Getting started with

Daraja HTTP Framework

3.1.0

Contents

Tutorial

Creating a “Hello, World!” application

The following short tutorial takes you through some of the basic steps of developing a “Hello, World!” example. This tutorial assumes you already have some familiarity with developing Delphi applications.

You will create an application that runs a HTTP server on the local computer and serves requests for the resource `http://127.0.0.1/tutorial/hello`.

This tutorial takes approximately 10 minutes to complete.

To complete this tutorial, you need the software and resources listed in the following table.

- Daraja HTTP Framework 3.1.0
- Indy 10.6
- Delphi 2009

Project Setup

The application you create will contain one Delphi project, which is a console program.

- in the IDE, use the project wizard to create a new console line application project
- add the path to the `<inst>\source` folder to the project search path
- save the project as `HelloWorldServer`

Creating the main unit

The main unit declares a web component, and a Demo procedure which configures and starts the server.

- create a unit with the code below and save it as `MainUnit.pas`

Code example

```
unit MainUnit;

interface

procedure Demo;
```

```
implementation

uses
  djWebComponent, djServer, djWebAppContext, djTypes;

type
  THelloWorldResource = class(TdjWebComponent)
  public
    procedure OnGet(Request: TdjRequest; Response: TdjResponse); override;
  end;

procedure THelloWorldResource.OnGet(Request: TdjRequest; Response: TdjResponse);
begin
  Response.ContentText := 'Hello, World!';
  Response.ContentType := 'text/plain';
end;

procedure Demo;
var
  Server: TdjServer;
  Context: TdjWebAppContext;
begin
  Server := TdjServer.Create(80);
  try
    Context := TdjWebAppContext.Create('tutorial');
    Context.Add(THelloWorldResource, '/hello');
    Server.Add(Context);
    Server.Start;
    WriteLn('Server is running, please open http://127.0.0.1/tutorial/hello');
    WriteLn('Hit enter to terminate.');
    ReadLn;
  finally
    Server.Free;
  end;
end;
```

Create the server configuration and start-up code

- open the `HelloWorldServer.dpr` application project file
- paste the code shown below

Code example

```
program HelloWorldServer;

{$APPTYPE CONSOLE}

uses
  MainUnit in 'MainUnit.pas';

begin
  Demo;
end.
```

Compiling and Running the Application

- In the IDE, press F9 to start the application

Note: depending on your system, a firewall warning can appear, notifying you that the program tries to open a server port.

Testing the Application

To test the server, open a web browser and navigate to `127.0.0.1/tutorial/hello`.

This will cause a HTTP GET request to be sent from the web browser, asking for the resource.

1. The server will parse the request, look up the given context `tutorial` and the web component responsible for this resource. It will find that `THelloWorldResource` is mapped to the path '/hello'.
2. The server then will invoke the GET request handler `THelloWorldResource.OnGet`, passing request and response objects.
3. Finally, the request handler code builds the response, which will be sent back to the web browser.

Terminating the HTTP server

To shut down the HTTP server, activate the console application window and press the enter key.

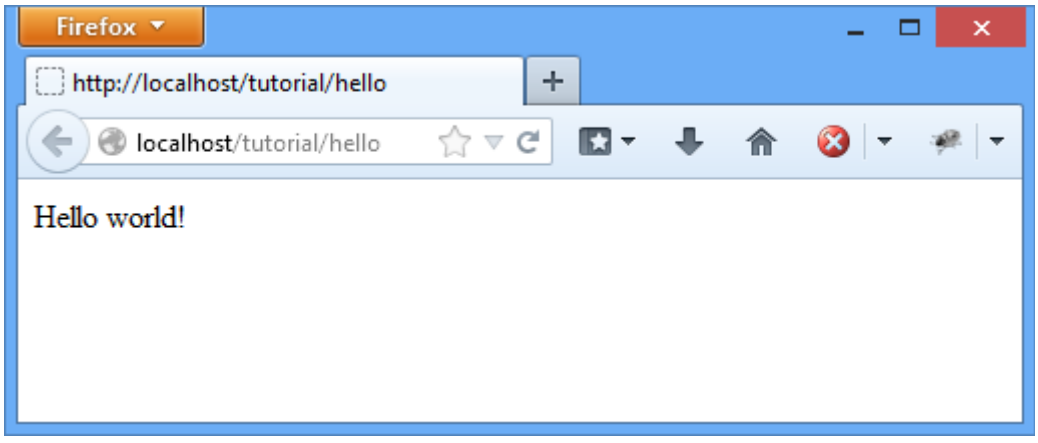

Illustration 1: Hello world example

Developing an HTTP Session application

The following short tutorial takes you through some of the basic steps of developing an example for HTTP sessions with the Daraja HTTP Framework framework.

This tutorial assumes you already have some familiarity with developing Delphi applications.

You will create an application that runs a HTTP server on the local computer and serves requests for the resource `http://127.0.0.1/tutorial/session`.

This tutorial takes approximately 10 minutes to complete.

To complete this tutorial, you need the software and resources listed in the following table.

- Daraja HTTP Framework 3.1.0
- Indy 10.6
- Delphi 2009

Project Setup

The application you create will contain one Delphi project, which is a console program.

- in the IDE, use the project wizard to create a new console line application project
- add the path to the `<inst>\source` folder to the project search path
- save the project as `HttpSessionServer`

Creating the Web Component

- create a unit with the code below and save it as `SessionDemoResource.pas`

Code example

```
unit SessionDemoResource;

interface

uses djWebComponent, djTypes;

type
  TSessionDemoResource = class(TdjWebComponent)
  public
    procedure OnGet(Request: TdjRequest; Response: TdjResponse); override;
  end;

implementation

uses SysUtils;

procedure TSessionDemoResource.OnGet(Request: TdjRequest; Response: TdjResponse);
var
```

```
  RequestCountForSession: string;
begin
  RequestCountForSession := Request.Session.Content.Values['count'];
  if RequestCountForSession = '' then RequestCountForSession := '1';

  Request.Session.Content.Values['count'] :=
    IntToStr(StrToInt(RequestCountForSession) + 1);

  Response.ContentText :=
    Format('Your Session ID is %s ', [Request.Session.SessionID]) + #10 +
    Format('I have received %s GET Requests during this session', [RequestCountForSession]);

  Response.ContentType := 'text/plain';
end;

end.
```

Create the server configuration and start-up code

- open the HttpSessionServer.dpr application project file
- paste the code shown below

Code example

```
program HttpSessionServer;

{$APPTYPE CONSOLE}

uses
  SessionDemoResource in 'SessionDemoResource.pas',
  djServer, djWebAppContext;

procedure Demo;
var
  Server: TdjServer;
  Context: TdjWebAppContext;
begin
  Server := TdjServer.Create(80);
  try
    Context := TdjWebAppContext.Create('tutorial', True);
    Context.Add(TSessionDemoResource, '/session');

    Server.Add(Context);
    Server.Start;

    WriteLn('Hit enter to terminate.');
    ReadLn;
  finally
    Server.Free;
  end;
end;

begin
  Demo;
end.
```

Compiling and Running the Application

- In the IDE, press F9 to start the application

Note: depending on your system, a firewall warning can appear, notifying you that the program tries to open a server port.

Testing the Application

To test the server, open a web browser and navigate to http://127.0.0.1/tutorial/session.

This will cause a HTTP GET request to be sent from the web browser, asking the server for the resource at this location.

1. The server will parse the request, look up the context ('tutorial') and the web component responsible for this resource. It will find that `TSessionDemoResource` is mapped to the path '/session'.
2. The server then will invoke the GET request handler `TSessionDemoResource.OnGet`, passing request and response objects.
3. Finally, the GET request handler builds the response, which will be sent back to the web browser.

Terminating the HTTP server

To shut down the HTTP server, activate the console application window and press the enter key.

Illustration 2: HTTP session example

Multiple Url Mappings of a Web Component

The following short tutorial takes you through some of the basic steps of creating a Daraja HTTP Framework framework application which uses multiple mappings to deliver different HTTP response content types depending on the HTTP request document path.

You will create an application that runs a HTTP server on the local computer and serves requests for two resource paths.

The response content type will depend on the request:

- for http://127.0.0.1/tutorial/fib.txt, the content type is `text/plain`
- for http://127.0.0.1/tutorial/fib.html, the content type is `text/html`

The application will read a query parameter with the name "n" to calculate the Fibonacci number for the parameter value.

A complete request URL will be for example

- http://127.0.0.1/tutorial/fib.html?n=8

This tutorial takes approximately 10 minutes to complete.

To complete this tutorial, you need the software and resources listed in the following table.

- Daraja HTTP Framework 3.1.0
- Indy 10.6
- Delphi 2009

Project Setup

The application you create will contain one Delphi project, which is a console program.

- in the IDE, use the project wizard to create a new console line application project
- add the path to the `<inst>\source` folder to the project search path
- save the project as `TwoMappingsServer`

Creating the Web Component

- create a unit with the code below and save it as `FibonacciResource.pas`

Code example

```
unit FibonacciResource;

interface

uses djWebComponent, djTypes;

type
  TFibonacciResource = class(TdjWebComponent)
  public
    procedure OnGet(Request: TdjRequest; Response: TdjResponse); override;
  end;

implementation

uses StrUtils, SysUtils;
```

```
function fib(n: Integer): Integer;
begin
  if n=0 then begin Result := 0; Exit; end;
  if n=1 then begin Result := 1; Exit; end;
  Result := fib(n-1) + fib(n-2);
end;

procedure TFibonacciResource.OnGet(Request: TdjRequest; Response: TdjResponse);
const
  INVALID_ARGUMENT_VALUE = -1;
var
  InputParam: Integer;
begin
  InputParam := StrToIntDef(Request.Params.Values['n'], INVALID_ARGUMENT_VALUE);
  if InputParam <= INVALID_ARGUMENT_VALUE then begin
    Response.ResponseNo := 500;
    Response.ContentText := 'Internal server error: missing or invalid value';
    Response.ContentType := 'text/plain';
  end else if EndsText('.txt', Request.Document) then begin
    Response.ContentText := IntToStr(fib(InputParam));
    Response.ContentType := 'text/plain';
  end else if EndsText('.html', Request.Document) then begin
    Response.ContentText := Format('<html><body>Result: <b>%d</b></body></html>',
[fib(InputParam)]);
    Response.ContentType := 'text/html';
  end;
end;

end.
```

Create the server configuration and start-up code

- open the TwoMappingsServer.dpr application project file
- paste the code shown below

Code example

```
program TwoMappingsServer;

{$APPTYPE CONSOLE}

uses
  djServer, djWebAppContext,
  FibonacciResource in 'FibonacciResource.pas';

procedure Demo;
var
  Server: TdjServer;
  Context: TdjWebAppContext;
begin
  Server := TdjServer.Create(80);
  try
    Context := TdjWebAppContext.Create('tutorial');
    Context.Add(TFibonacciResource, '/fib.txt');
    Context.Add(TFibonacciResource, '/fib.html');
```

```
    Server.Add(Context);
    Server.Start;
    WriteLn('Hit enter to terminate.');
    ReadLn;
  finally
    Server.Free;
  end;
end;

begin
  Demo;
end.
```

Compiling and Running the Application

- In the IDE, press F9 to start the application

Note: depending on your system, a firewall warning can appear, notifying you that the program tries to open a server port.

Testing the Application

To test the server, open a web browser and navigate to http://127.0.0.1/tutorial/fib.txt?n=7. This will cause a HTTP GET request to be sent, asking for the resource http://127.0.0.1/tutorial/fib.txt, and passing the query parameter n with value 7.

The server will parse the request, look up the context ('tutorial') and the web component responsible for this resource address. It will find that `TFibonacciResource` is mapped to the absolute path '/fib.txt'.

The server then will invoke the GET request handler `TFibonacciResource.OnGet`, passing request and response objects.

Finally, the code in `TFibonacciResource.OnGet` builds the response, which will be sent back to the web browser.

Illustration 3: Fibonacci calculation result as plain text

To receive the HTTP response as HTML, use the address http://127.0.0.1/tutorial/fib.html?n=7 – the same web component is mapped to the absolute path /fib.html. This time, the

code in OnGet will detect the .html extension and return a response with content type `text/html` and a simple HTML body back to the browser.

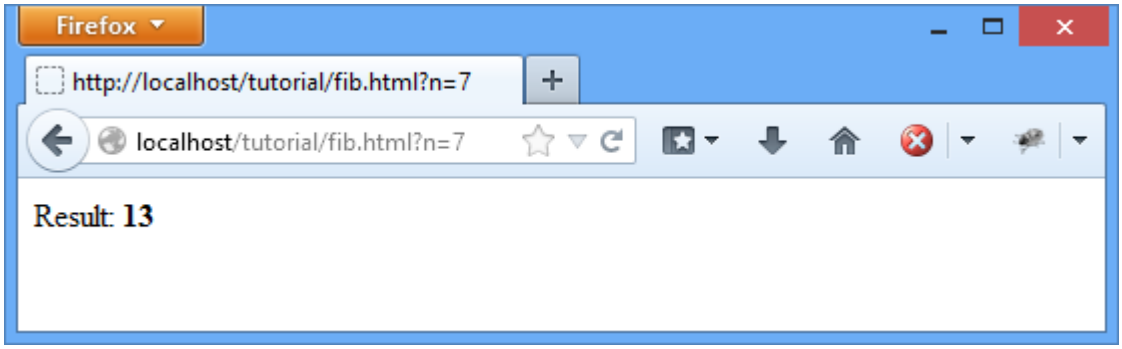

Illustration 4: Fibonacci calculation result as HTML

Terminating the HTTP server

To shut down the HTTP server, activate the console application window and press the enter key.

Installation

Requirements

Development Environment

- **Embarcadero Delphi** 2009 (Update 4)
  or
- **Free Pascal** 3.2.0 or newer
- **Internet Direct (Indy)** 10.6.2

IDE and project configuration

Option 1: configure source code paths

To make Daraja HTTP Framework and Internet Direct (Indy) available for a project,

- add the Daraja HTTP Framework `<Install>/source` folder to the project search path
- add the folders `<Indy>/Lib/Core`, `<Indy>/Lib/System` and `<Indy>/Lib/Protocols` to the project search path
- add the folder `<Indy>/Lib/Core` to the project include path

Note: Installation of the Indy packages in the IDE is not required

Option 2: compiled units

A second option which avoids repeated recompilation is to use compiled DCU files, and include their location in the library path.

Notes:

- if compiler settings change, the library units must be recompiled

Official Indy installation instructions

If you prefer a full installation of Indy packages in the IDE, please read the official documentation page at

http://www.indyproject.org/Sockets/Docs/Indy10Installation.aspx

Conditional Symbols

Conditional symbol	Description
DARAJA_LOGGING	Enables logging
DARAJA_PROJECT_STAGE_DEVELOPMENT	Enables development mode
DARAJA_MADEXCEPT	Enables exception stacktraces with madExcept, this requires development mode
DARAJA_JCLDEBUG	Enables exception stacktraces with JclDebug, this requires development mode

Table 1: Conditional symbols

Introduction to Web Components

General structure

A Web Component has two minimum requirements:

1. it must be a subclass of TdjWebComponent
2. it must override at least one of the predefined HTTP method handlers – OnGet, OnPost etc.

It may also optionally override two other methods

- the Init method, to perform initialization tasks
- the destructor (Destroy), to perform cleanup tasks

Example

A minimal example is shown below.

- It overrides **OnGet**, which has two parameters for the Indy HTTP Request and Response objects.
- In the method body, it sets the **ContentText** property of the Response object to '<html>Hello world!</html>':

Code example

```
type
  TMyWebPage = class(TdjWebComponent)
  public
    procedure OnGet(Request: TdjRequest; Response:
      TdjResponse); override;
  end;

implementation

{ TMyWebPage }

procedure TMyWebPage.OnGet(Request: TdjRequest;
  Response: TdjResponse);
begin
  Response.ContentText := '<html>Hello world!</html>;
  Response.ContentType := 'text/html';
  Response.CharSet := 'utf-8';
end;
```

Web Application Context

In a Daraja HTTP Framework web application, a **context** is an object that is created when the web application is started and destroyed when the web application is taken out of service.

The context object can contain initialization parameters which can be accessed from web components.

A context can provide **dynamic** and **static** resources:

- **Dynamic** resources are created by **Web Components** when a client sends a request which matches their *resource path mapping*
- **Static** resources are located in the *file system*. A special web component in the framework handles requests for static resources. It also *caches* requests to save bandwidth

The context path must be set in the constructor and can not be changed later.

Code example

```
Context := TdjWebAppContext.Create('demo');
```

By default, web components can not use HTTP sessions.

A second parameter optionally enables HTTP sessions for the context.

Code example

```
Context := TdjWebAppContext.Create('demo', True);
```

Path Mapping

Mapping rules

- the framework uses **path mappings** to find the matching Web Component for a request URL and executes its On... method handler
- if no match is found, and a default handler has been installed, the framework will try to serve the request using a **static resource**
- if no static file exists, the framework will return a 404 error

Supported Mapping Syntax

Supported mapping styles in order of priority are

- absolute paths, for example '/mypage.html'
- prefix mappings, for example '/myfolder/subfolder/*'
- suffix mappings, for example '*.html' or '*.page'

If two web components declare overlapping mappings, they will be processed in the order of their priority.

Example

- WebCompontentA maps to '*.html'
- WebCompontentB maps to '/myfolder/*'

In this case, the resource /myfolder/index.html will be handled by WebComponentB because a prefix mapping (/myfolder/*) has a higher priority than an extension mapping.

Multiple mappings

Multiple mappings per component are supported. Example:

- WebCompontentA maps to '*.html', '*.doc' and '*.pdf'
- WebCompontentB maps to '/secure/*' and '/protected/*'

With this mapping, resource /context/secure/example.pdf will be handled by WebComponentB, and resource /context/example.pdf will be handled by WebCompontentA.

Setting parameters of a context

A context contains a key-value map which can be used to keep configuration information.

Code example

```
Context.SetInitParameter('key', 'value');
```

To access the context init parameter, use GetInitParameter

Code example

```
procedure TExampleComponent.OnGet(Request: TdjRequest; Response: TdjResponse);
var
  Param: string;
begin
  Param := Config.GetContext.GetInitParameter('key');
end;
```

Use **GetInitParameterNames** to get a list of all parameter names:

Code example

```
Config.GetContext.GetInitParameterNames;
```

Unicode (UTF-8)

By setting the Response ContentType and CharSet, the server can respond with Unicode content[1]:

Code example

```
procedure THelloPage.OnGet(Request: TdjRequest;
  Response: TdjResponse);
```

1 Response.ContentType should be set before Response.CharSet

```
begin
  Response.ContentText := '中文';
  Response.ContentType := 'text/plain';
  Response.CharSet := 'utf-8';
end;
```

HTML Encoding of special characters

If the web component returns HTML, some characters have to be replaced as they are HTML entities.

Unit **djGlobal** contains the **HTMLEncode** function which replaces <, > , & and " to the corresponding HTML entities **<**, **>**, **&** and **"**.

Important: only inner text (between HTML elements) must be encoded.

Code example

```
procedure THelloPage.OnGet(Request: TdjRequest;
  Response: TdjResponse);
begin
  Response.ContentText := '<html><p>' + HTMLEncode('rhythm & blues') + '</p></html>';
  Response.ContentType := 'text/html';
  Response.CharSet := 'utf-8';
end;
```

Introduction to Web Filters

General structure

A Web Filter class must fulfill two requirements:

- it must be a subclass of TdjWebFilter
- it must override at least the DoFilter method

It may also optionally override other methods:

- the Init method
- the destructor

Declaration of DoFilter

A web filter must be inherited from TdjWebFilter and at least implement the DoFilter method.

Example: add text to the response

In this example, the filter first invokes the DoFilter method of the filter chain, and finally appends a hard coded text to the response.

Code example

```
TTestFilter = class(TdjWebFilter)
public
  procedure DoFilter(Context: TdjServerContext; Request: TdjRequest;
    Response: TdjResponse; const Chain: IWebFilterChain); override;
end;

procedure TTestFilter.DoFilter(Context: TdjServerContext; Request: TdjRequest;
  Response: TdjResponse; const Chain: IWebFilterChain);
begin
  Chain.DoFilter(Context, Request, Response);
  Response.ContentText := Response.ContentText + ' (filtered)';
end;
```

Declaration of an initialization method

A web filter may optionally declare a Init method which will be called by the framework on startup.

Example: add an initialization parameter value to the response

The example filter below reads a value from the filter configuration, and stores it in a field for later usage.

Code example

```
TFilterWithInit = class(TdjWebFilter)
private
  FInitParam: string;
public
  procedure Init; override;
  procedure DoFilter(Context: TdjServerContext; Request: TdjRequest;
    Response: TdjResponse; const Chain: IWebFilterChain); override;
end;

procedure TFilterWithInit.Init;
begin
  FInitParam := Config.GetInitParameter('key');
end;

procedure TFilterWithInit.DoFilter(Context: TdjServerContext;
  Request: TdjRequest; Response: TdjResponse; const Chain: IWebFilterChain);
begin
  Chain.DoFilter(Context, Request, Response);
  Response.ContentText := Response.ContentText + 'Param key=' + FInitParam;
end;
```

The next code excerpt is a test which uses the filter and verifies that the GET request to the resource has the expected request body content.

Code example

```
procedure TAPIConfigTests.TestFilterWithInit;
var
  Server: TdjServer;
  Context: TdjWebAppContext;
begin
  // configure
  Context := TdjWebAppContext.Create('web');
  Context.AddWebComponent(TExamplePage, '*.filter');
  with Context.AddWebFilter(TTestFilterWithInit, '*.filter') do
  begin
    SetInitParameter('key', 'Hello, World V3!');
  end;

  // run
  Server := TdjServer.Create;
  try
```

```
    Server.Add(Context);
    Server.Start;
    CheckGETResponseEquals('example, Param key=Hello, World V3!', '/web/page.filter');
  finally
    Server.Free;
  end;
end;
```

Web Components and multi-threading

Design rules for your Web Components

Design for multi-threaded operation

When we say that a program is *multi-threaded*, we are not implying that the program runs two separate instances simultaneously (as if you concurrently executed the program twice from the command line). Rather, we are saying that the same instance (executed only once) spawns multiple threads that process this *single instance of code*. This means that more than one sequential flow of control runs through the same memory block.

When multiple threads execute a single instance of a program and therefore share memory, multiple threads could possibly be attempting to read and write to the same place in memory.

What happens if you introduce a field in your Web Component and use it in the OnGet (or OnPost) method, when two or more threads execute it at the same time? Look at the example below. If two threads execute OnGet, they both will read and increment the value of the private *MyVar* variable, with unexpected results.

Code example

```
type
  TMyWebPage = class(TdjWebComponent)
  private
    MyVar: Integer;
  public
    procedure OnGet(Request: TdjRequest; Response:
      TdjResponse); override;
  end;

implementation

{ TMyWebPage }

procedure TMyWebPage.OnGet(Request: TdjRequest;
  Response: TdjResponse);
begin
  WriteLn(MyVar);

  Inc(Counter);

  // ... other code

  WriteLn(MyVar);
end;
```

It would not be practical to build a site that required a Web Component to be instantiated for each request. Web Components are multi-threaded by design, this means a single instance will handle all HTTP requests.

The framework allocates a thread for *each new request* for a single Web Component without any special programming.

To avoid multithreading problems, only use *read-only* or *application-wide* variables in a Web Component.

To ensure we have our own unique variable instance for each thread, we also can simply move the declaration of the variable from within the class to within the method using it.

If you discover that you must share a variable between Web Components and this variable is going to be read from and written to by multiple threads (and you are not storing it in a database), then you will require thread synchronization.

Synchronization

Thread synchronization is an important technique to know, but not one you want to throw at a solution unless required. Anytime you synchronize blocks of code, you introduce bottlenecks into your system.

Under most circumstances, there is only one instance of your Web Component, no matter how many client requests are in process. That means that at any given moment, there may be many threads running inside the *Service* method of your solo instance, all sharing the same instance data and potentially stepping on each others toes. This means that you should be careful to **synchronize** access to shared data (instance variables).

Web Components thread implementation

Q: Are Web Components multi-threaded?

A: Yes, Web Components are normally multi-threaded. The Web Component server allocates a thread for each new request for a single Web Component without any special programming. Each request thread for your Web Component runs as if a single user were accessing it alone, but you can use static variables to store and present information that is common to all threads, like a hit counter for instance.

Q: Are you saying there is only one Web Component instance for all requests?

A: The way that the server handles requests is not prescribed to this extent; it may use a single Web Component, it may use Web Component pooling, it depends on the internal system architecture. New threads are not necessarily created for every Web Component request but may be recycled through a worker thread pool for efficiency. The point is that you should write your Web Component code to take account of a multi-threaded context regardless of the server implementation you happen to be using.

Q: Can my Web Component control the number of threads it accepts?

A: Your Web Component should be designed to be thread safe and not to anticipate any limit to the number of concurrent requests it will receive.

Appendix

Breaking Changes

Breaking Changes in v 3.1

- Minimum supported version of the slf4p logging facade is 1.0.8

Breaking Changes in v 3.0

- Remove deprecated methods
- Use port 8080 as by default instead of port 80

Frequently Asked Questions

How do I allow connections from other computers?

By default TdjServer binds to 127.0.0.1, which does not allow connections from other computers.

Solution: either bind the server to IP address 0.0.0.0, or to the IP address of a specific network adapter.

Code example

```
// allow incoming connections to network adapter 10.10.1.50
Server := TdjServer.Create('10.10.1.50', 8080);
```

Also be aware that the firewall must be configured to allow incoming connections.

Why does the server fail to run with “port already in use” error?

If a server application is already listening on the same port as your code, your program will not be able to open the same port.

Here are two ways find the process which opened the port:

Find the process by its process ID

Go to a command line and enter

```
netstat -o -n -a | findstr :8080
```

The last number in the output is the process ID using port 8080.

Example:

```
  TCP    127.0.0.1:8080         0.0.0.0:0              LISTEN         5636
```

Find the process name (requires administrator privileges)

Run the following from an elevated command prompt:

```
netstat -ab
```

Example output:

```
  ...
  TCP    0.0.0.0:135            my-PC:0             LISTEN
  RpcSs
 [svchost.exe]
  TCP    0.0.0.0:8080           my-PC:0             LISTEN
 [MyOtherServer.exe]
  ...
```

Solutions:

- stop the application or service with this process ID, and try again
- use a different port (for example 8081)

Related articles:

- http://stackoverflow.com/questions/20558410
- http://serverfault.com/questions/316514

Advanced Topics

Exception stack traces

Stack traces with madExcept

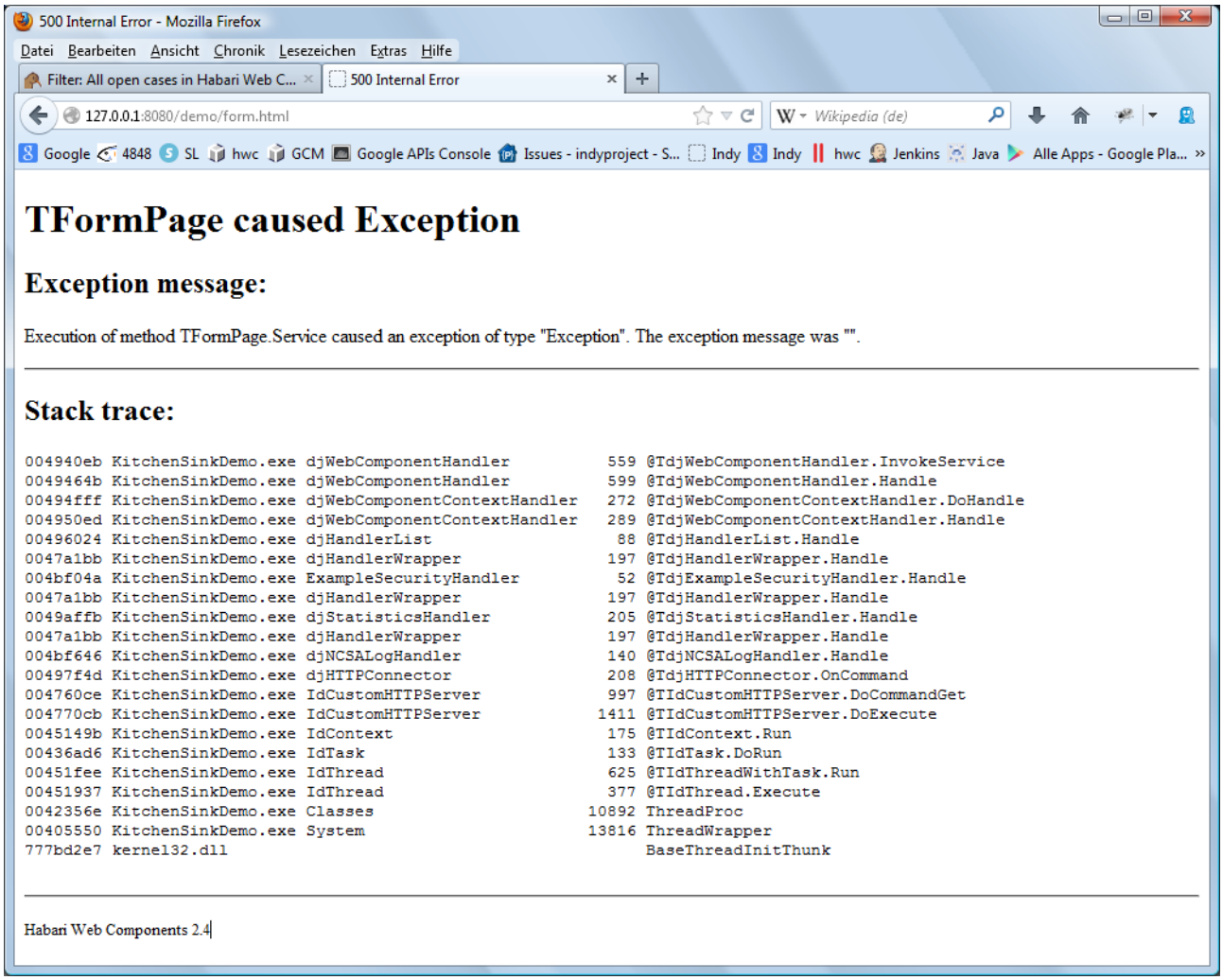

Illustration 5: Exception stack trace with madExcept

Declaring the conditional symbols `DARAJA_PROJECT_STAGE_DEVELOPMENT` and `DARAJA_MADEXCEPT` in the project settings will enable stack traces on exceptions. The stack traces will appear in the HTML response and – if `DARAJA_LOGGING` is defined - in the log output.

Note this is an unsupported bonus feature and only has been tested with madExcept version 3.

Stack traces with JclDebug

Declaring the conditional symbols `DARAJA_PROJECT_STAGE_DEVELOPMENT` and `DARAJA_JCLDEBUG` in the project settings will enable stack traces on exceptions. The stack traces will appear in the HTML response and – if `DARAJA_LOGGING` is defined – in the log output.

Note: this is an unsupported feature, tested with Jedi Code Library version 2.6.0.5178

Configuration of internal Indy HTTP Server

The new property HTTPServer of the class TdjHTTPConnector allows to query and modify properties of the internal Indy HTTP server component.

MaxConnections example

To reject new HTTP connections in high load situations, set the server property MaxConnections:

Code example

```
// allow a maximum of 100 concurrent connections
Connector.HTTPServer.MaxConnections := 100;
```

The library will log refused connections (if logging is enabled) with status “warning”.

Thread pool example

A thread pool with a maximum number of threads can be configured for the HTTP server.

In this code example a thread pool scheduler with 20 threads is created and used:

Code example

```
// create the thread pool scheduler
SchedulerOfThreadPool := TIdSchedulerOfThreadPool.Create(Connector.HTTPServer);
```

```
SchedulerOfThreadPool.PoolSize := 20;

// assign the thread pool scheduler to the internal Indy HTTP server
Connector.HTTPServer.Scheduler := SchedulerOfThreadPool;
```

Interceptor example

The unit tests include an example which shows how this property can be used to add an interceptor to the server.

After running this test, a file (httpIntercept.log) in the test source folder contains the log interceptor output.

Code example

```
procedure TAPIConfigTests.TestAddConnector;
var
  Server: TdjServer;
  Context: TdjWebAppContext;
  Connector: TdjHTTPConnector;
  Intercept: TIdServerInterceptLogFile;
begin
  Intercept := TIdServerInterceptLogFile.Create(nil);
  try
    Server := TdjServer.Create;
    try
      // add a configured connector
      Connector := TdjHTTPConnector.Create(Server.Handler);
      Connector.Host := '127.0.0.1';
      Connector.Port := 8080;

      // new property "HTTPServer"
      // here used to set a file based logger for the HTTP server
      Connector.HTTPServer.Intercept := Intercept;
      Intercept.Filename := 'httpIntercept.log';

      Server.AddConnector(Connector);

      Context := TdjWebAppContext.Create('get');
      Context.Add(TNoOpComponent, '/hello');
      Server.Add(Context);

      Server.Start;

      CheckEquals('', Get('/get/hello'));

    finally
      Server.Free;
    end;
  finally
    Intercept.Free
  end;
end;
```

Log output

```
127.0.0.1:49327 Stat Connected.
127.0.0.1:49327 Recv 26.06.2012 09:32:09: GET /get/hello HTTP/1.1<EOL>Host:
127.0.0.1<EOL>Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8<EOL>Accept-Encoding:
identity<EOL>User-Agent: Mozilla/3.0 (compatible; Indy Library)<EOL><EOL>
127.0.0.1:49327 Sent 26.06.2012 09:32:09: HTTP/1.1 200 OK<EOL>Connection: close<EOL>Content-
Length: 0<EOL>Server: Internet Direct (Indy) 10.5.8.0<EOL><EOL>
127.0.0.1:49327 Stat Disconnected.
0.0.0.0:0 Stat Disconnected.
```

SLF4P support

The framework supports the **Simple Logging Facade for Pascal** (SLF4P) which is available from GitHub at https://github.com/michaelJustin/slf4p.

Starting with version 3.1.0 of the framework, the minimum supported version of slf4p is 1.0.8.

IDE configuration

In order to compile with logging support, add the conditional symbol `DARAJA_LOGGING` to the project options:

- in Delphi, choose Project | Options... | Delphi Compiler > Conditional defines and add `DARAJA_LOGGING`
- in Lazarus, choose Project | Project Options … | Compiler Options > Other and add `-dDARAJA_LOGGING` in the Custom options field

Unit Tests

Expected Exceptions

During the unit tests, some expected exceptions will be thrown. In the Delphi IDE, these exceptions should not cause the debugger to interrupt program execution.

To fix this, either add the exceptions in the IDE options dialog ("Exception types to ignore") or check the option "Ignore this exception type" in the dialog:

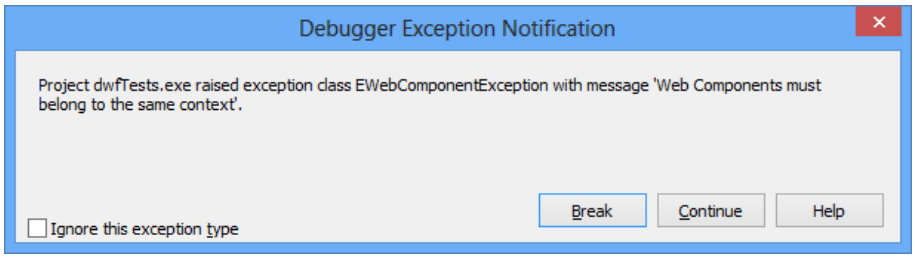

Illustration 6: Debugger Exception Notification

Currently these exceptions are thrown in the tests:

- EidHTTPProtocolException
- EWebComponentException
- EUnitTestException

Example Web Components

TdjDefaultWebComponent

For every context, the framework can provide a folder for static resources (images, scripts, style sheets or static HTML documents).

To add support for static resource folder, register TdjDefaultWebComponent:

Code example

```
Context.Add(TdjDefaultWebComponent, '/');
```

TdjDefaultWebComponent serves resources which do not match the URL mapping of any other web component, and tries to serve them from the static resource folder.

The static resource folder is located under the `webapps` directory of the server and has the same name as the context. Example directory:

```
\MyWebApp
 - MyWebApp.exe
 - \webapps
   - \myapp1 ← folder for static resource files in context “myapp1”>
```

TdjNCSALogFilter

TdjNCSALogFilter is a web filter which provides logging support in the NCSA log format.[2]

Example log output:

```
info | 127.0.0.1 - - [09/Apr/2013:16:00:15:853 +0200] "GET /demo/index.html HTTP/1.1" 200
9050
```

2 https://en.wikipedia.org/wiki/Common_Log_Format

TdjStatisticsFilter

TdjStatisticsFilter collects statistical information about number of requests and more.

Other optional units

ShutdownHelper

This unit is included in the `demo/commons` folder and registers a console control event handler.

It has one method, **SetShutdownHook**. This method takes a single parameter which is a **TdjServer** instance.

If the console window receives a Ctrl+C / break / close / shutdown / logoff event, the `Server.Stop` method will be called.

Code example

```
procedure SetShutdownHook(const Server: TdjServer);
```

Windows platform This unit can only be used on the Windows platform.

Third-Party Product Licenses

Internet Direct (Indy)

Indy will be compiled into applications built with Daraja HTTP Framework, this means that the Indy license applies to your application.

License information for Indy can be found at https://www.indyproject.org/license/

Index

Reference

Table Index

Illustration Index