1. The Purpose

The purpose of this project is to give the opportunity to test ZK applications without the necessity of a web server as well as to allow automated testing (e.g. Hudson) and the use of code analyzing tools such as cobertura, findbugs, etc.

1. The Problem

The application uses and works in a web server environment. The server forwards requests and responses to the servlet, which processes them. If a control (ZK component) is created, it will be impossible to perform a test with it, because each attempt to access the component’s functionality will lead to a crash. The reason is that the proper work of a component is performed in the execution thread only.

1. The Solution

This project creates a servlet runner, implemented especially for the ZK Layout and the Update servlets. Two wrappers which send requests to the servlets are implemented. This allows the ZK framework to operate as if it is on a real web server. The servlet runner emulates the web server.

1. The servlet runner

The main components of the servlet runner are:

* GuiEnvironment - The class used for the servlets management is org.tsc.emulation.GuiEnvironment. It is used to create the ServletContext and ServerConfig, as well as to manage the emulation of a client browser (org.tsc.emulation.Client)
* TestServletContext – this class emulates the ServletContext, loaded from the web server. Only the methods, used by the ZK framework are implemented.
* TestServletConfig – this class emulates the ServletConfig, loaded from the web server for each particular client session. Only the methods, used by the ZK framework are implemented.
* TestSession – this class emulates the HttpSessoion, which is loaded from the web server for each particular client session. Only the methods, used by the ZK framework are implemented.
* Client – this is an emulator of a client session. It is used to create a session, to send requests to the servlets and to receive access to Desktop, to Page, to the loaded through the Layout servlet page and to the created non embedded components.
* TestRequest – implements the HttpServletRequest. It is used to send ZK events to the servlets
* TestResponse – implements the HttpServletResponse
* TestLayoutServlet – a wrapper for DhtmlLayoutServlet. It emulates a request to a new page. After sending the request, the created Desktop is taken and assigned to the corresponding client.
* TestUpdateServlet – a wrapper for DhtmlUpdateServlet. This object can send requests to the ZK framework. It is also used to control the “in event listener” mode, which is necessary for manipulating the ZK components (mainly because of the smartUpdate operation)
* EventExecution – an auxiliary class. Allows sending specific events from a component to the ZK framework. The EventMapper auxiliary class is used to transform the ZK events to parameters of the HttpServletRequest.
* ExecutionEmulator – an auxiliary class. Allows creating components through the Executions.createComponents() or Executions.createComponentsDirectly() methods.
* ServerPush – emulator of the ServerPush part, which is executed in the client browser. The emulator is an additional thread which sends requests to the Update servlet, according to the type of ServerPush mechanism used. It is started when a client emulator is created and is stopped by its deactivation. The Polling and Comet ServerPush mechanisms are currently implemented.

**Servlets management**. The management of the servlets is performed via the static methods of class GuiEnvironment.

* Start: when method GuiEnvironment.create(webPath) is called:
  + a ServletContext with assigned path to the web directory is created;
  + TestLayout and TestUpdate Servlets are created and initialized.

It is mainly called in the setUpClass() methods of the tests.

* Restore the initial state: method restore(). Deregisters and deactivates all client sessions except the default one (if present) and restores the default condition of the default client. It is mainly called in the tearDown() method of test files.
* Stop: method destroy(). Deregisters and deactivates all client emulators and destroys the servlets.

**Client session management.**

* Start: the client session is activated via the instantiation of a Client and the call to the method *public HttpServletResponse create(String page)*. When instantiated, a new TestSession is created. This TestSession cannot be changed for the concrete client and is used by every request, sent from this client. The next step is to send a request to the TestLayoutServlet for creating the page, passed as a parameter. The servlet returns the Desktop, which was created during the execution. The Desktop is registered in the client. When the Desktop is already created, the emulator of the browser part of ServerPush is started.

When the client is started, it is also registered in the GuiEnvironment, where a relation Desktop Id – Client is created. In that way the client can be found on the base of the Desktop ID. If the testing is performed only with one client emulator (which happens in most cases), a Default Client can be created with the method create(String webPath, String page) when the GuiEnvironment is created. As a convenience, parts of the Client methods are moved to the GuiEnvironment as static methods.

* Restore the initial state: method restore(). Detaches all non embedded components, deletes the elements, waiting for download and deletes the session’s cookies
* Deactivation: method exit(). Restores the initial state, sends request to the Update Servlet to destroy the Desktop and deregisters itself from the GuiEnvironment.

**Sending request:** Sending of a request is performed via some of the executeCommand methods of the Client. Client creates a request and response, sets the parameters of the request and sends it via the TestUpdateServlet. If the sent command is an Event, then the request is transformed in the auxiliary class EventMapper to a Map, containing parameters. The typical way of sending events is via the EventExecution. The Desktop of the component, “sending” the request is taken. The Client of this Desktop is found via the GuiEnvironment. Finally, the request is sent via the Client, operating with this Desktop/component.

**Event Listener activation:** If some operation with the ZK components must be performed in event listener mode during the test, this mode can be activated/deactivated through the methods enter(client)/leave(client) of TestUpdateServlet. The method free() deactivates the event listener mode of the current thread. The active client for the current thread can be obtained through the getActiveClient() method.

**Access to the information, sent to the browser:** If it is necessary to track the information, sent from the servlets to the browser when sending a request, a TestResponse is used when response is received. TestResponse implements a method getWriterString() through which the string, sent to the browser can be read from the response.

**Upload/Download:** Additional functionalities of the web server are used by upload in ZK. That is why this operation is not performed in the way, standard for the servlet. The upload in the test environment consists in direct posting of UploadEvent in Execution. This is performed throughTestUpdateServlet.upload() method. Also the download cannot be fully emulated, that is why all instances of org.zkoss.util.media.Media, cached in the Desktop instance are taken through the method getAllDownloads() in Client.

**Component wrapper:** As a convenience, wrappers of the components are created. They take care of sending the proper events during testing (e.g. by setValue() of Textbox first ON\_CHANGING and then ON\_CHANGE event is sent). Another role of the wrapper is to prevent sending events, when the component cannot send events to the client browser (e.g. if invisible, read-only or deactivated). Some controls (such as Tree) have access methods to the components auxiliary classes, which ease the job during testing. All wrappers have a constructor with a parameter – an instance of the component, supported by them. They also have a constructor with an instance of spaceOwner and ID of the component, which must be wrapped.

**Component creation during a test:** in order to create a component during a test, it is necessary to activate Execution of the current thread. The execution must be ended after the methods *Executions.createComponents()/Executions.createComponentsDirectly()* are called. ExecutionEmulator is used for this purpose. If the component, which is created, is located in a file or is an application resource, some of the createComponent methods of the class can be used directly. If this is not enough, the *Executions.createComponents()/Executions.createComponentsDirectly()* of ZK can be used in a manner, analogical to *createComponent(Client client, String uri, Component parent, Map arg)*, but instead of *ComponentCreateUtil.createComponent(uri, parent, arg)*, the necessary method of Executions is used.

**Additional configurations/own implementations:** The functionality and implementations present in the library may be not enough for testing a certain program, but most of the implementations can be changed by overriding methods. For example, if another implementation of ServletContext, ServletConfig for Update or Layout servlet is required, the corresponding methods can be overridden in GuiEnvironmentImpl. For different implementation of HttpSession, HttpServletRequest or HttpServletResponse, the methods in Client can be overridden.

EXAMPLES:

* Servlets management – test file GuiEnvironmentTest
  + To create and initialize test environment, call the static method

*GuiEnvironment.create("/Demo/web");*

where “Demo/web” is the relative path to the web directory. By calling the test method testCreateAndDestroy(), an instance of the environment and the two servlets is created and the servlets are initialized.

* + To destroy the test environment, method GuiEnvironment.destroy() is called

By calling testGuiEnvironmentDestroy() the client Desktop, servlets and test environment are destroyed

* + Test environment restore – the purpose of this functionality is to close all clients, except the default one, as well as to close all dialog windows. This action is performed by calling the static method GuiEnvironment.restore()

By calling testGuiEnvironmentRestore() the Desktops of all client emulators are destroyed and the clients are deregistered from the GuiEnvironment.

* Clients management – test file GuiEnvironmentTest
  + Creating a client. A client can be created in two manners:
* Creating a default client when a test environment is created. It is the easier way of manipulating, because once created the client is destroyed when the test environment is destroyed. It allows easier access to some functionality, directly via the static methods of GuiEnvironment. The creation of such client is performed through the method *GuiEnvironment.create("/Demo/web", "index.html")*. The client can be directly accessed through the call of the method *GuiEnvironment.getDefaultClient()*
* Another way to create a client directly is the following:

Client client = new Client();

client.create("index.html");

In this way clients for each test can be created (or clients working in parallel). An example for such client’s creation can be seen in the test *testClientCreateAndDestory()*.

* + Destroy a client. To destroy a client the method *client.exit();* is called.

The call to this method causes the destruction of all components, created during the test, deletion of the Map, containing the downloads, deletion of the COOKIES for the session, stopping the client part, which supports the serverPush processing, release of the EventProcessor, in case of opened modal windows. It also sends a request to the update servlet for closed window and deregisters itself from the GuiEnvironment. If the client is created together with the GuiEnvironment, the client’s exit() method is called automatically by destruction of the GuiEnvironment through GuiEnvironment.destroy();

* + Restoring a client. It is performed by a call to *client.restore()* method.

This method call evokes destruction of all components, created during the test for this client, deletion of the Map, containing the downloads and deletion of the COOKIES for the session.

* Access to the main control – test file GuiEnvironmentTest

When a client is created, a request to the Layout servlet is sent, which evokes creation of a new Desktop for the client. The control, which was created on page opening, can be obtained through the call of client.getMainControl()

An example for using this method is the test testMainControlAccess(). First, a client for the index.html page is created. The control of type Html, which is created from the Layout servlet, is obtained by calling client.getMainControl(), after which a test for the links in the page is performed.

* Sending requests – To demonstrate the way in which requests are sent to the Update servlets the file SendRequest.zul is created. It contains a window and a textbox. When the value in the textbox is changed, this value is set as a title of the window.

The test file, containing the examples is SendRequestTest. Before running the tests, a test environment without a default client is created.

|  |
| --- |
| @BeforeClass  public static void setUpClass() throws Exception {  GuiEnvironment.create("/Demo/web");  } |

A new client for the page "/samples/SendRequest.zul" is created before running of each test. The window and textbox, which participate in the test, are taken from this new Client.

|  |
| --- |
| Client \_client = null;  Textbox \_textbox = null;  Window \_mainControl = null;  ….....  @Before  public void setUp() throws EmulationException {  \_client = new Client();  \_client.create("/samples/SendRequest.zul");  \_mainControl = (Window) \_client.getMainControl();  \_textbox = (Textbox) \_mainControl.getFellow("textbox");  } |

The restore() method of GuiEnvironment is called when the test completes, which destroys the Client (it is necessary to call it explicitly because we are not working with the default client)

|  |
| --- |
| @After  public void tearDown() throws Exception {  GuiEnvironment.restore();  } |

The test environment is destroyed after completing all tests

|  |
| --- |
| @AfterClass  public static void tearDownClass() throws Exception {  GuiEnvironment.destroy();  } |

There are four methods for sending requests to the Update Servlets

* + Sending requests directly via the update servlets. For this purpose instances of HttpServletRequest and HttpServletResponse must be created. They should have parameters, required by the DhtmlUpdateServlet. All necessary parameters for the DhtmlUpdateServlet are passed to the constructor of TestRequest. In order to send a request these parameters must be initialized correctly. The sending of the request is performed through the static method

get(HttpServletRequest request, HttpServletResponse response) of TestUpdateServlet.

An example of the use of such a method is displayed below:

|  |
| --- |
| @Test  public void testSendRequestViaUpdateServlet() throws ServletException, IOException{    // prepare request parameters  String desktopId = \_client.getDesktop().getId();  String componentId = \_textbox.getUuid();  String command = Events.ON\_CHANGE;  // create the request parameter values  Map values = new HashMap();  values.put("value", "test1");    // create the request and responce  TestRequest request = new TestRequest(\_client, desktopId, componentId, command, values);  TestResponse response = new TestResponse(\_client);    // check window title before request  assertEquals("Simple Window", \_mainControl.getTitle());    // send request  TestUpdateServlet.get(request, response);    // check window title after request  assertEquals("test1", \_mainControl.getTitle());  } |

* + Sending request through a client emulator. A more convenient way to send a request is to use the method executeCommand of the Client instance. In this case the client emulator takes care of creation of HttpServletRequest and HttpServletResponse. The Desktop ID is passed as a request parameter, as the client works with one Desktop only. Here is an example how to use this method:

|  |
| --- |
| @Test  public void testSendRequestViaClient() throws EmulationException{  // prepare request parameter values  Map values = new HashMap();  values.put("value", "test2");  // check window title before request  assertEquals("Simple Window", \_mainControl.getTitle());  // send request  \_client.executeCommand(\_textbox, Events.ON\_CHANGE, values);  // check window title after request  assertEquals("test2", \_mainControl.getTitle());  } |

* + Sending a request via the auxiliary class EventExecution. The most convenient way for sending requests directly is using the EventExecution. This class is created in order to simplify sending of standard requests from the components. All methods take as a parameter the component, which sends the request. For a desktop is considered the Desktop, to which the component is attached. The client, through which the event must be sent, is taken from the GuiEnvironment on the base of the desktop ID. If the operation requires additional parameters, they must be also set when the method is called. Example:

|  |
| --- |
| @Test  public void testSendRequestViaEventExecution() throws EmulationException{  // check window title before request  assertEquals("Simple Window", \_mainControl.getTitle());  // send request  EventExecution.executeChange(\_textbox, "test3");  // check window title after request  assertEquals("test3", \_mainControl.getTitle());  } |

* + Sending request via the emulator (wrapper) of the component. To make the work with the operations, performed by a certain component easier, emulators of the components are created. They can be created in two ways:
* Via a constructor with one parameter – an instance of the emulated component or
* Via a constructor with two parameters – a parent component (which must be in the same spaceOwner) and the Id of the component.

The purpose of the emulators is to reproduce the functioning of the elements. This functioning must approach as much as possible the real way by which the elements function in the browser as well as to allow direct access to the auxiliary functionality.

An example for functioning in the browser is the Textbox element. If it is read-only, disabled or invisible it is not allowed to change its value. By value change, onChanging event is sent first, and then OnChange, in exactly the same way as the actual component does in the browser. An example of using auxiliary functionalities is the Grid, which implements the methods:

|  |
| --- |
| public int size()  public String[] header()  public Column getColumn(int index)  public Column getColumnByLabel(String label)  public Row getRow(int index)  public String printString() |

An additional advantage of the emulators is that they give opportunity to send only those requests, which are supported by the corresponding component.

Here is an example for sending requests via the emulator:

|  |
| --- |
| @Test  public void testSendRequestViaWrapper() throws EmulationException{  // create the textbox emulator  org.tsc.emulation.components.Textbox emulator = new org.tsc.emulation.components.Textbox(\_textbox);  // alternative create method  org.tsc.emulation.components.Textbox alternativeEmulator = new org.tsc.emulation.components.Textbox(\_mainControl, "textbox");    // check window title before request  assertEquals("Simple Window", \_mainControl.getTitle());  // send request  emulator.setValue("test4");  // check window title after request  assertEquals("test4", \_mainControl.getTitle());  // send request over the alternative emulator  alternativeEmulator.setValue("test5");  // check window title after request  assertEquals("test5", \_mainControl.getTitle());  } |

* Work in event listener mode – test file WorkInEventListenerTest.

In some cases the work with the events, sent via the update servlet, is not enough. It may be necessary some values, which require work in event listener mode, to be changed. There is an opportunity to manage the event listener mode for the current (test) thread. Examples of such cases are the direct setting of a value or execution of some function which requires UiVisualizer or Execution.

To manage the event listener mode of the current thread, three methods in the TestUpdateServlet are implemented:

|  |
| --- |
| public static void enter(Client client);  public static void leave(Client client);  public static void free();  public static Client getActiveClient(); |

The getActiveClient() method returns the client, for which the current thread is in an event listener mode. If such client does not exist, returns null.

The free() method deactivates the event listener mode for the active client, if such client exists.

The leave() method evokes deactivation of the event listener mode for a given client, independently of the thread, which activated it.

The enter() method activates the event listener mode for the client for the current thread. If there is another client, who is active at the moment, he will be deactivated before the activation of the new client. If there is an active client and the method is called again for the same client, the event listener mode for the client will be deactivated and then activated again. By deactivating of the event listener mode all events, created during work after activation of the event listener mode and waiting in the Execution in order to be executed will be processed.

Examples:

Example of an operation, which requires UiVisualizer is the call of method setValue() of textbox, which evokes the execution of smartUpdate. setValue() causes IllegalStateException (see the test below)

|  |
| --- |
| @Test(expected = java.lang.IllegalStateException.class)  public void testSmartUpdateOutOfEventListener() throws EmulationException {  // smart update cause IllegalStateException  \_textbox.setValue("test1");  } |

On the other hand, if we activate event listener mode, the use of this method does not cause an Exception

|  |
| --- |
| @Test  public void testSmartUpdateInEventListener() throws EmulationException {  // activate the event listener  TestUpdateServlet.enter(\_client);  // now there no exeption  \_textbox.setValue("test2");  //leave the event listener mode for this thread  TestUpdateServlet.free();  } |

Example of events, waiting for execution in Execution. For the purpose of the test, the AutomatedGrid control, which extends Grid, is created. The call of changeSource(List source) method causes substitution of the Grid’s columns, replacing the RowRenderer according to the type of the elements in the source and reloading the ListModel of the grid. The grid works in three different ways according to the type of the elements in the source. We use the assumption that all elements are similar to the first one.

Let the first element be:

An array – columns('position 1', 'position 2',....'position n') . RowRenderer – AutomatedGrid$ArrayRowRenderer

An object with get methods – columns – names of the getters. RowRenderer – AutomatedGrid$ObjectRowRenderer

An object without get methods – one column with text – name of the object’s class. RowRenderer – AutomatedGrid$DefaultRowRenderer

Peculiarity in the use of this component is that by calling the changeSource() method, events are pushed in the Execution queue. These events will be processed after the thread leaves the event listener mode. For the example a list with three integer elements is passed as a source. Before the thread leaves and reenters the event listener mode via the method

TestUpdateServlet.enter(\_client); the table rows are created, but not rendered. After deactivation of the client the processing of the command (and the waiting events) is finished and the table is fully loaded.

|  |
| --- |
| @Test  public void testPendingEventsInExecutionQueue() throws EmulationException, Exception {  TestUpdateServlet.enter(\_client);// enter the event listener    // Create the instance of the object  AutomatedGrid instance = new AutomatedGrid();  instance.setPage(\_client.getPage());  \_client.ping(); // finalize execution of the event listener and get back  assertEquals("", GridUtil.printString(instance, true));  List<Integer> list = Arrays.asList(new Integer[]{2,5,9,0});  instance.changeSource(list);// Build the header, set the row renderer and update the model    // the grid rows are created but the renderer is not called because  // the events wait in the Execution queue  assertEquals("Integer", GridUtil.printString(instance, true));    // finalize the event listener execution and enter it again  TestUpdateServlet.enter(\_client);    // now the grid is loaded and all items are rendered  assertEquals("Integer\n2\n5\n9\n0", GridUtil.printString(instance, true));  } |

* Access to the servlet response – test file BrowserOutputTest.

The method getWriterString() in TestResponse can be used in case that the test needs to track the text, which the server sends to the browser. TestResponse is returned in all ways of sending request, listed above, except when using emulators.

Response by creating a client. In this test the received response is checked partially, because the generation of IDs and UIDs cannot be provided.

|  |
| --- |
| @Test  public void testLayoutServletOutput() throws ServletException, IOException, EmulationException {  Client client = new Client();  TestResponse response = (TestResponse) client.create("/samples/SendRequest.zul");  String result = response.getWriterString();  assertTrue(result.contains("['zul.inp.Textbox','"));  } |

Response after sending request to the Update servlet:

|  |
| --- |
| @Test  public void testUpdateServletOutput() throws EmulationException {  Client client = new Client();  client.create("/samples/SendRequest.zul");  Window window = (Window) client.getMainControl();  Textbox textbox = (Textbox) window.getFellow("textbox");  TestResponse response = (TestResponse) EventExecution.executeChange(textbox, "test1");  assertEquals("{\"rs\":[[\"setAttr\",[\"" + window.getUuid() + "\",\"title\",\"test1\"]]],\"rid\":3}", response.getWriterString());  } |

* Upload/Download – test file UploadDownloadTest

For the purpose of the test the control UploadDownload.zul is created, which uses Composer UploadDownload. This control allows pictures’ upload. If the file, which must be uploaded, is not a picture, a Message box is displayed. After uploading the picture the download button is activated and the value “Download” + file\_name is assigned to its Label. The picture can be already downloaded.

Unlike the execution of the requests, which pass the full path of processing of the servlet, the execution of upload operation is impossible in such manner, because it requires additional functionality of the web server. That is why the created Media is posted directly in the Execution and from this point forward the work of the servlet continues in its normal way. This is enough for tests execution. The same problem occurs when testing the download. By sending a request, which activates the download, the download Media is cached in the client’s Desktop. In order to stream it to the client, web server functionality is necessary. That is why the following method is created in the client emulator:

|  |
| --- |
| public Map getAllDownloads() throws EmulationException |

Which returns a reference to the Map in which the Desktop caches the downloads.

Example for using upload and download:

The file, used for upload is the picture m1.gif, which is a test resource. The execution of the command EventExecution.executeUpload(upload, uploadMedia); causes the visualization of the download button and sets its label to “Download m1.gif”, as would happen if uploaded through a browser.

Clicking on the download button evokes the addition of a new element in the download cache of the desktop. The result from the data comparison is that the download and upload medias has exactly the same content, but are actually two different objects.

|  |
| --- |
| @Test  public void testUploadDownload() throws Exception {  assertEquals(false, download.isVisible());  AImage uploadMedia = new AImage("m1.gif", getClass().getClassLoader().getResourceAsStream("m1.gif"));  EventExecution.executeUpload(upload, uploadMedia);    assertEquals(true, download.isVisible());  assertEquals("Download m1.gif", download.getLabel());    assertEquals(0, GuiEnvironment.getAllDownloads().size());  EventExecution.executeClick(download);  assertEquals(1, GuiEnvironment.getAllDownloads().size());    AMedia downloadMedia = (AMedia) GuiEnvironment.getAllDownloads().values().toArray()[0];  assertEquals(uploadMedia.getName(), downloadMedia.getName());  assertEquals(uploadMedia.getContentType(), downloadMedia.getContentType());  assertEquals(uploadMedia.getFormat(), downloadMedia.getFormat());  byte[] uploaded = uploadMedia.getByteData();  byte[] downloaded = uploadMedia.getByteData();  assertArrayEquals(uploaded, downloaded);  assertNotSame(uploaded, downloaded);  } |

* Component wrappers – The purpose of using emulators of the components is described above. As an addition, the creation of an emulator of the user control, which is tested, is described. The advantages of this techniques are the following:
  + Writing of more effective tests with less code
  + Encapsulation of functionalities, specific for the controller (Example: ChatEmulator.java)
  + By change of the elements or the structure, the adjustment is performed only in one place.

For the purpose of the test a control GridSample.zul is created, containing window, which uses the base class GridSample.java. In this control the AutomatedGrid, which is described above, is used. By selecting one of the radio buttons in the control, the source of the Grid is changed. When the emulator is created, two constructors are defined. Apart from the typed constructor which receives an instance of the tested component, another one, which creates an instance through Executions, is created. The GridSample emulator has two methods which return emulators of Grid and Radiogroup (the two components are placed in the control). To create a Grid its constructor with parameters a parent object and component ID is used:

|  |
| --- |
| new Grid(INSTANCE, "grid"); |

where INSTANCE is an instance of the emulated object and “grid” is the ID of the grid. The radiogroup does not have an ID. In order to get this group, all elements are walked.

The tests for controls, which use emulators is much more simple and clear. Here is an example:

|  |
| --- |
| @Test  public void testWithEmulatorComponent() throws Exception{  GridSampleEmulator instance = new GridSampleEmulator();    assertEquals(FileUtil.getFileContent("./test\_resources/emulator/emptyGrid.txt"), instance.getGrid().printString());    instance.getRadio().checkByLabel("Array");  assertEquals(FileUtil.getFileContent("./test\_resources/emulator/array-1-20.txt"), instance.getGrid().printString());  instance.getRadio().checkByLabel("Integer");  assertEquals(FileUtil.getFileContent("./test\_resources/emulator/integer-1-20.txt"), instance.getGrid().printString());  instance.getRadio().checkByLabel("UUID");  assertEquals(FileUtil.getFileContent("./test\_resources/emulator/uuid-1-20.txt"), instance.getGrid().printString());  } |

Another way to work with the emulators is to use the getFellow(String id) or getChildren() method of Component emulator(class org.tsc.emulation.components.Component). These methods return directly emulators of the components via ComponentWrapperUtil.

|  |
| --- |
| @Test  public void testWithoutControlEmulator() throws EmulationException {  Client client = new Client();  client.create("/samples/SendRequest.zul");  Window<org.zkoss.zul.Window> emulator = new Window((org.zkoss.zul.Window) client.getMainControl());  ((Textbox) emulator.getFellow("textbox")).setValue("test");  assertEquals("test", emulator.INSTANCE.getTitle());  } |

If the emulators from the project do not work in the way, required in a test, some functionality is missing or they are not implemented, an own implementation of an emulator can be performed. The emulator must be then registered for the concrete component. In the example test the emulator MyTextboxEmulator of the component Textbox is implemented. The setValue() method of the emulator does nothing. This emulator registers itself in ComponentWrapperUtil. After the test finishes, it deregisters itself, so that it won’t affect the other tests. As you can see from the example below, the setValue() method of the emulator does not send a request to the Update servlet and the window’s title remains unchanged.

|  |
| --- |
| @Test  public void testCustomComponentEmulator() throws EmulationException {  Client client = new Client();  client.create("/samples/SendRequest.zul");  ComponentWrapperUtil.registerCustomWrapper(org.zkoss.zul.Textbox.class, MyTextboxEmulator.class);  try {  Window<org.zkoss.zul.Window> emulator = (Window) ComponentWrapperUtil.wrapComponent(client.getMainControl());  ((MyTextboxEmulator) emulator.getFellow("textbox")).setValue("test");  assertEquals("Simple Window", (emulator.INSTANCE).getTitle());  } finally {  ComponentWrapperUtil.unregisterCustomWrapper(org.zkoss.zul.Textbox.class);  }  } |

* ServerPush – test file ServerPushTest

The chat application, described here: <http://richjava.wordpress.com/2010/11/04/zk-chat/> is used in order to test the functionality of ServerPush. For more convenience an emulator of chat.zul – ChatEmulator.java is made. Apart from getters of component emulators, ChatEmulator implements the following methods:

* public ChatRoom getChatRoom() – returns the ChatRoom, in which the client is registered.
* waitForLogin(String username) – waits for 15 seconds a client with the passed username to register itself in the ChatRoom. If this does not happen, an AssertionFailedError is thrown.
* public void waitForLogout(String username) – waits for 15 seconds a client with the passed username to deregister itself in the ChatRoom. If this does not happen, an AssertionFailedError is thrown.
* public void waitToBroadcastAllMessages() – waits until ServerPush sends all messages to all clients. If this does not happen within 15 seconds, an AssertionFailedError is thrown.
* login(String username) – sets the username in nameTb textbox and clicks the LoginBtn button. After that is waits for user’s registration in the ChatRoom and for sending the subsequent message to all other clients.
* public void logout() – clicks the ExitBtn button, waits for user’s deregistration from the ChatRoom and for sending the subsequent message to all other clients.
* public void sendMessage(String message) – sets a value in MsgTv and sends ON\_OK event to the window. After that waits this message to be received from all registered users.

There is one additional class, used for the test. This is ParalelChatter, which is a separate thread, executing the commands login(String name), logout() and sendMessage(String message) in a way, analogical to ChatEmulator. The difference is that the commands are sent not from the test thread, but from the ParalelChatter thread.

Ways of testing:

The first way test ServerPush is to create a client and then to send a command for client notification manually. An example for this is testSingleThreadExternalBroadcast(). After the client is created and is logged in the ChatRoom, the following method is called:

|  |
| --- |
| instance.getChatRoom().broadcast("none", "hello"); |

which broadcasts the message to the client through ServerPush.

|  |
| --- |
| @Test  public void testSingleThreadExternalBroadcast() throws Exception {  ChatEmulator instance = new ChatEmulator((Window) GuiEnvironment.getMainControl());  assertTrue(instance.loginVisible());  instance.login("me");  instance.waitForLogin("me");  assertFalse(instance.loginVisible());  assertEquals("~~~Welcome me~~~\n", instance.getText());  instance.getChatRoom().broadcast("none", "hello");  instance.waitToBroadcastAllMessages();  assertEquals("~~~Welcome me~~~\nhello\n", instance.getText());  instance.logout();  instance.waitForLogout("me");  assertTrue(instance.loginVisible());  assertEquals("", instance.getText());  } |

The second way to test ServerPush is to switch the test thread from one to the other client emulator. In the test bellow the requests to the Update servlet are sent through one thread only, which in this case does not influence the functioning of the controls at all.

|  |
| --- |
| @Test  public void testParalelWithSingleThread() throws Exception {  Client mainClient = GuiEnvironment.getDefaultClient();  Client otherClient = new Client();  otherClient.create("/chat/chat.zul");  ChatEmulator main = new ChatEmulator((Window) mainClient.getMainControl());  ChatEmulator other = new ChatEmulator((Window) otherClient.getMainControl());  main.login("main");  assertEquals("~~~Welcome main~~~\n", main.getText());  // second thread login  other.login("other");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\n", main.getText());  assertEquals("~~~Welcome other~~~\n", other.getText());  // main chatter send message  main.sendMessage("hello other");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\nmain:hello other\n", main.getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\n", other.getText());  // other send message  other.sendMessage("hi");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\nmain:hello other\nother:hi\n", main.getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\nother:hi\n", other.getText());  // main leave the chat  main.logout();  assertEquals("", main.getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\nother:hi\n~~~main has left the chat room~~~\n", other.getText());  // other leave the chat  other.logout();  assertEquals("", other.getText());  } |

The third way to test ServerPush is by using fully independent threads.

|  |
| --- |
| @Test  public void testParalelWithMultipleThreads() throws Exception {  ParalelChatter mainThread = new ParalelChatter();  mainThread.start();  ParalelChatter otherThread = new ParalelChatter();  otherThread.start();  mainThread.login("main");  assertEquals("~~~Welcome main~~~\n", mainThread.getEmulator().getText());  // second thread login  otherThread.login("other");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\n", mainThread.getEmulator().getText());  assertEquals("~~~Welcome other~~~\n", otherThread.getEmulator().getText());  // main chatter send message  mainThread.sendMessage("hello other");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\nmain:hello other\n", mainThread.getEmulator().getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\n", otherThread.getEmulator().getText());  // other send message  otherThread.sendMessage("hi");  assertEquals("~~~Welcome main~~~\n~~~other has joined this chatroom~~~\nmain:hello other\nother:hi\n", mainThread.getEmulator().getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\nother:hi\n", otherThread.getEmulator().getText());  // main leave the chat  mainThread.logout();  assertEquals("", mainThread.getEmulator().getText());  assertEquals("~~~Welcome other~~~\nmain:hello other\nother:hi\n~~~main has left the chat room~~~\n", otherThread.getEmulator().getText());  otherThread.logout();  mainThread.exit();  otherThread.exit();  } |

* creating a component with Executions during a test – test file CreateComponentTest

The need to create a new control during a test arises very often. For this purpose the auxiliary class ExecutionEmulator is created. The control with which the client works is “index.html”, created in setUpClass(). With the help of ExecutionEmulator, the control is created in spite of the functionalities of the MainControl.

|  |
| --- |
| @Test  public void testCreateComponentWithExecutionEmulator() throws EmulationException {  Window instance = (Window) ExecutionEmulator.createComponent("samples/SendRequest.zul", null, null);    assertTrue(GuiEnvironment.getCreatedComponents().contains(instance));  EventExecution.executeChange(instance.getFellow("textbox"), "test1");  assertEquals("test1", instance.getTitle());  } |

This can be also performed manually. In this case however we must also manage the event listener mode on our own.

|  |
| --- |
| @Test  public void testCreateComponentWithExecutions() throws EmulationException{  Client activeClient = TestUpdateServlet.getActiveClient();    TestUpdateServlet.enter(GuiEnvironment.getDefaultClient());    Window instance = (Window) Executions.createComponents("samples/SendRequest.zul", null, null);    TestUpdateServlet.free();  if (activeClient != null){  TestUpdateServlet.enter(activeClient);  }  assertTrue(GuiEnvironment.getCreatedComponents().contains(instance));  EventExecution.executeChange(instance.getFellow("textbox"), "test1");  assertEquals("test1", instance.getTitle());  } |

* Own implementations of the test environment – test file CustomEnvironmentTest

If the elements of the Environment are not enough for testing a certain project, the elements of the servlet-api can be overridden or replaced with own implementations. Example for own elements:

The following elements are created in the test package 'cusomenvironment'

* MySession – overrides TestSession and implements the functionality

public long getCreationTime();

* MyServletContext – overrides TestServletContext by using a default constructor, which sets the path to the web directory “/Demo/web” and the log level to WARNING;
* MyUpdateConfig and MyLayoutConfig, overriding TestServletConfig and implementing the method getServletName();
* MyRequest – overriding TestRequest. This implementation prints in System.out a dump of the request’s parameters when the request is created;
* MyResponse – overrides TestResponse. This implementation of HttpServletResponse uses as a PrintWriter a writer with an output stream – System.out. The data appears directly in System.out when the servlet starts writing in the response;
* In order to use the new implementations of MyRequst, MyResponse and MySession, Client must be also overridden. For this purpose the class MyClient is created. During testing it is used instead of Client;
* In order to use the new implementations of MyServletContext, MyUpdateConfig and MyLayoutConfig, the class MyEnvironmentImpl, which extends GuiEnvironmentImpl is implemented;
* In order to use MyEnvironmentImpl and to create a default client of type MyClient, MyEnvironment must be used instead of GuiEnvironment. If MyEnvironment is used during testing, the new servlet-api implementations will be used.

An example for using of this servlet version is the test file CustomEnvironmentTest.

Creating test environment:

|  |
| --- |
| @BeforeClass  public static void setUpClass() throws Exception {  MyEnvironment.create("/Demo/web", "index.html");  } |

It is not necessary to make anything additional in order to work with the new implementations of the servlet-api. MyRequst and MyResponse print in System.out

|  |
| --- |
| @Test  public void testSendRequest() throws EmulationException {  Window sendRequest = (Window) ExecutionEmulator.createComponent("samples/SendRequest.zul", null, null);  assertTrue(GuiEnvironment.getCreatedComponents().contains(sendRequest));  EventExecution.executeChange(sendRequest.getFellow("textbox"), "test1");  assertEquals("test1", sendRequest.getTitle());  } |

The call of

|  |
| --- |
| EventExecution.executeChange(sendRequest.getFellow("textbox"), "test1"); |

produces the following output:

|  |
| --- |
| -----------------------------------------------------  Request : Tread main Session session-0 desktop:z\_wu00 command:onChange component [uid:xYREg id:textbox Textbox]  values[start:0, value:test1, bySelectBack:false]  ...  Response : Tread main Session session-0  {"rs":[["addAft",["xYRE0",[  ['zul.wnd.Window','xYREf',{$$onSize:false,$$onMaximize:false,$$onOpen:false,$$onMinimize:false,$$onZIndex:false,$onClose:true,$$onMove:false,width:'200px',height:'80px',style:'padding:10px',prolog:'\n ',title:'Simple Window',position:'center',border:'normal'},[  ['zul.inp.Textbox','xYREg',{id:'textbox',$$onError:false,$$onChange:false,$onChange:true,width:'150px',prolog:'\n '},[]]]]]]],["setAttr",["xYREf","title","test1"]]],"rid":3} |

Be careful and use only MyClient instead of a Client during testing.

|  |
| --- |
| @Test  public void testUploadDownload() throws EmulationException, IOException {  MyClient client = new MyClient();  client.create("samples/UploadDownload.zul");  Window uploadDownload = (Window) client.getMainControl();  EventExecution.executeUpload(uploadDownload.getFellow("upload"), new AImage("m1.gif", getClass().getClassLoader().getResourceAsStream("m1.gif")));  EventExecution.executeClick(uploadDownload.getFellow("download"));  assertEquals(1, client.getAllDownloads().size());  // check if composer used  assertEquals(UploadDownload.class, uploadDownload.getAttribute("$composer").getClass());  } |

* Access to MessageBox and detached dialogs

MessageBoxEmulator is created in order to work with MessageBox during testing. The access to the opened MessageBox dialog during testing is done with the method MessageBoxEmulator.get(client) or MessageBoxEmulator.get(). The second one is used if the client is the default one for the GuiEnvironment. The call to one of these methods searches an instance of MessageboxDlg in root elements of the Page and returns the emulator of the dialog, which was found first.

|  |
| --- |
| @Test  public void testMessageBox() throws Exception {  Client client = new Client();  client.create("samples/UploadDownload.zul");  Button upload = (Button) client.getMainControl().getFellow("upload");  Button download = (Button) client.getMainControl().getFellow("download");    AMedia uploadMedia = new AMedia("empty", null, null, getClass().getClassLoader().getResourceAsStream("empty"));  EventExecution.executeUpload(upload, uploadMedia);    assertEquals(false, download.isVisible());    MessageBoxEmulator msg = MessageBoxEmulator.get(client);  assertEquals("Not an image: empty", msg.getMessage());  msg.getOK().click();  assertTrue(msg.isClosed());  } |

The access to components, using the “use” clause in the zul file (e.g. GridSample.zul) is performed in a similar way. In the test testGetComponentByClass an instance of this control is created and is not attached anywhere. After that the first detached component which implements GridSample is taken by calling GuiEnvironment.getComponent(GridSample.class).

|  |
| --- |
| @Test  public void testGetComponentByClass() throws Exception {  // create not attached window  GridSample gridSample = (GridSample) ExecutionEmulator.createComponent("samples/GridSample.zul", null, null);    // find first detached component instance of GridSample  GridSample instance = (GridSample) GuiEnvironment.getComponent(GridSample.class);  assertEquals(gridSample, instance);  } |

It is a little bit more complicated to find a component, which does not use the “use” clause. In this case a rule must be defined, which determines if a certain control is the one we are looking for. This can be its title, id, value, etc. To simplify the testing an emulator of the component, containing a static method, implementing the search logic can be created (e.g. GridSampleEmulator.getIfCreated(client)).

In the test below a popup of type UploadDownload is created. The search criteria for the control, which must be found is to be an instance of Window and to have as an attribute a "$composer" object from class UploadDownload (the Composer for UploadDownload.zul). After that we search all detached components for a one, which fulfils the criteria.

|  |
| --- |
| @Test  public void testFindComponent() throws EmulationException, IOException {  // create a popup upload download window  TestUpdateServlet.enter(GuiEnvironment.getDefaultClient());  Window uploadDownload = (Window) ExecutionEmulator.createComponent("samples/UploadDownload.zul", null, null);  uploadDownload.doPopup();  TestUpdateServlet.leave(GuiEnvironment.getDefaultClient());  // search the UploadDownload window in detached components  Window instance = null;  for (Component cmp : GuiEnvironment.getCreatedComponents()) {  if (cmp instanceof Window  && cmp.getAttribute("$composer") != null  && cmp.getAttribute("$composer").getClass() == UploadDownload.class) {  instance = (Window)cmp;  break;  }  }  assertEquals(uploadDownload, instance);  } |