bdLoad

Tutorial and Reference Manual

Rawld Gill , ALTOVISO LLC < rgill@altoviso.com >

This document was generated 2011-02-11 02:17:09.

Abstract

bdLoad is a CommonJS [http://www.commonjs.org/] Modules/AsynchronousDefinition [http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition] compliant script injection loader for use loading browser-based JavaScript applications. Its design places a high value on minimum size and maximum run-time and build-time configurability. This tutorial is intended for the loader consumer and describes how to use all loader features. The tutorial also includes additional information to help build the reader's intuition and mental model of how Modules/AsynchronousDefinition loaders work in general and the backdraft loader works in particular.

Table of Contents

Overview

The use of JavaScript in the browser has advanced to the point where applications that include tens of thousands of lines of code are frequently encountered. In order to control the complexity–and, therefore, cost–of such applications, a carefully designed system for dividing large code stacks into modules and then reassembling these modules in the browser must be employed. This is the job of a loader. Since the loader's job is to load code, it is usually the first code executed in an application. Because of this special status, loaders may define a few other types of machinery:

- Configuration control
- Application bootstrap
- DOM content loaded detection
- Global namespace management

- Error detection and recovery
- Debugging machinery

Loaders are difficult to get right. Most of the problem is rooted in the desire to add features, yet keep the loader small and unobtrusive. After all, once the application is loaded, the loader basically becomes a chunk of dead code that doesn't do anything. Clearly to pay for downloading 10K onto an iphone just to load the real application is a nonstarter. So there are lots of tradeoffs and some disagreement among loader developers about which are the right tradeoffs to take. The backdraft loader was designed with the following values:

Loader size is unimportant during development; it is often critically important for released applications.
: During development, the http server is usually on the developers machine and the size of the loader has no effect on load times; conversely, when deploying to certain client targets (e.g., phones), size may be the most important feature when selecting a loader.

The loader must be highly configurable, both at run-time and build-time.
: Since the development and release requirements may be quite different as indicated above, the loader must be highly configurable. Also, different types of applications require different loader features. Programmers are highly resistive to paying for features they don't use–particularly with respect to bandwidth costs for released applications.

The loader must be able to operate without its injection machinery for released applications.
: The loader must support constructing a release that precaches all resources in a single resource and then simply uses the loader to evaluate those resources in the correct order. In this scenario, the loader injection machinery must be removable.

Core Api

require and define

The core loader API is simple, containing but two functions: `require` and `define`. Both of these functions reside in the global namespace and are available after the loader itself has been defined.

The global function `require` causes JavaScript resources to be evaluated; it has the following signature:

```
function require(configuration, dependencies, callback);

[optional]object configuration;
array of string dependencies;
[optional]function callback;
```

If `configuration` is provided, then the loader's configuration is adjusted; I'll discuss configuration in the section called “Loading the Loader”. Next, the JavaScript resources implied by the strings contained in `dependencies` are evaluated, and finally `callback` (if any) is applied to the results of those evaluations. The JavaScript resources are termed "modules", and the strings contained in the dependency vector are termed "module identifiers". `require` returns itself so that chaining several `require` applications is easy. Finally, `require` is an asynchronous function, and there is no guarantee that all of the prescribed processing has completed prior to it's return.

In order to fully understand how all of this works, we must understand...

- how a particular module identifier given in `dependencies` is resolved into some chunk of JavaScript code

- how a particular chunk of JavaScript code is evaluated and returns a value to the loader which may then be passed to `callback`

Let's answer the second question first.

When the loader is requested to evaluate a JavaScript resource, it retrieves the chuck of JavaScript that embodies the resource and causes the evaluation of that chunk in the execution environment. The actual methods used for retrieval and evaluation vary depending upon both the execution environment and the loader configuration. From now on, I'll use the term "load" to indicate the aggregate of the retrieval and evaluation process. In the browser environment, JavaScript resources are usually loaded by attaching a `script` element to the `head` element with its `src` attribute pointing to the resource. I say "usually" because the backdraft loader allows JavaScript resources to be precached.[1]

In general, the loader has no control about what a script actually does; in some environments, the loader doesn't even have control over the order of execution of demanded scripts. Further, notice that since a script may be loaded by attaching a `script` element to the document, the loader has no way of collecting a result from the script. Instead, the script must explicitly publish a result to the loader. This is the purpose of the second core function, `define`.

The global function `define` publishes the value of a module to the loader; it has the following signature:

```
void define(moduleId, dependencies, factory);

[optional]string moduleId;
[optional]array of string dependencies;
any factory;
```

`define` causes the modules given by the module identifiers contained in `dependencies` to be evaluated and then associates the value implied by `factory` with the identifier given by `moduleId` and remembers the association. If `factory` is a function, then the module value is computed by applying the function to the values of the modules implied by the dependency vector; otherwise, the module value is taken to be the value of `factory` directly. Just like `require`, `define` is asynchronous and returns immediately with no guarantee that all of the prescribed processing has completed prior to it's return. Finally, if `moduleId` is missing, then the loader derives `moduleId` from the module identifier in the dependency vector that caused the resource that contained the `define` application to be loaded. For example, if the code...

```
require(["arithmetic"]);
```

...caused the loader to load a script containing the code...

```
define({
  add: function(x, y) { return x + y; },
  sub: function(x, y) { return x - y; }
});
```

...then the loader can derive that the `define` application has the implied `moduleId` of "arithmetic". In almost all cases, `moduleId` should not be provided explicitly in a `define` application, but rather should be implied. I'll explain why later.

We can now see how the loader becomes aware of module values:

[1]This capability is used by the backdraft build system which bundles several JavaScript resources into a single resource that is loaded with a single http transaction, thereby improving the load performance (see the section called “Building Release Systems”

- the dependency vector in either a `require` or `define` application demands modules
- `define` applications contained in those modules cause the loader to associate a module value with a module identifier and remember the association.

The values passed to either the `callback` argument (in the case of `require`) or the `factory` argument (in the case of `define`, when `factory` is a function) are just module values previously associated with module identifiers. For example,

```
require(
  ["dijit/layout/TabContainer", "bd/widgets/stateButton"],
  function(tabContainer, stateButton) {
    // do something with tabContainer and stateButton...
  }
);
```

... and ...

```
define(
  ["dijit/layout/TabContainer", "bd/widgets/stateButton"],
  function(tabContainer, stateButton) {
    // do something with tabContainer and stateButton...
  }
);
```

...both gain access to the values of the `dijit/layout/TabContainer` and `bd/widgets/stateButton` modules by the *loader two-step*:

- list the module identifier in the dependency vector
- provide a parameter in the callback function definition (in the case of `require`) or the factory function definition (in the case of `define`, when `factory` is a function) that receives the value of the module listed in the dependency vector

The items in the dependency vector are matched to parameters in the callback/factory function by position. The parameter names are not significant to the loader. For example, this is perfectly legal, if not ridiculous, code:

```
require(
  ["dijit/layout/TabContainer"],
  function(supercalifragilisticexpialidocious) {
    // do something with tabContainer...
  }
);
```

Of course this all assumes that any module identifier specified in a dependency vector always results in loading a script that includes a `define` application. But, what if you just want to download and evaluate a chunk of code that doesn't define a module? That's OK too. The loader machinery will detect when the resource has been evaluated and notice that a module was not defined. In this case the loader simply notes that the module isn't really a module, but just a chunk of code. If you happen to demand the value for such a "nonmodule", the loader will provide `undefined`.

There's one last detail about retrieving module values we need to cover. What if some random chunk of code existing somewhere in your application wants a module, say `bd/widgets/stateButton`, and further, that chunk of code is not part of a callback or factory function or didn't include `bd/widgets/stateButton` in the dependency vector, yet that same random chunk of code happens to know that `bd/widgets/stateButton` has been defined? To solve this problem, the loader includes a way to directly retrieve module values from the module namespace maintained by the loader with the alternate `require` signature:

`any` **`require`**`(`*`moduleId`*`);`

`string` *`moduleId`*`;`

When `require` is provided a single string argument, that argument is interpreted as a module identifier and `require` returns the *current* value associated with that module identifier. If the given module has not been defined, then `require` returns undefined. If the given module has not be loaded, it will not load the module.

My advice: don't use this feature; it just opens up a potential program error in your application when the module you think is defined actually is not. And don't use `supercalifragilisticexpialidocious` for a variable name.

Enough nonsense; the important point is this: the space of module identifiers forms a namespace managed by the loader, and this namespace can be used by application authors to manage the global namespace. This idea is sometimes misconstrued: it is wrong to say the loader "doesn't allow global variables." The loader has no control over such matters. It is up to individual programmers to determine whether or not to pollute the global namespace. The loader merely gives machinery that programmers may use to store their top-level names.

Relative Module Identifiers

So far, we've seen module identifiers appear in two locations:

- in the `moduleId` argument to the `define` function
- in the `dependencies` argument to both the `require` and `define` functions

Module identifiers given in the dependencies vector in a `define` function application can be *relative* identifiers. For example, consider the `define` application...

```
define(
  "myPackage/myModule",
  ["myPackage/utils", "myPackage/myModule/mySubModule"],
  function(utils, submodule) {
    // do something spectacular
  }
);
```

When the loader is processing this `define` application, it understands it is defining the module `myPackage/myModule`; this is termed the "reference module" with respect to this `define` application. The loader allows module identifiers to be relative to the reference module. Therefore, you can rewrite the define application above as follows:

```
define(
  "myPackage/myModule",
  ["./utils", "./myModule/mySubModule"],
  function(utils, submodule) {
```

```
  // do something spectacular
  }
);
```

The relative module identifiers "./utils" and "./myModule/mySubModule" are relative to the reference module `myPackage/myModule`, where "./" $\Rightarrow$ "myPackage/". You can *loosely* think of "." as the parent "directory" of the current module. So we have

"./utils" $\Rightarrow$ "myPackage/utils

and

"./myModule/mySubmodule" $\Rightarrow$ "myPackage/myModule/mySubmodule

Recall that the `moduleId` argument can be implied. So, we can write...

```
require("myPackage/myModule");
```

...and then, in the JavaScript resource implied by myPackage/myModule, write...

```
define(
  ["./utils", "./myModule/mySubModule"],
  function(utils, submodule) {
    // do something spectacular
  }
);
```

Once again, ./utils and ./myModule/mySubModule are relative to the reference module, which in this case is implied. This is the best practice for defining modules:

- The module name should never be specified explicitly.
- Any module identifiers that are members of the same package as the module being defined should be specified as relative identifiers.

If these rules are followed, then the loader can provide a very powerful feature to package consumers: *the top-level namespace can be fully controlled to load two different packages with the same name and/ or two different versions of the same package*. We'll explore this in detail in the section called “Package Name Clashes”.

Let's finish up one last detail about reference modules. Suppose I have a factory function that, depending on program flow, needs to conditionally require and execute some code. For example,

```
// this is the resource for the module "myApp/topLevelHandlers"

define(
  ["dojo"],
  function(dojo) {
    dojo.connect(dojo.byId("debugButton"), "click" function() {
      require(
        ["myApp/perspectives/debug"],
        function(perspective) { perspective.open(); }
      );
    });
    // etc.

  }
);
```

The factory function simply hooks up an event handler that loads some code *if and when* the user clicks a particular button. This code is perfectly legal code, but it can be better. Notice how the `require` application uses a fully-qualified (that is, not relative) module identifier. But, since this code is in the `myApp/topLevelHandlers` module, we ought to be able to write "./perspectives/debug" instead of "myApp/perspectives/debug". Unfortunately, the global require function doesn't know anything about reference modules. What we need is a way to remember the reference module for later use.

You can get this effect by specifying the module identifier "require" in the dependency vector:

```
// this is the resource for the module "myApp/topLevelHandlers"

define(["dojo", "require"], function(dojo, require) {
  dojo.connect("debugButton", "click" function() {
    require(
      ["./perspectives/debug"],
      function(perspective) { perspective.open(); }
    );
  });

  // etc.
});
```

The require application is now executed on the lexical variable `require`--not the global `require` function. The loader arranges for this require to resolve module identifiers with respect to the reference module in which it was provided. This context-sensitive `require` function is termed a "context require". The resulting code now abides by the best practice of always using relative module identifiers when defining a module.

Resolving Module Identifiers to URLs

Let's now turn to the question of how the loader resolves a module identifier into a resource URL. Module identifiers look like file system paths, for example, `bd/widgets/stateButton`. They are given by a sequence of names separated by forward-slashes. The names may be any legal JavaScript identifier, and, by convention, are camel-case.[2] Notice that since the loader maintains a namespace of module identifiers associated with module values, the loader effectively maintains a hierarchical namespace. The top-level names in this namespace (that is, the first name in any module identifier) are termed "package identifiers" by the loader. For example, the module identifier `bd/widgets/stateButton` indicates the package `bd`, and it is correct to say `bd/widgets/stateButton` indicates the `widgets/stateButton` module in the `bd` package.

The loader may be configured to be aware of zero or more packages. This begs the question, what if a particular module identifier indicates a package that does not exist in the loader configuration? In this case, the package is taken as the "default" package and you can imagine the prefix `"default/"` is implied.[3]

[2]In particular, the CommonJS http://wiki.commonjs.org/wiki/Modules/1.1 specification requires module names be composed of camel-case identifiers.
[3]`default` is not actually defined inside the loader, but rather is used as a device to give you a mental model of how the loader works.

Then there's the other oddity when the module identifier specifies only a package identifier. In this case, the module identifier is said to indicate the package "main module" and you can imagine the suffix "/main" is implied. Let's go through a few cases to make this clear:

- Given module identifier "X", and a package named "X" has not been named in the configuration, the package identifier is implied to be "default", and the fully qualified module identifier is "default/X".
- Given module identifier "X", and a package named "X" has been named in the configuration, the package identifier is "X" and the fully qualified module identifier is "X/main".
- Given module identifier "X/Y", and a package named "X" has not been named in the configuration, the package identifier is implied to be "default", and the fully qualified module identifier is "default/X/Y".
- Given module identifier "X/Y", and a package named "X" has been named in the configuration, the package identifier is "X" and the fully qualified module identifier is "X/Y".

The take away is that a fully qualified module identifier always begins with a package identifier and always includes at least one module name within that package.

When a configuration is given for a particular package, it includes the following configuration variables:

name	The package identifier
location	A URL fragment that points to the package resources
lib	A URL fragment that, when concatenated with the location variable, points to the package JavaScript resources; if missing, defaults to "lib".
main	The name of the package main resource; if missing, defaults to "main".
packageMap	A JavaScript object that maps package names referenced in the package resources to package names known to the loader (the packages known to the loader are precisely the packages for which a configuration exists); this will be discussed in the section called “Package Name Clashes”.
urlMap	A vector of transform functions that transform computed URLs; this will be discussed in the section called “URL and Path Mapping”.

The conceptual "default" package has the following configuration values:

```
// this is ONLY conceptual; there is no real package!
name: "default",
location: "",
lib: "",
packageMap: {},
urlMap: []
// note: it's impossible to demand the main module
// in the default package
```

We now have enough information to describe how module identifiers found in dependency vectors are transformed into URLs:

URL Computation for Module Identifiers

1. The reference module is determined. If the URL is being computed consequent a module identifier in a dependency vector in a global `require` application, then there is no reference module; otherwise,

the URL must be being computed consequent to a module identifier in a dependency vector in either a context require or a `define` application, and each of these kinds of applications always have an associated reference module.

2. The reference package (if any) is determined. This is simply the package indicated by the reference module determined in Step 1.

3. If the module identifier is relative, then all of the relative names ("." and "..") are removed by concatenating the reference module, "/../", and the module identifier being resolve, and then replacing all "/./" sequences with "/" and collapsing all "x/y/../z" sequences to "x/z".

4. The target package and target module indicated by the normalized module identifier as computed in Step 3 are determined. The package name given by the module identifier is mapped by the `packageMap` configuration variable for the reference package (if any); if mapped successfully, then the mapped name indicates the target package (I'll explain the purpose of this in the section called “Package Name Clashes”). If no mapping occurs, and the package name is known to the loader, that name indicates the target package; otherwise, the "default" target package is indicated. In the first two cases (the package name was successfully mapped or was known to the loader) the target module is given by the module identifier with the first name removed; otherwise, the target module is given by the entire module identifier. If the target module is empty after the first name is removed, then the target module is set to the `main` configuration variable given by the target package.

5. The computed URL is set to the `location` configuration property of the target package concatenated with the `lib` configuration variable of the target package concatenated with the target module; concatenation inserts a "/" at each location.

6. The `paths` mapping is applied to the computed URL; I'll describe this in the section called “URL and Path Mapping”.

7. If the URL computed so far is not an absolute URL (that is, a URL that begins with a protocol like "http:" or a slash), then the value of the package-independent `baseUrl` configuration variable is prepended to the computed URL.

8. If the URL computed so far does not include a file type, then suffix ".js" is appended to the computed URL.

9. The URL map for the target package (if any) and the package-independent URL map (if any) are applied; I'll describe this in the section called “URL and Path Mapping”.

Assuming no package mapping, path mapping, or URL mapping (Steps 4, 6, and 9), let's look at some examples. Suppose the loader was configured as follows:

```
baseUrl: "./",
packages: [
  {
    name: "bd",             // the backdraft package
    location: "packages/bd" // the package baseUrl
  }, {
    name: "math",
    location: "packages/math-v1",
    lib: "source",
    main: "core"
  }
]
```

Here are some example transformations from module identifiers to URLs.

bd $\Rightarrow$ ./packages/bd/lib/main.js
The main module in the `bd` package. Since the `bd` package did not specify a `lib` configuration value, the default of "lib" is used; similarly, for the `main` configuration variable since this is the main package.

bd/widgets/stateButton $\Rightarrow$ ./packages/bd/lib/widgets/stateButton.js
The `widgets/stateButton` module in the `bd` package.

math $\Rightarrow$ ./packages/math-v1/source/core.js
The main module in the `math` package. Notice that both the `lib` and `main` configuration variables where given.

math/singleVariableDerivation $\Rightarrow$ ./packages/math-v1/source/singleVariableDerivation.js
The `singleVariableDerivation` module in the `math` package.

app/mainPanel $\Rightarrow$ ./app/mainPanel.js
The module `app/mainPanel` in the conceptual "default" package.

The Redundancy of require

One last observation before we conclude this section: the function `require` is unnecessary. Any `require` application can be converted to a `define` application by inserting an unused module identifier. For example,

```
require(
  ["this/module", "that/module", "the/other/module"],
  function(this, that, theOther) {
    // do something with this, that, and the other
  }
);
```

Can be rewritten as...

```
var uniqueModuleId= "_" + (new Date).getTime();
define(
  uniqueModuleId,
  ["this/module", "that/module", "the/other/module"],
  function(this, that, theOther) {
    // do something with this, that, and the other
  }
);
```

There's nothing special about the way I computed the unique module identifier; you could use any identifier that you know is not in use as a real module identifier. This is, in fact, exactly how the backdraft loader implements require.[4]I mention this concept not to suggest you should dispense with `require` and only use `define`, but rather to help you build on your mental model of how the loader works.

That's it for the core loader interface. There are quite a few additional features that are available. But even without these additional features, the API provides a very elegant solution for decomposing a

[4]Since the loader owns the module namespace, it can create a module identifier that is guaranteed forever unique.

complex program into a set of packages, further decomposing those packages into set of modules, and then reassembling them into a well-defined namespace in the run-time environment.

Loading and Configuring the Loader

Loading the loader into a page is trivial: simply add a script element to the page that references the loader resource. For example:

```
<script src="scripts/require.js"></script>
```

The example assumes that the "require.js" resource contains the loader definition and resides in the "scripts" directory, and the scripts directory is in the same directory as the page resource.[5] Adjust the `src` attribute accordingly if this is not the case.

When the loader definition is evaluated, it optionally computes a few configuration variables:

Automatic Configuration Variables

baseUrl	The URL path from which the loader was loaded. In the example above, the URL would be "scripts/".
main	The value of the `data-main` attribute (if any) on the `baseNode` (see next). If `main` is provided, then the loader causes `require([`*main*`])` to be executed after the loader is defined. I'll discuss this more below.
baseNode	The DOM node of the `script` element that loaded the loader. Some applications may decide to hang additional configuration off this node through custom node attributes. Since the loader needs to find the node in order to compute `baseUrl` as described above, it remembers the node in the `baseNode` configuration variable.

The process of computing `baseUrl`, `main`, and `baseNode` as described above, requires the loader sniff the document to find the `script` element that loaded the loader. The loader does this by searching for a `script` element with a `src` attribute that has a value that ends with the substring "require.js". If you rename the loader resource to something else, you'll have to adjust the sniffing code in the loader (look for the regular expression `/require\.js$/`). Notice that it doesn't make much sense to make this process configurable since if your going to provide a configuration that alters the sniff procedure, you might as well just include the configuration variables that the sniff procedure finds and dispense with the sniff procedure altogether. In fact, the sniff procedure is automatically disabled if `baseUrl` is provided in a configuration before the loader is defined (see below), and can even be removed from the code in a specially built version of the loader (see the section called “Building Release Systems”). I recommend not using the sniff procedure.

Once the loader is defined, the two-function load API (`require` and `define`) is defined in the global space. If `require` is defined as a function before loading the loader, the loader definition code will result in a no-op. In this sense, the first loader on the page wins. However, the global variable `require` *can* be defined as an *object* prior to defining the loader. In this case, `require` is understood to hold a configuration, which is nothing more than a JavaScript object with property (name, value) pairs for the various configuration variables. Here's how aa configuration provided before the loader is defined might look:

```
<head>
```

[5]This is a pretty sloppy description. Where the resources actually reside is the business of the http server. It is more-correct to say the http server resolves the URL ./scripts/require.js to the resource that defines the loader.

```
  <!-- configure the loader -->
  <script>]
    // require being defined in the global space
    var require= {
      baseUrl: "./",

      main: "myApp/main",

      // packages
      [{
        name: "util1",
        location: "packages/util1"
      }]
    };
  </script>

  <!-- load the loader -->
  <script src="scripts/require.js"></script>
```

The `require` function (both the global and context-require varieties) provides yet another overload signature that allows configuring the loader after it has been loaded:

`function` **`require`**`(`*`configuration`*`);`

`object` *`configuration`*`;`

For this signature, require returns itself to allow for each chaining. `configuration` is a JavaScript object containing configuration settings just as described above. So, you can also use the following pattern to configure the loader:

```
<head>
  <!-- load the loader -->
  <script src="scripts/require.js"></script>

  <!-- configure the loader -->
  <script>
    require({
      baseUrl: "./",

      // more configuration properties....

    });
  </script>
```

Finally, recall that the primary signature of `require` allows you to pass a configuration object as the first argument. For example...

```
require({baseUrl:"./"}, ["myApp/myModule"], function(m) {
  // ...
});
```

It is possible to provide configuration values to the loader at any point in the lifetime of the program. The loader maintains a single, global configuration. Consequently, new configuration variables are additive, but providing new values for existing variables results in overwriting the current values of those variables. For example:

```
require({baseUrl:"./"});
//require.baseUrl==="./"


require({baseUrl:"scripts"});
//require.baseUrl==="scripts"

require({
  packages: [{
    name: "mine",
    location: "packages/mine"
  }]
});

//the package "mine" is now known to the loader

require({
  packages: [{
    name: "yours",
    location: "packages/yours"
  }]
});

// BOTH the packages "mine" and "yours" are known
// to the loader.
```

The full list of configuration variables and an exhaustive example configuration is given in the section called “Configuration Variables”.

Package Name Clashes

We can now turn to the problem of package name clashes, in particular how two different packages with the same name or two different versions of the same package can be loaded into the same application.

Suppose two brilliant but independent programmers publish packages of utility functions. Because they are so brilliant neither programmer can imagine the need for yet another util package and therefore both decide to name their packages "util". You and I are just lowly developers trying to please our clients. So naturally we want to leverage brilliance and use one of these util packages. Unfortunately, we quickly discover that neither util author is quite as brilliant as they assumed and neither package contains all of the utils we need; instead, we need *both* packages. Now, we're in a jam because we have two different packages with the same name.

Fortunately, the solution is trivial. Simply install the two util packages in two different directory trees and then identify the two trees as two different packages to the loader. For example:

```

baseUrl: "./",
packages: [{
    name: "util1",
    location: "packages/util1"
  }, {
    name: "util2",
    location: "packages/util2"
  }]
```

You can access the packages through `require`:

```
require(["util1", "util2"], function(util1, util2) {
  // make client happy
});
```

Or in your own module definitions through `define`:

```
define(["util1", "util2"], function(util1, util2) {
  // make client happy
});
```

The loader resolves "util1" and "util2" into the URLs `./packages/util1/lib/main.js` and `./packages/util2/lib/main.js`, respectively. Assuming the util package authors followed the best practice and did not explicitly provide a `moduleId` argument in the define applications that create their modules, the loader provides the names "util1" and "util2" as derived from the module identifiers that caused the respective scripts to be evaluated.

Next, let's see what happens to the names in the dependency vector given when a module from one of the packages is defined. Suppose `./packages/util1/lib/main.js` contains the following code:

```
// this is the definition of the first util package main module

define(["./strings", "./collections"], function(strings, collections) {
  var core= {}, p;
  for (p in strings) core[p]= strings[p];
  for (p in collections) core[p]= collections[p];
  return core;
});
```

Since the reference module for the definition above is `util1/main`. Consequently, "./strings" ⇒ "util1/strings" and "./collections" ⇒ "util1/collections", which is exactly what we want.

Notice what has happened. We renamed the util packages on our system by specifying but two lines in a configuration, yet the packages behave perfectly without any alteration whatsoever. The key point is this: so long as the util authors used relative module identifiers to refer to modules within their own packages, we, as util package consumers, can use two different util packages, both named "util" by their authors, in the same program by providing an appropriate configuration. Pretty awesome! It gets better.

Suppose the author of the first util package used yet another package in his implementation. Maybe the definition of the `collections` module within the first util package looks like this:

```
define(["dojox/collections"], function(collections) {
  // do something with collections and return a
  // collections API for the util package
});
```

So far, so good. All the author needs to do is explain that his package depends on the `dojox` package. We, as the package consumer, simply download the `dojox` package, install it in our local package tree, inform the loader where to find it through the `packages` configuration variable and everything will work perfectly.

Not to be outdone, the second util author also depends on the `dojox` package. But, as luck would have it, the second util package depends on a different version of the `dojox` package. The loader can easily accommodate two different versions of the same package just as it can accommodate two different libraries with the same name. The problem occurs when both of the util libraries refer to the same name–"dojox"– yet expect this name to resolve to different packages.

The first thing to do is install the two different `dojox` packages just like we installed the two different util packages:

```
[{
  name: "util1",
  location: "packages/util1"
}, {
  name: "util2",
  location: "packages/util2",
}, {
  name: "dojox1",
  location: "packages/dojox-version-1-6"
}, {
  name: "dojox2",
  location: "packages/dojox-version-1-4"
}]
```

Recall that when the first utility package demands a dojox module by writing `define(["dojox/collection"], //...`, the loader will resolve that name with respect to the reference module– which is a util1 module. So, by providing a map that instructs the loader how to resolve package names with respect to a particular package, we can solve this problem. Here's the configuration that solves the dojox name clash:

```
[{
  name: "util1",
  location: "packages/util1",
  packageMap: {dojox:"dojox1}
}, {
  name: "util2",
  location: "packages/util2",
  packageMap: {dojox:"dojox2}
}, {
  name: "dojox1",
  location: "packages/dojox-version-1-6"
}, {
  name: "dojox2",
  location: "packages/dojox-version-1-4"
}]
```

Anytime the "dojox" package identifier is seen in a module from the first util package (configured as the `util1` package), the loader will inspect the `packageMap` configuration variable of the `util1` package and see that "dojox" maps to "dojox1". This is part of Step 4 in URL Computation for Module Identifiers. Similarly, the loader will resolve the package identifier "dojox" in the second util package to "dojox2". From there, the standard URL resolution algorithm proceeds as usual.

URL and Path Mapping

Back in URL Computation for Module Identifiers when I described how a module identifier is transformed into a URL, I mentioned that the computed URL could be submitted to a further transform in Step 9 of the process. That further transform is caused by the `urlMap` configuration variable. Each package may specify a `urlMap` and a single, package-independent `urlMap` configuration variable may also be provided.

`urlMap` is a vector of functions that take a URL and optionally transform that URL into other URL. If a particular function is not interested in a particular URL, it simple returns falsy to indicate as such. Once the loader transforms a module identifier to a URL, it submits the computed URL to each function in the URL map for the package (if any) implied by the reference module and then to the package-independent URL map (if any). The first function that returns a value wins and that ends of the transformation. Let's look at an example.

Suppose all HTML templates for the package "acmeWidgets" are stored in the `templates` directory rather than the `lib` directory, and further all template have the file type ".html". So, for example, when an acmeWidgets module specifies the module identifier "acmeWidgets/templates/radioButton", it is intending on pointing to `acmeWidgets/templates/radioButton.html`, not `acmeWidgets/`**lib**`/templates/radioButton`**.js**. This could be accomplished with the following acmeWidgets package configuration:

```
{
  name: "acmeWidgets",
  location: "packages/acmeWidgets",
  urlMap: [
    function(name) {
      match= name.match(/^(.+)\/lib\/templates\/(.+)(\.js)$/);
      if (match) {
        return match[1] + "/templates/" + match[2] + ".html"
      }
      return 0;
    }
  ]
}
```

Remember, the URL map will be traversed after the standard module identifier to URL transform has taken place. For example, the standard transform will take `acmeWidgets/templates/radioButton` to *baseUrl*`/packages/acmeWidgets/lib/templates/radioButton.js`. The urlMap provided above simply transforms "lib/templates" into "templates" and replaces the "js" suffix with "html".

Often, code that demands a module that is known to not be JavaScript will explicitly include the file type in the module identifier, for example "acmeWidgets/templates/radioButton.html". In the case, the resulting transform is simplified:

```
  function(name) {
    match= name.match(/^(.+)\/lib\/templates\/(.+)/);
    if (match) {
      return match[1] + "/templates/" + match[2];      }
    return 0;
  }
```

This is an example of a transform that finds a constant string (that is, the string doesn't depend on `name`) and replaces it with another constant string. In fact, this kind of transform is so common, that the backdraft loader provides a shortcut method of specifying it: provide a (`pattern`, `replacement`) ordered pair, indicating that `pattern` (a regular expression), if found in name, should be replaced with `replacement` (a string). The URL map given above could then be rewritten:

```
urlMap: [
    [/\/lib\/package\//, "templates"]
]
```

The backdraft loader also includes the so-called "paths" map that's available in other loaders. A paths map maps a set of path prefixes to replacement strings. For example, a `paths` configuration variable set like this...

```
paths: {
  "myPackage/some/path":"some/other/path"
}
```

...would map "myPackage/some/path/myModule" to "some/other/path/myModule". The backdraft loader honors any paths configuration and applies such configuration in Step 6 of URL Computation for Module Identifiers. Of course the same effect could be achieved through the urlMap feature.

In my work, I've found the paths feature lacking. It doesn't allow prioritization and only works to fix a limited set of transforms at the beginning of the path. One of the best examples of how the urlMap feature can be used to great advantage is in the dojo-sie project where I map some dojo modules to the v1.x tree and others to the v2.x tree. Of course this is a fairly unusual circumstance. Indeed, in most cases you ought not to need even a paths transform, let alone a prioritized, generalized transform like the urlMap. Nevertheless, when you need it, other solutions fall short.

Finally, the entire module identifier to URL transform process is available to client code through the method `require.toUrl`:

`string` **`require.toUrl`**`(`*`name`*`);`

`string` *`name`*`;`

`require.toUrl` submits the name argument to the module identifier to URL transform and returns the result. The `toUrl` method is provided off both the global require function and context requires created with respect to a reference module. Naturally, the latter resolves names with respect to the reference module that created the context require.

Plugins

The loader loads JavaScript resources. But there are other kinds of resources, for example templates and internationalization ("i18n") bundles that an application may need to load. Your particular application may define yet other specialized kinds of loadable resources of which the loader knows nothing. In order to accommodate this problem, the loader provides an extension point termed a "plugin" that allows a module identifier to be specified that delegates the loading of that module to additional machinery that is "plugged in" to the loader.

Here's how it works. When a module identifier contains an exclamation point, the loader splits the name into two module identifiers at the "!". The module identifier to the left of the "!" gives the name of a plugin (itself a module); the identifier to the right gives the identifier to delegate to the that plugin for loading. The loader loads the plugin (once) which must return a value that is an object that contains the function `load`:

```
void load(contextRequire, moduleId, callback);

function contextRequire;
string moduleId;
function callback;
```

Once the plugin has been loaded, the loader sends the module identifier to the right of the "!" to the `load` function:

- A context-require is manufactured with respect to the reference module that is demanding the plugin module and is passed in the `contextRequire` argument.
- The module identifier to the right of the "!" is passed in the `moduleId` argument.
- A a single-argument function that receives the value that the plugin computes for the module `moduleId` is manufactured and passed in the `callback` argument.

The plugin "loads/computes" (whatever that means to the plugin) the module implied by the `moduleId` argument and reports the value of that module to the loader through the `callback` function. Like all module values, the loader remembers the value of the module, and future demands for the same module do not result in calling the plugin `load` function. This system is incredibly elegant and powerful.[6]Here is an example of loading some raw text with the `text` plugin:

```
require(["text!myPackage/templates/myModule"], function(template) {
  // template is a string loaded from the resource implied by
  // myPackage/myModuole/myTemplage
});
```

Here is the text plugin included with dojo-sie:

```
define(["dojo"], function(dojo) {
  return {
    load: function(require, id, loaded) {
      dojo.xhrGet({
        url: require.toUrl(id),
        load: function(text) {
          loaded(text);
        }
      });
   }
  };
})
```

To my eye, this is just about as beautiful as code can get!

Notice that plugins are the primary consumers of the `require.toUrl` API. In the example above, the demand for "text!myPackage/templates/myModule" caused the loader to request the text plugin to load "myPackage/templates/myModule". It is likely that a URL map entry was also specified in the configuration so that `require.toUrl("myPackage/templates/myModule")` returns a URL something like "./packages/myPackage/templates/myModule.html".

[6]The original plugin idea in the context of the loader was invented by James Burke and this particular implementation was proposed by Kris Zyp.

DOM Content Loaded Detection

The backdraft loader includes machinery to detect the DOM content loaded condition as well as connection machinery to register and then fire callbacks upon detection of this condition. The method `require.addOnLoad` registers callbacks:

```
void addOnLoad(context, callback);

[optional]object context;
function or string callback;
```

`addOnLoad` composes `context` and `callback` into a function application:

- if context is provided and callback is a string, then callback is composed as `context[callback].call(context)`
- if context is provided and callback is a function, the callback is composed as `callback.call(context)`
- if context is not provided, the callback is compose as `callback.call(null)`

When a callback is provided to `addOnLoad`, if the DOM content loaded condition has already been achieved, the callback is immediately executed; otherwise, the loader queues the callback and then runs the queue of callbacks when the DOM content loaded condition is finally detected. The loader configuration variable `pageLoaded` indicates whether or not the DOM content loaded condition has been achieved.[7]

This functionality is often used in applications that use a synchronous loader, for example applications constructed with the Dojo v1.x synchronous loader. However, it is much less useful with an asynchronous loader such as the backdraft loader. To understand why, consider the typical application pattern you'll see in a HTML document:

1. Include a loader; this goes in the `head` element.
2. Instruct the loader to load the application source code; this also goes in the `head` element.
3. Include the html `body` element and its contents.

With a synchronous loader, Steps 1 and 2 are accomplished synchronously. The code is often downloaded by using synchronous XHR. On the other hand, the asynchronous loader never blocks on synchronous XHR calls. Consequently, browsers typically load the body element much quicker–well before the application code is downloaded. To see this effect, observe the difference in behavior of nontrivial dijit programs using either the dojo v1.x synchronous loader compared to the backdraft loader. The dojo-sie project has posted a few examples at http://dojo-sie.org/demos.html.

I find executing code dependent on the DOM content loaded status to be of little value. If the application is trivial, then the page should load fast enough that the difference between the page load event and the DOM content loaded status is negligible. In this case, a listener can be connected to the page load event like any normal event, obviating the need for the specialized and sometimes-flaky DOM content loaded machinery. If, on the other hand, the application includes so much code that it causes a noticeable delay loading, then it should be restructured so that a tiny, self-contained pacifier is included in the initial HTML resource and the real application loads while the pacifier is occupying the user. See http://bdframework.org/pacify.html for an example of this pattern. This said, DOM content loaded machinery is included if you want to use it.

[7] If your application knows that the condition has been achieved (if, for example, your application loads itself consequent to the page load event), you can explicitly set `pageLoaded` to true in the configuration. On some browsers, DOM content loaded detection is sometimes not predictable, and this technique purports to avoid the problem. Then again, if you know that the page is loaded, you shouldn't need to use this machinery. This feature is included primarily for compatibility with other loaders, and I neither see its value nor recommend its use.

Error Detection and Recovery

There are three places the loader can detect errors:

1. When a module identifier resolves to a URL that causes a download or evaluation error.

2. When a module factory causes an exception.

3. When an `onPageLoad` callback causes an exception.

Remember, the loader's job is to load chunks of JavaScript code that are expected to exist and behave. If you need to load things that may not exist and/or may not behave, and you have some plan for how to handle non-existing/misbehaving scripts, then the solution is to write a plugin to handle such resources. Over the years I've seen many elaborate error recovery systems that purport to take over and fix things when code fails to execute as expected. I always find it incredible that programmers claim that they can build systems that recognize, catch, and repair code that doesn't work as expected. This is a non sequitur: the claim asserts the ability to build faultless systems that have faults.[8]

In fact, usually the best strategy is to catch the defect at the earliest possible point in the execution flow, report all that's know about the defect, disable the particular section of the program that failed or terminate the entire program. Often the worst thing to do is muddle on as if nothing happened. Indeed, sometimes the user won't recognize the defect immediately, and then report a failure that occurs later in execution that becomes much harder to explain because the root cause was ignored.[9]

With this in mind, the loader provides the `onError` method on the global `require` function:

```
any require.onError(messageId, args);
```

```
string messageId;
array of any args;
```

`onError` maintains the property `require.onError.listeners`, which is an array of callback functions. `require.onError` sends `messageId` and `args` to the debug console and applies (`messageId`, `args`) to all callbacks in `require.onError.listeners`. As the callbacks are being applied, `onError` remembers the result of the first callback that returns a truthy value (if any). If such a result was detected, `onError` returns this result; otherwise, onError returns `0`.

The loader publishes the following three `messageId` values with associated arguments upon error detection.

1. "loader/timeout", when one or more demanded resources fail to arrive within the time limit prescribed by the configuration variable `timeout`. `args` contains an object that lists the modules that failed to arrive.

2. "loader/exec", when a module factory causes an exception. `args` contains the following values:

   a. the exception object that was thrown

   b. the fully-resolved module identifier of the module factory that threw the exception

   c. the first argument sent to the factory

   d. the second argument sent to the factory

[8]It's another matter entirely to design a fault-tolerant system to fix things when expected faults occur.
[9]Or, even worse, never know the program was misbehaving and make wrong decisions based on wrong program output.

e. etc.
If a truthy value is returned by one of the callbacks, then the value of the module that threw the exception is set to this value; otherwise, the value of the module is set to the exception object that was caught.

3. "loader/onLoad", when an `onLoad` callback causes an exception. `args` includes the exception object that was thrown.

Clients may connect to onError by pushing a callback into `require.onError.listeners`. Clients may also use `require.onError` to publish their own error detection; the loader reserves `messageIds` that start with "loader/".

Tracing

Because `require` and `define` are designed to operate asynchronously, debugging strange load problems can be frustrating. Also beware that adding breakpoints will often change the execution flow of an asynchronous process. To help this situation, the loader includes tracing machinery that sends messages to the console and then optionally uses that machinery to report loader events.

The loader provides the `trace` method on the global `require` function:

```
void require.trace(groupId, args);
```

```
string groupId;
array of any args;
```

`require.trace` calls `console.log(groupId, args[0], args[1], ...)` if and only if `req.traceSet[groupId]` is truthy.

The loader provides the following `require.traceSet` group identifiers with associated semantics:

- "loader-define": reports when the loader `define` function was applied.
- "loader-runFactory": reports just before a module factory is executed
- "loader-execModule": reports just before the dependencies for a module are evaluated
- "loader-execModule-out": reports just after all dependencies have been evaluated and the factory has ben executed.
- "loader-defineModule": reports when the arguments for a previous `define` application are about to be processed by the loader.

These can be turned on by adding an initial value of `traceSet` to the configuration. For example,

```
var require= {
  traceSet: {
    "loader-define":1,
    "loader-defineModule":1,
    "loader-runFactory":1,
    "loader-execModule":1,
    "loader-execModule-out":1
  },
  // the rest of the configuration...
```

This property can also be adjusted during program execution.

Debugging asynchronous load problems can be tricky; see Debugging Asynchronous Loading Problems [http://bdframework.org/docs/debugAsyncLoad/debugAsyncLoad.html] for more advice.

has.js

The backdraft loader is highly configurable. It uses the has.js [https://github.com/phiggins42/has.js] API to control certain aspects of its configuration. The has.js client-side API consists of a the single function has:

`boolean` **`has`**`(`*`feature`*`);`

`string` *`feature`*`;`

`has` returns true if `feature` exists, false otherwise. The backdraft loader blocks several sections of code with `has` expressions, for example,

```
  if (has("loader-timeout")) {
    // loader code to implement timeout detection goes here
  } else {
    // loader code to operate without timeout detection goes here
  }
```

Most of the `has` configuration options are intended to be used with a build tool that strips unused parts of the loader for a released system. However, one of the nice ideas behind `has` is the ability to turn features on and off during developement/testing without having to rebuild the application.

The backdraft loader will use an existing `has` implementation if it has been defined on the global variable `has` prior to defining the loader. On the other hand, if no such has is available, then the loader will define a trivial has implementation that looks like this:

```
has= function(name) {
  return require.hasMap[name];
};
```

`has` features/values may be added/edited in `require.hasMap` through the loader configuration variable `hasValues`. For example to add "myFeature" that is has the value of true, provide a configuration like this:

```
require({
  hasValues: {
    myFeature:true
  }
});
```

Providing/editing has feature values through the configuration machinery will not delete existing has feature values.

If you load a module that provides an alternate `has` implementation, then you may find it desirable to mix the `has` values known to the loader into the alternate `has` implementation. If the `has` feature "loader-pushHas" is true, then the loader will leverage the `has` API's `add` function to publish all `has` feature values known to the loader to the alternate `has` when the alternate `has`'s factory is executed.

The standard development loader includes the has features listed below; many–including the entire loader `has` implementation–can be removed in built versions of the loader depending upon your requirements.

Table 1.

feature	**semantics**
loader-sniffApi	The loader define machinery that will attempt to find the script element that included the loader definition, and, if found, the loader will calculate the `baseUrl` configuration variable from the `src` attribute, check for and read a `data-main` attribute, and set the value of the `require.baseNode` to this DOM node; this machinery is deleted otherwise. See the section called “Loading the Loader”.
"loader-traceApi"	The loader will define the `require.trace` method and make the trace groupIds listed in the section called “Tracing”available; this machinery is deleted otherwise. See the section called “Tracing”
"loader-errorApi"	The loader will define the `require.onError` method and make the error reporting listed in the section called “Error Detection and Recovery” available; this machinery is deleted otherwise.
"loader-injectApi"	The loader will define machinery to inject modules as given by URLs implied by module identifiers into the execution environment; this machinery is deleted otherwise.
"loader-timeoutApi"	The loader will define machinery to monitor the elapsed time since the last resource was requested and if/when this time exceeds the configuration variable `timeout`, publish an error as described in 1; this machinery is deleted otherwise.
"loader-catchApi"	The loader will surround each module factory execution and each `onLoad` callback with a try-catch block and exceptions will be reported as described in the section called “Error Detection and Recovery”; otherwise, the loader does not catch any exceptions.
"loader-pageLoadApi"	The loader will detect the DOM content loaded event and fire the onLoad queue; otherwise the detection machinery is deleted and callbacks applied to `addOnLoad` are fired immediately.
"loader-requirejsApi"	The loader will cause the following definitions in order to maintain more-complete requirejs compatibility: • `require.nameToUrl` === `require.toUrl` • the `ready` configuration variable will be considered equal to the backdraft `onLoad` configuration variable • `require.ready` === `require.addOnLoad` • `require.def` === `define` These aliases are not defined otherwise.
"loader-undefApi"	The loader will provide the method `undef` on the `require` function which causes a module to be discarded as if it was never defined; this machinery is deleted otherwise.
"loader-libApi"	The loader will publish its internal functions `isEmpty`, `isFunction`, `isString`, `isArray`, `forEach`, `setIns`,

feature	**semantics**
	`setDel`, `mix`, `uid`, `on`, and `injectScript` as methods of `require`.
"nativeXhr"	Indicates the environment includes a native XHR implementation; `has` standard feature detect.
"loader-amdFactoryScan"	The loader will implement the CommonJS factory scan described in Modules/AsynchronousDefinition [http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition] so long as the environment provides a working `toString` method on function objects.
"function-toString"	Indicates the environment includes a working `toString` method on function objects; `has` standard feature detect.
"dom"	Indicates the environment include the DOM; `has` standard feature detect.
"dom-addEventListener"	Indicates the environment includes W3C `addEventListener` method on DOM nodes.
"console"	Indicates the environment includes a debug console.
"console-log-apply"	Indicates the environment allows calling the `apply` method on the debug console.

Building Release Systems

Let's take a step back and think about the loader from a very high-level perspective. What is its *real* purpose?

I believe the loader exists to allow programmers to divide a big program up into packages and further divide those packages into modules, and then later reassemble it all in a controlled manner–both in terms of namespace and dependency management–in the run-time environment. During development, you want to keep things divided because debugging a 10K line script is a bad thing.

Notice that once development is complete, the loader loses much of its value. In fact, if you could take a snapshot of a freshly loaded, complete application (assuming no dynamic loading after the program starts running), you could just zip up this snap shot and distribute it as your application in a single file. This would have several advantages:

- The latency inherit in multiple http transactions would be essentially be zero (one transaction).
- The single script would be compressed more effectively than many smaller scripts.
- There would be zero synchronous inject events (for example, module A demands module B which demands module C); everything arrives at once.

Depending upon how the module resources where packaged in the single file, most of the loader would be completely unnecessary, and could therefore be discarded.

The backdraft loader includes a couple of features that allow these kinds of optimizations. First, the configuration variable `cache` contains a map from fully resolved module name (a string) to a function that causes the same code to execute that would have executed if the given module had been resolved to a URL and loaded. This allows providing a configuration that may include many scripts in a single file. Before the loader injects a module, it checks to see of it exists in the cache, and if so, simply executes the injection function; otherwise it operates normally and resolves the module to a URL and injects the given URL.

You can take this a step further if all of the modules for a particular application are contained in the cache. In this case, you can build the loader with the `has` feature "loader-injectApi" set to false. To get the absolutely smallest loader, build the loader without the sniff, trace, timeout, buildTools, requirejs, and undef APIs and the loader-amdFactoryScan. Depending upon your design, you may be able to also remove the pageLoad and lib APIs. Generally, I recommend keeping the catch and error APIs. This loader will be very small indeed, coming in at a mere 1.7K bytes!

ALTOVISO will be releasing a build tool in the winter of 2010. But a simple build tool is trivial to construct: simply wrap each module resource in a function and assign it to the `cache` property name `"`*packageId*`*`*moduleId*`"`as given by the fully resolved module.

Configuration Variables Reference

Table 2. Configuration Variables

feature	**semantics**
Basic, Package-Independent Configuration	
timeout	(integer, `0`) The number of milliseconds to wait for all requested resources to arrive before signalling an error by require.onError (see 1); zero indicates never signal an error.
baseUrl	(string) The value to prepend to computed URLs that are not absolute (see 7).
paths	(map:pathPrefix --> replacement[string]) A map, given as a JavaScript object, that maps a prefix to a computed URL to a replacement prefix for that computed URL (see 6.
urlMap	(array of function(string) --> [string | falsy]) A list of functions that take a computed URL and transform that URL into another URL or falsy (see 9
Package Configuration	

packagePaths	(map:path --> array of packageId[string]) A map, given as a JavaScript object, that maps a package location to a set of packages that ar rooted at that location. For example...
	
	 `packagePaths: {`
	`  packages: ["myPackage", yourPackage"],`
	`  extPackages: ["dojo"]`
	`}`
	
	 ...is equivalent to ...
	
	 `packages: [{`
	`  name: "myPackage",`
	`  location: "packages/myPackage"`
	`}, {`
	`  name: "yourPackage",`
	`  location: "packages/yourPackage"`
	`}, {`

feature	**semantics**

	`  name: "dojo",`
	`  location: "extPackages/dojo"`
	`}]`

packages	(array of packageConfigurationObject) An array or package configuration objects. Each object is a JavaScript object with the properties `name`, `location`, `lib`, `main`, `urlMap`, and `packageMap`; the `lib`, `main`, `urpMap`, and `packageMap` properties are optional. `lib`, if missing, defaults to `"lib"`. `main`, if missing, defaults to `"main"`. `urlMap`, if missing, defaults to `[]`. `packageMap`, if missing, defaults to `{}`.
has.js Configuration	
has	The `has.js` `has` function. The loader definition prefers this configuration variable to any existing global `has` function; and it prefers an existing global `has` function to its own, trivial has implementation.
hasValues	(map:featureName --> boolean) A map, given as a JavaScript object, that maps has.js features to values for use with the loader's has implementation. This configuration variable is not used when an external has.js implementation is provided.
useLoaderHas	(boolean) If true the module has is set to the `has` implementation known to the loader.
Build Configuration	
cache	(map:fullyQualifiedModuleId-->function() --> void) A map from a fully qualified module identifier to a function that, when executed, has the same effect as if the implied module was loaded. The fully qualified module identifier is of the form `"`*packageId*`*`*moduleId*`"`, where packageId is a package identifier known to the loader (because a package configuration with the given name exists), or the empty string, denoting the conceptual default package.
modules	(map:fullyQulaifiedModuleId --> moduleInfo) A map from fully qualified module identifier to module information that is maintained by the loader. Module information is a JavaScript object, but is subject to change without notice. This structure can be useful to watch when debugging loading problems. It should not be edited by client processes.
execQ	(array of moduleInfo) The list of modules that are waiting to run their factories. This structure can be useful to watch when debugging loading problems. It should not be edited by client processes.
Tracing Configuration	
trace	(function, see `require.trace` for prototype) The loader tracing function; may be replaced configuration. See the section called “Tracing”.
traceSet	(map:traceGroupId --> boolean) A map, given as a JavaScript object, that indicates which trace groups should emmit tracing output.
Error Publishing Configuration	
onError	(function, see `require.onError` for prototype)The loader error reporting function; may be replaced by configuration. See the section called “Error Detection and Recovery”

feature	**semantics**
DOM Content Loaded Configuration	
pageLoaded	(boolean) True if DOM content loaded is true; false otherwise.
skipIeDomLoaded	(boolean) True if an only if the Internet Explorer DOM content loaded detection algorithm should not be executed.
ready	(function) Function to execute upon detection of DOM content loaded; synonym for `addOnLoad`.
addOnLoad	(function) Function to execute upon detection of DOM content loaded.
Constant Configuration Calculated by the Loader	
host	`"browser"` if and only if loader is executing in the browser environment.
isBrowser	`true` if and only if the loader is executing in the browser environment.
baseNode	(DOM node) The script element that included the loader definition.