**Лабораторная работа №2**

**Проверка на стороне клиента**

**Цель работы** - получение практических навыков организации проверки на стороне клиента.

**Теоретические сведения**

Даже если источником истины является модель предметной области и в конечном итоге необходимо проводить проверки на этом уровне, проверку можно выполнить как на уровне модели предметной области (на сервере), так и на уровне интерфейса пользователя (на клиенте).

Проверка на стороне клиента очень удобна для пользователей.

Она экономит время, которое в противном случае тратилось бы на круговой путь к серверу, в результате которого выдавались бы ошибки проверки. С точки зрения бизнеса даже доли секунды, умножаемые в сотни раз каждый день, позволяют значительно сократить расходуемое время, деньги и усилия. Простая и немедленная проверка позволяет пользователям работать эффективнее и повышает точность входных и выходных данных.

Так же как модель представления отличается от модели предметной области, проверка модели представления и проверка модели предметной области имеют сходные черты, но выполняют разные функции. Если вы беспокоитесь о принципе DRY («Не повторяйся»), то в этом случае повторное использование кода может означать взаимозависимость, а в корпоративном приложении взаимозависимость стороны сервера и стороны клиента гораздо хуже, чем нарушение принципа «Не повторяйся».

Даже при проведении проверки на стороне клиента следует всегда проверять команды или входящие объекты переноса данных в серверном коде, ведь API сервера являются возможным вектором атаки. Как правило, рекомендуется выполнять обе проверки, поскольку, с точки зрения взаимодействия с пользователем, в клиентском приложении лучше действовать на опережение и не допускать ввод пользователем недопустимых данных.

Поэтому в коде на стороне клиенты вы обычно проверяете модели представления. Можно также проверять исходящие объекты переноса данных или команды клиента, прежде чем отправлять их службам.

Выполнение проверки на стороне клиента зависит от типа создаваемого клиентского приложения.

Принцип DRY (от английского «Don't Repeat Yourself» – не повторяйся) является одним из основополагающих принципов разработки. Например, в модели MVC (паттерн программирования, разделяющий архитектуру приложения на три модуля: модель, представление и контроллер -­ model, view, controller) рекомендуется задавать функциональные возможности или поведение только один раз, а затем отражать их в других местах в приложении. Это позволяет свести к минимуму объем кода, а также снизить риск возникновения в нем ошибок и упростить его тестирование и поддержку. Правила проверки декларативно определяются в одном месте (в классе модели) и затем применяются в рамках всего приложения.

Пример  
Добавление правил проверки к модели фильма (файл Movie.cs). Класс предоставляет набор встроенных атрибутов проверки, которые декларативно применяются к любому классу или свойству. Кроме того, этот класс содержит атрибуты форматирования, такие как DataType, которые обеспечивают форматирование и не предназначены для проверки.

Требуется обновить класс, чтобы использовать преимущества встроенных атрибутов проверки Required, StringLength, RegularExpression и Range.

public class Movie

{

public int Id { get; set; }

[StringLength(60, MinimumLength = 3)]

[Required]

public string Title { get; set; }

[Display(Name = «Release Date»)]

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

[Column(TypeName = «decimal(18, 2)»)]

public decimal Price { get; set; }

[RegularExpression(@»^[A-Z]+[a-zA-Z»«'\s-]\*$»)]

[Required]

[StringLength(30)]

public string Genre { get; set; }

[RegularExpression(@»^[A-Z]+[a-zA-Z0-9»«'\s-]\*$»)]

[StringLength(5)]

[Required]

public string Rating { get; set; }

}

Атрибуты проверки определяют поведение для свойств модели, к которым они применяются: атрибуты Required и MinimumLength указывают, что свойство должно иметь значение. Тем не менее, чтобы удовлетворить требованиям проверки, пользователю достаточно ввести пробел. Атрибут RegularExpression ограничивает набор допустимых для ввода символов. В приведенном выше коде в полях Genre и Rating можно использовать только буквы (заглавная первая буква, пробелы, числа и специальные символы не допускаются). Атрибут Range ограничивает диапазон значений. Атрибут StringLength позволяет задать максимальную и при необходимости минимальную длину строкового свойства: «The StringLength attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as decimal, int, float, DateTime) are inherently required and don't need the [Required] attribute». Наличие правил проверки помогает повысить степень надежности приложения. Это также гарантирует, что в любом случае будут выполнены все проверки и в базе данных не будут случайно оставлены поврежденные данные.

Пользовательский интерфейс проверки ошибок в модели MVC

Если запустить приложение и ввести в форму какие-либо недопустимые значения, то функция проверки jQuery на стороне клиента обнаружит ошибку, сведения о которой отобразятся в соответствующем сообщении.

Для каждого поля, содержащего недопустимое значение, в форме автоматически отображается соответствующее сообщение об ошибке проверки. Эти ошибки применяются как на стороне клиента (с помощью JavaScript и jQuery), так и на стороне сервера (если пользователь отключает JavaScript).

Серьезное преимущество заключается в том, что для реализации этого пользовательского интерфейса проверки не требуется изменять код в классе MoviesController или представлении Create. В контроллере и представлениях, создаваемых в рамках этого руководства, автоматически применяются правила проверки, для определения которых к свойствам класса модели Movie были применены атрибуты. При проверке с использованием метода действия Edit применяются те же правила.

Данные формы передаются на сервер только после того, как будут устранены все ошибки проверки на стороне клиента.

Принципы работы проверки. Вам может быть интересно, как пользовательский интерфейс проверки создается без обновления кода контроллера или представлений. В следующем примере кода показаны два метода Create.Create

// GET: Movies/Create

public IActionResult Create()

{

return View();

}

// POST: Movies/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind(«ID,Title,ReleaseDate,Genre,Price, Rating»)]Movie movie)

{

if (ModelState.IsValid)

{

\_context.Add(movie);

await \_context.SaveChangesAsync();

return RedirectToAction(«Index»);

}

return View(movie);

}

Первый метод действия Create (HTTP GET) отображает исходную форму создания. Вторая версия ([HttpPost]) обрабатывает передачу формы. Второй метод Create (версия [HttpPost]) вызывает ModelState.IsValid, который определяет наличие ошибок проверки в фильме. При вызове этого метода оцениваются все атрибуты проверки, которые были применены к объекту: «Calling this method evaluates any validation attributes that have been applied to the object».

При наличии ошибок проверки в объекте метод Create повторно отображает форму. Если ошибок нет, метод сохраняет новый фильм в базе данных. В этом примере форма передается на сервер только после того, как будут устранены все ошибки проверки, обнаруженные на стороне клиента. Второй метод Create не вызывается до тех пор, пока на стороне клиента присутствуют ошибки проверки. При отключении JavaScript в браузере также отключается проверка на стороне клиента.

**Задание:**

1. Найти и развернуть приложение (можно в архитектуре MVC), обеспечивающее ввод данных о некотором объекте.
2. Создайте проверку вводимых данных на стороне клиента.
3. Проверьте правильность ввода данных с помощью отрицательного и положительного теста.
4. Полученные результаты оформить в виде отчета.

**Контрольные вопросы:**

1. Что из себя представляет архитектура MVC?
2. Как реализуются проверка на стороне клиента?
3. Что включается в описание операции?
4. Почему при отключении JavaScript отключается проверка на стороне клиента?