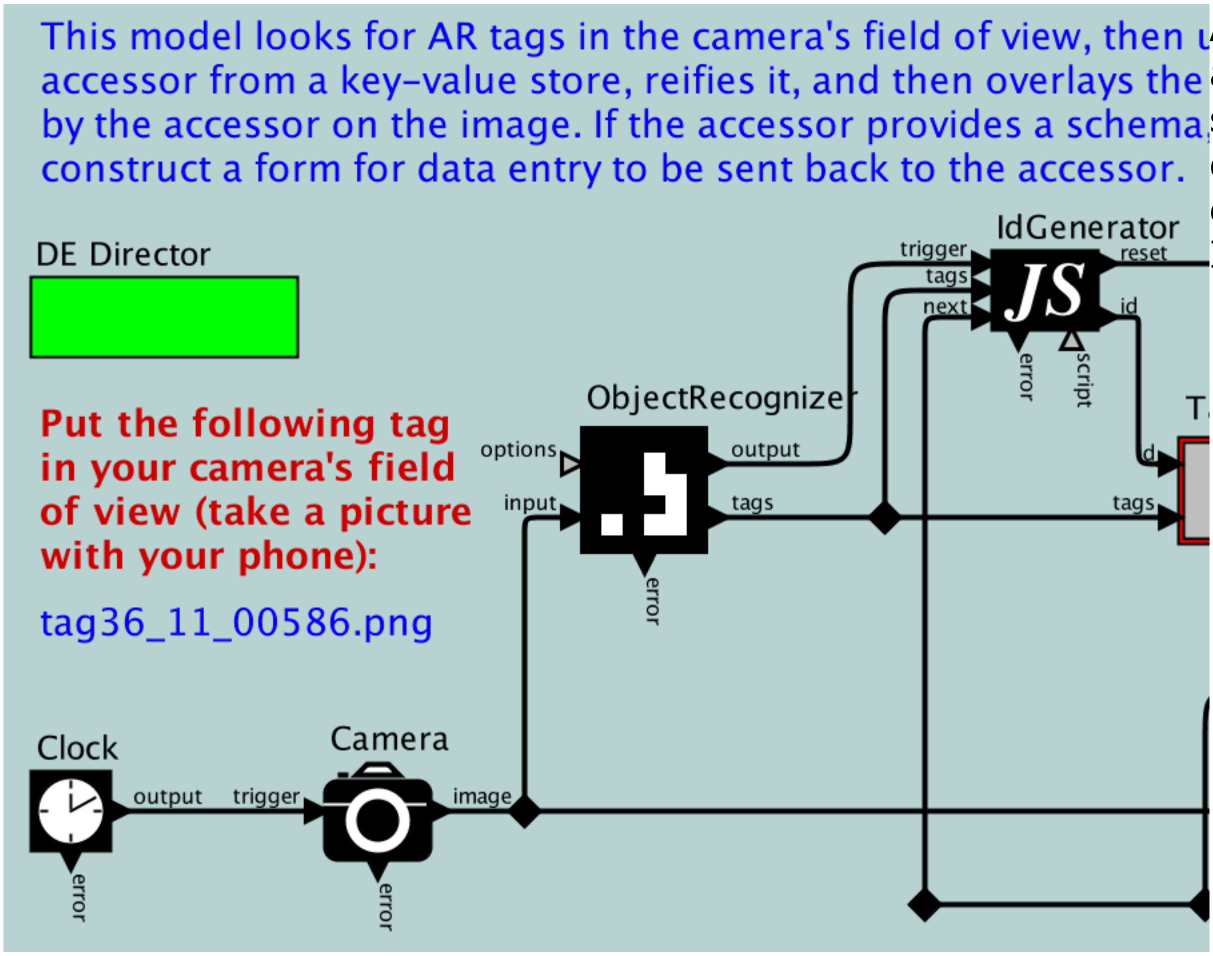
Accessors

What are Accessors?


This model looks for AR tags in the camera's field of view, then u
accessor from a key-value store, reifies it, and then overlays the
by the accessor on the image. If the accessor provides a schema,
construct a form for data entry to be sent back to the accessor.
DE Director
Put the following tag in your camera's field of view (take a picture with your phone):
tag36_11_00586.png
IdGenerator
trigger
tags
next
reset
id
error
script
ObjectRecognizer
options
input
output
tags
error
T
id
tags
Clock
output
error
Camera
trigger
image
error

Access
accessil
service
develop
enable
Interne

An **actor** is a software component that reacts to streaming input events and produces streaming output events. An **accessor** is an actor that wraps a (possibly remote) device or service in an actor interface. An **accessor host** is a service running in the network or on a client platform that hosts applications built as a composition of accessors that stream data to each other. The host is like a browser for things.

See the overview presentation. For a quick start using Node.js as a host, see the Node host.
Accessors embrace **concurrency**, **atomicity**, and **asynchrony**. The actor model, which governs interaction

between accessors, permits accessors to execute concurrently with segregated private data and a message-passing interface for interaction. Internally, many accessors use **asynchronous atomic callbacks** (**AAC**) to invoke remote services and handle responses asynchronously and atomically. See comparisons with related technologies for insight into how accessors work.

Accessors are defined in a JavaScript file that includes a specification of the **interface** (inputs, outputs, and parameters) and an implementation of the **functionality** (reactions to inputs and/or production of outputs). Any JavaScript file that conforms with the **accessor specification** defines an **accessor class**.

The TerraSwarm accessor library provides a collection of example accessors. This library is maintained via an SVN repository that permits many contributors to add accessors to the library.

An **instance** of an accessor is created by a **swarmlet host** that evaluates the JavaScript in the accessor definition. At this time, there are at least three accessor hosts compatible with **accessor specification 1.0**:

- A **browser host**, which allows inspection of the accessor, and if the accessor is suitable for execution in a browser, interactive invocation of the accessor.
- A **Node.js host**, an interactive program that runs in Node.js that allows instantiation and execution of accessors.
- A **Ptolemy II host**, which supports composition of accessors with visual block diagrams and provides a large library of actors that the accessors can interact with.

To experiment with accessors now, see Getting Started with Accessors.

This work is supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work is also supported by the

1. Minimal Accessor Example

Accessors that are to be checked in the `accessors` repository must follow the Ptolemy Coding Style, which includes the license and JSHint directives. However, it is not required that accessors include those features, so they are documented elsewhere. Below is a description of the minimal accessor.

A minimal accessor that takes a numeric input, doubles it, and sends the result to an output is given below:

```
exports.setup = function () {
    this.input('input');
    this.output('output');
};
exports.fire = function () {
    this.send('output', this.get('input') * 2);
};
```

This accessor has two of the four parts, an interface definition in the `setup()` function, and a specification of the function to be performed. In this case, the function is specified by defining a `fire()` function, which will be invoked by the host whenever a new input is provided, and possibly at other times. To specify a function that is performed only exactly when a particular new input named 'input' is provided, use an input handler, as in this example:

```
exports.setup = function () {
    this.input('input');
    this.output('output');
};
exports.initialize = function () {
    var self = this;
    this.addInputHandler('input', function () {
        self.send('output', self.get('input') * 2);
    });
};
```

In this case, the function to be performed is given by an input handler function. Notice the commonly used JavaScript idiom, where the value of the variable `this` is captured in a variable called `self` (this name is arbitrary), and then within the callback function, `self` is used instead of `this`. In JavaScript, when a function is invoked, the value of the variable `this` is the object on which the function is invoked. The `setup` and `initialize` functions are invoked by the host on the accessor, which has functions `input`, `output`, etc., defined. But the input handler function is not assured of being invoked on the accessor. Any callback function that the accessor defines, such as the function passed to `addInputHandler` above, should be written in such a way that it will work no matter what object the host invokes it on. A browser host, for example, may invoke it with `this` equal to the window object. To understand the `this` keyword in Java, see

- http://www.quirksmode.org/js/this.html
- http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

More interesting examples are found on the TerraSwarm accessor repository.

2. Interface Definition

An **accessor interface** specifies what it requires of an accessor host and what its inputs, outputs, and parameters are. An accessor interface definition may include the following elements in the accessor specification file:

- require: Modules required by the accessor.
- An `exports.setup()` function that can invoke the following functions:
  - input: Specify an input to the accessor.
  - output: Specify an output from the accessor.
  - parameter: Specify a parameter for the accessor.
  - implement: Declare that the accessor implements another interface.
  - extend: Declare that the accessor extends another accessor.

Inputs, outputs, and parameters may optionally have specified data types. The `exports.setup()` function may also invoke `instantiate()` and `connect()` as explained below in the composition section. All of the above except `require()` are functions of the accessor, so they should be invoked with the prefix `this` as illustrated above. The `require()` function, on the other hand, is a top-level function that can be invoked from anywhere. See Top-Level JavaScript Functions for a complete list of such functions.

3. Accessor Documentation

Documentation and metadata such as author, version, etc., are given using JSDoc tags in comments in the JavaScript accessor specification file. For example, the above minimal accessor might be documented as follows:

```
/** Double the value provided at *input* and send to
*output*.
 *  @accessor Minimal
 *  @author Edward A. Lee (eal@eecs.berkeley.edu)
 *  @input {number} input A numeric input.
 *  @output {number} output The output for the doubled
value.
 */
```

The main body of the documentation is given after the first `/**` in the accessor specification file. It can include any formatting commands supported by Markdown. For example, above, `*input*` will be rendered in italics. The documentation may include any of the following tags, in alphabetic order:

- @accessor *Name*: The name of the accessor. This should be capitalized and should match the name of the accessor file (without the .js extension). This is an accessor-specific tag used by the JSDoc Ptdoc Plugin
- @author *Author*: The author(s) of the accessor. This is arbitrary text and may optionally include contact information. See the JSDoc @author documentation.
- @input {type} *name*: A description of the input with the specified name and an optional type. If the input has a default value, that should be explained in the text. See JSDoc @type tag documentation for formatting of the type. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @module *Name*: The name of the accessor again, if the accessor is also available as a module. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @output {type} *name*: A description of the output with the specified name and an optional type. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @parameter {type} *name*: A description of the parameter with the specified name and an optional type. The default value should be explained in the text, if given. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @version *version*: A version designator for the accessor. For example, if the accessor is stored under an SVN version control system, this might specify `$$Id:$$`, which SVN will automatically replace with the current version number. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin. Note that to avoid issues with expanding $, we use `$$Id$$` instead of `$Id$`.

See the Accessor JSDoc page for how to invoke JSDoc.

4. Functionality

In addition to an interface, an accessor may have some functionality given in JavaScript. The script can define local state variables used by the accessor and functions to be invoked by the **swarmlet host**. The accessor's script will be executed in an **accessor context** provided by the host. The context provides some built-in functions and modules that are available for accessors to use. A swarmlet host *must* provide all the built-in functions and modules and *may* provide some or all of the optional modules. The set of required functions and modules is deliberately small, so that implementing a basic accessor host is easy.

Accessors run in a JavaScript host that provides a standard set of functions and modules that enable the accessor to interact with devices and services.

- Index of functions for quick reference.
- Action Functions: A set of functions that an accessor may define that are invoked by the host to execute a swarmlet. The swarmlet host will always invoke these in such a way that the variable `this` is the accessor itself, which defines the functions such as `input`
- Top-Level JavaScript Functions: These function enable the accessor to get inputs and produce outputs. They also provide a small set of basic mechanisms that are commonly found in a JavaScript environment.
- Built-In JavaScript Modules: All accessor hosts are required to support all built-in modules. Modules conform with the CommonJS Module Specification.
- Optional JavaScript Modules: More sophisticated capabilities that may or may not be supported by a particular accessor host. These modules must also conform with the CommonJS Module Specification specification.
- JSDoc, auto-generated code documentation.

5. Composition

- instantiate: Instantiate a contained accessor (see Composition).
- connect: Connect one or two contained accessors (see Composition).

An accessor may contain other accessors. To instantiate an accessor within another , use instantiate, as illustrated in the following example:

```
exports.setup = function() {
    var gain = this.instantiate('TestGain',
'test/TestGain');
    gain.setParameter('gain', 4);
}
```

[$[Get Code]]

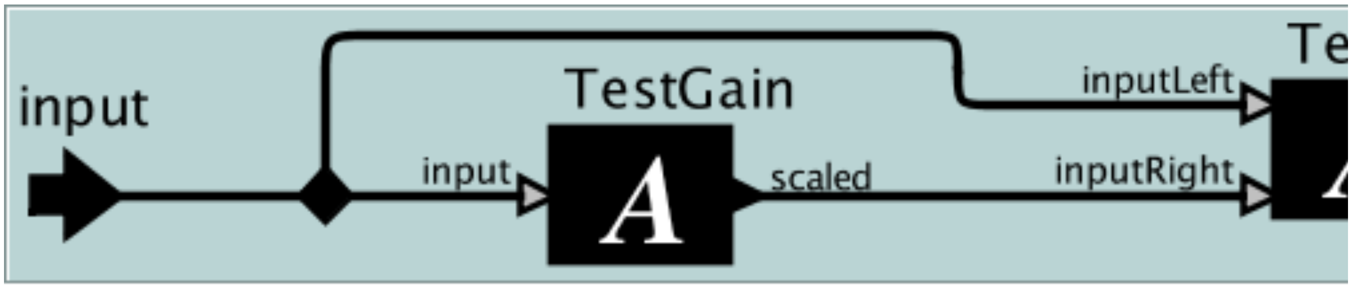
This example instantiates a contained accessor `test/TestGain`, which is a test accessor from the TerraSwarm library that simply multiplies its input by a constant. It has a *gain* parameter, which in this case is being set to the constant 4. It could also be set to the parameter value of the composite using `getParameter()` (see parameter).

Any contained accessor will be initialized and wrapped up whenever its container is initialized and wrapped up (see execution below). But contained accessors are most useful if their inputs and outputs are connected to other contained accessors or to inputs and outputs of the container accessor. Connections are made using the connect function, as illustrated by the following example:

```
exports.setup = function() {
    this.input('input', {'type':'number', 'value':0});
    this.output('output', {'type':'number'});
    var gain = this.instantiate('TestGain',
'test/TestGain');
    gain.setParameter('gain', 4);
    var adder = this.instantiate('TestAdder',
'test/TestAdder');
    this.connect('input', adder, 'inputLeft');
    this.connect('input', gain, 'input');
    this.connect(gain, 'scaled', adder, 'inputRight');
    this.connect(adder, 'sum', 'output');
}
```

[$[Get Code]]

This connects two contained accessors (an instance of `test/TestGain` and one of `test/TestAdder`) as shown here:



- Container - Accessors hosts are not required to provide access to the container

6. Execution

An accessor is executed by a **swarmlet host**, a program that includes a JavaScript interpreter and provides the script with a collection of library functions and modules. All interaction between the accessor script and the network, devices, the file system, and other accessors is mediated by these libraries. Hence, the host controls what the accessor script has access to. Not all hosts can execute all accessors. The browser host, for example, limits network and file system access according to a browser security policy. When an accessor is instantiated by a host, the host checks compatibility and reports if the accessor is incompatible with that host.
The life of an accessor has four phases:

- instantiation: The host reads and executes the JavaScript file defining the accessor. If in that file an `exports.setup` function is defined, then that function will be executed in this phase.

- initialization: If the accessor defines an `exports.initialize` function, then that function is invoked in the initialization phase. This marks the beginning of the execution of a swarmlet.
- reaction: The accessor reacts to inputs and callbacks, as explained in detail below.
- wrapup: If the accessor defines an `exports.wrapup()` function, then that function is invoked in the wrapup phase. This marks the end of the execution of the swarmlet using the accessor.

The host instantiates an accessor exactly once. After instantiation, execution of the accessor consists of initialization, zero or more reactions, followed by wrapup. This execution pattern may be repeated multiple times.

A **reaction** of an accessor is triggered by the swarmlet host. When the host does this depends on the host. The Browser Host for example, will trigger a reaction when the user clicks a button labeled 'react to inputs' on a web page displaying the accessor. The Node Host will trigger a reaction when the script being executed invokes the `react()` function on the accessor instance. The Cape Code Host will trigger a reaction when an input to the accessor arrives.

A reaction performs exactly the following actions, in order:

- Invocation of any timeout callbacks, if the time is appropriate.
- Invocation of any registered handlers for any input that has received new data. These handlers will be invoked first in the order in which inputs are defined, and then, if an input has multiple handlers, in the order in which the handlers were added.
- Invocation of any registered handlers for new input data that the accessor has sent to itself. An accessor will continue checking for self-provided new input data until none is available.
- Invocation of any handlers that have been registered to be invoked when *any* new input arrives. These handlers will be invoked in the order in which they were added.
- Reaction of any contained accessors that have been provided with inputs (see below).
- If the accessor defines an `exports.fire` function, then that function is invoked now.

In the above, "new input" means input that has arrived from an external source since the last reaction. Inputs sent to the accessor by itself are handled in the same reaction.

A host is free to invoke a reaction whenever it chooses after it has invoked `initialize()` and before it has invoked `wrapup()`, even if there are no new inputs. Hence, anything the accessor does in its `fire()` function should make sense regardless of whether new inputs have been provided.

When invoking reactions of contained accessors, the swarmlet host follows a specific deterministic model of computation. First, contained accessors are

assigned a **priority** based on their position in the graph of connected accessors. An accessor *A* with an output connected to accessor *B* will have a higher priority than *B*, unless the output of *A* is declared to be **spontaneous** (see output). The host will invoke reactions in priority order, ensuring that any accessor that feeds data to another accessor will react first.
An accessor may invoke `stop` which will invoke `wrapup()` on any contained accessors and then stop the host.

Accessor Functions

The following functions should be invoked as `a.f()`, where `a` is the accessor and `f` is the function name. To invoke from within one of the accessor functions like an input handler, use `this.f()`.

- addInputHandler: Add a function to handle new inputs.
- connect: Connect one or two contained accessors (see Accessor Specification#Composition).
- extend: Declare that the accessor extends another accessor.
- get: Retrieve the value of an input.
- getParameter: Retrieve the value of a parameter.
- input: Specify an input to the accessor.
- implement: Declare that the accessor implements another interface.
- instantiate: Instantiate a contained accessor (see Accessor Specification#Composition).
- output: Specify an output from the accessor.
- parameter: Specify a parameter for the accessor.
- removeInputHandler: Remove an input handler.
- send: Send an output or to the accessor's own input
- setDefault: Set the default value of an input.
- setParameter: Set the value of a parameter.
- stop: Request that the enclosing swarmlet stop execution.

Functions Exported by an Accessor

The following functions are invoked by the host on the accessor (so that `this` will be the accessor, and `this.f()` can be used to invoke any of the above functions with name `f`).

- fire: Fire the accessor.
- initialize: Initialize the accessor.
- latestOutput: Return the most recent output sent via the named output.
- provideInput: Provide to a named input the given value.
- react: React to any inputs that have been provided.

- setParameter: Set a parameter with the given name to the given value.
- setup: Set up the actor interface.
- wrapup: Wrap up execution of the accessor.

Top-Level Functions

The following function can be invoked in any context by the accessor simply as `f()`, where `f` is the function name.

- clearInterval: Cancel a function being invoked periodically.
- clearTimeout: Cancel a function pending invocation after a delay.
- error: Signal an error.
- getResource: Get a resource from the swarmlet host.
- require: Specify a module required by the accessor.
- setInterval: Specify a function to be invoked periodically.
- setTimeout: Specify a function to be invoked after a delay.

http-client Module

Library for making HTTP requests.

Cross-origin requests are generally prohibited for scripts (and, therefore, accessors) running in a browser. See Browser Considerations for details.

API

- **get**(*url* | *options*, *responseCallback*): Make an HTTP GET request and call the callback when the request has been satisfied.
- **post**(*url* | *options*, *responseCallback*): Make an HTTP POST request and call the callback when the request has been satisfied.
- **put**(*url* | *options*, *responseCallback*): Make an HTTP PUT request and call the callback when the request has been satisfied.
- **request**(*url* | *options*, *responseCallback*): Make an HTTP request and call the callback when the server has responded.

In all cases, the first argument is either a string with a URL or a JavaScript object with a number of fields defined below. Also, in all cases, the *responseCallback* function, if provided, will be called when the request has been fully satisfied. The *responseCallback* function will be passed one argument, an instance of the **IncomingMessage** class, defined below. The difference between **request**() and the other functions is that the former only begins the interaction with the server, whereas the rest are complete, self-contained requests. Specifically, if you use **request**, you can use the returned object to send additional information to the server, for example to PUT a file on the server. Both functions return an instance of the **ClientRequest** class, defined below.

Usage

A simple use to read the contents of a web page looks like this:

```
  var http = require('@accessors-modules/http-client');
http.get('http://ptolemy.eecs.berkeley.edu', function(response) {
console.log('Got response: ' + response.body);   });
```

To view all returned parameters from http.get():

```
  var http = require('@accessors-modules/http-client');
http.get('http://ptolemy.eecs.berkeley.edu', function(response) {
console.log('Got response: ' + JSON.stringify(response));   });
```

POST and PUT requests typically specify a body, using options.body. (This URL will give an error since it doesn't allow posting).

```
  var http = require('@accessors-modules/http-client');   var options =
{      body : {test : 'this is a test'},      url :
'http://ptolemy.eecs.berkeley.edu'   };   http.post(options,
function(response) {     console.log('Got response: ' +
JSON.stringify(response));   });
```

Options

The *options* argument to the above functions can be a string URL or a JavaScript object with the following (optional) fields:

- **body**: An object containing the request body. Images are supported in CapeCode only.
- **headers**: An object containing request headers. By default this is an empty object. Items may have a value that is an array of values, for headers with more than one value.
- **keepAlive**: A boolean that specified whether to keep sockets around in a pool to be used by other requests in the future. This defaults to false.
- **method**: A string specifying the HTTP request method. This defaults to 'GET', but can also be 'PUT', 'POST', 'DELETE', etc.
- **outputCompleteResponseOnly**: A boolean specifying if only a complete response is allowed, or if multi-part responses are acceptable.
- **timeout**: An integer specifying the maximum time to wait for a response, in milliseconds. Defaults to 5000 ms.
- **url**: A string that can be parsed as a URL, or an object containing the following fields:
    - **host**: A string giving the domain name or IP address of the server to issue the request to. This defaults to 'localhost'.

  - **path**: Request path as a string. This defaults to '/'. This can include a query string, e.g. '/index.html?page=12', or the query string can be specified as a separate field (see below). An exception is thrown if the request path contains illegal characters.
  - **protocol**: The protocol. This is a string that defaults to 'http'.
  - **port**: Port of remote server. This defaults to 80.
  - **query**: A query string to be appended to the path, such as '?page=12'.

IncomingMessage

The *responseCallback* function, if provided, will be passed an instance of **IncomingMessage**, which has these fields:

- **statusCode**: an integer indicating the status of the response. A description of HTTP response codes can be found here: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Codes in the 300 range indicate redirects, and codes 400 or above indicate errors.
- **statusMessage**: a string with the status message of the response.
- **body**: a string with the body of the response.

FIXME: At this time, only UTF-8-encoded string bodies are supported. Also, by including the body of the response in the argument to the callback, rather than a socket, we limit the size of the body. A large response (e.g. a video file) could easily overwhelm available RAM memory.

ClientRequest

The *request* and *get* functions both return an instance of ClientRequest, which has the following functions:

- **end**(): Call this to end the request (no more data to send to the server). The **get**() function automatically calls this for you, but **request**() does not.
- **write**(*data*, *encoding*): Write data (e.g. for a POST request) using the specified encoding (e.g. 'UTF-8').

The ClientRequest class is an event emitter that emits the following events:

- 'error': If an error occurs. The message is passed as an argument.
- 'response': A response is received from the server. This event is automatically handled by calling responseCallback, if responseCallback is not null, but you can also listen for this event.

Browser Considerations

Cross-Origin Requests

For security reasons, scripts in a browser are generally prohibited from certain types of data transfer in cross-origin requests. The same-origin policy states that two pages have the same origin if the protocol, port (if given), and host are the same for both pages.

Client-server coordination is required to achieve cross-origin communication. Two techniques are cross-origin resource sharing (CORS) and JSON with padding (JSONP).

Cross-Origin Resource Sharing (CORS)
A server may stipulate that cross-origin requests are allowed. This is called cross-origin resource sharing (CORS).

- For simple requests, a CORS-aware browser automatically adds an `Origin` request header specifying the host. For example:

```
Origin: http://www.terraswarm.org
```

A server sets the response header `Access-Control-Allow-Origin` to the list of hosts allowed access, using * for all hosts. For example:

```
Access-Control-Allow-Origin : *
```

or

```
Access-Control-Allow-Origin : http://www.terraswarm.org
```

The client checks this header. If access is allowed, the content is loaded.

- For non-simple preflighted requests, the client first sends an HTTP OPTIONS request with additional information.
- History and more examples are available here.

JSON with padding (JSONP)
The JSON with padding technique takes advantage of the fact that `<script>` tags in HTML are exempt from cross-origin restrictions. For example, this is allowed:

```
<script src="http://www.otherdomain.com/thescript.js"> </script>
```

In a JSONP request, the client specifies some text to pre-pend to a JSON response. The server replies with the text plus the JSON data enclosed in parenthesis. If the client defines a function locally and sends the function name as the text, this function is then executed with the JSON data as an argument.

For example, calling http://ip.jsontest.com/ will return your IP address in JSON form. Example:

```
{"ip": "8.8.8.8"}
```

A client could define a function that displays data:

```
function showIP(data) {   alert(data); }
```

Next, the client asks the server to prepend this function name to the returned JSON by appending `?callback=[function_name]` to the request URL. For example, calling http://ip.jsontest.com/?callback=showIP will return this response:

```
showIP({"ip": "8.8.8.8"});
```

The `showIP` function then executes with the returned data as an argument.

jQuery's `getJSON()` function will issue a JSONP request if the URL ends in `?callback?`. For example:

```
jquery.getJSON("http://jsonplaceholder.typicode.com/posts/1?callback=?"
, function(data) {    console.log(JSON.stringify(data)); });
```

Here's another example.
Here's the original JSONP idea.

APIs and Libraries

XMLHttpRequest

XMLHttpRequest is an API that allows a client to fetch and send data without having to do a full page refresh. All modern browsers (Chrome, IE7+, Firefox, Safari, and Opera) include a built-in XMLHttpRequest implementation. XMLHttpRequest supports:

- Both asynchronous and synchronous requests
- Both text and binary data
- Event listeners
- Form submission and file upload

Please see Mozilla's XMLHttpRequest for a complete list of methods and browser support, and Using XMLHttpRequest for examples.

jQuery

jQuery is a "fast, small, and feature-rich JavaScript library" that is supported by modern browsers. jQuery defines an easy-to-use API for common HTML manipulation and data transfer tasks.

jQuery includes an AJAX (Asynchronous Javascript and XML) method that wraps XMLHttpRequest for issuing requests. There is also a shorthand GET method.

Browserify

Browserify is a bundling system and set of modules that replicates most node.js APIs and functionality in a browser. Browserify allows one to use the "requires" syntax in a browser and creates a bundle of all modules needed for a particular piece of code by collecting all of the "require" dependencies. For example:

```
  browserify main.js > bundle.js
```

will recursively locate all dependencies starting in the `main.js` file and output flattened Javascript to `bundle.js`. The HTML page then includes this bundle using a script tag:

```
<script src="bundle.js"> </script>
```

Browserify includes both HTTP and HTTPS modules. These modules depend on many others, so accessors that use browserify would need to either include bundles or have the browserify source code available in the accessor repository or other location.

jQuery-browserify

There is a browserify version of jQuery, jQuery-browserify. Unfortunately jQuery-browserify has no license, so the terms of use are unknown.