NJWebRock Documentation

1. Prerequisites and Setup

Before using NJWebRock, ensure the following:

1.1 Required JARs

Place these JAR files in your project's WEB-INF/lib:

a)njwebrock.jar — The core NJWebRock framework
b)gson.jar — Handles JSON serialization/deserialization
c)itextpdf.jar — Generates PDF documentation of services
Note: All classes in these JARs should be available in the classpath

1.2 web.xml Entry for NJWebRock

1.2.1

```
<context-param>
   <param-name>SERVICE_PACKAGE_PREFIX</param-name>
   <param-value>bobby</param-value>
</context-param>
```

Explanation:
a)param-name: Must always be SERVICE_PACKAGE_PREFIX.
b)param-value: Name of the folder inside the classes folder (under WEB-INF/classes) that will be scanned for user-defined service classes.
Example: If your services are in WEB-INF/classes/bobby/..., then param-value = bobby.
This is required for NJWebRock to locate the user service classes.

1.2.2

```
<servlet>
   <servlet-name>NJWebRock</servlet-name>
   <servlet-class>com.thinking.machines.webrock.NJWebRock</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>NJWebRock</servlet-name>
```

```
    <url-pattern>/schoolService/*</url-pattern>
</servlet-mapping
```

Explanation:

a)servlet-name: Always NJWebRock.
b)servlet-class: Always com.thinking.machines.webrock.NJWebRock
c)url-pattern: The user can decide this.
d)Example: /schoolService/*
NOTE : NJWebRock will handle all requests and responses matching this pattern.
This allows the framework to intercept and process all HTTP requests for the given URL pattern.

1.2.3

```
<servlet>
    <servlet-name>NJWebRockStarter</servlet-name>
    <servlet-class>com.thinking.machines.webrock.NJWebRockStarter</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>NJWebRockStarter</servlet-name>
    <url-pattern>/njWebRockStarter</url-pattern>
</servlet-mapping>
```

Explanation:

a)Must remain exactly word-to-word.
b)This servlet runs on startup (<load-on-startup>1</load-on-startup>).
c)Scans all folders inside the SERVICE_PACKAGE_PREFIX folder (e.g., bobby) and creates the framework's internal data structures for service handling.
d)Developers do not change the class name, servlet name, or URL pattern — this is essential for framework initialization.

1.2.4

web.xml Entry for NJWebRockJSFileGeneratorOnStartup

```
<servlet>
    <servlet-name>NJWebRockJSFileGeneratorOnStartup</servlet-name>
    <servlet-class>com.thinking.machines.webrock.NJWebRockJSFileGeneratorOnStartup</servlet-class>
    <load-on-startup>2</load-on-startup>
    <init-param>
        <param-name>jsonFileName</param-name>
        <param-value>script.json</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>NJWebRockJSFileGeneratorOnStartup</servlet-name>
    <url-pattern>/njWebRockJSFileGeneratorOnStartup</url-pattern>
</servlet-mapping>
```

Explanation:

a)Optional: User adds this servlet only if they want JS files auto-generated.
b)param-name: Always jsonFileName.
c)param-value: Name of the JSON file located in WEB-INF that defines entities and services.
d)Purpose: Reads the JSON file and generates JS files for all entities and services.
Example: Student.js, StudentService.js, etc.

Output Location:

Generated JS files are stored in WEB-INF/js/ (created if it doesn't exist).
Example JSON file (script.json):

```
{
  "entities": [
    {
      "name": "Student",
      "fields": [ "rollNumber", "name", "gender" ]
    }
  ],
  "services": [
    {
      "name": "StudentService",
      "basePath": "/StudentService",
      "webPath": "/schoolService",
      "methods": [
        { "name": "add",    "http": "POST", "path": "/add",   "body": "Student" },
        { "name": "delete", "http": "GET",  "path": "/delete", "query": [ "rollNumber" ] },
        { "name": "getAll", "http": "GET",  "path": "/getAll" },
        { "name": "get",    "http": "GET",  "path": "/getByRollNumber", "query": [ "rollNumber" ] },
        { "name": "update", "http": "POST", "path": "/update", "body": "Student" }
      ]
    }
  ]
}
```

generated js files

Student.js

```
// POJO for Student
class Student {
constructor(rollNumber, name, gender) {
this.rollNumber = rollNumber;
this.name = name;
this.gender = gender;
}
}
```

StudentService.js

```
// Service class for StudentService
class StudentService {
add(student) {
return new Promise(function(resolve, reject) {
$.ajax({
url: "/njwebrock/schoolService/StudentService/add",
method: 'POST',
data: JSON.stringify(student),
contentType: 'application/json',
success: function(r) { resolve(r); },
error: function(e) { reject(e); }
});
});
}

delete(rollNumber) {
return new Promise(function(resolve, reject) {
$.ajax({
url: "/njwebrock/schoolService/StudentService/delete?rollNumber="+rollNumber,
method: 'GET',
success: function(r) { resolve(r); },
error: function(e) { reject(e); }
});
});
}

getAll() {
return new Promise(function(resolve, reject) {
$.ajax({
url: "/njwebrock/schoolService/StudentService/getAll",
method: 'GET',
success: function(r) { resolve(r); },
error: function(e) { reject(e); }
});
});
}

get(rollNumber) {
return new Promise(function(resolve, reject) {
$.ajax({
url: "/njwebrock/schoolService/StudentService/getByRollNumber?rollNumber="+rollNumber,
method: 'GET',
success: function(r) { resolve(r); },
error: function(e) { reject(e); }
});
});
}
```

```
update(student) {
return new Promise(function(resolve, reject) {
$.ajax({
url: "/njwebrock/schoolService/StudentService/update",
method: 'POST',
data: JSON.stringify(student),
contentType: 'application/json',
success: function(r) { resolve(r); },
error: function(e) { reject(e); }
});
});
}

}
```

explanation of script.json how to use

1. JSON File Location

Place the file inside your application at: WEB-INF/script.json

The file name can be customized using the jsonFileName init parameter in
NJWebRockJSFileGeneratorOnStartup servlet.

2. JSON File Structure

The JSON file has two main sections:
2.1 Entities :Defines the data entities for which POJO-like JS classes will be generated.

Structure:

```
"entities": [
   {
      "name": "EntityName",
      "fields": ["field1", "field2", "field3"]
   }
]
```

name → The entity name (used as class name in JS).
fields → List of fields for the entity (will become class properties in JS).

2.2 Services
Defines service endpoints and their associated HTTP methods.

Structure:

```
"services": [
```

```
  {
    "name": "ServiceName",
    "basePath": "/ServiceBasePath",
    "webPath": "/ServletURLPattern",
    "methods": [
      { "name": "methodName", "http": "GET/POST", "path": "/methodPath", "body": "EntityName", "query":
      ["param1","param2"] }
    ]
  }
]
```

name → Name of the service (used in JS service class).
basePath → Path appended after the servlet URL pattern.
webPath → The servlet mapping URL pattern (e.g., /schoolService).
methods → List of service methods:
name → Method name
http → GET or POST
path → Path relative to basePath
body → Entity sent in the request body (for POST)
query → Array of query parameters (for GET requests)

1.2.5

Note : For serving auto-generated JS files (like DTOs and Services), the user must add this mandatory entry in web.xml:

```
<servlet>
  <servlet-name>NJWebRockJSFileServing</servlet-name>
  <servlet-class>com.thinking.machines.webrock.NJWebRockJSFileServing</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>NJWebRockJSFileServing</servlet-name>
  <url-pattern>/iii</url-pattern>
</servlet-mapping>
```

Rules:

a)servlet-name → must always be NJWebRockJSFileServing.
b)servlet-class → must always be com.thinking.machines.webrock.NJWebRockJSFileServing.
c)Only url-pattern can be chosen by the user (in your example /iii).

How to use in HTML (JS inclusion)

.)Once the servlet is mapped, user can load generated JS files like this:

```
<script src="/njwebrock/iii?name=Student.js"></script>
<script src="/njwebrock/iii?name=StudentService.js"></script>
```

Here:

a)/njwebrock → your webapp context path.
b)/iii → servlet mapping decided by user.

name parameter → filename of the generated JavaScript to be served.

2)Annotations in NJWebRock

To use annotations : import com.thinking.machines.webrock.annotations.*;
To use RequestScope,SessionScope or ApplicationScope : import com.thinking.machines.webrock.scope.*;
To use ApplicationDirectory : import com.thinking.machines.webrock.*;
To use ServiceException : import com.thinking.machines.webrock.exceptions.*;

2.1 Annotation: @Path

Purpose: Defines the URL path mapping for classes and methods in NJWebRock.
Makes the framework able to scan and expose your class/method as a service.

Usage in Your Example

```
@Path("/Student")
public class Student
```

@Path("/Student") on a class → makes this class discoverable by NJWebRockStarter.
The URL path for the class will start with /Student.

```
@Path("/getName")
public String getName()
```

@Path("/getName") on a method → exposes this method as a microservice endpoint.

The full URL becomes: /schoolService/Student/getName (assuming /schoolService/* is your servlet mapping).

When a client hits this URL, the framework executes getName() and returns the result.

Key Points from the Example

a)Class-level @Path → Defines the base path for all methods in that class.
b)Method-level @Path → Defines the endpoint for that specific method.
c)Return types → Can be String or objects(automatically serialized if needed) or void

```
import com.thinking.machines.webrock.annotations.*;

@Path("/Student")
public class Student {


@Path("/getName")
```

```
public String getName() {
   return "Nitin Jat";
}

}
```

Explanation:

Method getName() is accessible at:
When a client hits this URL (/schoolService/Student/getName), NJWebRock executes the method and returns the result.

2.2 Annotation: @GET / @POST

Purpose:
Restrict which HTTP method (GET or POST) can be used to access a service method.
Ensures that clients use the correct HTTP verb when calling the microservice.

Usage Example
```
@Path("/Student")
public class Student {

   @GET
   @Path("/getName")
   public String getName() {
      return "Nitin Jat";
   }

}
```

Explanation:
a)@GET → This method can only be invoked using an HTTP GET request.
b)@POST → Similarly, @POST can be used to restrict access to POST requests.
c)The full URL of the service (with servlet mapping /schoolService/*) becomes:
/schoolService/Student/getName

If a client tries to access this URL with a different HTTP method (e.g., POST for a @GET method), NJWebRock will reject the request.

Key Points
a)Use @GET for read-only operations.
b)Use @POST for create/update operations.
c)Can be combined with @Path at method level to define endpoint URL.

d)Helps NJWebRock enforce HTTP method type safety for each service.

2.3 Annotation: @Forward

Purpose: Automatically forwards a request to another endpoint after the annotated method finishes execution.
Can forward to another service class's method or a static resource (like HTML/JSP).

Usage Example

```
@Path("/Student")
public class Student {

   @Forward("/Teacher/putAttendence")
   @GET
   @Path("/computeAndForward")
   public void computeAndForward() {
      // Method logic executed first
      // Then request is automatically forwarded to putAttendence method of Teacher class
   }

}
```

Explanation:

a)computeAndForward() is executed first.
b)After completion, NJWebRock forwards the request to: /schoolService/Teacher/putAttendence
Teacher → the target service class
putAttendence → the target method in Teacher class
c)You can also forward to a static resource like /index.html.

Key Points

a)Can be combined with @GET / @POST and @Path.
b)Useful for request chaining between services or redirecting to static resources.

Forward target can be:
a)Another service class method (/ClassName/methodName)
b)A static HTML/JSP page (/index.html)

```
@Forward("/index.jsp")
@GET
@Path("/redirectToHomeJSP")
public void redirectToHomeJSP() {
   // Executes some logic
   // Then forwards to index.jsp
}
```

2.4 @RequestParameter Annotation

Purpose :

The @RequestParameter annotation is used when you want to bind a request parameter (sent by the client in the query string or form data) to a method argument in your microservice.

Note : You must specify the name of the parameter that will come in the request.

The framework automatically converts the parameter from String (which is how it comes in HTTP requests) to the method argument's Java type (such as int, double, boolean, String, etc.) or it can be custom object.

Example 1: Simple Use Case

```
@Path("/Student")
public class StudentService
{
   @GET
   @Path("/getDetails")
   public String getDetails(@RequestParameter("rollNumber") int rollNumber)
   {
      return "Requested student roll number is: " + rollNumber;
   }
}
```

If the client calls: http://localhost:8080/app/Student/getDetails?rollNumber=101

Then the framework will automatically pick up rollNumber=101 from the request, convert it into an int, and pass it to the method.

The output will be: Requested student roll number is: 101

Example 2: Multiple Parameters

```
@Path("/Student")
public class StudentService
{
   @GET
   @Path("/register")
   public String register(
      @RequestParameter("name") String name,
      @RequestParameter("age") int age)
   {
      return "Student registered: " + name + " (" + age + " years old)";
   }
}
```

Request URL: http://localhost:8080/app/Student/register?name=Nitin&age=20
Output: Student registered: Nitin (20 years old)

2.5 @AutoWired Annotation

Purpose :

The @AutoWired annotation is used in NJWebRock when you want the framework to automatically inject values into your class variables from one of the three scopes: Request Scope ,Session Scope,Application Scope

This avoids the need to manually fetch values from request, session, or application.
You just declare a variable, annotate it with @AutoWired(name="..."), and provide a setter method for it.

Syntax

```
@AutoWired(name="objectName")
private DataType variableName;

public void setVariableName(DataType variableName) {
   this.variableName = variableName;
}
```

Explanation

a)name="objectName" → The name of the object that is present in request/session/application scope.
b)DataType → The type of value you expect (like int, String, UserBean, etc.).

setVariableName(...) → The setter method must follow JavaBeans naming convention:
a)Start with set
b)Followed by camelCase variable name.

Simple Example

Suppose you are storing a discount value in request scope.
Instead of fetching it manually from request.getAttribute("discount"), you can use @AutoWired.

```
@Path("/Shop")
public class ShopService
{
   @AutoWired(name="discount")
   private int discount;

   public void setDiscount(int discount)
   {
      System.out.println("Discount AutoWired: " + discount);
      this.discount = discount;
   }

   @GET
   @Path("/calculate")
```

```
    public String calculateTotal()
    {
        int price = 100;
        int finalPrice = price - discount;
        return "Final price after discount: " + finalPrice;
    }
}
```

How It Works

a)The framework will look into the request/session/application scope for an attribute named "discount".
b)If found, its value will automatically be passed to the setter setDiscount(int discount).
c)The discount field is then populated before the service method (calculateTotal) executes.

Key Points for Users

a)Always provide a setter method matching the variable name.
b)Variable name should follow JavaBeans standard (e.g., discount → setDiscount).
c)The value is auto-injected before method execution, so you can directly use it in your logic.
d)Scope can be request/session/application — whichever has the attribute at runtime.

2.6 @OnStartup Annotation

Purpose :

The @OnStartup annotation is used when you want certain methods of a class to run automatically at server startup.
This is useful for initializing resources, pre-loading configurations, or setting up tasks before the application starts serving requests.

Syntax

```
@OnStartup(priority = <number>)
@Path("/methodName")
public void methodName() {
    // initialization logic
}
```

Important

a)priority → Defines the execution order of methods marked with @OnStartup.
b)Smaller number = Higher priority.
Example: priority=1 runs before priority=2.
c)If multiple methods have the same priority, their execution order is non-deterministic (any order).
d)The method can be annotated with @Path just like normal request handlers, but it will also run automatically on startup.

Simple Example

```
public class StartupService
{
    @OnStartup(priority=1)
```

```
    @Path("/initDatabase")
    public void initDatabase()
    {
        System.out.println("Database initialized on server startup");
        // Example: Load DB connections or seed data
    }

    @OnStartup(priority=2)
    @Path("/loadCache")
    public void loadCache()
    {
        System.out.println("Cache loaded on server startup");
        // Example: Pre-load application cache
    }
}
```

How It Works

a)When the server starts, the framework scans for all methods annotated with @OnStartup.
b)These methods are executed before handling any incoming requests.
c)Methods run in the order of their priority value (smaller number first).
d)If two or more methods have the same priority, they may run in any order.

Use Cases

a)Initialize database connections
b)Load configuration files
c)Warm up cache or static data
d)Run background setup tasks

Key Points for Users

a)Always use priority values carefully to control the execution order.
b)Keep startup methods fast and lightweight (slow methods may delay server boot time).
c)You can have multiple startup methods across different classes.
d)Startup methods are called once per application start, not per request.

2.7 Annotation: @InjectRequestScope

Purpose : The @InjectRequestScope annotation is used when a developer wants direct access to the RequestScope object inside a service class or method. This allows the service to store and retrieve request-specific attributes for the duration of a single HTTP request.

Important

a)It can be applied at class level or method level:
.)At class level, all setters/fields marked with RequestScope will automatically be injected before any method execution.
.)At method level, only that method will have request scope injected.

How It Works

a)Declare a field of type RequestScope inside your class.
b)Write a setter method for it. Important: the setter must have a parameter of type RequestScope.
c)Use requestScope.setAttribute(key, value) to store values in the request.
d)Use requestScope.getAttribute(key) to fetch values (returns Object type).

Example:

```
import com.thinking.machines.webrock.annotations.*;
import com.thinking.machines.webrock.scope.*;

@Path("/slope")
@InjectRequestScope   // Applied at class level
public class Slope {

   private RequestScope requestScope;

   // Setter required for injection; parameter type MUST be RequestScope
   public void setRequestScope(RequestScope requestScope) {
      this.requestScope = requestScope;
   }

   @GET
   @Path("/save")
   public void saveData() {
      // Storing data in request scope
      requestScope.setAttribute("msg", "Hello from request scope!");
      System.out.println("Data saved in request scope.");
   }

   @GET
   @Path("/fetch")
   public void fetchData() {
      // Retrieving data from request scope
      Object msg = requestScope.getAttribute("msg");
      System.out.println("Fetched from request scope: " + msg);
   }
}
```

Explanation

a)@InjectRequestScope ensures that the framework injects a RequestScope object into the class.
b)The setter setRequestScope(RequestScope requestScope) is mandatory; it allows the framework to pass the request scope object correctly.
c)Using a different parameter type in the setter will break the injection.
d)RequestScope is request-specific, so its data exists only for the lifetime of that HTTP request.

2.8 Annotation: @InjectSessionScope

Purpose : The @InjectSessionScope annotation is used when a developer wants direct access to the SessionScope object inside a service class or method. This allows the service to store and retrieve session-specific attributes that persist across multiple requests from the same client.

Important

a)It can be applied at class level or method level:
.)Class level → All setter fields marked with SessionScope are injected before any method execution.
.)Method level → Only that method receives the session scope injection.

How It Works

a)Declare a field of type SessionScope inside your class.
b)Write a setter method with a parameter of type SessionScope (mandatory).
c)Use sessionScope.setAttribute(key, value) to store session-level data.
d)Use sessionScope.getAttribute(key) to fetch session-level data (returns Object).

Example :

```
import com.thinking.machines.webrock.annotations.*;
import com.thinking.machines.webrock.scope.*;

@Path("/user")
@InjectSessionScope
public class UserService {

   private SessionScope sessionScope;

   // Setter MUST have parameter type SessionScope
   public void setSessionScope(SessionScope sessionScope) {
      this.sessionScope = sessionScope;
   }

   @POST
   @Path("/login")
   public void login(@RequestParameter("username") String username) {
      sessionScope.setAttribute("currentUser", username);
      System.out.println("User logged in: " + username);
   }

   @GET
   @Path("/profile")
   public void getProfile() {
      Object user = sessionScope.getAttribute("currentUser");
      System.out.println("Current session user: " + user);
   }
}
```

Explanation

a)@InjectSessionScope injects a SessionScope object automatically.
b)The setter must have SessionScope type parameter.
c)Data stored in session scope persists across multiple requests from the same client.
d)Useful for storing login status, preferences, or temporary user data.

2.9 Annotation: @InjectApplicationScope

Purpose : The @InjectApplicationScope annotation is used when a developer wants access to the ApplicationScope object, which is shared globally across all sessions and requests.

Explaination

a)It can be applied at class level or method level:
.)Class level → All setter fields marked with ApplicationScope are injected before any method execution.
.)Method level → Only that method receives the application scope injection.

How It Works

a)Declare a field of type ApplicationScope in your class.
b)Write a setter method with ApplicationScope type parameter (mandatory).
c)Use applicationScope.setAttribute(key, value) to store data accessible globally.
d)Use applicationScope.getAttribute(key) to fetch global data (returns Object).

Example

```
import com.thinking.machines.webrock.annotations.*;
import com.thinking.machines.webrock.scope.*;

@Path("/app")
@InjectApplicationScope
public class AppService {

private ApplicationScope applicationScope;
// Setter MUST have parameter type ApplicationScope
   public void setApplicationScope(ApplicationScope applicationScope) {
      this.applicationScope = applicationScope;
   }

   @GET
   @Path("/setConfig")
   public void setConfig() {
      applicationScope.setAttribute("appName", "NJWebRock");
      System.out.println("Application name set in global scope");
   }

   @GET
   @Path("/getConfig")
   public void getConfig() {
```

```
        Object name = applicationScope.getAttribute("appName");
        System.out.println("Application name: " + name);
    }
}
```

Explanation

a)@InjectApplicationScope injects the global ApplicationScope object.
b)The setter must have ApplicationScope type parameter.
c)Data stored in application scope is shared across all sessions and requests.
d)Useful for storing configuration, global counters, or shared resources.

2.10 Annotation: @InjectApplicationDirectory

Purpose : The @InjectApplicationDirectory annotation is used when a developer wants direct access to the application's directory (the base folder where the application is deployed) through the ApplicationDirectory object.

Important

a)It can be applied at class level or method level:
.)Class level → All fields of type ApplicationDirectory with proper setter will be injected automatically.
.)Method level → Only that method receives the injection.

How It Works

a)Declare a field of type ApplicationDirectory in your class.
b)Write a setter method with a parameter of type ApplicationDirectory (mandatory).
c)The framework injects the directory object, giving access to the underlying File object representing the application's root directory.
d)You can use getDirectory() and setDirectory(File directory) methods of ApplicationDirectory to interact with the file system.

Example

```
import com.thinking.machines.webrock.annotations.*;
import com.thinking.machines.webrock.*;
import java.io.File;

@Path("/appFiles")
@InjectApplicationDirectory
public class FileService {

    private ApplicationDirectory applicationDirectory;

    // Setter MUST have parameter type ApplicationDirectory
    public void setApplicationDirectory(ApplicationDirectory applicationDirectory) {
        this.applicationDirectory = applicationDirectory;
    }
```

```
    @GET
    @Path("/printPath")
    public void printAppPath() {
        File dir = applicationDirectory.getDirectory();
        System.out.println("Application directory path: " + dir.getAbsolutePath());
    }

    @POST
    @Path("/setPath")
    public void setAppPath(File newDir) {
        applicationDirectory.setDirectory(newDir);
        System.out.println("Application directory updated: " + newDir.getAbsolutePath());
    }
}
```

Explanation

a)The framework injects an ApplicationDirectory object into the service class before executing any methods.
b)The setter must have ApplicationDirectory as the parameter type for injection to work correctly.
c)Using the ApplicationDirectory object, you can safely access or update the application's root directory programmatically.

Use Cases

a)Read/write files inside the application folder.
b)Access configuration files or subdirectories.
c)Dynamically update resources within the deployment directory.

Key Points

a)Always declare a field of type ApplicationDirectory.
b)Always provide a setter with parameter type ApplicationDirectory.
c)Can be used at class level or method level depending on your requirements.

2.11 Annotation: @InjectRequestParameter

Purpose : The @InjectRequestParameter annotation is used to extract request parameters automatically from HTTP requests and inject them into service class fields.

Important

a)The framework will convert the parameter to the field’s type (e.g., int, String) and call the setter automatically.
b)The field must have a corresponding setter method, following the naming convention: set + CamelCase of field name.

How It Works

a)Declare a field in your service class.
b)Annotate the field with @InjectRequestParameter("paramName"), where "paramName" is the name of the parameter sent in the request.

c)Write a setter method for the field; the parameter type of the setter must match the field type.
d)During request processing, the framework will:
e)Check if the request contains a parameter with the given name.
f)Convert the value to the correct type.
g)Call the setter method automatically to set the field.

Example

```
import com.thinking.machines.webrock.annotations.*;

@Path("/products")
public class ProductService {

    @InjectRequestParameter("price")
    private int price;

    // Setter method for injection (must match field type)
    public void setPrice(int price) {
        this.price = price;
        System.out.println("Price set via request parameter: " + price);
    }

    @GET
    @Path("/showPrice")
    public void showPrice() {
        System.out.println("Current product price: " + price);
    }
}
```

Request Example: GET /products/showPrice?price=500

a)The framework extracts the price parameter (500) from the request.
b)Converts it to int.
c)Calls setPrice(500).

Key Points

a)Field type and setter parameter type must match.
b)Setter method name: set + CamelCase of the variable name.
c)Parameter in the request is case-sensitive.

2.12 @SecuredAccess Annotation

Purpose :

The @SecuredAccess annotation is used to protect methods or entire classes by adding a security check (guard method) before the actual service method executes.

Important :

a)If the guard check passes, the requested method is executed.
b)If the guard check fails (i.e., guard method throws a ServiceException), then the framework denies access and does not run the requested method.
c)This ensures that only authorized users or valid requests can access sensitive parts of your application.

Usage

a)At Class Level → All methods of that class will be secured.
b)At Method Level → Only that method will be secured.

Ex.

```
@SecuredAccess(checkPost="com.myapp.security.LoginGuard", guard="verify")
@Path("/employee")
public class EmployeeService {

   @GET
   @Path("/getAll")
   public List<Employee> getAllEmployees() {
      // This method will only run if guard check passes
   }
}
```

Parameters
a)checkPost → Fully qualified class name that contains the guard method.
b)guard → The method name inside the checkPost class that will be executed before the actual method.

Example: Method Level Security

```
@Path("/orderService")
public class OrderService {

   @GET
   @Path("/viewOrders")
   @SecuredAccess(checkPost="com.myapp.security.AuthGuard", guard="checkUser")
   public void viewOrders() {
      System.out.println("Orders displayed!");
   }
}
```

Here:

a)Before viewOrders() runs, AuthGuard.checkUser() will execute.
b)If checkUser() throws ServiceException, access is denied.

Example: Class Level Security

```
@SecuredAccess(checkPost="com.myapp.security.AuthGuard", guard="checkUser")
@Path("/productService")
public class ProductService {

    @GET
    @Path("/listProducts")
    public void listProducts() {
        System.out.println("Products listed!");
    }

    @POST
    @Path("/addProduct")
    public void addProduct(@RequestParameter("name") String name) {
        System.out.println("Product added: " + name);
    }
}
```

Here: Both listProducts() and addProduct() are protected by AuthGuard.checkUser().

Writing a Guard Class

a)A guard class is a simple Java class that contains the logic to allow or deny access.
b)If guard method completes successfully → access is granted.
c)If guard method throws ServiceException → access is denied.

```
import com.thinking.machines.webrock.scope.*;
import com.thinking.machines.webrock.exceptions.ServiceException;
import com.thinking.machines.webrock.annotations.*;

@InjectApplicationScope
public class AuthGuard {
    private ApplicationScope applicationScope;

    public void setApplicationScope(ApplicationScope applicationScope) {
        this.applicationScope = applicationScope;
    }

    public void checkUser() throws ServiceException {
        Object status = applicationScope.getAttribute("loginStatus");
        if(status == null || !"success".equals(status.toString())) {
            throw new ServiceException("Access Denied: Please login first.");
        }
    }
}
```

How it Works (Flow)

a)Client sends request → e.g., /orders/view.
b)Framework checks if @SecuredAccess is present.

c)If yes, framework calls the guard method.
d)If guard method completes → requested method executes.
e)If guard method throws ServiceException → request is blocked with "Access Denied".

Best Practices

a)Always use meaningful guard method names (checkLogin, verifyAdmin, validateRole, etc.).
b)Place guard logic in a separate dedicated class (security should be centralized).
c)Use ApplicationScope/SessionScope to store login/session info for validation.
d)Keep guard methods lightweight (avoid DB heavy operations for each request).

In short: @SecuredAccess allows you to secure APIs with guard methods. It is like a security checkpoint – only if the guard approves, your method runs.

3)Features Of NJWebRock

3.1 Automatic JSON to Java Object Mapping

One of the key features of the NJWebRock framework is its ability to automatically convert incoming JSON request bodies into Java objects.
This eliminates the need for developers to manually parse JSON using libraries like Gson or Jackson.

When a request is sent with JSON data, the framework:

a)Reads the request body.
b)Parses the JSON object.
c)Maps it to the corresponding Java class (POJO / Bean) based on the method parameter type.
d)Automatically passes the populated object to the service method.

This makes handling structured data very convenient.

Example

Student Class(Student.java)

```
package bobby.test;
public class Student
{
private int rollNumber;
private String name;
private char gender;

// Default constructor
public Student()
{
```

```
this.rollNumber=0;
this.name="";
this.gender='o';
}

// Getters and setters
public void setRollNumber(int rollNumber) { this.rollNumber=rollNumber; }
public int getRollNumber() { return this.rollNumber; }

public void setName(String name) { this.name=name; }
public String getName() { return this.name; }

public void setGender(char gender) { this.gender=gender; }
public char getGender() { return this.gender; }
}
```

Service Method

```
@Path("/StudentService")
public class StudentService
{
@POST
@Path("/add")
public void add(Student student)
{
  // Framework automatically injects JSON request as Student object
  int rollNumber=student.getRollNumber();
  String name=student.getName();
  char gender=student.getGender();
//do whatever operation needs to be done
  }
}
```

Client-Side Request (AJAX Example)

```
// Creating a Student JavaScript object
let student = {
  rollNumber: 101,
  name: "Amit",
  gender: "M"
};
```

// Sending request

```
function add(student) {
  return new Promise(function(resolve, reject) {
    $.ajax({
      url: "/njwebrock/schoolService/StudentService/add",  // Endpoint URL
      method: 'POST',                                  // Only POST request allowed
```

```
    data: JSON.stringify(student),                  // JSON payload
    contentType: 'application/json',                // Sending JSON format
    success: function(r) { resolve(r); },
    error: function(e) { reject(e); }
   });
  });
}
```

Explanation

The client sends a JSON object:

```
{
  "rollNumber": 101,
  "name": "Amit",
  "gender": "M"
}
```

The framework automatically:
a)Parses this JSON.
b)Creates an instance of Student.
c)Calls the add(Student student) method with this populated object.

So the developer does not need to write any manual JSON parsing code.

3.2 Return Value Forwarding

The NJWebRock framework supports forwarding execution from one service method to another using the @Forward annotation.

In addition to simple forwarding, the framework also provides an advanced feature:

If a method returns a value, and the method to which the request is forwarded accepts a parameter of the same type, then the returned value is automatically passed as an argument to the forwarded method.

This makes it possible to create multi-step processing flows without writing any glue code.

Example

Step 1 – Service with Return + Forward

```
@Path("/StudentService")
public class StudentService
{
  @POST
  @Path("/create")
  @Forward("/Operation/process")
```

```
  public Student create(Student student)
  {
    // Imagine inserting into DB here
    System.out.println("Student created: " + student.getName());

    // Return the student object
    return student;
  }
```

Step 2 – Forwarded Service

```
@Path("/Operation")
class Operation
{
  @POST
  @Path("/process")
  public void process(Student student)
  {
    // The returned student object from create() will come here automatically
    System.out.println("Processing student: " + student.getRollNumber());
  }
}
```

Flow Explanation

a)Client sends a POST request with JSON data to /StudentService/create.
b)Framework converts JSON → Student object → calls create(Student student).
c)The create() method returns a Student object.
d)Because @Forward("/Operation/process") is present and the process() method accepts a Student, the returned object is injected as the argument into the process() method.

The client only calls ("/create"), but internally, both create and process methods are executed in sequence.

Client-Side Request (AJAX)

```
let student = {
  rollNumber: 201,
  name: "Ravi",
  gender: "M"
};

$.ajax({
  url: "/tmwebrock/schoolService/StudentService/create",
  method: 'POST',
  data: JSON.stringify(student),
  contentType: 'application/json',
  success: function(r) { console.log("Request completed"); },
  error: function(e) { console.log("Error: ", e); }
});
```

Output
Student created: Ravi
Processing student: 201

3.3 ServiceDoc Tool Documentation

The ServiceDoc tool is a command-line utility provided with the NJWebRock framework.
Its purpose is to automatically generate a structured PDF document containing metadata of all the services exposed in your project.

Instead of manually maintaining service documentation, developers can run this tool after building their project, and it will scan the compiled classes, extract annotations (@Path, @GET, @POST, @Forward, etc.), and create a clean, well-formatted PDF file with details of every service.

Why ServiceDoc?
a)Eliminates manual effort of writing documentation.
b)Ensures documentation is always in sync with the code.
c)Provides a professional PDF suitable for sharing with teams, clients, or as API reference.
d)Captures details like service class name, package name, method parameters, return types, forward paths, injectables, etc.

Usage
Usage: java -classpath <classpath-to-jar-and-classes> com.thinking.machines.webrock.ServiceDoc <classes-folder> <output-pdf-file>

Where
a)<classpath-to-jar-and-classes> → Path where all required jars (including njwebrock.jar, gson.jar, itextpdf.jar) and compiled classes are available.
b)<classes-folder> → Path to your project's WEB-INF/classes folder where compiled .class files are located.
c)<output-pdf-file> → Destination path where the generated PDF should be stored.

Example Command
> java -classpath c:\tomcat9\webapps\testframework\WEB-INF\lib*;c:\tomcat9\webapps\testframework\WEB-INF\classes;. com.thinking.machines.webrock.ServiceDoc c:\tomcat9\webapps\testframework\WEB-INF\classes c:\pdf\services.pdf

Output : Service documentation generated at: c:\pdf\services.pdf

It generates a table-based PDF with all this information neatly formatted.

Best Practices
a)Always run ServiceDoc after compiling your project (mvn clean install or javac) so it picks up the latest changes.

b)Store generated PDFs in a /docs folder inside your project for easy sharing.
c)Regenerate documentation whenever a new service or annotation is added.
d)Use this PDF as part of API contracts with frontend or client teams.

Error Handling

a)If the <classes folder> path is wrong → tool prints Invalid classes folder.
b)If the PDF file path is invalid → tool prints Unable to create PDF.
c)If required jars (itextpdf.jar, njwebrock.jar, etc.) are missing → tool throws ClassNotFoundException.