Урок 5

Django — AJAX, JavaScript, jQuery

Особенности использования фреймворка в привязке к технологии AJAX, языку программирования JavaScript и библиотеке jQuery.

AJAX, JavaScript и jQuery

- ***Что представляет собой каждое из понятий: AJAX, JavaScript и jQuery?***

AJAX — это не приложение или библиотека, которые можно установить. Это подход, который используется при проектировании пользовательских интерфейсов веб-приложений и подразумевает «фоновый» обмен данными браузера (клиентской стороны) с веб-сервером. При использовании **AJAX** обновление данных не вызывает полную перезагрузку веб-страницы. Повышается скорость работы веб-приложений, их удобнее применять. Благодаря технологии **AJAX** веб-приложения становятся похожи на десктопные программные продукты, в которых перезагрузка страниц не происходит. Одним из классических примеров использования **AJAX** является онлайн-чат. Пользователи могут обмениваться сообщениями, не покидая свои текущие веб-страницы.

В приложениях на основе AJAX-технологии загружается только изменившаяся часть веб-страницы или принимается/передается набор данных в JSON- или XML-формате. Затем содержимое страницы меняется посредством **JavaScript**.

Чтобы понять принцип работы AJAX, сравним его с традиционным подходом к построению веб-приложений.

Традиционный подход:

1. Пользователь переходит по ссылке на веб-странице.
2. При этом браузер формирует и отправляет запрос на веб-сервер.
3. Далее веб-сервер генерирует новую страницу и передает ее браузеру, который выполняет полную перезагрузку веб-страницы.

Подход AJAX:

1. Пользователь переходит по ссылке на веб-странице.
2. JavaScript-сценарий определяет информацию, необходимую для обновления страницы.
3. Браузер передает соответствующий запрос на сторону веб-сервера.
4. Далее сервер возвращает только ту часть документа, на которую пришел запрос.
5. JavaScript-сценарий выполняет изменения с учетом полученной информации (при этом не происходит полная перезагрузка веб-страницы).

JavaScript — это язык программирования, который используется для написания ПО, выполняемого на стороне клиента и реализующего интерактивность веб-страниц в браузере. Веб-страница анализируется браузером как документ с древовидной структурой, содержащий все элементы страницы. В этой связи принято говорить об объектной модели документа (Document Object Model — DOM). Благодаря JavaScript разработчик может получать доступ к этим элементам и управлять ими.

Доступ к элементам — через связанные с ними id или имена тегов. Для этого используются методы **getElementsById** и **getElementsByTagName**. Обратиться к элементу страницы можно и с помощью операторов перемещения по дереву объектов страницы (**parentNode**, **firstChild**, **lastChild** и т.д.). После получения доступа к элементу можно изменить его положение на странице, содержимое, внешний вид и другие параметры.

jQuery — это библиотека, основанная на JavaScript. Она упрощает ряд задач: манипулирование DOM, добавление анимаций, управление стилизацией веб-страниц и реализацию асинхронности (AJAX).

jQuery — одна из наиболее популярных библиотек для клиентской стороны. Она фокусируется на взаимодействии JavaScript, HTML и CSS и упрощает разработку клиентской части сайта.

jQuery — не единственная в своем роде. Существуют другие фреймворки, основанные на JavaScript, — например, **AngularJS**, **NodeJS**, **BackboneJS**. Они упрощают реализацию сложной интерактивности или помогают решить такие задачи, которые на чистом JavaScript практически невыполнимы.

Преимущества jQuery

- ***В чем преимущества библиотеки jQuery?***

По сравнению с традиционным подходом программирования на «чистом» JavaScript — значительно уменьшается количество программного кода, необходимого для выполнения скрипта. Это снижает временные затраты на разработку и повышает читабельность кода.

1. Проще осваивать jQuery, чем изучать особенности программирования на «чистом» JavaScript.
2. Проще работать с объектной моделью документа и легче получить доступ к множеству эффектов для элементов дерева объектной модели.
3. Проще организовывать работу приложений на базе технологии AJAX, реализовывать асинхронное получение и отправку данных.
4. Можно реализовать много сложных моментов, связанных с кроссбраузерной совместимостью разрабатываемых приложений.
5. Есть мощная база плагинов, обеспечивающих реализацию в веб-приложении функционала любой сложности.
6. Синтаксис jQuery понятен даже для начинающих разработчиков, его можно быстро изучить.
7. Удобная документация, упрощающая освоение библиотеки, а также активное интернет-сообщество, готовое оказать поддержку по любым вопросам.

Преимущества и недостатки AJAX

- ***В чем преимущества и недостатки AJAX?***

Преимущества:

1. **Экономия трафика.** Поскольку при использовании AJAX перезагрузка всей веб-страницы не требуется, значительно сокращается трафик при работе с приложением.
2. **Снижение нагрузки на сервер.** Правильная реализация AJAX-механизма может в несколько раз снизить нагрузку на сервер.
3. **Ускорение «ответа» интерфейса.** Так как загрузка только изменившейся части документа значительно быстрее, чем перезагрузка всей страницы, то перед пользователем быстрее отображаются результаты действий. И без мерцания страницы, как в случае с полной перезагрузкой.
4. **Широкие возможности для интерактивной обработки.** Технология AJAX позволяет дополнительно реализовать в приложении полезные опции, —например, вывод подсказок при указании данных, проверку доступности имени при регистрации пользователя. AJAX — это

идеальный механизм для разработки чатов и панелей, в которых регулярно обновляются данные.

Недостатки:

1. **Требование включенного JavaScript.** Поддержка этого языка программирования может отсутствовать в некоторых браузерах — в частности, в старых версиях. Кроме того, JavaScript может быть отключен в целях безопасности. Это приводит к тому, что все AJAX-модули приложения становятся недоступными.

2. **Нет взаимосвязи со стандартными возможностями браузера.** Для динамически создаваемых страниц не поддерживается регистрация в списке посещений, поэтому у пользователя нет возможности вернуться к просмотренным ранее страницам. Url-адрес в данном случае является постоянным, поэтому пользователь не может сохранять закладки.

3. **Проблема фиксации просмотра новых страниц.** Исходя из специфики работы технологии AJAX, теряет актуальность статистика просмотра новых страниц сайта, формируемая различными сервисами.

4. **Сложности отображения нестандартных кодировок в ряде AJAX-сценариев.** Эту проблему много обсуждают в интернете.

5. **Замедление обновления страницы при большом количестве AJAX-запросов.** Если таких запросов на одной странице становится слишком много, скорость ее обновления может снижаться по сравнению с традиционным подходом (без AJAX).

Подключение jQuery и JavaScript к шаблону Django-проекта

- ***Как подключить библиотеку jQuery и JavaScript-сценарий к html-шаблону Django-проекта?***

На практике чтобы подключить файлы сценариев к html-шаблонам Django-проекта создают в базовом шаблоне блока такую конструкцию:

```
{% block additional_js %}
{% endblock %}
```

Аналогичный блок, но с содержимым, необходимо создать в шаблоне-наследнике и указать в нем набор инструкций, которые будут динамически подгружаться в базовый шаблон. В качестве инструкций указываются привязки веб-сценариев: привязка файла **jquery.js** с кодом библиотеки jQuery и привязка файла с JavaScript-сценарием для реализации клиентской функциональности. Пример блока добавления привязок веб-сценариев:

```
{% block additional_js %}
    <!--подключаем код библиотеки JQuery-->
    <script src="{% static "js/jquery.js" %}" type="text/javascript"></script>
    <!--подключаем код, реализующий клиентскую функциональность-->
    <script src="{% static "js/imagepool.js"
    type="text/javascript" %}"></script>
{% endblock %}
```

Тег **static** отвечает за генерацию url-адреса по указанному относительному пути (относительно каталога **static** проекта). Файлы веб-сценариев принято размещать в дополнительном вложенном каталоге **js**.

Для корректной работы статики необходимо добавить инструкцию загрузки хранилища статических файлов:

```
{% load staticfiles %}
```

Выполнение кода с помощью jQuery

- ***Как обеспечить выполнение кода с помощью библиотеки jQuery?***

Для выполнения скрипта посредством библиотеки jQuery необходимо поместить весь код скрипта в специальную функцию, которая выполняется при полной загрузке объектной модели документа (DOM). Готовность модели DOM определяется методом **.ready()**.

Пример функции для выполнения скрипта в jQuery:

```
$(document).ready(function() {
    // Код обработчика для .ready()
});
```

Как видно из фрагмента кода, мы используем привычный синтаксис языка JavaScript. В данном случае **document** является основным элементом документа в браузере. Метод **.ready()** соответствует событию, которое срабатывает при готовности выбранного элемента **document**. Далее создается сама функция (оператор **function()**), в которой вызывается код **JavaScript/jQuery**.

Рассмотрим специализированные элементы созданной функции — в частности, символ **$()**. Он соответствует селектору для выбора нужного HTML-тега для последующих манипуляций. Кроме того, символ **$()** является псевдонимом для **jQuery**.

Сравним два фрагмента кода, которые по логике выполнения идентичны, но различаются в используемых конструкциях:

```
$(document).ready(function() {
});
```

```
jQuery.ready(function() {
});
```

Конструкция **document.ready()** выполняется при загрузке модели **DOM** документа. При этом корректным подходом является размещение любого количества таких функций на одной веб-странице.

Селекторы в jQuery

- ***Что такое селекторы в jQuery? Какие типы селекторов существуют, и какие из них самые быстрые?***

Чтобы выполнять операции с элементами веб-страницы, а не с целым веб-документом, необходим механизм доступа к этим элементам. Для поиска HTML-элементов в библиотеке jQuery используются селекторы разных видов:

- **#id** — обращаемся к единственному элементу веб-страницы по заданному идентификатору (атрибут id);
- **element** — получаем доступ ко всем элементам с данным именем;
- **.class** — отбираем и выполняем операции только с элементами данного класса;
- **.class.class** — отбираем все элементы, относящиеся к перечисленным классам;
- ***** — отбираем все элементы.
- **selector1, selector2, …, selectorN** — отбираем элементы в зависимости от сочетания перечисленных селекторов.

Первые два типа селекторов в jQuery являются самыми быстрыми.

Рассмотрим примеры использования селекторов.

Выбираем все элементы с **id = help**:

```
$('#help');
```

Выбираем все элементы **span** с **id = help**:

```
$('span#help');
```

Выбираем все элементы с **class = test**:

```
$('.test');
```

Выбираем все элементы **span** с **class = test**:

```
$('span.test');
```

Выбираем все элементы **span** в элементах **div**:

```
$('div, span');
```

Выбираем предыдущий и следующий элементы от найденного:

```
$('#help').prev();
```

```
$('#help').next();
```

Конструкции window.onload, document.ready(), $(function(){...});

- ***Что представляют собой конструкции window.onload, document.ready(), $(function(){...});?***

window.onload. Метод **onload** глобального объекта **window** (окно браузера) является событием модели **DOM**. Выполняется после загрузки всего содержимого веб-страницы, в том числе изображений.

$(document).ready(). Событие, возникающее в момент готовности DOM-дерева. Ожидание загрузки изображений не требуется.

$(function(){...});.** Аналог конструкции выполнения скрипта при полной загрузке объектной модели документа: **$(document).ready(function(){ … });.

AJAX-запрос с помощью jQuery

- ***Можно ли сделать AJAX-запрос с помощью jQuery? Какие существуют методы для реализации этой задачи?***

Да. Реализация технологии AJAX при разработке веб-приложения является одной из ключевых возможностей, предлагаемых библиотекой jQuery. При этом асинхронное взаимодействие клиентского приложения и сервера можно осуществлять несколькими путями:

- **load()**. Использование данного метода — один из наиболее легких способов извлечь данные с серверной стороны. При его успешном выполнении происходит формирование HTML-кода и его размещение в нужный DOM-элемент страницы.
- **$.getJSON()**. Позволяет запросить у сервера данные в формате JSON. Это удобнее и быстрее, чем использовать XML-формат.
- **$.getScript()**. Подгружает JavaScript-файл с сервера на клиентскую страницу с помощью метода **GET** и выполняет содержащийся в нем сценарий.
- **$.get()**. Позволяет выполнить к серверу HTTP GET-запрос и загрузить данные в клиентское приложение без перезагрузки страницы.
- **$.post()**. Данный метод аналогичен предыдущему, но данные передаются на сервер без перезагрузки страницы с использованием метода **POST**.
- **$.ajax()**. Базовый метод библиотеки jQuery для реализации асинхронности. Позволяет обратиться к серверу без перезагрузки страницы. Это низкоуровневый метод, у которого множество настроек. Заложен в основу работы всех других методов обеспечения асинхронности с помощью библиотеки jQuery.

Реализация события нажатия кнопки в jQuery

- ***Как сделать, чтобы при нажатии на тег, соответствующий кнопке, выводилось сообщение? Как выполнить имитацию нажатия кнопки?***

Рассмотрим простейший программный код, с помощью которого выполнена html-разметка кнопки:

```
<p style="text-align: center"><button>Кнопка</button>
```

Теперь реализуем скрипт, который поместим в функцию, являющуюся отправной точкой выполнения скрипта в jQuery. Чтобы проверить работоспособность этого и других листингов кода в методичке, можете воспользоваться онлайн-средой разработки веб-приложений **jsFiddle**. Она позволяет осуществлять запуск и отладку программного кода, написанного на HTML, JavaScript и CSS. Также предусмотрена возможность использовать библиотеку jQuery.

```
$(document).ready(function () {
    // Обработчик нажатия на кнопку
    $('button').on('click', function() {
        alert('Кнопка нажата!');
    })
    // Имитация нажатия на кнопку
    $('button').trigger('click');
});
```

При выполнении данного скрипта после загрузки документа пользователь должен увидеть указанную надпись. Это реализация нажатия кнопки, расположенной на веб-странице. Если выполнить нажатие кнопки, надпись выводится повторно.

Проверка существования элемента средствами jQuery

- ***Как проверить в jQuery существование определенного элемента?***

Рассмотрим разметку строчного элемента веб-страницы с помощью html-тега **span**:

```
<span class="letter" style="color: red; font-size: 200%;">Простейший
текст</span>
```

Для проверки существования данного элемента веб-страницы напишем следующий код:

```
$(document).ready(function () {
    if ($('.letter').length > 0){
    alert('Тег span существует');
    };
});
```

Поскольку указанный тег существует, будет выведено соответствующее сообщение. Здесь используется jQuery-функция **length()** — она возвращает количество элементов в jQuery-объекте.

Выполнение только одного запуска обработчика

- ***Как реализовать запуск обработчика для элемента только один раз?***

Простейший пример использования html-тега **p**:

```
<p id='test_p'>Запускаем обработчик один раз</p>
```

Реализуем скрипт, который будет обеспечивать только один запуск обработчика нажатия на тег **p** и единственный вывод сообщения:

```
$(document).ready(function () {
    $('#test_p').one('click', function() {
        alert('Выводим сообщение только один раз');
    });
});
```

Способы привязки обработчиков к событиям в JavaScript

- ***Какие существуют способы привязки обработчиков к событиям в JavaScript?***

В JavaScript назначить обработчик события можно несколькими способами. Рассмотрим их применительно к событию клика мыши по html-элементу.

1. *target*.**onclick = function(event) {** *код обработчика* **}**

В данном случае обработчику в качестве аргумента передается объект события **event**. Использование этого способа подключения обработчика рассмотрим на примере html-разметки обычной кнопки:

```
<input type="button" value="Простая кнопка" id="btn">
```

Код обработчика:

```
btn.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " - тип события, " + event.currentTarget + " - элемент");
};
```

В данном случае в модальном окне выводится сообщение с типом события, на которое привязан обработчик, и элемент, к которому относится событие.

2. *target*.**addEventListener(“click”, ...)**

Позволяет к одному событию привязать любое количество обработчиков, которые выполняются в порядке указания. Рассмотрим использование данного типа подключения обработчика применительно к предыдущему примеру с кнопкой.

К событию клика данной кнопки привяжем три обработчика, каждый из которых отвечает за вывод служебного сообщения.

```
btn.addEventListener('click', first);
btn.addEventListener('click', second);
btn.addEventListener('click', third);

function first() { alert('Вызов первой функции'); }
function second() { alert('Вызов второй функции'); }
function third() { alert('Вызов третьей функции'); }
```

3. *target*.**attachEvent(“on”+имя_события, обработчик)**

Доступен только в браузере Internet Explorer версии 9 (и более ранних).

```
function my_func() {
  alert( 'Вызов обработчика!' );
}
btn.attachEvent("onclick", my_func)
```

Объекты window и document

- ***В чем разница между объектами window и document?***

В JavaScript все манипуляции с элементами **DOM** осуществляются через глобальный объект **window**, который содержит все глобальные переменные, обработчики, местоположение, историю. В нем находится и объект **document**, который является свойством объекта **window** и представляет все дерево элементов **DOM**. Все узлы (ноды) представляют собой части объекта **document**. Следовательно, для него можно применять методы **getElementById** и **addEventListener**, чтобы получить доступ к нужному элементу веб-страницы и привязать к нему обработчик. Применять эти методы к глобальному объекту **window** недопустимо — это приведет к ошибке.

Доступ к элементу из DOM

- ***Какие существуют способы доступа к элементу из DOM-дерева?***

Для выполнения этой задачи в объекте **document** предусмотрены следующие методы:

1. **getElementById**. Возвращает только один элемент, соответствующий переданному идентификатору.

html-код:

```
<span id="span_id">Простой текст</span>
```

js-код:

```
elem = document.getElementById("span_id")
alert(elem)
```

2. **getElementsByClassName**. Возвращает html-коллекцию, содержащую элементы, имеющие заданное имя класса.

html-код:

```
<span id="span_id_1" class="span_class">Простой текст 1</span>
<span id="span_id_2" class="span_class">Простой текст 2</span>
```

js-код:

```
elems = document.getElementsByClassName("span_class")
alert(elems)
```

3. **getElementsByTagName**. Метод аналогичен предыдущему, но осуществляет отбор элементов не по имени класса, а по имени html-тега.

html-код:

```
<span id="span_id_1" class="span_class">Простой текст 1</span>
<span id="span_id_2" class="span_class">Простой текст 2</span>
```

js-код:

```
elems = document.getElementsByTagName("span");
alert(elems)
```

4. **querySelector**. Возвращает первый узел документа, соответствующий указанному селектору.

html-код:

```
<span id="span_id_1" class="span_class">Простой текст 1</span>
<span id="span_id_2" class="span_class">Простой текст 2</span>
```

js-код:

```
elem = document.querySelector(".span_class");
alert(elem)
```

5. **querySelectorAll**. Возвращает список узлов документа, соответствующих указанному селектору.

html-код:

```
<span id="span_id_1" class="span_class">Простой текст 1</span>
<span id="span_id_2" class="span_class">Простой текст 2</span>
```

js-код:

```
elems = document.querySelectorAll(".span_class");
alert(elems)
```

6. **getElementsByName**. Возвращает список элементов, имя которых (атрибут **name**) совпадает с указанным. Применяется только для html-тегов, для которых по спецификации предусмотрено использование атрибута **name**.

html-код:

```
<span id="span_id_1" class="span_class" name="span">Простой текст 1</span>
<span id="span_id_2" class="span_class" name="span">Простой текст 2</span>
```

js-код:

```
elem = document.getElementsByName("span");
alert(elem)
```

7. **getElementsByTagNameNS**. Возвращает список элементов с указанным названием тега в пространстве имен.

Синтаксис метода:

```
elems = document.getElementsByTagNameNS(имя_пространства_имен, название_тега)
```

Пример:

```
elems = document.getElementsByTagNameNS("http://www.w3.org/1999/xhtml", "span");
```

Отслеживание ошибок

- ***Как в JavaScript-сценарии организовать отслеживание ошибок?***

Для этого в JavaScript используется конструкция **try {...} catch {...}**. Рассмотрим пример с преобразованием строки в число:

```
try {
    var text = 'stroka';
    alert(text.Number());
    alert('Все в порядке!');
} catch(e) {
    alert('Ошибка!');
}
```

Свойства innerhtml и outerhtml

- ***В чем разница между свойствами innerhtml и outerhtml?***

Свойство **innerHTML** определяется для любого элемента дерева **DOM** веб-страницы и позволяет получить его содержимое. Свойство **outerHTML** возвращает полный html-код элемента веб-страницы, включая блочные теги. Рассмотрим пример:

```
<div id='div_id'>
    <span class='inner'></span>
</div>
```

Использование свойства **innerHTML**:

```
alert(document.getElementById('div_id').innerHTML)
```

Вернет результат:

```
<span class='inner'></span>
```

Использование свойства **outerHTML**:

```
alert(document.getElementById('div_id').outerHTML)
```

Вернет результат:

```
<div id='div_id'>
    <span class='inner'></span>
</div>
```

Практическое задание

Продолжим работать над каталогом товаров. На данный момент приложение действует в синхронном режиме, т.е. пользователь нажимает кнопку добавления, после чего выполняется переход на соответствующую страницу.

Усовершенствуем работу проекта: сделаем его более похожим на десктопное приложение — без всяких перезагрузок страницы. Форма должна загружаться, как модальное окно, как бы «поверх» содержимого главной страницы. После этого пользователь может указать данные нового товара и сохранить его в базе. При этом модальное окно формы должно закрываться, а изменения, т.е. строка с добавленным товаром, должна отобразиться в таблице на главной странице. Таким образом мы реализуем асинхронную работу приложения. Для этого необходимо использовать знания языка программирования JavaScript и библиотеки jQuery. Разобьем выполнение задания на несколько этапов:

Доработка шаблонов

1. **goods_list.html**

В первую очередь необходимо внести изменения в шаблон **goods_list.html**. Создадим пустую структуру вложенных блочных элементов **div** (три элемента). Ее содержимое будет динамически изменяться при нажатии кнопки добавления товара, т.е. будет загружаться модальное окно формы добавления товара. Данную структуру можно реализовать следующим образом:

```
<div class="modal fade" id="modal-good">
  <div class="modal-dialog">
    <div class="modal-content">

    </div>
  </div>
</div>
```

В данном случае первый блочный элемент **div** с идентификатором **modal-good** соответствует модальному окну формы добавления товара. Второй **div** позволяет определить стилизацию компактного окна, а третий будет содержать html-код самой формы. Он будет динамически передаваться в этот **div**, и пользователь увидит на экране подгружаемую форму. Приведенный код необходимо разместить в конце шаблона **goods_list.html**.

Далее в этом же шаблоне **goods_list.html** следует изменить разметку кнопки добавления товара: убрать из тега **button** тег ссылки **<a>** и заменить его простым тегом строки span с названием кнопки. Тегу **button** назначить два атрибута: **class** (необходим для получения доступа к элементу из файла-скрипта) и **data-url** (позволяет идентифицировать страницу, генерируемую посредством jQuery).

Разметка кнопки при этом может принять такой вид:

```
<button type="button" class="js-create-good" data-url="{% url 'good_create' %}">
```

Следующее, что необходимо сделать с файлом **goods_list.html**, — подключить скрипт на языке JavaScript, который позволит силами библиотеки jQuery реализовать асинхронное взаимодействие с сервером. При этом тег **script** надо поместить в соответствующий тег шаблона **{% block javascript %}{% endblock %}**.

Поскольку динамически у нас должна обновляться не вся главная html-страница, а лишь табличная часть со списком товаров, создадим дополнительный шаблон **partial_goods_list.html**. В него поместим шаблонный тег цикла, отвечающего за формирование таблицы с данными. При этом в файле шаблона **goods_list.html** выполним импорт файла нового шаблона **partial_goods_list.html**.

2. **base.html**

В этом файле необходимо добавить подключение библиотеки jQuery. Для локального подключения следует скачать и поместить в директорию **static/js** файл с кодом библиотеки — например, **jquery-3.3.1.min.js**. В эту же директорию следует добавить файл **bootstrap.min.js**, который позволит упростить стилизацию элементов страницы.

3. **good_create.html**

Это шаблон, в котором находится разметка формы добавления товара. Его следует модифицировать, указав в атрибуте **action** тега **form** имя url-шаблона адреса, при переходе на который запускается контроллер записи в модель введенных данных. В нашем случае это шаблон **good_create.html**.

Также в данный шаблон необходимо поместить тег импорта разметки самой формы **form.html** в блочный элемент и добавить две кнопки: сохранения данных в форме и ее закрытия.

Пример кода до изменений:

```
<form method="post" enctype="multipart/form-data">
    {% include "form.html" %}
        <div class="submit-button"><input type="submit"
value="Добавить"></div>
</form>
```

И после изменений:

```
<form method="post" action="{% url 'good_create' %}"
class="js-good-create-form">
  {% csrf_token %}
  <div class="modal-body">
    {% include "form.html" %}
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-default"
data-dismiss="modal">Close</button>
    <button type="submit" class="btn btn-primary">Create good</button>
  </div>
</form>
```

Атрибут **data-dismiss** со значением **modal** позволяет реализовать закрытие формы по нажатию данной кнопки.

4. **form.html**

В шаблон формы необходимо выполнить загрузку инструментов пакета **django-widget-tweaks**:

```
{% load widget_tweaks %}
```

Доработка контроллеров

Необходимо доработать один из двух наших контроллеров и написать третий. С первым — **good_list** — все в порядке, а во втором контроллере необходимо реализовать только проверку типа запроса: **POST** или **GET**. При этом также должен генерироваться экземпляр формы. Возвращать этот контроллер должен вызов дополнительного контроллера-функции **save_good_form** с со следующими параметрами: объект запроса, объект формы, шаблон. Поскольку данный контроллер должен «открыть» форму добавления товара, в качестве шаблона следует передавать **good_create.html**.

Контроллер **save_good_form** используется для проверки валидности и формирования словаря **data**, в котором мы будем указывать, является ли форма валидной, а также передавать преобразованный в строку шаблон и контекст (метод **render_to_string**). Данный контроллер будет возвращать ответ в формате **JSON**. Это позволит работать с формой и контекстом из файла скрипта средствами библиотеки jQuery, то есть асинхронно делать форму и передавать данные на сторону сервера.

Пример того, как может быть реализован дополнительный контроллер **save_good_form**:

```
def save_good_form(request, form, template_name):
    data = dict()
    if request.method == 'POST':
        if form.is_valid():
            form.save()
            data['form_is_valid'] = True
            goods = Good_Item.objects.all()
            data['html_good_list'] = render_to_string('good_list.html', {
                'goods': goods
            })
        else:
            data['form_is_valid'] = False
    context = {'form': form}
    data['html_form'] = render_to_string(template_name, context,
request=request)
    return JsonResponse(data)
```

Написание скрипта

Этот скрипт должен обрабатывать переданные в формате JSON данные: обращаться с помощью селекторов к элементам веб-страницы и осуществлять манипуляции над ними. Подразумевается динамическое обновление содержимого главной страницы: подгрузка шаблона формы, а также асинхронное (без перезагрузки главной страницы) сохранение данных в модель.

Для этого в файле сценария должны быть реализованы две функции. Первая отвечает за загрузку формы, вторая — за сохранение введенных в форму данных. Для запуска первой функции необходимо выполнить ее привязку к событию нажатия на кнопку добавления товара. Для запуска второй привязка должна быть выполнена к событию отправки формы. И функции, и инструкции их вызова должны быть сохранены в функции-обертке **$(function(){....программный код….});**, чтобы решить всю задачу средствами библиотеки jQuery.

Например, данный скрипт может быть реализован следующим образом:

```
$(function () {

  /* Код функций */

  var loadForm = function () {
    var btn = $(this);
    $.ajax({
      url: btn.attr("data-url"),
      type: 'get',
      dataType: 'json',
      beforeSend: function () {
        $("#modal-good").modal("show");
      },
      success: function (data) {
        $("#modal-good .modal-content").html(data.html_form);
      }
    });
  };

  var saveForm = function () {
    var form = $(this);
    $.ajax({
      url: form.attr("action"),
      data: form.serialize(),
      type: form.attr("method"),
      dataType: 'json',
      success: function (data) {
        if (data.form_is_valid) {
          $("#good-table tbody").html(data.html_good_list);
          $("#modal-good").modal("hide");
        }
        else {
          $("#modal-good .modal-content").html(data.html_form);
        }
      }
    });
    return false;
  };

  /* Подключение функций */

  $(".js-create-good").click(loadForm);
  $("#modal-good").on("submit", ".js-good-create-form", saveForm);

});
```

Дополнительные материалы

1. В чем разница между AJAX с JavaScript и jQuery?
2. Отличия между ajax, jQuery и простым js.
3. Вопросы кандидату на должность front-end разработчика.
4. JQuery and Django.
5. Использование JS для фронт-энда в Django.
6. Среда разработки jsFiddle.

Используемая литература

1. Ответы на вопросы на собеседование jQuery.
2. Ответы на вопросы на собеседование JavaScript.
3. AJAX.
4. jQuery.
5. JavaScript.