

Geomajas dojo face

Geomajas Developers and Geosparc

Geomajas dojo face

by Geomajas Developers and Geosparc

1.5.8-SNAPSHOT

Copyright © 2010-2011 Geosparc nv

Table of Contents

1. Introduction	1
2. dojo Configuration	2
1. Using Google maps raster layer	4
3. dojo Widgets	6
4. dojo example application	7
1. Project layout	7
2. Maven dojo plug-in	8

List of Tables

1.1. List of dojo face modules	1
4.1. List of dojo-example projects	7
4.2. Dojo plug-in configuration parameters	9

List of Examples

2.1. Context listener and configuration locations	2
2.2. Servlet definitions	3
2.3. Servlet mappings	3
2.4. Define constants in external file	4
2.5. Include configuration in HTML	4
2.6. Google API inclusion	4
2.7. Google API inclusion using Google loader	4
4.1. Dispatcher servlet definition and mapping	7
4.2. Custom build script inclusion	7
4.3. Maven Dojo plugin configuration	8

Chapter 1. Introduction

Table 1.1. List of dojo face modules

Name	Purpose
geomajas-dojo-client	Client side module for the dojo AJAX interface.
geomajas-dojo-client-shrinksafe	Server side module for the dojo AJAX interface, after shrinking to combine everything in one JavaScript file.
geomajas-dojo-server	Server side module for the dojo AJAX interface.
documentation	Module containing this documentation.

Chapter 2. dojo Configuration

For the dojo face to function properly, it is important that your web.xml contains the references to the necessary information and servlets. Let's dissect an example web.xml file.

The file starts with the reference and configuration of the Geomajas context listener. This assures the application configuration is available for Geomajas and indicates the location of the additional configuration files. The configuration files are always searched on the classpath. You may include the "classpath:" or "classpath*:" prefix if you want (recommended to allow your IDE to check existence of the resources).

Example 2.1. Context listener and configuration locations

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
  <display-name>Geomajas dojo face example application</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:clientcfg/*.xml
      classpath:servercfg/*.xml
      classpath:applicationContext.xml
    </param-value>
  </context-param>

  <listener>
    <listener-class>org.geomajas.servlet.GeomajasContextListener</listener-class>
  </listener>
```

Next up is the definition of the actual servlets which are needed. The order in which they need to be initialised is also specified.

- `JsonServlet`: the servlet which allows you to execute commands by sending the request as JSON and which returns the result as JSON as well.
- `DispatcherServlet`: this is primarily used to deliver the Geomajas and dojo script files and resources, assuring that they are properly cached and gzip compressed. It can also be used by plug-ins to additional services, for example for delivering PDF files by a printing plug-in.

Example 2.2. Servlet definitions

```

<servlet>
  <servlet-name>JsonServlet</servlet-name>
  <servlet-class>org.geomajas.dojo.server.servlet.JsonServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:META-INF/geomajasWebContext.xml</param-value>
    <description>Spring Web-MVC specific (additional) context files.</description>
  </init-param>
  <!--
  <init-param>
    <param-name>files-location</param-name>
    <param-value>/home/joachim/apps/java/geomajas/geomajas/geomajas-dojo-cl
    <description>
      When this is specified, files are searched here first.
      Files which are found at this locations are not cached.
    </description>
  </init-param>
  -->
  <load-on-startup>3</load-on-startup>
</servlet>

```

To finish, the servlet mappings need to be defined. You don't really have a lot of choice here, as the client side JavaScript expects these values.

Example 2.3. Servlet mappings

```

<servlet-mapping>
  <servlet-name>JsonServlet</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/d/*</url-pattern>
</servlet-mapping>
</web-app>

```

To finish, the servlet mappings need to be defined. You don't really have a lot of choice here, as the client side JavaScript expects these values.

In many cases you also want to do some client side configurations. This can be done in one of two ways.

You can include a "geomajas-constants.js" file in your code (the name of the file can differ).

Example 2.4. Define constants in external file

```

var djConfig={
  isDebug: false,
  parseOnLoad: true,
  usePlainJson: true,
  locale: "en"
};

var geomajasConfig={
  dijitTheme: "soria",
  showLog: true,
  useLazyLoading: true, // use lazy loading
  lazyFeatureIncludesDefault: 12, // by default, only include style + label
  lazyFeatureIncludesSelect: 15, // attributes + geometry + style + label (see Ge
  lazyFeatureIncludesAll: 15 // attributes + geometry + style + label (see Ge
};

```

You can override some configurations in your html file, on the script tags in head.

Example 2.5. Include configuration in HTML

```

<script type="text/javascript" src="d/resource/dojo/dojo.js" djConfig="
  isDebug: false,
  parseOnLoad: true,
  usePlainJson: true,
  locale: 'en'">
</script>
<script type="text/javascript" src="d/resource/example/example.js" geomajasConf
  dijitTheme: 'soria',
  showLog: false,
  useLazyLoading: true,
  lazyFeatureIncludesDefault: 12,
  lazyFeatureIncludesSelect: 15,
  lazyFeatureIncludesAll: 15 "></script>

```

1. Using Google maps raster layer

To use the Google maps raster layer in the dojo face, you have to include the Google maps API files in your application to allow Geomajas to use those. This can be done using code like the following

Example 2.6. Google API inclusion

```

<script type="text/javascript"
  src="http://maps.google.com/maps?file=api&v=2&sensor=false&key="
</script>

```

or

Example 2.7. Google API inclusion using Google loader

```

<script type="text/javascript" src="http://www.google.com/jsapi?key=ABCDEFGH">
</script>
<script type="text/javascript">
  google.load("maps", "2.95");
</script>

```

Note that you do need to have a Google maps key ("ABCDEFGH" is used in the example above) if you want to deploy your site (it should work without when running your tests on "localhost").

In your layer, you can determine the kind of layer by setting the "satellite" property. When this is true, satellite images will be displayed. When this is false, the normal Google maps images are used (street map).

Chapter 3. dojo Widgets

Each widget needs a list of parameters, use case, information about the commands it uses, how to customize, list of topics it consumes and provides.

TODO.....

Chapter 4. dojo example application

1. Project layout

For simple applications which don't have any custom JavaScript code, the dojo-simple project layout with only one module is recommended. Anything more advanced is better of using the project layout of the dojo-example project. This allows using dojo's shrunk builds which combine all JavaScript in one large cachable file, thus optimizing load time. As this shrinking processes has a massive impact on project build time (and makes the code difficult to debug), you want to be able to test without this shrinking step. This is tackled using the multi-module project. It has the following sub-projects :

Table 4.1. List of dojo-example projects

geomajas-dojo-example-client	JavaScript module code
geomajas-dojo-example-modules	generates dojo layer script with collected modules
geomajas-dojo-example-modules-shrinksafe	generates compressed version (shrinksafe) of dojo layer
geomajas-dojo-example-modules-war	web application with html and configuration

As opposed to the conventional practice of putting JavaScript directly in the war project, our module concept allows you to put this code on the classpath, including the possibility of using a separate jar project. This jar will be added to the web application as a normal java library and the JavaScript files in it will be served from the class path by the ResourceController which is linked in by the DispatcherServlet. The appropriate mapping should be added to the web.xml:

Example 4.1. Dispatcher servlet definition and mapping

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-c
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:META-INF/geomajasWebContext.xml</param-value>
    <description>Spring Web-MVC specific (additional) context files.</descr
  </init-param>
  <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/d/*</url-pattern>
</servlet-mapping>
```

By structuring JavaScript code as jar artifacts, our Maven dojo plug-in is capable of making this code accessible to the browser by importing a single dojo layer script in the web page. If the name of the layer is **example**, the import statements will be:

Example 4.2. Custom build script inclusion

```
<script type="text/javascript" src="d/resource/dojo/dojo.js"
  djConfig="parseOnLoad:true, usePlainJson :true"></script>
<script type="text/javascript" src="d/resource/example/example.js"
  geomajasConfig="showLog :false"></script>
```

As you can see, some configuration parameters can be passed by adding a geomajasConfig attribute to the script tag, exactly like dojo does with djConfig. The above import statements will be exactly the

same for the compressed or uncompressed version of the layer. While the compressed version contains all necessary code in a single file (using the dojo custom build facility), the uncompressed version is just a set of `dojo.require()` statements, one for each module.

Our Maven dojo plug-in has two different configurations, depending on whether the custom build should be applied or not. The two different configurations have been applied in the `geomajas-dojo-example-modules-shrinksafe` and `geomajas-dojo-example-modules` projects, respectively. By defining a Maven dependency in `geomajas-dojo-example-modules-war` on one or the other, the compressed or uncompressed version of the JavaScript libraries is used. The switch between the two options is managed by a Maven profile in the parent project (`geomajas-dojo-example`). This should be passed in the Maven build command:

- compressed/shrunked build: `mvn install`
- uncompressed build: `mvn -DskipShrink install`

2. Maven dojo plug-in

The Maven dojo plug-in is capable of executing a custom dojo build from within Maven. This essentially collects and/or compresses (using `shrinksafe`) JavaScript modules that have been bundled as jar artifacts.

A typical configuration looks as follows:

Example 4.3. Maven Dojo plugin configuration

```
<plugin>
  <groupId>org.geomajas</groupId>
  <artifactId>geomajas-maven-dojo</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <goals>
        <goal>build</goal>
      </goals>
      <configuration>
        <layerName>example</layerName>
        <localeList>en,fr,nl,pt,sp</localeList>
        <modules>
          <module>
            <groupId>org.geomajas</groupId>
            <artifactId>geomajas-dojo-client</artifactId>
            <version>${geomajas-face-dojo-version}</version>
            <requires>geomajas.geomajas</requires>
            <type>jar</type>
          </module>
          <module>
            <groupId>org.geomajas</groupId>
            <artifactId>geomajas-dojo-example-client</artifactId>
            <version>${project.version}</version>
            <requires>tutorial.tutorial</requires>
            <type>jar</type>
          </module>
        </modules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The plug-in has the following goals:

- `collect`: generates a dojo layer script with the necessary `dojo.require()` statements to import all the modules
- `build`: generates a compressed version (`shrinksafe`) of the dojo layer script

The following parameters can be configured:

Table 4.2. Dojo plug-in configuration parameters

Name	Description	Default value
<code>layerName</code>	The name of the target layer file (without the ".js" extension). The file is put in its own module directory and has to be imported as follows: <code><script src="d/resource/layerName/layerName.js" /></code>	modules
<code>localeList</code>	List of locales to be included in the build file (see dojo custom build parameters)	en
<code>layerOptimize</code>	Compression type to be used (see dojo custom build parameters). Options include "shrinksafe", "shrinksafe.keepLines", "packer".	shrinksafe
<code>modules</code>	A list of module tags, one for each JavaScript module. Beware: <ul style="list-style-type: none"> • the modules should be added as jar dependencies first. • The scope for the dependency can be provided for a shrink build (this reduces the size of the war). Each module tag should provide a <code>requires</code> parameter. This parameter is the argument of the <code>dojo.require()</code> statement needed to import the module from dojo.	empty list

The output of the plug-in consists of generated JavaScript files. The precise output depends on the goal. If the goal is "collect", it is a single file. If the goal is "build", it contains the complete contents of the release directory of the dojo build. In all cases, the JavaScript output is generated in the folder `<projectdir>/target/generated-resources/dojo`, which is added as a resource folder to the Maven project.