Математические структуры и моделирование
2013. № 2(28). С. 80–87

УДК 004.4

ОПТИМИЗАЦИЯ ВЕБ-ИНТЕРФЕЙСА НА СТОРОНЕ КЛИЕНТА ДЛЯ РАБОТЫ С ТАБЛИЧНЫМИ ДАННЫМИ

Е.А. Илюшечкин, И.С. Молодых

В работе описана задача создания клиентской части эффективного веб-интерфейса для работы с табличными данными на основе возможностей HTML5, а также приведены основные принципы программной реализации такого интерфейса.

Введение

С развитием глобальных сетей и веб-технологий веб-приложения стали одним из наиболее распространённых средств обработки удалённо хранящихся данных. В современных веб-приложениях приходится сталкиваться с задачей модификации больших массивов однотипных записей, которые могут быть представлены в виде таблицы. В дальнейшем подобные данные будем называть табличными.

Как правило, модификация табличных данных организуется по следующей схеме: пользователю предлагается выбрать запись из (возможно, отфильтрованного по некоторому критерию) списка, после чего он попадает в режим редактирования записи. По завершению работы с записью новые данные отсылаются на сервер, а пользователь вновь возвращается к списку. Данный подход обладает двумя недостатками.

1. **Большие накладные расходы на пересылку:** к полезному трафику в каждом запросе добавляется служебный заголовок. При наличии куки большого объёма, что является весьма типичным для корпоративных приложений, длина заголовка может превышать длину данных.

2. **Большая нагрузка на сервер:** для каждого запроса необходимо разобрать заголовок, выполнить проверки, связанные с авторизацией, инициализировать некоторый контекст.

Можно избавиться от этих недостатков, если кэшировать на стороне пользователя сделанные изменения, а затем передавать данные обо всех изменённых

Омский государственный университет им. Ф.М. Достоевского
E-mail: ilushechkinea@yandex.ru, jairinka@gmail.com

записях одним запросом (в пакетном режиме). Подключив механизмы браузеров Local Storage и Application Cache, относящиеся к стандарту HTML5, возможно кэшировать и веб-страницы, а сами данные сохранять на неопределенный срок. Помимо снижения расходов на пересылку и обработку данных, долгосрочное кэширование данных и веб-страниц на стороне пользователя позволяет реализовать две дополнительные возможности:

1. **Отложенное редактирование.** Пользователь может приостанавливать работу с данными, сохраняя сделанные изменения локально, и отсылать результат по завершении редактирования всех записей. Тем самым пользователь управляет целостностью вносимых изменений и может отложить публикацию результата, например, для консультации с более компетентным специалистом.

2. **Оффлайновый режим работы.** Можно вносить изменения в локальную копию данных при отсутствии подключения к сети, например, в дороге, и посылать результат при выходе в сеть.

Поскольку обрабатывать табличные данные приходится во многих приложениях с веб-интерфейсом, целесообразно создать типовое и гибкое решение, лишённое перечисленных недостатков и обладающее указанными возможностями. Данная работа описывает опыт создания такого решения.

1. Постановка задачи и инструментарий

Любое веб-приложение делится на серверную часть, выполняющуюся на веб-сервере, и клиентскую часть, выполняющуюся в браузере пользователя. Создать единое универсальное решение для серверной части приложения невозможно, так как серверная часть может быть написана на различных, не совместимых между собой языках программирования. Напротив, клиентская часть всегда выполняется в интерпретаторе языка JavaScript, встроенном в любой популярный браузер. Поэтому в данной работе рассматривается реализация клиентской части и интерфейс взаимодействия между клиентом и сервером.

В обсуждаемой реализации для передачи структурированных данных между клиентом и сервером используется формат JSON: он имеет малые накладные расходы (по сравнению, например, с иногда используемым XML) и хорошую встроенную поддержку в языке JavaScript. Вся передача данных осуществляется в фоновом режиме без перезагрузки страницы – используется стандартная технология AJAX. Для кэширования данных на стороне клиента используется встроенная во все современные браузеры технология Local Storage (далее – локальное хранилище) [1].

Часто значение определённого поля таблицы может выбираться из нескольких фиксированных вариантов, из некоторого справочника. Для реализации автономной работы эти справочники также необходимо загружать на сторону клиента. Вот минимальный набор методов, предоставляемых серверной стороной:

1. **loadData –** запрос текущих данных таблицы;
2. **saveData –** сохранение сделанных изменений на сервере;
3. **loadDictionary(name) –** запрос справочника с уникальным именем name.

Несмотря на то, что все современные браузеры имеют встроенный интерпретатор JavaScript, единый стандарт на набор доступных из скрипта объектов, их полей и методов до сих пор не реализован. Для решения проблемы кроссбраузерности, то есть, возможности одинакового выполнения клиентской части приложения во всех стандартных браузерах, предлагается использовать библиотеку, абстрагирующую разработчика от особенностей конкретных браузеров и предоставляющую набор универсальных примитивов. В настоящее время существует несколько подобных библиотек. Авторами была выбрана библиотека jQuery, имеющая большую популярность, множество расширений и богатый функционал. Данный выбор, однако, не является единственно возможным и наиболее эффективным. В частности, для решения частных случаев рассматриваемой задачи можно выбрать более легковесные и быстрые библиотеки.

Библиотека jQuery [2] предоставляет удобный способ расширения своего функционала в виде плагина – подключаемого модуля определённой структуры. Предлагаемое здесь решение было выполнено именно в такой форме, поскольку методы плагина вызываются в том же стиле, что и методы самой библиотеки. Это делает код приложения более унифицированным и, учитывая популярность jQuery, более читаемым сторонними разработчиками.

Плагин выполняет следующие основные функции:

1. **Отрисовка графического интерфейса:** таблицы с данными и элементов для ввода информации.
2. **Общение с сервером:** асинхронное получение данных таблицы и справочников, пакетная отправка списка сделанных изменений.
3. **Отслеживание изменений:** формирование текущего списка изменений в виде объекта, который может сохраняться в локальном кэше или отправляться на сервер.
4. **Отложенное редактирование:** сохранение изменений в локальном кэше, их автоматическое наложение после запуска страницы, очистка кэша.
5. **Оффлайновый режим работы:** определение наличия связи с сервером или, по меньшей мере, подключения к сети и выполнение соответствующих действий при отсутствии связи.

Можно выделить ряд дополнительных функций, упрощающих взаимодействие пользователя с таблицей: сортировка по столбцам, постраничная разбивка, клавиатурная навигация, отображение панели управления (например, в описываемой реализации на страницу выводилась панель с кнопками «Обновить с сервера», «Сохранить на сервер», «Сохранить локально»).

Авторами был проведен обзор существующих библиотек для работы с таблицами на стороне клиента. Ни одна из рассмотренных библиотек (Flexigrid [3], jqGrid [4], Ingrid [5]) не выполняла всех перечисленных функций, в частности, везде отсутствовала возможность отложенного редактирования и оффлайнового режима работы (табл. 1).

Таблица 1. Реализация возможностей в других библиотеках

Плагин	**Flexigrid**	**jqGrid**	**Ingrid**
Отрисовка графического интерфейса	+	+	+
Асинхронный обмен данными с сервером	+	+	+
Пакетная передача данных об изменениях	-	+	-
Отложенное редактирование	-	-	-
Возможность оффлайн-работы	-	-	-

2. Архитектура плагина

Целесообразно разбить разработку плагина на несколько относительно независимых, взаимодействующих друг с другом компонентов, которые можно реализовать, например, в виде объектов JavaScript:

1. **Модуль общения с сервером** отправляет AJAX-запросы на заданный адрес и обрабатывает ответы сервера. В результате он либо возвращает данные другому компоненту плагина (менеджеру словарей или модулю отображения данных), либо передаёт информацию об ошибке модулю оповещений. Запросы могут генерироваться как действиями пользователя (нажатие на определённую кнопку), так и вызовами от других компонентов.

2. **Менеджер словарей** управляет информацией о словарях, предоставляет эту информацию, когда нужно отобразить список выбора для определённого поля таблицы. При необходимости подгружает отсутствующую информацию с сервера с помощью модуля общения с сервером, либо из локального хранилища.

3. **Менеджер локального хранилища** позволяет кэшировать данные таблиц, словарей, список внесённых изменений в локальное хранилище браузера и загружать их по требованию.

4. **Модуль отображения данных** отвечает за отображение табличных данных и полей их редактирования. Сюда же выносятся дополнительные функции, обеспечивающие интерактивность работы пользователя вроде сортировки, постраничной разбивки и клавиатурной навигации.

5. **Монитор изменений** отвечает за формирование списка изменений со времени последнего обновления данных с сервера.

6. **Менеджер блока управления** отображает панель с кнопками для вызова методов сохранения данных на сервер, помещения изменений в локальный кэш, обновления данных с сервера. Здесь же задаются параметры фильтрации данных на стороне клиента. Может быть объединён с модулем отображения данных.

7. **Модуль оповещений** уведомляет пользователя об ошибках, доступности сервера и различных событиях, а также выводит отладочные сообщения в консоль браузера.

3. Принципы реализации

Предлагаемое решение для работы с табличными данными, выполненное в виде плагина библиотеки jQuery, состоит из трёх файлов: ядра, файла настроек и файла CSS, первый из которых не зависит от среды применения, а два других могут изменяться под конкретные нужды. Прилагающийся файл настроек изначально содержит настройки по умолчанию и одновременно является справочником по доступным для изменения параметрам. Настройки выполнены в форме единого JavaScript-объекта и состоят из:

1. Перечня словарей с указанием ссылок для их получения с сервера.

2. Набора столбцов таблицы и их параметров (заголовок, доступность для редактирования, привязанный словарь и т. д.).

3. Шаблонов html-разметки.

4. Шаблонов селекторов.

5. Глобальных параметров таблицы (CSS-класс контейнера таблицы, идентификатор таблицы для различения нескольких таблиц на странице).

6. Пользовательских обработчиков для некоторых событий.

3.1. Шаблонизация как средство достижения гибкости

В общем случае шаблон – это описание некоторой универсальной абстрактной структуры данных или алгоритмов, которая при заполнении реальными данными позволяет решать класс схожих задач. Шаблонизация библиотеки является основой её гибкости – ядро, оперирующее шаблонами, может использоваться везде без изменений, изменяться будет лишь файл настроек, приспосабливающий это ядро под конкретную среду применения. Подобное разделение ядра и настроек позволяет легко обновлять библиотеку без необходимости повторения внесённых в предыдущую версию модификаций. Кроме того, файл настроек имеет (в отличие от файла, содержащего код) декларативный характер и представляет контекст применения библиотеки в гораздо более читаемом виде.

Чтобы выводимая таблица гармонично вписывалась в веб-интерфейс, для её элементов может понадобиться особая разметка (например, иногда необходимо оборачивать данные ячеек таблиц в дополнительные блоки). Если жёстко задать html-разметку выводимой таблицы в коде библиотеки, то для использования библиотеки в составе конкретного сайта может понадобиться модификация её кода. Поэтому, чтобы сделать плагин универсальным, имеет смысл использовать шаблоны разметки – переменные, содержащие фрагменты разметки с заполнителями, замещающимися на реальные данные. Например, для ячейки таблицы шаблон может иметь вид **<td> {data} </td>**. При выводе на страницу параметр **{data}** замещается на реальные данные ячейки функцией плагина, получающей на вход имя шаблона и список фактических значений его аргументов. Вынесение шаблонов в файл настроек позволяет не только достичь большей гибкости, но и отделить код от данных, что является хорошей практикой разработки.

Параметры, отвечающие за внешний вид блоков html-разметки (цвет, форму, отступы и т. д.), в современных веб-приложениях сконцентрированы в файлах CSS-стилей. Элементам страницы приписываются неуникальные атрибуты **class** и уникальные атрибуты **id**, а в CSS-файле сосредоточены правила отображения при определённой комбинации значений этих и некоторых других атрибутов. Хотя наличие или отсутствие CSS-файла не влияет на функционал скрипта в образце файла настроек, распространяемом с плагином, все шаблоны html-разметки используют файл стилей для определения всех параметров отображения описываемых ими элементов. При изменении настроек рекомендуется сохранять этот принцип и не использовать в шаблонах html-разметки атрибутов, непосредственно влияющих на внешний вид блоков. Использование такого подхода способствует дальнейшей структуризации данных за счёт разделения информации об отображении и о структуре.

Для поиска элементов страницы и манипуляции ими в jQuery и её аналогах применяются выражения специального вида – селекторы. Поскольку селекторы используются в коде неоднократно, а гибкая разметка может вносить изменения в исходную структуру таблицы, имеет смысл отделить селекторы от кода и сосредоточить их в одном месте – в файле настроек. При этом некоторые селекторы тоже необходимо задавать в виде шаблонов. Например, шаблон селектора для выбора содержимого n-й ячейки имеет по умолчанию вид **td:eq('{n}') > ***, а для приведённого выше примера разметки – **td:eq('{n}') > div > ***.

Чтобы без модификации ядра менять не только внешний вид отображаемых элементов, но и поведение плагина, при разработке применялся принцип проектирования, называемый шаблонным методом [6]. Код библиотеки содержит функции-заглушки, вызываемые там, где пользователю, вероятно, понадобится дополнить стандартную обработку данных и событий своими действиями. Эти функции могут быть наполнены реальным содержанием, если пользователь определит их в файле настроек. Поскольку jQuery, как и другие библиотеки, позволяет пользователю создавать собственные события, для получения аналогичной возможности допустимо использовать событийно-ориентированный

подход: вместо вызова функции-заглушки генерировать специальное событие. Тогда пользователь будет не переопределять в настройках функции-заглушки, а указывать обработчики для этих специальных событий. Сравнение этих двух подходов может быть предметом отдельного исследования.

3.2. Особенности работы с локальным хранилищем

Локальное хранилище в современных браузерах позволяет хранить строковые пары ключ-значение. Для сохранения объектов JavaScript в виде строки и их последующего восстановления можно использовать встроенные методы сериализации и десериализации, предоставляемые скриптовыми движками браузеров.

Как правило, квота хранимых данных на один домен составляет 5МБ. Очевидно, для обеспечения работы с большим объёмом данных необходимы некоторые организационные меры. Например, можно опубликовать на сайте инструкции по увеличению квоты в настройках различных браузеров.

Для экономии используемого в локальном кэше места стоит применять методы сжатия данных. Если учесть, что при сериализации имена полей объектов попадают в результат, можно перед публикацией библиотеки пропустить её через так называемый минификатор (программы для уменьшения размера скрипта), который в том числе может уменьшить длину имён всех используемых переменных до 2-3 символов.

3.3. Особенности создания оффлайн-приложения

Для обеспечения работы с локально кэшированными данными без подключения к сети необходимо использовать механизм современных браузеров, позволяющий кэшировать файлы веб-страниц,– Application Cache [1]. По сути, это требует лишь перечисления всех кэшируемых элементов в файле манифеста и указания этого файла в атрибутах страницы. Файл манифеста никак не зависит от кода библиотеки, но, чтобы избежать проблем с обновлением прочих страниц сайта, рекомендуется иметь отдельную версию страницы, подлежащую кэшированию, которая содержит непустой атрибут манифеста.

Библиотека должна уметь определять наличие подключения к сети. Браузеры позволяют отслеживать доступность сети с помощью специального объекта **navigator**, и во многих случаях этого достаточно. Однако оказывается, что таким образом отслеживается не подключение к Интернет или корпоративной сети, а лишь наличие активного сетевого интерфейса. Например, при тестировании библиотеки выяснилось, что браузер считает, что он находится в сети, даже если сеть недоступна, но активен один из виртуальных сетевых интерфейсов для связи с виртуальной машиной. Эту особенность стоит учесть в реализации и вместо проверки стандартного объекта браузера выполнять тестовый запрос на сервер веб-приложения.

Заключение

В данной работе была рассмотрена задача создания эффективного пользовательского интерфейса веб-приложения для работы с табличными данными, а также предложен путь её универсального решения на основе опыта авторов. Безусловно, это не единственный возможный способ решения описанной задачи. Кроме того, современные возможности веб-браузеров стоит применить и для решения прочих проблем, связанных с улучшением интерфейса пользователя.

Развитие предложенного здесь решения можно продолжить по нескольким направлениям.

1. Корректная обработка конфликтов редактирования – ситуаций, когда одна запись редактируется несколькими пользователями.

2. Поиск наилучшего способа сжатия данных в кэше для более эффективного использования локального хранилища.

3. Возможность редактирования связанных таблиц и валидация табличных связей на стороне клиента.

Литература

1. Mark Pilgrim. Dive into HTML5. URL: http://diveintohtml5.info/index.html (дата обращения: 11.10.2013).
2. jQuery API documentation. URL: http://api.jquery.com (дата обращения: 11.10.2013).
3. Flexigrid. Official site. URL: http://flexigrid.info (дата обращения: 11.10.2013).
4. jqGrid. Official site. URL: http://www.trirand.com/blog (дата обращения: 11.10.2013).
5. Ingrid. Official site. URL: http://reconstrukt.com/ingrid (дата обращения: 11.10.2013).
6. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. СПб. : Питер, 2001. 368 с.