**TMWEBRock Documentation**

**Introduction**

TMWEBRock is a robust J2EE web services framework designed to simplify web application development through a flexible and intuitive structure. This documentation provides a comprehensive guide on configuring the framework to scan specific packages, handling requests, and leveraging various annotations for enhanced functionality.

**Configuration in web.xml**

To instruct the framework to scan specific packages for services, configure the `StartupServlet` in the `web.xml` file. Define the following parameters within the servlet mapping:

<servlet>

<servlet-name>StartupServlet</servlet-name>

<servlet-class>com.yourpackage.StartupServlet</servlet-class>

<init-param>

<param-name>SERVICE\_PACKAGE\_PREFIX</param-name>

<param-value>com.yourpackage.services</param-value>

</init-param>

</servlet>

Here, `SERVICE\_PACKAGE\_PREFIX` is the parameter name, and `com.yourpackage.services` is the package prefix for scanning services.

**Scanning Process**

The `StartupServlet` scans the specified package for classes annotated with `@Path`. It extracts information about the class and its methods to create `Service` objects. If a method also has a `@Path` annotation, the framework stores the service details in its data structure with the key as the concatenated path of the class and method.

**Request Handling**

When a request is received, the framework checks its data structure. If a matching service is found, it creates an instance of the corresponding class and invokes the method associated with the requested path. The result is then returned as the response. If no matching service is found, a 404 error page is returned.

**Annotations**

**@Get and @Post Annotations**

**@Get Annotation**: Applied to a class and method, it signifies that methods with `@Path` annotations should only be processed for GET requests. Other request types result in a method-not-allowed error page.

If the `@Get` annotation is applied to a class, it indicates that any method with a `@Path` annotation in that class should only be processed for GET requests. Other request types will result in a method-not-allowed error page.

**@Post Annotation**: Applied to a class and methods, it indicates that methods with `@Path` annotations should only be processed for POST requests.

Similarly, if the `@Post` annotation is applied to a class, it means that methods with a `@Path` annotation in that class should only be processed for POST requests.

**Example 1:**

@Get

@Path("/student")

public class abcd

{

}

**Example 1:**

@Post

@Path("/student")

public class abcd

{

}

**@Path on Methods with @Get and @Post**: Specifies that a method should be serviced only for the indicated request type. If a different type is received, a method-not-allowed error page is sent.

If the `@Path` annotation is applied to a method and either `@Get` or `@Post` annotation is also present, it means that the method should be serviced only when a request of the specified type comes in. If a request of a different type is received, a method-not-allowed error page will be sent.

**Example 1:**

@Get

@Path("/student")

public class abcd

{

@Get

@Path("/lmn")

public String lmn()

{

}

}

**Example 2:**

@Get

@Path("/student")

public class abcd

{

@Post

@Path("/lmn")

public String lmn()

{

}

}

**Example 3:**

@Post

@Path("/student")

public class abcd

{

@Get

@Path("/lmn")

public String lmn()

{

}

}

**Example 4:**

@Post

@Path("/student")

public class abcd

{

@Post

@Path("/lmn")

public String lmn()

{

}

}

**@Forward Annotation**: Applied to a method, processes the specified path after method execution. A 500 error page is returned if the path is not found.Internally, the `forwardTo` property of the `Service` class is set to the value specified in the `@Forward` annotation.

@Forward("/student/pqr")

@Post

@Path("/student")

public class abcd

{

@Post

@Path("/lmn")

public String lmn()

{

}

}

**@OnStartup(priority=1)**: Applied to methods for execution after data structure creation during server startup. Arranged based on priority in a map. No further forwarding checks are performed.

The services with this annotation are arranged based on their priority in a map of `Map<Integer, List<Service>>`. After execution, no further checks for forwarding are performed. The function should have a return type of `void` and no parameters.

@Get

@Path("/student")

public class abcd

{

@OnStartup(priority=1)

@Path("/mksl")

@Post

public void mksl()

{

System.out.println("I am inside method mksl of class abcd");

}

}

**@InjectApplicationDirectory, @InjectApplicationScope, @InjectSessionScope, @InjectRequestScope**:

These annotations are applied only at the class level. If any of these annotations are present, the corresponding scope is injected into the class properties before any method is processed. This demonstrates the Inverse of Control (IoC) principle, allowing the framework to handle the setup of required dependencies.

@InjectSessionScope

@InjectApplicationScope

@InjectApplicationDirectory

@InjectRequestScope

@Get

@Path("/student")

public class abcd

{

private ApplicationDirectory directory;

private SessionScope sess;

private RequestScope requestScope;

private ApplicationScope applicationScope;

public void setSess(SessionScope session)

{

this.sess=session;

System.out.println("SetSessionScope of abcd get called "+sess);

}

public void setRequestScope(RequestScope request)

{

this.requestScope=request;

System.out.println("SetRequestScope of abcd get called "+requestScope);

}

public void setDirectory(ApplicationDirectory directory)

{

this.directory=directory;

System.out.println("SetApplicationDirectory of abcd get called "+directory);

}

public void setApplicationScope(ApplicationScope application)

{

this.applicationScope=application;

System.out.println("SetApplicationScope of abcd get called "+applicationScope);

}

@Forward("/abcddd/so")

@Path("/pqr")

public void pqr()

{

System.out.println("I am inside method pqr of class abcd");

sess.setAttribute("Hello","Great");

requestScope.setAttribute("Ujjain","Is the city of gods");

}

}

**@Autowired(name="")**:

The `@Autowired` annotation is applied to class properties. It enables the injection of dependencies based on the specified name. If a matching dependency is found in the request scope, session scope, or servlet context, it is injected. The priority is given to the request scope, followed by session scope, servlet context, and finally, `null` if not found.

Point to be noted: Material received based on the name must match the type of the property.

Detailed Autowiring Information: To store autowiring details, TMWEBRock uses an ArrayList of FieldValuePair. Example: List<FieldValuePair> details = service.getArrayList();

**FieldValuePair contains three properties:**

1. field.getType() (field type)
2. annotationss.name() (annotation name value)
3. field.getName() (field name).

@Path("/abcddd")

public class tom

{

@Autowired(name="Hello")

private String hello;

@Autowired(name="Ujjain")

private String ujjain;

public void setHello(String name)

{

this.hello=name;

}

public void setUjjain(String name)

{

this.ujjain=name;

}

@Forward("/student/jkmk")

@Post

@Path("/so")

public void so()

{

System.out.println("I am inside so of abcd and i fetch : "+hello+"from sessionScope");

System.out.println("I am inside so of abcd and i fetch : "+i+"from applicationScope");

System.out.println("I am inside so of abcd and i fetch : "+ujjain+"from requestScope");

}

}

**@RequestParameter("") and @InjectRequestParameter("")**: Used for mapping method parameters to incoming request parameters and injecting request parameters into class properties before method processing, respectively.

These annotations are used to handle request parameters. For methods, `@RequestParameter` is applied to method parameters to map them directly to incoming request parameters. For class properties, `@InjectRequestParameter` is applied to inject request parameters into the class before any method is processed.

@Get

@Path("/student")

public class abcd

{

@InjectRequestParameter("name")

private String great;

@InjectRequestParameter("id")

private int rollNumber;

public void setGreat(String great)

{

this.great=great;

}

public void setRollNumber(int rollNumber)

{

this.rollNumber=rollNumber;

}

@Forward("/student/pqr")

@Post

@Path("/lmn")

public String lmn(@RequestParameter("name")String name,@RequestParameter("id")int rollNumber)

{

System.out.println("Value of name is : " +name);

System.out.println("Value of id is : " +rollNumber);

return "Hello happy to see you here";

}

@Forward("/aaa")

@Path("/jkmk")

public void abcddm()

{

System.out.println("Value of name is : "+great);

System.out.println("Value of id is : "+rollNumber);

System.out.println("I am inside method jkmk of class abcd");

}

}

**Security with @SecuredAccess**: Applied at the class or method level, checks access conditions before processing the service.

If the `checkPost` class and `guard` method are defined, they are executed before the actual service. If an exception is generated during these checks, a 404 error page is sent.

@Get

@Path("/student")

public class abcd

{

@SecuredAccess(checkPost="bobby.secure.Login",guard="efgh")

@Forward("/abcddd/so")

@Path("/pqr")

public void pqr(ApplicationScope applicationScope)

{

}

}

**Handling JSON Requests in TMWEBRock**

TMWEBRock supports JSON requests for methods in your web application. This section outlines the considerations and guidelines for handling JSON requests effectively.

**JSON Request Parameters**

When a method is designed to handle JSON requests, consider the following points:

1. **No Parameters Case**: If a method has no parameters, TMWEBRock will still handle JSON requests seamlessly. The method will execute without expecting any specific input parameters.

2. **Single Parameter for JSON Data**: If the method expects JSON data, there's no need for the `@RequestParameter` annotation. The method parameter itself can directly be of the type that corresponds to the structure of the incoming JSON data.

3. **Mixed Parameters**: If the method has other parameters in addition to JSON data, ensure that the first parameter corresponds to the structure of the incoming JSON data. The subsequent parameters (n-1) can only be of types such as `SessionScope`, `RequestScope`, or `ApplicationDirectory`.

**Example**

Consider the following methods in your web service:

1. **No Parameters Case**

@Path("/api/noParameter")

public void processNoParameter()

{

// Method logic without any parameters

}

2. **Single Parameter for JSON Data**:

@Path("/api/singleParameter")

public void processJsonData(MyJsonDataType jsonData)

{

// Method logic handling the JSON data directly

}

3. **Mixed Parameters**:

@Path("/api/mixedParameters")

public void processDataAndScope(MyJsonDataType jsonData, @InjectApplicationScope ApplicationScope appScope)

{

// Method logic handling the JSON data and additional parameters

}

**Handling JSON Requests in TMWEBRock**

1. **Request Mapping**: Ensure that your request is mapped to the correct endpoint that corresponds to the `@Path` annotation of the method.

2. **JSON Data**: When making a request, include the JSON data in the request body. For methods expecting JSON data, the parameter in the method signature directly handles it.

3. **Mixed Parameters**: If additional parameters are present in the method, ensure that they are of the correct types and scopes as needed. Currently, only `SessionScope`, `RequestScope`, or `ApplicationDirectory` types are supported.

By adhering to these guidelines, your TMWEBRock-powered web application can seamlessly handle and process JSON requests in various scenarios.

**Automatic Injection of Scopes and Directory**

When a method's parameter is of type SessionScope, RequestScope, ApplicationScope, or ApplicationDirectory, TMWEBRock automatically injects the corresponding instances into the method parameters.

**Module-specific JS Files**:

To generate module-specific JS files automatically, the user needs to specify the `jsFile` parameter in the `StartupServlet` configuration. The generated JS file will be placed in the `WEB-INF/js` folder. Users can include the JS file in their HTML pages using `<script src='/TMWEBRock/jsFile?name=abcd'></script>`.

**Form Element Naming and Automatic Data Binding in TMWEBRock**

TMWEBRock simplifies the handling of form data by providing automatic data binding to method parameters. This section outlines the conventions for form element naming and demonstrates how to enable automatic data binding in your web application.

**Form Element Naming**

1. **Consistent Id and Name**: The `id` and `name` attributes of form elements should be consistent and identical. This consistency is crucial for TMWEBRock to correctly map form data to corresponding POJO properties.

<!-- Example of consistent id and name for a form input -->

<input type="text" id="username" name="username" />

**Automatic Data Binding**

To enable automatic data binding for your form, follow these steps:

1. **Form Submission with Reference**:

<!-- Example of form with a unique id -->

<form id="form1" onsubmit="return submitForm('form1')">

<!-- Form elements with consistent id and name attributes -->

<input type="number" id="id" name="id" />

<input type="text" id="name" name="name" />

<input type="text" id="gender" name="gender" />

<button type="submit">Submit</button>

</form>

In the form tag, the `onsubmit` attribute calls a JavaScript function (`submitForm`) with the form's unique id.

2. **JavaScript Function to Submit Form**:

window.addStudent = function(formID) {

$('#whatever').text("");

var s = new StudentService();

var a = s.add(formID);

a.then(

function(result)

{

if(result.successful==true)

{

$('#whatever').append(result.result);

}

else

{

$('#whatever').append(result.exception);

}

},

function(error)

{

$('#whatever').append(error.exception);

});

return false;

};

3. **Java Method with Automatic Data Binding**:

@Path("/add")

public void add(Student student)

{

// Method logic handling form data with automatic data binding

// the name and id of form field (`id` , `name` , `gender` in this example) matches Student class properties .

}

Ensure that the name and id of form field (`id` , `name` , `gender` in this example) matches Student class properties .

By following these conventions and enabling automatic data binding, TMWEBRock streamlines the handling of form data in your web application. This approach minimizes manual parameter handling, making your code more concise and readable.

**Configuration in web.xml for Module Mapping**

To map the framework to specific modules, define servlet mappings in the `web.xml` file:

For example:

<servlet>

<servlet-name>ServletOne</servlet-name>

<servlet-class>com.thinking.machines.webrock.TMWebRock</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ServletOne</servlet-name>

<url-pattern>/schoolService/\*</url-pattern>

</servlet-mapping>

Here, `/schoolService/` is the module name, and requests with this prefix will be processed by the TMWEBRock framework.

**Conclusion**

TMWEBRock streamlines the development of J2EE web services by providing a simple configuration mechanism and a powerful scanning process for handling requests. By following the outlined conventions and annotations, developers can efficiently build robust web applications with this framework.

**ServicesDoc**

**Introduction**

"ServicesDoc" is a tool designed to generate pdf containing details for various modules and services in a web application.

**Getting Started**

To use the tool, run the following command in the command line:

java -classpath path\_to\_jar\_file\_of\_ServicesDoc com.thinking.machines path\_to\_classes\_or\_jar\_file output\_pdf\_file\_path

**Command Line Arguments**

**Argument 1**: `path\_to\_classes\_or\_jar\_file`

**Description**: Specify the path to the classes folder or the path to the JAR file containing the web application modules and services.

**Argument 2**: `output\_pdf\_file\_path`

**Description**: Specify the path where you want to create the PDF file. The file will be named `output.pdf`.

**Output**

The generated PDF file will contain detailed documentation for each class, organized in a neat and readable format. The documentation will include information about the modules, services, endpoints, parameters, and other relevant details.

**Example**

java -classpath ServicesDoc.jar com.thinking.machines /path/to/webapp/classes /output/path/documentation.pdf

**Notes**

- Ensure that the specified paths are accurate.

- Make sure the classes or JAR file contains the necessary information for documentation.

- The tool may have additional dependencies that need to be satisfied.

**Conclusion**

Congratulations! You have successfully generated documentation for your web application modules and services using ServicesDoc. Explore the generated documentation to understand and utilize the functionalities provided by your web application.