УДК 004.45

В.П. Молчанов

Харьковский национальный экономический университет им. Семена Кузнеца, Харьков

АРХИТЕКТУРА ПРОЦЕССОВ КЛИЕНТСКОЙ ЧАСТИ WEB-ПРИЛОЖЕНИЙ

В статье рассмотрены вопросы организации вычислений на клиентской части WEB-приложений, обеспечивающих обучение на рабочем месте. Сформулированы требования к таким приложениям и рассмотрены возможные средства повышения эффективности приложений за счет организации вычислений в виде взаимодействующих процессов. В качестве средств рассмотрены новые API HTML5.

Ключевые слова: *обучение на рабочем месте, WEB-приложения, параллельные процессы, API.*

Введение

Для разработки приложений, обеспечивающих обучение на рабочем месте, предлагается множество специальных средств и технологий (от *Flash* до *Adobe Director*). Однако ориентация на использование сети интернет и развитие ее базовых средств позволяют рассматривать эти базовые средства в качестве альтернативы специализированным.

Говоря о возможностях базовых средств WEB-технологии в контексте разработки *e-learning*, можно выделить два направления: возможности изобразительных средств и возможности, влияющие на архитектуру (структуру) приложения. Специфические требования к изобразительным средствам создания обучающих приложений существуют, и использование для этих целей HTML, CSS и JS достаточно подробно рассмотрены [1]. А анализ возможностей WEB-приложений и их соответствия требованиям (потребностям) со стороны разработчиков средств обучения освещены мало и представляют определенный интерес.

Основной материал статьи

Для обучающей системы характерным является структура процессов, характер и объем данных обмена. Представление о структуре процессов и данных может дать анализ педагогических сценариев [2; 3]. Анализ этих сценариев позволяет представить работу обучаемого в виде некоторой обобщенной схемы, содержащей различные действия.

Вначале выполняются действия по загрузке основного приложения, в ходе которой восстанавливается состояние последнего обращения. Затем следует некоторая работа в среде основного приложения. Эта работа может прерываться для обдумывания, выполнения других действий или ожидания некоторых событий.

Работа в основном приложении может потребовать запуска параллельного приложения, например, справочника, который должен быть визуально доступен и связан с основным приложением. У каждого приложения свой интерфейс и свои функции, но они должны быть связаны.

В ходе работы могут запускаться вспомогательные программы для выполнения отдельных задач с обращением к базам данных и другим источникам. Полученные данные могут быть сохранены локально для ускорения последующего использования или использованы в основном приложении. Кроме того, вспомогательные программы могут потребоваться для выполнения действий, требующих значительных ресурсов (например, анализ и отрисовка трехмерных изображений и т.п.). Их особенностью может быть отсутствие визуального интерфейса, но обязательно требуется связь по данным с запускающим их приложением.

Описанная схема может иметь множество вариантов в зависимости от числа и последовательности действий, однако, они могут быть обобщены и проиллюстрированы схемой, приведенной на рис. 1.

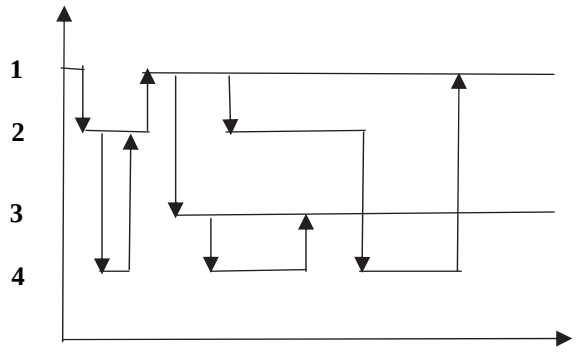

Рис. 1. Схема взаимодействия процессов

На рис.: 1 – это основной процесс, 2 – некоторые расчеты (например, рендеринг трехмерных моделей), 3 – обращение к справочным материалам, 4 – работа с данными (выбор из баз данных, сохранение и т.п.).

Даже беглый взгляд на эту схему приводит к мысли о многопоточности и параллельных процессах. Их реализация на сервере не вызывает особых проблем, существующие технологии обеспечивают потребности разработчиков. Однако интерес представляет подход с максимальным использование возможностей клиента. Так как в этом случае уменьшается зависимость от качества соединения с сервером, снижается трафик, и в конечном счете можно обеспечить более комфортную работу для обучаемого.

Какие из этого вытекают требования?

Должна быть обеспечена одновременная работа нескольких связанных приложений. Эта связь может быть в форме синхронизации, обмена данными или сообщениями.

Должна быть обеспечена возможность запуска нескольких параллельных процессов с возможностью обмена данными.

Должна быть обеспечена возможность сохранения данных на клиенте и возможность их использования.

Должно быть обеспечено эффективное взаимодействие с сервером в тех случаях, когда этого невозможно избежать.

Причем с точки зрения структуры (архитектуры) такого приложения традиционные возможности средств WEB-технологий (протоколы, языки, API) имеют ряд ограничений. Это последовательная работа основного клиента (браузера), ограничения на связи и запуск программ, взаимодействие с сервером без сохранения истории. Эти ограничения можно преодолеть с помощью различных подходов, например, куки (*cookie*), создание срытых полей форм и многое другое. Однако, включение возможностей по преодолению этих ограничений в браузеры на основе реализации новых стандартов обеспечит более высокий уровень решения проблем, а значит более высокое качество, большую надежность.

С точки зрения возможностей по организации вычислительных процессов на машине пользователя в браузере основной интерес представляют новые программные интерфейсы (API), разрабатываемые в рамках стандарта HTML5. Их последние версии существенно расширяют возможности WEB-приложений по взаимной коммуникации. Эти API содержат объекты и методы, обеспечивающие разработку программ для реализации требуемой архитектуры.

Естественно, каждый разработчик снабжает свой браузер такими интерфейсами по собственному разумению, хотя в большинстве случаев ориентируясь на стандарты. В этой связи степень поддержки разработки обучающей среды на основе множества взаимодействующих процессов, которые осуществляют обмен данными, будет разной в разных браузерах.

В новых API, которые так или иначе могут быть использованы при организации вычислительного процесса на клиенте, а это *Application Cache*, *Server-Sent Events* и *WebSocket* реализованы средства, позволяющие преодолеть недостатки широко распространенного AJAX [4]. Все они предполагают обмен данными с сервером.

Application Cache API — совокупность функций, обеспечивающих продвинутое кэширование ресурсов WEB-приложения, с помощью которых можно использовать загруженные ранее ресурсы без подключения к сети Интернет. В отличие от стандартного буфера, в который помещаются только файлы, загруженные в процессе просмотра страниц, в *Application Cache* можно поместить файлы, загружаемые с сервера, в соответствии со специальной инструкцией, находящейся в файле.

Файл содержит три секции. Первая (CACHE) содержит список файлов, которые необходимо сохранить для автономной работы. Страницы, использующие эти файлы, сохраняются автоматически при посещении. Вторая (FALLBACK) содержит перечень страниц с адресами перенаправления, при отсутствии копий в кэше. Третья (NETWORK) содержит пути к файлам, которые не могут использоваться без подключения с Интернету. Информация об этом файле помещается в качестве значения атрибута *manifest*, например, *<html lang="ru" manifest="/offline.manifest">*.

Использование *Application Cache* API может не только увеличить скорость загрузки страниц пользователям и уменьшить размер трафика, но и снизить нагрузку на сервер (число обращений).

Дополнить эту возможность позволяет технология хранилища данных, которая обеспечивает сохранение данных на время сессии (объект *sessionStorage*) и между сессиями (объект *localStorage*). Кроме того, сохраняемые данные привязаны к домену, с которым работает браузер. Данные сохраняются в виде пар ключ-значение, можно использовать методы объекта (*setItem("key", "value"), getItem("key"*), *removeItem("key"*) и т.п.) или как ассоциативный массив (*localStorage["key"] = "value";*). Все сохраняемые данные относятся к строковому типу. Соответствующий интерфейс дает возможность устанавливать, редактировать и удалять данные. Для хранилищ каждого типа и по каждому домену используется отдельное представление *Storage*-объекта, они функционируют и управляются отдельно друг от друга. Например, можно сохранять индивидуальные пользовательские настройки различных сред и восстанавливать их при возвращении пользователя к работе с ресурсом. Сохраненные

данные доступны только сценариям и не пересылаются по сети.

Современный стандарт *Server-Sent Events* позволяет браузеру создавать специальный объект *EventSource*, который обеспечивает соединение с сервером, повторное соединение в случае обрыва и генерирует соответствующие события при поступлении данных. При создании объекта (*var eventSource=new EventSource("/events/subscribe")*) браузер автоматически подключается к серверу по заданному адресу и начинает генерировать события по его сообщениям. Таких событий три: *onopen* – при успешном установлении соединения, *onerror* – при ошибке соединения, *onmessage* – при поступлении данных.

Благодаря этому появляется возможность получать данные с сервера по инициативе и в моменты, определяемые сервером, а не клиентом. Это эффективно в тех случаях, когда осуществляется односторонняя и достаточно частая передача данных с сервера клиенту. Отпадает необходимость многократных запросов для проверки готовности данных.

Еще одним расширением, служащим для организации взаимодействия с сервером, является *WebSocket*. С помощью его API создается постоянное двунаправленное сетевое соединение между браузером пользователя и сервером. По сути, это расширение протокола HTTP. После установления TCP-соединения оно остается открытым, обеспечивая постоянный канал обмена данными. Сами данные пересылаются асинхронно с обеих сторон без выполнения *http*-запроса.

На клиенте создается объект var ob_sc=new WebSocket("src"), src - URL-адрес к которому производится подключение. Сервер, если он поддерживает такое взаимодействие с этого адреса, должен ответить на websocket-запрос. Объект обеспечивает отправку сообщений (метод ob_sc.send('mess')), обработку сообщений сервера (ob_sc.onmessage) и ошибок (ob_sc.onerror). Пересылаются строки или бинарные данные.

Таким образом, все три рассмотренных подхода расширяют возможности по организации работы с сервером. Причем новые подходы обеспечивают более высокую эффективность в тех ситуациях, когда AJAX оказывается малоэффективным (например, пересылка данных по инициативе со стороны сервера или интенсивный двухсторонний обмен данными небольшого объема).

Все рассмотренные новшества направлены на совершенствование организации вычислений, связанных с взаимодействием между клиентом и сервером. Однако появились и средства для эффективной организации параллельности при выполнении скриптов в браузере. Это два API: *Web Messaging* и *Web workers*.

Web Messaging обеспечивает передачу данных между HTML-документами, открытыми в разных окнах браузера. Это могут быть разные вкладки, окна, созданные тегом OBJECT, или фреймы. Они имеют разный контекст, недоступный из соображений безопасности. Однако потребность в таком обмене реально существует. В нашем случае (приложения, ориентированные на обучение), это может быть взаимодействие со справочными данными их поиском и фильтрацией, использование вспомогательных приложений и т.п. Основным препятствием был запрет на доступ к контексту из другого домена. *Web Messaging* API свободен от этого ограничения.

Для обмена используется метод *postMessage* и событие *onmessage*. Метод вызывается в скрипте на странице, которая является отправителем. Получатель идентифицируется в контексте отправителя. При этом можно управлять контролем источника, открытого в окне получателя. Если такой контроль включен, браузер сравнивает домен, указанный при отправке сообщения, с открытым в окне: *win.postMessage(sendObject,"location")*. При использовании в качестве второго параметра "*" контроль отключается. В качестве отправляемых данных (*sendObject*) может быть любой объект.

На странице-получателе должен быть подключен обработчик события *onmessage*, в котором и выполняются действия по использованию полученных данных. Решение о доверии к источнику сообщения ложится на принимающую сторону, которая может проанализировать домен источника и содержимое сообщения.

Общая схема работы выглядит следующим образом. Отправитель вызывает *postMessage*. Если второй параметр не "*", то браузер проверяет, совпадает ли источник получателя с указанным. Если совпадает, то для получателя генерируется событие *onmessage*, в объекте которого передаются источник сообщения и данные. Данные извлекаются и используются на странице-получателе.

И наконец, самым востребованным и напрямую отвечающем требованиям параллельности вычислений является API *Web workers*. Суть предложенного подхода состоит в том, что скрипт, размещенный в отдельном файле, загружается в браузер и выполняется в отдельном потоке. Для управления этим потоком создается специальный объект (*var myWorker = new Worker("worker.js");*). Запуск потока осуществляется автоматически после создания объекта, уничтожение выполняется методом *myWorker.terminate()*. Взаимодействие страницы с этим потоком организуется путем обмена сообщениями (событие *message* и метод *postMessage*). Сообщения могут передаваться в обе стороны. Для получения и обработки сообщений на основной странице и в запущенном потоке необходимо ис-

пользовать соответствующие обработчики событий. В сообщениях могут передаваться как строки, так и объекты.

Для скриптов, запускаемых таким способом в отдельном потоке, имеется ряд ограничений: отсутствие доступа к дереву DOM основной страницы и объектам window, document и parent. Доступны объекты navigator, location (только для чтения), importScripts(), XMLHttpRequest и методы setTimeout(), setInterval(). Есть возможность совместного использования потоков (объект SharedWorker()).

Выводы

В целях проверки возможности реализации требуемой архитектуры процессов в браузере и оценки ее эффективности был разработан каркас приложения, в котором вместо функциональных блоков обучающего приложения использованы фрагменты кода для загрузки процессора и средства трассировки. Проверка работы в разных браузерах подтверждает реализуемость и эффективность рассмотренных решений.

Список литературы

1. Молчанов В. Анализ реализации новых WEB-стандартов в массовом программном обеспечении / В. Молчанов // Системи обробки інформації : збірник наукових праць. – Х.. Харківський університет Повітряних сил імені Івана Кожедуба, 2016. – Вип. 4 (141). – 226 с.

2. Можаева Г.В. Как подготовить мультимедиа курс? - [Электронный ресурс] / Г.В. Можаева, И.В. Тубалова. – Электрон. дан. – Режим доступа: http://www.ict.edu.ru/ft/003620/index.html.

3. Педагогічний дизайн засобів електронного навчання на робочому місці: монографія / Під ред. д.е.н., проф. В.С. Пономаренка, д.е.н., проф. О.І. Пушкаря. – Х.: ХНЕУ ім. С. Кузнеця, 2017. – 276 с.

4. Интерфейсы веб API – [Электронный ресурс]. – Электрон. дан. - Режим доступа: https://developer.mozilla.org/ru/docs/Web/API.

Поступила в редколлегию 14.03.2017

Рецензент: д-р экон. наук проф. А.И. Пушкарь, Харьковский национальный экономический университет им. Семена Кузнеца, Харьков.

АРХІТЕКТУРА ПРОЦЕСІВ КЛІЄНТСЬКОЇ ЧАСТИНИ WEB-ДОДАТКІВ

В.П. Молчанов

У статті розглянуті питання організації обчислень на клієнтської частини WEB-додатків, що забезпечують навчання на робочому місці. Сформульовано вимоги до таких додатків і розглянуті можливі засоби підвищення ефективності програм за рахунок організації обчислень у вигляді взаємодіючих процесів. Як засоби розглянуті нові API HTML5.

***Ключові слова**: навчання на робочому місці, WEB-додатки, паралельні процеси, API.*

ARCHITECTURE PROCESSES CLIENT-SIDE WEB-APPLICATIONS

V. Molchanov

The article deals with the organization of computations on the client side WEB-applications that provide on the job training. The requirements for such applications and considered possible means of increasing the efficiency of applications through algorithms in the form of interacting processes. As the funds are considered new API HTML5.

***Keywords:** workplace learning, WEB-application, parallel processes, the API.*