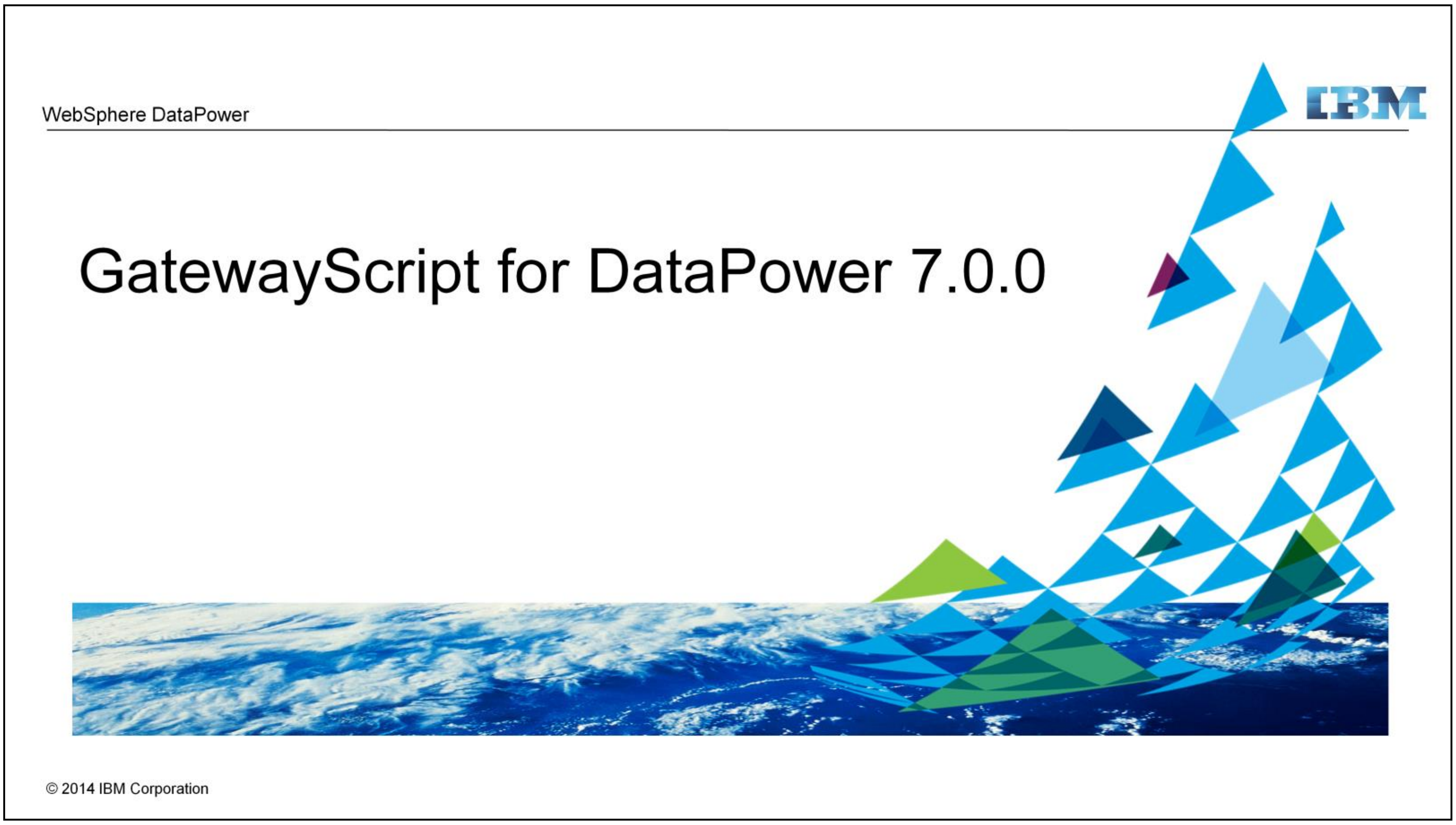
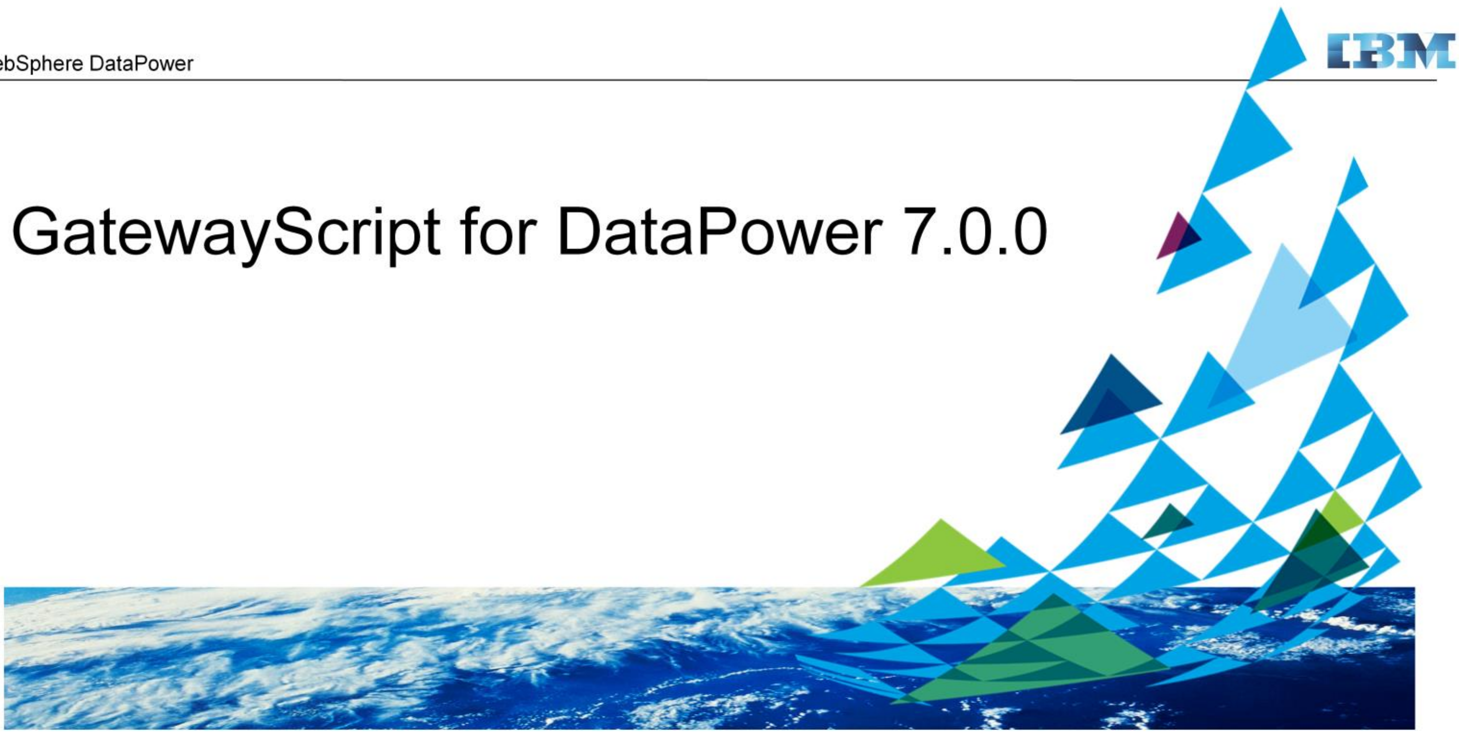
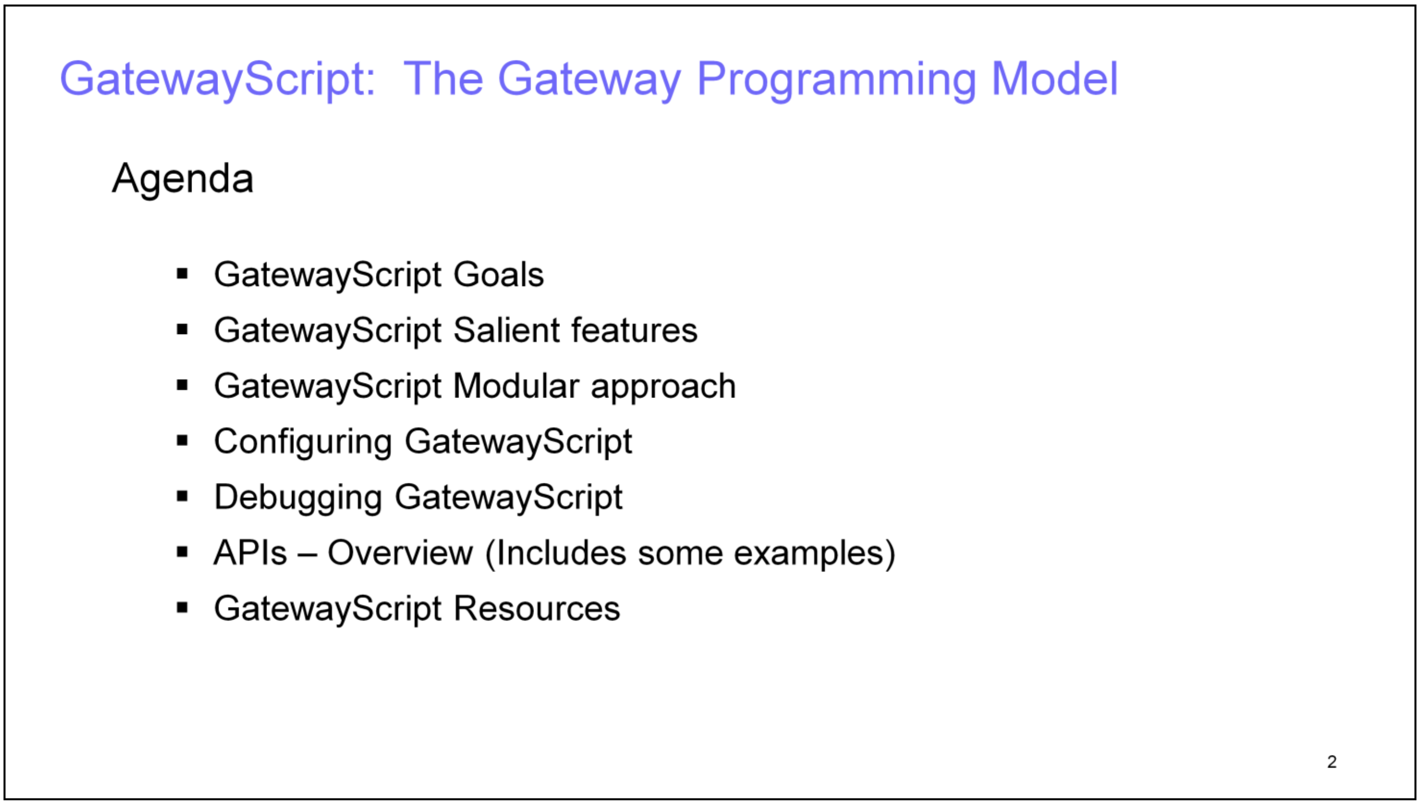


WebSphere DataPower

GatewayScript for DataPower 7.0.0



Introducing DataPower GatewayScript. This feature is new for the 7.0 release.



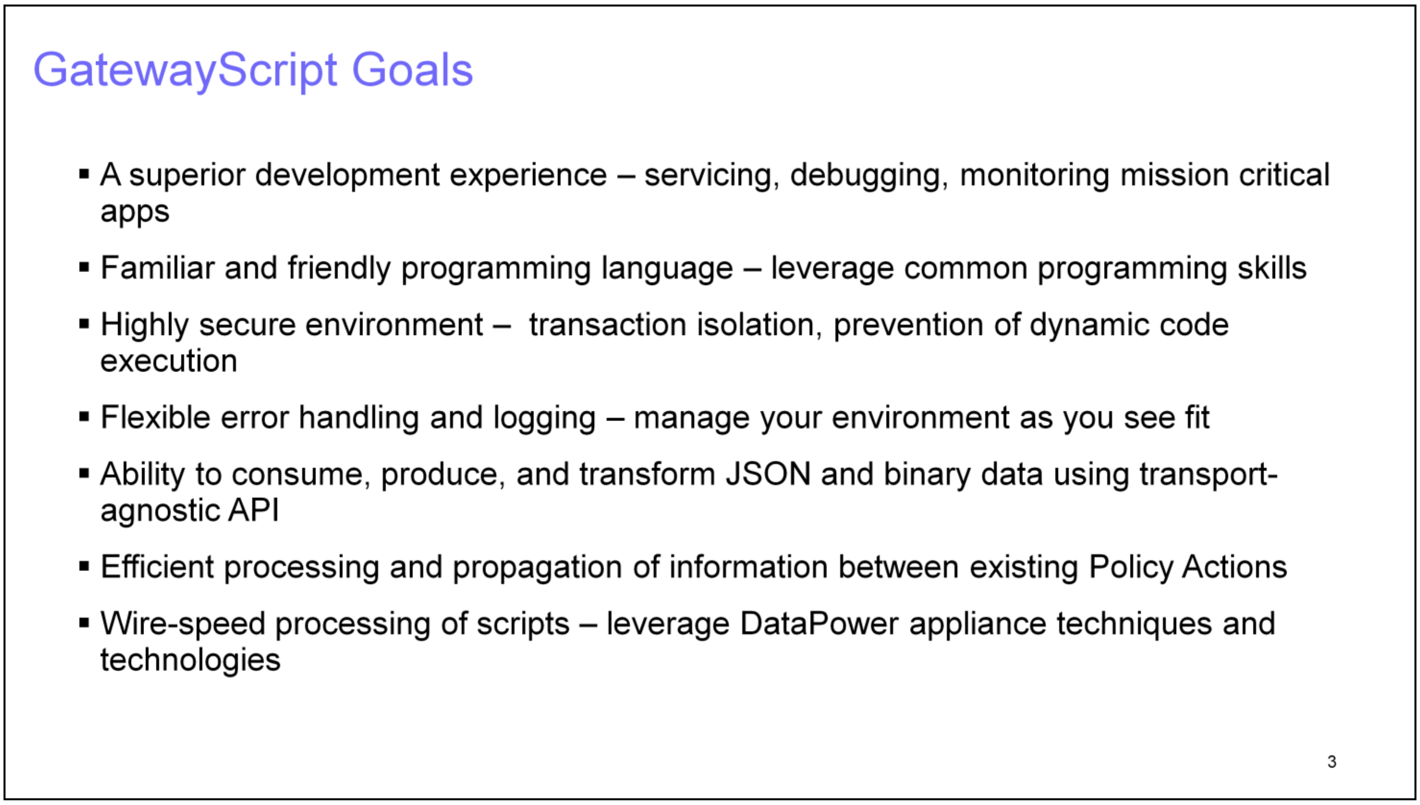
GatewayScript: The Gateway Programming Model

Agenda

- GatewayScript Goals
- GatewayScript Salient features
- GatewayScript Modular approach
- Configuring GatewayScript
- Debugging GatewayScript
- APIs – Overview (Includes some examples)
- GatewayScript Resources

2

The agenda for this presentation is as follows. First, we will discuss the goals that were set for the GatewayScript functionality. Next, we will go over the features that are implemented towards those goals. We will discuss the modular structure of GatewayScript, and how a customer can extend that structure. Next, we will share some examples of how to configure GatewayScript, and how to use the GatewayScript debugger. The next section will cover the APIs available to GatewayScript applications. Finally, we will share some resources for additional learning.



GatewayScript Goals

- A superior development experience – servicing, debugging, monitoring mission critical apps
- Familiar and friendly programming language – leverage common programming skills
- Highly secure environment – transaction isolation, prevention of dynamic code execution
- Flexible error handling and logging – manage your environment as you see fit
- Ability to consume, produce, and transform JSON and binary data using transport-agnostic API
- Efficient processing and propagation of information between existing Policy Actions
- Wire-speed processing of scripts – leverage DataPower appliance techniques and technologies

3

The first goal is to improve the overall user experience. IBM drives to continually improve in the areas of application development, monitoring, debugging, and servicing.

With the addition of GatewayScript, the DataPower appliances incorporate a programming language that has a much larger audience. XSLT is an incredibly powerful tool, but it requires a more specialized skill set. GatewayScript is a programming language that is familiar to the large numbers of programmers, especially to new college graduates.

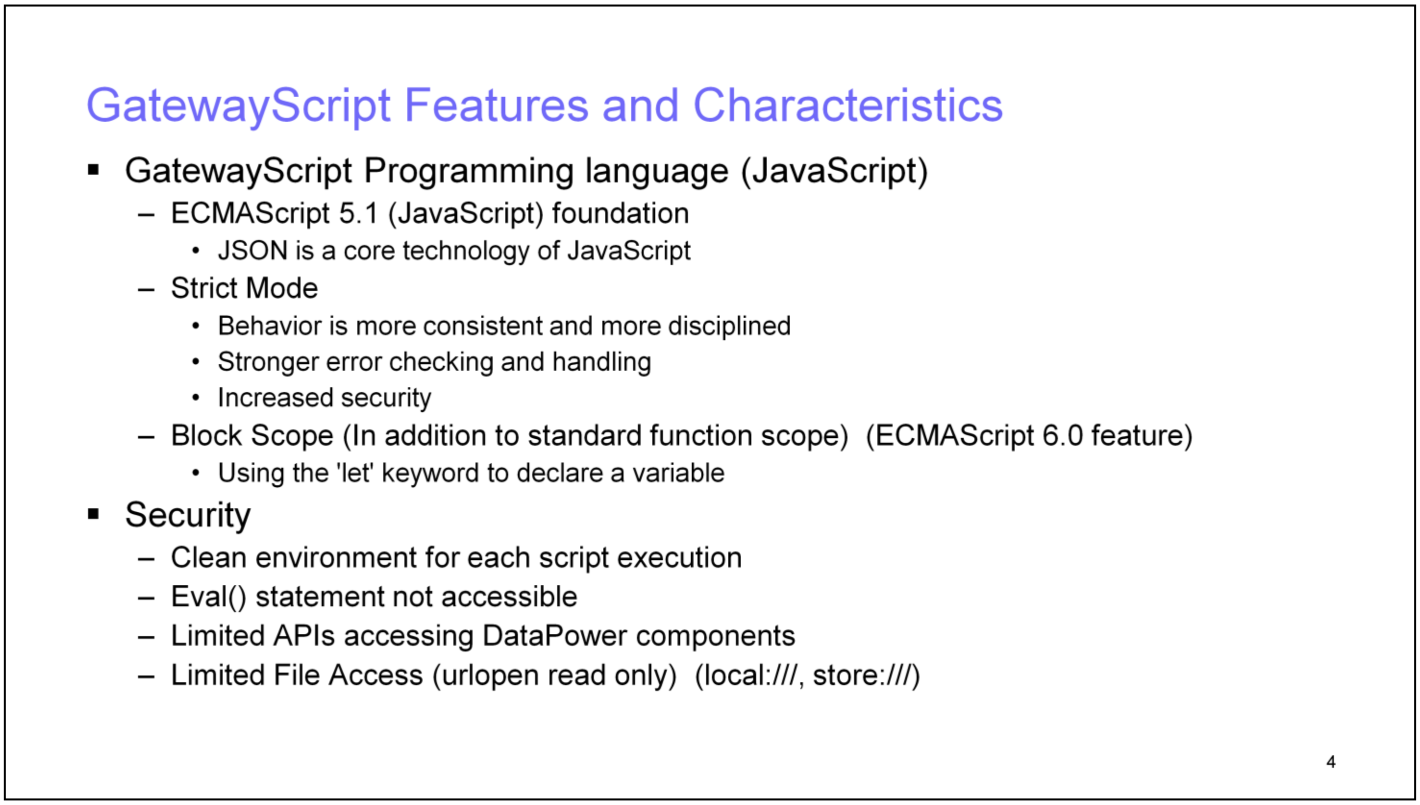
Security is critical to the DataPower philosophy. GatewayScript is no different. GatewayScript follows the same principles that XSLT and the other transform technologies provide.

Error handling is consistent in all aspects of GatewayScript. Logging is consistent with the DataPower system logging strategy. But, GatewayScript also allows the developer to use logging in a flexible and easy manner.

JSON is at the heart of most Web 2.0 RESTful applications, and it is the payload of choice for mobile applications. In earlier DataPower firmware versions, several features were added to extend the JSON processing capabilities. GatewayScript adds the ability to work with JSON as easily and as efficiently as the DataPower XSLT implementation works with XML.

The GatewayScript actions and behaviors are intuitive and consistent with all of the other actions that the DataPower appliance provides. GatewayScript actions can communicate with non-GatewayScript actions and vice versa.

Finally, the GatewayScript feature performance is commensurate with XSLT.



GatewayScript Features and Characteristics

- GatewayScript Programming language (JavaScript)
  - ECMAScript 5.1 (JavaScript) foundation
    - JSON is a core technology of JavaScript
  - Strict Mode
    - Behavior is more consistent and more disciplined
    - Stronger error checking and handling
    - Increased security
  - Block Scope (In addition to standard function scope) (ECMAScript 6.0 feature)
    - Using the 'let' keyword to declare a variable
- Security
  - Clean environment for each script execution
  - Eval() statement not accessible
  - Limited APIs accessing DataPower components
  - Limited File Access (urlopen read only) (local:///, store:///)

4

The next two slides discuss the features and characteristics of GatewayScript.

GatewayScript is based on the JavaScript programming language. More precisely, it supports the ECMAScript 5.1 version of the specification.

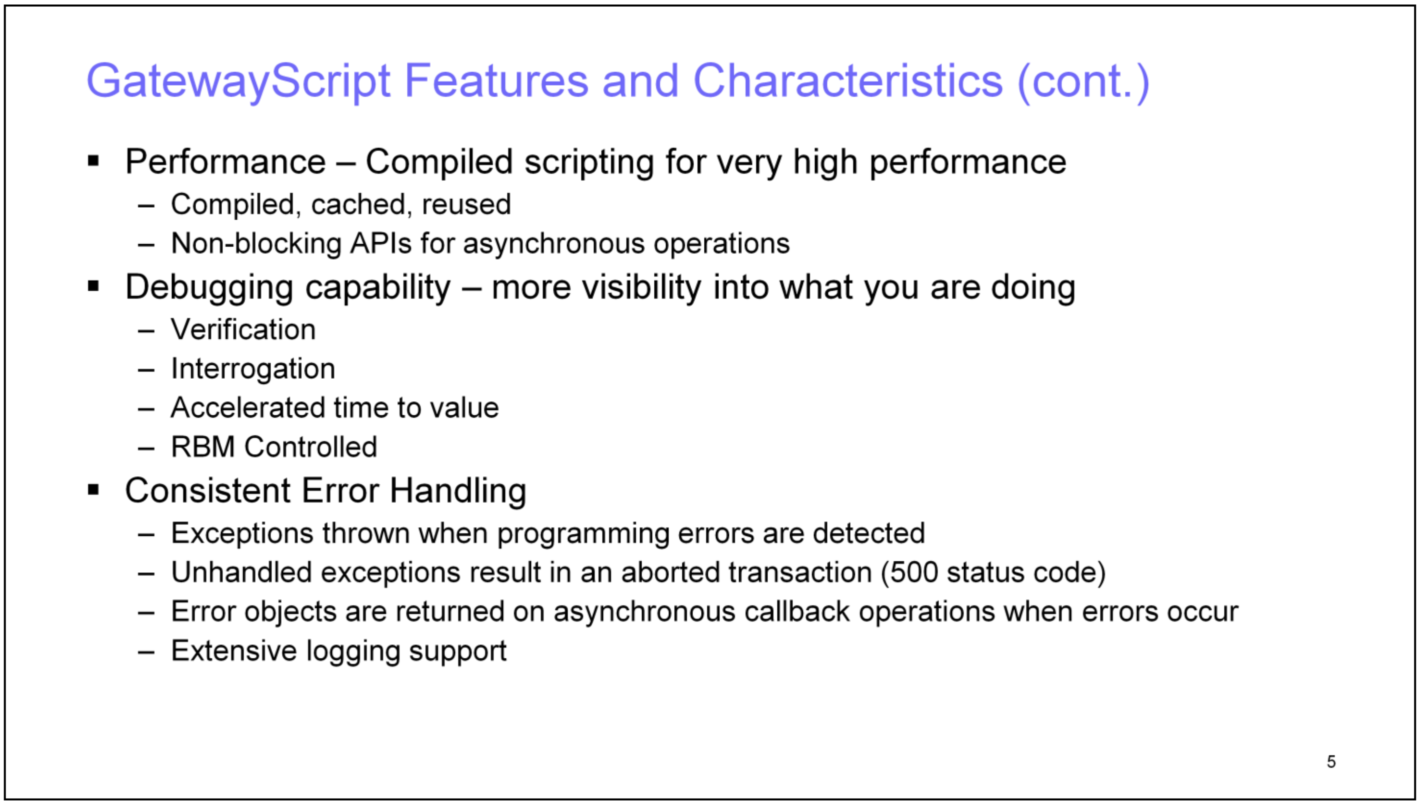
Why JavaScript? JavaScript uses JSON as part of the language itself. JSON is essentially the data structure or object structure of the JavaScript language. JavaScript is an incredibly popular language, and has been a mainstay of web applications for many years. There is JavaScript running in the browsers, and in the backend servers, and now within the Gateway or middleware tier.

On the DataPower appliance, JavaScript runs in ‘strict’ mode only. Strict mode provides a more consistent and more disciplined behavior. For example, if a script attempts to write a property of a DataPower API object and that object is frozen, an immediate exception is thrown. The ‘strict’ mode behavior supports stronger error checking and handling and increases security.

An ECMAScript 6.0 feature is supported in GatewayScript. That feature is “Block Scoping”. ECMAScript 5.1 provides for only function scoping of variables. The block scoping feature allows a developer the ability to provide better separation and visibility characteristics. To use block scoping, a variable must be declared with the ‘let’ keyword instead of the ‘var’ keyword.

Security has several aspects. First, the execution environment for each script must be pristine. There are no cases where one GatewayScript action can affect another action whether running in parallel or running serially. There are no threading implications where proper locking is required to protect critical sections. With the new GatewayScript implementation, all of these aspects are true. Additionally, the Eval() statement is disabled.

In addition, GatewayScript limits access to the DataPower file system. GatewayScript provides a series of libraries that give the developer access to specific APIs. All access to DataPower facilities is provided through these APIs. Access to the file system, for example is done with the urlopen API. And, that API allows access to only the local:/// and store:/// file systems. There is no direct access to read or write a file with a GatewayScript application.



GatewayScript Features and Characteristics (cont.)

- Performance – Compiled scripting for very high performance
  - Compiled, cached, reused
  - Non-blocking APIs for asynchronous operations
- Debugging capability – more visibility into what you are doing
  - Verification
  - Interrogation
  - Accelerated time to value
  - RBM Controlled
- Consistent Error Handling
  - Exceptions thrown when programming errors are detected
  - Unhandled exceptions result in an aborted transaction (500 status code)
  - Error objects are returned on asynchronous callback operations when errors occur
  - Extensive logging support

5

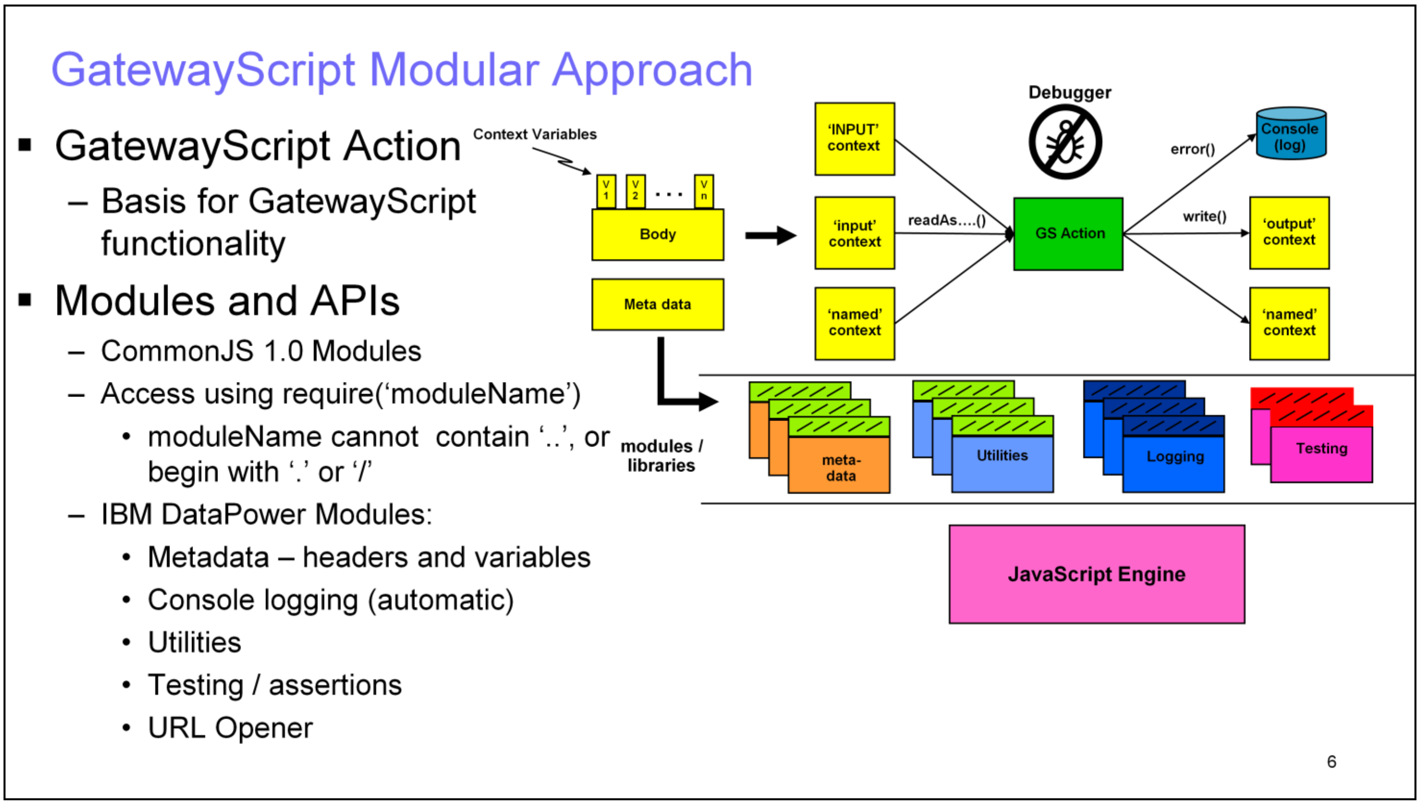
Performance is very important in the GatewayScript implementation. All scripts are compiled into object code. The object code is then cached and run. When a subsequent request comes in that requires an already compiled file, the cached copy is run directly. The cached copy is run unless the underlying file time stamp changes. If a newer version is found, it is compiled, and replaces the obsolete cached version.

GatewayScript also implements non-blocking APIs for asynchronous operations. An example of an asynchronous operation is the urlopen function. A request is transmitted off box and a response is received to that request might take a substantial amount of time. The non-blocking API allows processing to continue doing productive work in the GatewayScript application in parallel with the asynchronous operation. When the asynchronous operation completes, a function callback is invoked to address processing of the response. This implementation allows for a high level of concurrency and efficiency.

Debugging. For the first time, DataPower firmware includes a debugger. The debugger is available through only the Command Line Interface (CLI) in release 7.0. With the debugger, a GatewayScript program can be verified by stepping through the code and interrogating variables. The debugger is Roll Based Management (RBM) controlled, so you can limit access to these facilities.

The error handling for GatewayScript is simple and consistent. If the DataPower appliance detects a programming error, then an exception is thrown immediately. If the exception is not caught (that is, the exception is unhandled), then the transaction aborts and returns a 500 response code to the client. Asynchronous operations do not lend themselves to exceptions. So, for asynchronous operations, an error object is returned to the asynchronous function callback if an error occurs. The exception objects and the error objects are consistent. They contain textual information as to what happened. They also contain more in-depth descriptions and suggestions on how to remedy the situation.

The logging infrastructure is extended dramatically for GatewayScript. A large number of error messages are available to help the administrator understand error situations. Script developers have many options to log the information that they see fit.



GatewayScript Modular Approach

- GatewayScript Action
  - Basis for GatewayScript functionality
- Modules and APIs
  - CommonJS 1.0 Modules
  - Access using require('moduleName')
    - moduleName cannot contain '..', or begin with '.' or '/'
  - IBM DataPower Modules:
    - Metadata – headers and variables
    - Console logging (automatic)
    - Utilities
    - Testing / assertions
    - URL Opener

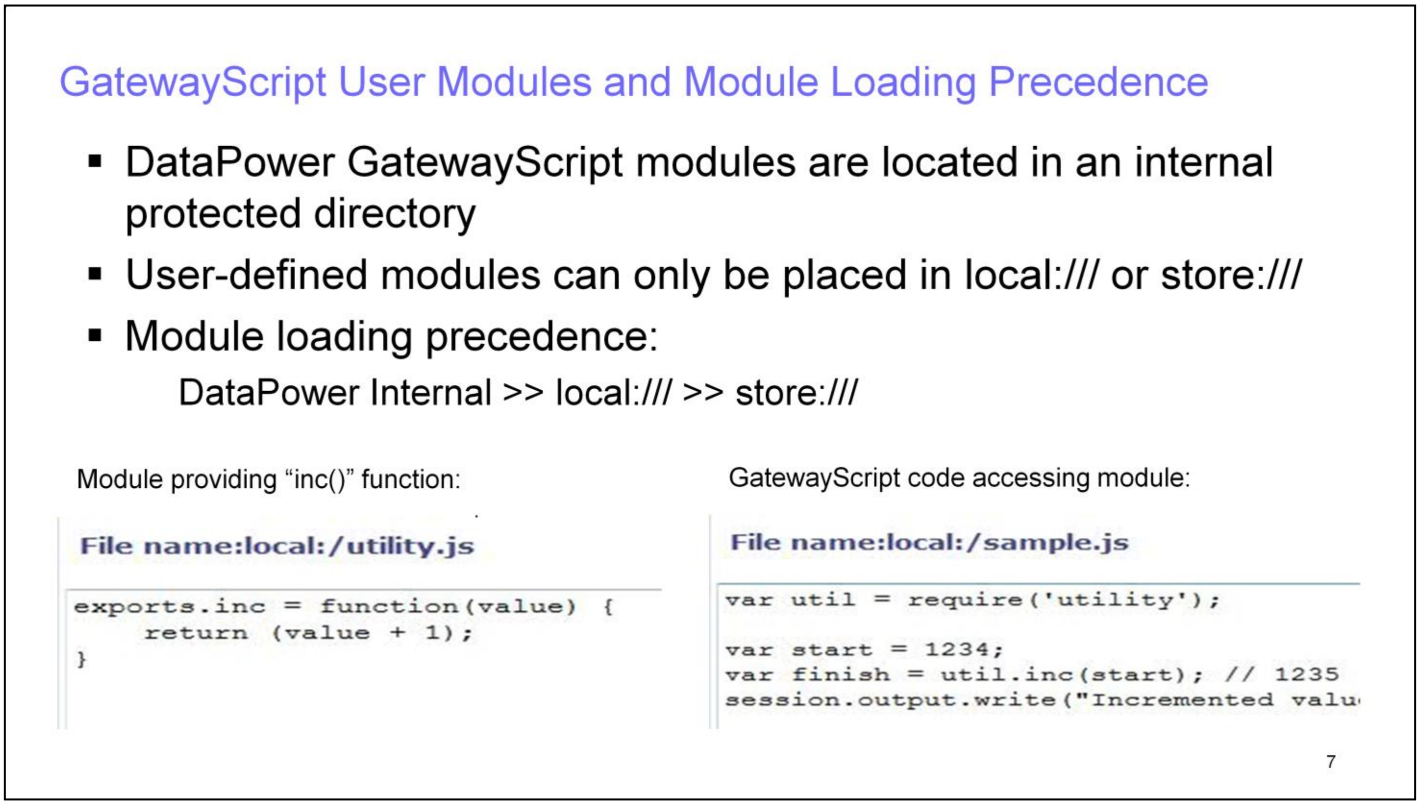
Context Variables
V1
V2
. . .
Vn
Body
Meta data
modules / libraries
'INPUT' context
'input' context
'named' context
readAs....()
Debugger
GS Action
error()
write()
Console (log)
'output' context
'named' context
meta-data
Utilities
Logging
Testing
JavaScript Engine

6

DataPower firmware implements GatewayScript with a modular approach. These modules or libraries can be loaded on demand by the application. This implementation supports an extendable system with a high degree of isolation between modules. The modular approach is also available to write or include modules from third parties. GatewayScript supports the CommonJS 1.0 Module format. If the module complies with the module specification, it loads and runs. Modules must also comply with the APIs and restrictions in the DataPower appliance. For example, a module cannot arbitrarily access the file system.

The GatewayScript environment is shown in the diagram. The GatewayScript processing action is the only way to invoke GatewayScript in version 7.0. The GatewayScript action is shown as the green rectangle. From within the action, a running program can read an 'input' context. A context in DataPower is often the payload or body of a request or response. Lowercase 'input' refers to reading the context configured as input to that action. You can also access the 'INPUT' (all capital letters) context, which is the payload of the request as it was received off the wire. Lastly, you can access a 'named' context that was created for this transaction either by this action or by a previous action. Likewise, you can write the payload to an 'output' context or a 'named' context. You can also write any output to the system log through the console API.

Context variables, that are shown attached to a context, can also be created, read, or written. The context variable is a typed variable and is easily accessed through a synchronous API, whereas the context must be accessed through an asynchronous API. But, there is much more required than just accessing the body of a request. There are operations such as interrogating headers, checking the URL, digging through query strings, and other operations. These types of operations work on the metadata of the transaction. In GatewayScript, this metadata is accessible through the module interface. More details on the modules that are included with GatewayScript are discussed in a future chart.



GatewayScript User Modules and Module Loading Precedence

- DataPower GatewayScript modules are located in an internal protected directory
- User-defined modules can only be placed in local:/// or store:///
- Module loading precedence:

  DataPower Internal >> local:/// >> store:///

Module providing “inc()” function:

File name:local:/utility.js

```
exports.inc = function(value) {
    return (value + 1);
}
```

GatewayScript code accessing module:

File name:local:/sample.js

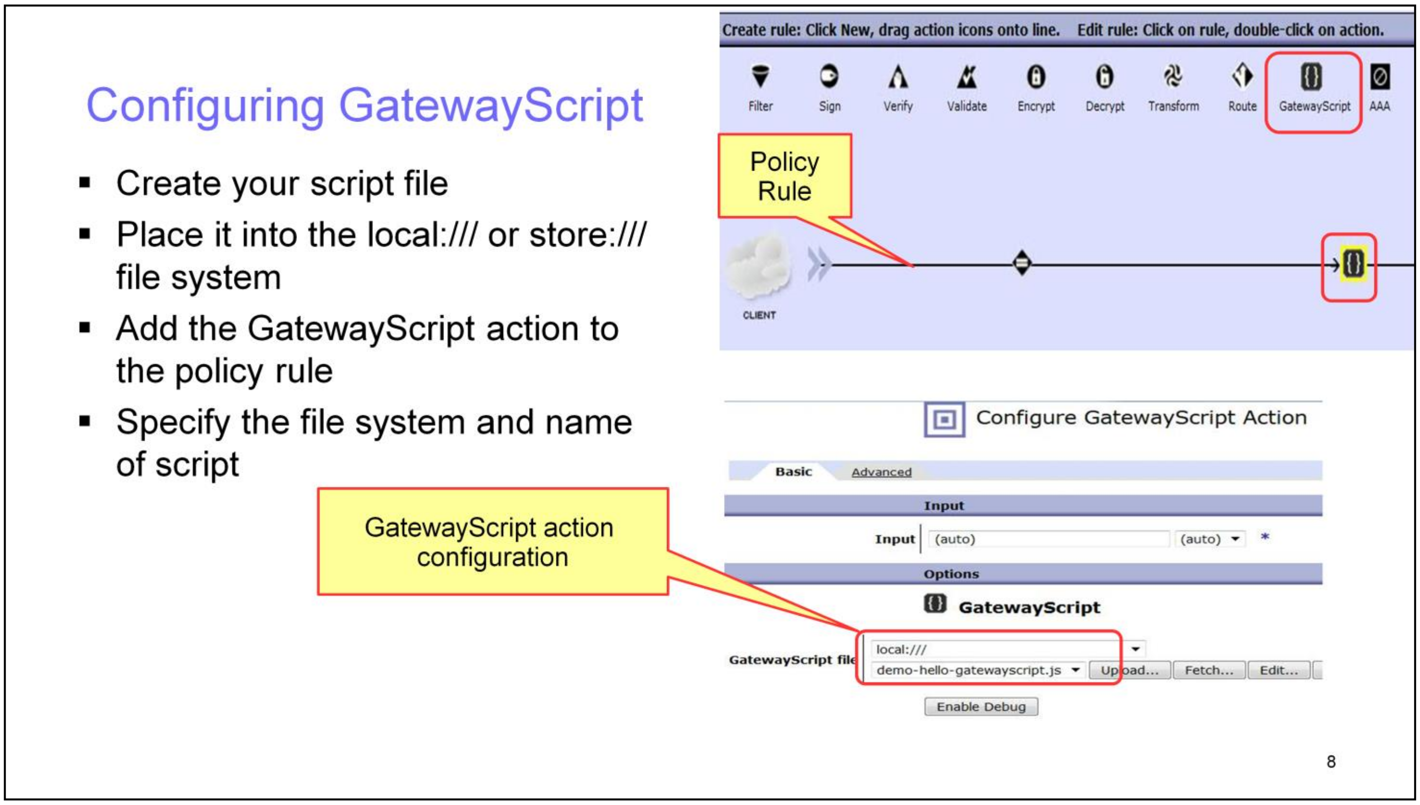
```
var util = require('utility');

var start = 1234;
var finish = util.inc(start); // 1235
session.output.write("Incremented valu
```

7

All modules that are included in the DataPower firmware are held in an internal directory that cannot be read or written. Modules that are supplied by a customer must be placed in either the local:/// or store:/// directories. When a module is requested, the appliance first searches its internal directory followed by the local:/// directory, and then the store:/// directory. You can override the search precedence by specifying the fully qualified file name, for example “require(‘local:///utility.js’);”.

Also shown on this chart is a simple module and how to run it. The file that is holding this module is utility.js, and it is placed in the local:/// directory. The module contains a single increment function named “inc()”. The file on the right is a script that runs as a GatewayScript action. The first line includes the utility module with the “require” statement. Basically, the util variable is assigned the exported object where each attribute is a function or an object that is exported by the module. In this case, only one function is exported, the inc() function. The inc() function is run by the “var finish = util.inc(start);” statement.

This example is very simple, but you can see how easy it is to create a modular solution.



Configuring GatewayScript

- Create your script file
- Place it into the local:/// or store:/// file system
- Add the GatewayScript action to the policy rule
- Specify the file system and name of script

Create rule: Click New, drag action icons onto line. Edit rule: Click on rule, double-click on action.

Policy Rule

Configure GatewayScript Action

Input: (auto) (auto) *

Options

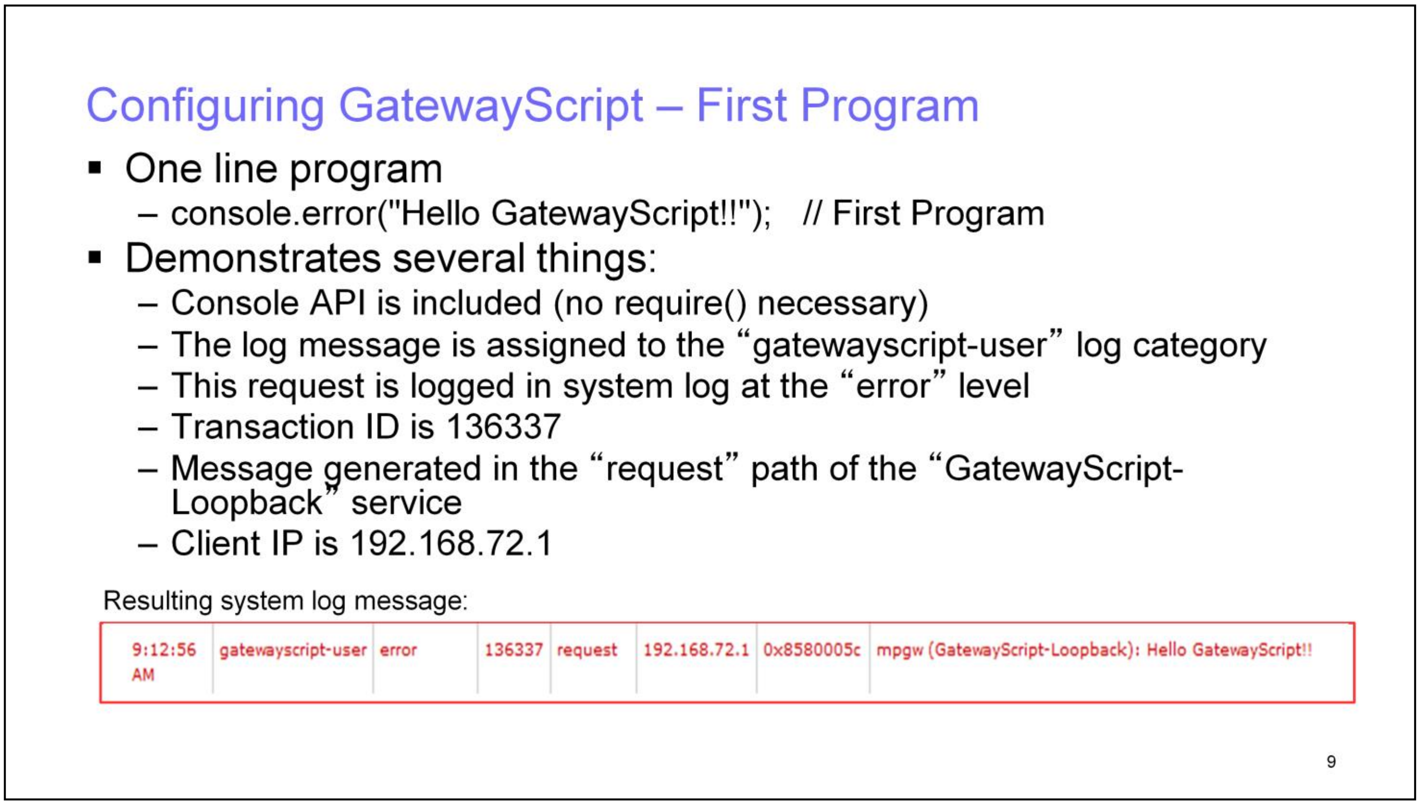
GatewayScript

GatewayScript file: local:/// demo-hello-gatewayscript.js

GatewayScript action configuration

8

Configuring a GatewayScript action is a simple task. You must first create a file that contains a GatewayScript program. Place that file in either the local:/// or store:/// directories. The only way to run a GatewayScript program is through a Gateway Script processing policy action. The upper right diagram shows a processing policy rule. Along the top of the display, you see a pallet of icons. One of the icons, new for 7.0, is the GatewayScript icon. Drag a GatewayScript action to your rule just like any other action. Double-click the icon in the rule, and the lower window opens up. Select the file system where the file that contains the GatewayScript program is, and then select the file. Click done, and the GatewayScript action is configured.



Configuring GatewayScript – First Program

- One line program
  - console.error("Hello GatewayScript!!");   // First Program
- Demonstrates several things:
  - Console API is included (no require() necessary)
  - The log message is assigned to the “gatewayscript-user” log category
  - This request is logged in system log at the “error” level
  - Transaction ID is 136337
  - Message generated in the “request” path of the “GatewayScript-Loopback” service
  - Client IP is 192.168.72.1

Resulting system log message:

```
9:12:56  gatewayscript-user  error  136337  request  192.168.72.1  0x8580005c  mpgw (GatewayScript-Loopback): Hello GatewayScript!!
AM
```

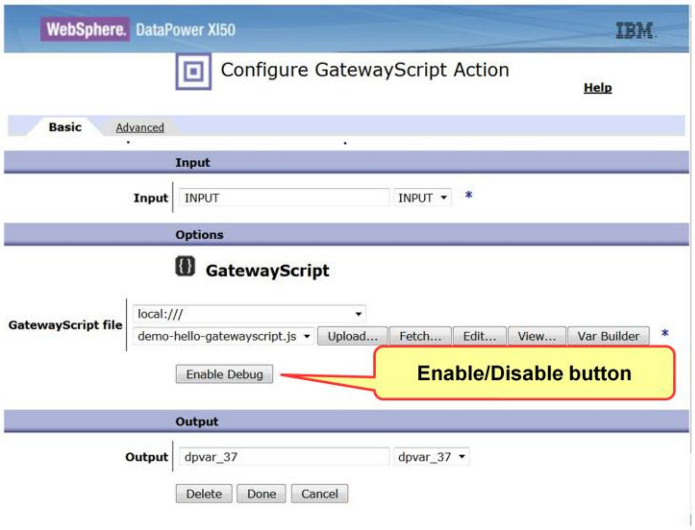
9

Here is your first GatewayScript program. It is not a long program, just a single line: “console.error(“Hello GatewayScript!!”);”.   This line logs the “Hello GatewayScript!!” message to the system log. But, a great deal of information is also logged. The first field is a time stamp. The “gatewayscript-user” is the logging category. This log message was logged at the ‘error’ level. You might, instead, use console.info() to log the message at the ‘info’ level. There is a transaction ID of 136337. The action is running on the ‘request’ path. It was received from a client with the IP address of 192.168.72.1. It has a message ID of 0x8580005c, which is used for all GatewayScript user generated log messages. It shows the service type (“mpgw”) and the service name (“GatewayScript-Loopback”), and finally, the specified text.

A lot can be done with a single line of code in GatewayScript.

Debugging GatewayScript

- Two conditions must be met:
  - The GatewayScript action must have debugging enabled
  - The Script invoked in the GatewayScript Action must contain a “debugger;” statement



Configure GatewayScript Action

Help

Basic | Advanced

Input

Input: INPUT | INPUT *

Options

GatewayScript

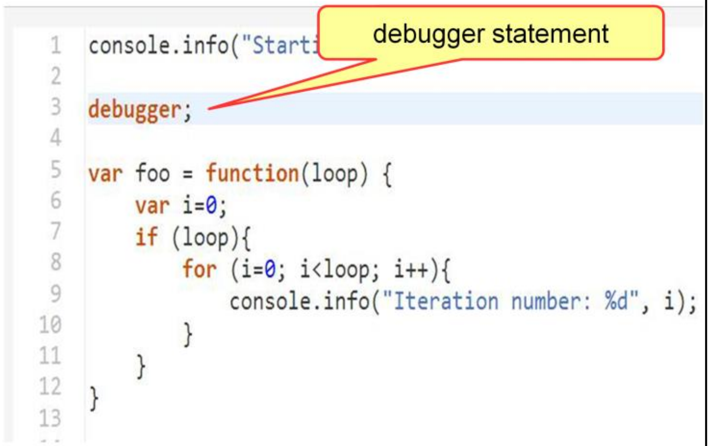
GatewayScript file: local:/// | demo-hello-gatewayscript.js | Upload... | Fetch... | Edit... | View... | Var Builder *

Enable Debug

Output

Output: dpvar_37 | dpvar_37

Delete | Done | Cancel



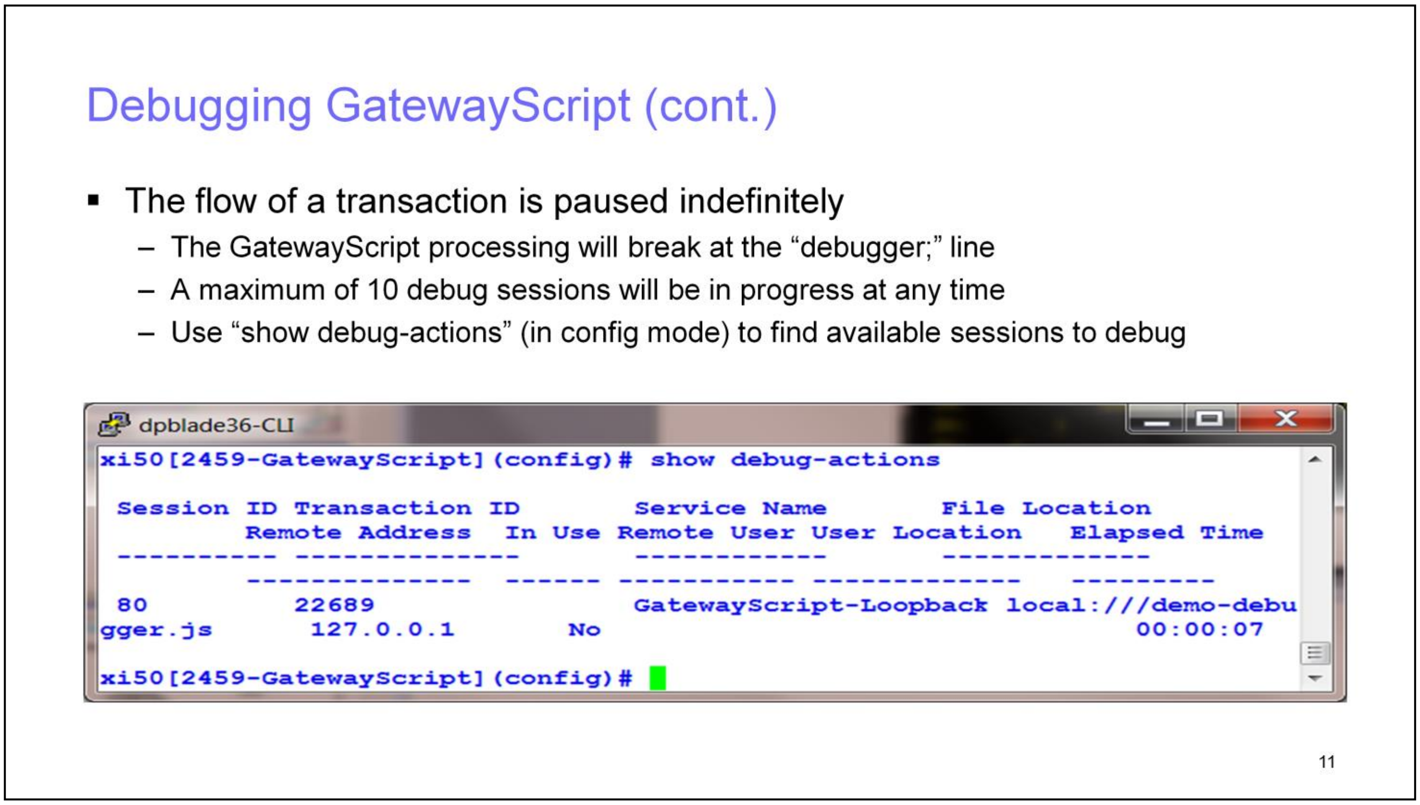
```
console.info("Starti

debugger;

var foo = function(loop) {
    var i=0;
    if (loop){
        for (i=0; i<loop; i++){
            console.info("Iteration number: %d", i);
        }
    }
}
```

The next few charts show how to run the debugger and what its capabilities are.

For the first time, DataPower firmware includes a debugger explicitly for the GatewayScript feature. The debugger is a Command Line Interface (CLI) based debugger only. To access the debugger, two conditions must be met:

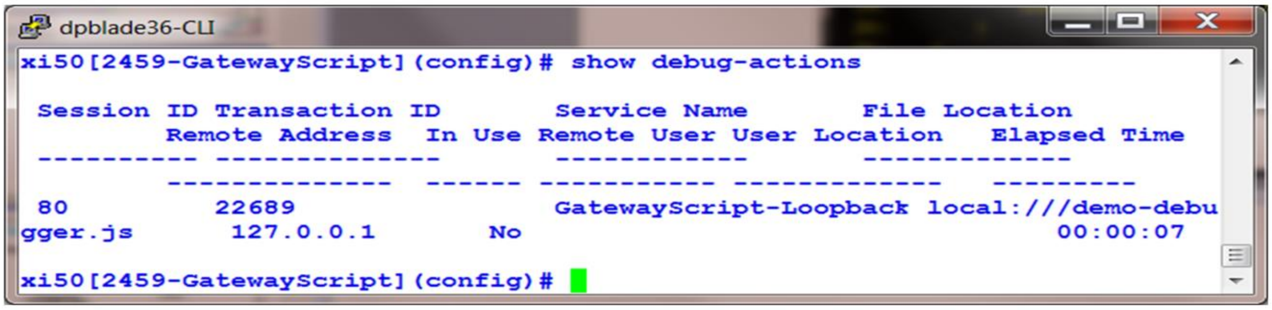
In the GatewayScript Action configuration, you must click the “Enable Debug.” This configuration option is not persisted, so if you reboot the box, all debugging is disabled.

You must have a “debugger;” statement in the flow of the script. The debugger statement acts as the first breakpoint in your script.



Debugging GatewayScript (cont.)

- The flow of a transaction is paused indefinitely
  - The GatewayScript processing will break at the "debugger;" line
  - A maximum of 10 debug sessions will be in progress at any time
  - Use "show debug-actions" (in config mode) to find available sessions to debug



```
xi50[2459-GatewayScript](config)# show debug-actions

 Session ID Transaction ID       Service Name       File Location
        Remote Address  In Use Remote User User Location   Elapsed Time
 ---------- --------------       ------------       -------------
        --------------  ------ ----------- -------------   ---------
 80         22689                GatewayScript-Loopback local:///demo-debu
gger.js      127.0.0.1      No                                00:00:07

xi50[2459-GatewayScript](config)#
```

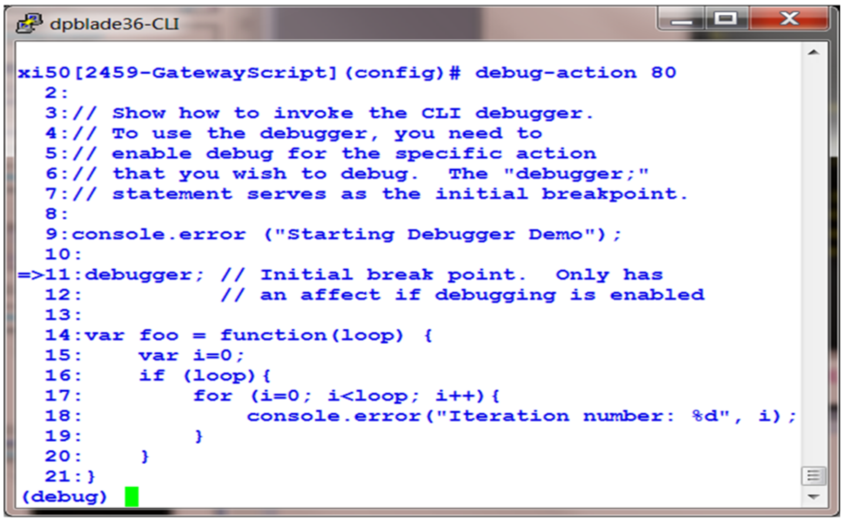
After you meet the two conditions on the previous slide, when a request is received that runs that GatewayScript action, the request hangs indefinitely. If two requests are sent, both hang. And, any other requests also hang, up to a maximum of 10. After the 10th request is suspended, all other requests will ignore the debugger; statements and run as though no "debugger;" statement is present.

Each held request is kept indefinitely until the session is debugged, canceled, or the box is rebooted. To find out what requests are held, enter the "show debug-actions" command while in config mode. The command output shows you the current set of debug sessions. There is one session for each transaction that is suspended. Note in the screen capture that there is a session id of '80'. The output lists the Transaction ID, the Service Name, the File Location, and the IP address of the client. Also listed is whether this debug session is in use, that is, whether the session is being debugged by another user. If the session is already being debugged, the user name and IP address are displayed.

The important thing to remember is that you need the Session ID to tell the debugger which session you want to debug.

Debugging GatewayScript (cont.)

- Enter the CLI debugger – gdb-like interface
  - Must be in config mode in the domain where the action ran
  - debug-action <session ID> -- enter the CLI debugger until the script completes.



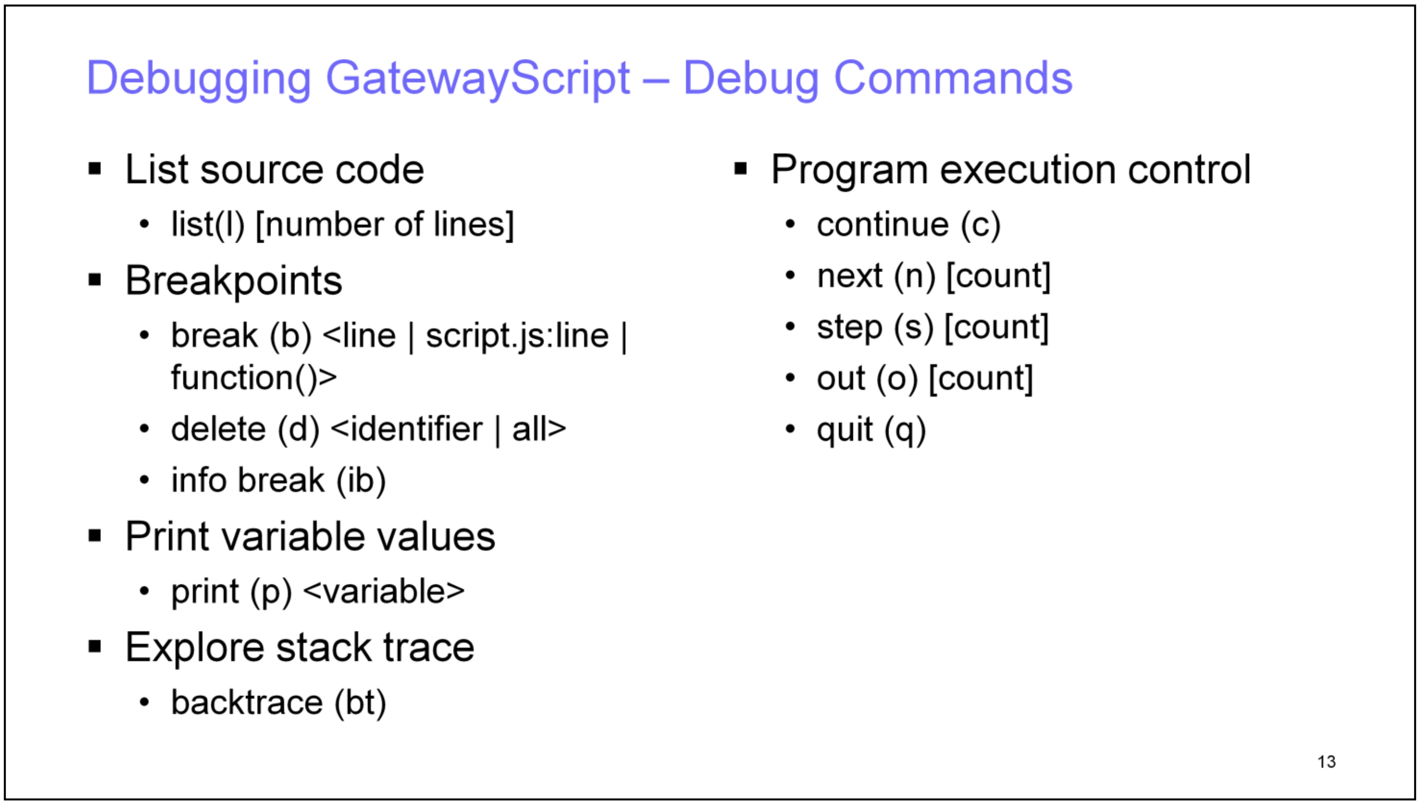
dpblade36-CLI

```
xi50[2459-GatewayScript](config)# debug-action 80
   2:
   3:// Show how to invoke the CLI debugger.
   4:// To use the debugger, you need to
   5:// enable debug for the specific action
   6:// that you wish to debug.  The "debugger;"
   7:// statement serves as the initial breakpoint.
   8:
   9:console.error ("Starting Debugger Demo");
   10:
=>11:debugger; // Initial break point.  Only has
   12:           // an affect if debugging is enabled
   13:
   14:var foo = function(loop) {
   15:     var i=0;
   16:     if (loop){
   17:          for (i=0; i<loop; i++){
   18:               console.error("Iteration number: %d", i);
   19:          }
   20:     }
   21:}
 (debug)
```

After you display the sessions and pick a session to debug, enter “debug-action” followed by the session ID number. You immediately enter the CLI debugger. It displays the current list of code and includes an ‘=>’ indicator at the program counter location.   On entry, execution stops at the “debugger;” statement.

From this point on, you can enter commands in a similar fashion to the GDB debugger.

It is worth noting that the “debugger;” statement must be in the execution path. If you have conditional logic around it, your debug session is created only if that conditional logic is satisfied. This logic provides a key debugging tool. If you have something that is failing on an infrequent request, you can set up special logic to catch that request real time and then trace through the logic.



Debugging GatewayScript – Debug Commands

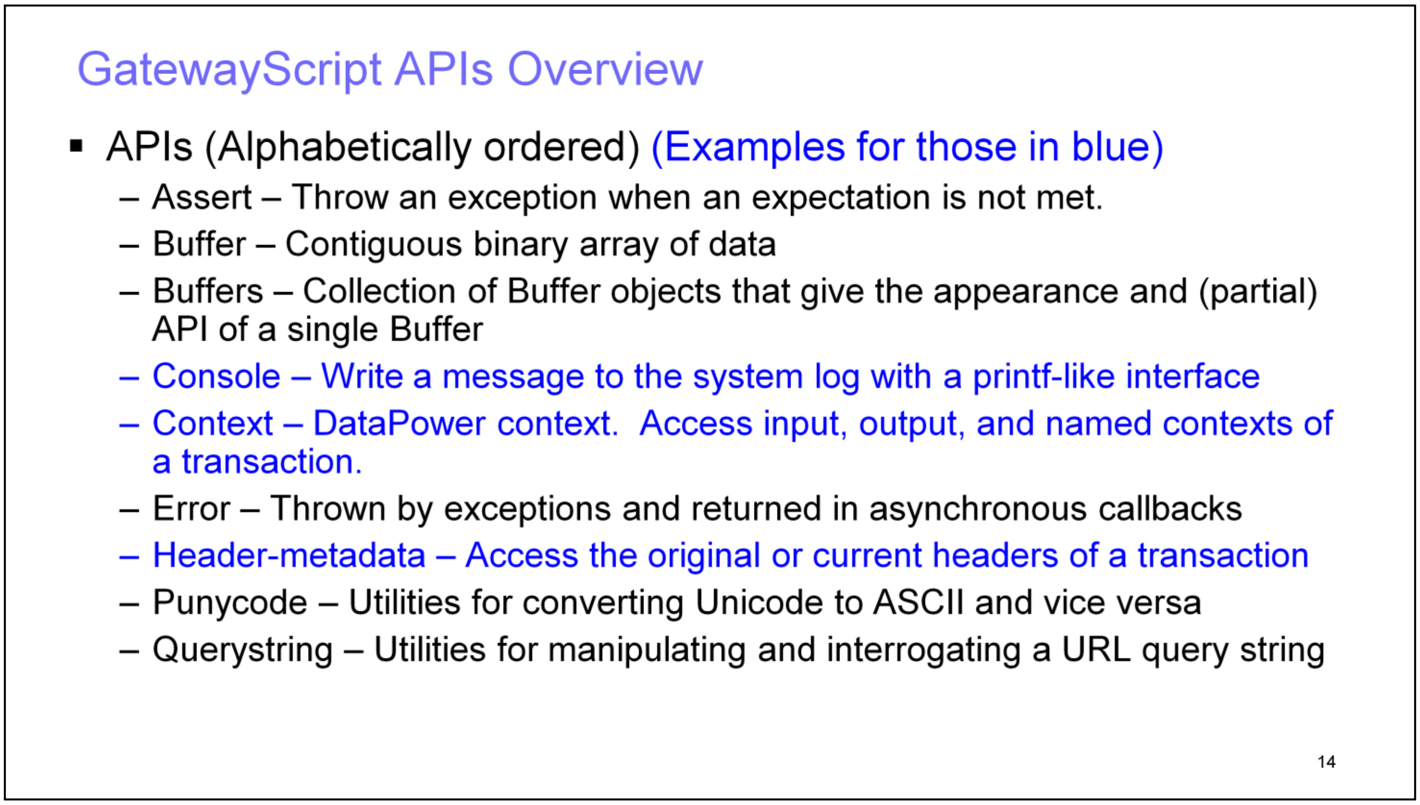
- List source code
  - list(l) [number of lines]
- Breakpoints
  - break (b) <line | script.js:line | function()>
  - delete (d) <identifier | all>
  - info break (ib)
- Print variable values
  - print (p) <variable>
- Explore stack trace
  - backtrace (bt)
- Program execution control
  - continue (c)
  - next (n) [count]
  - step (s) [count]
  - out (o) [count]
  - quit (q)

13

Here is the main set of commands available through the debugger.

You can list source code, set or delete breakpoints, print the contents of a variable, explore a stack trace, or step into, step out of, step over, continue or quit.

It is a small, but powerful set of capabilities for investigating your code.



GatewayScript APIs Overview

- APIs (Alphabetically ordered) (Examples for those in blue)
  - Assert – Throw an exception when an expectation is not met.
  - Buffer – Contiguous binary array of data
  - Buffers – Collection of Buffer objects that give the appearance and (partial) API of a single Buffer
  - Console – Write a message to the system log with a printf-like interface
  - Context – DataPower context. Access input, output, and named contexts of a transaction.
  - Error – Thrown by exceptions and returned in asynchronous callbacks
  - Header-metadata – Access the original or current headers of a transaction
  - Punycode – Utilities for converting Unicode to ASCII and vice versa
  - Querystring – Utilities for manipulating and interrogating a URL query string

14

The next few slides give an overview of the modules and APIs that are supported by GatewayScript. We cover the items in blue in more detail in this presentation.

The Assert module provides a tool for monitoring contracts or expected values. If you assert that a condition is true, but when the code runs, that assertion is violated, an exception is thrown.

The Buffer object is a continuous array of binary data. It can be used for interrogating and manipulating non-JSON data such as text or HTML.

The Buffers (with an ‘s’) object is a collection of Buffer objects. The Buffers API contains a subset of the Buffer API so that you can manipulate it as though it were a contiguous buffer.

The Console module provides access to the system log. It provides a printf-like interface where you pass a format string followed by a variable number of substitution arguments. You can log at any logging level that DataPower supports.

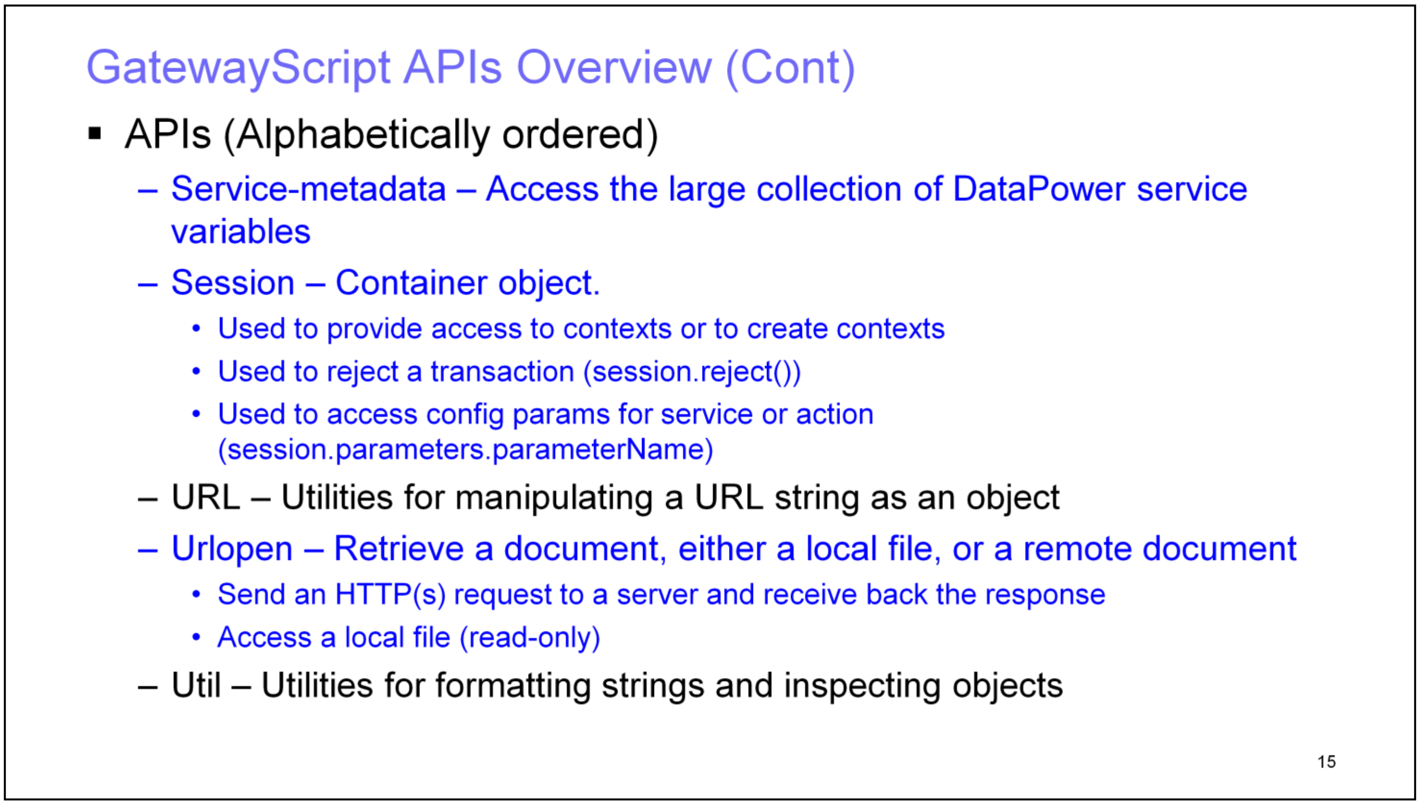
The Context object is a container that typically contains the payload of a request or response. Contexts are read with an asynchronous function call.

The Error object is thrown by exceptions and is also returned in an asynchronous callback if an error occurred. The error object contains a text string of the failure. It also contains stack information, a description of possible causes of the error, and a suggestion on how to remedy the situation.

The Header-metadata module is the API that a GatewayScript application uses to access the headers of a request or response. You can access the original headers, which are the headers that are received right off the wire. Or, you can access the current headers at that moment in the processing flow.

The Punycode module is a collection of utilities for converting Unicode to ASCII and vice versa.

The Querystring module is an API for manipulating and interrogating the query string of a URL.



GatewayScript APIs Overview (Cont)

- APIs (Alphabetically ordered)
  - Service-metadata – Access the large collection of DataPower service variables
  - Session – Container object.
    - Used to provide access to contexts or to create contexts
    - Used to reject a transaction (session.reject())
    - Used to access config params for service or action (session.parameters.parameterName)
  - URL – Utilities for manipulating a URL string as an object
  - Urlopen – Retrieve a document, either a local file, or a remote document
    - Send an HTTP(s) request to a server and receive back the response
    - Access a local file (read-only)
  - Util – Utilities for formatting strings and inspecting objects

15

The Service-metadata module is used to access the large collection of DataPower service variables. With few exceptions, the service variables that are available in XSLT are also available through GatewayScript. Service variables can be accessed either by URL or by a convenient dotted notation.

The Session object is a container that represents the transaction. You use the session object to access or create contexts, and reject transactions. Also, with the session object, you can read configuration parameters that are passed from the GatewayScript action configuration into the running GatewayScript application.

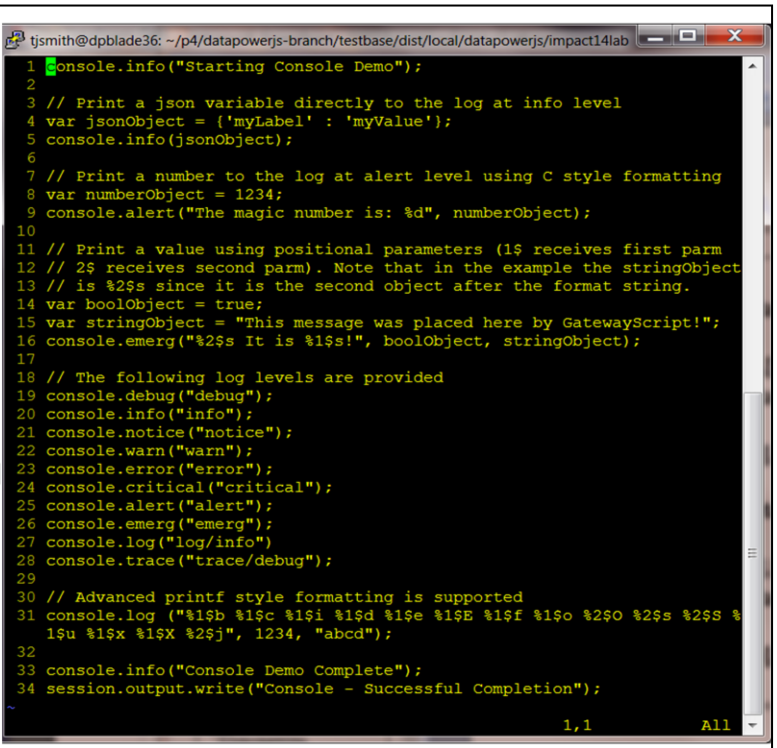
The URL module is a collection of utilities for processing a URL string.

With the Urlopen module, your script can send a request and receive a response for an external resource. Currently, only HTTP, HTTPS, and accessing files are allowed through the urlopen module. The only way to read a file from your local:/// or store:/// directories is through the urlopen module. Files that are read are cached in the document cache for future reads.

The Util module is a collection of utilities for formatting strings and inspecting objects.

GatewayScript API – Console

- **console.log([data],[…])** – same as console.info()
- **console.debug ([data],[…])** – log at debug level
- **console.info ([data],[…])** – log at info level
- **console.notice ([data],[…])** – log at notice level
- **console.warn ([data],[…])** – log at warning level
- **console.critical ([data],[…])** – log at critical level
- **console.alert ([data],[…])** – log at alert level
- **console.emerg ([data],[…])** – log at emergency level
- **console.trace ([data],[…])** – same as console.debug()



tjsmith@dpblade36: ~/p4/datapowerjs-branch/testbase/dist/local/datapowerjs/impact14lab

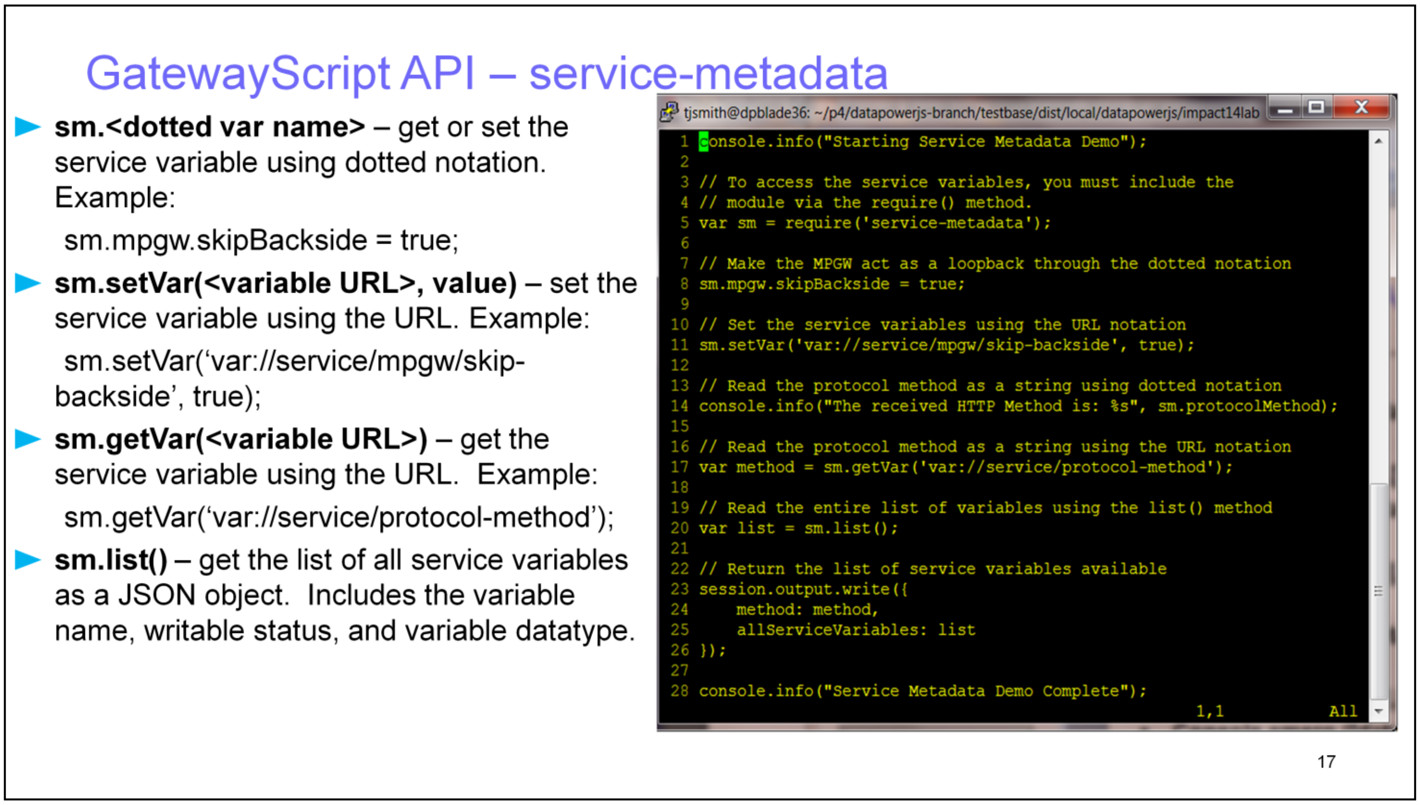
```
console.info("Starting Console Demo");

// Print a json variable directly to the log at info level
var jsonObject = {'myLabel' : 'myValue'};
console.info(jsonObject);

// Print a number to the log at alert level using C style formatting
var numberObject = 1234;
console.alert("The magic number is: %d", numberObject);

// Print a value using positional parameters (1$ receives first parm
// 2$ receives second parm). Note that in the example the stringObject
// is %2$s since it is the second object after the format string.
var boolObject = true;
var stringObject = "This message was placed here by GatewayScript!";
console.emerg("%2$s It is %1$s!", boolObject, stringObject);

// The following log levels are provided
console.debug("debug");
console.info("info");
console.notice("notice");
console.warn("warn");
console.error("error");
console.critical("critical");
console.alert("alert");
console.emerg("emerg");
console.log("log/info")
console.trace("trace/debug");

// Advanced printf style formatting is supported
console.log ("%1$b %1$c %1$i %1$d %1$e %1$E %1$f %1$o %2$O %2$s %2$S %
1$u %1$x %1$X %2$j", 1234, "abcd");

console.info("Console Demo Complete");
session.output.write("Console - Successful Completion");
~
                                                    1,1           All
```

16

The Console API is the mechanism of writing to the DataPower System logs. No require statement is required to have access to this API. You can use numerous methods to denote the log record's severity, for example, console.debug() is used for debug log messages. Also, the log and trace methods are equivalent to the info and debug methods. These methods are provided to maintain compatibility with the console API.

The data to be logged can contain C formatting specifiers. Positional parameters are also supported to format the same parameter in different ways. Positional parameters can be used to have the substitution in the log text string in a different order than the values presented. The following formatting specifiers are supported:

%b binary	%c unsigned char
%i unsigned decimal	%d unsigned decimal
%e scientific notation	%E scientific notation uppercase
%f floating point	%o unsigned octal
%O object inspect	%s string - toString()
%S string uppercase	%u unsigned decimal
%x unsigned hexadecimal	%X unsigned hexadecimal uppercase
%j JSON.stringify	



GatewayScript API – service-metadata

- **sm.<dotted var name>** – get or set the service variable using dotted notation. Example:

  sm.mpgw.skipBackside = true;

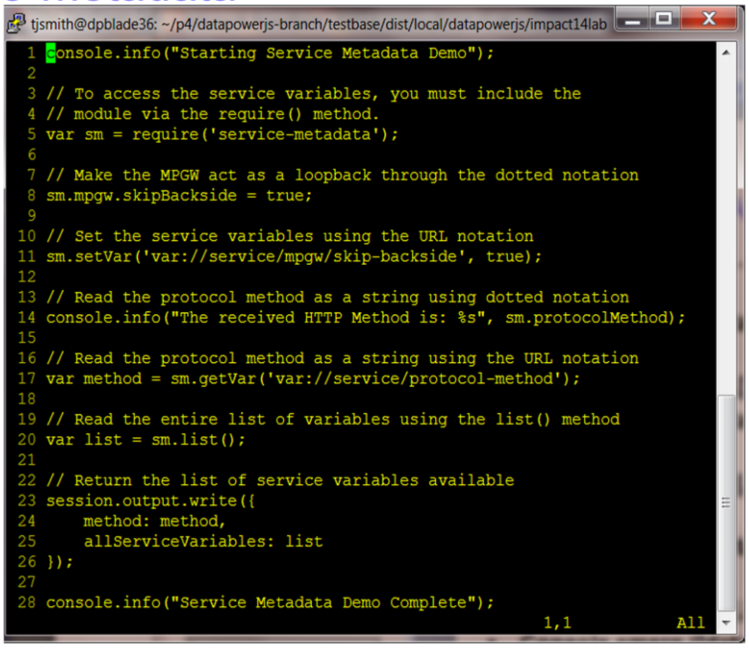
- **sm.setVar(<variable URL>, value)** – set the service variable using the URL. Example:

  sm.setVar('var://service/mpgw/skip-backside', true);

- **sm.getVar(<variable URL>)** – get the service variable using the URL. Example:

  sm.getVar('var://service/protocol-method');

- **sm.list()** – get the list of all service variables as a JSON object. Includes the variable name, writable status, and variable datatype.


tjsmith@dpblade36: ~/p4/datapowerjs-branch/testbase/dist/local/datapowerjs/impact14lab

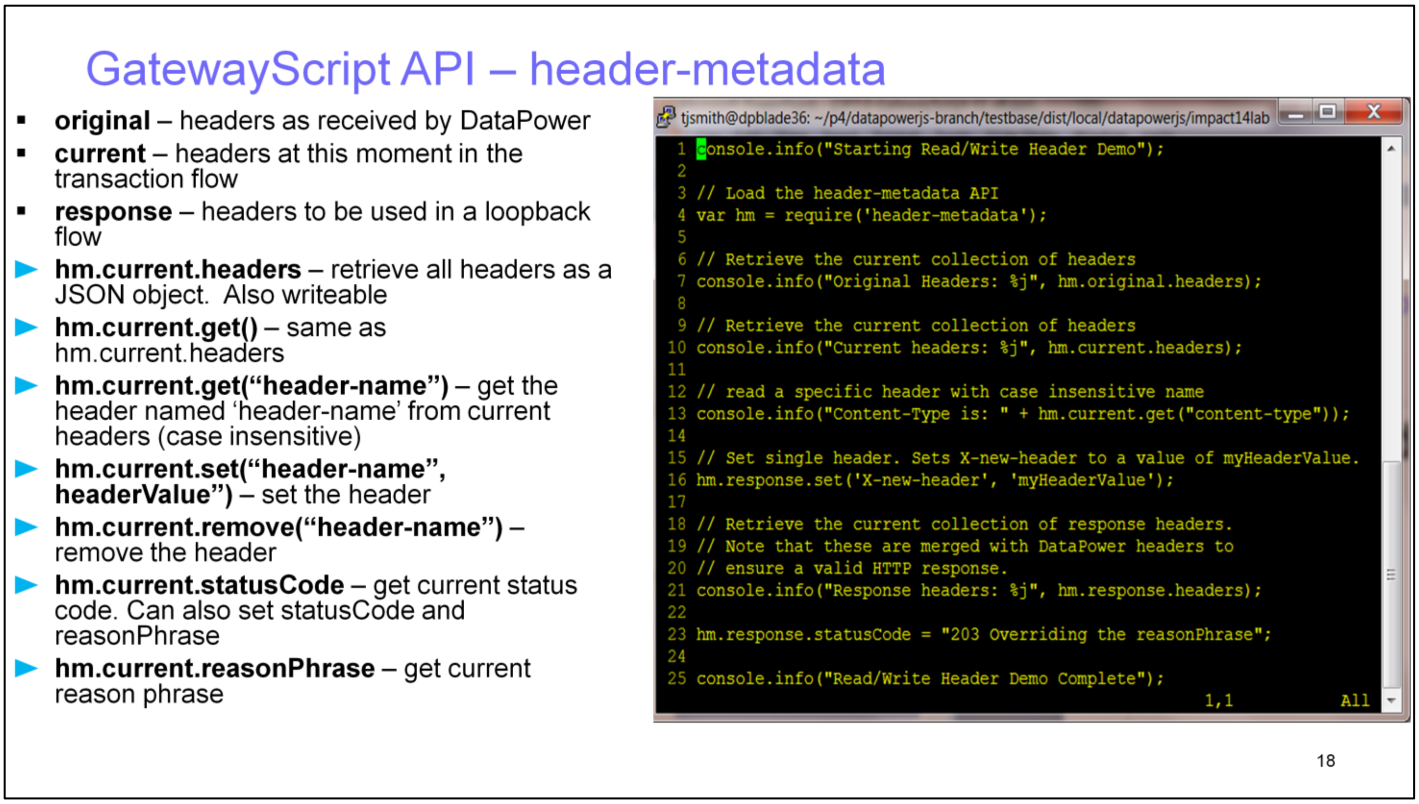
```
console.info("Starting Service Metadata Demo");

// To access the service variables, you must include the
// module via the require() method.
var sm = require('service-metadata');

// Make the MPGW act as a loopback through the dotted notation
sm.mpgw.skipBackside = true;

// Set the service variables using the URL notation
sm.setVar('var://service/mpgw/skip-backside', true);

// Read the protocol method as a string using dotted notation
console.info("The received HTTP Method is: %s", sm.protocolMethod);

// Read the protocol method as a string using the URL notation
var method = sm.getVar('var://service/protocol-method');

// Read the entire list of variables using the list() method
var list = sm.list();

// Return the list of service variables available
session.output.write({
    method: method,
    allServiceVariables: list
});

console.info("Service Metadata Demo Complete");
                                                          1,1           All
```

17

The Service Metadata API provides access to DataPower service variables from GatewayScript. This API is made available by using the var sm = require('service-metadata'); statement in the GatewayScript code. Service variables can be accessed with two notations. First, a URL notation such as what is used by XSLT style sheets and is used by the getVar and setVar methods. The URL notation syntax would be of the form var://service/variablename, for example, var://service/mpgw/skip-backside. Finally, the dotted notation is used directly from the service metadata variable and uses lower camel case notation. To translate a URL notation variable to a dotted notation variable, replace any "/" (slash) in the URL format with a "." period. If a "-" (hyphen) is present in the URL format, remove the hyphen and change the following character to uppercase. For example, the equivalent to var://service/mpgw/skip-backside is sm.mpgw.skipBackside.

Setting a service variable in GatewayScript enforces the data type of the desired value for some variables and throws an exception if an incorrect type is used. Specifically, the number data type, for example a timeout variable, is a number, not a string. Also, the boolean data type, for example, mpgw.skipBackside is set to true, not a 1 as in XSLT.

Also, a list method is available that shows all supported service variables in a JSON format. This format includes the variable name, writable attribute, and data type.



GatewayScript API – header-metadata

- **original** – headers as received by DataPower
- **current** – headers at this moment in the transaction flow
- **response** – headers to be used in a loopback flow

- **hm.current.headers** – retrieve all headers as a JSON object. Also writeable
- **hm.current.get()** – same as hm.current.headers
- **hm.current.get("header-name")** – get the header named 'header-name' from current headers (case insensitive)
- **hm.current.set("header-name", headerValue")** – set the header
- **hm.current.remove("header-name")** – remove the header
- **hm.current.statusCode** – get current status code. Can also set statusCode and reasonPhrase
- **hm.current.reasonPhrase** – get current reason phrase

tjsmith@dpblade36: ~/p4/datapowerjs-branch/testbase/dist/local/datapowerjs/impact14lab

```
console.info("Starting Read/Write Header Demo");

// Load the header-metadata API
var hm = require('header-metadata');

// Retrieve the current collection of headers
console.info("Original Headers: %j", hm.original.headers);

// Retrieve the current collection of headers
console.info("Current headers: %j", hm.current.headers);

// read a specific header with case insensitive name
console.info("Content-Type is: " + hm.current.get("content-type"));

// Set single header. Sets X-new-header to a value of myHeaderValue.
hm.response.set('X-new-header', 'myHeaderValue');

// Retrieve the current collection of response headers.
// Note that these are merged with DataPower headers to
// ensure a valid HTTP response.
console.info("Response headers: %j", hm.response.headers);

hm.response.statusCode = "203 Overriding the reasonPhrase";

console.info("Read/Write Header Demo Complete");
                                                    1,1           All
```

The Header Metadata API provides access to HTTP protocol headers from GatewayScript. This API is made available by using the var hm = require('header-metadata'); statement in the GatewayScript code. For each rule type, specifically the request, response, error rules, there are two header types. First, the original header type represents what comes into the rule on the wire and is read only. The current header type represents the current state of the headers for the rule. It includes any DataPower injected headers, such as X-Client-IP, and any headers that are programmatically added, removed, or modified by the processing rule. The third header type is the response header. It is equivalent to the current header in a response or error rule. The response header gives you access to the response header from the request rule. It was specifically added for skipBackside or loopback firewall scenarios that do not have response rule processing, although it is not restricted to this use case.

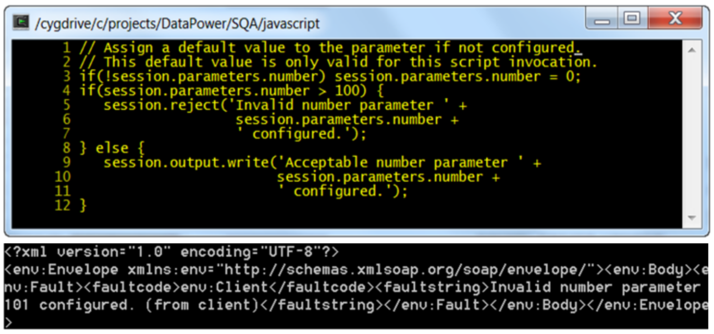
You can get all original and current headers as a JSON object, and can set all current headers with a JSON object. You can also get and set individual headers. Headers might be coalesced, a string with comma-separated values, or non-coalesced, an array with separate elements for each value. Setting a header as a coalesced or non-coalesced header is controlled by the script. On the wire, headers like set-cookie are presented as non-coalesced, which on the wire are unique headers of the same name with different values.

The header API gets and sets the backend protocol status code and reason phrase. To get or set these values in a request rule, use hm.response. The status code and reason phrase are set with the hm.response.statusCode variable. If only a status code is provided, a default reason phrase is assumed based on the HTTP specification. For example, a statusCode of 402 assumes the reason phrase of "Payment Required".

Note that hm.response is not the same as x-dp-response-code header that is used by XSL style sheets for a similar function. The x-dp-response-code header is not supported by GatewayScript.

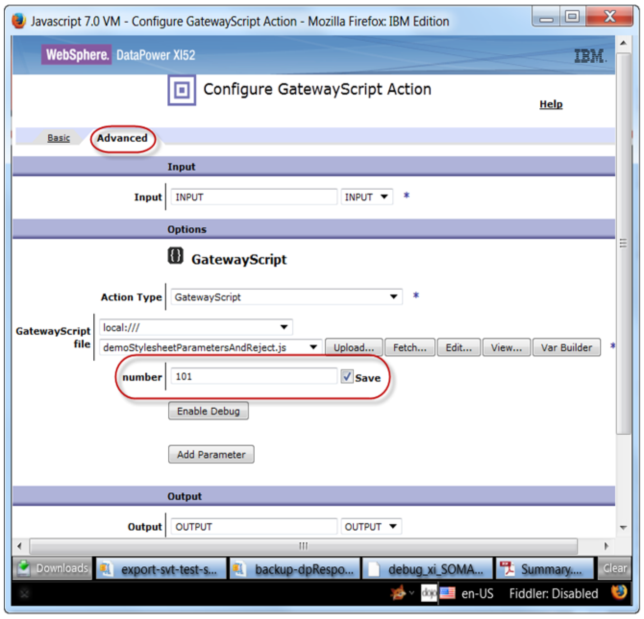
GatewayScript API – Session “Stylesheet” Parameters and Reject

- “Stylesheet” Parameters – session.parameters.name
  - Defined at service or action level
  - Action level overrides service level if both configured
  - Service level may be overridden by var://service/config-param/ service variable
- Reject – session.reject(“optional reject message”)
  - Processing rule error set after the action completes.
  - Use logic to determine if you need this behavior
    - No “Accept” API to change this behavior once set


/cygdrive/c/projects/DataPower/SQA/javascript

```
// Assign a default value to the parameter if not configured.
// This default value is only valid for this script invocation.
if(!session.parameters.number) session.parameters.number = 0;
if(session.parameters.number > 100) {
   session.reject('Invalid number parameter ' +
                  session.parameters.number +
                  ' configured.');
} else {
   session.output.write('Acceptable number parameter ' +
                        session.parameters.number +
                        ' configured.');
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"><env:Body><e
nv:Fault><faultcode>env:Client</faultcode><faultstring>Invalid number parameter
101 configured. (from client)</faultstring></env:Fault></env:Body></env:Envelope
>
```


Javascript 7.0 VM - Configure GatewayScript Action - Mozilla Firefox: IBM Edition

Configure GatewayScript Action

Basic | Advanced

Input
Input: INPUT | INPUT *

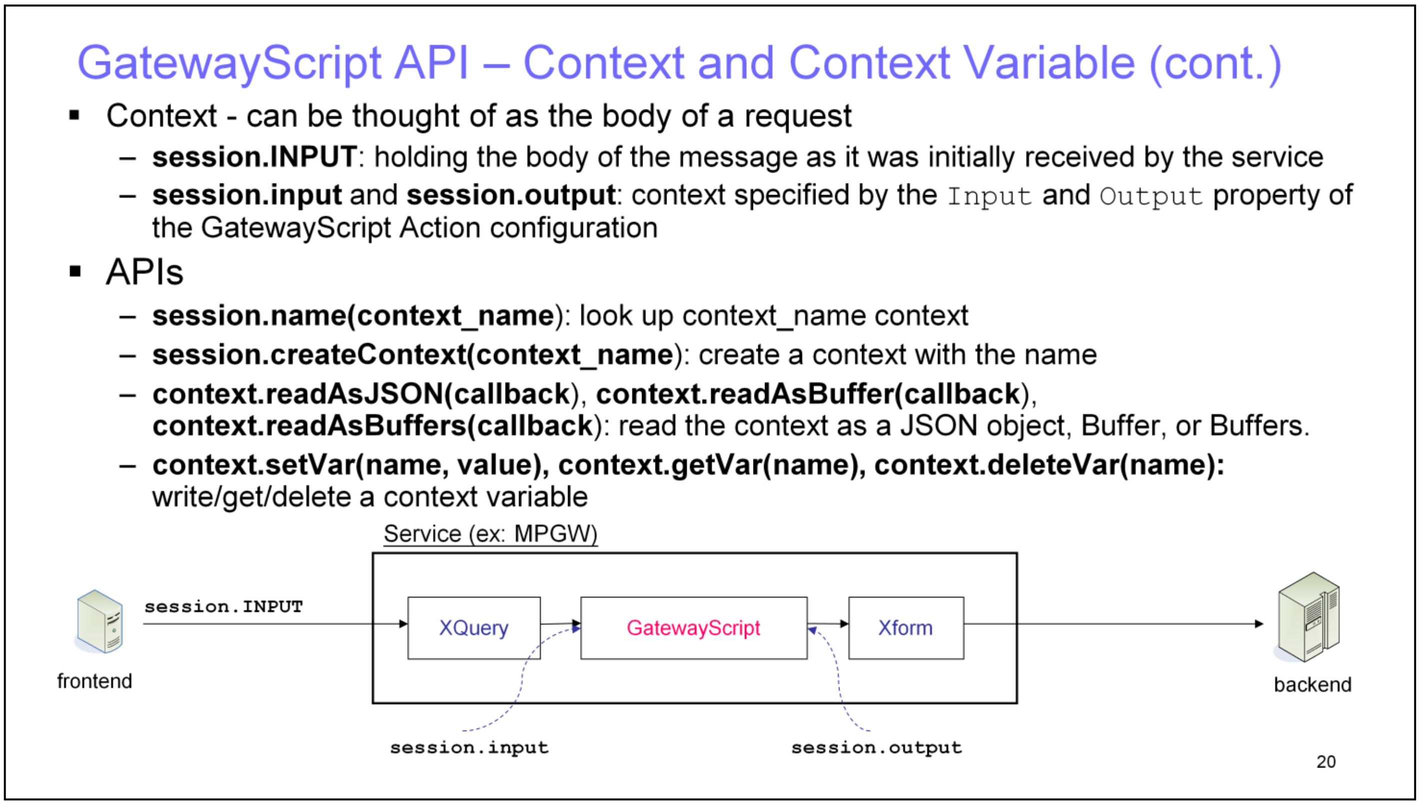
Options
GatewayScript
Action Type: GatewayScript *
GatewayScript file: local:/// | demoStylesheetParametersAndReject.js *
number: 101 [✓ Save]

Output
Output: OUTPUT | OUTPUT

19

Parameters are useful to externalize literals that the code requires for the action or service, which makes the code portable between various lifecycle environments. For example, a target URL for the urlopen API might reference a development server or a production server, and the parameter can be modified during deployment from one environment to another without touching the code. Session Parameters can be specified at a service or action level. These parameters are available on the session object. The action level configuration overrides the service level if both are present. Also, the service level might be changed by a service variable. Setting session.parameter.varname is only for this invocation of the script. However, modifying the parameter within the script does provide a mechanism to provide a default value if one was not configured

The session object also has a reject method. This method acts like the dp:reject extension function in XSL. The script continues to completion and causes an error for the action in the processing policy. If not overridden by an on-error action, it drives error rule processing. If an on-error action is configured before the GatewayScript action, the rule processes the error based on the on-error action's configuration. Unlike XSL, there is no mechanism to change your mind as there was with the dp:accept extension function. Instead, keep a mutable variable with a boolean state and conditionally do a reject if required. The method also takes an optional argument of text that shows up in default error rule response, in error related service variable if needed in custom error processing, and in the system logs.



GatewayScript API – Context and Context Variable (cont.)

- Context - can be thought of as the body of a request
  - **session.INPUT**: holding the body of the message as it was initially received by the service
  - **session.input** and **session.output**: context specified by the `Input` and `Output` property of the GatewayScript Action configuration
- APIs
  - **session.name(context_name)**: look up context_name context
  - **session.createContext(context_name)**: create a context with the name
  - **context.readAsJSON(callback)**, **context.readAsBuffer(callback)**, **context.readAsBuffers(callback)**: read the context as a JSON object, Buffer, or Buffers.
  - **context.setVar(name, value), context.getVar(name), context.deleteVar(name):** write/get/delete a context variable

20

GatewayScript provides APIs to access context and context variables.

Context can be thought of as the body of a request.

The following diagram illustrates several different contexts.

To access the original body of the message that is sent to a DataPower service, you can use session.INPUT.

To access the Input of a GatewayScript action, you can use session.input, and to access the Output of a GatewayScript action, you can use session.output.

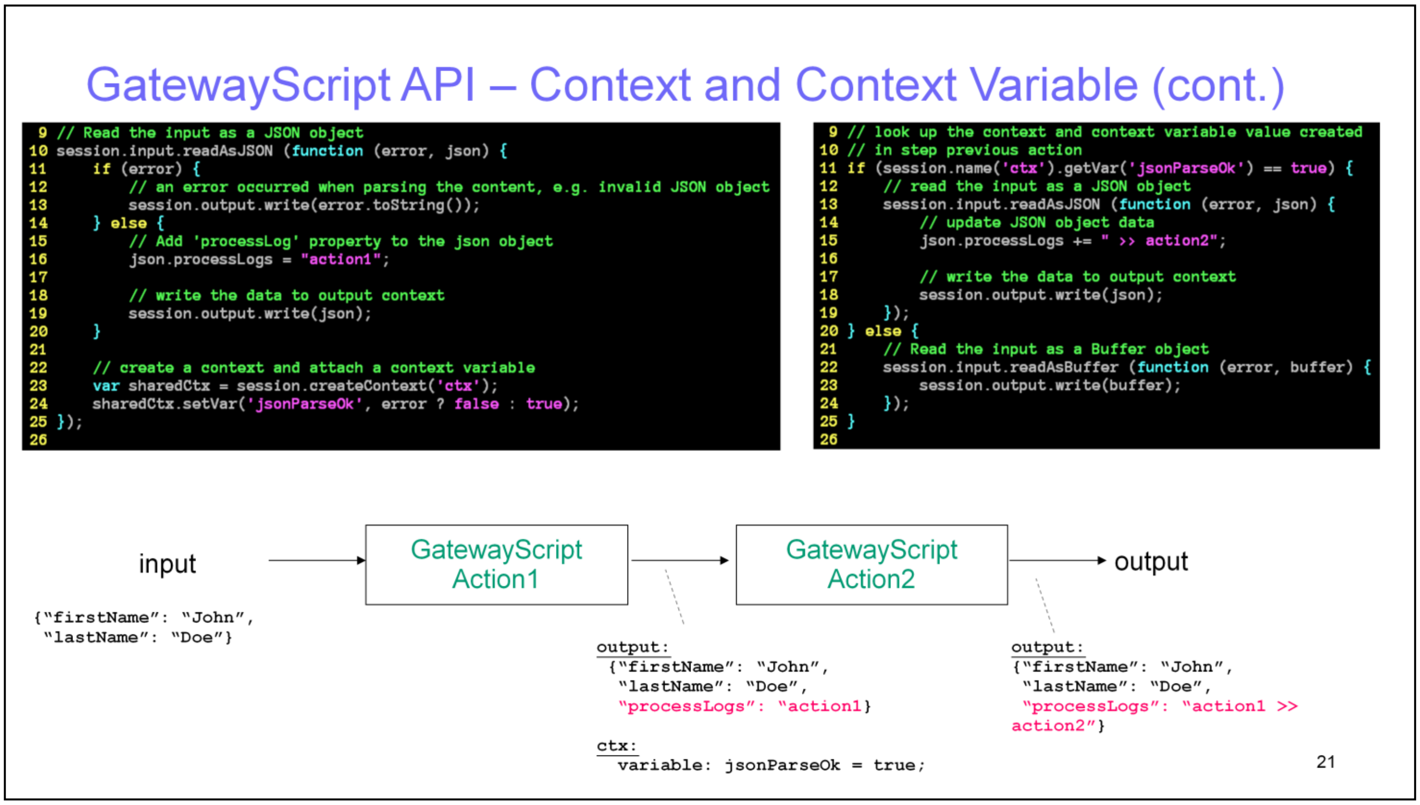
The following contexts and context variable-related APIs are supported:

- session.name(context_name): look up the context with the ‘context_name’ name

- session.createContext(context_name): create a context with the name

- context.readAsJSON, context.readAsBuffer, context.readAsBuffers: read the body of the context as a JSON object, Buffer object, or Buffers object

- context.setVar(), context.getVar(), context.deleteVar(): write, get, and delete a context variable from the context



GatewayScript API – Context and Context Variable (cont.)

```
 9 // Read the input as a JSON object
10 session.input.readAsJSON (function (error, json) {
11     if (error) {
12         // an error occurred when parsing the content, e.g. invalid JSON object
13         session.output.write(error.toString());
14     } else {
15         // Add 'processLog' property to the json object
16         json.processLogs = "action1";
17
18         // write the data to output context
19         session.output.write(json);
20     }
21
22     // create a context and attach a context variable
23     var sharedCtx = session.createContext('ctx');
24     sharedCtx.setVar('jsonParseOk', error ? false : true);
25 });
26
```

```
 9 // look up the context and context variable value created
10 // in step previous action
11 if (session.name('ctx').getVar('jsonParseOk') == true) {
12     // read the input as a JSON object
13     session.input.readAsJSON (function (error, json) {
14         // update JSON object data
15         json.processLogs += " >> action2";
16
17         // write the data to output context
18         session.output.write(json);
19     });
20 } else {
21     // Read the input as a Buffer object
22     session.input.readAsBuffer (function (error, buffer) {
23         session.output.write(buffer);
24     });
25 }
26
```

input → GatewayScript Action1 → GatewayScript Action2 → output

```
{"firstName": "John",
 "lastName": "Doe"}
```

```
output:
 {"firstName": "John",
  "lastName": "Doe",
  "processLogs": "action1}

ctx:
  variable: jsonParseOk = true;
```

```
output:
{"firstName": "John",
 "lastName": "Doe",
 "processLogs": "action1 >>
action2"}
```

21

This slide shows an example that illustrates how to use the GatewayScript context and context variable APIs.

In the following diagram, a service is configured with two GatewayScript actions. The source code that is associated with the two actions is shown at the top of the slide.

Look at the code of the first action -

- On line #10, it reads data from the session.input context and tries to parse the read-in data as a JSON object. In this example, because the input is a valid JSON string ({firstName: “John”, lastName: “Doe”}), the readAsJSON() API successfully parses the JSON and stores the parsed result into the ‘json’ variable.

- On line #16, the code adds ‘processLog’ property to the json object and sets the value to “action1”. In the diagram, the intermediary output from the first GatewayScript action contains this new property, “processLogs”: “action1”.

- On line #23, the code creates a new context that is named ‘ctx’ and stores a ‘jsonParseOk’ context variable that indicates whether the first action successfully parsed the JSON data (line #24). This context variable is examined by the second GatewayScript action that is explained next.

Look at the code of the second action -

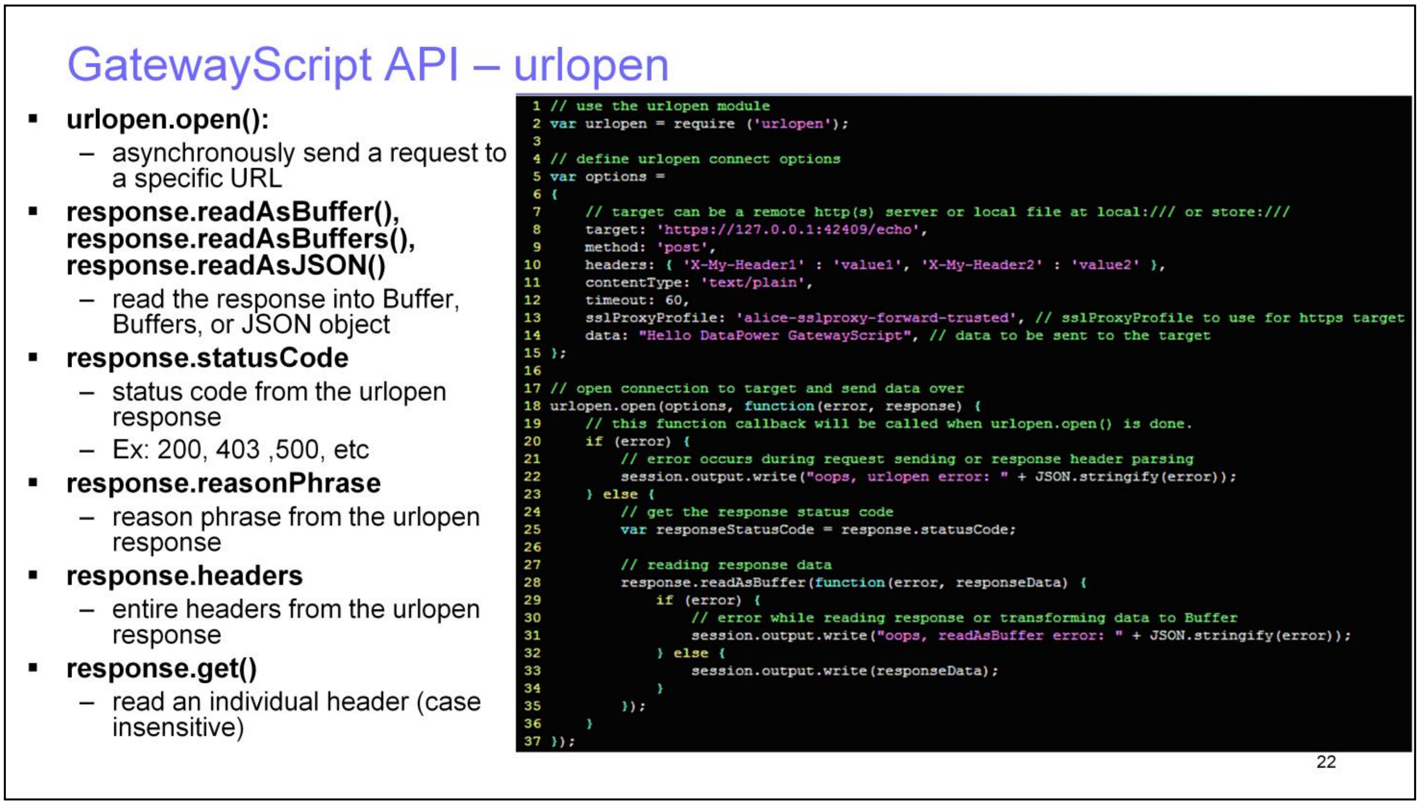
- On line #11, the code locates the ‘ctx’ context and examines the value of ‘jsonParseOk’ context variable.

- If the context variable contains ‘true’, it assumes that the previous action successfully parsed the JSON. Then, on line #13, the code tries to read the input as a JSON object and amends “ >> action 2” to the processLogs property on line #15.

- Finally, on line #18, the processed JSON object is serialized to the output.

- In the diagram, you can see that the final output from the GatewayScript action contains the “processLogs”: “action1 >> action2” property.

Through this example, you can learn how to read and write data from a context and also how to use context variables to exchange data between different actions.



GatewayScript API – urlopen

- **urlopen.open():**
  - asynchronously send a request to a specific URL
- **response.readAsBuffer(), response.readAsBuffers(), response.readAsJSON()**
  - read the response into Buffer, Buffers, or JSON object
- **response.statusCode**
  - status code from the urlopen response
  - Ex: 200, 403 ,500, etc
- **response.reasonPhrase**
  - reason phrase from the urlopen response
- **response.headers**
  - entire headers from the urlopen response
- **response.get()**
  - read an individual header (case insensitive)

```
 1 // use the urlopen module
 2 var urlopen = require ('urlopen');
 3
 4 // define urlopen connect options
 5 var options =
 6 {
 7     // target can be a remote http(s) server or local file at local:/// or store:///
 8     target: 'https://127.0.0.1:42409/echo',
 9     method: 'post',
10     headers: { 'X-My-Header1' : 'value1', 'X-My-Header2' : 'value2' },
11     contentType: 'text/plain',
12     timeout: 60,
13     sslProxyProfile: 'alice-sslproxy-forward-trusted', // sslProxyProfile to use for https target
14     data: "Hello DataPower GatewayScript", // data to be sent to the target
15 };
16
17 // open connection to target and send data over
18 urlopen.open(options, function(error, response) {
19     // this function callback will be called when urlopen.open() is done.
20     if (error) {
21         // error occurs during request sending or response header parsing
22         session.output.write("oops, urlopen error: " + JSON.stringify(error));
23     } else {
24         // get the response status code
25         var responseStatusCode = response.statusCode;
26
27         // reading response data
28         response.readAsBuffer(function(error, responseData) {
29             if (error) {
30                 // error while reading response or transforming data to Buffer
31                 session.output.write("oops, readAsBuffer error: " + JSON.stringify(error));
32             } else {
33                 session.output.write(responseData);
34             }
35         });
36     }
37 });
```

22

GatewayScript provides the urlopen API so that you can send and read data from a URL target. Currently, HTTP, HTTPS, and local file access are supported.

The example code that shown on the right illustrates the urlopen API usage.

- To use urlopen API, the first thing to do is to require the ‘urlopen’ module, as shown on line #2.

- Lines #4 through #15 define a JSON object that contains the following urlopen options:

  - line#8: The urlopen connect target

  - line#9: The method that is used for the urlopen. In HTTP and HTTPS, the standard HTTP methods (for example GET, POST) are supported. The method is case insensitive.

  - line#10: Defines headers that are sent to the urlopen target

  - line#11: Defines the content type of the payload that is sent to the urlopen target

  - line#12: urlopen connection timeout in seconds

  - line#13: If the target is HTTPS, then the sslProxyProfile is used for the SSL handshaking with urlopen target

  - line#14: The data that is sent to the target. The data can be a string or any other JavaScript objects, for example, Buffer.

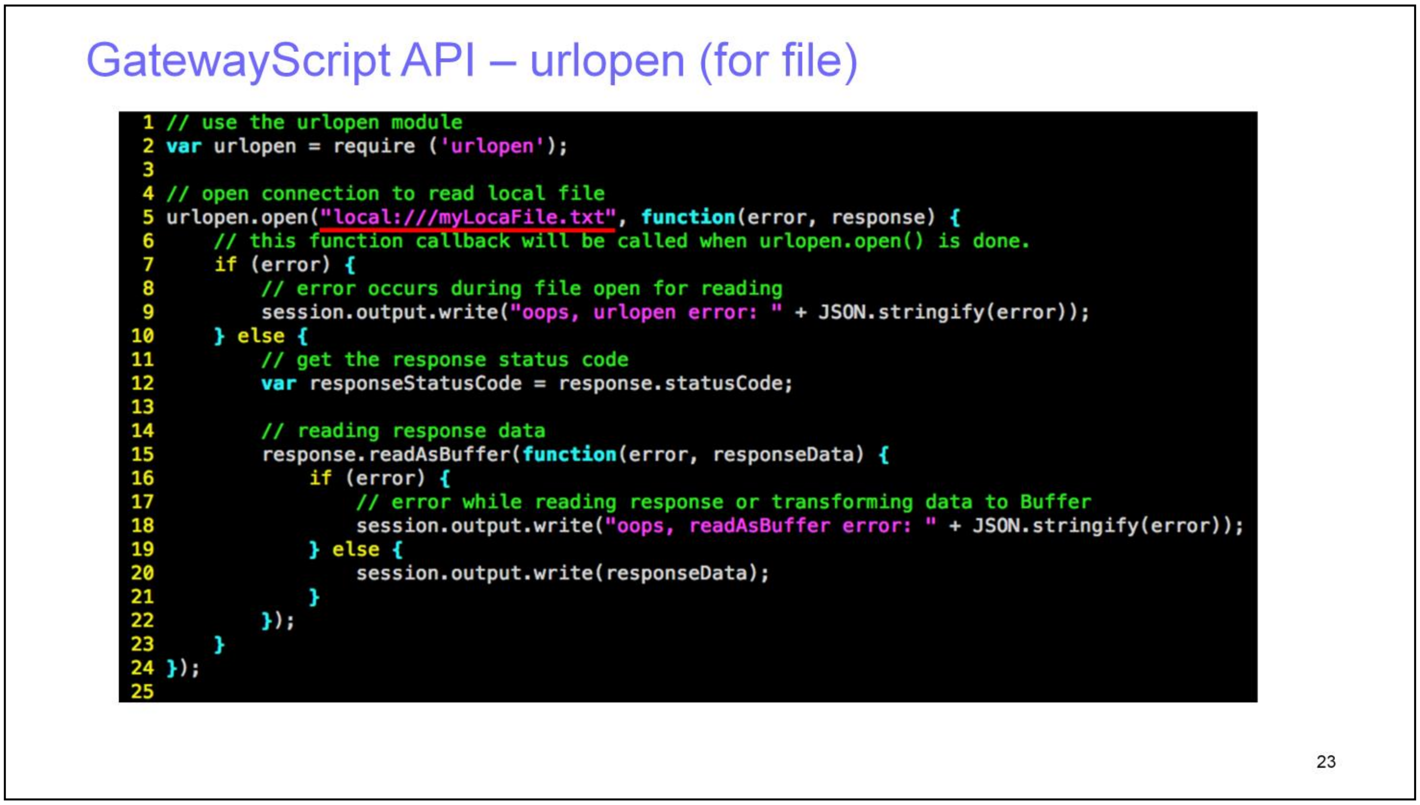
- urlopen.open() attempts to do the urlopen according to the options defined in the urlopen. After urlopen connects to the target (or fails to connect to the target), on line #18, the function(error, response) callback is called.

- line #20: If an error occurs during the urlopen connection, the error object is not NULL and contains the error detail

- line #25: If the urlopen target was successfully connected and data was sent, then you can use response.statusCode to learn the response status from the target server. For example, if the target is an HTTP server, then a 200 statusCode indicates that the server successfully processed the request.

- line #28: Use response.readAsBuffer() (or readAsBuffers(), readAsJSON()) to retrieve the response data from the target server.

Though not illustrated in this example, you can also use response.headers() and response.get() (see left side APIs) to retrieve the header data that is returned from the urlopen target.

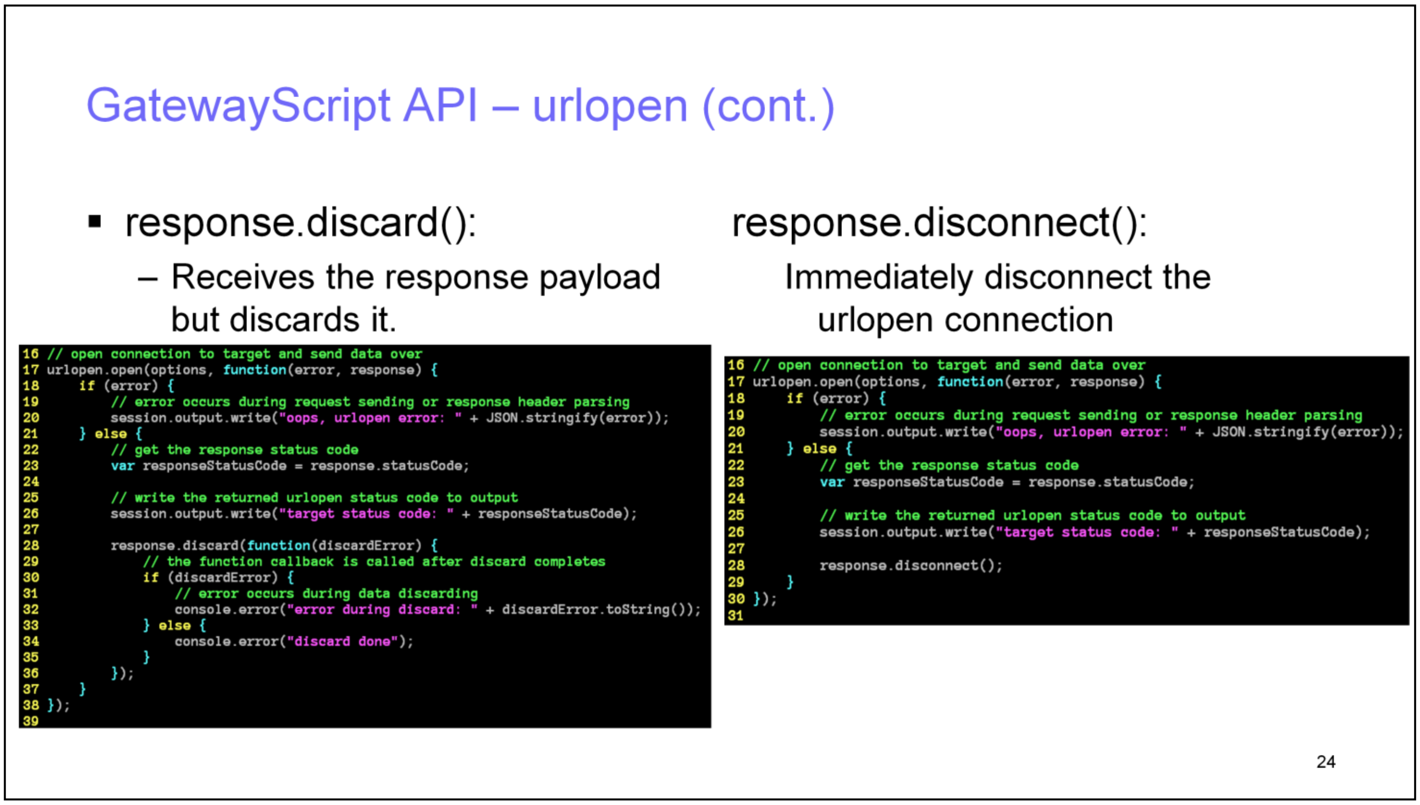


GatewayScript API – urlopen (for file)

```
 1 // use the urlopen module
 2 var urlopen = require ('urlopen');
 3
 4 // open connection to read local file
 5 urlopen.open("local:///myLocaFile.txt", function(error, response) {
 6     // this function callback will be called when urlopen.open() is done.
 7     if (error) {
 8         // error occurs during file open for reading
 9         session.output.write("oops, urlopen error: " + JSON.stringify(error));
10     } else {
11         // get the response status code
12         var responseStatusCode = response.statusCode;
13
14         // reading response data
15         response.readAsBuffer(function(error, responseData) {
16             if (error) {
17                 // error while reading response or transforming data to Buffer
18                 session.output.write("oops, readAsBuffer error: " + JSON.stringify(error));
19             } else {
20                 session.output.write(responseData);
21             }
22         });
23     }
24 });
25
```

This urlopen example illustrates how to use the API to retrieve the contents of a local file.

All the code logic and flows are similar to what was illustrated in the previous slide, except line #5. Instead of using a JSON object to define the urlopen options, you can use a simple string as a parameter in the urlopen.open() api. The string is treated as the urlopen target name.



GatewayScript API – urlopen (cont.)

- response.discard():
  - Receives the response payload but discards it.

```
16 // open connection to target and send data over
17 urlopen.open(options, function(error, response) {
18     if (error) {
19         // error occurs during request sending or response header parsing
20         session.output.write("oops, urlopen error: " + JSON.stringify(error));
21     } else {
22         // get the response status code
23         var responseStatusCode = response.statusCode;
24
25         // write the returned urlopen status code to output
26         session.output.write("target status code: " + responseStatusCode);
27
28         response.discard(function(discardError) {
29             // the function callback is called after discard completes
30             if (discardError) {
31                 // error occurs during data discarding
32                 console.error("error during discard: " + discardError.toString());
33             } else {
34                 console.error("discard done");
35             }
36         });
37     }
38 });
39
```

response.disconnect():

Immediately disconnect the urlopen connection

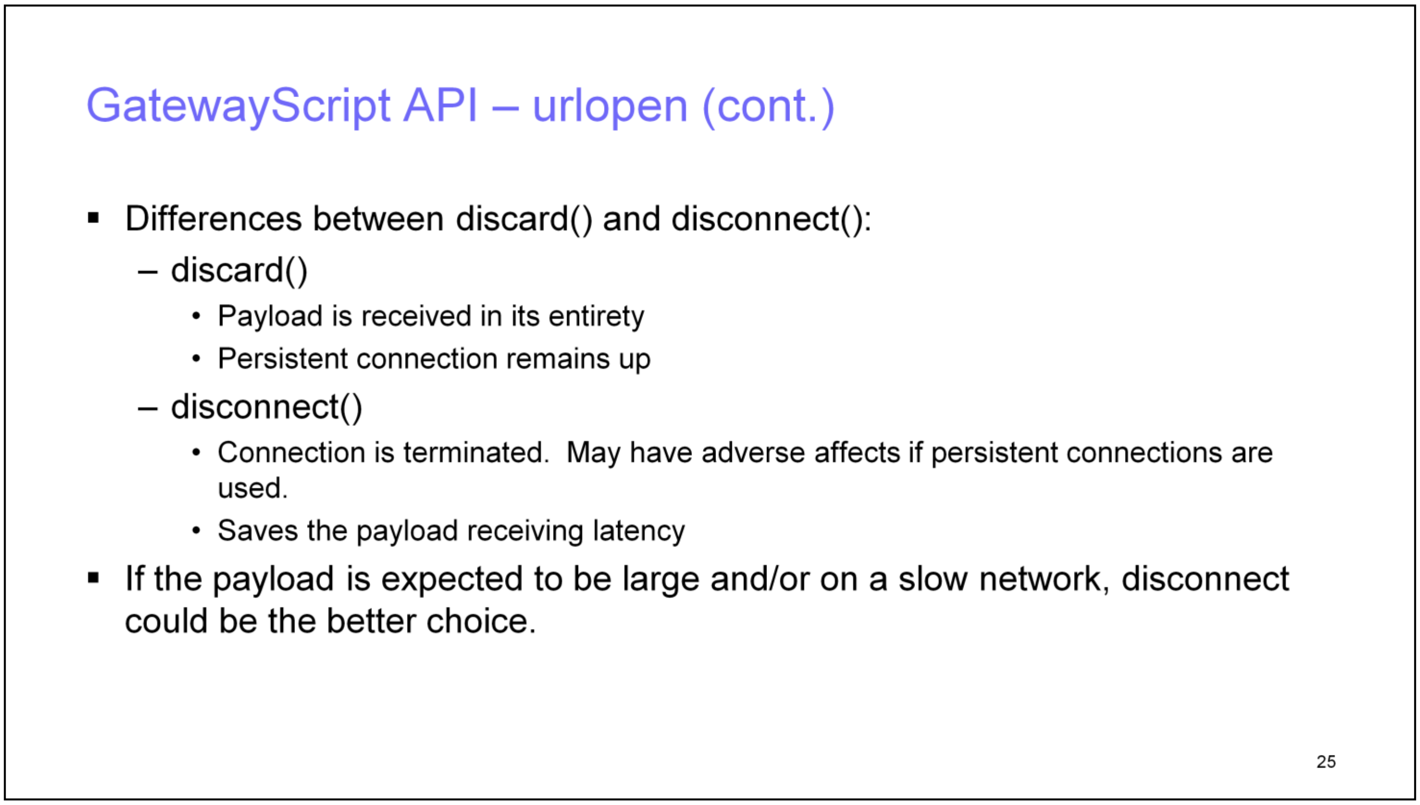
```
16 // open connection to target and send data over
17 urlopen.open(options, function(error, response) {
18     if (error) {
19         // error occurs during request sending or response header parsing
20         session.output.write("oops, urlopen error: " + JSON.stringify(error));
21     } else {
22         // get the response status code
23         var responseStatusCode = response.statusCode;
24
25         // write the returned urlopen status code to output
26         session.output.write("target status code: " + responseStatusCode);
27
28         response.disconnect();
29     }
30 });
31
```

24

Sometimes, you might want to check whether a urlopen target is alive and but you do not want to retrieve the data from the target. Under that situation, you can use response.discard() and response.disconnect().

response.discard() receives the response payload from the urlopen target server but discards the received data. Use urlopen.open() to connect to the target (line #17) and response.status to check the server response (line #23). Then, instead of using response.readAsBuffer, readAsBuffers, or readAsJSON, use response.discard() to discard the data (lines #28 - #36). When the discard is done, the callback that is registered to the response.discard() is invoked. Then, you can use the discardError to check whether there were any errors during the data discard (line #30).

response.disconnect() immediately disconnects the urlopen connection. See line#28 in the example code on the right. This code shows a synchronous call where there is no callback mechanism provided.



GatewayScript API – urlopen (cont.)

- Differences between discard() and disconnect():
  - discard()
    - Payload is received in its entirety
    - Persistent connection remains up
  - disconnect()
    - Connection is terminated. May have adverse affects if persistent connections are used.
    - Saves the payload receiving latency
- If the payload is expected to be large and/or on a slow network, disconnect could be the better choice.

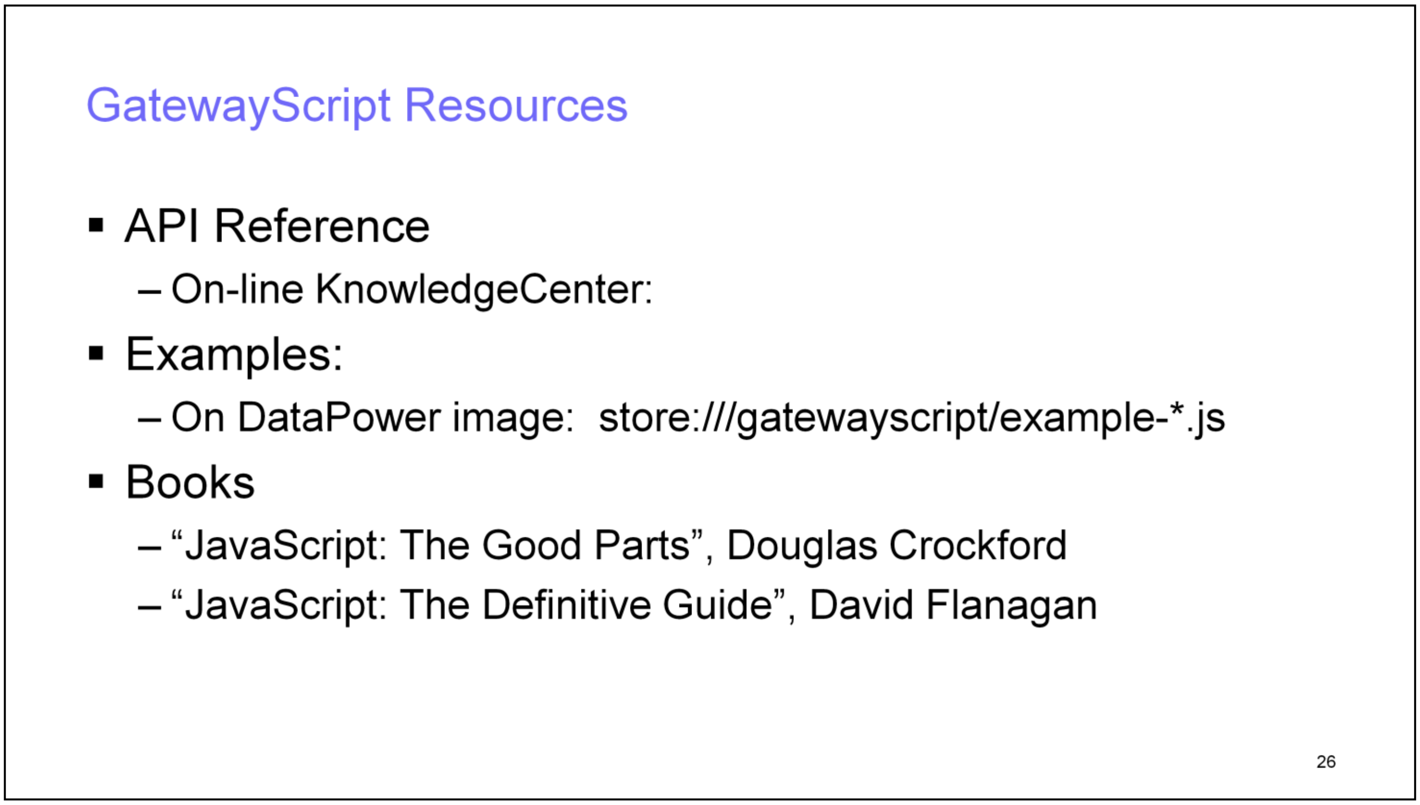
25

The differences between response.discard() and response.disconnect() are shown here.

When you use discard(), the full server response data is received by the appliance. If the underlying connection is a persistent connection, that connection is kept alive.

And when you use disconnect(), the server response is not received because the connection is immediately terminated. If the underlying connection is a persistent connection, that connection is dropped. The connection is no longer reusable.

Make an educated decision about whether to use discard() or disconnect(). If the payload is large and or the network is slow, disconnect() might be a better choice, and vice versa.



GatewayScript Resources

- API Reference
  - On-line KnowledgeCenter:
- Examples:
  - On DataPower image: store:///gatewayscript/example-*.js
- Books
  - “JavaScript: The Good Parts”, Douglas Crockford
  - “JavaScript: The Definitive Guide”, David Flanagan

26

For API information, consult the API Reference in IBM Knowledge Center. This reference gives the description of the APIs along with examples of how to use them.

The DataPower firmware contains examples of how to use various APIs in GatewayScript. You can cut and paste these examples into your code. They are in the store:///gatewayscript/ directory on the appliance with file names of “example-*.js” where ‘*’ is a wildcard.

Also, if you would like to read some material on the JavaScript language, you might consider “JavaScript: The Good Parts”. This text analyzes the JavaScript language and recommends various approaches for clean and maintainable code. The other text, “JavaScript: The Definitive Guide” is an excellent reference.

Trademarks, disclaimer, and copyright information