуки та «зірки» ми бачимо на самому поданні, а запитання, найімовірніше, будуть приховані, відповіді на них надійдуть індивідуально на пошту користувача. Це означає, що в процесах відображення даних і збору даних в одному поданні можуть бути задіяні різні моделі.

Контролер — це та частина системи, яка «знає» про те, які моделі потрібно запросити для того чи іншого подання або заповнити даними, введеними в поданні. Іншими словами, Контролер узгоджує роботу інших частин програми: Моделі та Подання.

Цілі та завдання Controller

Контролер покращує безпеку програми

Головна мета контролера — забезпечити роль посередника між Моделлю і Поданням. Відразу може виникнути запитання «навіщо потрібен посередник?». Чому Подання не може безпосередньо звернутися до Моделі для отримання або передачі необхідних даних?

Насамперед для поліпшення безпеки. Пряме звернення до Моделі розкриває її розташування і відкриває можливості зловмисних дій. Ми пам'ятаємо, що всі коди HTML і JS на сайті є відкритими і кожен може подивитися, як і куди відправляються дані з його форм. Тому Подання звертається до Контролера, а він уже передає дані Моделі.

Контролер забезпечує повну валідацію даних

Якщо допустити пряме звернення до моделей, то до їхніх завдань доведеться додавати валідацію даних, тобто відволікатися від завдань бізнес-логіки обробки даних і займатися перевірками кожного отриманого параметра. Ми вже говорили про те, що завдання валідації відноситься до Подання. Це так, але якщо дані будуть передані в обхід Подання — прямими запитами, то первинна перевірка не буде пройдена, і в систему потраплять дані, які не пройшли перевірку. Тому одним із найголовніших завдань Контролера є повна перевірка або валідація даних. Багато в чому ці перевірки дублюються (з Поданням), але без перевірок Контролер просто неприпустимий. Можна сказати, що Подання забезпечує первинну перевірку, а Контролер — повну.

Контролер моніторить потенційні атаки

Ба більше, до цих перевірок потрібно додати аналіз на потенційні атаки, коли від одного користувача надходить занадто багато запитів. Очевидно, що без Контролера ці завдання повинна буде вирішувати Модель. Подання взагалі не може вирішувати такі завдання, оскільки атаки проводять в обхід цього модуля. Однак, аналіз атак погано поєднується з бізнес-логікою, тому найбільш логічно перекласти його на Контролер. Більше того, визначити від якого користувача надходять запити можуть не всі моделі, а тільки ті, які мають відсилання на користувачів. Наприклад, модель товарів не зможе розв'язувати зазначене завдання, оскільки серед її да-

них користувачі взагалі не значаться. Відповідно, завдання моніторингу потенційних атак має вирішуватися саме на рівні Контролера.

Контролер забезпечує авторизацію

Після перевірки даних і можливих зловмисних дій виникає наступне завдання — авторизація. Авторизацією називається надання доступу до певних ресурсів лише тим користувачам, які мають необхідні права. Якщо особливих прав для ресурсу не потрібно, то для нього перевірки не потрібні і до нього може бути повний доступ. Як уже було зазначено, не кожна модель може забезпечити перевірку користувача, а Подання можна спробувати «обійти». Значить це завдання знову належить Контролеру.

Контролер узгоджує роботу Моделі та Подання

Після успішного проходження всіх перевірок, необхідно запросити дані від певних моделей або, навпаки, розділити дані і передати їх у потрібні моделі. Якщо при цьому все проходить успішно, то слід сформувати одне з уявлень, якщо ж виникають помилки отримання або передавання даних, то потрібно переключитися на подання для помилок (наприклад, на сторінку 404) або повідомити поточне подання про аварійну ситуацію. Ми вже називали це завдання узгодженням Моделі та Подання, зазначимо лише кілька особливостей.

Моделі для подання можуть перебувати в різних місцях, наприклад, курси валют може постачати API деякого банку, доступну кількість товарів — складська облікова програма, а дані про користувачів — модель аутентифікації. Причому для деяких моделей можливі альтернативи, наприклад, якщо один із банків не функціонує, то можна запросити курс від іншого банку. Для інших моделей варіанти не допускаються — кількість товарів може повідомити лише склад.

Як підсумок, Контролер виконує завдання «пляшкового горлечка» — єдиного місця, через яке проходять дані між Моделлю і Поданням. Це дає змогу перевірити і проаналізувати дані, розподілити їх між композитними учасниками, а також поліпшити їхню безпеку.

Зобразимо описану схему компонент та їхньої взаємодії на малюнку.

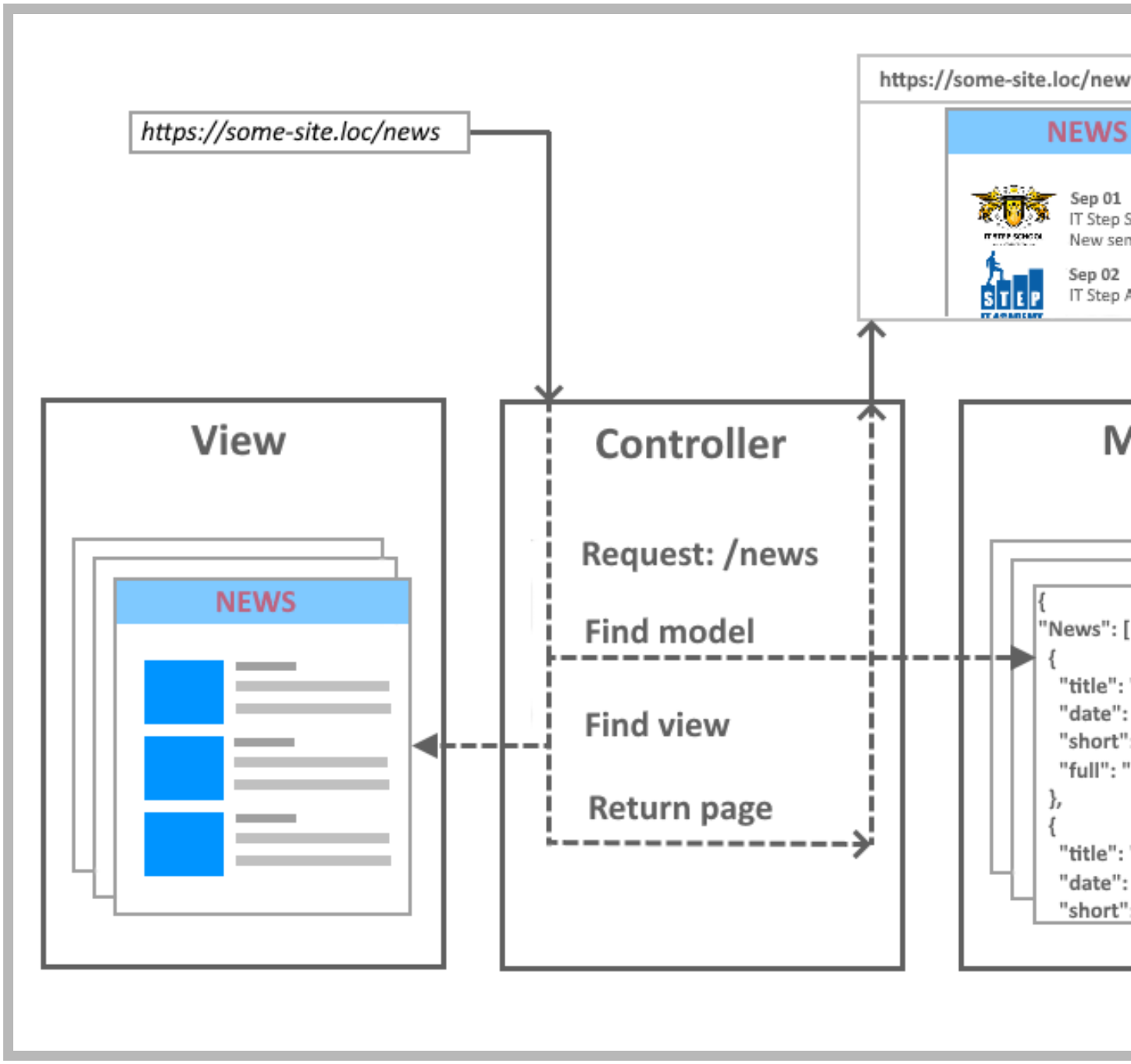


Рисунок 1. Схема патерну MVC

Малюнок схематично ілюструє структуру програми та процес взаємодії її частин:

- користувач вводить у браузері адресу, у прикладі це *https://some-site.loc/news.*
- браузер передає запит у програму і першою частиною в обробці виступає Контролер.
- Контролер аналізує адресу, виділяючи її мету — сторінка новин (*/news*).
- Серед усіх моделей вибирається та, яка постачає дані про новини
- Серед усіх подань вибирається те, яке відповідає за новини
- Подання заповнюється даними про останні новини і результат передається в браузер. Користувач бачить сторінку з новинами.

6. Приклади використання патерну MVC

Як приклад використання патерну MVC створимо демонстраційну програму. Повний код прикладу знаходиться в папці з уроком (файл «*js12_1.html*»). Код відносно великий, тому повністю його наводити в тексті уроку не будемо, а зосередимося на тих елементах, які реалізують патерн.

Програма є інструментом для студента і дає змогу працювати з розкладом іспитів на навчальному курсі, а також з оцінками, отриманими на іспитах. Для простоти використання прикладу, у програму від початку закладено розклад із трьох іспитів і оцінки за ними. Є можливість перегляду підсумкових оцінок у вигляді списку або «плитки», перегляду розкладу іспитів у табличному поданні, а також суміщеної інформації — і розкладу, і оцінок на одній сторінці.

Скопіюйте файл із кодом прикладу («*js12_1.html*») на свій комп'ютер і відкрийте його в браузері. Зовнішній вигляд сторінки

має відповідати наведеному нижче малюнку:

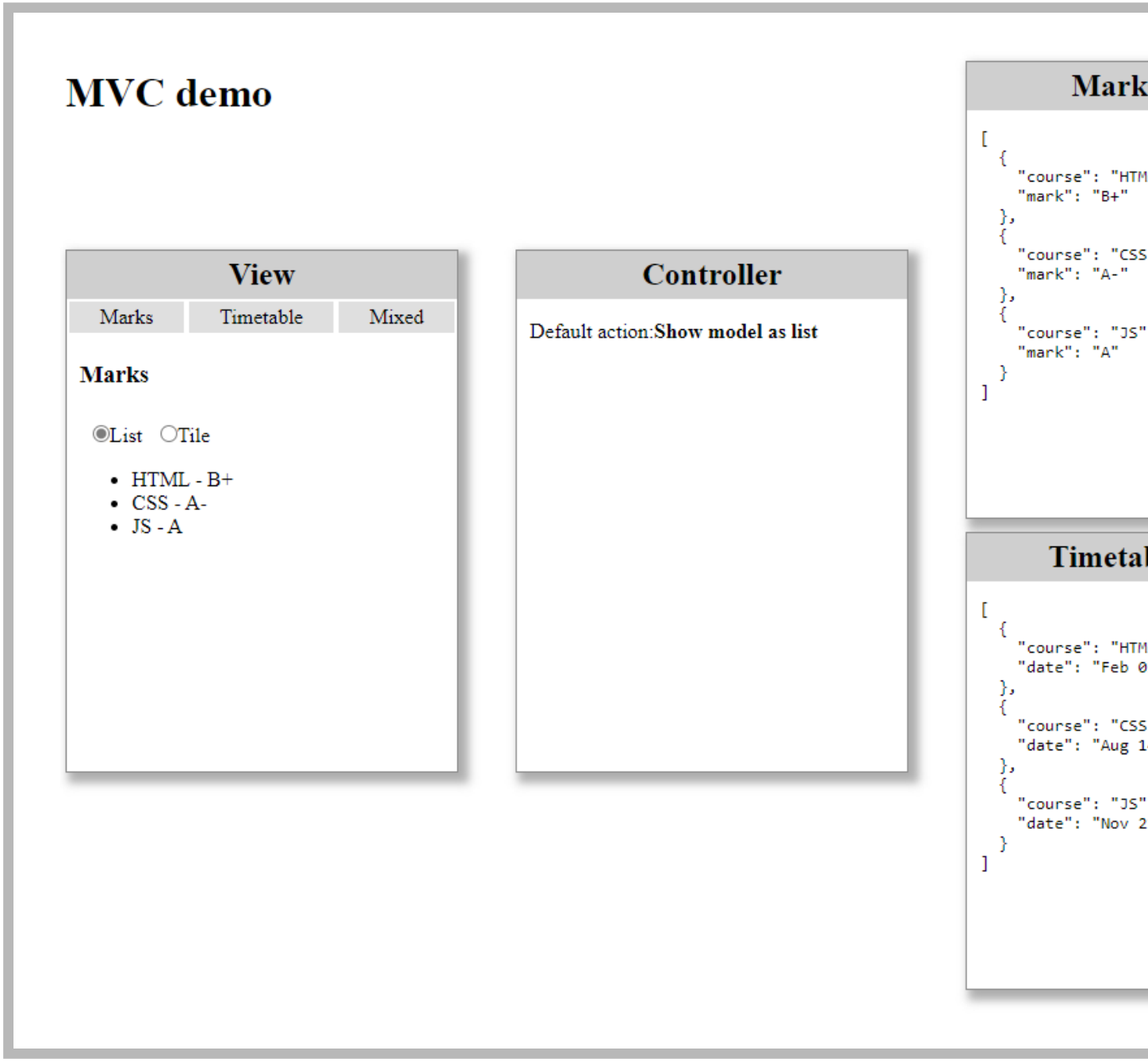


Рисунок 2. Зовнішній вигляд демонстраційної програми

У звичайних програмах користувач бачить тільки Подання, однак, у нашому прикладі з демонстраційною метою відображаються також блоки, що ілюструють роботу Контролера і Моделі. Тому блок View слід сприймати як інтерфейс користувача, а решту блоків — як підказку про невидимі частини.

Одразу уточнимо одну особливість — ця програма імітує патерн, а не є типовим його представником. Для створення повноцінної MVC програми потрібен великий програмний комплекс із базою даних, бекендом і фронтендом. У нашому випадку ми представимо роботу всіх частин у вигляді окре-

мих об'єктів JavaScript, однак, будемо повністю дотримуватися цілей і завдань кожної з частин програми, а також принципів передачі даних між ними.

Як видно з малюнка, склад прикладу містить комплекс із таких елементів:

- двох моделей, що забезпечують роботу з даними про розклад іспитів (*Timetable Model*) та отримані оцінки (*Marks Model*). Моделі мають початкові тестові дані, відображені на інтерфейсі програми
- трьох подань (*View*), що відображають оцінки (*Marks*), розклад іспитів (*Timetable*) і комплексні дані про розклад і оцінки (*Mixed*). Подання перемикаються натисканням на відповідні елементи в блоці View. При цьому подання Marks саме по собі містить варіативність відображення оцінок у вигляді списку (*List*) і у вигляді блоків-плиток (*Tile*). Їх перемикання здійснюється радіокнопками
- Контролера, який керує передачею даних між моделями і поданнями, а також веде журнал (лог) своїх дій. Під час відкриття сторінки в браузері має з'явитися напис «*Default action:Show model as list*», що означає успішне початкове завантаження всіх даних.

Розглянемо, як цей комплекс працює. Моделі представлені двома класами: «MarksModel» і «TimetableModel». Їхній склад схожий, тому розглянемо докладніше один із них:

```
class MarksModel {
    constructor() {
        this.marks = [
            { course:"HTML", mark:"B+" },
            { course:"CSS", mark:"A-" },
            { course:"JS", mark:"A" }
        ] ;
        this.out = document.querySelector(".marks-model pre") ;
```

```
        this.updateDemoBlock() ;
    }
    getMarks() {
        return this.marks ;
    }
    addMark( data ) {
        this.marks.push( data ) ;
        this.updateDemoBlock() ;
    }

// Next method is only for Example purpose
    updateDemoBlock() {
        this.out.innerText = JSON.stringify( this.getMarks(), n
    }
}
```

Нагадаємо, модель — це модуль, що забезпечує взаємодію з даними програми. Відповідно, основними методами моделі є ті, які маніпулюють з даними. У нашому прикладі це «getMarks» і «addMark». Перший метод «getMarks» метод повертає всі оцінки (marks), які містяться в даних програми. Другий метод «addMark(data)» додає до наявних оцінок нову, дані про яку передаються через параметр «data».

Конструктор класу моделі імітує взаємодію з реальними даними шляхом заповнення масиву оцінок деякими демонстраційними даними. У реальних моделях ці дані мають бути отримані з інформаційного центру програми — бекенда.

Додатковий метод «updateDemoBlock» призначений для того, щоб відображати всі дані моделі в демонстраційному блоці подання «Marks Model» (див. рисунок 1). Наприклад, цей метод викликається після кожного додавання оцінки методом «addMark». Це призведе до оновлення даних на сторінці з кожною зміною моделі.

У реальних моделях задачі оновлення демонстраційних блоків немає, тому подібних методів у них не реалізують — усі задачі відображення мають вирішуватися в Погляді (*View*). Однак у нашому прикладі Подання — це окремий блок і в нього є свої завдання. Для того щоб не домішувати до нього відображення блоків-моделей, кожна з моделей реалізує свою демонстрацію самостійно, відзначаючи цей факт коментарем перед методом.

Друга модель «TimetableModel» відповідає за дані про розклад, її склад практично такий самий, тільки назви методів «getTimetable()» і «addTimetable(data)» відрізняються, згідно з назвою моделі.

Подання також реалізовано у вигляді класу.

```
class View {
    constructor() {
        this.out = document.querySelector("#view-content") ;
        this.marksTemplate = '<h3>Marks</h3>
        <label><input type="radio" name="disp" id="radio-list"
        <label><input type="radio" name="disp" id="radio-tile"
        {{body}}';
        this.timetableTemplate = '<h3>Timetable</h3>{{body}}';
        this.mixedTemplate = '<h3>Mixed view</h3>{{body}}
        <div><br/>
            <label><input type="text" name="add-course" placeho
            <label><input type="date" name="add-date" /></label
            <label><input type="text" name="add-mark" placehold
            <input type="button" value="Add mark" onclick="wind
        </div>';
    }
    showMarksList( data ) {
        var body = "<ul>" ;
        for( let record of data ) {
```

```
            body += '<li>${record.course} - ${record.mark}</li>
        }
        body += "</ul>" ;
        this.out.innerHTML = this.marksTemplate.replace('{{body
        document.querySelector("#radio-list").checked = true;
    }
    showMarksTile( data ) {
        var body = "<div>" ;
        for( let record of data ) {
            body += '<div class='tile'>${record.course} <br/> $
        }
        body += "</div>" ;
        this.out.innerHTML = this.marksTemplate.replace('{{body
        document.querySelector("#radio-tile").checked = true;
    }
    showTimetable( data ) {
        var body = "<table class='timetable'><tr><th>Course</th
        for( let record of data ) {
            body += '<tr><td>${record.course}</td><td>${record.
        }
        body += "</table>" ;
        this.out.innerHTML = this.timetableTemplate.replace('{{
    }
    showMixed( data ) {
        var body = "<table class='timetable'><tr><th>Course</th
        for( let record of data ) {
            body += '<tr><td>${record.course}</td><td>${record.
        }
        body += "</table>" ;
        this.out.innerHTML = this.mixedTemplate.replace('{{body
    }
}
```

Методи цього класу відповідають за різні форми подання: «showMarksList(data)» показує оцінки у вигляді списку, «showMarksTile(data)» також показує оцінки, але у вигляді плиток, «showTimetable(data)» показує розклад іспитів і «showMixed(data)» показує суміщені дані (розклад та оцінки), а також дає змогу додати інформацію про нові позиції розкладу та оцінок.

Зверніть увагу, кожен із методів приймає параметр «data». На перший погляд це може здаватися зайвим, адже кожен із методів міг би звернутися до «своєї» моделі (або до кількох моделей) і отримати необхідні дані. Однак ми повинні пам'ятати, що пряма взаємодія Подання і Моделі не допускається, тільки через Контролер. У нашому простому прикладі всі модулі програми знаходяться поруч і, дійсно, можуть взаємодіяти, але ми намагаємося реалізувати реальну схему обміну даними. А в реальній схемі Подання не може звертатися до Моделі, тому приймає дані як параметр, а підготовку цього параметра буде здійснювати Контролер.

У конструкторі класу створюються шаблони (*templates*) окремо для кожного з уявлень, що перемикаються: «marksTemplate», «timetableTemplate» і «mixedTemplate». Суть роботи Подання можна уявити як заповнення заздалегідь підготовлених шаблонів розмітки даними, що в неї передаються.

У розмітці шаблонів використовуються відсилання до дій, наприклад, кнопка «Add mark» запускає метод «window.controller.addData()». Усі дії вказують на контролер. Цим ще раз підкреслимо, що Подання взаємодіє тільки з Контролером як у питаннях отримання даних, так і для завдань із додавання.

Модуль Контролера, аналогічно з попередніми модулями, є класом:

```
class Controller {
    constructor() {
        this.out = document.querySelector(".controller p") ;
    }
    defaultAction() {
        window.timetableModel.updateDemoBlock();
        window.view.showMarksList( window.marksModel.getMarks()
        this.out.innerHTML = "Default action:<b>Show model as l
    }
    marksAsList() {
        window.view.showMarksList( window.marksModel.getMarks()
        this.out.innerHTML += "User action: <b>Show model as li
    }
    marksAsTile() {
        window.view.showMarksTile( window.marksModel.getMarks()
        this.out.innerHTML += "User action: <b>Show model as ti
    }
    timetable() {
        window.view.showTimetable( window.timetableModel.getTim
        this.out.innerHTML += "User action: <b>Show timetable</
    }
    mixed() {
        consttimetable = window.timetableModel.getTimetable();
        constmarks = window.marksModel.getMarks();
        vardata = [] ;
        for( leti = 0; i < marks.length; i++ ) {
            letmark = { ...marks[i] };
            mark.date = timetable[i].date;
            data.push(mark);
        }
        window.view.showMixed( data );
        this.out.innerHTML += "User action: <b>Show mixed</b><b
    }
    addData() {
```

```
        constcourse = document.querySelector("input[name='add-c
        constmark = document.querySelector("input[name='add-mar
        constdate = document.querySelector("input[name='add-dat
        window.marksModel.addMark( {
            'course':course,
            'mark':mark
        } ) ;
        window.timetableModel.addTimetable( {
            'course':course,
            'date':newDate(date).toDateString().substring(4, 10
        } ) ;
        this.mixed() ;
    }
}
```

Як уже зрозуміло з аналізу попередніх класів, завданням Контролера є підготовка даних шляхом запиту їх із Моделей і передача цих даних у Подання для відображення. Методи класу контролера відповідають за різні варіанти роботи всієї програми: «marksAsList()», «marksAsTile()» — відображають представлення оцінок у різних видах, «timetable()» — представляє розклад, «mixed()» — поєднані дані, «defaultAction()» — реалізує початкове відображення, що буде показане під час початкового завантаження сторінки. Ще один метод «addMark()» відповідає за зворотний потік даних — з Подання в Модель. У цьому методі забезпечується розділення комплексних даних, отриманих зі змішаного (mixed) подання, і поміщення їх у різні моделі. Додатково, кожен із методів контролера оновлює блок «Controller» на демонстраційній сторінці, додаючи до нього інформацію про метод, що спрацював.

Решта вмісту файлу «*js12_1.html*» відповідає за розмітку демонстраційних блоків і їх стилізацію. Усі ці теми ми проходили раніше, тому тут не будемо наводити детальні поясне-

ння. Єдине, що потрібно ще зазначити, це створення об'єктів моделей і збереження їх в об'єкті «window» для можливості подальшого використання:

```
document.addEventListener( "DOMContentLoaded", () => {
    window.marksModel = newMarksModel();
    window.timetableModel = newTimetableModel();
    window.view = newView();
    window.controller = newController();
    window.controller.defaultAction() ;
});
```

Відкрийте файл «*js12_1.html*» у браузері, послідовно виконайте такі дії:

- натисніть радіокнопку «Tile»
- натисніть вкладку «Timetable»
- натисніть вкладку «mixed»
- у нижній частині вкладки додайте дані, наприклад, курс — «MVC», дата — «15.01», оцінка «B», натисніть кнопку «Add mark»

Зовнішній вигляд сторінки зрештою має бути схожий на такий рисунок:

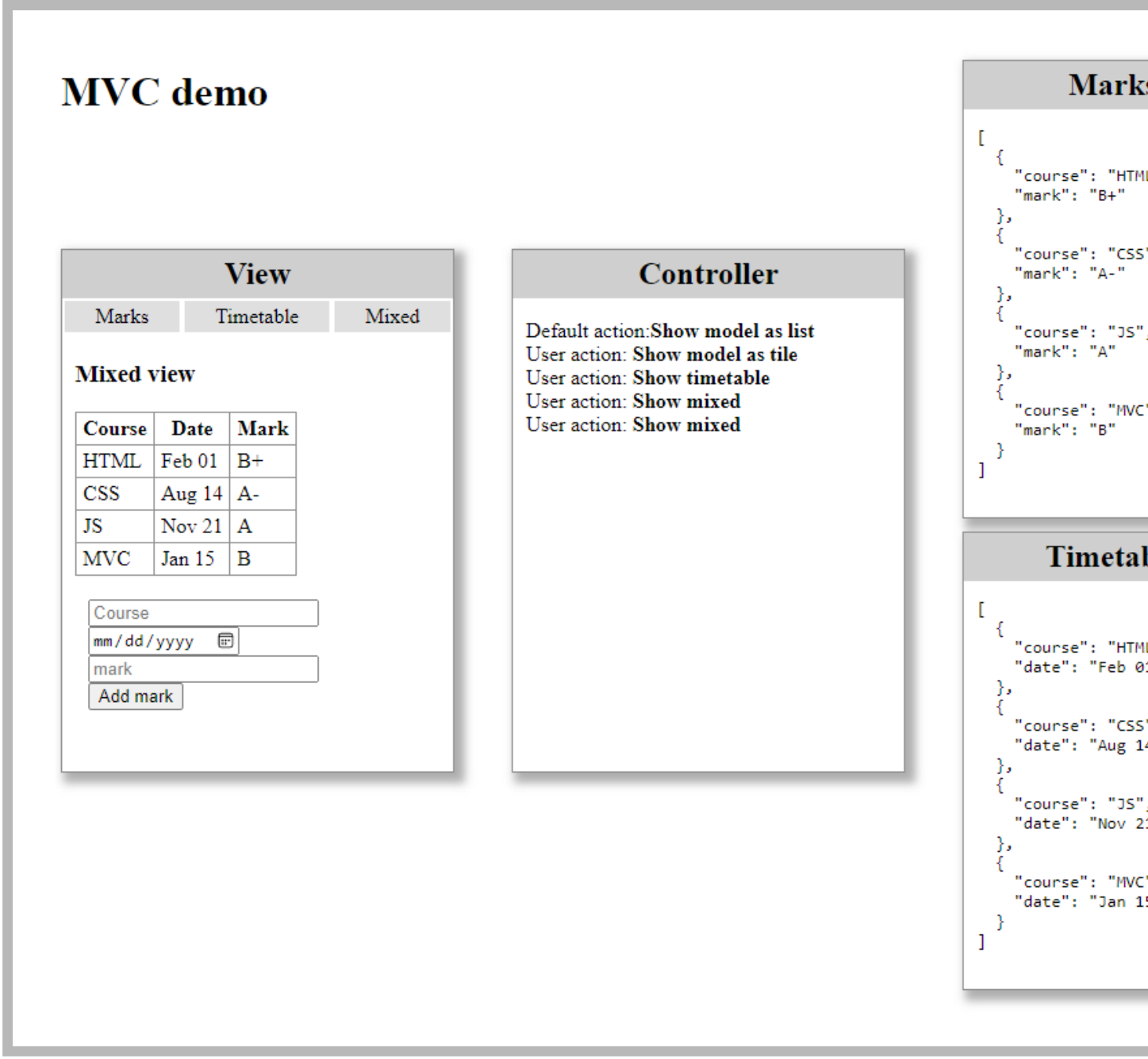


Рисунок 3. Інтерфейс після виконаних дій

Розглянемо докладніше, що відбувається в процесі кожної з виконаних дій.

Натискання на радіокнопку «Плитка»:

- запускає метод «window.controller.marksAsTile()», який є специфічним для об'єкта контролера (на цей метод посилаються в шаблоні перегляду «marksTemplate»).
- контролер запитує дані з моделі «window.marksModel.getMarks()» і передає їх методу перегляду «window.view.showMarksTile».
- у цьому методі вид генерує дані про бали у вигляді блоків « » циклічно доданих до змінної «body». Після дода-

вання всіх оцінок дані вставляються в шаблон «marksTemplate» рядком, замінюючи частину шаблону «{{body}}»:

```
this.out.innerHTML = this.marksTemplate.replace('{{body}}',
```

Отримані дані поміщаються в блок «this.out.innerHTML», який відповідає за вміст перегляду View.

- В результаті блок «Перегляд» набуває вигляду:

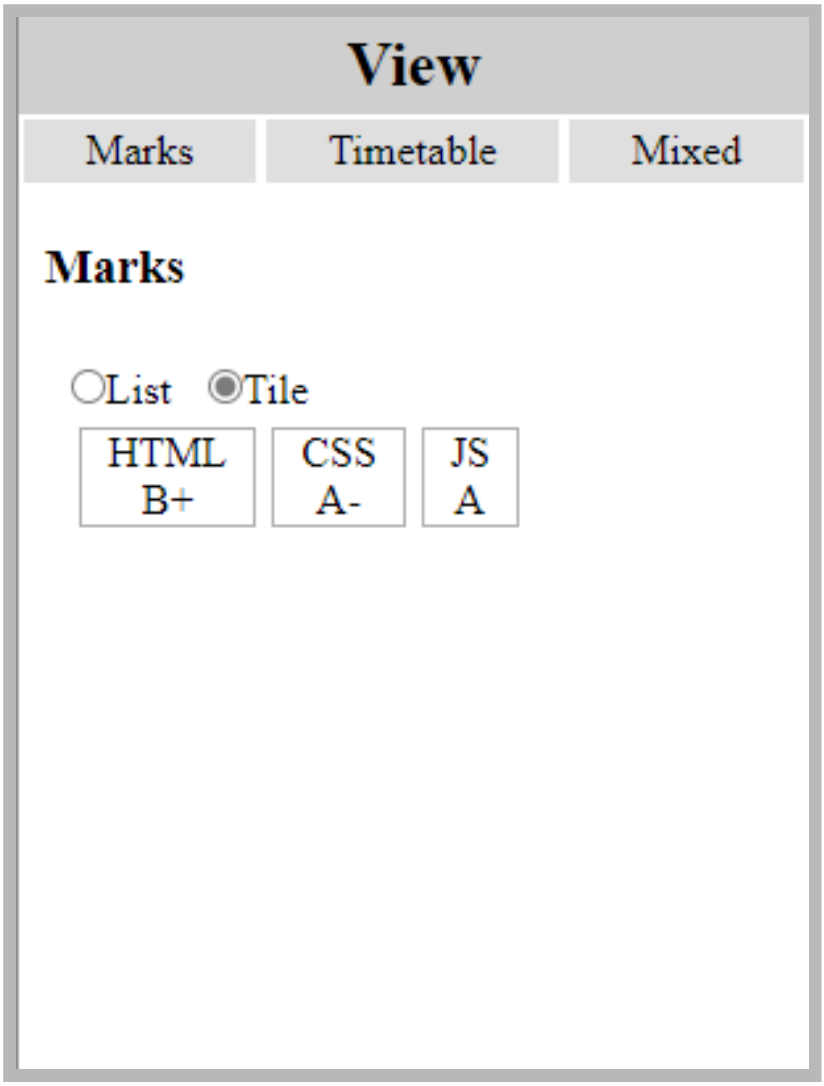


Рисунок 4. Вигляд плитки оцінок

Аналогічні дії відбуваються і під час перемикання інших подань: спочатку викликаються методи контролера, вони збирають дані з моделей і передають їх у методи подання, а вони вже, зі свого боку, заповнюють свої шаблони і вставляють їх у HTML блоку «View».

Розглянемо зворотний процес — додавання нових даних у моделі. Нагадаємо, на вкладці «mixed» ми ввели дані в поля введення (курс — «MVC», дата — «15.01», оцінка «B») і натиснули кнопку «Add mark».

- запускається метод «window.controller.addData()» контролера (посилання на метод у шаблоні подання)
- у методі «addData()» контролера визначаються змінні «course», «mark» і «date» з полів введення даних
- формується об'єкт для моделі «marksModel» і викликається метод "" у цій моделі

```
window.marksModel.addMark( {
    'course':course,
    'mark':mark
} ) ;
```

- аналогічно додаються дані в модель «imetableModel».
- після додавання викликається метод «this.mixed()», що оновлює змішане подання. Оновлення вікон моделей відбуваються автоматично їхніми власними методами «updateDemoBlock()», що викликаються під час додавання даних до моделей.
- у підсумку всі вікна оновлюються, і ми бачимо нові дані в моделях і поданні, а також у журналі контролера.

На цьому первинне знайомство з патерном MVC можна вважати завершеним. Підбиваючи деякі підсумки, можна зазначити, що патерн дає змогу розділяти програмний комплекс на кілька відносно самостійних частин, що значно спрощує командну роботу, а також систематизує потоки даних.

Як і більшість інших хороших шаблонів, патерн MVC є доволі абстрактним, що не надає чітких схем побудови своїх структурних частин або протоколів їхньої взаємодії. Усе, що можна сказати про патерн MVC, даючи йому загальне визначе-

ння, то це те, що він виділяє в структурі програми три складові частини, зазначені в назві шаблону. Більш докладно сенс цих частин ми розглянули в уроці, на завершення розглянемо кілька популярних запитань щодо цього патерну.

Чи означає це, що MVC-програма повністю складається тільки з цих трьох частин? Не обов'язково. У програмі можуть бути й інші структурні частини, головне, щоб серед них були три обов'язкових: Модель, Подання та Контролер. Також структурний патерн MVC не накладає обмежень ні на поведінкові, ні на породжувальні алгоритми програми, лише на її структуру.

Чи є ці структурні частини строго визначеними за своїм складом? Ні, у програмі може бути (і зазвичай є, як ми побачили в уроці) кілька моделей, кілька уявлень і кілька контролерів. Причому всі з них можуть бути присутніми в різних кількостях. Кожна з частин може мати власну внутрішню структуру, ба більше, наприклад, різні контролери можуть досить сильно відрізнятися один від одного за внутрішнім устроєм і виконуваними завданнями. Те саме можна сказати і про інші частини шаблону.

Чи існують різновиди патерну MVC? Так, у широкому сенсі слова термін MVC є збірним для кількох різновидів шаблонів, найвідомішими з яких є MVP (*Model-View-Presenter*) і MVVM (*Model-View-Viewmodel*). Як видно з назви, у цих патернах відрізняється частина, що відповідає за функції контролера, однак, ширший термін «контролер» цілком може замінити будь-який із них.

І навпаки, у вузькому сенсі термін MVC використовується в назвах певних програмних продуктів, наприклад, «ASP.NET MVC» або «Spring MVC». Ці продукти є реалізаціями патерну MVC, водночас самі по собі являють собою шаблони для створення програм. Наявність таких продуктів може призво-

дити до певної плутанини в термінології, оскільки через велику популярність описаних вище продуктів, їхні внутрішні терміни починають сприйматися як універсальні, хоча, це не зовсім вірно. Наприклад, те, що є «моделлю» в ASP, не слід вважати універсальним визначенням для всіх інших реалізацій патернів.

Чи потрібно прагнути реалізувати патерн MVC у кожному новому проєкті?

Скоріше ні, на малих проєктах цей патерн не демонструє всіх своїх переваг. Як і більшість патернів проєктування, MVC краще підходить для великих проєктів або, хоча б, середніх. Можливо, ви вже звернули увагу на те, що в розглянутому нами прикладі низка дій викликає питання про доцільність. Складається враження, що деякі дії можна зробити простіше і коротше. Це і є наслідком того, що структурний шаблон реалізується в малому проєкті. Будемо сподіватися, що незабаром ви, як підготовлені фахівці, візьметеся до масштабніших розробок.

Завдання для самостійної роботи.

Метод «addData()» контролера розділяє дані і додає їх у моделі, але не забезпечує їхню перевірку. Реалізуйте перевірку даних за такими правилами:

- якщо немає даних про назву предмета або дату його іспиту, то жодні моделі не оновлюються, дані про оцінку (якщо вони є) ігноруються;
- якщо є дані про предмет і дату, але немає даних про оцінку, то оновлюється тільки модель розкладу, модель оцінок не змінюється;
- якщо є всі дані, то оновлюються обидві моделі.

Домашнє завдання

Напевно ви помітили, що наведений код має невеликий недолік: метод контролера «addData()» безпосередньо звертається до елементів подання. Такий підхід, зокрема, не дає можливості провести первинну перевірку даних у Поданні перед передачею їх у Контролер. Покращена схема роботи має забезпечити таку взаємодію, за якої між модулями програми передаються тільки дані, а елементи модулів, як-от поля введення, не мають бути доступні з іншого модуля. Тому завдання полягає в такому:

- у Погляді (клас View) потрібно створити метод для збору даних, що вводяться (наприклад, з тією самою назвою «addData()», але можна вибрати й інше);
- в Контролері необхідно змінити сигнатуру методу, вказавши, що дані будуть передані ззовні «addData(data)»;
- переключити обробник натискання кнопки «Add mark» на метод з Подання (це вказано в шаблоні «mixedTemplate» у конструкторі класу View);
- метод «addData()» подання перенести з контролера коди отримання даних з полів введення, а також реалізувати їх первинну перевірку (на порожнечу), якщо перевірка не завершується успіхом, то слід видавати повідомлення (можна методом alert);
- у разі успішної первинної перевірки дані мають бути передані до контролера і в ньому вже розподілені за моделями.