Разработчикам API
JavaScript API общее описание.

www.amiro.ru

Amiro JS API.

amiJSAPI содержит как объекты, расширяющие возможности Javascript, так и классы, которые необходимы для работы со страницами и взаимодействия с плагинами.

JS API необходим для отображения данных, полученных с сайта, без перезагрузки страницы. Для этого:

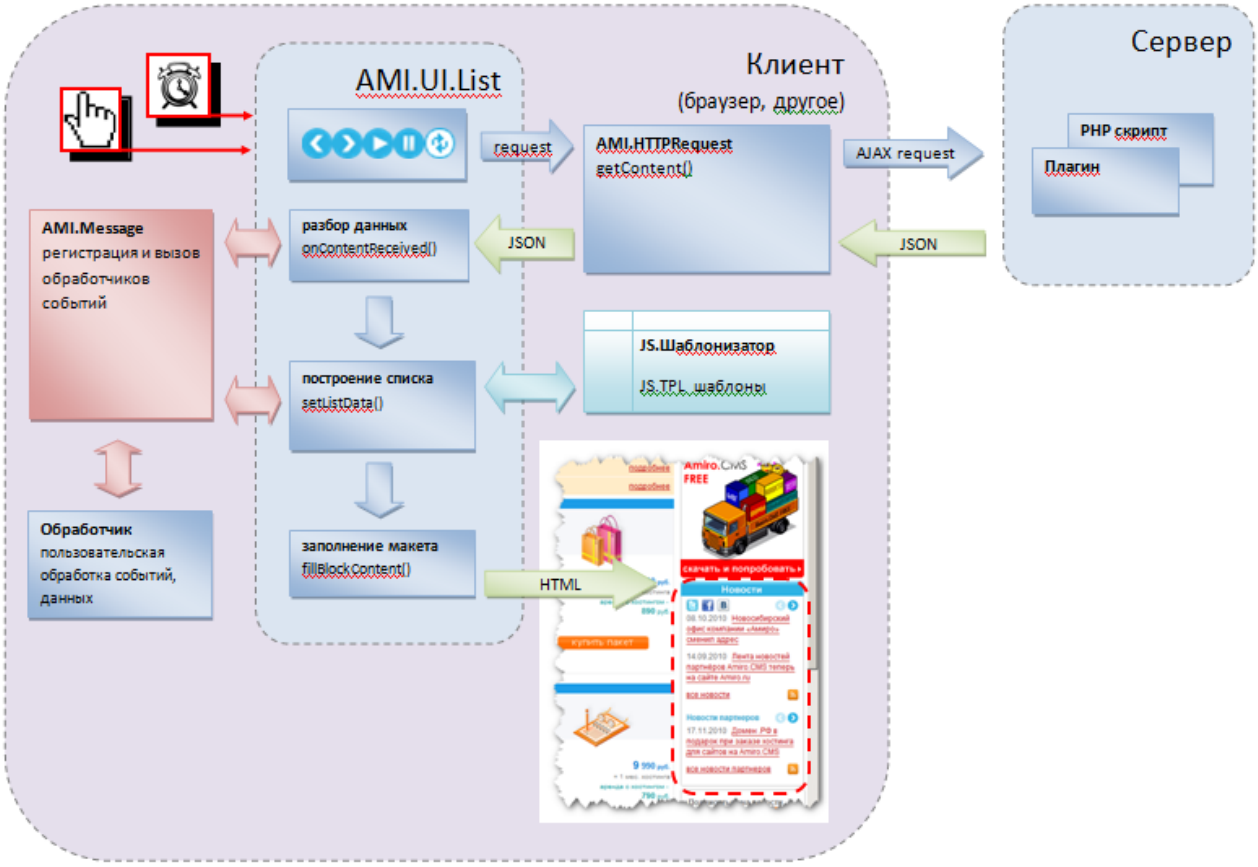
1. По запросу страницы Amiro.CMS строит её из спецблоков и данных текущего модуля. Спецблоком может быть так же элемент JS API. Он представляет собой плагин, который выводит все необходимые HTML данные для начала асинхронной работы спецблока.
2. Изначально выводится страница, содержащая блоки, обслуживаемые JS API. Содержимое этих блоков, как правило, состоит из:
    a. HTML кода блока по-умолчанию;
    b. HTML шаблонов, небходимых для оформления данных, из которых состоит блок. Шаблоны для JavaScript функционала хранятся в том же шаблоне системы, где все сеты, отображающие содержимое спецблока. Каждый шаблон для JS API. Конструкции @@, используемые в JS шаблоне игнорируются PHP шаблонизатором. Для вывода шаблона в HTML код страницы используется тэг SCRIPT с атрибутом type=”text/html ”.
        - JavaScript, содержащего основные параметры блока и инициализацию необходимого компонента JS API.
    - Как только показано содержимое по-умолчанию API отправляет запрос на сервер (AMI.HTTPRequest) для получения необходимой порции данных.
    - При получении данных JavaScript компонент вставляет их в необходимый шаблон (AMI.Template) и строит содержимое блока.
3. На всех этапах API отправляет сообщения для зарегистрированных объектов-слушателей на дополнительную обработку данных (AMI.Message).

Amiro JS API полностью готов для создания собственных интерфейсов пользователей. Тем не менее в объекте AMI.UI созданы несколько уже готовых компонент пользовательского интерфейса для работы со странице. Это:

1. AMI.UI.Suggestion – подсказки для введённой пользователем строки в текстовом поле.
2. AMI.UI.MediaBox – показ media-объектов на странице, таких как картинки, FLASH в отдельном слое на странице.
3. AMI.UI. Multiselect – замена стандартного <select multiple=”multiple”> на графический контрол.

4. AMI.UI.List – Объект, предназначенный для работы со списочным блоком данных, таким как обыкновенный спецблок.

Схема работы JS API на примере AMI.UI.List:



Для того чтобы упростить работу с готовыми объектами и не использовать постоянно длинные цепочки при вызове методов можно сделать переопределение объекта и работать уже с новым. Например:

```
<script type=”text/javascript”>
   var myEvent = new Object();
   myEvent.prototype = AMI.Browser.Event;
   myEvent.set(‘var’, ‘value’, 1);
</script>
```

AMI – объект (далее контейнер), который содержит в себе все классы и объекты для работы с amiJSAPI. API так же включает в себя AMI.Browser, который содержит набор вспомогательных функций, предоставляющих стандартный доступ к параметрам и свойствам страниц одинаково подо всеми браузерами.

AMI.Browser – контейнер, который содержит в себе объекты и функции, которые предназначены для стандартизации работы с параметрами страниц, такими как определение размеров или позиций, под всеми браузерами. Методы объекта:

- *int getWindowWidth()* - определение ширины окна браузера.
- *int getWindowHeight()* - определение высоты окна браузера.
- *int getDocumentWidth()* - определение ширины документа (страницы).
- *int getDocumentHeight()* - определение высоты документа.
- *int getDocumentLeft()* - определение координаты левой видимой точки документа в окне браузера.
- *int getDocumentTop()* - определение координаты верхней видимой точки документа в окне браузера.
- *array getPointerPosition(oEvent)* – определение X и Y координаты курсора относительно документа. В возвращаемом массиве 0 индекс содержит X координату, а 1й - Y координату. Параметр oEvent должен содержать последнее актуальное событие.
- *array getObjectPosition(oObject, bStopOnRelative)* – определение X (0й индекс возвращаемого массива) и Y (1й индекс) координаты заданного объекта oObject относительно документа. Флаг bStopOnRelative указывает останавливаться ли на первом абсолютном родителе элемента.
- *int getCaretPosition(textObject)* – определение смещения курсора в содержимом текстового поля textObject.
- *int setCaretPosition(textObject, position)* – установка курсора в текстовом поле textObject на заданную позицию position.
- *void setOpacity (oElement, iOpacity)* – установка полпрозрачности уровнем iOpacity для элемента iOpacity.

AMI.Browser.Cookie – объект, в котором хранятся методы для работы с cookie браузера.

- *void set(sName, sValue, iHours)* – установка переменной cookie sName в значение sValue на заданный период iHours.
- *void get(sName)* – получение значения переменной cookie sName.
- *void del(sName)* – удаление переменной cookie sName.

AMI.Browser.Event – объект, который содержит в себе методы для работы с событиями браузера (onLoad, onClick, onMouseOver и т.п.).

- *object validate(oEvent)* – получение корректного объекта события, основываясь на переданном объекте oEvent.
- *void addHandler(oTarget, sEvent, oHandler)* – установка обработчика oHandler события sEvent (без «on») для объекта oTarget. oHandler должен быть функцией, которая будет вызвана с параметром event.
- *void removeHandler(oTarget, sEvent, oHandler)* – удаление обработчика oHandler события sEvent для объекта oTarget.

- *void stopProcessing(oEvent)* – остановка распространения события oEvent по DOM модели родителям и выполнения действия по-умолчанию.
- *object getTarget(oEvent)* – получение объекта для которого было вызвано событие oEvent.
- *bool fire(oTarget, sEvent)* – вызов события sEvent для объекта oTarget без участия пользователя.

AMI.Browser.DOM – объект, методы которого позволяют работать с объектной моделью браузера более эффективно.

- *object create(sTagName, sId, sClassName, sStyles, oParentNode)* – создание элемента в объектной модели по заданному тэгу *sTagName* с параметрами ID sId, классом sClassName и стилями sStyles. Если задан родительский элемент oParentNode, то созданный элемент будетпомещён последним в его объектной модели.
- *object append(oParentNode, oNode)* – удаление элемента oNode из объектной модели oParentNode.
- *string getStyle(oObject, sStyleName)* – определение значения стиля sStyleName для элемента oObject.

API

AMI.Message – объект для работы с сообщениями, которые посылают различные методы amiJSAPI. Позволяет регистрировать обработчиков сообщений и обрабатывать данные, присланные через сообщения.

- *void addListener(sMessage, oCallback)* – добавление обработчика oCallback для определённого сообщения sMessage API. Функция oCallback должна принимать 2 параметра (param1 и param2), которые передаются через сообщения. Параметры передаются как объекты, т.е. ими можно манипулировать и это отразится в том методе, который вызвал событие.
- *void removeListener(sMessage, oCallback)* – удаление обработчика oCallback для события sMessage.
- *void send(sMessage, param1, param2)* – создание сообщения sMessage с последующим вызовом всех обработчиков. В процессе работы изменяется переданный в метод объект param2, который возвращается как результат обработки.

AMI.Template – объект для работы с шаблонами: замена переменных в шаблоне на данные, взятые из переданного методу массива. Шаблон поддерживает простые переменные и условия с неограниченным уровнем вложенности.

- *string parse(content, aData)* – обработка указанного шаблона content с заменой переменных, значения которых перутся из объекта-хэша aData и возвращение готовой строки.

Синтаксис шаблона.

1. Переменные в шаблоне заключаются между @@.
2. Условный блок начинается с @@if(условие)@@ и заканчивается @@endif@@. Условный блок может содержать @@else if(условие)@@ и @@else@@. Условие аналогично таковому в Javascript, переменные берутся из массива, передаваемого как параметр метода parse.

Пример шаблона:

```
@@if(ext_img_small != "")@@
   <img src="@@ext_img_small@@" alt="" />
@@endif@@
<div class="ami_resp_row_announce_news">@@announce@@</div>
```

AMI.String – объект для работы со строками, специфичными для Amiro.CMS.

- *string decodeHTMLSpecialChars(content)* – замена в строке content закодированных специальных символов на обычные, например # на #, < на < и т.п.
- *mixed decodeJSON(data)* – восстановление данных JavaScript, сериализованных в строку data.

AMI.HTTPRequest – класс, который предназначен для асинхронной загрузки данных с того же домена, на котором работает сайт. Так же этот класс позволяет отправлять данные формы в параллельном потоке. В обоих случая регистрируется обработчик, который будет вызван при получении данных.

- *void addVariable(key, value)* – установка переменной key со значением value, которая будет отправлена на сервер с запросом.
- *void getContent(method, url, variables, callbackFunction)* – формирование запроса url на сервер методом method (POST, GET) с переменными variables. Как результат вызывается зарегистрированный обработчик callbackFunction, которому параметром передаётся содержимое.
- *void submitForm(url, callbackFunction)* – отправка данных формы url на сервер с аналогичным getContent действием.

AMI.UI – класс, который является базовым для всех UI классов и является их контейнером. Каждый UI класс (компонент UI) – это самостоятельный компонент, содержащий набор функций для обработки и представления

данных, полученных с сервера. AMI.UI предоставляет функционал, который одинаково необходим всем компонентам.

• • • • • • • • • • • • • • • • • UI

AMI.UI.Suggestion – класс, необходимый для вывода подсказок при вводе значения в текстовое поле.

Объект работает на основе плагина ami_ajax_responder, где в качестве модуля используется search_history.

Чтобы подключить подсказки к текстовому полю необходимо для самого поля задать ID, например, idSearchField и прописать атрибут autocomplete=”off”. Ниже вставить следующий JavaScript код:

```
var suggestionRequestData = {
   id_plugin: 'ami_ajax_responder',
   locale:      'ru',
   module:    'search_history',
   order:      'quantity',
   dir:         'D',
   limit:       '10',
   offset:     '0'
};
var searchVariants = new AMI.UI.Suggestion('idSearchField', suggestionRequestData);
```

Инициализация объекта и расставление обработчиков событий происходит автоматически после создания переменной, содержащей этот объект.

AMI.UI.MediaBox – объект для отображения медиа-данных, таких как изображения, FLASH, FLV.

Объект AMI.UI.MediaBox уже создан и готов к использованию, создавать его заново с помощью new не нужно. Для открытия изображения используется метод open.

- *void addSkin (sSkin, aExtensions, iWidthAddon, iHeightAddon)* – создание темы оформления для определённых типов.
    - sSkin содержит класс стилей, который будет использован совместно с MediaBox (пример - MediaBoxBlack).
    - aExtensions – массив расширений файлов, к которым будет применён стиль.
    - iWidthAddon – добавочная ширина для окна с медиа-файлом.
    - iHeightAddon – добавочная высота для окна с медиа-файлом.
- *void open(imageUrl, imageWidth, imageHeight)* – открытие окна с медиа-файлом.
    - imageUrl – ссылка на изображение.

- imageWidth – ширина медиа-файла. Если не задана, то окно будет автоматически подстроено под ширину медиа-файла после его загрузки.
- imageHeight – высота медиа-файла. Если не задана, то окно будет автоматически подстроено под ширину медиа-файла после его загрузки.

AMI.UI.Multiselect – замена стандартного <select multiple=”multiple”> на графический контрол.

Класс AMI.UI.Multiselect необходим для замены на странице стандартного select контрола с множественным выбором на графический, которым можно манипулировать стилями. Возможности объекта:

1. Он заменяет собой указанный встроенный контрол, вставая на ту же позицию и сохраняя те же размеры (если не изменена высота ряда). Если задано количество видимых рядов (size=”X”), то контрол адаптируется так же под этот размер.
2. Для окна контрола используется свой класс, так же как и для вариантов.
3. Множественное выделеные делается кликом мыши по каждому варианту. Удерживать кнопку SHIFT нет необходимости.
4. Возможность выделать или снять выделение сразу с нескольких вариантов одним движением мыши, кликнув на первом варианте и перетаскивая указатель мыши над остальными вариантами, удерживая левую кнопку мыши нажатой.

Контрол перед созданием инициализирует все необходимые данные, получает значения из элемента select, к которому он привязан и после этого скрывает элемент select. Тем не менее он манипулирует данными select и при отправке формы происходит отправка данных оригинального select. Подключается AMI.UI.Multiselect к нужному select следующим образом:

```
<select name="someSelect[]" id="idSomeSelect" multiple="multiple" size="5">
   <option value="1">Значение 1</option>
   <option value="2">Значение 2</option>
</select>
<script type="text/javascript">var oSomeField = new AMI.UI.Multiselect('idSomeSelect')</script>
```

Мараметры конструктора AMI.UI.Multiselect(idField, iWidth, iRowHeight) следующие:

- *string selectId* – Обязателен. ID оригинального элемента select.
- *int iWidth* – ширина контрола. Если параметр не задан, то будет задана ширина элемента select, из которого создаётся контрол.

- *int iRowHeight* – Высота одного ряда с вариантом. Если для select задан параметр size, то высота будет адаптирована под необходимое количество рядов.

Классы стилей, служащие для оформления контрола:

- mSelectFrame – окно контрола (DIV).
- mSelectOption – ряд с невыбранным вариантом (DIV).
- mSelectOptionSelected – ряд с выбранным вариантом (DIV).

AMI.UI.List – класс для работы со списком: запрос данных с сервера, разбор ответа, вывод списка по заданному шаблону. Данные для списка загружаются из плагина в Amiro.CMS

- *void showProgress()* – отображение слоя с прогрессом.
- *void hideProgress()* – скрытие слоя с прогрессом.
- *void load(lastAction)* – загрузка содержимого списка с текущими параметрами. lastAction может быть опущет, т.е. нужен только для режима автообновления, по этому параметру система опознаёт какое действие (previous, next, load) следует прекратить в случае отстутствия данных в возвращённом ответе от сервера. При этом, если по результатам загрузки скрывается кнопка следующей страницы, то отправляется сообщение ON_AMI_LIST_NEXT_DISABLE, для открытия кнопки вновь отправляется ON_AMI_LIST_NEXT_ENABLE. Те же сообщения для кнопки предыдущих страниц: ON_AMI_LIST_PREVIOUS_DISABLE и ON_AMI_LIST_PREVIOUS_ENABLE.
- *void onContentReceived(state, content)* – колбэк-функция, вызываемая при получении данных content, от запроса, инициированного load(). Переменная state указывает тип ответа: 1 – успешный, остальные – нет.
- *void setListData(aRowsData)* – заполнение рядов таблицы из массива данных. Перед заполнением каждого ряда отправляется сообщение ON_AMI_LIST_DRAW_ROW с массивом данных для одного ряда в качестве второго параметра.
- *void previousPage()* – запрос на отображения предыдущей страницы.
- *void nextPage()* – запрос на отображение следующей страницы.
- void playPause(forceAction) – переключение между режимом автообновления и остановки. Если указан режим в параметре forceAction (play или pause), то он принудительно установится.

Объект AMI.UI.List является самым частоприменяемым. Далее будет рассмотрен пример создания новостного блока на странице с использованием этого объекта.

Как создать свой HTML блок, отображающий данные модуля новости (news), используя объект AMI.UI.List.

Для использования такого плагина необходимо на странице добавить блок HTML кода, в который контроллер будет вставлять полученные данные.

```
<div id=”ami_resp_outer_News”>
        <div id="amiNavNews">
          <!--Элементы управления, если необходимо, которые не будут затронуты системой, но будут скрыты при запросе на сервер-->
        </div>
        <div id="amiProgressNews">
          <!--Содержимое блока, который отображается в момент запроса данных с сервера (лоадер), если этого блока нет, то будет создан блок по-умолчанию автоматически-->
        </div>
        <div id="amiContainerNews">
          <!--Блок, в котором будет построено содержимое таблицы-->
        </div>
</div>
```

Вместо «News» можно поставить свои идентификаторы, по которым контроллер сможет определить «свои» объекты. Важно чтобы готовые ID не повторялись на странице. Содержимое, которое выводится в блоки по указанным ID или заполняется при создании макета (ID, отмеченные жирным шрифтом заполняются вручную):

- **amiNav{postfix}** – заполняется вручную. Блок с элементами управления списка. Например, предыдущая, следующая страницы, обновить страницу и т.п. В этом блоке возможно создавать элементы с ID amiNavPreviuosNews, amiNavNextNews, amiNavPlayPauseNews. Для элементов с такими ID автоматически будут изменяться классы стилей при необходимости (скриытие, деактивирование и т.п.).
- **amiProgress{postfix}** – заполняется вручную. Блок, который выводится поверх блока с данными в момент, когда идёт загрузка. Если элемента нет на странице, то он создаётся системой автоматически.
- amiContainer{postfix} – заполняется JS API. Блок, который заполняется рядами списка после получения данных.

После этого блока должно быть определение объекта, который инициализируется из класса AMI.UI.List. Пример:

```
<script type=”text/javascript”>
   var parameters = {
      id_plugin: 'ami_ajax_responder',
      module:   'news',
      locale:    'ru'
      order:    'date',
      dir:       'D,
      limit:    '20',
      offset:    '0',
```

```
        id_page: 0
    };

    var oNewsBlock = new AMI.UI.List('News', parameters);
    oNewsBlock.load();
</script>
<script type=”text/html” id=”templateRowNews”>
    <div>@@date@@ / @@header@@</div>
</script>
```

Чтобы включить автообновление блока, в массив parameters нужно добавить следующие переменные:

- refreshInterval – время в секундах между обновлениями
- refreshIntervalMultiplier – множитель для интервала (увеличение в геометрической прогрессии).
- refreshNumber – количество итерация обновления. Максимально – 50
- refreshType – тип перехода на следующую порцию данных: next, previous, reload.

Первым параметром при инициализации объекта указывается постфикс для ID блоков, вторым – параметры, из которых будет составлен первый GET запрос. Сразу после определения объекта ничего не произойдёт, запрос не будет сделан. Для того, чтобы сделать сразу запрос и заполнить данными блоки делается вызов метода load(). Объект контроллера ищет блоки, с которыми он работает по перечисленным выше ID, приставив к ним постфикс, переданный первым параметром. Аналогично ищется шаблон ряда с ID: templateRow{postfix} (в примере это – News, т.е. templateRowNews). Если любой шаблон не будет найден, то он будет считаться равным пустой строке.

Для загрузки списка вызывается метод load(). При загрузке на сервер посылается запрос с указанными при инициализации данными. Сервер формирует строку в формате JSON из массива данных, которые вернул плагин.

Клиентская часть (Javascript) получает данные в формате JSON и декодирует их в Javascript массив. Массив имеет заранее определённый формат:

```
array(
    “code” => 0,
    // статус ошибки, 0 – если не было ошибки”
    // произвольные данные, которые будут доступны для своей обработки
    “data” => array(
            List => array(
                array(‘date’ => ‘2009.12.31’, ‘header’ => ‘Скоро новый год’),
                array(‘date’ => ‘2010.01.12’, ‘header’ => ‘С новым годом!’),
                array(‘date’ => ‘2010.12.31’, ‘header’ => ‘Опять…’)
            )
    )
);
```

После декодирования данных AMI.UI.List строит ряды, открывает и скрывает необходимые блоки, которые были указаны выше. Перед тем как построить из шаблона очередной ряд контроллер посылает сообщение ON_AMI_LIST_DRAW_ROW, параметром которого является массив с данными для шаблона ряда. Для того, чтобы указать свои правила кодирования необходимо зарегистрировать собственный обработчик этого события. Для этого после инициализации дописывается соответственный Javascript код:

```
function newsListListener(aData){
   // Изменение данных для ряда
   aData[‘header’] = ‘<strong>’ + aData[‘header’]  + ‘<strong>’;
   return aData;
}
AMI.Message.addListener('ON_AMI_LIST_DRAW_ROW', newsListListener);
```

Обработчиков для одного сообщения может быть сколько угодно много. Так же можно переопределить функцию, которая строит все ряды, чтобы самостоятельно заполнить блок с данными модуля, имея полученные от сервера данные:

```
oNewsBlock. setListData= function(aAllRowsData){
   this.oListContent = ‘готовое содержимое блока списка’;
}
```

• jQuery

В комплект поставки Amiro.CMS входит библиотека jQuery версии 1.6.1.

В разделе администрирования сайта, доступ к функциям библиотеки возможен через изолированный объект AMI.$, полностью соответствующий стандартному объекту jQuery той же версии. На публичной части сайта этот объект недоступен, поэтому, настоятельно не рекомендуется использовать объект AMI.$ в тех участках кода, которые могут попасть на публичную часть.

Пример:
```
AMI.$(document).ready(function(){
    alert('Hello world!');
});
```

Изоляция имени необходима для предоставления возможности подключения пользователем любой другой версии библиотеки jQuery и исключения конфликтов, связанных с разницей версий.

Дополнительная библиотека jQuery UI не входит в поставку, и не может работать с объектом AMI.$. Для того, чтобы использовать jQuery UI на стороне раздела администрирования, ее необходимо подключить вручную, и в месте ее подключения определить стандартные объекты jQuery:
```
<script>
window.jQuery = window.$ = AMI.jQuery;
</script>
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.13/jquery-ui.min.js"></script>
```

Для того, чтобы использовать библиотеку jQuery на публичной части, ее необходимо подключить вручную, для каждого из существующих макетов. Сделать это можно в разделе “Макеты страниц”, на вкладке “Код для HEAD”.

При этом, для экономии трафика и ускорения времени загрузки сайта, рекомендуется подключать jQuery из файловых хранилищ Google или Яndex, в зависимости от аудитории вашего сайта.

Подключение из хранилища Google:
http://code.google.com/intl/ru/apis/libraries/devguide.html

Подключение из хранилища Яndex:
http://api.yandex.ru/jslibs/libs.xml

Пример подключения jQuery версии 1.4.2 из хранилища Google:
```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"></script>
```