ARCADA

Modular JavaScript

A Comparison of Module Loaders

Matias Rask

Degree Thesis

Information Technology

2012

Matias Rask

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	3728
Författare:	Matias Rask
Arbetets namn:	Modular JavaScript A Comparison of Module Loaders
Handledare (Arcada):	Jonny Karlsson
Uppdragsgivare:	Nord Software

Sammandrag:
Det här examensarbetet handlar om hur modulärprogrammering kan implementeras i JavaScript som saknar ett nativt modulsystem. JavaScript har vuxit från ett simpelt skriptspråk till ett fullt utvecklat programmeringsspråk för webbapplikationer. På grund av att webbapplikationer kan innehålla flera tusen rader kod, är det viktigt att utvecklarna kan strukturera och spjälka upp källkoden för att underlätta upprätthållandet av applikationen. Examensarbetet omfattar en teknisk beskrivning på hur JavaScript-baserade webbapplikationer körs i webbläsare samt en teoretisk och en praktisk jämförelse mellan modulladdarna Dojo Toolkit 1.6, Googles Closure library och Asynchronous Module Definition (AMD). En experimentell applikation har programmerats med hjälp av ovannämnda modulladdare för att kunna testa och jämföra de praktiska egenskaperna.

Nyckelord:	Dojo Toolkit, Closure library, Asynchronous Module Definition, AMD, Nord Software, JavaScript.
Sidantal:	34
Språk:	Engelska
Datum för godkännande:	20.11.2012

DEGREE THESIS	
Arcada	
Degree Programme:	Information technology
Identification number:	3728
Author:	Matias Rask
Title:	Modular JavaScript A Comparison of Module Loaders
Supervisor (Arcada):	Jonny Karlsson
Commissioned by:	Nord Software

Abstract:
This thesis covers modular programming and how it can be implemented in JavaScript which lacks a native module system. JavaScript has grown from a simple scripting language into a complete programming language for web applications. Since web applications can include thousands of lines of code it is important for developers to be able to structure and split up application source code into more manageable modules to simplify the application maintenance. In this thesis, a description of how JavaScript web applications are executed in a web browser and a theoretical and a practical comparison of the module loaders Dojo Toolkit 1.6, Google Closure library and Asynchronous Module Definition (AMD) is provided. A sample application has been made with the help of the above mentioned module loaders to compare and test the practical differences.

Keywords:	Dojo Toolkit, Closure library, Asynchronous Module Definition, AMD, Nord Software, JavaScript.
Number of pages:	34
Language:	English
Date of acceptance:	20.11.2012

CONTENTS

LIST OF ACRONYMS

AMD – Asynchronous Module Definition

API – Application Programming Interface

CDN – Content Delivery Network

CPU – Central Processing Unit

GB – Giga Byte

GUI – Graphical User Interface

HTML – HyperText Markup Language

HTTP – Hypertext Transfer Protocol

IE – Internet Explorer

KB – Kilo Byte

OS – Operating System

RAM – Random-access Memory

UI – User Interface

URL – Uniform Resource Locator

XHR – XMLHttpRequest

LIST OF FIGURES

LIST OF TABLES

1 INTORUDCTION

This thesis is done for Nord Software which is a Finnish software company with global customers. Nord Software's expertise lies within web-technologies with JavaScript being the main programming language for the client platform. Nord Software tries to be on the edge when it comes to knowledge, understanding and implementation of JavaScript.

The Asynchronous Module Definition (AMD) (Burke 2012) is a JavaScript tool that has recently attracted the company's attention. A major limitation of JavaScript is its lack of native modules and package systems and AMD is one approach for adding these features.

This thesis discusses the limitations of JavaScript related to modules and examines different modular implementations with special focus on AMD.

1.1 Background

JavaScript started as a scripting language for the web browsers in 1997. Although the development of the language started two years earlier, 1997 was the year when the language was launched. In the early days of JavaScript its main focus was to bring some slight interactivity to the user interface (UI) (Champeon 2001). Today a web site/application might utilize thousands of lines of JavaScript code with several developers contributing to the code base. The transition of JavaScript from a simple scripting language to a complete programming language has brought a number of problems for developers creating larger applications with JavaScript. The main problem is how to write reusable and maintainable code.

Organizing code into classes and modules makes the code more reusable in various situations. A module is regarded as a self-contained, reusable unit of code that can be easily distributed and used by different programs. Modules generally represent a separation of concerns and improve maintainability of the code base. At the moment JavaScript lacks the native functionality to define and load modules, but the next version of JavaS-

cript will have a module system implemented (harmony:modules 2011). It will take years before the next version of JavaScript has gained an adequate amount of users so that the support of legacy systems (web browsers of today) is not needed anymore.
Many JavaScript libraries and frameworks include some kind of module system, but all of these implementations have their own syntax and are not compatible with other implementations. Until the next version of JavaScript the AMD specification has become the *de facto* standard for JavaScript modules.

1.2 Aims and Goals

This thesis describes how JavaScript is interpreted by the web browsers and what problems developers face due to the asynchronous nature of the web. The goal of this thesis is to examine different modular implementations of JavaScript and their uses. Test cases for each implementation are presented and compared. A sample application is made with the help of each examined module loader to compare and test the practical differences. The tests include a comparison of module loading speed and module dependency loading techniques.

2 JAVASCRIPT IN WEB BROWSERS

2.1 Web Documents and Web Applications

Web pages can be split into two sections, web documents and web applications. In web documents JavaScript is unobtrusive and is not needed for the web page to function correctly. JavaScript is most often used to enhance the user experience by e.g. creating animations and other visual effects.

In addition to the JavaScript features that web documents use, web applications also use the services provided by the web browser environment. Web applications are possible because the web browsers have grown beyond their original role as tools for displaying documents and are moving towards simple operating systems (OS). An OS usually allows the user to organize icons, files and folders via its graphical user interface (GUI). An analogue to this in web browsers is the bookmarking system where the user can organize bookmarks in folders where bookmarks represent web documents and web applications. An OS can run multiple windowed programs where a web browser can have multiple documents and web applications running in multiple tabs. An OS defines low-level APIs for networking, data storage, graphics and all these features are found in web browsers today to some extent.

As the web browsers work as simplified OS, web applications can access the underlying services that the web browsers provide with the use of JavaScript. Perhaps the best known of these services is the XMLHttpRequest (XHR) object (XMLHttpRequest 2012), which gives the JavaScript access to networking through HTTP requests. Web applications use this service to retrieve data from the server without the need of a page reload.

The new HTML5 specification (Pieters 2012) defines some additional APIs for web applications, such as data storage (file API, database API) and background threads. As JavaScript runs in a single thread any central processing unit (CPU) time consuming computation will freeze the rest of the web application, including the UI. Threads were in-

troduced in HTML5, known as WebWorkers, which allow code to be run in the background.

JavaScript is more crucial for web applications as it is for web documents as web documents should function without JavaScript. As web applications use the OS services provided by the browser, a web application is not expected to work without JavaScript. (Flanagan 2011 p. 310)

2.2 Embedding JavaScript

JavaScript can be embedded into a web page in four ways:

- Inline between opening and closing *script* tags
- From an external file with the help of the *script* tags *src* attribute
- In a HTML elements event handler attribute
- With the *javascript:* URL protocol

The *script* tags also have a type attribute, which tells the web browser what kind of scripting language lies between the tags. Internet Explorer (IE) is the only browser supporting the Microsoft's proprietary VBScript scripting language. If VBScript is used the type attribute needs to be *text/vbscript*. In all modern web browsers the *type* attribute defaults to *text/javascript*, therefore when implementing JavaScript in a web page the type attribute is not needed.

2.2.1 The <script> Element

The *script* tags have an optional source (*src*) attribute that specifies a URL of the script file to be loaded. Usually a JavaScript file loaded this way contains only JavaScript without the *script* tags or any other HTML code. The script loaded via the *src* attribute behaves exactly the same way if the content of the script file would be between the opening and closing *script* tags. When loading script files with the *src* attribute any code between the opening and closing *script* tags is ignored. If the *scr* attribute is not used then the JavaScript code between the opening and closing tag is parsed and executed.

By loading script files with the *src* attribute versus inline script there are several advantages:

- Program logic gets separated from the presentation markup, i.e. JavaScript is separated from the HTML.
- If multiple HTML documents refer to the same code the need to make changes in several HTML documents when the code changes is eliminated.
- When several HTML documents refer to the same script file it is only downloaded once as the web browser usually caches the file.
- The HTML document can originate from one server and the script files can be loaded from some other servers.
- Combining the cache mechanism of the browser and the ability to load the scripts from external servers makes the use of content delivery networks (CDN) feasible (Nygren et al. 2010). If the web page uses some well used JavaScript library, which is deployed from a CDN, there is a chance that the script files are already cached by the web browser.

2.2.2 HTML Element Event Handler Attribute

To provide some interactivity on a web page JavaScript functions need to be bound to events. Such events can be clicking a HTML element, submitting a form etc. One way to call a JavaScript function is to assign a function to a HTML element event handler attribute i.e. JavaScript inside an elements *onclick*, *onmouseover* or *onchange* attribute is executed when the respective event fires.

Event handler attributes defined in HTML has the drawback that they combine program logic with presentation and possible code change can be tedious, as mentioned in the previous section. Alternative to this method of event binding is to use the web browser's JavaScript method *addEventListener*. The *addEventListener* method is available to almost every browser, except the Internet Explorer (IE) family prior to version 9. Earlier versions of IE use the *attachEvent* method for event binding. These event binding methods offer the approach to separate program logic from presentation.

2.2.3 The URL Protocol

One way to execute JavaScript code is to use the *javascript:* URL protocol. This protocol is rarely used in a document but is useful when implemented as a bookmarklet which is a JavaScript code block saved as a bookmark in the browser. When this bookmarklet is launched it executes the code with the current web page as its context (Flanagan 2011 p. 311). Bookmarklets can be used to run general debugging scripts or games e.g. an Asteroids clone where the user controls a space ship to destroy visible HTML element on the web page.

2.3 JavaScript Execution

A JavaScript program consists of all the JavaScript code in the web page regardless the way the code has been inserted. All of these separate chunks of code share the single global window object and thus share all the same global objects, variables and functions. If two JavaScript code blocks define the same global object then the code block that is executed last will overwrite the earlier object. The fact that almost everything in JavaScript is mutable and bound to the global object is something that can have negative side effects. For example inclusion of several different JavaScript libraries, where both might define some global helper functions with the same name, can break the program as the API might have changed.

2.3.1 How is JavaScript Executed in the Web Browser?

There are two phases of JavaScript execution in the web browser environment. In phase one, asynchronous phase, the document content is loaded and code, both inline and external via the *src* attribute, from the *script* tags is executed in the order as the script tags appear in the HTML document.

In the second phase the JavaScript interpreter enters an asynchronous, event-driven, mode. In this mode the JavaScript interpreter waits for the web browser invoked events defined via the HTML element event attributes or events that are registered via the *addEventListener* or *attachEvent* functions.

2.3.2 Timeline

The timeline of JavaScript execution in web browsers is (Flanagan 2011 p. 323):

- The web browser creates a document object and starts parsing the HTML. The web browser adds HTML elements and text nodes into the document object. The document objects *readyState* property has the value *loading*.
- When the web browser encounters *script* elements it adds those elements to the document and executes the JavaScript found within the opening and closing *script* tags or loads the external script file and then executes the content of it. At this point JavaScript code within the *script* tag can access its own *script* element and all HTML elements found before it in the document object.
- If a *script* tag has an *async* attribute set the browser starts to download the script file but it does not wait for the download to be ready before it continues to parse the HTML document. Instead the script will be executed as soon as it is possible after it has been downloaded.
- When the whole HTML document is parsed the document objects *readyState* property will get the value *interactive* assigned to it.
- If any *script* tag had a *defer* attribute set, the JavaScript defined gets executed.
- The web browser fires a *DOMContentLoaded* event signaling that the HTML document has been parsed. At this point the JavaScript interpreter transitions from phase one, the synchronous mode, to phase two, the asynchronous mode. There might still be some *async* scripts that have not been executed.
- At this point the HTML document has been completely parsed but there might still be some resources that have not been loaded e.g. images. When all external resources and *async* scripts have been loaded the documents *readyState* property is changed to *complete* and the load event is fired.
- The browser is now ready with parsing and is waiting to fire event handlers based on the user interactions.

3 THE NEED FOR MODULES

3.1 What are Modules?

By organizing code into classes code becomes more modular and reusable across applications. Modules are more than just classes since a module can be a set of classes, a utility function or code to be run on invocation. Any piece of JavaScript can be a module if it is written in modular fashion. Usually a module is represented by a single source file with the file name reflecting the modules name.

The purpose of modules is to allow the assembly of larger applications from various sources and to guarantee a successful execution of the code without unwanted side-effects i.e. the code should run fine in the presence of third party code. To avoid side-effects the modules should avoid polluting the global object, allowing other modules and bits of code to run in a clean, non-altered, environment. (Flanagan 2011 p. 246)

There are few JavaScript libraries that provide a module system, of which Dojo Toolkit, version 1.6 and 1.8, and Google's Closure library are presented in more detail and evaluated in chapters 4 and 5. Both the Dojo Toolkit 1.6 and the Closure library define *provide* and *require* functions for declaring and requiring modules. The AMD specification, utilized in the Dojo Toolkit 1.8, defines *define* and *require* methods for module handling.

3.2 The Global Object

The global object feature of JavaScript has been a target of criticism (Crockford 2006). In web browsers the global object is referenced as the *window*. All globally available functions, objects and properties are accessed via this object. It is important for a module not to pollute the global object as JavaScript is a weakly-typed language and almost every variable can be overwritten.

One way to avoid the pollution of the global object is to use namespaces. Although namespaces were once planned to be included natively into JavaScript, the idea has then

been dropped (Eich 2008). But namespaces can be simulated via objects. In modular programming this is done by introducing only one new global object per top-level module and nesting all sub-modules under this object. This ensures a minimal footprint on the global object but with deeply nested modules this approach results into long namespaces. This method is used by both the Dojo toolkit 1.6 and Google's Closure library (See chapter 4 and 5 for more details). (Flanagan 2011 p. 246)

3.3 Loading JavaScript

As a module should be represented by a JavaScript source file the addition of modules into the application can be achieved by loading the JavaScript source files into the HTML document via the *script* tag (chapter 2.2). Without a client-side module loader the *script* tags need to be added manually to the document. The order of the *script* tags is important as any dependencies a module might require needs to be loaded prior to the module.

In addition to *script* tags, adding JavaScript into the HTML document can be done by the function *eval* which evaluates a string of JavaScript code. An XHR request can be used to load JavaScript code from the server which is then passed to the *eval* function. This allows dynamic loading of modules, but the usage of *eval* is discouraged as the evaluated code is executed with the privileges of the caller and *eval* is slower than the alternatives, as the JavaScript interpreter cannot optimize the evaluated code (eval 2012).

4 MODULE LOADERS

As mentioned earlier, JavaScript does not yet provide a native module system. However, there are third party module loaders such as Dojo toolkit 1.6, Google closure library, and AMD that can be used with JavaScript. This section provides an overview and a comparison of these module loaders.

4.1 Dojo Toolkit Version 1.6

The Dojo Toolkit is an open source, modular JavaScript library provided by the non-profit organization Dojo Foundation (Dojo Foundation 2012). It provides a package system for declaring and loading modules and module dependencies. Declaration of modules in Dojo Toolkit 1.6 is done with a function called *dojo.provide* and importing modules is done via the *dojo.require* function (Lennon 2011).

The *dojo.provide* function tells the module loader that a specific module has been loaded and each module source file must contain at least one *dojo.provide* call at the top of the file. For example the module *prime* in the *myLib.Math.prime.js* source file must call *dojo.provide("myLib.Math.prime")* before declaring the modules code. (dojo.provide 2012)

The *dojo.require* function loads a module by name. If the module has already been loaded then nothing is done, else the name is translated into a file path and the file is then loaded from the server. For example *dojo.require("myLib.Math.prime")* loads the file *myLib.Math.prime.js*.

The *dojo.require* uses the *XHR + eval* approach (see chapter 3.3) to load modules and module dependencies dynamically. This leads into debugging difficulties and some slight overhead in module loading as the response from the XHR request needs to be evaluated to JavaScript.

Modules created with the Dojo Toolkit 1.6 can only be loaded by the Dojo Toolkit 1.x module loader. The module loader is synchronous and code execution is halted until the

required modules are loaded and parsed. Developing applications with the Dojo Toolkit 1.6 can be cumbersome due to difficulties with debugging and the synchronous nature of the loader.

4.2 Closure Library

The Closure Library is the JavaScript library used internally by Google but is provided for public use under the Apache License 2.0. The Closure Library's package and module loading system resembles the Dojo Toolkit's module loader. Both provide a *provide* and a *require* method for defining and loading modules. The difference between Dojo Toolkit 1.6 and Closure library is how modules are loaded from the server. Dojo uses synchronous XHR requests to load data from the server, where Closure library appends *script* tags into the HTML document.

Loading modules with Closure library is done in a similar way as with Dojo Toolkit 1.6, with the *goog.require* method. This method takes a module name as an argument, but it does not automatically map the required module names into source file paths, unlike the Dojo Toolkit 1.6 module loader. Closure module loader needs to be told which source files contain which modules. This module and dependency resolution mapping can be achieved with the dependency calculator python script that comes bundled with the Closure library. The script requires *python* and is meant to be run on UNIX systems (Using ClosureBuilder 2012).

The *goog.require* function call should not be used in the same *script* tag as the code that uses the required modules. This is because *goog.require* adds code to the document after the script tag containing the call. (Getting Started with the Closure Library 2012)

Developing application with the Closure library requires *python* because of the dependency script. As with the Dojo Toolkit 1.6, modules developed with the Closure library cannot be used with other module loaders.

4.3 The Asynchronous Module Definition

The Asynchronous Module Definition API specifies a mechanism for defining modules in a way that modules and their dependencies are loaded asynchronously. This technique suits well for the web browser environment as synchronous loading halts the JavaScript interpreter until the file is loaded and parsed.

AMD started as a transport format on the CommonJS wiki (Modules/Asynchronous Definition 2012), but grew from a script loader to a module definition API. When no consensus was reached on the CommonJS wiki on AMDs role the AMD proposal was separated from CommonJS wiki into its own wiki. The CommonJS wiki focuses more on server side, synchronous environment, implementations of JavaScript and AMD is purely meant for asynchronous environments, i.e. the web browser. (Burke 2012)

One of the goals of the AMD API was to prevent the pollution of the global object. The AMD API adds only two functions to the global scope, *require* and *define*, where Dojo Toolkit 1.6 and Closure library both export modules into the global object (see chapter 3.2).

The AMD specification defines a method to load non-AMD resources with the help of loader plugins. This is a valuable benefit when non-AMD dependencies are needed to be loaded, e.g. HTML fragments, template files or CSS.

4.3.1 The define Method

The *define* method is used to create modules. It takes three arguments: an optional module ID, an optional dependency list and a non-optional factory function or object literal.

4.3.1.1 The module ID

The ID argument is a string literal and it is optional. The ID looks like a file system path and when the module is required, by the *require* or the *define* method, the ID is normalized to an absolute URL.

The module ID format must comply with the following:

- A module identifier is a string of “terms” delimited by forward slashes.
- A term must be a camelCase identifier, “.”, or “..”.
- Module identifiers may not have file-name extension like “.js”.
- Module identifiers may be "relative" or "top-level". A module identifier is "relative" if the first term is "." or "..".
- Top-level identifiers are resolved off the conceptual module name space root.
- Relative identifiers are resolved relative to the identifier of the module in which *require* is written and called.

For example the module ID *myLib/Math/prime* translates into a file path of *myLib/Math/prime.js*.

When defining a module the ID is not recommended to be defined and is seen as a deprecated feature of the AMD API. If the module ID is left out, as recommended, then the defined module is seen as an anonymous module meaning that the AMD module loader knows which module to load based on the identifier in the dependency list.

Only build tools should add the module ID into the module definition. This way many modules can be included into one file to reduce the amount of HTTP requests to the server.

4.3.1.2 The Dependency List

The second, optional, argument is the dependency list. It is a list of module IDs that are dependencies required by the module that is being defined. The dependencies are resolved before the defined modules factory function is called and the return values of the dependencies are passed as arguments to the factory function in the order the dependencies are introduced in the list.

4.3.1.3 The Module Factory

The third argument, factory, is the meat of the defined module. It can be a function or an object literal. If it is an object literal, then that object is the return value of the module. If it is a function then it is executed only once after all dependencies, if any, are resolved. The return value of the factory is the exported value of the module.

There are three reserved keywords that have special meaning and resolution in the dependency list:

- *require* – A context-sensitive *require* method. Any requested modules ids are resolved relative to the module that made the request.
- *exports* – Instead of explicitly returning a value in the factory function, properties can be attached to the *exports* object. Attaching properties to the *exports* object is the only way to define modules that are in a circular dependency.
- *module* – The *module* object has one property, *exports*. This *module.exports* is the same thing as the *exports* object described above.

4.3.2 The require Method

The *require* function is used to configure the loader and to load AMD modules. It takes three arguments, an optional configuration object, an optional dependency list and an optional callback function which is run after all dependencies are resolved.

4.3.2.1 The Configurations Object

The configuration object is used to configure the AMD loader. As of the time of writing the configuration object of AMD loaders is still in draft. An AMD loader is not required to implement all configuration values defined in the AMD wiki, but if an AMD loader is to provide a capability that matches these configuration values, it should use these names, structures and behaviors:

- *baseUrl* – The *baseUrl* is a string that indicates the root used for ID-to-path resolutions. Relative paths are relative to the current working directory. In web browsers this is the directory containing the web page running the script.

- *paths* – The *paths* is an object, a hash-map, where every property is an absolute module ID prefix, and the value is either a string or an array. If the value is a string then it is the path to the module; relative to the *baseUrl* or an absolute path. If the value is an array then it should act as a failover list which means that if the loader is unable to load the module from the first path in the list then it should try to load it from the second path and so on.
- *packages* – The *packages* configuration option is an array of package configuration objects. A package configuration object consisting of the following properties:
    - *name* – A module ID prefix, the first segment of an absolute module ID.
    - *location* – Path of the module package relative to the *baseUrl*.
    - *main* – The main module of the package. Default value is *main*.
- *map* – The *map* object is used to rewrite request from modules with specific module ID prefix to point to another module.

4.3.3 AMD Loader Plugins

Loader plugins extend the AMD loader to load non JavaScript files, e.g. JavaScript Object Notation (JSON) files, text files etc. Loader plugins are AMD defined modules with a specific API. The plugins functionality is invoked by adding its module identifier before an exclamation mark in the dependency. The part after the exclamation mark is the plugins resource identifier. (Burke 2012)

An example of using the *text* loader plugin:

```
require([text!myModule/readme.txt], function(contentOfReadme) {
        // Do something with the content of the readme.txt
});
```

4.4 Summary

Loading modules with the Dojo Toolkit 1.6 and Google's Closure library is verbose. Both loaders use the *objects as namespace* technique to minimize the global object footprint. Both libraries provide mechanisms for defining and loading loosely coupled

modules, but they don't mix and match. Modular application components made with the Closure Library cannot be used by the Dojo loader and vice versa.

AMD is an improvement compared to declaring modules into the global object because (Why AMD? 2012):

- Clear declaration of dependencies and avoids the use of globals.
- IDs can be mapped to different paths allowing swapping out implementation. This is useful for mocks and unit tests.
- Encapsulation of the module definition. Avoiding the pollution of the global namespace.

An AMD loader is an improvement compared to Dojo Toolkit 1.6 and Closure library module loaders because:

- Dependency injection is done with the HTML *script* tags.
- Easier debugging compared to evaluated scripts.
- No need for server specific languages and tools.
- Module loader plugins ease loading of resources.

Below is a table that shows the module loaders feature differences.

Table 1. Differences between module loaders

Module loader differences	Dojo Toolkit 1.6	Google's Closure library	AMD
Asynchronous module loading	No	Yes	Yes
Modules can be used by other module loaders	No	No	Yes
Easy to debug	No	Yes	Yes
Server-side tool dependant	No	Yes	No
Loads modules with *eval* and *XHR*	Yes	No	No
Loads modules by injecting *script* tags into the HTML document	No	Yes	Yes

Support of loading other resources (text, json, css, etc.)	No	No	Yes
Avoid declaration of modules in the global namespace	No	No	Yes

5 A PRACTICAL COMPARISON OF MODULE LOADERS

A sample program was developed to compare the differences between Dojo Toolkit 1.6, Google's Closure Library and an AMD module loader (Dojo Toolkit 1.8). The sample application shows the basics of defining modules, loading modules and how dependencies are resolved. The application itself is a primary number calculator which lets the user choose the range from which prime numbers are to be searched.

5.1 Building the Sample program with Different Module Loaders

The code structure of the sample application is independent of which module loader is used. The application consists of one HTML document, a module loader, the core application module and one helper library module. Initially, the module loader is loaded and configured in the HTML document and then the application module is loaded and started. Only the main application module is needed to be manually required as the module loaders take care of loading all of the dependencies the application requires.

5.1.1 Initializing the Application Module

In the Dojo Toolkit 1.6 version of the application the core module of Dojo Toolkit is loaded. This core module includes the module loader among other often used methods, such as HTML element selector methods etc. After the core module and the module loader is loaded to the HTML document the application's main module is loaded using the *dojo.require* method.

Loading the core Dojo Toolkit module:

```
<script type="text/javascript" src="js/lib/dojo/dojo.js"></script>
```

Loading the application module with the help of the Dojo Toolkit module loader is done next:

```
<script type="text/javascript">
    dojo.require('myApp.App');
```

```
    // Rest of the application initialization is done here
</script>
```

After the *dojo.require('myApp.App')* function call the application module is loaded and accessible through the global object *myApp.App*. The application is instantiated by calling the *new* operator on the constructor and binding the application into a variable:
var app = new myApp.App();

The Closure library module loader behaves in similar way. The Closure library core module, including the module loader, is loaded in the HTML document:
<script type="text/javascript" src="lib/closure/goog/base.js"></script>

The Closure library's module loader needs a dependency map to tell it from where to load which module. The dependency map is loaded next:
<script type="text/javascript" src="deps.js"></script>

A call to the *goog.require* adds code to the HTML document after the *script* tag containing the call. Because of this all module required directly in the HTML document needs to be loaded in a separate *script* tag:

```
<script type="text/javascript">
    goog.require('myApp.App');
    // All module used directly in the HTML document are loaded here
</script>
```

As with the *dojo.require* calling *goog.require('myApp.App')* results in a global object *myApp.App* which is usable in similar way as in the Dojo Toolkit version of the sample application.

The AMD is a bit different. Because of the factory function of the *require* method the global object is not polluted as all variables declared inside this factory function are not visible outside of the function.

Loading and initializing the application module with the AMD loader:

```
require(['App'], function(App) {
    var app = new App();
    // Rest of the application initialization is done here
});
```

5.1.2 Application Dependencies

The sample applications main module has only one dependency, the library module *primary*. This module returns a method that calculates a list of all primary numbers between the first primary number, number two (2), and the input value.

The *primary* module dependency is loaded by the application module. With Dojo Toolkit 1.6 this is achieved with the following:

```
dojo.provide('myApp.App');
dojo.require('myLib.Math.primary');
dojo.declare('myApp.App', null, {
    primes: function(n) {
        return myLib.Math.primary(n);
    }
});
```

First the application module is provided with the *dojo.provide* method. Before the module is defined any dependencies are loaded. Here there is only one dependency, the *primary* module. After all dependencies are loaded the module can be defined. The *dojo.declare* method is a helper method to create classes in JavaScript. Here it is used to declare the *myApp.App* class.

The Closure library needs to be told from where to load modules and dependencies. The dependency map that needs to be loaded manually into the HTML document looks like this:

```
goog.addDependency('../../myApp/App.js', ['myApp.App'], ['myLib.Math.primary']);
goog.addDependency('../../myLib/Math/primary.js', ['myLib.Math.primary'], []);
```

The *goog.addDependency* method tells the module loader the file path of a source file, which modules reside in that file and what the dependencies are for the modules in that file.

The application module made with the help of Closure library resembles the Dojo Toolkit 1.6 application module:

```
goog.provide('myApp.App');
goog.require('myLib.Math.primary');
myApp.App = function() {};
myApp.App.prototype.primes = function(n) {
        return myLib.Math.primary(n);
};
```

Definition of the application module with the AMD syntax:

```
define(['dojo/_base/declare', 'myLib/Math/primary'],
function(declare, primary) {
        return declare([], {
                primes: function(n) {
                        return primary(n);
                }
        });
});
```

Compared to the Dojo Toolkit 1.6 the AMD version of the application module explicitly requests the *declare* method. In Dojo Toolkit 1.6 this method is loaded in conjunction with the module loader.

5.2 Loading the Modules of the Sample Application

The application is loaded both from the file-system, to check if the module loaders can be used without an HTTP server, and from a lightweight HTTP server, *Mongoose* (Mongoose 2012).

Loading the Dojo Toolkit 1.6 version of the application from the file-system proved to be impossible because of the XHR security restrictions of the browsers. The Dojo Toolkit 1.6 method of loading modules with XHR cannot be used to load data from the local file-system. Development and testing of modular applications without an HTTP server is impossible with Dojo Toolkit 1.6. Both the Closure library and the AMD module loaders are capable of loading modules from the local file-system as their loading techniques do not involve any XHR requests. Some of the AMD loader plugins might utilize XHR to load resources, but the JavaScript modules are loaded through the HTML *script* tags. All three libraries operate without problems when used over HTTP.

The Closure library requires a separate dependency script to tell it from where to load custom made modules. This dependency script can be built by hand but becomes tedious to manage when there are several modules and module dependencies used in the application. Development of applications with the Closure library works best in conjunction with an HTTP server.

5.3 Performance Tests of Module Loaders

Module loaders performance was tested by measuring the time it takes for the module loaders to load the application, how big the application and the module loaders are in terms of kilobytes (KB) and how many HTTP requests each module loader perform.

The tests were run with the Google Chrome web browser (Version 23.0.1271.64 m) on a 64-bit Windows 7 PC with Intel Core i7-2630QM 2.00GHz CPU and 8 GB of RAM. The application consists of only two custom modules. The number of custom modules was kept to a minimum to compare the differences between the JavaScript libraries and their module loaders.

5.3.1 Timeline comparisons

The Google Chromes *Timeline* pane is a good tool for measuring loading times of modules. It shows the exact point of time when a request was made and how long it took. It also shows when the *onload* event was triggered. The time spent for loading resources is

shown in blue in Figure 1 where the timeline of the Dojo Toolkit 1.6 module loader is shown. The timeline shows that the XHR requests are synchronous and modules are loaded sequentially.

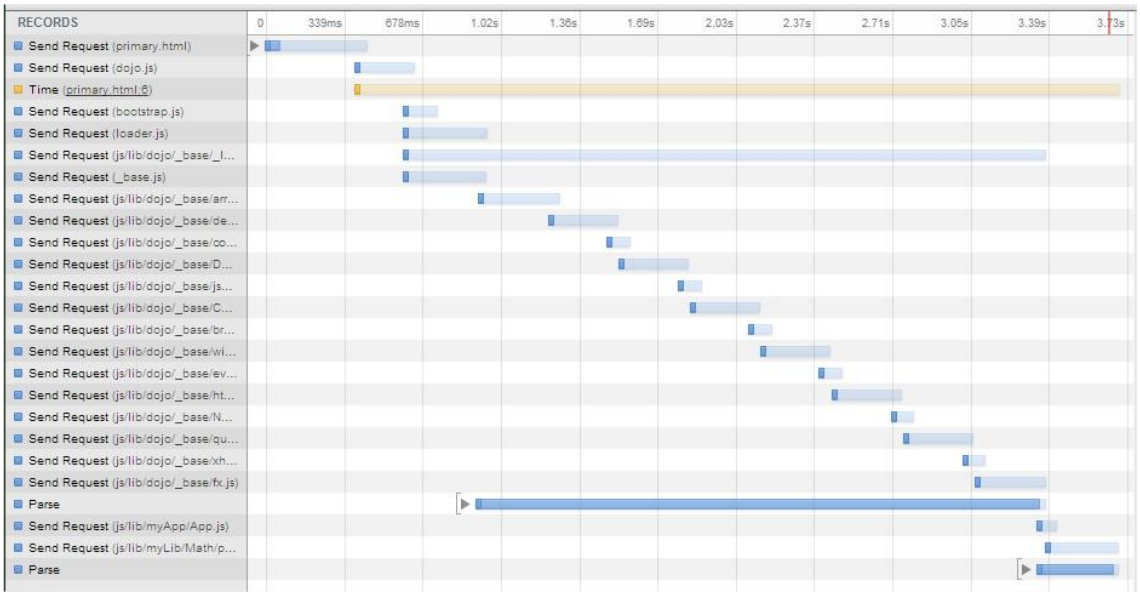

Figure 1. Dojo Toolkit 1.6 module loading timeline

The Closure library loads modules asynchronously and the timeline shows that the requests are parallel (Figure 2).

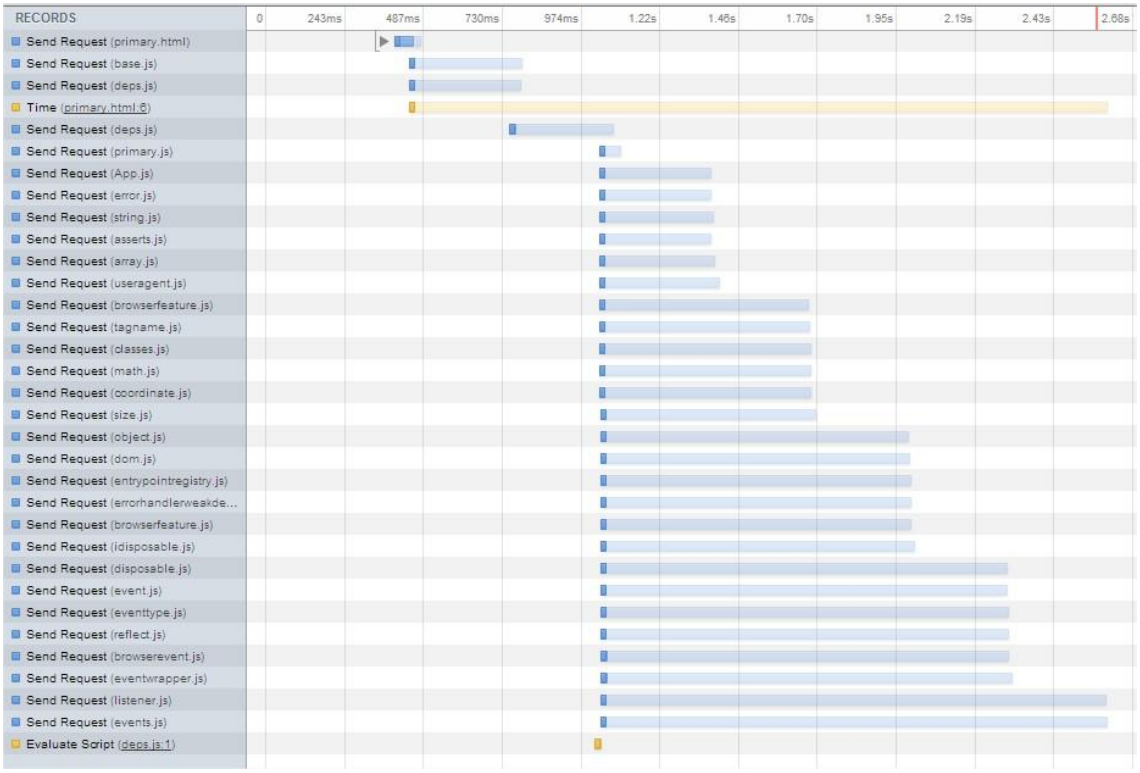

Figure 2. Closure library module loading timeline

Figure 3 show that the Dojo Toolkit 1.8 AMD compliant module loader loads the modules in parallel fashion.

Figure 3. AMD module loading timeline

These graphs show that both Closure library and AMD module loaders load modules parallel and the Dojo Toolkit 1.6 module loader loads modules in sequence. Parallel

loading of modules is faster compared to sequential loading, as several modules can be loaded from the server at the same time.

5.3.2 Module Loading Comparison

Table 2 shows the differences between the module loaders when it comes to loading the test application. The test was performed by loading the application ten times with each of the module loaders. The browsers cache was emptied between every reload to ensure comparable results. The number of request and the size of the application remains constant (for each module loader) between each application load, but the time it takes for the bowser to trigger the *onload* and the *DOMContentLoaded* events varies from application load to application load.

Table 2. Module loader performance over HTTP

Loader perfromance	Dojo Toolkit 1.6	Closure library	AMD (Dojo Toolkit 1.8)
No. of requests	23	31	15
Transferred KB	397,90	592,17	182,8
onload event (s)	3,17	1,89	1,27
DOMContentLoaded event (s)	3,17	1,89	0,35

The AMD module loader outperformed both the Closure library and Dojo Toolkit 1.6 module loaders in terms of number of requests needed to load the application components, size of the application and the time it takes for the browser to parse the page content. This shows that the AMD approach results in faster load times because only the used modules are loaded. Both Dojo Toolkit 1.6 and Closure library load modules that are not necessarily used by the application.

6 CONCLUSION

Modules present a separation of concerns and improve the maintainability of the code base. Even though JavaScript has grown from a simple scripting language into a full blown programming language, it lacks a native module definition and loading system. Many JavaScript libraries and frameworks include some kind of module system, but all of these implementations have their own syntax and are not compatible with other implementations. The Asynchronous Module Definition (AMD) is *de facto* standard for module loaders. Modules that follow this standard can be loaded with any AMD compliant module loader.

In this thesis three different module loaders are evaluated and a simple sample application is built against each module loader specification. The module loaders tested are the Dojo Toolkit 1.6's, Google's Closure library's and the AMD compliant Dojo Toolkit 1.8's module loaders. Module loader performance was tested by comparing the application size and the loading time of the modules. Loading modules with the Closure library and the AMD module loader were faster than with the Dojo Toolkit 1.6's module loader. This is because the AMD and the Closure library load modules in parallel fashion where the Dojo Toolkit 1.6 loads modules sequentially. Usage of the AMD module loader resulted in fewer modules to be loaded as it loads only the required modules, where both Closure library and Dojo Toolkit 1.6 load additional resources. This correlates directly into the total download size of the application.

Modules built with the *Dojo Toolkit 1.6* and the *Closure library* are not usable with other module loaders as both frameworks implement their own, proprietary, module definition syntax and loading mechanism. AMD specification compliant modules can be used with any other AMD specification compliant module loader.

The AMD module loader is overall a better module loader than the Dojo Toolkit 1.6 and the Closure library module loaders because:

- Modules can be loaded by other AMD compliant module loaders.
- Only modules that are used are loaded.

- None-AMD resources can be loaded with the help of loader plugins.
- It does not pollute the global object.

REFERENCES

Alman, B. 2010, Immediately-Invoked Function Expression (IIFE), published 15.11.2010. Available: http://benalman.com/news/2010/11/immediately-invoked-function-expression/ Accessed 13.10.2012

Burke, J. 2012, AMD, published 22.5.2011. Available: https://github.com/amdjs/amdjs-api/wiki/AMD Accessed 13.10.2012

Champeon, S. 2001, O'Reilly Web DevCenter, published 6.4.2001. Available: http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html Accessed 13.10.2012

Crockford, D. 2006, Global Domination, published 1.6.2006. Available: http://yuiblog.com/blog/2006/06/01/global-domination Accessed 13.10.2012

Croll, A. 2010, Understanding JavaScript's 'undefined', published 16.8.2010. Available: http://javascriptweblog.wordpress.com/2010/08/16/understanding-undefined-and-preventing-referenceerrors/ Accessed 13.10.2012

Dojo Foundation. 2012, Dojo Toolkit. Available: http://dojotoolkit.org/ Accessed 14.10.2012

dojo.provide. 2012, Dojo Toolit Documentation. Available: http://dojotoolkit.org/reference-guide/1.6/dojo/provide.html Accessed 14.10.2012

Eich, B. 2008, ECMAScript Harmony, published 13.8.2008. Available: https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html Accessed 13.10.2012

eval. 2012, Mozilla Developer Network, published 31.3.2005. Available: https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/eval#Don.27t_use_eval.21 Accessed 14.10.2012

Flanagan, D. 2011, *JavaScript: The Definitive Guide, 6th Edition*. O'Reilly Media, 1100 p.

Functions and function scope. 2012, Mozilla Developer Network, published 9.9.2005. Available: https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope Accessed 30.9.2012

Getting Started with the Closure Library. 2012, Google Developers. Available: https://developers.google.com/closure/library/docs/gettingstarted#step4 Accessed 16.10.2012

harmony:modules. 2012, ECMAScript wiki, published 23.3.2011. Available: http://wiki.ecmascript.org/doku.php?id=harmony:modules Accessed 29.10.2012

Lennon, J. 2011, Dojo from the ground up, Part 2: Mastering object-oriented development with Dojo, published 1.1.2011. Available: http://www.ibm.com/developerworks/web/library/wa-ground2/index.html Accessed 27.10.2012

Modules/Asynchronous Definition. 2012, CommonJS, published 9.9.2012. Available: http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition Accessed 29.10.2012

Mongoose. 2012, Mongoose – easy to use web server. Available: http://code.google.com/p/mongoose/ Accessed 29.10.2012

Nygren, E. Sitaraman, R. K. and Sun, J. 2010, *The Akamai Network: A Platform for High-Performance Applications*, published 7.2010. Available: http://www.akamai.com/dl/technical_publications/network_overview_os___r.pdf Accessed 1.11.2012

Object.freeze. 2012, Mozilla Developer Network, published 4.10.2010. Available: https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/freeze Accessed: 13.10.2012

Osmani, A. 2011, Essential JavaScript Namespacing Patterns, published 23.9.2011. Available: http://addyosmani.com/blog/essential-js-namespacing/ Accessed 13.10.2012

Pieters, S. 2012, HTML5 differences from HTML4, published 22.1.2008. Available: http://dev.w3.org/html5/html4-differences/#new-apis Accessed 1.11.2012

Using ClosureBuilder. 2012, Google Developers. Available: https://developers.google.com/closure/library/docs/closurebuilder Accessed 16.10.2012

Venners, B. 2003, Orthogonality and the DRY Principle, published 10.3.2003. Available: http://www.artima.com/intv/dry.html Accessed: 7.10.2012

Why AMD?. 2012, Require JS a JavaScript module loader. Available: http://requirejs.org/docs/whyamd.html Accessed: 7.10.2012

XMLHttpRequest. 2012, Mozilla Developer Network, published 4.8.2005. Available: https://developer.mozilla.org/en-US/docs/DOM/XMLHttpRequest Accessed: 1.11.2012

APPENDIX A: SAMPLE APPLICATION MADE WITH DOJO TOOLKIT 1.6

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Modular JavaScript Dojo 1.6</title>
  <script type="text/javascript">
    console.time('DepsLoaded');
    var moduleStartTime = new Date().getTime(),
      moduleEndTime;
  </script>
  <script type="text/javascript" src="js/lib/dojo/dojo.js"></script>

  <script type="text/javascript">
    dojo.require('myApp.App');
    dojo.ready(function() {
      console.timeEnd('DepsLoaded');
      moduleEndTime = new Date().getTime();

      var app = new myApp.App(),
        loaded = dojo.byId('modulesLoaded'),
        input = dojo.byId('number'),
        result = dojo.byId('result'),
        button = dojo.byId('calculate'),
        report = dojo.byId('report'),
        value = 100,
        primes, startTime, endTime, totalTime;

      loaded.innerHTML = '<strong>Module load time: </strong>' + (moduleEndTime -
moduleStartTime) + 'ms';

      dojo.connect(button, 'onclick', function() {
        value = input.value;
        startTime = new Date().getTime();
        primes = app.primes(value);
        endTime = new Date().getTime();
        totalTime = endTime - startTime;
        result.value = primes.join(", ");

        report.innerHTML = '<strong>Total: </strong>' + totalTime + 'ms';
      });
    });
  </script>

  <style type="text/css">
    body, html {
      height: 100%;
      width: 100%;
      margin: 0;
    }
    h1 {
      margin-bottom: 20px;
    }
    .center {
      margin-left: auto;
      margin-right: auto;
      text-align: center;
    }
    #result {
      width: 600px;
      height: 300px;
    }
  </style>
</head>
<body>
```

```
  <div class="center">
    <h1>Dojo 1.6</h1>
    <h2>Primary number finder</h2>
    <div id="modulesLoaded"></div>
    <textarea id="result" readonly></textarea>
    <div>
      <input id="number" type="number" value="100" />
      <button id="calculate" type="button">Calculate</button>
    </div>
    <div id="report">

    </div>
  </div>
</body>
</html>
```

App.js

```
dojo.provide('myApp.App');
dojo.require('myLib.Math.primary');
dojo.declare('myApp.App', null, {
  primes: function(n) {
    return myLib.Math.primary(n);
  }
});
```

primary.js

```
dojo.provide('myLib.Math.primary');
myLib.Math.primary = function(n) {
  n = parseInt(n);

  var a = !!window.Int8Array ?
      new Uint8Array(n+1) :         // If typed arrays are supported use those else
      new Array(n+1),               // use normal arrays which are much slower
      max = Math.floor(Math.sqrt(n)), // Don't do factors higher than this
      p = 2,                          // The first prime is 2
      i, l,
      primes = [];                  // Store the found primes

                                    // Mark all non-prime indexes
  while (p <= max) {                // Calculate primes less than max
    for (i = 2 * p; i <= n; i += p) {  // Mark multiples of p as composite
      a[i] = 1;
    }
    while(a[++p]);                  // Next unmarked index is a prime
  }
  for (i = 2, l = a.length; i < l; i++) {
    if (!a[i]) {
      primes.push(i);
    }
  }

  return primes;
};
```

APPENDIX B: SAMPLE APPLICATION MADE WITH CLOSURE LIBRARY

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Google Closure</title>
  <script type="text/javascript">
    console.time('DepsLoaded');
    var moduleStartTime = new Date().getTime(),
      moduleEndTime;
  </script>
  <script type="text/javascript"  src="lib/closure/goog/base.js"></script>
  <script type="text/javascript" src="deps.js"></script>
  <script type="text/javascript">
    goog.require('myApp.App');
    goog.require('goog.dom');
    goog.require('goog.events');
  </script>

  <style type="text/css">
    body, html {
      height: 100%;
      width: 100%;
      margin: 0;
    }
    h1 {
      margin-bottom: 20px;
    }
    .center {
      margin-left: auto;
      margin-right: auto;
      text-align: center;
    }
    #result {
      width: 600px;
      height: 300px;
    }
  </style>
</head>
<body>
  <div class="center">
    <h1>Closure library</h1>
    <h2>Primary number finder</h2>
    <div id="modulesLoaded"></div>
    <textarea id="result" readonly></textarea>
    <div>
      <input id="number" type="number" value="100" />
      <button id="calculate" type="button">Calculate</button>
    </div>
    <div id="report">

    </div>
  </div>

  <script type="text/javascript">
    // Google Closure library does not implement a document ready method.
    console.timeEnd('DepsLoaded');
    moduleEndTime = new Date().getTime();
    (function() {
      var app = new myApp.App(),
        loaded = goog.dom.getElement('modulesLoaded'),
        input = goog.dom.getElement('number'),
        result = goog.dom.getElement('result'),
        button = goog.dom.getElement('calculate'),
        report = goog.dom.getElement('report'),
        value = 100,
        primes, startTime, endTime, totalTime;
```

```
      loaded.innerHTML = '<strong>Module load time: </strong>' + (moduleEndTime -
moduleStartTime) + 'ms';

      goog.events.listen(button, goog.events.EventType.CLICK, function() {
        value = input.value;
        startTime = new Date().getTime();
        primes = app.primes(value);
        endTime = new Date().getTime();
        totalTime = endTime - startTime;
        result.value = primes.join(", ");

        report.innerHTML = '<strong>Total: </strong>' + totalTime + 'ms';
      });
    }());
  </script>
</body>
</html>
```

deps.js

```
goog.addDependency('../../myApp/App.js', ['myApp.App'], ['myLib.Math.primary']);
goog.addDependency('../../myLib/Math/primary.js', ['myLib.Math.primary'], []);
```

App.js

```
goog.provide('myApp.App');
goog.require('myLib.Math.primary');
myApp.App = function() {};
myApp.App.prototype.primes = function(n) {
  return myLib.Math.primary(n);
};
```

primary.js

```
goog.provide('myLib.Math.primary');
myLib.Math.primary = function(n) {
  n = parseInt(n);

  var a = !!window.Int8Array ?
        new Uint8Array(n+1) :    // If typed arrays are supported use those else
        new Array(n+1),        // use normal arrays which are much slower
    max = Math.floor(Math.sqrt(n)),    // Don't do factors higher than this
    p = 2,                // The first prime is 2
    i, l,
    primes = [];            // Store the found primes

                      // Mark all non-prime indexes
  while (p <= max) {            // Calculate primes less than max
    for (i = 2 * p; i <= n; i += p) {  // Mark multiples of p as composite
      a[i] = 1;
    }
    while(a[++p]);            // Next unmarked index is a prime
  }
  for (i = 2, l = a.length; i < l; i++) {
    if (!a[i]) {
      primes.push(i);
    }
  }

  return primes;
};
```

APPENDIX C: SAMPLE APPLICATION MADE WITH AMD SPECIFICATION

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Modular JavaScript Dojo 1.8</title>
  <script type="text/javascript">
    console.time('DepsLoaded');
    var moduleStartTime = new Date().getTime(),
      moduleEndTime;
  </script>
  <script type="text/javascript" data-dojo-config="async: 1, tlmSiblingOfDojo: 0,
isDebug: 0" src="js/lib/dtk/dojo/dojo.js"></script>
  <script type="text/javascript" src="js/bootstrap.js"></script>

  <script type="text/javascript">
    require(['myLib/has'], function() {
    require(['App', 'dojo/dom', 'dojo/on', 'dojo/domReady!'], function(App, dom, on) {
      console.timeEnd('DepsLoaded');
      moduleEndTime = new Date().getTime();

      var myApp = new App(),
        loaded = dom.byId('modulesLoaded'),
        input = dom.byId('number'),
        result = dom.byId('result'),
        button = dom.byId('calculate'),
        report = dom.byId('report'),
        value = 100,
        primes, startTime, endTime, totalTime;

      loaded.innerHTML = '<strong>Module load time: </strong>' + (moduleEndTime -
moduleStartTime) + 'ms';

      on(button, 'click', function() {
        value = input.value;
        startTime = new Date().getTime();
        primes = myApp.primes(value);
        endTime = new Date().getTime();
        totalTime = endTime - startTime;
        result.value = primes.join(", ");

        report.innerHTML = '<strong>Total: </strong>' + totalTime + 'ms';
      });
    });
    });
  </script>

  <style type="text/css">
    body, html {
      height: 100%;
      width: 100%;
      margin: 0;
    }
    h1 {
      margin-bottom: 20px;
    }
    .center {
      margin-left: auto;
      margin-right: auto;
      text-align: center;
    }
    #result {
      width: 600px;
      height: 300px;
    }
  </style>
</head>
<body>
```

```
  <div class="center">
    <h1>AMD</h1>
    <h2>Primary number finder</h2>
    <div id="modulesLoaded"></div>
    <textarea id="result" readonly></textarea>
    <div>
      <input id="number" type="number" value="100" />
      <button id="calculate" type="button">Calculate</button>
    </div>
    <div id="report">

    </div>
  </div>
</body>
</html>
```

bootstrap.js

```
require({
  noGlobals: true, // Dojo loader specific configuration.
                   // Implicitly not using the legacy namespaces.
  baseUrl: 'js/', // Defines the base URL from where the laoder loads local packages
  packages: [ // Package definitions related to the baseUrl
    { name: 'dojo', location: 'lib/dtk/dojo' }, // The top module identifier dojo and
its location
    { name: 'myLib', location: 'lib/myLib' }, // Custom or third party library
    { name: 'App', location: 'myApp', main: 'App' } // Pointer to the application
  ],
  aliases: [            // The aliases property is a Dojo loader specific implementation
that allows the
    ['text', 'dojo/text'],    // loader to alias a package or dependency to another
name.
    ['has', 'dojo/has']
  ]
});
```

has.js

```
define(['has', './has/typedArrays'], function(has) {
  return has;
});
```

typedArrays.js

```
define(['has'], function(has) {
  has.add('typed-arrays', function(global) {
    return !!global.Int8Array;
  });
  return has;
});
```

App.js

```
define([
  'dojo/_base/declare',
  'has!typed-arrays?myLib/Math/primaryOptimized:myLib/Math/primary'
  // Use has module loader plugin to load an optimized version of the primary number
  // module if typed arrays are supported by the browser.
], function(
  declare,
  primary
) {
  return declare([], {
    primes: function(n) {
      return primary(n);
    }
  });
});
```

primary.js

```
define(function() {
```

```
  return function(n) {
    n = parseInt(n);

    var a = new Array(n+1),
      max = Math.floor(Math.sqrt(n)), // Don't do factors higher than this
      p = 2,                          // The first prime is 2
      i, l,
      primes = [];                    // Store the found primes

                                      // Mark all non-prime indexes
    while (p <= max) {                // Calculate primes less than max
      for (i = 2 * p; i <= n; i += p) {  // Mark multiples of p as composite
        a[i] = 1;
      }
      while(a[++p]);                  // Next unmarked index is a prime
    }
    for (i = 2, l = a.length; i < l; i++) {
      if (!a[i]) {
        primes.push(i);
      }
    }

    return primes;
  }
});
```

primaryOptimized.js

```
define(function() {
  return function(n) {
    n = parseInt(n);

    var a = new Uint8Array(n+1),      // Optimized to use Uint8Array instead of Array
      max = Math.floor(Math.sqrt(n)), // Don't do factors higher than this
      p = 2,                          // The first prime is 2
      i, l,
      primes = [];                    // Store the found primes

                                      // Mark all non-prime indexes
    while (p <= max) {                // Calculate primes less than max
      for (i = 2 * p; i <= n; i += p) {  // Mark multiples of p as composite
        a[i] = 1;
      }
      while(a[++p]);                  // Next unmarked index is a prime
    }
    for (i = 2, l = a.length; i < l; i++) {
      if (!a[i]) {
        primes.push(i);
      }
    }

    return primes;
  }
});
```