WBTS Project - Architecture Guide 1.2

Updated July 16th, 2011

Table of Contents

Code Style

This section briefly describes how the WBTS system was coded in its stylistic and syntactical form. People looking to modify or build off of WBTS should read this section to get an understanding of the coding style chosen.

Classes – All class names are camel case such as “ClassName”. Method names are lowercase camel case such as “methodName”.

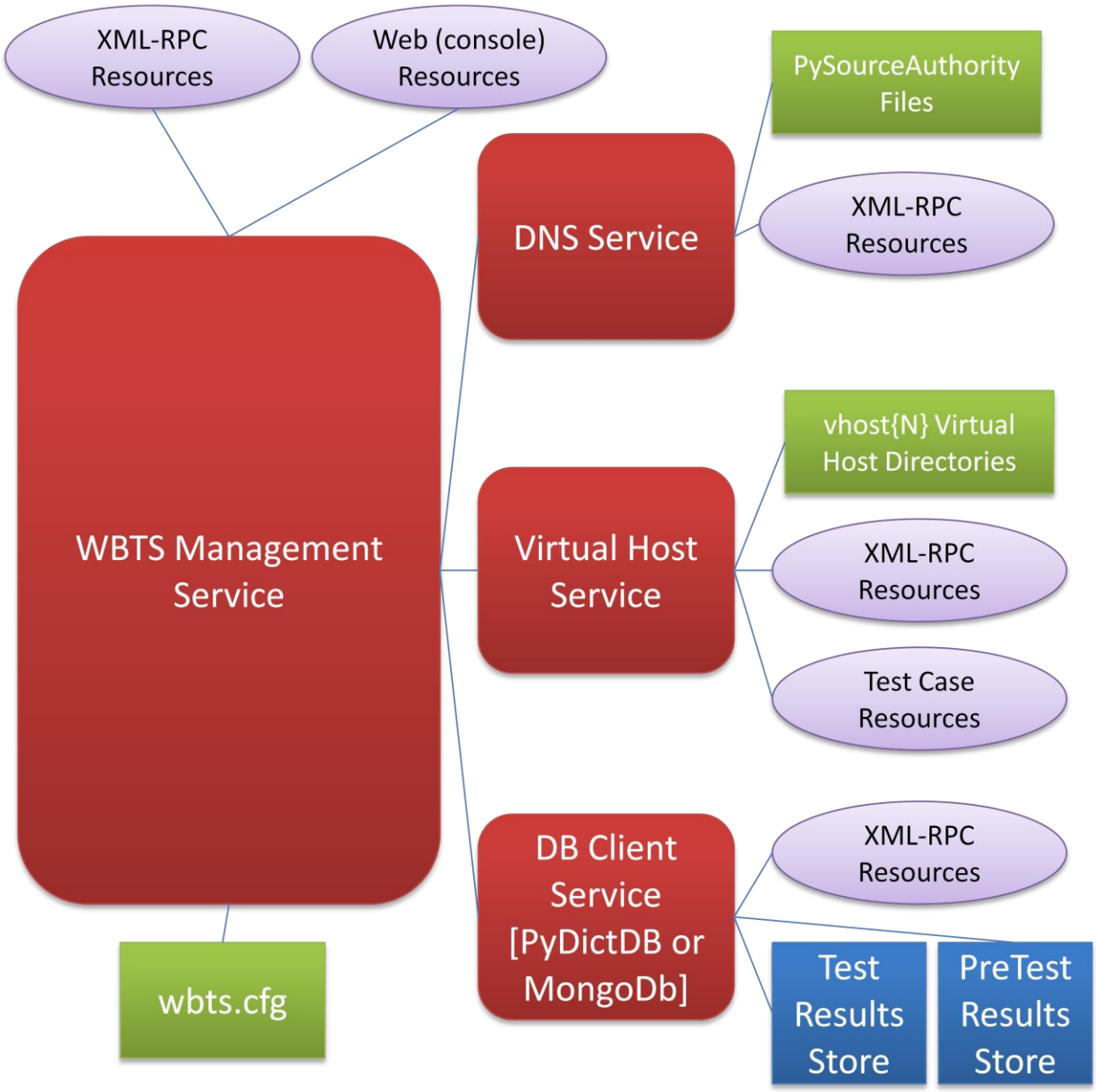
Class Members – All class members begin with an “m” – denoting that they are member variables -- and are of camel case such as “self.mMemberVariable”.

Function names – Function names are all lowercase with underscores in between words such as “function_name”

Method and Function Variables – Like function names, variable names are all lower case with underscores in between words such as ”variable_name”.

System Overview and Architecture

As with most code these days, WBTS is very much just glue that holds together framework code. In this case WBTS uses the Twisted framework. The Twisted framework was selected for multiple reasons such as performance but most importantly because it offers a multitude of pre-written services that are useful for testing User-Agents.



For people who are not familiar with the Twisted framework, the technology can appear quite daunting. Especially with the usage of the asynchronous deferred design. It is highly recommended that anyone wishing to modify WBTS learn the Twisted framework first, which although may take a while to understand, is a worthy investment. The Twisted documentation can explain in great detail how services are handled, as such it will not be documented here.

The majority of application logic is housed in the Management Service. As it has access to all of the various services, you will see a reference of it passed around to various classes on initialization.

Application Startup

As WBTS is a Twisted application you will need to use the "twistd" application to launch it. For Microsoft Windows based systems the command line would be: "twistd.py -noy wbts.tac" and for Unix based systems just "twistd -noy wbts.tac". The system starts in the following manner.

1. "wbts.tac" calls the wbts.config.wbts_config script to read in the configuration values and ensure they are sane.
2. Next wbts.wbts_application creates a Twisted multiservice and starts services in the following order:
    a. Management Service (also houses the storage system)
    b. VirtualHosts (HTTP and HTTPS)
    c. DNS Service
3. If we are a posix based system, we will shed privileges, if configured, to the desired user.
4. Call setServiceParent and begin waiting for events/requests.

Directory Layout

WBTS is laid out in the following structure on the file system.

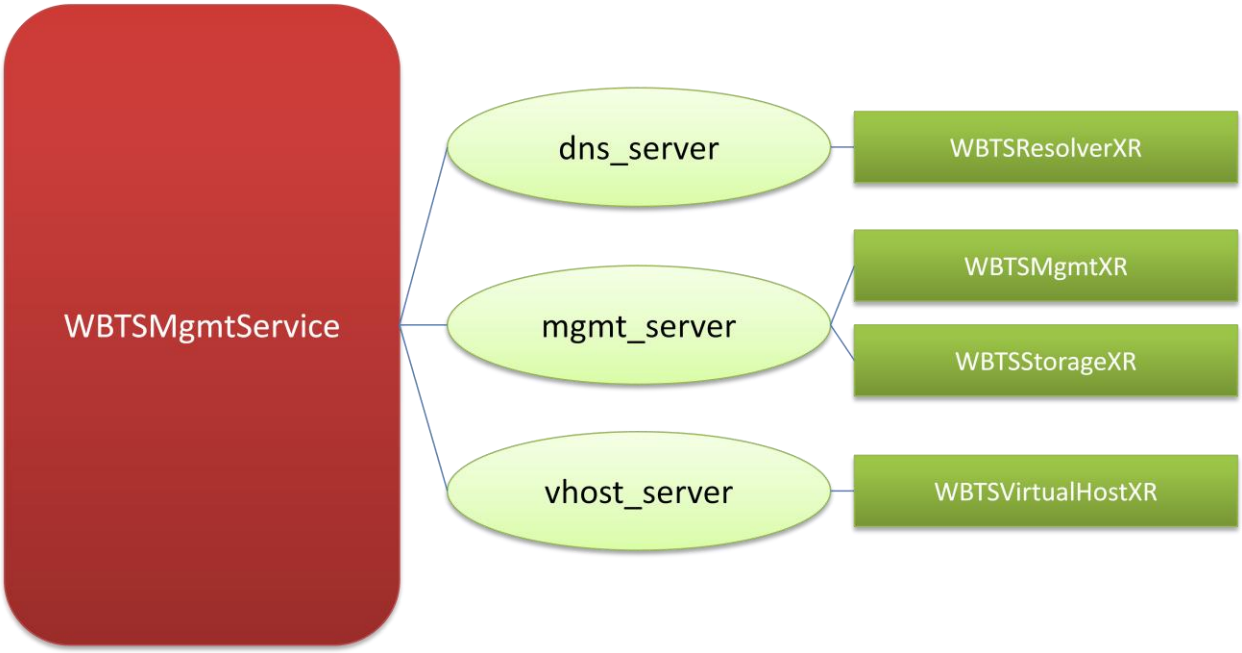
Directory	Description
/wbts/	The wbts application code directory
/config/	The startup configuration validator
/dns/	Code for the DNS Service
/resc/	Resources for data storage
/storage/	The storage system (currently supports pydictdbs and mongodb)
/utils/	Various utility modules and functions
/web/	Contains the code for handling test cases, custom HTTP processors and code that modifies the twisted HTTP channel.
/data/	The directory to store data if using the PydictDB data store.
/named_root/	The PySourceAuthority files, where each ".py" file contains the records of a single DNS authority.
/web_root/	Contains the various resources (.html/.css/.js) for the virtual hosts and the management interface.
/mgmt/	The html and java script for the management interface
/shared/	A directory that is shared across all virtual hosts and the management interface.
/vhost_root/	The directory containing the virtual hosts.
/vhost{N}/	A directory to be mapped to a domain name. Multiple domain names can map to the same directory.

Components

WBTS can be broken into a number of different components and services. This section will outline how these pieces interact as well as the design of each component. Again, the majority of logic is handled by the Twisted framework and anyone wishing to learn the intricacies of how these services are working is strongly recommended to read the Twisted documentation and source.

Management Service

The management service is a web based user interface to make management of WBTS services easy as well as providing a graphical interface to reviewing test case results. The management of WBTS is done primarily through services and components that have exposed XML-RPC interfaces.



The MgmtService has a method called “addServiceXRResources” which takes a reference to the service, the XML-RPC resource class and the URI to map it to within the MgmtService. This allows new services to easily expose their own methods via XML-RPC.

wbts.cfg

The WBTS Configuration file is in a common Unix configuration file format. It is modified by using Python’s built in ConfigParser module. It contains multiple sections with the usual key = value format. All sections are required. For the most part, each value is validated that it is somewhat sane, then assigned to a “config” dictionary object that contains all configuration parameters. The logic for handling the HTTP processors (which are used to handle say, PHP script, PL script or the built in twisted RPY script) is a little more involved. For processors, the key is extension (such as PHP) or file type of the processor and the value contains either a one or two pieces of information, separated by a comma. The first piece is the namespace to the class that handles that extension type. For example the configuration line “rpy = script.ResourceScript” signifies that ”.rpy” files should be parsed by the

twisted.web.script.ResourceScript processor. Because non-twisted processors are supported by WBTS, an alternate namespace (not twisted.web.*) is also available. It is possible to build custom processors by adding them to the wbts.web.processors module. For example to support PHP script, you will need to first add a class to the wbts.web.processors.py script such as the one shown below:

```
class PHPProcessor(twcgi.FilteredScript):
   filter = '' # Points to the php parser
```

To have the virtual hosts actually call this processor, you will need to add a line containing not only the namespace to the processor, but also the path to the executable that will parse it. In our above example the line would look like this: “php = processors.PHPProcessor,/usr/bin/php”. Note the comma separating the module’s path and the executable used to actually process the PHP scripts.

For virtual hosts that are defined in the “[VirtualHostMappings]” the hostname is the key, and the value is the directory where the vhost should be mapped to. In this way, multiple domains can easily be mapped to the same directory. It should be noted that the vhost directory is appended to the [WebSettings] “vhost_root” path. So in the event that: “vhost_root = /opt/wbts/web_root/vhost_root” and a VirtualHost is mapped such as “attacker.com = vhost1” the full path would resolve to ”/opt/wbts/web_root/vhost_root/vhost1”. The rest of the configuration is rather straightforward. The “config” dictionary object is passed to the management service and the various other services where they can easily access the values.

Since configurations can become quite complex a helper script called “create_config.py” exists in the same directory as the wbts.tac. This script can be used to quickly and easily generate new configurations.

DNS Service

The DNS service is the twisted implementation of “twisted.names.server.DNSServerFactory”. This service has been slightly modified to allow for creation of DNS authorities and records which are available instantaneously after creation. The majority of service logic is found in the wbts.dns_server module’s WBTSResolverService. This service also supports a client resolver. By assigning an alternate DNS server, you gain the ability to resolve other records outside of the ones that WBTS has the authority for. This still allows you to control domains that you don’t actually own, such as www.attacker.com or www.victim.com, but still resolve other hostnames.

DNS Records

The DNS records are basic Twisted Record_<type> records. However, an “id” parameter is dynamically added to each record, making it possible to easily find, add or remove them from our running service. Records of PyAuthoritySource and are loaded from the “named_root” directory assigned in the wbts.cfg file. An example of an authority file can be seen below:

```
zone = [
    SOA('attacker.com',
       mname = 'ns1.attacker.com',
       rname = 'root.attacker.com',
       serial = 2003010601,
```

```
        refresh = '3600',
        retry = '3600',
        expire = '3600',
        minimum = '3600'
    ),
    NS('attacker.com','ns1.attacker.com', 3600),
MX('attacker.com', 0, 'mail.attacker.com', 3600),
A('attacker.com', '192.168.3.6', 3600),
A('mail.attacker.com', '127.0.0.1', 3600),
CNAME('ftp.attacker.com','attacker.com', 3600),
CNAME('www.attacker.com','attacker.com', 0)
]
```

Of course you can always see Twisted's documentation for a more detailed description of the PySourceAuthority files and how they are configured. Keep in mind that only a few record types have been implemented. Currently, WBTS's DNS service supports the following record types: A, AAAA, MX, CNAME, NS and of course SOA.

We also create a new basic authority type defined as "MemoryAuthority" for dynamic records. This class is defined in wbts.dns.authority and just allows us to create a very basic authority without having to open or parse files but rather assign the necessary records to have the host names immediately resolvable. (Oddly enough, this type of authority object does not exist in Twisted, hence the reason it was created.) After a new authority or records have been modified, we need to update the zones or authorities by calling the WBTSDNSServerFactoryFromService's updateZones method. Once called, it will refresh our resolvers and the records will be available from the DNS service.

After changes have been made to records or authorities they usually want to be saved to disk. The WBTSResolverService also contains the logic for saving or converting "MemoryAuthorities" to the PySourceAuthority files. This process just pulls out the important information from the records, and 'stringifies' it into a proper PySourceAuthority file format. It should be noted this code is really ugly and should probably be rewritten at some point.

DNS Rebinding

One of the goals of WBTS is to allow for simplifying rebinding style attacks. To do this a special resource was created and mapped to all the virtual hosts under the URI "rebind". This resource, defined in wbts.web.vhost_resources under RebindRequest, allows for a single request to kick off the entire process. All that is required is the following:

1. A DNS Record that has a TTL of 0.
2. The zone/authority name (attacker.com)
3. The host name (www.attacker.com)
4. The record_type (A)
5. The new value(1.2.3.4)
6. The delay in seconds to flip the record back (5).

Since we are modifying records, we need to lock records so that the original value is not lost. This is done by appending the record we are rebinding to a list of locked records and basically denying any rebind requests until the record is returned to its original value.

You will not be allowed to rebind to a hostname that is controlled by WBTS. Since the client's "Host" header will still point to www.attacker.com (in the above example), WBTS will simply return www.attacker.com's contents. This is due to the fact that we are not binding VirtualHosts to their own interfaces and IP addresses. Unfortunately, when using an alternate DNS server, there appears to be a problem with twisted.names resolving CNAMEs that are pointing to a system outside of the DNS Servers authority files. The end result is that currently, we can only do DNS rebinding using A records in conjunction with using an alternate DNS server.

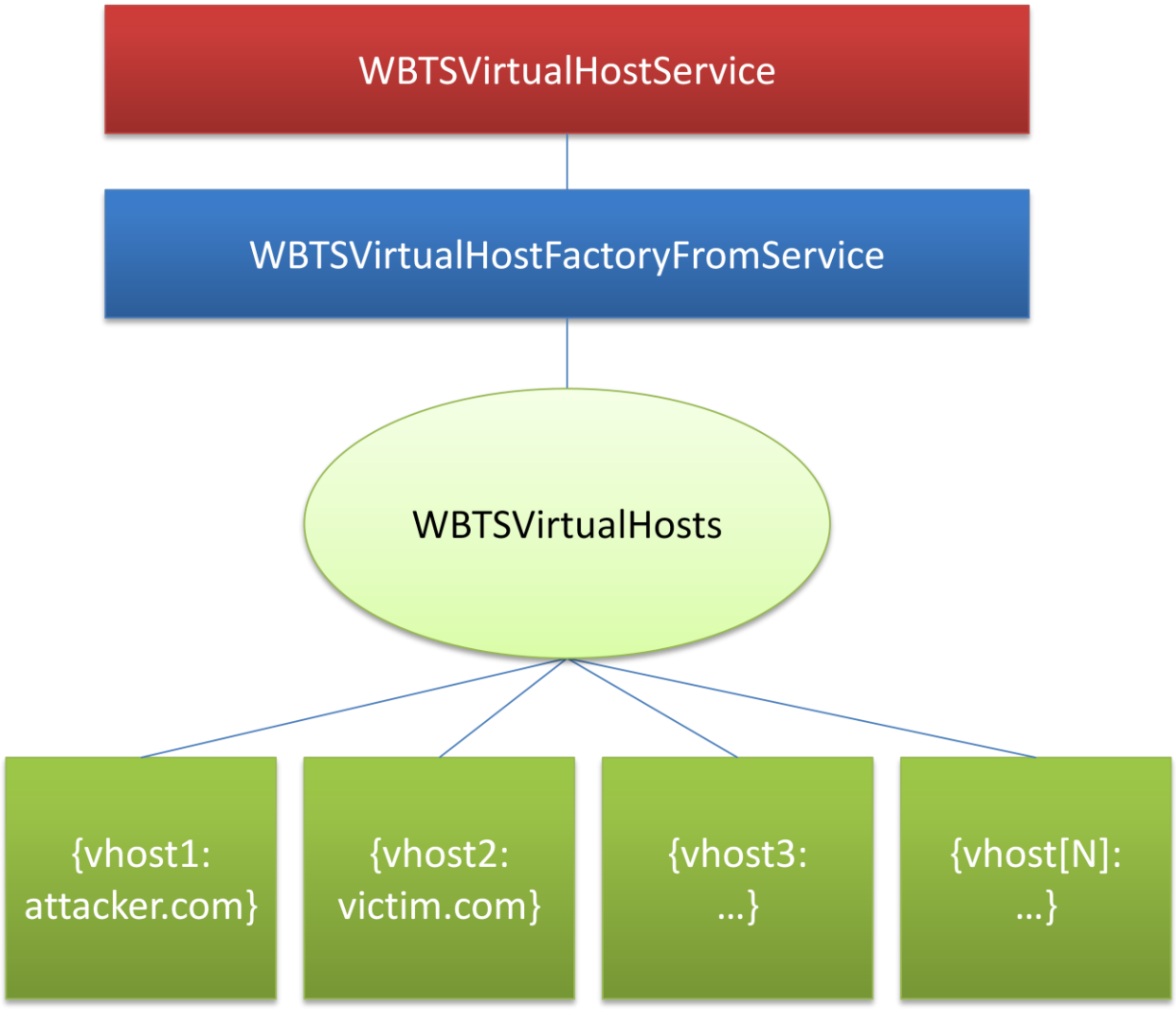
DNS Management XML-RPC Methods

All resources are defined in the "wbts.dns_server.WBTSResolverXR" and return data in XML-RPC format.

Resource	Description
getAlternateDNS	Retrieves the alternate DNS server and port.
setAlternateDNS	Set's the alternate DNS server and port.
getNamedRoot	Retrieves the directory that the records are stored in.
setNamedRoot	Sets the directory to store records in.
reloadAuthoritiesFromDisk	Reloads the DNS records from disk.
doesAuthorityExist	Checks if the authority (by name) exists
getAuthortiies	Returns a list of the currently served authorities.
doesRecordExistForAuthority	Checks if a record exists for the authority
getAuthorityRecord	Takes an authority name and returns the SOA record.
writeAuthorityRecord	Saves record to disk.
writeRecords	Writes all records to disk.
getRecordByIdAsDict	Takes an authority name and record id and returns a dict object of the record.
updateRecordForAuthority	Updates the authorities record with new values.
createRecordForAuthority	Creates a new record for the supplied authority.
removeRecordById	Removes the record identified by id from the authority.
createAuthority	Creates a new authority record
updateAuthority	Updates the authority record (SOA) information.
removeAuthority	Removes the authority from the system.
getAllAuthoritiesAsDict	Returns all authority records as a dict object.
getAuthorityRecordsAsDict	Returns all records for an authority as a dict object.

Virtual Hosts Service

The individual hosts used for testing are made available as virtual hosts. This allows a tester to quickly and easily test Same Origin Policy issues. The host resources and processors are handled by the wbts.vhost_server module's "WBTSVirtualHosts" class.



The WBTSVirtualHosts houses all of the virtual hosts. On application startup, its job is to map all of the processors, shared resources, extensions and the test case related resources to each of the virtual hosts.

Testing Resources

1. **ShowRequest** - While developing the various test cases, it was quickly realized that Twisted did not really support a method for looking at the raw request data that was sent to the server (think TRACE method.) To implement this ability, the default HTTPChannel needed to be modified. The WBTS implementation of the HTTPChannel can be found in wbts.web.channel.WBTSHTTPChannel. All the modification does is create a buffer to store the each individual received line data every time "lineReceived" is called. Since Keep-Alives will keep the data of multiple requests, we need to split out the data from the previous request into separate value called "last_request." This works out well because the "allContentReceived" method will be called after each individual request, regardless if we have a connection with multiple requests in a single channel.
2. **RebindRequest** – The details of this are described in the DNS Section under DNS Rebinding.

3. **GetResourceInfo** – This resource takes a single parameter “info”. If the info parameters value is “path” then it will respond with WBTSs shared directory path (as seen from the local file system). This is useful for tests which require file:// urls where the file system is either the same, or WBTS is running on the same machine as the target browser.
4. **SubmitTestForm** – This resource is for when a test requires information that is sent to a form. The resource will simply read in the posted form results and create the test case object. The raw output will be read from the WBTSHTTPChannel and assigned a variable. This resultant page of the form post will create the test case object and submit the results back automatically. The page template is in the ” web_root/shared/resources/test_form.template.html” file.

Parameter	Description
testcase_url	The test case URL with the client_id in the URI as a query parameter. It will be parsed out automatically into testcase/client_id
description	A description of the test
input	The input to the test
expected_result	The expected result
type_type	The test that is being performed, either “header” or ”upload”

5. **GetQueryOutput** – This resource will echo back data inside of an HTML page. A request with the “q” parameter will have the value inserted into an HTML page two times. This is useful for testing cross-site related issues when it is possible to have some influence of data returned in a page. The page template is in the” web_root/shared/resources/query_output.template.html” file.
6. **HeaderSetter** – This resource is actually a custom static.File resource. It is mapped to the entire shared directory as well as the testcase directory. Although it is used for serving files, its primary function is for allowing custom response headers to be injected dynamically. This is done by overriding the getChild method of the resource to look for the “headers” parameter. A full list of required parameters can be found in the below table.

Parameter	Description
headers	A single string containing multiple headers separated by CRLF characters. (example: headers=header1:%20value1%0d%0aheader2:%20value2)
encode	This tells the HeaderSetter resource how the ‘headers’ parameter was encoded. The valid options for encoding a location is either ‘b64’ (base64 encoding) or ‘url’. By default, encode is set to URL as spaces must be encoded on the query line.

7. **RedirectRequest** – A lot of same origin policy issues can be directly attributed to mishandling of HTTP redirect messages. This resource was built to aid in testing various redirect style bugs. It allows you to create custom redirect codes and locations.

Parameter	Description
loc	The location to be redirected to (example: http://google.com)
code	The HTTP response code the server should use. While a client may check 301 response codes to see if they are same origin, what about 307…?
encode	This tells the RedirectRequest resource how the 'loc' parameter was encoded. The valid options for encoding a location is either 'b64' (base64 encoding) or 'url'.

8. **SavePreTest** – This resource is used for saving test results prior to a test case being executed. The reasoning is that some test cases may never actually complete, for example calling some method which causes an internal error which causes the rest of the script to fail and not execute. The pre-test results are saved in a temporary document store "pretest_results". A process exists to take the pre-test data and populate it into the result document store, in the event that the automation system gives up. Using this pre-test resource is not mandatory, and can be safely ignored unless a test is known, or has a high chance of failing miserably. Note, that prematurely killing the blade runner (either JS or python versions) will cause the pretest data to be stuck in the pretest document store. This store is cleaned out on start of the WBTS service.

Parameter	Base 64 encoded?	Description
testcase_url	No	The URL of the test case.
target_host	Yes	The target hostname
user_agent	No	The user agent (taken from the request header.)
testcase	No	The test case name (the filename)
expected_result	Yes	The expected result of the test (as determined by the testcase)
input	Yes	The input that was used for the test case (as determined by the test case)
description	No	A description of the test case
client_id	No	The client id, which is parsed out of the URL (?client_id=<id>)
test_passed	No	A Boolean value for if the test passed or not (as determined by the test case)

9. **FailTest** – This resource will fail a test that exists in the pre-test document store. It takes a single parameter, the test case URL with the client_id as a query parameter. It will parse out the test case and the client id and move the pre-populated, failed version of the test into the real results document store and finally, removing the record from the pretest store.

Parameter	Description
testcase_url	The URL of the test case.

10. **SaveTest** – This resource is used for saving the results of a test case. For each test case, the following information will be saved to the "test_results" document store. It should be noted that not all parameters are required.

Parameter	Base 64 encoded?	Description
testcase_url	No	The URL of the test case.
target_host	Yes	The target hostname
user_agent	No	The user agent (taken from the request header.)
testcase	No	The test case name (the filename)
expected_result	Yes	The expected result of the test (as determined by the test case)
input	Yes	The input that was used for the test case (as determined by the test case)
output	Yes	The output data from the test case.
description	No	A description of the test case
test_passed	No	A Boolean value for if the test passed or not (as determined by the test case)

Testing Resources URL Mapping

The below table summarizes the URL mappings of the testing resources. Please see the WBTSVirtualHosts _addTestingResources method if you wish to add your own.

Resource	URI Mapping
ShowRequest	/showRequest
RebindRequest	/rebind
GetResourceInfo	/resourceInfo
SubmitHeaderTestForm	/submitHeader
GetQueryOutput	/getOutput
RedirectRequest	/redirect
SavePreTest	/tc/savePreTest
FailTest	/tc/failTest
SaveTest	/tc/saveTest

Virtual Host Management XML-RPC Resources

Only a few methods are exposed to the Virtual Host Management service.

Resource	Description
getVirtualHostList	Returns a list of virtual hosts
addVirtualHost	Takes a domain name and a path and creates a new virtual host. Note directories are NOT created and must exist on the system.
addVirtualHosts	Takes a dictionary of {domain:path} virtual hosts and creates them.
removeVirtualHost	Removes the mapping of the virtual host.

WBTS Storage System

WBTS can be configured to use either a local PyDictDB or a MongoDB data store. If you wish to implement your own storage system, see the storage.py module for the base class to extend. There are two primary objects a Database and a Table. The Database class is primarily concerned with handling saving/loading/connecting. The table class is for dealing with the data itself.

Storage System XML-RPC Resources

For the management side of the DB, the WBTSStorageXR class exposes the data store's methods to other services.

Resource	Description
get_type	Returns the data store type (mongo/pydict)
find_one	Takes a db name and a search key (dictionary {col:value}) and returns only the first record.
find	Takes the db name and alternatively a starting point (int) and a limit of records to return (int) and alternatively a search key (dictionary {col:value}) and returns all records found within skip:limit
count	Takes a db name and returns the number of records in the specified db.
insert	Takes a db name and a record to insert (dictionary of {col:value}).
remove	Takes a db name and a search key and removes all records found.
drop	Takes a db name and drops all data/data store from our service.
save	Only used for WBTSPydictDB, saves the data as a pickled data file to disk.

Test Case System

The test case system is a collection of server side and client side code which allows for multiple forms of automation. Some browsers, in particular mobile ones, make it difficult to automate with an external application. For this reason a "web only" (i.e. java script) version of automation was implemented. The test case resources are built using the XML-RPC methods that. People looking to parse the test case information should look at using the default python xmlrpclib module.

The test cases themselves are single files. For the most part they are written in HTML. However, tests requiring more interactivity can be built using any of the defined processors (.rpy/.php etc). The test cases are found in the "web_root/shared/testcases" directory and mapped to each virtual host as "/testcases/". Test cases are broken into their individual test types (such as header tests, object based tests, various browser regression tests, etc.) and put into their own sub-directories. The test case id is the filename itself. So it is strongly recommended to make the file names as unique as possible.

Server Side

Test Case Resources

The test case system is mapped to all WBTS virtual hosts under the '/cases' URI. As stated above, all of the resources return data in XML format.

Class.Method	URI Mapping	Description
TestCases.xmlrpc_getTestTypes	/cases/getTestTypes	Returns a list of all directories found in the shared test case directory.
TestCases.xmlrpc_ getCasesOfType	/cases/ getCasesOfType	Returns all cases (except those files found in the excluded.txt file) found in the subdirectory of "test_type" where test_type is a specified argument.

In the sub-directories of the top level '/testcases/' path you may include a text file called 'exclude.txt'. The purpose of this file is to allow excluding files (one per line) from being returned in the getTestCases or getCasesOfType XML-RPC methods. This is useful for when you have supporting files that are not accessed directly or if you wish to remove specific tests. If you wish to exclude an entire directory (and all of its sub directories) just append a / to the end of the directory name. So to exclude the "junk" folder you would add "junk/" or "junk\" to its own line in the "exclude.txt" text file. Please note this file can exist in any directory. If you wish to exclude certain "css" test cases, please put the "exclude.txt" file inside of the css directory.

Blade System

The blade automation system resource is mapped to the '/cases' URI. A single method (getTestProgress) is available for checking the progress of test cases. Using a configurable timer to determine when to go to the next case is not the best method for running through multiple test cases (however, this method is supported in the external blade application). Because of this, a method exists for the blade client to send in a client id and test case id to check up on the browsers progress of a particular test. The blade

system can continually check to see when the test case data has been recorded on the server side, and trigger the browser to move on to the next case after a certain period of time.

It is possible that a test case may cause the browser to hang or stop responding. In the event that this happens the pre-test functionality can be used. You can fill out TestCase object with data such as the description, input information, client id and other supporting information and prematurely send it to the server. This information is stored in a temporary document, **hard coding** the test as **failed**. Next the test can be executed, and the JsBlade or Blade system can check if the test worked by calling getTestProgress with the test case and client id. If the test did indeed fail, the failTest method will take the information from the temporary data store and move it into the final results store, after which it will remove it from the temporary data store.

Class.Method	URI Mapping	Description
TestCases.xmlrpc_getTestProgress	/cases/getTestProgress	Takes the testcase (filename) and client id, and checks if the result exists in the results document. It will return a 1 for exists, a 0 otherwise.
TestCases.xmlrpc_failTest	/cases/failTest	Takes the test case (filename) and client id, and initiates the fail test sequence. (Copying the pretest data if it exists into the final document store and removing it from the pretest store)

Client Side TestCase Object

The heart of the client side test case system can be found in the testcase.js file found in the "web_root/shared/scripts" folder. This object is created once per test case. The test case data is encoded using the built in java script escape function. It should be noted that only browsers which implement JavaScript maybe used with WBTS. Thankfully, these days it is quite hard to find a browser that does not support JavaScript. However, attempts were made to use standard functions that should work on the majority of browsers.

The test case object has methods that allow you to pass in a callback, provided you need to check the results of a specific test. For example, say you wish to test which headers may be added and would like to see the exact client request. By calling the '/showRequest' resource, you can get the response data and, using JavaScript, parse out the pertinent data. Then you would assign the response data to your test case object and record it using the "saveTest" method. In the event that you believe a test will cause the interpreter to fail, but would like to determine that the client browser at least started the test, you can use the "savePreTest" method and save all the details prior to running the test. This method

takes a call back which you can use to call into your test function. After you run your test, you then use the same test case object's "saveTest" method to save the results and remove it from the temporary document store.

The Blade Process Flow (Standard Test)

1. The blade client is started with a target virtual host, the browser type, and other client side configuration values.
2. The browser instance is automatically started (or connected to if an emulator is used).
3. The blade client connects to the WBTS host and gets a list of test cases.
4. The blade client generates a unique id (client_id)
5. The blade client starts up its own monitor looping call for closing pop-ups and dialog boxes.
6. The blade client loads the first test case URL, but appends ?client_id=<client_id> to the end of each test case URI and tells the browser to start loading that URL.
7. The test case is run in the browser and fills out the TestCase java script object with the results. This includes the current URL with the "?client_id=<client_id>" data.
8. The test case uses the XmlHttpRequest object to send the test case object's data to the server.
9. The server parses out the test case results, and parses out the client_id and adds that to the result document store.
10. The blade client asks the WBTS server the progress of test case for that client_id.
    a. The WBTS server checks to see if the result for client_id for that test case exists.
    b. If it exists, it will respond with 1, if it does not exist it, it responds with 0.
11. If the test was not complete, poll every 3 seconds for 3 iterations. On the final iteration, move on to next test logging locally which test case failed.

The Blade Process Flow (Pre-Test Version)

1. Follow steps 1-7 from the standard test version.
2. The JavaScript method "savePreTest" is called to temporarily store the results. This method contains an argument for a call back which is passed a function or method to run after the pretest data is saved.
3. The call back method/function is called and the test is run.
4. The test case then calls the "saveTest" method and sends in the results to the server.
5. The server parses out the test case results, and parses out the client_id
6. It takes the client_id/testcase information and checks if there is data in the pretest document store and adds that to the result document store.
7. The blade client asks the WBTS server the progress of test case for that client_id.
    a. The WBTS server checks to see if the result for client_id for that test case exists.
    b. If yes, respond with "1", if no, respond with "0"
8. If the test was not complete, poll every 3 seconds for 3 iterations. On the final iteration, call 'failTest' with the testcase URL (containing the client_id parameter/value). This tells the WBTS server to copy over the pre-test results into the results table and delete the pre-test data.

The JsBlade Process Flow (Standard Test)

The JsBlade is a java script implementation of the blade automation system. Basically, it's a single java script file called "runner.js" found in the "/shared/scripts" path.

1. Browse manually to the "/bladei.html" or "/blade.html" pages to load the blade runner page.
2. JsBlade will automatically load the available test types
3. Chose a test type from the pull down list
4. Manually click Start to start the tests.
5. The JsBlade object will create a unique id
6. The cases of the selected type (from the pull down list) will be retrieved from the getTestTypes resource.
7. The iframe / frame element's src attribute will be updated with the testcase url plus an appended client_id parameter and the generated client_id value.
8. After one second, the jsblade object will call the /cases uri with a getTestProgress request, passing in the client_id and the testcase.
9. If a 0 is returned, then it will call setTimeout, checking again in 2 seconds. If that fails again, then 4 seconds, then 4 seconds again finally giving up and moving on to the next case. However, if getTestProgress returns a 1, the jsblade object will immediately go to the next case.
10. After all cases have been visited the script will finish.

The JsBlade Process Flow (Pre-Test Version)

1. Follow steps 1-7 as outlined above in the Standard Version.
2. The test case sets up the JavaScript "TestCase" object and populates the various required data.
3. The JavaScript method "savePreTest" is called to temporarily store the results. This method contains a argument for a call back which is passed a function or method to run after the pretest data is saved.
4. The call back method/fuction is called and the test is run.
5. The test case calls then calls the "saveTest" method and sends in the results to the server.
6. The server parses out the test case results, and parses out the client_id
7. It takes the client_id/testcase information and checks if there is data in the pretest document store. and adds that to the result document store.
8. The blade client asks the WBTS server the progress of test case for that client_id.
    a. The WBTS server checks to see if the result for client_id for that test case exists.
    b. If yes, respond with "1", if no, respond with "0"
9. If the test was not complete, poll every 3 seconds for 3 iterations. On the final iteration, move on to next test. After one second, the jsblade object will call the /cases uri with a getTestProgress request, passing in the client_id and the testcase.

10. If a 0 is returned, then it will call setTimeout, checking again in 2 seconds. If that fails again, then 4 seconds, then 4 seconds again finally giving up and moving on to the next case. However, if getTestProgress returns a 1, the jsblade object will immediately go to the next case.
11. After all cases have been visited the script will finish.

DOCUMENT HISTORY	Date	Author	Version
Initial Draft	9/20/2010	Isaac Dawson	1.0
Minor modifications/fixes	9/21/2010	Isaac Dawson	1.1
Minor modifications, updated to reflect new architecture	7/16/2011	Isaac Dawson	1.2