# Test-Debug-Driven Development of **csapreproc.js**

## Contents

## Introduction

Development of CSA preprocessor core file, **csapreproc.js**, is done incrementally by modifying it to make it pass new test cases while not to fail existing test cases.

Test cases are presented as JavaScript programming code, as if the code is programmed by human developers, included in HTML files. Each HTML file is one test execution unit. A CSA preprocessor uses csapreproc.js to process an HTML file, analyze JavaScript programming code included in the HTML file, and generates a client side JavaScript file and one or more server side JavaScript files. Development of csapreproc.js is to modify it to make it generate desired client side JavaScript file and server side JavaScript files, through a debugging process.

A test utility is provided to execute all test cases. Run the utility to check whether a modification of csapreproc.js makes it fail existing test cases.

## Setup of Test Environment

### Web Environment

A test environment can be a Windows computer with IIS enabled and PHP installed. PHP is used by some development utility.

Server technologies to be supported should be installed. To support PHP as a server technology, PHP 7 is needed to run V8Js. To support ASP.Net as a server technology, .Net Framework 3.5 is needed and ClearScript (https://clearscript.codeplex.com/) is used to run JavaScript. To support other server technologies, corresponding software should be installed. For example, install Node.js to support using Node.js as server technology.

A web site is created on the Windows computer as the test environment. Suppose the web site name is **CSATest**. Then the URL for the web site is http://localhost/CSATest

Let's use **[CSATest Folder]** to represent the physical folder for the web site of http://localhost/CSATest. The files and folder structure of the test environment are shown below.

**[CSATest Folder]** HTML files adjusted by the CSA Preprocessor are placed in this folder

**esprima.js** – file provided by Esprima ( https://github.com/ariya/esprima)
**escodegen.browser.js** – file provided by Esprima ( https://github.com/ariya/esprima)
**csa.php** – this is a server side dispatcher for using PHP as a server technology
**csa.aspx** – this is a server side dispatcher for using ASP.NET as a server technology
**csaServer.js** – this is a server side dispatcher for using Node.js as a server technology
**readfile.php** – this is a PHP utility for supporting development of the CSA Preprocessor.
**[assets]** this is a folder needed by Esprima. See https://github.com/ariya/esprima to see what files and folders should be under this folder
**[bin]** this is a folder needed by ASP.NET as a server technology. Files inside this folder are listed in file ASPXBIN.txt
**[clientjs]** this folder holds client JavaScript files generated by a CSA preprocessor, and client side libraries.
**csa.js** – it is the client side dispatcher
**FileAPIclient.js** – it is a client side library to support file upload in a CSA environment
**[csapre]** this folder holds CSA preprocessor
**csapreproc.js** – the core functionality of a CSA Preprocessor is coded in this file. Development of the CSA Preprocessor is mostly done by modifying this file
**csapreproc.html** – this file is a web page to host csapreproc.js and making functionality of csapreproc.js available to other utilities, for example, the test utility.
**csapreproc_debug.html** – this file is a web page to host csapreproc.js for developing csapreproc.js by debugging it inside a web browser. For details of how to use it to do development, see DebugDrivenDevelopment.PDF
**csapreproc_test.html** – this file is a web page used by a test utility to compare files to determine if test cases are passed or not.
**[testcases]** this folder holds all test HTML files. Each HTML file is one execution unit for test cases.
**[libPhp]** this folder holds PHP library files
**V8IsExt.php** – this file creates a class to extend V8Js for using PHP as a server technology for CSA
**STI_DbExecuter.php** – this file implements a sample database API
**STI_FileWriter.php** – this file implements a sample file-writer API
**STI_MailSender.php** – this file implements a sample mail-sender API
**STI_ServerFileAPI.php** – this file implements a sample FileAPI API
**DbCredential.php** – this file provides credential support for STI_DbExecuter.php
**mailID.php** – this file provides credential support for STI_MailSender.php
**[serverjs]** this folder holds server side JavaScript files generated by the CSA Preprocessor, and server side libraries
**mailtool.js** – a sample server side API for sending emails
**database.js** – a sample server side API for executing database commands
**FileAPIserver.js** – a sample server side API for uploading files
**serverA.js** – a sample server side API
**server.js** – a sample server side API
.

## PHP as Server Technology

To use PHP as a server technology for CSA, a server side dispatcher is implemented in a PHP file, csa.php, which is placed in the test web site folder. The “server-technology-independent API” is implemented in PHP files, which are placed in folder “libPhp”.

## Node.js as Server Technology

To use Node.js as a server technology for CSA, a server side dispatcher is implemented in a server side JavaScript file, csaServer.js, which is placed in the test web site folder. The “server-technology-independent API” is implemented in server side JavaScript files, which are placed in folder “serverjs”.

## ASP.NET as Server Technology

To use ASP.Net as server technology for CSA, server side dispatcher is implemented in a DLL, csa.DLL which is launched by a file csa.aspx. The file csa.aspx is placed in the test web site folder. The “server-technology-independent API” implementations have to be in DLL files, which are placed in folder “bin”.

The client side dispatcher, csa.js, makes an asynchronous server connection to csa.aspx which acts as a server side dispatcher for using ASP.NET as server technology. But asp.aspx needs to load a DLL, CSA.DLL, to perform CSA functionality. CSA.DLL will load needed DLL files which implements “server-technology-independent API” used by server side JavaScript code. This section lists projects for generating CSA.DLL and other DLL files.

The projects are in the following GitHub folder: Source/Samples/ServerImplementation/ASPX

### JScriptInterface

This C# project compiles to CSA.DLL. It is a server side dispatcher invoked by csa.aspx. It executes server side JavaScript via ClearScript (https://clearscript.codeplex.com/). It also loads other DLL files which are referenced in the server side JavaScript to be executed.

### Database

This C# project compiles to database.DLL. It provides a `STI_DbExecuter` class which is referenced in a sample “Server-Technology-Independent API” file database.js.

### WebMail

This C# project compiles to webmail.DLL. It provides a `STI_MailSender` class which is referenced in a sample “Server-Technology-Independent API” file mailtool.js.

### FileAPI

This C# project compiles to FileAPI.DLL. It provides a `STI_ServerFileAPI` class which is referenced in a sample “Server-Technology-Independent API” file FileAPIserver.js.

### ServerTasks

This C# project compiles to ServerTasks.DLL. It provides a `ServerTaskX` class which is referenced in a sample “Server-Technology-Independent API” file ServerA.js.

## CSA Preprocessor Utility

A Windows standalone program, csapreproc.exe, is provided as a CSA Preprocessor utility. It is a C# program in Visual Studio 2012 project.

### Usage

This utility is executed via following command line parameters.

- /h “full path of an HTML file to be processed” – the HTML file contains JavaScript programming code to be processed. For example, an HTML file contained in the “testcases” folder.
- /o “output folder” – the files generated by this utility will be saved in this folder. Suppose the HTML file is Page1.html specified via “/h” parameter described above, then following files will be generated in the output folder:
    - page1_client.js – it contains client side JavaScript code
    - page1_server1.js, page1_server2.js, … -- one or more server side JavaScript files

    - page1_p.html – it is an HTML file generated from page1.html by adjusting its JavaScript file inclusions: Server side JavaScript libraries and some client side libraries are removed, page1_client.js is included, and a client side dispatcher, csa.js, is included.
- /u “URL to csapreproc.html” – by above chapter “Setup of Test Environment”, this parameter is http://localhost/CSATest/csapre/csapreproc.html
- /j “folder for JavaScript files included in the HTML file” – if this parameter is not given then the folder for the HTML file is used. Suppose a JavaScript library file is included by <script src=”clientjs/mylib.js”> then the folder specified by this parameter is combined with “clientjs/mylib.js” to form a full physical path of the JavaScript file.
- /s – if this flag is given then the utility will close once the processing finishes. It indicates “silent mode”, and message boxes will not be displayed.

## Source code

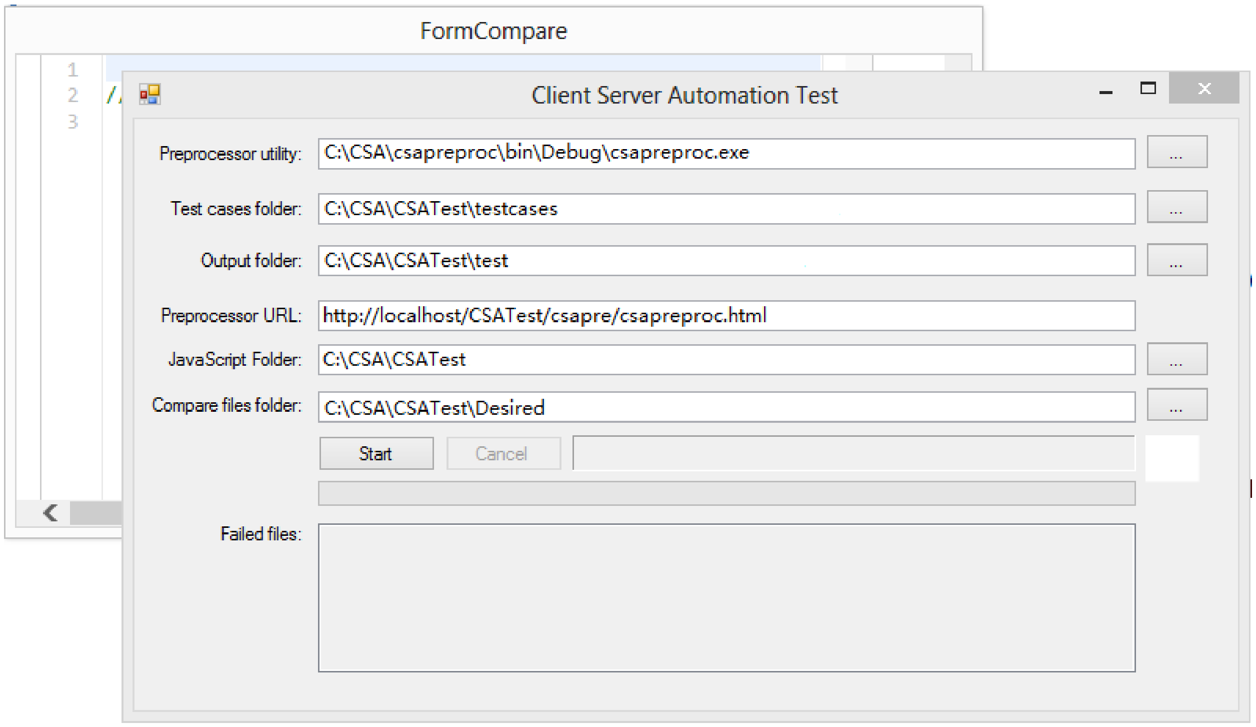
Source code for this utility can be downloaded from Github.

## Test Utility

A Windows standalone program, csaTest.exe, is provided as a utility for testing the CSA Preprocessor. It is a C# program in Visual Studio 2012 project.

### Usage

It is a Windows Form program. It provides an UI for entering and remembering all parameters.

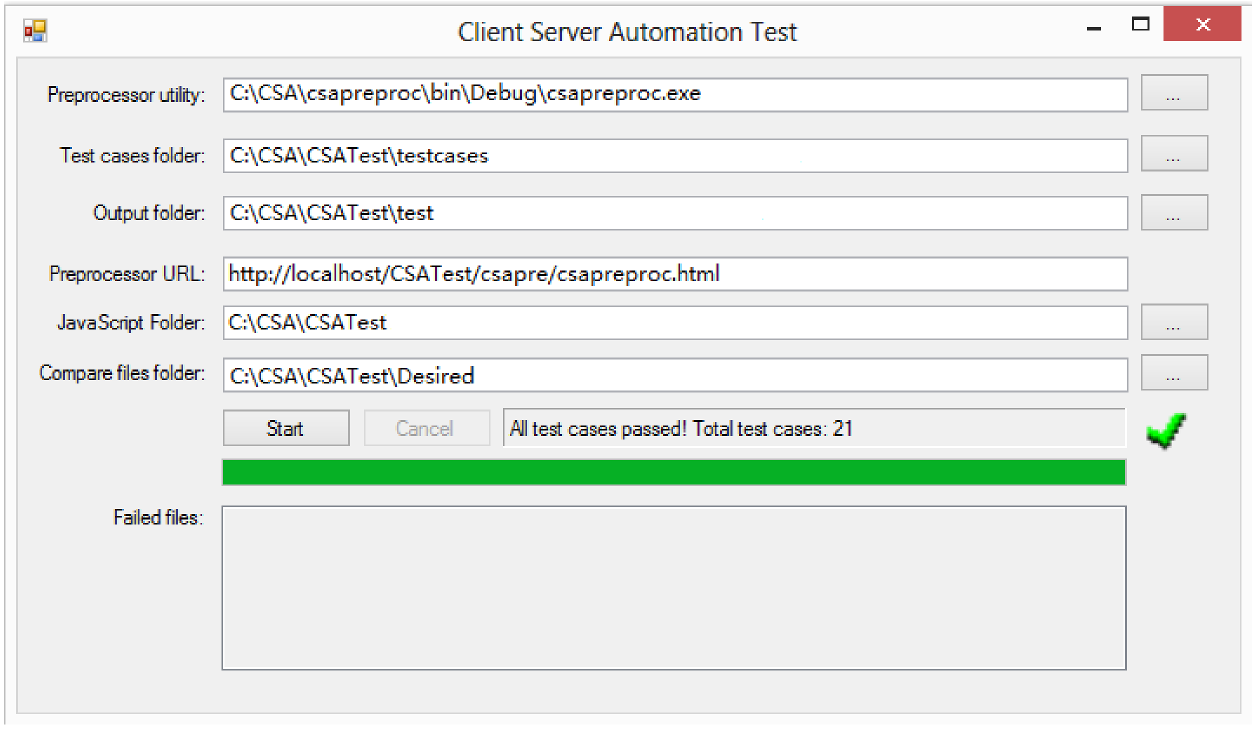


Preprocessor utility: C:\CSA\csapreproc\bin\Debug\csapreproc.exe
Test cases folder: C:\CSA\CSATest\testcases
Output folder: C:\CSA\CSATest\test
Preprocessor URL: http://localhost/CSATest/csapre/csapreproc.html
JavaScript Folder: C:\CSA\CSATest
Compare files folder: C:\CSA\CSATest\Desired
Failed files:

The parameters will be remembered in a file named config.xml.

- Preprocessor utility – this is a full path to csapreproc.exe. See a previous chapter “CSA Preprocessor Utility” for details about csapreproc.exe.
- Test cases folder – this is the folder holding all test case HTML files
- Output folder – this is the folder for saving files generated by csapreproc.exe. See description of “/o” parameter of csapreproc.exe.
- Preprocessor URL – this is the URL to csapreproc.html. See chapter “Setup of Test Environment”. See description of “/u” parameter of csapreproc.exe.
- JavaScript Folder – this is the folder for locating JavaScript files. See description of “/j” parameter of csapreproc.exe.
- Compare files folder – this is the folder holding all desired JavaScript files for all test cases. This utility compares generated files with files in this folder to determine whether test cases are passed.

Click “Start” button to start testing.



Preprocessor utility: C:\CSA\csapreproc\bin\Debug\csapreproc.exe

Test cases folder: C:\CSA\CSATest\testcases

Output folder: C:\CSA\CSATest\test

Preprocessor URL: http://localhost/CSATest/csapre/csapreproc.html

JavaScript Folder: C:\CSA\CSATest

Compare files folder: C:\CSA\CSATest\Desired

All test cases passed! Total test cases: 21

## Source code

Download source code from Github.

A web page, csapreproc_test.html, is used by this utility to compare existing files with generated files to determine if desired files are generated.

## Use of Test Driven Environment

To use the above test-driven development environment to modify csapreproc.js follow steps below.

1. Create a HTML web page and include JavaScript code representing new test cases. Place the HTML web page in the "Test cases folder"
2. Use the CSA Preprocessor Utility to process the HTML file and examine generated client side JavaScript file and server side JavaScript files
3. If the generated files are not what you desired then modify csapreproc.js to make it generate desired files. See Debug-Driven chapter for adopting a debug-driven process to modify csapreproc.js
4. During step 3, run the test utility after a modification of csapreproc.js to make sure that any modifications of csapreproc.js do not damage existing test cases.
5. Once desired output files are reached, copy the desired output files to "Compare files folder". Thus, any future modifications of csapreproc.js will not damage the work performed in step 3.

## Debug-Driven Development

In step 3 of the above "test-driven" development, we need to modify file csapreproc.js to make it to generate desired JavaScript code. We will use a web browser to debug csapreproc.js to see why it does not generate desired output and modify csapreproc.js accordingly. This is a process of debugging csapreproc.js. This chapter describes utilities which are helpful for the process.

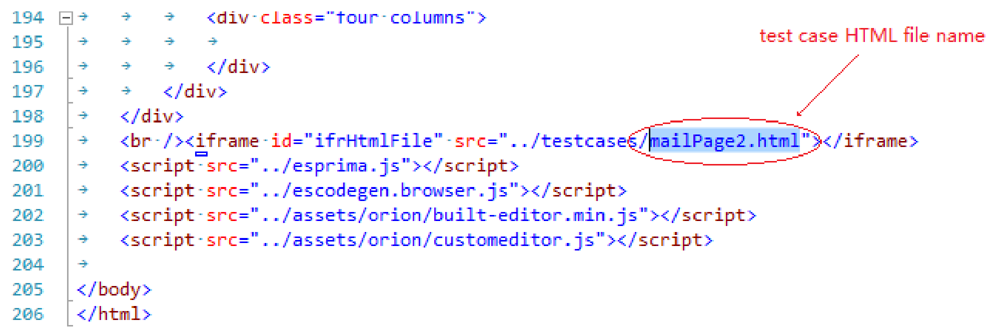
### Debug in a web browser

Almost all kinds of web browsers provide debuggers. In this chapter, we will use Chrome as an example to do debugging of csapreproc.js.

#### Assign test case HTML file

In folder [csapre], there is a web page, **csapreproc_debug.html**. It hosts file csapreproc.js, which is the file we are going to debug and modify.

A test case HTML file is to be loaded by csapreproc_debug.html and processed by csapreproc.js. We use a web browser to debug csapreproc.js to make it generate desired client side and server side JavaScript files. Since we are going to repeatedly launch csapreproc_debug.html for the same test case HTML file, we hard code the test case HTML file name in csapreproc_debug.html:



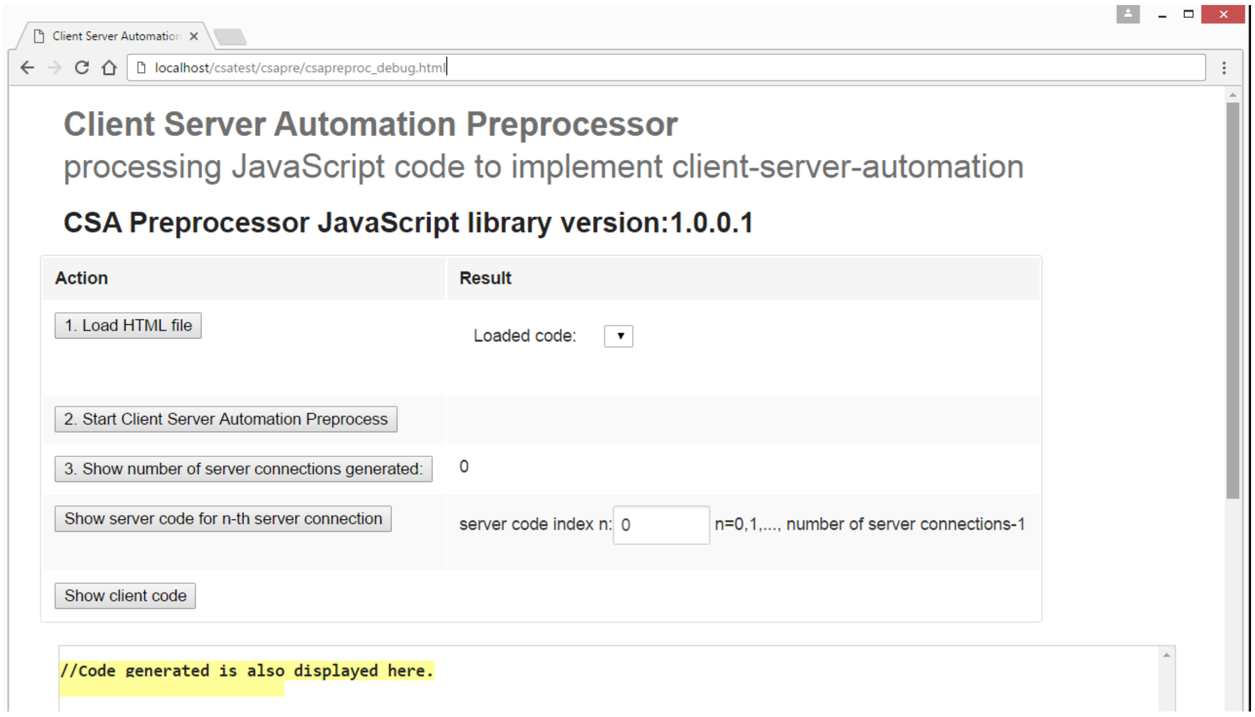
```
            <div class="tour columns">

            </div>
        </div>
    </div>
    <br /><iframe id="ifrHtmlFile" src="../testcases/mailPage2.html"></iframe>
    <script src="../esprima.js"></script>
    <script src="../escodegen.browser.js"></script>
    <script src="../assets/orion/built-editor.min.js"></script>
    <script src="../assets/orion/customeditor.js"></script>

</body>
</html>
```

#### Process test case HTML file

Use URL http://localhost/CSATest/csapre/csapreproc_debug.html to start debugging:


Client Server Automation Preprocessor

processing JavaScript code to implement client-server-automation

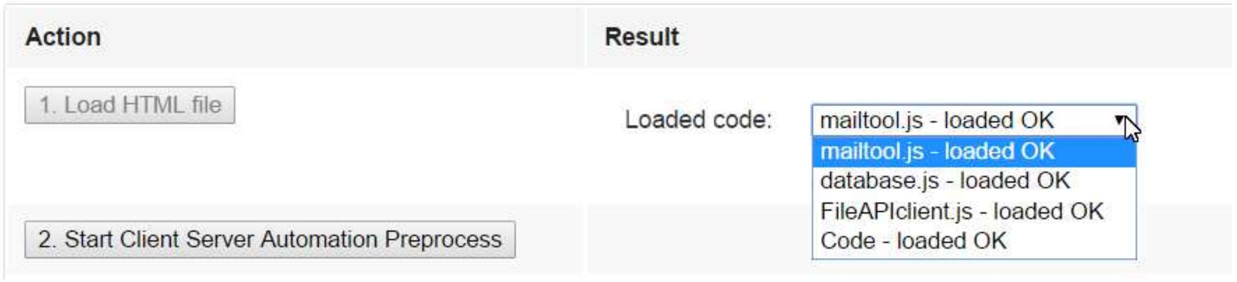
CSA Preprocessor JavaScript library version:1.0.0.1

| Action | Result |
|---|---|
| 1. Load HTML file | Loaded code: |
| 2. Start Client Server Automation Preprocess | |
| 3. Show number of server connections generated: | 0 |
| Show server code for n-th server connection | server code index n: 0 n=0,1,..., number of server connections-1 |
| Show client code | |

```
//Code generated is also displayed here.
```

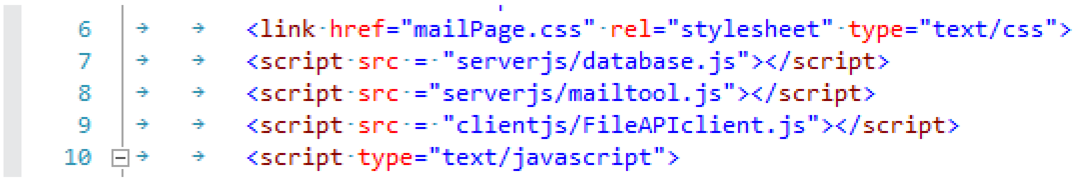
A typical process is as following.

1. Click “Load HTML file” button. It will read JavaScript include files contained in the test case HTML file. You will see a debug window showing file reading via a PHP program, readfile.php. If this step is successful then a drop-box lists all the JavaScript include files loaded.



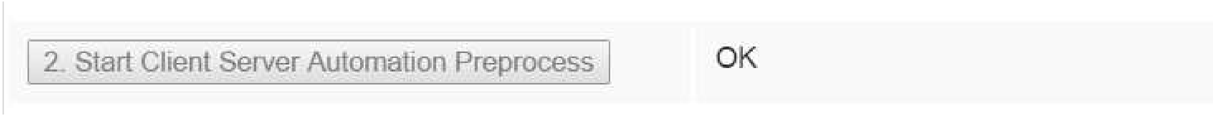
| Action | Result |
|---|---|
| 1. Load HTML file | Loaded code: mailtool.js - loaded OK |
| 2. Start Client Server Automation Preprocess | |

The above example shows the loading of following JavaScript inclusions in a test case HTML file:



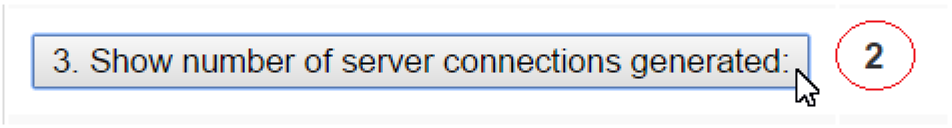
```
        <link href="mailPage.css" rel="stylesheet" type="text/css">
        <script src = "serverjs/database.js"></script>
        <script src ="serverjs/mailtool.js"></script>
        <script src = "clientjs/FileAPIclient.js"></script>
        <script type="text/javascript">
```

2. Click “Start Client Server Automation Preprocess” button. CSA preprocessing starts. If it is successful then an “OK” message is displayed.

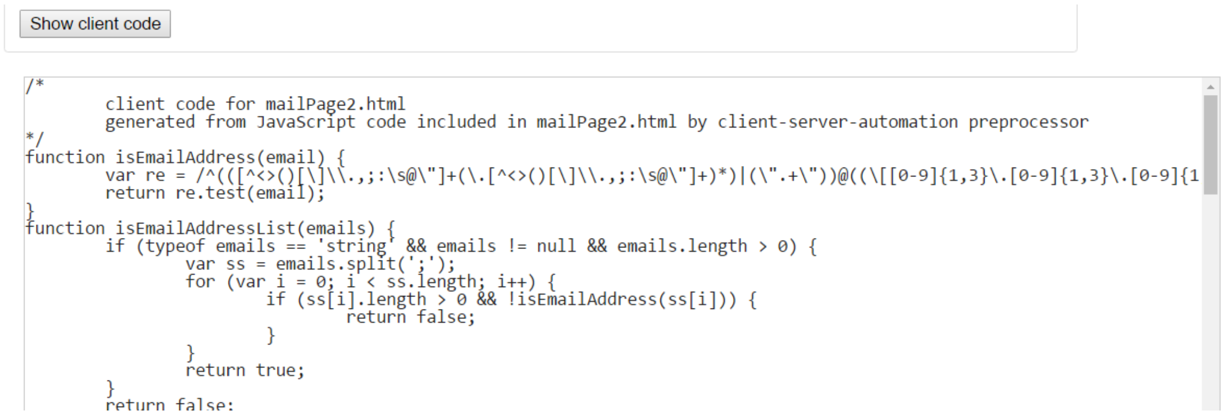


| | |
|---|---|
| 2. Start Client Server Automation Preprocess | OK |

3. Click “Show number of server connections generated” button to see if desired number of server connections are generated.

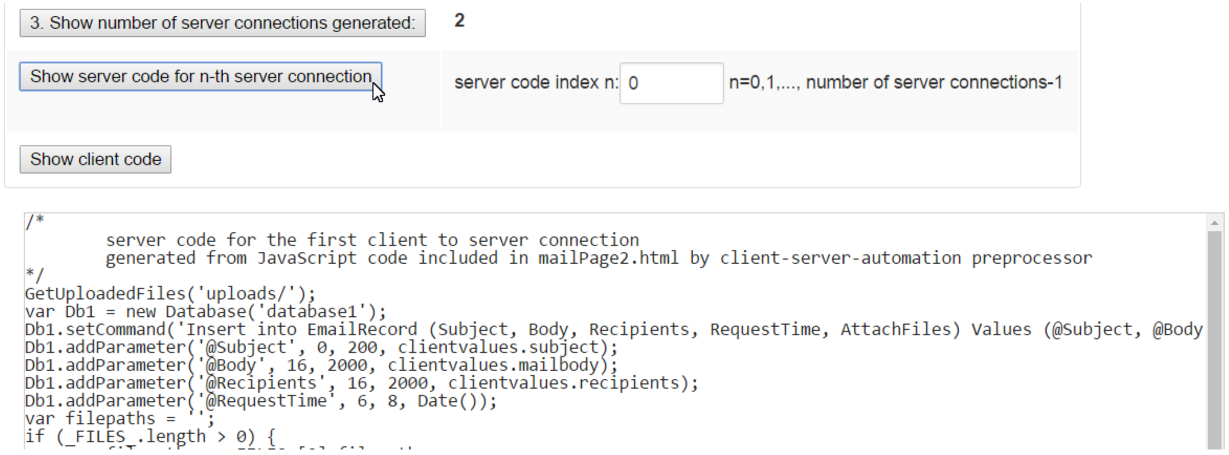


4. A box shows client side JavaScript code generated. Examine it to see if it is what you expected.



```
/*
        client code for mailPage2.html
        generated from JavaScript code included in mailPage2.html by client-server-automation preprocessor
*/
function isEmailAddress(email) {
        var re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1
        return re.test(email);
}
function isEmailAddressList(emails) {
        if (typeof emails == 'string' && emails != null && emails.length > 0) {
                var ss = emails.split(';');
                for (var i = 0; i < ss.length; i++) {
                        if (ss[i].length > 0 && !isEmailAddress(ss[i])) {
                                return false;
                        }
                }
                return true;
        }
        return false;
```

5. To see server side JavaScript code generated, enter a connection index between 0 and the number of server connections, click a button “Show server code for n-th server connection”. Examine the code to see if it is what you expected.

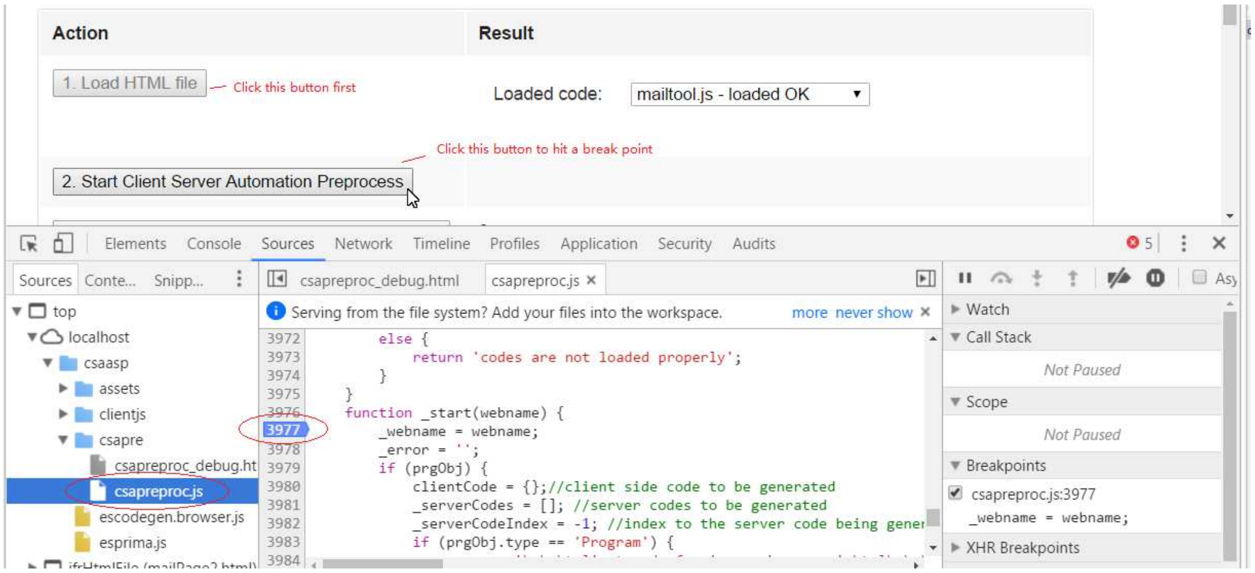

3. Show number of server connections generated:
2
Show server code for n-th server connection
server code index n: 0
n=0,1,..., number of server connections-1
Show client code

```
/*
        server code for the first client to server connection
        generated from JavaScript code included in mailPage2.html by client-server-automation preprocessor
*/
GetUploadedFiles('uploads/');
var Db1 = new Database('database1');
Db1.setCommand('Insert into EmailRecord (Subject, Body, Recipients, RequestTime, AttachFiles) Values (@Subject, @Body
Db1.addParameter('@Subject', 0, 200, clientvalues.subject);
Db1.addParameter('@Body', 16, 2000, clientvalues.mailbody);
Db1.addParameter('@Recipients', 16, 2000, clientvalues.recipients);
Db1.addParameter('@RequestTime', 6, 8, Date());
var filepaths = '';
if (_FILES_.length > 0) {
```

## Debug CSA Preprocessor

If the client side code or the server side code are not what we expected then we may start debugging csapreproc.js to see where the code does not do what we expected.

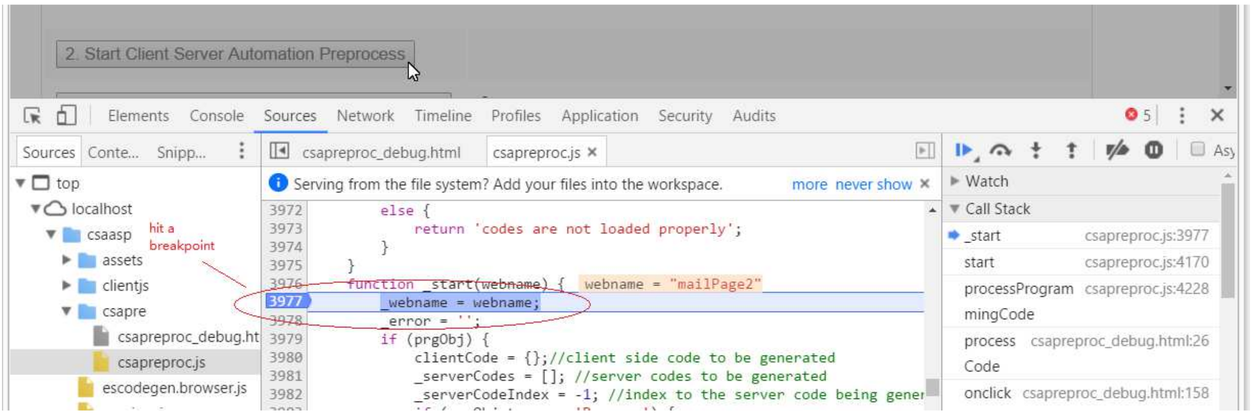
Step 2 of the previous section is calling csapreproc.js to do CSA preprocessing. We may set breakpoints in csapreproc.js so that when button “Start Client Server Automation Preprocess” is clicked, we can go step by step in csapreproc.js.



Serving from the file system? Add your files into the workspace. more never show

```
        else {
            return 'codes are not loaded properly';
        }
    }
    function _start(webname) {
        _webname = webname;
        _error = '';
        if (prgObj) {
            clientCode = {};//client side code to be generated
            _serverCodes = []; //server codes to be generated
            _serverCodeIndex = -1; //index to the server code being gener
            if (prgObj.type == 'Program') {
```

csapreproc.js:3977
_webname = webname;



Serving from the file system? Add your files into the workspace. more never show

```
        else {
            return 'codes are not loaded properly';
        }
    }
    function _start(webname) {  webname = "mailPage2"
        _webname = webname;
        _error = '';
        if (prgObj) {
            clientCode = {};//client side code to be generated
            _serverCodes = []; //server codes to be generated
            _serverCodeIndex = -1; //index to the server code being gener
```

```
_start          csapreproc.js:3977
start           csapreproc.js:4170
processProgram  csapreproc.js:4228
mingCode
process   csapreproc_debug.html:26
Code
onclick   csapreproc_debug.html:158
```

## Code Structure of CSA Preprocessor

The JavaScript file csapreproc.js implements core functionality of CSA Preprocessing. It goes through JavaScript code to be processed line by line to identify client/server related tasks and weave such tasks into the JavaScript code to be generated. This section describes code structure of csapreproc.js to help you debug it and modify it.

A closure named `CSAPREPROC` is created in csapreproc.js to encapsulate core functionality of CSA preprocessing. `CSAPREPROC` includes variables, constants and functions.

### Constants

- CSAVERSION – it is CSA Preprocessor version
- CLIENT_CONTEXT, SERVER_CONTEXT – represent execution context: client side execution or server side execution
- CODEBRANCH_UNKNOWN, CODEBRANCH_1, CODEBRANCH_2, CODEBRANCH_3, CODEBRANCH_4, CODEBRANCH_5, CODEBRANCH_6 – types of “if” branches

## Variables

- libObjs – objects loaded from JavaScript libraries
- libFileToObjectMapping – a mapping between an object and a JavaScript file defining the object
- prgObj – JavaScript code to be processed
- _webname – test case HTML file name without extension
- clientCode – client side JavaScript code generated
- _serverCodeIndex – index of server connection, it can be 0, 1, …, number of server connections - 1
- _serverCodes – server side JavaScript code generated
- _error – error message if processing fails

## Interface Functions

- csaPreprocessorVersion – returns CSA Preprocessor version
- loadCode – load JavaScript library code or JavaScript code to be processed
- start – start CSA preprocessing. It should only be called after using "loadCode" to load libraries and programming code to be processed
- getServerCodeCount – get number of server connections generated by "start" function
- `getServerCode – get server code generated`
- `getClientCode – get client code generated`

## Entry point

Function "_start" defined in `CSAPREPROC` starts CSA preprocessing. It goes through JavaScript code represented by variable "prgObj", line by line, to generates client side JavaScript code represented by variable "clientCode" and server side JavaScript code represented by an array "_serverCode".

This entry function treats programming code to be processed as contents of a function body, calls a function "`processFunctionBody`" to process each line of the programming code.

## Process function body

The function "`processFunctionBody`" goes through each line of code, uses a "switch..case" to pick certain types of statements and handle them in some specific ways; other types of statements are not handled and simply copied to the generated client side code.

Probable bugs:

- A type of statements is not handled, but the type of statements needs specific processing. To generate desired code, add a new "case" to pick the type of statements and do special handling. Usually this modification will not damage existing test cases.
- A type of statements is already picked up by a "case", but the handling does not generate desired code. To generate desired code, do minimum modifications of the handling code to make it generate desired code. It is likely to damage existing test cases. Run the test utility to see if any existing test cases are damaged.

## Execution scope

A variable named “scope” is used to represent all levels of execution scopes. This variable can have a “scope” property to represent a lower level scope. It is passed into most processing functions.

When a new lower level of execution scope is encountered, use a function “`appendscope`” to add a new “scope” property to the lowest level of “scope” property of the “scope” variable. The function “`appendscope`” returns the original lowest level “scope” property. A typical usage of using “appendscope” in csapreproc.js is shown below.

```
var s = appendscope(scope); //s.scope is the newly added scope property

…processing code in the lower scope…

delete s.scope; //remove the scope property on finishing processing code in the scope
```

During processing of the code, a scope may include following properties.

- `currentContext` – indicate the execution context to be CLIENT_CONTEXT or SERVER_CONTEXT
- `callback – true: the scope is for a callback function`
- `asyncFuncs – an array of function names of those functions using server values`
- `uploads – an array of upload value names`
- `serverVariables – it holds all server variables, each property name is a variable name, each property has an init property, which is the variable initializer.`
- `servercall – code to make a server connection`
- `servercallIndex – index of the server connection`
- `servercode – code assumed to be in server execution context`
- `func – a function declaration object generated from the original function declaration. Its params property can be used to determine object types. For example, if there is only one parameter then the parameter’s dataTransfer.files property can be assumed to be of type FileList.`
- `variables – it holds all variables in the scope, each variable becomes a property of “variables”, each property may have following properties. Each property of the “variables” is created by processing a “VariableDeclaration” statement by a function “registerVarToScope”.`
    - `isFileList – it indicates that the variable is an instance of FileList.`
    - `isHtmlElement` – it is an HTML element
    - `isDataTransfer` – it is a dataTransfer variable
    - `isParam` – it is function parameter

You may add new properties to a scope to help processing code.

## Execution context switching

The “`currentContext`” property of a scope indicates the scope’s execution context; it can be in a CLIENT_CONTEXT or in a SERVER_CONTEXT.

Function “_start” creates the top level scope and set its “`currentContext`” to CLIENT_CONTEXT.

While processing each line of code, if a server operation is detected then a new scope is created and its "currentContext" is set to SERVER_CONTEXT.

While in SERVER_CONTEXT, if a client operation is detected then a new scope is created and its "currentContext" is set to CLIENT_CONTEXT.

If you do not get expected separation of client side code and server side code then you may check places where detection of execution context switching should take place.

For example, if "currentContext" is CLIENT_CONTEXT and function "isServerFunction" return true then a client-to-server execution context switching is detected.

## Upload and download values

When generating server code, client values used in the server code need to be uploaded. Function "collectUploads" collects upload values; it is called when generating a server connection.

When generating a client callback function for a server connection, server values used at client side should be downloaded. Function "findDownloadAndStateValues" collects download values and state-values when generating server code.

State-values are server values of one server code and used in server codes of subsequent server connections. We assume web servers are stateless and maintain state-values at client side. State-values are downloaded to client side and then uploaded.

If you do not see expected download and upload generated then debug functions "collectUploads" and "findDownloadAndStateValues" to make corrections.