

# MoMo

Integrated Water Resources  
Management      Mongolia



## MoMo3 Workshop

2016-02-22 – 2016-02-26

# Table of Contents

<a href="#">Introduction</a>	0
<a href="#">Overview</a>	1
<a href="#">Prerequisites</a>	1.1
<a href="#">References</a>	1.2
<a href="#">Setup</a>	1.3
<a href="#">Data</a>	1.4
<a href="#">GeoServer</a>	2
<a href="#">Basics</a>	2.1
<a href="#">Administration</a>	2.1.1
<a href="#">Login and frontend overview</a>	2.1.1.1
<a href="#">About &amp; Status</a>	2.1.1.2
<a href="#">Server Status</a>	2.1.1.2.1
<a href="#">GeoServer Logs</a>	2.1.1.2.2
<a href="#">Contact Information</a>	2.1.1.2.3
<a href="#">About GeoServer</a>	2.1.1.2.4
<a href="#">Services</a>	2.1.1.3
<a href="#">Settings</a>	2.1.1.4
<a href="#">Security</a>	2.1.1.5
<a href="#">Publishing</a>	2.1.2
<a href="#">Prerequisites</a>	2.1.2.1
<a href="#">Workspaces</a>	2.1.2.2
<a href="#">Vector layers</a>	2.1.2.3
<a href="#">Import</a>	2.1.2.3.1
<a href="#">Store</a>	2.1.2.3.2
<a href="#">Layer</a>	2.1.2.3.3
<a href="#">Preview</a>	2.1.2.3.4
<a href="#">Style</a>	2.1.2.3.5
<a href="#">Raster layers</a>	2.1.2.4
<a href="#">Preparation</a>	2.1.2.4.1
<a href="#">Store</a>	2.1.2.4.2
<a href="#">Layer</a>	2.1.2.4.3
<a href="#">Preview</a>	2.1.2.4.4
<a href="#">Group layers</a>	2.1.2.5
<a href="#">Layer</a>	2.1.2.5.1
<a href="#">Preview</a>	2.1.2.5.2
<a href="#">Advanced</a>	2.2
<a href="#">REST</a>	2.2.1
<a href="#">Read</a>	2.2.1.1
<a href="#">Create</a>	2.2.1.2

Update	2.2.1.3
Delete	2.2.1.4
GeoWebCache	2.2.2
Prerequisites	2.2.2.1
Configure a new gridset	2.2.2.2
Configure a cached layer	2.2.2.3
Generate map tiles	2.2.2.4
Check cache directory	2.2.2.5
Check cache-headers	2.2.2.6
JavaScript	3
Beginners	3.1
Basics	3.1.1
Comments	3.1.1.1
Variables	3.1.1.2
Types	3.1.1.3
Equality	3.1.1.4
Numbers	3.1.2
Creation	3.1.2.1
Basic Operators	3.1.2.2
Advanced Operators	3.1.2.3
Strings	3.1.3
Creation	3.1.3.1
Concatenation	3.1.3.2
Length	3.1.3.3
Conditional Logic	3.1.4
If	3.1.4.1
Else	3.1.4.2
Comparators	3.1.4.3
Concatenate	3.1.4.4
Arrays	3.1.5
Indices	3.1.5.1
Length	3.1.5.2
Loops	3.1.6
For	3.1.6.1
While	3.1.6.2
Do...While	3.1.6.3
Functions	3.1.7
Declare	3.1.7.1
Higher order	3.1.7.2
Objects	3.1.8
Creation	3.1.8.1
Properties	3.1.8.2

---

Mutable	3.1.8.3
Reference	3.1.8.4
Prototype	3.1.8.5
Delete	3.1.8.6
Enumeration	3.1.8.7
Global footprint	3.1.8.8
Advanced	3.2
OpenLayers	4
Basics	4.1
Creating a map	4.1.1
Dissecting your map	4.1.2
Resources	4.1.3
Layers and Sources	4.2
WMS sources	4.2.1
Tiled sources	4.2.2
Proprietary tile providers	4.2.3
Vector data	4.2.4
Image vector source	4.2.5
Controls	4.3
Scale line control	4.3.1
Select interaction	4.3.2
Draw interaction	4.3.3
Modify interaction	4.3.4
Vector Topics	4.4
Formats	4.4.1
Styling concepts	4.4.2
Custom styles	4.4.3
Custom Builds	4.5
Concepts	4.5.1
Create custom builds	4.5.2
ExtJS	5
GeoExt	6
Synopsis	7
Glossary	

---

## Welcome to the workshop

# Introduction to core technologies behind the MoMo geoportal

## Sources

- [Workshop URL](#)
- [Download workshop \(ZIP\)](#)
- [Download workshop \(PDF\)](#)
- [Download workshop \(EPUB\)](#)

## Authors

- Marc Jansen ([jansen@terrestris.de](mailto:jansen@terrestris.de))
- Daniel Koch ([koch@terrestris.de](mailto:koch@terrestris.de))

# Overview

## Prerequisites

### Start the workshop

This workshop is intended to be executed in combination with an Linux image delivered on an USB flash drive especially prepared for this workshop. To run this image (and the workshop) please follow these steps:

1. Connect the provided USB flash drive to your computer and turn on the computer.
2. Normally you should see the Linux Mint boot screen similar to the image below after only a few seconds.



*Linux Mint bootscreen.*

**Note:** If your computer isn't booting Linux Mint, please ensure your PC is able and correctly configured to boot from an USB device. To do so, access your BIOS by pressing `DEL` or `F2` during the early boot process (usually you will see the correct key displayed on-screen during the boot process). Press the required key at the correct time and your computer's BIOS will appear. Once you're in the BIOS, try to find a menu called something like `Boot` or similar. Navigate to this menu and look for some sort of an entry called `Boot Option Priorities` or similar. Usually you will find the boot order as a priority list in which you can move the USB device boot option up to the top of this list. Now save your changes and exit the BIOS to restart your computer.

### Login to Linux

After the successful boot you should see the Linux Mint login screen as shown in the image below:



*Linux Mint login screen.*

Use the following credentials to login:

- User: momo-ws
- Password: momo-ws

Congratulations! Now you are ready to start the workshop! 



# Workshop reference book

Here you will find some useful informations about the workshop image.

## Credentials

- Linux:
  - User: momo-ws
  - Password: momo-ws
- GeoServer:
  - User: admin
  - Password: momo-ws
- PostgreSQL:
  - User: momo-ws
  - Password: momo-ws

## Useful paths

- Your home directory: /home/momo3-ws
- Workshop directory: /home/momo3-ws/materials
- Tomcat webapp directory: /var/lib/tomcat8/webapps

## Useful terminal commands

As you may not familiar with Linux you will find a small list containing the most helpful terminal commands used in this workshop.

### Navigation

- Navigate to a directory: \$ cd {PATH\_TO\_DIRECTORY}
- Navigate to the upper directory: \$ cd ..
- Navigate to your home directory: \$ cd ~
- Navigate to the root directory: \$ cd /
- List all files and directories of a folder (inlong list format): \$ ls -l or \$ ll

### File and directory manipulation

- Creation
  - Of a file: \$ touch {FILE\_PATH\_AND\_NAME}
  - Of a directory: \$ mkdir {DIRECTORY\_PATH\_AND\_NAME}
- Removal
  - Of a file: \$ rm {FILE\_PATH\_AND\_NAME}
  - Of a directory: \$ rm -rf {DIRECTORY\_PATH\_AND\_NAME}
- Change ownership:
  - For a single file \$ chown {GROUP\_NAME}:{USER\_NAME} {FILE\_PATH\_AND\_NAME}
  - For a completer folder (recursively) \$ chown -R {GROUP\_NAME}:{USER\_NAME} {FILE\_PATH\_AND\_NAME}

### Execution

- Make a file executable: \$ chmod +x {FILE\_PATH\_AND\_NAME}

- Run an executable file: `$ ./myExecutable.sh`

## Compress and extract

- Create an archive: `$ tar cvzf {FILE_PATH_AND_NAME}.tar.gz`
- Extract an archive: `$ tar xvzf {FILE_PATH_AND_NAME}.tar.gz`

## Services

- Start/Stop/Restart PostgreSQL: `$ sudo service postgresql start|stop|restart`
- Start/Stop/Restart Apache: `$ sudo service apache2 start|stop|restart`
- Start/Stop/Restart Tomcat: `$ sudo service tomcat8 start|stop|restart`

## Other useful commands

- Execute command with super user (root) permissions : `$ sudo {COMMAND_TO_EXECUTE}`
- Show manual of a tool: `$ man {COMMAND_TOOL_NAME}`
- Show terminal history: `$ history`
- Live-monitoring of a changing file (e.g. a logfile): `$ tail -f {FILE_PATH_AND_NAME}`
- Execute the last command used: `$ !!`

## Exercises

1. Open the terminal by clicking the `Terminal` icon () in the bottom toolbar.
2. Navigate to your home directory and create a folder named `notes` by typing:

```
$ cd ~
```

```
$ mkdir notes
```

3. Go to the newly created folder and create a file named `workshop-notes.md` by typing:

```
$ cd notes/
```

```
$ touch workshop-notes.md
```

4. Open this file with `gedit` and enter `Linux is great!` by typing:

```
$ gedit workshop-notes.md
```

# Workshop Linux Mint Setup

The provided workshop Linux image is just a Linux Mint with pre-installed and configured programs and tools. This page lists the changes to the default system configuration only and is no integral part of the workshop.

- Operating system:
  - Linux Mint [17.3 Cinnamon Edition](#) (32bit)
- Additionally installed software:
  - [Apache 2](#)
    - Installed from package manager
    - Linked home directory `/home/momo` to <http://localhost:80/momo>
    - Added to autostart
  - [Apache Tomcat 8](#)
    - Installed from [here](#)
    - Added to autostart
  - [GeoServer 2.8.2](#)
    - Installed from [here](#)
    - Installed plugins:
      - [pyramid-plugin](#)
      - [wps-plugin](#)
      - [oracle-plugin](#)
      - [importer-plugin](#)
    - Published via Apache 2 on port 80
      - <http://localhost:80/geoserver>
  - [Atom Editor](#)
    - Installed from [here](#)
    -  MJ list additional plugins
  - [Chrome](#)
    - Installed from [here](#)
  - [PostgreSQL / PostGIS](#)
    - Installed from package manager
    - Added to autostart
  - [git](#)
    - Installed from package manager
  - [nvm](#)
    - Installed from [here](#)
  - [bash-git-prompt](#)
    - Installed from [here](#)
  - [QGIS](#)
    - Installed from [here](#)
  - [GDAL](#)
    - Installed from package manager
- Removed packages:

```
$ sudo apt-get remove thunderbird vlc vlc-plugin-notify vlc-plugin-pulse \
  vlc-data vlc-nox totem-common brasero banshee gimp hexchat pidgin totem \
  seahorse cowsay mint-backgrounds-qiana mint-backgrounds-rafaela \
  mint-backgrounds-rebecca mint-backgrounds-rosa sox ttf-indic-fonts-core \
  ttf-punjabi-fonts ttf-wqy-microhei fonts-kacst fonts-kacst-one \
  fonts-khmeros-core fonts-lao fonts-lklug-sinhala fonts-thai-tlwg \
  fonts-tibetan-machine fonts-tlwg-garuda fonts-tlwg-kinnari fonts-tlwg-loma \
  fonts-tlwg-norasi fonts-tlwg-purisa fonts-tlwg-sawasdee fonts-wqy-microhe \
  fonts-noto fonts-sil-abyssinica fonts-sil-padauk fonts-takao-pgothic \
  fonts-tlwg-umpush fonts-tlwg-waree gimp-help-en firefox firefox-locale-en \
  simple-scan transmission-common transmission-gtk mintwelcome
```

- Removed remaining dependencies no longer needed

```
$ sudo apt-get autoremove
```

- Updated and upgraded packages to latest version

```
$ sudo apt-get -y update && sudo apt-get -y upgrade
```

- Data used in the workshop:

- Natural Earth Large scale data, 1:10m
  - Cultural
    - Download [here](#)
  - Physical
    - Download [here](#)
  - Raster (Ocean bottom)
    - Download [here](#)
  - Extracted to `~/materials/natural_earth`
- OSM sample export Mongolia
  - Download [here](#)
  - Extracted to `~/materials/osm_mongolia`

## Workshop data

This section shows the data and the data sources we'll use in the workshop.

### Natural Earth ( `~/materials/natural_earth` )

The [Natural Earth](#) dataset is a free collection of vector and raster data published by the North American Cartographic Information Society to encourage mapping. In this workshop we will use the datasets `Cultural`, `Physical` and `Raster` in large scale (1:10m).

### OpenStreetMap ( `~/materials/osm_mongolia` )

[OpenStreetMap](#) is a free editable map database of the world. In this workshop we use a small excerpt of the huge OSM database focused on mongolia which is provided by the company geofabrik as a collection of single [shapefiles](#).

## GeoServer

GeoServer is an Open Source software server written in Java that allows users to share and edit geospatial data. Designed for interoperability, it publishes data from any major spatial data source using open standards. GeoServer is the reference implementation of the Open Geospatial Consortium (OGC) Web Feature Service (WFS) and Web Coverage Service (WCS) standards, as well as a high performance certified compliant Web Map Service (WMS). GeoServer forms a core component of the Geospatial Web.

In this module we will focus on the geodata-management using the GeoServer administration frontend. Thus we will learn both how to set the most common configure options and how to load, publish, style, and share geospatial data with GeoServer.

The workshop is subdivided into two main categories:

- [GeoServer Basics \(Administration & Publishing\)](#)
- [GeoServer Advanced \(REST & GeoWebCache\)](#)

*Please note:* Parts of this workshop are heavily inspired by workshops prepared by the [GeoServer community](#) and [boundless](#). Feel free to look at these sources for further informations.

## GeoServer Basics

In this section we'll learn how to deal with the most common use cases when working with the GeoServer: Manage the running GeoServer instance by adjusting the most important configuration settings and publishing geospatial data.

- [Administration settings](#)
- [Publishing geospatial data](#)

# Administration

This module will give you a brief introduction into the GeoServer administration frontend where we'll focus on the mainly used configuration settings. The basis structure of the following chapters is directly inspired by GeoServers frontend composition (Note: The `Data` and `Tile Caching` sections will be treated in the particular chapters [Publishing](#) and [GeoWebCache](#)).

- [Login and frontend overview](#)
- [About & Status](#)
- [Services](#)
- [Settings](#)
- [Security](#)

# Login and frontend overview

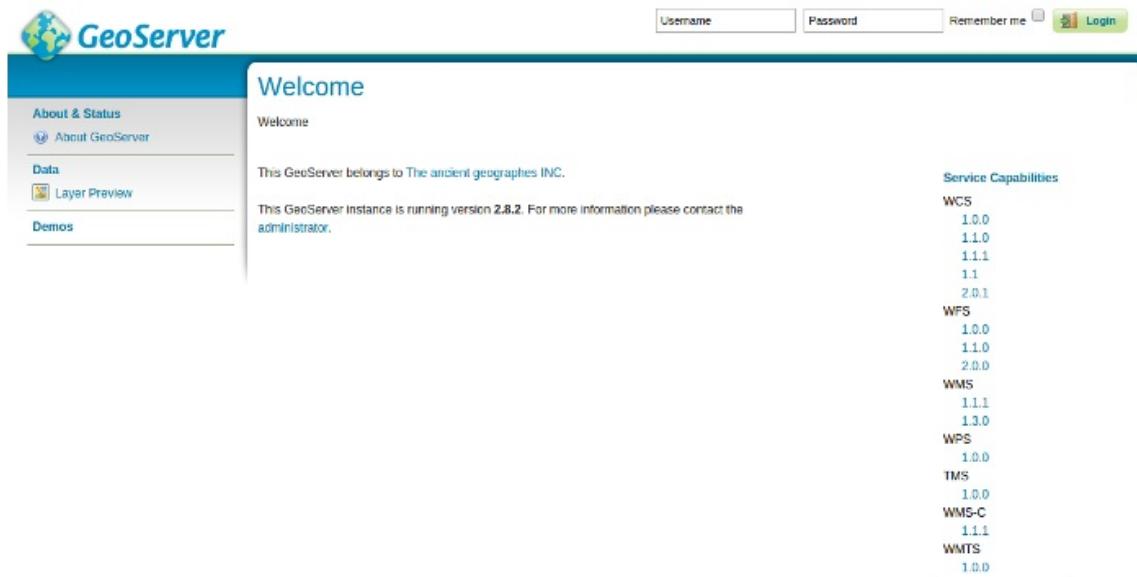
GeoServer includes a web-based administration interface. Most GeoServer configuration can be done through this interface, without the need to edit configuration files by hand or use an API. This section will give a brief overview to the web interface. Subsequent sections will use the web interface in greater detail.

## Welcome page

To open the GeoServer UI open the following address in your browser:

<http://localhost/geoserver>

The initial page is called the `Welcome page`. To return to the Welcome page from anywhere, just click the GeoServer logo in the top left corner of the page.



While the unauthenticated/anonymous Welcome page is not void of features, it really just lets you see things (configured on geoserver) but not touch them (and make configuration changes).

For security reasons, most GeoServer configuration tasks require you to be logged in first. By default, the GeoServer administration credentials are `admin` and `geoserver`, although this can and should be changed (see chapter [Security](#)).

## Login

Log in into GeoServer by using the default administration credentials from above.

Welcome

Welcome

This GeoServer belongs to [The ancient geographies INC.](#)

22 Layers	<a href="#">Add layers</a>
12 Stores	<a href="#">Add stores</a>
9 Workspaces	<a href="#">Create workspaces</a>

**Service Capabilities**

- WCS**
  - 1.0.0
  - 1.1.0
  - 1.1.1
  - 1.1
  - 2.0.1
- WFS**
  - 1.0.0
  - 1.1.0
  - 2.0.0
- WMS**
  - 1.1.1
  - 1.3.0
- WPS**
  - 1.0.0
- TMS**
  - 1.0.0
- WMS-C**
  - 1.1.1
- WMPS**
  - 1.0.0

This GeoServer instance is running version 2.8.2. For more information please contact the administrator.

*GeoServer welcome page*

After logging in, many more options will be displayed.

## Basic layout

Use the links on the left side column to manage GeoServer, its services, data, security settings and more. Also on the main page are direct links to the capabilities documents for each service (WFS, WMS, WCS). We'll be using the links on the left under `data` - among them Layer Preview, Workspaces, Stores, Layers, Layer Groups, and Styles - very often in this workshop, so it is good to familiarize yourself with their location. Thus we'll start this module in the following chapters by introducing the frontend structure.

## About and Status

This section of the web administration interface provides a high level overview of the running application. Contact information for OGC services is also managed here.

- [Server Status](#)
- [GeoServer Logs](#)
- [Contact Information](#)
- [About GeoServer](#)

# Server Status

The Server Status page shows you many metainformation about the current GeoServer configuration and overall status.

## Server Status

Summary of server configuration and status

	Action	
Data directory	/var/lib/tomcat7/webapps/geoserver/data	
Locks	0	<a href="#">Free locks</a>
Connections	6	
Memory Usage	571 MB	<a href="#">Free memory</a>
JVM Version	Oracle Corporation: 1.7.0_80 (Java HotSpot(TM) 64-Bit Server VM)	
Available Fonts	GeoServer can access 638 different fonts. <a href="#">Full list of available fonts</a>	
Native JAI	false	
Native JAI ImageIO	false	
JAI Maximum Memory	1 GB	
JAI Memory Usage	0 KB	<a href="#">Free memory</a>
JAI Memory Threshold	75.0	
Number of JAI Tile Threads	7	
JAI Tile Thread Priority	5	
ThreadPoolExecutor Core Pool Size	5	
ThreadPoolExecutor Max Pool Size	10	
ThreadPoolExecutor Keep Alive Time (ms)	30000	
Update Sequence	196	
Resource Cache		<a href="#">Clear</a>
Configuration and catalog		<a href="#">Reload</a>

*Server status page.*

It provides a useful diagnostic tool in a testing and production environment and should be your first place to go if are facing any problem with your running GeoServer instance. *Note:* Certainly it's always useful to have a look at the [Logs](#) additionally.

## Status indicators

The following table describes the current status indicators (as described [here](#)).

Option	Description
Data directory	The absolute path to your data directory.
Locks	A WFS has the ability to lock features to prevent more than one person from updating the feature at one time. If data is locked, edits can be performed by a single WFS editor. When the edits are posted, the locks are released and features can be edited by other WFS editors. A zero in the locks field means all locks are released. If locks is non-zero, then pressing <code>Free locks</code> releases all feature locks currently held by the server, and updates the field value to zero.
Connections	Refers to the numbers of vector stores, in the above case 6, that were able to connect.
Memory Usage	The amount of memory currently used by GeoServer. Clicking on the <code>Free Memory</code> button, cleans up memory marked for deletion by running the garbage collector.
JVM Version	Denotes which version of the JVM (Java Virtual Machine) is been used to power the server.
Available Fonts	A list of all fonts GeoServer has access to. These can be referenced in the layer style.
Native JAI	GeoServer uses Java Advanced Imaging (JAI) framework for image rendering and coverage manipulation. When properly installed (true), JAI makes WCS and WMS performance faster and more efficient.
Native JAI ImageIO	GeoServer uses JAI Image IO (JAI) framework for raster data loading and image encoding. When properly installed (true), JAI Image I/O makes WCS and WMS performance faster and more efficient.
JAI Maximum Memory	Expresses in bytes the amount of memory available for tile cache.
JAI Memory Usage	Run-time amount of memory is used for the tile cache. Clicking on the <code>Free Memory</code> button, clears available JAI memory by running the tile cache flushing.
JAI Memory Threshold	Refers to the percentage, e.g. 75, of cache memory to retain during tile removal. JAI Memory Threshold value must be between 0.0 and 100.
Number of JAI Tile Threads	The number of parallel threads used by to scheduler to handle tiles.
JAI Tile Thread Priority	Schedules the global tile scheduler priority. The priority value is defaults to 5, and must fall between 1 and 10.
ThreadPoolExecutor Core Pool Size	The imageMosaic reader may load, in parallel, different files that make up the mosaic by means of a ThreadPoolExecutor. A global ThreadPoolExecutor instance is shared by all the readers supporting and using concurrent reads. Here the current core pool size of the ThreadPoolExecutor is listed.
ThreadPoolExecutor Max Pool Size	Here the current maximum core pool size of the ThreadPoolExecutor is listed.
ThreadPoolExecutor Keep Alive Time (ms)	The time to be waited by the ThreadPoolExecutor before terminating an idle thread in case there are more threads than available in the core pool size.
Update Sequence	Refers to the number of times the server configuration has been modified.
Resource Cache	GeoServer does not cache data, but it does cache connection to stores, feature type definitions, external graphics, font definitions and CRS definitions as well. The <code>clear</code> button forces those caches to empty and makes GeoServer reopen the stores and re-read image and font information, as well as the custom CRS definitions stored in <code> \${GEOSERVER_DATA_DIR}/user_projections/epsg.properties</code> .
Configuration and catalog	GeoServer keeps in memory all of its configuration data. If for any reason that configuration information has become stale (e.g. an external utility has modified the configuration on disk) the <code>Reload</code> button will force GeoServer to reload all of its configuration from disk.

## Exercise

1. Open up the `Server Status` page on your GeoServer and press the button `Free memory` besides `Memory Usage`. What do you observe?



# GeoServer Logs

GeoServer displays the contents of the application logs directly through the web interface. Reading the logs can be very helpful when troubleshooting. To view the logs, click on `GeoServer Logs` on the left under `About & Status`.

## GeoServer Logs

Show the GeoServer log file contents

```
Maximum console lines 1000 Refresh
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)
at org.geoserver.filters.SessionDebugFilter.doFilter(SessionDebugFilter.java:48)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)
at org.geoserver.filters.FlushSafeFilter.doFilter(FlushSafeFilter.java:44)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)
at
org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:88)
at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:76)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:220)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:122)
at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:501)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:170)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:98)
at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:950)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:116)
at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:408)
at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1041)
at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:607)
at org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.run(JIoEndpoint.java:313)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:745)
Caused by: org.geowebcache.storage.StorageException: Thread 41 Unknown layer momo-rest:countries_rest. Check the logfiles, it may not have loaded properly.
    at org.geowebcache.storage.CompositeBlobStore.store(CompositeBlobStore.java:309)
    at org.geowebcache.storage.CompositeBlobStore.delete(CompositeBlobStore.java:125)
    at org.geoserver.gwc.ConfigurableBlobStore.delete(ConfigurableBlobStore.java:130)
    at org.geowebcache.storage.DefaultStorageBroker.delete(DefaultStorageBroker.java:53)
    at org.geoserver.gwc.GWC.layerRemoved(GWC.java:564)
    ... 109 more
Caused by: org.geowebcache.GeoWebCacheException: Thread 41 Unknown layer momo-rest:countries_rest. Check the logfiles, it may not have loaded properly.
    at org.geowebcache.layer.TileLayerDispatcher.getTileLayer(TileLayerDispatcher.java:105)
    at org.geowebcache.storage.CompositeBlobStore.forLayer(CompositeBlobStore.java:326)
    at org.geowebcache.storage.CompositeBlobStore.store(CompositeBlobStore.java:307)
    ... 109 more
2016-02-05 10:14:38,625 INFO [catalog.rest] - DELETE layer countries_rest
2016-02-05 10:14:38,627 INFO [catalog.rest] - DELETE feature typedb_momo_ws_rest,countries_rest
```

[Download the full log file](#)

*Logging page.*

## Exercise

1. Open up the `GeoServer Logs` section and investigate the last entries in your logfile. Can you find any conspicuous or interesting entry?

## Contact Information

Each OGC web service served by the GeoServer contains the contact details associated with the server as part of their capabilities document. You can read and manipulate these information in the section `Contact Information` under `About & Status`.

### Contact Information

Set the contact information for this server.

**Contact**

Claudius Ptolomaeus

**Organization**

The ancient geographies INC

**Position**

Chief geographer

**Address Type**

Work

**Address**

**Address Delivery Point**

**City**

Alexandria

**State**

**ZIP code**

**Country**

Egypt

**Address Electronic Mail Address**

**Telephone**

**Fax**

**Email**

claudius.ptolomaeus@gmail.com

**Submit**

**Cancel**

*Contact page.*

## Exercise

1. Fill in your contact informations in `Contact Information` page.
2. Click `Submit`.
3. Open the following link to request the `GetCapabilites` document provided by your GeoServer instance and verify your changes:

<http://localhost/geoserver/ows?SERVICE=WMS&REQUEST=GetCapabilities&VERSION=1.1.0>

*Hint:* You can also request the above GetCapabilities document by visiting the [Welcome Page](#) and select the links on the right handed side.



## About GeoServer

The about page lists general informations about the GeoServer you're running. These listings are especially helpful, if you want to contact any support or if you want to know the installed GeoServer version for getting compatible community modules and extensions.

*Note:* You'll see a link `Documentation` at the end of the page. This link will guide you to the official documentation of the project and is a **very** helpful address if you need any information about GeoServer.

### About GeoServer

General information about GeoServer

#### Build Information

##### Version

2.8.2

##### Git Revision

f1366aa5e0c9d477b9c6a05fd31d59e0e01985f9

##### Build Date

23-Jan-2016 02:21

##### GeoTools Version

14.2 (rev 73d5c95ed430b6c891eb2f0dfebb742ba01780fe)

##### GeoWebCache Version

1.8.1 (rev f22447f59ca37f72bfe6bc746d834b0e73e7c3fc/f22447f59ca37f72bfe6bc746d834b0e73e7c3fc)

#### More Information

GeoServer is a full transactional Java implementation of the Open Geospatial Consortium's specifications for Web Feature Service (WFS) and Web Coverage Service (WCS) with an integrated Web Map Service (WMS).

This web administration interface allows for easy configuration of GeoServer. After logging in, please use the menus on the left to navigate through the interface. The About and Status menu lists technical details about the running GeoServer instance. The Data menu is used to configure data sources and styling. The Service menu provides service-wide configuration options. The Settings menu provides configurations options that apply to all services (i.e. server-wide). The Tile Caching menu allows configuration of the embedded tile cache. The Security menu allows configuration of access controls (authentication and authorization). The Demos menu has examples of GeoServer in action. The Tools menu allows access to administrative tools (e.g. for loading data).

Useful Links:

[Documentation](#)

[Wiki](#)

[Bug Tracker](#)

*About page.*

## Exercise

1. Follow the mentioned `Documentation` link and have a quick look around to see the valuable resources available here!

# Services

The `Services` section is for configuring the services published by GeoServer, where we can manage:

- The metadata, resource limits, and SRS availability for WCS.
- The metadata, feature publishing, service level options, and data-specific output for WFS.
- The metadata, resource limits, SRS availability, and other data-specific output for WMS.

The following exercises will give you a brief introduction in the most important administration options available here.

For further instructions please have a look at the official GeoServer [documentation](#).

## Limit SRS output list

The default GetCapabilities document contains a comprehensive list of all available spatial reference systems (SRS) of your GeoServer WMS server instance. Generally it is not needed, that you list all supported systems as you typically want to publish data in a limited list of projections only. Furthermore limiting this list will reduce the file size of the responding GetCapabilites document!

1. Open the following URL in your browser to open the GetCapabilites document of your GeoServer instance: [GetCapabilites](#)
2. In the resulting XML document find the element `Layer` which contains the large list (~6000 lines) of all supported EPSG projections.

In the next step we'll limit this list to contain the systems `EPSG:4326`, `EPSG:3857` and `EPSG:900913` only.

1. Navigate to `Services > WMS`.
2. Find the description field entitled with `Limited SRS list` and fill in `4326, 3857, 900913`.

**Limited SRS list**

4326, 3857, 900913

Output bounding box for every supported CRS

3. Click `Submit`.
4. Reopen the [GetCapabilites document](#) and you will note, that the document is reduced by a huge amount of lines.

## Disable WFS-T

GeoServer is configured to act as a fully transactional Web Feature Service server per default. A transactional WFS allows creation, deletion, and updating of features. This implies, that each published feature type can be edited by any client. Generally you don't want to allow clients to edit your data published with GeoServer (unless you really want to allow it), especially if your GeoServer is accessible globally through the Internet. In the next iteration we're going to disable the WFS-T functionality.

1. Navigate to `Services > WFS`.
2. Find the checkbox group entitled with `Service Level` and select the level `Basic` to disable WFS-T compatibility.

**Service Level**

- Basic
- Transactional
- Complete

3. Click `Submit`.
4. Optional: Import a given WMS layer into QGIS and try to edit it.



## Settings

The `Settings` section involves configuration settings that apply to the entire server. Again, instead of explaining each checkbox, we'll focus on the most important administration tools available in this section and its subsections.

For further instructions please have a look at the official GeoServer [documentation](#).

### Set handle data and configuration problems

This setting determines how GeoServer will respond when a layer becomes inaccessible for some reason. By default, when a layer has an error (for example, when the default style for the layer is deleted), a service exception is printed as part of the capabilities document, making the document invalid. For clients that rely on a valid capabilities document, this can effectively make a GeoServer appear to be "offline". As administrator you may prefer to configure GeoServer to simply omit the problem layer from the capabilities document, thus retaining the document integrity and allowing clients to connect to other published layers.

1. Go to `Settings` > `Global`.
  2. Select `Skipping misconfigured layers` in the combo entitled with `Handle data and configuration problems in capabilities documents by...`
- `Handle data and configuration problems in capabilities documents by...`

`Skipping misconfigured layers` ▾
3. Click `Submit`.

### Reduce the number of decimals

To reduce the output size returned in a GetFeature response (and therefore optimizing the bandwith) we can restrict the number of decimal places in a GetFeature response. Here we will set the value to 2.

1. Go to `Settings` > `Global`.
  2. Set `Number of Decimals` to 2.
- |                                 |
|---------------------------------|
| <code>Number of Decimals</code> |
| 2                               |
3. Click `Submit`.

### Change logging level

At some point in your production (or development) usage of GeoServer you'll encounter a problem where you'll need to get further and detailed informations to find the cause of the problem. In this case you can increase the level of logging. The following steps will guide you how to set the logging level to a verbose-like level containing valuable informations about the image processing process.

1. Go to `Settings` > `Global`.
2. Select the check box beside `Verbose Exception Reporting` to return service exceptions with full Java stack traces.
3. Find the description field entitled with `Logging Profile` and select the profile `VERBOSE_LOGGING.properties` to enable the `DEBUG` level logging on GeoTools and GeoServer.

Verbose Messages

Verbose Exception Reporting

Enable Global Services

Handle data and configuration problems in capabilities documents by...

Choose One ▾

Number of Decimals  
8

Character Set  
UTF-8

Proxy Base URL

Logging Profile  
 DEFAULT\_LOGGING.properties  
 GEOSERVER\_DEVELOPER\_LOGGING.properties  
 GEOTOOLS\_DEVELOPER\_LOGGING.properties  
 PRODUCTION\_LOGGING.properties  
 QUIET\_LOGGING.properties  
 TEST\_LOGGING.properties  
**VERBOSE\_LOGGING.properties**

Log to StdOut

4. Click Submit .

5. To validate the changes please open the OpenLayers layer preview for a layer of your choice ( Data → Layer Preview ) and zoom slightly into the map.

6. Go to About & Status → GeoServer Logs to open up the logfile and scroll down to the end of the file. Have a look at the timestamp and you'll notice plenty of logs for just a simple GetMap request (you did in the layer preview).

```

2016-02-09 11:58:50,475 DEBUG [geoserver.filters] - Compressing output for mimetype: text/html;charset=UTF-8
2016-02-09 11:58:50,477 DEBUG [filter.GeoServerSecurityContextPersistenceFilter$1] - SecurityContextHolder now cleared, as request processing completed
2016-02-09 11:58:52,081 DEBUG [security.IncludeQueryStringAntPathRequestMatcher] - Checking match of request : 'Path: /web/, QueryString: wicket:interface:=75:form:IFormSubmitListener::'; against '/web/**'
2016-02-09 11:58:52,081 DEBUG [security.IncludeQueryStringAntPathRequestMatcher] - Matched Path: /web/, QueryString: wicket:interface:=75:form:IFormSubmitListener:: with /web/**
2016-02-09 11:58:52,081 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,081 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,081 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,081 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,082 DEBUG [org.geoserver] - Thread 35 locking in mode WRITE
2016-02-09 11:58:52,082 DEBUG [org.geoserver] - Thread 35 got the lock in mode WRITE
2016-02-09 11:58:52,083 DEBUG [org.geoserver] - Thread 35 releasing the lock in mode WRITE
2016-02-09 11:58:52,083 DEBUG [filter.GeoServerSecurityContextPersistenceFilter$1] - SecurityContextHolder now cleared, as request processing completed
2016-02-09 11:58:52,087 DEBUG [security.IncludeQueryStringAntPathRequestMatcher] - Checking match of request : 'Path: /web/, QueryString: wicket:bookmarkablePage:=org.geoserver.web.admin.LogPage&lines=1000'; against '/web/**'
2016-02-09 11:58:52,088 DEBUG [security.IncludeQueryStringAntPathRequestMatcher] - Matched Path: /web/, QueryString: wicket:bookmarkablePage:=org.geoserver.web.admin.LogPage&lines=1000 with /web/**
2016-02-09 11:58:52,088 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,088 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,088 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,088 TRACE [ows.OWSHandlerMapping] - No handler mapping found for [/web/]
2016-02-09 11:58:52,089 DEBUG [org.geoserver] - Thread 44 locking in mode WRITE
2016-02-09 11:58:52,089 DEBUG [org.geoserver] - Thread 44 got the lock in mode WRITE
2016-02-09 11:58:52,091 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/css/blueprint/screen.css to file (URI is not hierarchical), falling back to the inputstream for polling
2016-02-09 11:58:52,092 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/css/blueprint/print.css to file (URI is not hierarchical), falling back to the inputstream for polling
2016-02-09 11:58:52,092 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/css/geoserver.css to file (URI is not hierarchical), falling back to the inputstream for polling
2016-02-09 11:58:52,093 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/css/blueprint/ie.css to file (URI is not hierarchical), falling back to the inputstream for polling
2016-02-09 11:58:52,093 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/js/jquery-1.2.6.min.js to file (URI is not hierarchical), falling back to the inputstream for polling
2016-02-09 11:58:52,094 DEBUG [geoserver.web] - cannot convert url: jar:file:/var/lib/tomcat7/webapps/geoserver/WEB-INF/lib/gs-web-core-2.8.2.jar!/org/geoserver/web/js/jquery.inline-info.js to file (URI is not hierarchical), falling back to the inputstream for polling

```

7. As the given settings are not really needed for the moment (we aren't facing any problems) we can reset the logging level to `DEFAULT_LOGGING.properties`.

# Security

GeoServer has a robust security subsystem, modeled on Spring Security. Most of the security features are available through the Web administration interface. A detailed explanation of all security options is far beyond the goals of this workshop and the default settings are almost good enough for the basic usage of GeoServer as well. Here we'll focus on the most important fact after installing a GeoServer: Changing the default `admin` user and master password. (You may have recognized the appropriate warnings on the GeoServer welcome page.)

For further instructions please have a look at the official GeoServer [documentation](#).

## Change master password

The master password serves two purposes:

- Protect access to the keystore.
- Protect access to the GeoServer Root account (The root account is always active, regardless of the state of the security configuration. Much like its UNIX-style counterpart, this account provides "super user" status, and is meant to provide an alternative access method for fixing configuration issues. The user name for the root account is `root`. Its name cannot be changed.).

To change the master password follow these steps:

1. Go to `Security > Passwords`.
2. Select `Change password`.
3. In the upcoming form use the following values to change the password to `momo-ws`:
  - *Current password:* `geoserver`
  - *New password/Confirmation:* `momo-ws` (Please use a more secure password in a real world GeoServer usage!)
4. Click `Change Password`.

## Change admin user password

To change the password for the user `admin` please follow these steps:

1. Go to `Security > Users, Groups, Roles`.
2. Open panel `Users/Groups` where you'll see a list of all current users of your GeoServer instance. At the moment (and in most future applications) you'll find the user `admin` only.
3. Select the user `admin` by clicking on its username.
4. In the upcoming form use the following values to change the password to `momo-ws`:
  - *Password/Confirm Password:* `momo-ws` (Please use a more secure password in a real world GeoServer usage!)

User name	admin
<input checked="" type="checkbox"/> Enabled	
Password	*****
Confirm password	*****

5. Click `Save`.

# Publishing

In this section we'll learn the core principals about the GeoServer data management and how to prepare and import (vector-/raster) data into GeoServer and how to publish it as vector, raster and group layer.

- [Publish vector layer](#)
- [Publish raster layer](#)
- [Publish group layer](#)

## Prerequisites

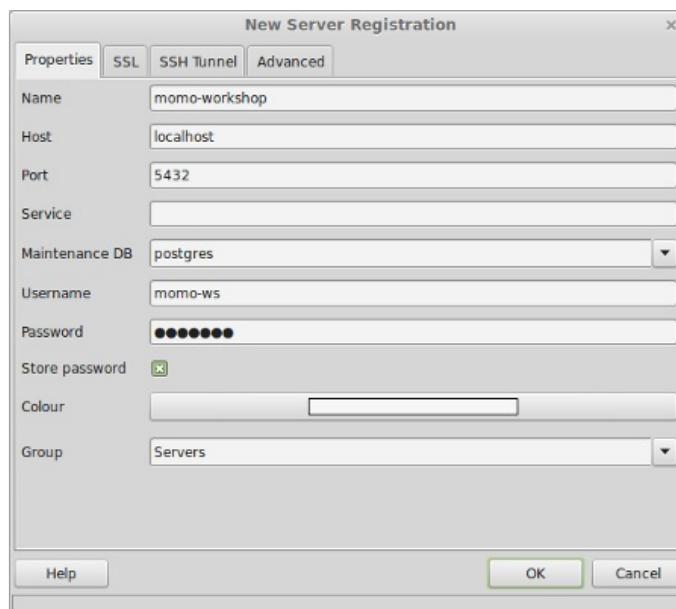
Before we can start importing and reading data in this modules, we have to create a new database on the (already installed) PostgreSQL database server as data source for new layers. Thus we will use the administration tool `pgAdmin III`. Let's start by opening it:

1. Click `Menu` in the lower left corner and search for `pgadmin`
2. In the resulting list select `pgAdmin III` ( ) to open the tool.

## Add a new server connection in pgAdmin

Once pgAdmin has started we can create a new connection to our PostgreSQL database server:

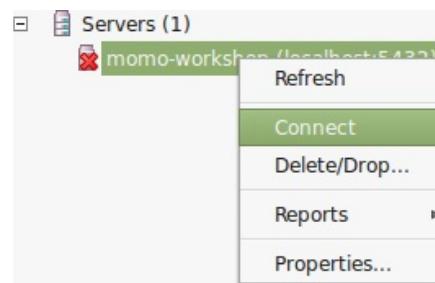
1. Create a new server connection by selecting `File > Add Server` in the top menu bar and enter the following:
  - o `Name`: momo-workshop
  - o `Host`: localhost
  - o `Port`: 5432
  - o `Username`: momo-ws
  - o `Password`: momo-ws
  - o `Store password`: checked



2. Click `OK`

## Creating a database

Now we can connect to this server by a double click on the newly created entry in the left hand sided `Object browser` (or open the context menu for this entry and select `connect` as shown below).



*Open up the server connection*

Within the next steps we will create a new database on this database server:

1. Open the SQL-Query window by clicking the icon in the upper toolbar. Note: If the icon is greyed out, select the existing database `postgres` first.
2. Copy the following SQL block into the SQL-Query window:

```
CREATE DATABASE db_momo_ws
WITH OWNER = momo-ws
ENCODING = 'UTF8'
TABLESPACE = pg_default
CONNECTION LIMIT = -1;
```

3. Click `Execute query` () in the upper toolbar to run the query.
4. After successful execution go back to the `Object browser`, select the server and refresh the actual view (by pressing `Refresh` the selected object () in the top toolbar) and ensure you have a new database entry named `db_momo_ws` present.
5. Close the SQL-Query window.

## Creating a schema

Once the database is created, we'll create a new schema in this database. This schema will be used to store any geodata table we are going to import in this workshop.

1. Select the newly created database `db_momo_ws` in the `Object browser` and open the SQL window (). If you haven't closed the SQL-Query window before, please verify that you are connected to the correct database in the upper toolbar. Otherwise all subsequent SQL queries will be executed on the wrong database!
2. Copy the following SQL block into the SQL-Query window to create a new schema named `geodata` :

```
CREATE SCHEMA geodata
AUTHORIZATION momo-ws;
```

3. Click `Execute query` () to run the query.
4. Refresh the `Object browser` and ensure the new schema is being created in the database `db_momo_ws`.

## Enable spatial functionality

In the final step we will add support for geographic objects by enabling the spatial database extension PostGIS for our database `db_momo_ws`.

1. Open the SQL window (if not already opened) and paste in the following SQL block to spatially enable the database `db_momo_ws` :

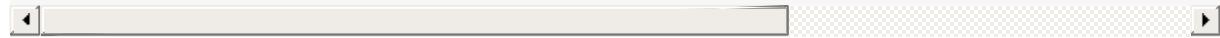
```
CREATE EXTENSION postgis;
```

2. Click `Execute query` to run the query.
3. Ensure the extension is being successfully installed by executing the following query:

```
SELECT PostGIS_full_version();
```

The corresponding output should look like:

```
"POSTGIS="2.1.2 r12389" GEOS="3.4.2-CAPI-1.8.2 r3921" PROJ="Rel. 4.8.0, 6 March 2012" GDAL="GDAL 1.10.1, released
```



# Workspace

A workspace (sometimes referred to as a namespace) is the name for a notional container for grouping similar data together. It is designed to be a separate, isolated space relating to a certain project. Using workspaces, it is possible to use layers with identical names without conflicts.

Workspaces are usually denoted by a prefix to a layer name or store name. For example, a layer called streets with a workspace prefix called nyc would be referred to by nyc:streets. This would not conflict with another layer called streets in another workspace called dc (dc:streets).

Stores and layers must all have an associated workspace. Styles may optionally be associated with a workspace, but can also be global.

Technically, the name of a workspace is a URI, not the short prefix. A URI is a Uniform Resource Identifier, which is similar to a URL, but does not need to resolve to a web site. In the above example, the full workspace could have been <http://nyc> in which case the full layer name would be <http://nyc:streets>. GeoServer intelligently replaces the workspace prefix with the full workspace URI, but it can be useful to know the difference.

## Creating a new workspace

In this section we are going to create a new workspace called momo .

1. Navigate to Data > Workspaces .
2. Click Add new workspace and enter the following:
  - Name: momo
  - Namespace URI: <http://localhost:80/momo>
  - Default Workspace: checked
3. Click Submit

**New Workspace**

Configure a new workspace

<b>Name</b>	momo
<b>Namespace URI</b>	<a href="http://localhost:80/momo">http://localhost:80/momo</a>
The namespace uri associated with this workspace	
<b>Default Workspace</b>	<input checked="" type="checkbox"/>
<input type="button" value="Submit"/> <input type="button" value="Cancel"/>	

Add new workspace

The workspace has been created and is now active. The green check mark indicates that the workspace is the default.

## Vector layers

In this section we will learn how to import a shapefile to a spatially enabled PostgreSQL database, set up a new vector store and publish a new vector layer with GeoServer.

- [Import shapefile to database](#)
- [Create a new PostgreSQL/PostGIS datastore](#)
- [Publish a new vector layer](#)
- [Preview the layer using GeoServer's layer preview](#)
- [Changing the layer style](#)

# Import

Our workshop data (see [here](#)) is actually given as a collection of single shapefiles. In order to enhance the performance while read and write processes and enable writing at all (shapefiles can't be manipulated via WFS-T after they have been published), we will import the given data into our own PostgreSQL database entity we just created.

## Import country polygons

1. Open a new terminal window and navigate to the materials directory:

```
$ cd ~/materials
```

2. List all directories in the current folder with:

```
$ ls -l
```

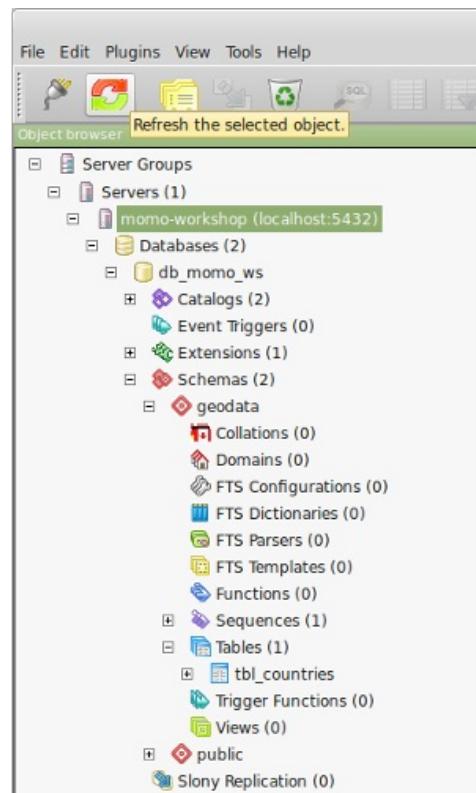
3. You should be able to see the folders `natural_earth` and `osm_mongolia`. If you inspect them further on, you will see that each folder contains a wide list of single shapefiles, but for the moment we want to use a small portion of them to import into the database only. Therefore we are going to start with importing a shapefile involving worldwide country polygons.
4. Navigate to directory `materials/natural_earth/10m_cultural` and check if you can find a file named

```
ne_10m_admin_0_countries.shp .
```

5. To import the shapefile we will use the command line tool `shp2pgsql`. The following command will transform the input shapefile `ne_10m_admin_0_countries.shp` with the input encoding `LATIN1` and input projection `4326` to a SQL statement which will fill the data in a new table `tbl_countries` in schema `geodata`. After execution the output of `shp2pgsql` is directly piped to `psql` which will run the SQL output against the workshop database. You can easily copy the command below into the terminal window and execute it subsequently.

```
$ shp2pgsql \
-s 4326 \
-W LATIN1 \
-I ne_10m_admin_0_countries.shp \
geodata.tbl_countries | \
psql \
-h localhost \
-p 5432 \
-U momo-ws \
-W \
-d db_momo_ws
```

6. After the successful execution of the above command (re-)open `pgAdmin III`, mark the database server `momo-workshop` in tree view and click `Refresh the selected object` to refresh the database/table list and expand the tree to `Databases > db_momo_ws > Schemas > geodata > Tables` (if not already happened) as shown below:



7. Select table `tbl_countries` and click `View the data in the selected object` to open the data view for the imported table as shown below:

pgAdmin III

File Edit Plugins View Tools Help

Object browser

View the data in the selected object.

Properties Statistics Dependencies

Edit Data - momo

No limit

	gid [PK] serial	scalerank smallint	featurecla character v	labelrank double pre	sovereign character v	sov_a3 character v	adm
1	1	3	Admin-0 cou 5		Netherlands	NL1	1
2	2	0	Admin-0 cou 3		Afghanistan	AFG	0
3	3	0	Admin-0 cou 3		Angola	AGO	0
4	4	3	Admin-0 cou 6		United King	GB1	1
5	5	0	Admin-0 cou 6		Albania	ALB	0
6	6	3	Admin-0 cou 6		Finland	FIL	1
7	7	0	Admin-0 cou 6		Andorra	AND	0
8	8	0	Admin-0 cou 4		United Arab	ARE	0
9	9	0	Admin-0 cou 2		Argentina	ARG	0
10	10	0	Admin-0 cou 6		Armenia	ARM	0
11	11	5	Admin-0 cou 4		United Stat	US1	1
12	12	0	Admin-0 cou 4		Antarctica	ATA	0
13	13	3	Admin-0 cou 6		France	FR1	1
14	14	3	Admin-0 cou 6		Antigua and	ATG	0
15	15	0	Admin-0 cou 2		Australia	AU1	1
16	16	0	Admin-0 cou 4		Austria	AUT	0
17	17	0	Admin-0 cou 5		Azerbaijan	AZE	0
18	18	0	Admin-0 cou 6		Burundi	BDI	0
19	19	0	Admin-0 cou 2		Belgium	BEL	0
20	20	0	Admin-0 cou 5		Benin	BEN	0
21	21	0	Admin-0 cou 3		Burkina Fas	BFA	0
22	22	0	Admin-0 cou 3		Bangladesh	BGD	0
23	23	0	Admin-0 cou 4		Bulgaria	BGR	0
24	24	5	Admin-0 cou 4		Bahrain	BHR	0
25	25	3	Admin-0 cou 4		The Bahamas	BHS	0
26	26	0	Admin-0 cou 5		Bosnia and	BIH	0
27	27	6	Admin-0 cou 8		Bajo Nuevo	BN1	0
28	28	6	Admin-0 cou 6		France	FR1	1
29	29	0	Admin-0 cou 4		Belarus	BLR	0
30	30	0	Admin-0 cou 6		Belize	BLZ	0
31	31	5	Admin-0 cou 6		United King	GB1	1
32	32	0	Admin-0 cou 3		Bolivia	BOL	0
33	33	0	Admin-0 cou 2		Brazil	BRA	0
34	34	3	Admin-0 cou 5		Barbados	BRB	0
35	35	0	Admin-0 cou 6		Brunei	BRN	0

Scratch pad

Retrieving details on table tbl\_countries... Done.

8. Congratulations! You've successfully imported a shapefile into PostgreSQL that can now easily be published through the GeoServer instance!

# Store

A store is the name for a container of geographic data. A store refers to a specific data source, be it a shapefile, database, or any other data source that GeoServer supports.

A store can contain many layers, such as the case of a database that contains many tables. A store can also have a single layer, such as in the case of a shapefile or GeoTIFF. A store must contain at least one layer.

GeoServer saves the connection parameters to each store (the path to the shapefile, credentials to connect to the database). Each store must also be associated with one (and only one) workspace.

A store is sometimes referred to as a "datastore" in the context of vector data, or "coveragestore" in the context of raster (coverage) data.

## Creating a new store

Now we can add a new store to our new workspace `momo`. This store tells GeoServer how to connect to the data source, in our case the PostgreSQL database.

1. Navigate to `Data` > `Stores`.
2. Click `Add new Store`.
3. Click `PostGIS - PostGIS Database`
4. Set the Workspace to `momo` if it isn't set already.
5. Configure the new store as follows:
  - o `Data Source Name:` `db_momo_ws`
  - o `Enabled:` checked
  - o `dbname:` `postgis`
  - o `host:` `localhost`
  - o `port:` `5432`
  - o `database:` `db_momo_ws`
  - o `schema:` `geodata`
  - o `user:` `momo_ws`
  - o `passwd:` `momo_ws`
6. Click `Save`

## New Vector Data Source

Add a new vector data source

PostGIS  
PostGIS Database

### Basic Store Info

Workspace \*

momo ▾

Data Source Name \*

db\_momo\_ws

Description

[empty text area]

Enabled

### Connection Parameters

dtype \*

postgis

host \*

localhost

port \*

5432

database

db\_momo\_ws

schema

geodata

user \*

momo\_ws

passwd

[redacted]

Namespace \*

<http://localhost:80/momo>

*Add new store*

# Layer

A layer (sometimes known as a featuretype) is a collection of geospatial features or a coverage. Typically a layer contains one type of data (points, lines, polygons, raster) and has a single identifiable subject (streets, houses, country boundaries, etc.). A layer corresponds to a table or view from a database, or an individual file.

GeoServer stores information associated with a layer, such as projection information, bounding box, and associated styles. Each layer must be associated with one (and only one) workspace.

## Publishing a layer

Once the new store is created, GeoServer automatically gives us the option of publishing layers from this store. Here we chose the table `tbl_countries` by clicking `Publish`.

### New Layer

Add a new layer

You can create a new feature type by manually configuring the attribute names and types. [Create new feature type...](#)  
On databases you can also create a new feature type by configuring a native SQL statement. [Configure new SQL view...](#)  
Here is a list of resources contained in the store 'db\_momo\_ws'. Click on the layer you wish to configure

Published	Layer name	Action
	tbl_countries	<a href="#">Publish</a>

*Publish a layer*

## Minimal layer configuration

After publishing, GeoServer automatically gives us the option of configuring the newly created layer. For the moment, we want to set up the layer with some basic configuration only. Thus we ignore custom styling or caching that will be handled later on.

1. Configure the new layer as follows:
  - *Name:* countries
  - *Enabled:* checked
  - *Advertised:* checked
  - *Title:* Countries
  - *Abstract:* Countries of the world.
  - *Native SRS:* EPSG:4326
  - *Declared SRS:* EPSG:4326
  - *SRS handling:* Keep native
2. Let GeoServer calculate the bounds of the data by clicking `compute from data`.
3. Convert the native bounds to the Lat/Lon Bounding Box by clicking `compute from native bounds`.
4. Click `save`.

**Basic Resource Info**

**Name**  
  
 Enabled  
 Advertised

**Title**

**Abstract**  
  
 ↴

**Keywords**

**Current Keywords**

**New Keyword**  
   
**Vocabulary**

**Metadata links**  
 No metadata links so far

Note only FGDC and TC211 metadata links show up in WMS 1.1.1 capabilities

**Data links**  
 No data links so far

**Coordinate Reference Systems**

**Native SRS**

**Declared SRS**

**SRS handling**

**Bounding Boxes**

**Native Bounding Box**

Min X	Min Y	Max X	Max Y
-181,80000305175	-90,868171691894	181,80001831054	84,502273559570

[Compute from data](#)

**Lat/Lon Bounding Box**

Min X	Min Y	Max X	Max Y
-181,80000305175	-90,868171691894	181,80001831054	84,502273559570

[Compute from native bounds](#)

*Publish a new layer*

# Previewing a layer

You just published the layer with GeoServer! Now let's see how it looks by using the [Layer Preview](#).

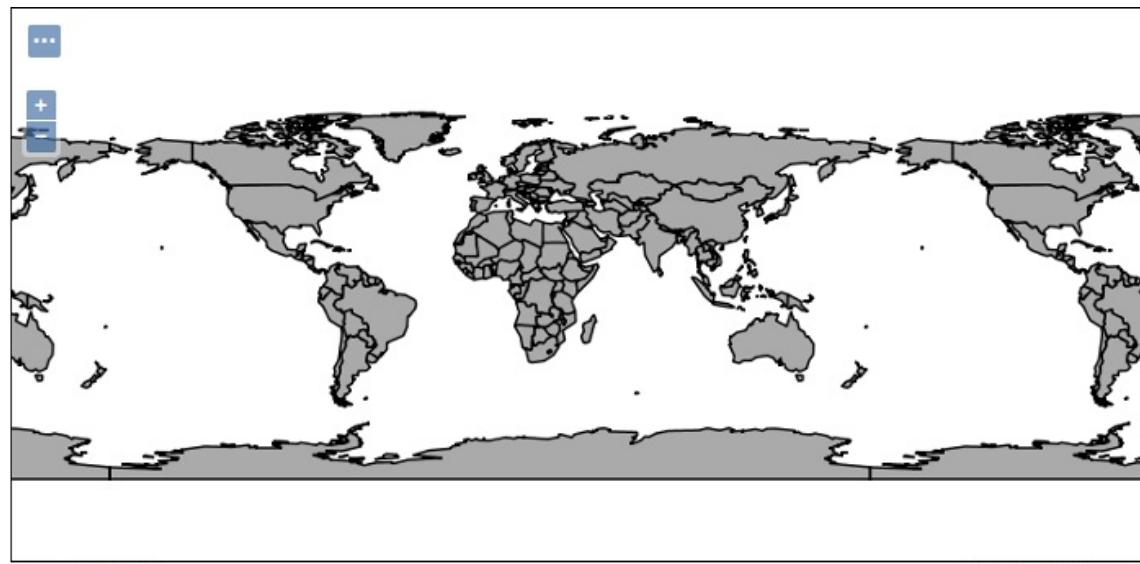
1. Navigate to [Data](#) > [Layer Preview](#)
2. Search for `countries`.
3. Click [OpenLayers](#).

## Layer Preview

List of all layers configured in GeoServer and provides previews in various formats for each.

Results 1 to 1 (out of 1 matches from 23 items)				
Type	Name	Title	Common Formats	All Formats
	momo:countries	Countries	OpenLayers KML GML	Select one ▾

*Layer Preview*



**countries**

fid	scalerank	featurecla	labelrank	sovereign	sov_a3	adm0_dif	level	type	admin	adm0_a3	geou_dif	geounit	gu_a
countries.153	0	Admin-0 country	3.0	Mongolia	MNG	0.0	2.0	Sovereign country	Mongolia	MNG	0.0	Mongolia	MNG

*Hello world!*

As you published the layer `countries` GeoServer not only serves this layer as WMS, in addition it automatically publishes the feature type via its WFS server.

1. Return to the [Layer Preview](#) site and search for `countries` (see steps 1. and 2. ahead).
2. Select a WFS format (e.g. the common format `GeoJSON`) in the dropdown menu [All formats](#).

## Layer Preview

List of all layers configured in GeoServer and provides previews in various formats for each.

<< < > >> Results 1 to 1 (out of 1 matches from 23 items)					countries
Type	Name	Title	Common Formats	All Formats	
	momo:countries	Countries	OpenLayers KML GML	Select one GeoTiff 8-bits JPEG KML (compressed) KML (network link) KML (plain) OpenLayers PDF PNG PNG 8bit SVG Tiff Tiff 8-bits <b>WFS</b> CSV GML2 GML3.1 GML3.2 <b>GeoJSON</b> KML Shapfile	

*WFS Preview*

After selecting the entry you should see a new browser tab or window containing the GeoJSON representation of the layer `countries` similar to following excerpt:

```
{
  "type": "FeatureCollection",
  "totalFeatures": 254,
  "features": [
    {
      "type": "Feature",
      "id": "countries.1",
      "geometry": {
        "type": "MultiPolygon",
        "coordinates": [
          [
            [
              [
                [
                  [
                    [-69.99693762899994, 12.577582098000022],
                    [-69.93639075399997, 12.531724351000037],
                    [-69.92467200399997, 12.519232489000018],
                    (...),
                    ...
                  ]
                ]
              ]
            ],
            ...
          ]
        ],
        "geometry_name": "geom",
        "properties": {
          "scalerank": 3,
          "featurecla": "Admin-0 country",
          "labelrank": 5,
          "sovereign": "Netherlands",
          ...
        }
      },
      ...
    }
  ];
}
```



## Style

A style is a visualization directive for rendering geographic data. A style can contain rules for color, shape, and size, along with logic for styling certain features or points in certain ways based on attributes or scale level.

Every layer must be associated with at least one style. GeoServer recognizes styles in Styled Layer Descriptor (SLD) format. The Styling section will go into this topic in greater detail.

## Create and assign a style to a layer

1. Go to `Data` > `Styles` > `Add a new style`
2. Create a new style as follows:
  - o *Name*: countries
  - o *Workspace*: momo
  - o *Format*: SLD
  - o Copy and paste the following SLD content into the style field:

```

<?xml version="1.0" encoding="UTF-8"?>
<sld:StyledLayerDescriptor
  xmlns="http://www.opengis.net/sld"
  xmlns:sld="http://www.opengis.net/sld"
  xmlns:ogc="http://www.opengis.net/ogc"
  xmlns:gml="http://www.opengis.net/gml"
  version="1.0.0">
  <sld:NamedLayer>
    <sld:Name>countries</sld:Name>
    <sld:UserStyle>
      <sld:Name>Countries</sld:Name>
      <sld>Title>Countries</sld>Title>
      <sld:FeatureTypeStyle>
        <sld:Name>countries</sld:Name>
        <sld:Rule>
          <sld:PolygonSymbolizer>
            <sld:Fill>
              <sld:CssParameter name="fill">#EDEDED</sld:CssParameter>
            </sld:Fill>
            <sld:Stroke>
              <sld:CssParameter name="stroke">#969696</sld:CssParameter>
              <sld:CssParameter name="stroke-width">0.5</sld:CssParameter>
            </sld:Stroke>
          </sld:PolygonSymbolizer>
          <sld:TextSymbolizer>
            <sld:Label>
              <ogc:PropertyName>name</ogc:PropertyName>
            </sld:Label>
            <sld:Font>
              <CssParameter name="font-family">DejaVu Sans</CssParameter>
              <CssParameter name="font-size">10</CssParameter>
            </sld:Font>
            <sld:LabelPlacement>
              <sld:PointPlacement>
                <sld:AnchorPoint>
                  <sld:AnchorPointX>0.5</sld:AnchorPointX>
                  <sld:AnchorPointY>0.5</sld:AnchorPointY>
                </sld:AnchorPoint>
              </sld:PointPlacement>
            </sld:LabelPlacement>
            <sld:Halos>
              <sld:Radius>1</sld:Radius>
              <sld:Fill>
                <CssParameter name="fill">#FFFFFF</CssParameter>
              </sld:Fill>
            </sld:Halos>
            <sld:Fill>
              <CssParameter name="fill">#707070</CssParameter>
            </sld:Fill>
          </sld:TextSymbolizer>
        </sld:Rule>
      </sld:FeatureTypeStyle>
    </sld:UserStyle>
  </sld:NamedLayer>
</sld:StyledLayerDescriptor>

```

3. Go to `Data` → `Layers`, search for `countries` and select it in the list.

## Layers

Manage the layers being published by GeoServer

- [Add a new resource](#)
- [Remove selected resources](#)

Results 1 to 1 (out of 1 matches from 20 items)					
Type	Workspace	Store	Layer Name	Enabled?	Native SRS
	momo	db_momo_ws	<a href="#">countries</a>		EPSG:4326

Select a layer.

1. Go to tab `Publishing`.

## Edit Layer

Edit layer data and publishing

### momo:countries

Configure the resource and publishing information for the current layer

Data	Publishing	Dimensions	Tile Caching

*Publishing tab*

1. Select `momo:countries` in dropdown list `Default Style`.

#### WMS Settings

Queryable

Opaque

#### Default Style

`momo:countries` ▾



*Select the default style*

1. Click `Save`.

2. Open the layer preview for the layer `countries` and you will see that the layer will have a new appearance (light grey polygon fill) including labels for each country.

**countries**

<b>fid</b>	<b>scalerank</b>	<b>featurecla</b>	<b>labelrank</b>	<b>sovereignt</b>	<b>sov_a3</b>	<b>adm0_dif</b>	<b>level</b>	<b>type</b>	<b>admin</b>	<b>adm0_a3</b>	<b>geou_dif</b>	<b>geounit</b>	<b>gu_a</b>
countries.153	0	Admin-0 country	3.0	Mongolia	MNG	0.0	2.0	Sovereign country	Mongolia	MNG	0.0	Mongolia	MNG

*Layer preview centered to mongolia.*

## Raster layers

In this section we will learn how to prepare a large GeoTIFF file with GDAL, set up a new raster store and publish a new raster layer with GeoServer.

- [Prepare large GeoTIFF with GDAL](#)
- [Create a new ImagePyramid datastore](#)
- [Publish a new raster layer](#)
- [Preview the layer using GeoServer's layer preview](#)

# Preparing the data

An image pyramid builds multiple mosaics of images, each one at a different zoom level, making it so that each tile is stored in a separate file. This comes with a composition overhead to bring back the tiles into a single image, but can speed up image handling as each overview is tiled, and thus a sub-set of it can be accessed efficiently (as opposed to a single GeoTIFF, where the base level can be tiled, but the overviews never are).

Our input raster from natural earth is a simple huge GeoTIFF file (~400MB) without overviews. Not exactly what we'd want to use for high performance data serving, but good for redistribution and as a starting point to build a pyramid.

In order to build the pyramid we'll use the `gdal_retile.py` utility, part of the GDAL command line utilities and available for various operating systems.

1. Open terminal and navigate to directory `~/materials/natural_earth/OB_LR`.
2. Create a new folder named `pyramid` with:

```
$ mkdir OB_LR_pyramid/
```

1. Run the following command that will build a pyramid (Note: This may take a while!):

```
$ gdal_retile.py -v \
-s_srs EPSG:4326 \
-r bilinear \
-levels 4 \
-ps 512 512 \
-co "TILED=YES" \
-co "COMPRESS=JPEG" \
-targetDir OB_LR_pyramid/ \
OB_LR.tif
```

Short explanation:

- **-v**: Verbose output, allows the user to see each file creation scroll by, thus knowing progress is being made.
- **-r bilinear**: Use bilinear interpolation when building the lower resolution levels. This is key to get good image quality without asking GeoServer to perform expensive interpolations in memory.
- **-levels 4**: The number of levels in the pyramid.
- **-ps 512 512**: Each tile in the pyramid will be a 512x512 GeoTIFF.
- **-co "TILED=YES"**: Each GeoTIFF tile in the pyramid will be inner tiled.
- **-co "COMPRESS=JPEG"**: Each GeoTIFF tile in the pyramid will be JPEG compressed (trades small size for higher performance, try out it without this parameter too).
- **-targetDir pyramid**: Build the pyramid in the pyramid directory. The target directory must exist and be empty
- **OB\_LR.tif**: The source file
- As GeoServer needs to have read and write access to the pyramid we just created, we'll move the `OB_LR_pyramid` folder to the GeoServer `data` directory:

```
$ sudo mv OB_LR_pyramid/ /var/lib/tomcat7/webapps/geoserver/data/data/
```

1. Navigate to the data directory:

```
$ cd /var/lib/tomcat7/webapps/geoserver/data/data/
```

1. Assign read and write access to the `tomcat7` user:

```
$ sudo chown -R tomcat7:tomcat7 OB_LR_pyramid; sudo chmod -R 755 OB_LR_pyramid/
```



# Creating a new store

1. Go to `Data` > `Stores` > `Add a new Store`
2. Select `ImagePyramid`

## Raster Data Sources

- [ArcGrid](#) - ARC/INFO ASCII GRID Coverage Format
- [GeoTIFF](#) - Tagged Image File Format with Geographic information
- [Gtopo30](#) - Gtopo30 Coverage Format
- [ImageMosaic](#) - Image mosaicking plugin
- [ImagePyramid](#) - Image pyramidal plugin
- [WorldImage](#) - A raster file accompanied by a spatial data file

*Add ImagePyramid store*

1. Create the new store as follows:

- *Workspace*: momo
- *Data Source name*: ocean-bottom-relief
- *Enabled*: checked
- *URL*: `file:data/OB_LR_pyramid`

## Add Raster Data Source

Description

ImagePyramid  
Image pyramidal plugin

### Basic Store Info

Workspace \*

Data Source Name \*

Description

Enabled

### Connection Parameters

URL \*

*Configure ImagePyramid store*

1. Click `Save`.

## Publishing a layer

Once the new store is created, GeoServer automatically gives us the option of publishing layers from this store. Here we chose the entry `OB_LR_pyramid` by clicking `Publish`.

### New Layer

Add a new layer

Add layer from `momo:ocean-bottom-relief`

On stores you can also create a new coverage view by merging different coverages as a multibands coverage. [Configure new Coverage view ...](#)  
Here is a list of resources contained in the store 'ocean-bottom-relief'. Click on the layer you wish to configure

<code>&lt;&lt;</code>	<code>&lt;</code>	<code>1</code>	<code>&gt;</code>	<code>&gt;&gt;</code>	Results 0 to 0 (out of 0 items)
<b>Published</b>				<input type="text"/> <b>Layer name</b>	<b>Action</b>
				<code>OB_LR_pyramid</code>	<code>Publish</code>
<code>&lt;&lt;</code>	<code>&lt;</code>	<code>1</code>	<code>&gt;</code>	<code>&gt;&gt;</code>	Results 0 to 0 (out of 0 items)

[Configure ImagePyramid store](#)

## Minimal layer configuration

After publishing, GeoServer automatically gives us the option of configuring the newly created layer. For the moment, we want to set up the layer with some basic configuration only. Thus we ignore custom styling or caching that will be handled later on.

1. Configure the new layer as follows:
  - **Name:** ocean-bottom-relief
  - **Enabled:** checked
  - **Advertised:** checked
  - **Title:** Ocean bottom relief
  - **Abstract:** Blended depth colors and relief shading of the ocean bottom derived from CleanTOPO2 data.
  - **Native SRS:** EPSG:4326
  - **Declared SRS:** EPSG:4326
  - **SRS handling:** Keep native
2. Let GeoServer calculate the bounds of the data by clicking `compute from data`.
3. Convert the native bounds to the Lat/Lon Bounding Box by clicking `compute from native bounds`.

**Basic Resource Info**

**Name**  
ocean-bottom-relief

Enabled

Advertised

**Title**  
Ocean bottom relief

**Abstract**  
Blended depth colors and relief shading of the ocean bottom derived from CleanTOPO2 data.

---

**Keywords**

**Current Keywords**  
WCS  
ImagePyramid  
OB\_LR\_pyramid

**New Keyword**

**Vocabulary**

---

**Metadata links**  
No metadata links so far

Note only FGDC and TC211 metadata links show up in WMS 1.1.1 capabilities

---

**Data links**  
No data links so far

---

**Coordinate Reference Systems**

**Native SRS**  
EPSG:4326

**Declared SRS**  
EPSG:4326

**SRS handling**

---

**Bounding Boxes**

**Native Bounding Box**

Min X	Min Y	Max X	Max Y
-180	-89.99999999982	179.9999999996	90

**Lat/Lon Bounding Box**

Min X	Min Y	Max X	Max Y
-180	-89.99999999982	179.9999999996	90

*Publish a new layer.*

1. Click **Save**.

## Previewing a layer

You just published the raster layer with GeoServer! Now let's see how it looks by using the `Layer Preview`.

1. Navigate to `Data` > `Layer Preview`
2. Search for `ocean`.
3. Click `openLayers`.



*The bathymetry layer.*

## Group layers

In this section we will learn how to set up a new grouped layer with GeoServer.

- [Publish a new grouped layer](#)
- [Preview the layer using GeoServer's layer preview](#)

## Layer group

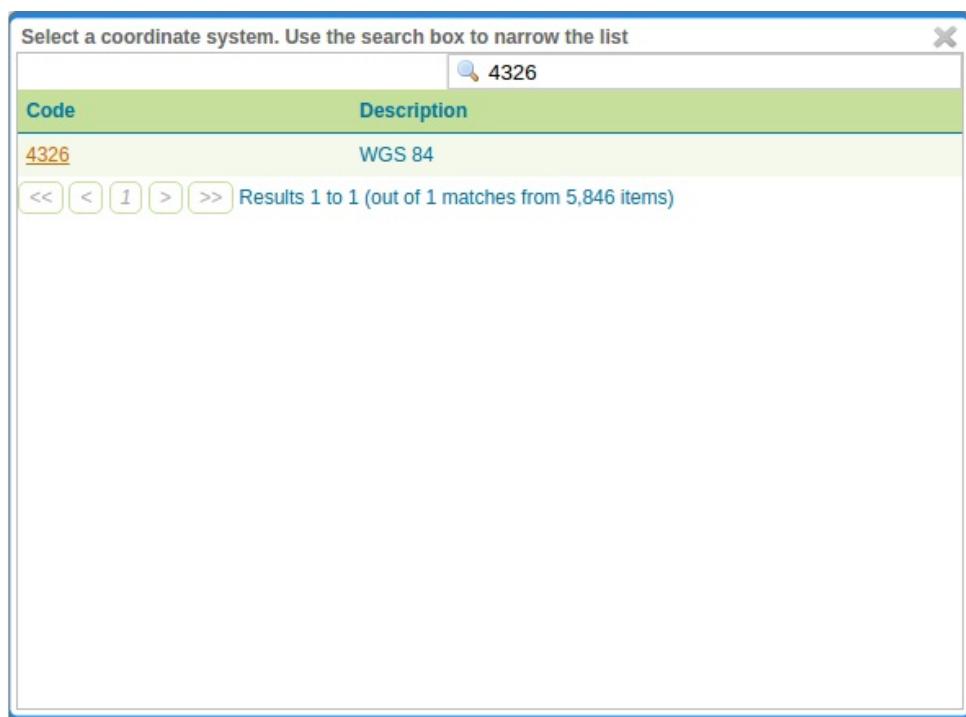
A layer group, as its name suggests, is a collection of layers. A layer group makes it possible to request multiple layers with a single WMS request. A layer group contains information about the layers that comprise the layer group, the order in which they are rendered, the projection, associated styles, and more. This information can be different from the defaults for each individual layer.

Layer groups do not respect the concept of workspace, and are relevant only to WMS requests.

## Create grouped layer

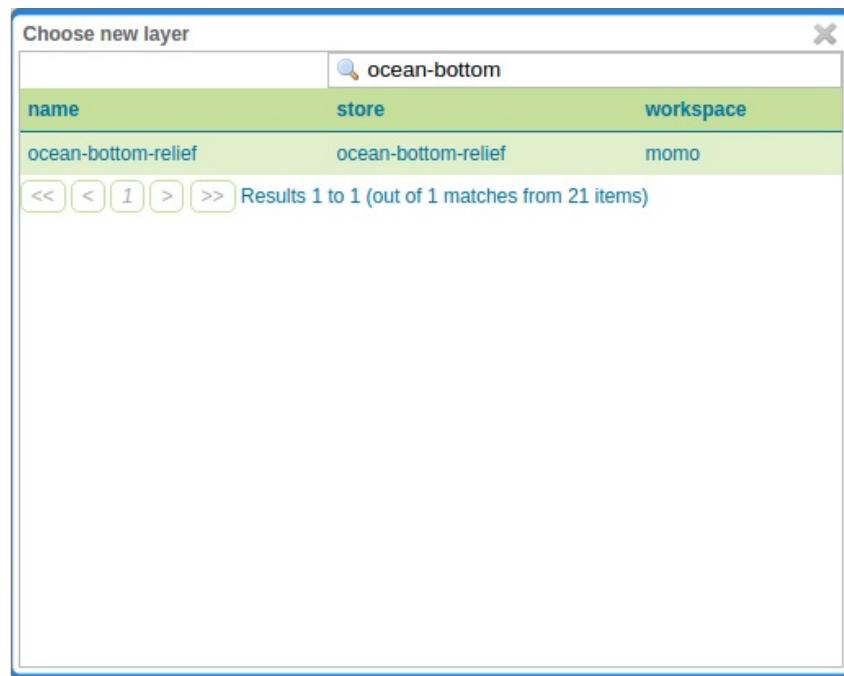
Use countries and ocean bottom relief

1. Go to `Data`  $\rightarrow$  `Layer Groups`  $\rightarrow$  `Add new layer group`.
2. Create a new layer group as follows:
  - $\circ$  `Name:` world-layer
  - $\circ$  `Title:` World layer
  - $\circ$  `Workspace:` momo
3. Find and select `EPSG:4326` under `Coordinate Reference System`.



Select the SRS

1. Click `Layers`  $\rightarrow$  `Add Layer`, search for `ocean-bottom` and click the `ocean-bottom-relief` in the list to add the layer to the group.



*Add layer to layergroup.*

1. Repeat the step above for layer `countries` .
2. Let GeoServer generate the bounds for this layer by pressing `Generate bounds` .

## Layer group

Edit the contents of a layer groups

**Name**  
world-layer

**Title**  
World layer

**Abstract**

**Workspace**  
momo ▾

**Bounds**

Min X	Min Y	Max X	Max Y
-181,80000305175	-90,868171691894	181,800018310541	90

**Coordinate Reference System**  
EPSG:4326 Find... EPSG:WGS 84...

**Generate Bounds**

**Mode**  
Single ▾

**Layers**

Add Layer... Add Layer Group...

Drawing order	Layer	Default Style	Style	Remove
1 ↓	momo:ocean-bottom-relief	<input type="checkbox"/>	raster	
2 ↑	momo:countries	<input type="checkbox"/>	countries	

<< < 1 > >> Results 1 to 2 (out of 2 items)

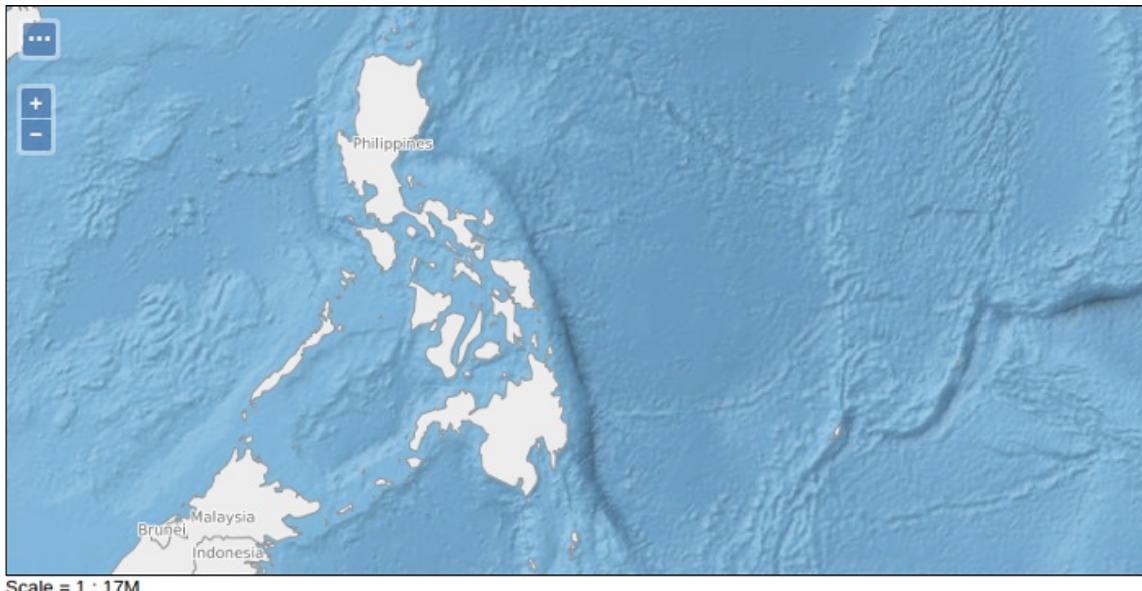
Configure a new group layer.

1. Click **Save**.

## Previewing a layer

You just published the group layer with GeoServer! Now let's see how it looks by using the `Layer Preview`.

1. Navigate to `Data`  $\rightarrow$  `Layer Preview`
2. Search for `world-layer`.
3. Click `OpenLayers`.



*The group layer centered to the Philippines*

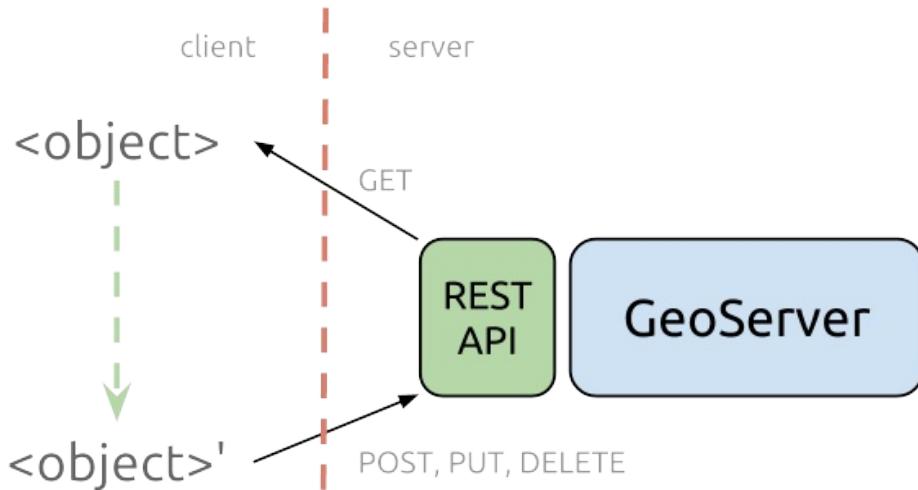
## GeoServer Advanced

In this section we'll learn two aspects of advanced usage of GeoServer, where we'll focus on the configuration via the REST interface and the caching of layers with the built-in caching engine GeoWebCache.

- [REST API](#)
- [Caching with GeoWebCache](#)

## Excursion: REST interface

This chapter will give you a short introduction to GeoServers REST (Representational State Transfer) interface. The REST API allows you to read, write, update and remove (almost) all GeoServer catalog elements directly via the HTTP protocol. These can include, for example, the manipulation of workspaces, data storages, layer styles and layers itself. A benefit of using the REST interface is that you can script recurring steps of work, for instance publishing a large number of layers at once from a remote machine.



*Principle of operation, source:*

[https://github.com/boundlessgeo/workshops/blob/master/workshops/geoserver/adv/doc/source/catalog/img/rest\\_theory.png](https://github.com/boundlessgeo/workshops/blob/master/workshops/geoserver/adv/doc/source/catalog/img/rest_theory.png)

## What is REST?

REST (sometimes as ReST) is an acronym for Representational State Transfer and is an architectural style for the realization of web services and is therefore frequently mentioned in connection with **RESTful Webservices**. The idea behind is that one should be able to use simple and lightweight HTTP calls to connect between (web-)clients and remote servers. So the capabilities of the REST API consists of the actions (verbs) we can use to make HTTP requests combined with the configurable resources in GeoServer. For each of the resources in GeoServer (workspaces, stores, layers, styles, layer groups, etc.) we can perform the following operations ([source](#)):

Operation	Description
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.
POST	The POST method requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI.
PUT	The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.
PATCH	The PATCH method applies partial modifications to a resource.
DELETE	The DELETE method deletes the specified resource.
HEAD	The HEAD method asks for a response identical to that of a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.
OPTIONS	The OPTIONS method returns the HTTP methods that the server supports for the specified URL.
CONNECT	The CONNECT method converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.
TRACE	The TRACE method echoes the received request so that a client can see what (if any) changes or additions have been made by intermediate servers.

To sum it up, in the GeoServer REST API we are able to use the methods as follows:

- GET to read an existing resource
- POST to add a new resource
- PUT to update an existing resource
- DELETE to remove a resource

In relation to the methods mentioned above each request will respond with a certain response code:

Status code	Status text	Description
200	OK	Request was successful
201	Created	A resource (e.g. a layer) was successfully created
403	Forbidden	Not authorized
404	Not Found	Resource or endpoint not found
405	Method Not Allowed	Wrong operation for resource or endpoint (e.g. GET-request, but only PUT/POST allowed)
500	Internal Server Error	Error while execution (e.g. syntax error in request)

# Reading the catalog

In this module we will learn how to read out the GeoServer configuration via the REST API.

As already mentioned in the previous chapter, a key condition of REST is the addressability. Thereby each catalog configuration (= resource or endpoint) in GeoServer has an unique URL.

At first we will investigate the REST API via the browser. At the same time we are using the HTTP operation `GET` to *retrieve* information from the server.

1. Open up a browser window and navigate to the following URL (Note: You will be prompted for your GeoServer user and password):

```
http://localhost/geoserver/rest
```

## Geoserver Configuration API

- [imports](#)
- [workspaces](#)
- [namespaces](#)
- [styles](#)
- [layers](#)
- [layergroups](#)
- [reload](#)
- [reset](#)
- [about/manifest](#)
- [about/version](#)
- [settings](#)
- [settings/contact](#)
- [services/wms/settings](#)
- [services/wfs/settings](#)
- [services/wcs/settings](#)
- [templates](#)

You will see a simple HTML list which contains the top endpoints provided by the REST API. The list view is fully controllable and clearly assigned. A selection in the browser (for example the entry **workspaces**) navigates the browser to unique URL `{{book.geoServerBaseUrl}}/rest/workspaces`. The structure of the list (when selecting a workspace) follows the logical structure of the GeoServer catalog we already met in the previous sections:

```
workspace
  |
  +--datastore
    |
    +--featuretype
```

The above actions in the browser will call an endpoint in HTML format by default. The GeoServer also supports the formats `JSON` (JavaScript Object Notation) and `XML` (Extensible Markup Language), which are particularly relevant in the manipulation of a resource we will use later on.

1. Switch to a new tab in your browser. Then open and compare the following outputs:

```
http://localhost/geoserver/rest/workspaces
```

```
http://localhost/geoserver/rest/workspaces.json
```

```
http://localhost/geoserver/rest/workspaces.xml
```

2. In the next step we want to get a full description of the feature type `countries` we created in the previous module in format `JSON`. Copy the following request in your browser and explore the output:

```
http://localhost/geoserver/rest/workspaces/momo/datastores/db_momo_ws/featuretypes/countries.json
```

## Creating a new resource

In this exercise we are going to use the REST interface in combination with the HTTP operations `POST` and `PUT` to *create* a resource on the server. In contrast to the previous module here we are going to use the command line tool *cURL* to access the catalog. *cURL* is a command line tool to transfer data from or to a server using one of the supported protocols (e.g. HTTP or FTP). For more about the tool have a look at [here](#).

## Creating a new workspace

In this module we are going to repeat the steps we have done in chapter [Publishing a vector layer](#). But as we don't want to override our progress (or any individual changes) made to the workspace `momo`, we will create new workspace `momo-rest` for the ensuing exercises.

1. Open up the terminal (if not already openend) and type in the following command to create a new workspace named `momo-rest` :

```
$ curl \
-v \
-u admin:momo-ws\
-XPOST \
-H "Content-type: text/xml" \
-d "<workspace>
<name>momo-rest</name>
</workspace>" \
http://localhost/geoserver/rest/workspaces
```

The call above differs in two essential points from the previous read operations: Unlike the HTTP operation `GET` we use the operation `POST` and in addition we transfer a `XML` content containing a simple workspace definition to the unique endpoint `workspaces`.

2. Hit `Enter` to execute the above command and you will see an output like this:

```
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
* Server auth using Basic with user 'admin'
> POST /geoserver/rest/workspaces HTTP/1.1
> Authorization: Basic YWRtaW46Z2Vvc2VydmVy
> User-Agent: curl/7.35.0
> Host: localhost:80
> Accept: */*
> Content-type: text/xml
> Content-Length: 57
>
* upload completely sent off: 57 out of 57 bytes
< HTTP/1.1 201 Created
< Date: Wed, 03 Feb 2016 10:30:39 GMT
< Location: http://localhost:80/geoserver/rest/workspaces/momo-rest
* Server Noelios-Restlet-Engine/1.0..8 is not blacklisted
< Server: Noelios-Restlet-Engine/1.0..8
< Transfer-Encoding: chunked
<
* Connection #0 to host localhost left intact
```

Here, two informations are crucial to us:

- `HTTP/1.1 201 Created` : The request has been successfully processed and the resource has been created.
- `http://localhost/geoserver/rest/workspaces/momo-rest` : The REST endpoint URL of our new workspace.

3. We can verify that the workspace was actually created either by using the GeoServer UI or the REST interface:

- Open the GeoServer [user interface](#), navigate to the page `Data` → `Workspaces` and ensure a new workspace named `momo-rest` is available in the list.
- Open the terminal and run the following command to get a `XML` representation of all available workspaces:

```
$ curl \
-v \
-u admin:momo-ws \
-XGET \
-H "Accept: text/xml" \
http://localhost/geoserver/rest/workspaces
```

## Creating a new store

Now that we have created a new workspace, we'll add a new data store to it. Here we are reusing the database we already [created](#) (and added to the geoserver).

1. Open the terminal and insert the following command to create a new PostGIS datastore named `db_momo_ws_rest` :

```
$ curl \
-v \
-u admin:momo-ws \
-XPOST \
-H "Content-type: text/xml" \
-d "<dataStore>
<name>db_momo_ws_rest</name>
<connectionParameters>
<host>localhost</host>
<port>5432</port>
<database>db_momo_ws</database>
<schema>geodata</schema>
<user>momo-ws</user>
<passwd>momo-ws</passwd>
<dbtype>postgis</dbtype>
</connectionParameters>
</dataStore>" \
http://localhost/geoserver/rest/workspaces/momo-rest/datastores
```

2. Hit `Enter` to execute the command. This will result in the following output, assuring that the store was successfully created:

```
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
* Server auth using Basic with user 'admin'
> POST /geoserver/rest/workspaces/momo-rest/datastores HTTP/1.1
> Authorization: Basic YWRtaW46Z2Vvc2VydmVy
> User-Agent: curl/7.35.0
> Host: localhost:80
> Accept: */*
> Content-type: text/xml
> Content-Length: 347
>
* upload completely sent off: 347 out of 347 bytes
< HTTP/1.1 201 Created
< Date: Wed, 03 Feb 2016 10:59:04 GMT
< Location: http://localhost:80/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest
* Server Noelios-Restlet-Engine/1.0..8 is not blacklisted
< Server: Noelios-Restlet-Engine/1.0..8
< Transfer-Encoding: chunked
<
* Connection #0 to host localhost left intact
```

3. Once again we can verify the successful creation in the GeoServer UI (`Data > Stores`).

## Publishing a layer

In the next step we're going to publish the table `tbl_countries` as a new layer.

1. Open the terminal and insert the following command to create a new feature type (and layer) named `countries_rest` :

```
$ curl \
-v \
-u admin:momo-ws \
-XPOST \
-H "Content-type: text/xml" \
-d "<featureType>
<name>countries_rest</name>
<nativeName>tbl_countries</nativeName>
<title>Countries</title>
<nativeCRS>EPSG:4326</nativeCRS>
<enabled>true</enabled>
</featureType>" \
http://localhost/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest/featuretypes
```

2. And again, verify that the response contains the lines

```
HTTP/1.1 201 Created
```

and

```
Location: http://localhost/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest/featuretypes/countries_re
```

3. Additionally we can also have a look at the [preview page](#) to ensure the layer is correctly published.

## Create and upload style

We can use the REST API both to create a new style object in GeoServer and to insert an existing SLD-file into it. At first we need to create a new SLD file on our local machine we'll need in the next step. For this purpose we can use the style already used in the previous [module](#).

1. Open the terminal and navigate to your home directory with:

```
$ cd ~
```

2. Create and open a new SLD file `countries-style.sld` in this directory with:

```
$ nano countries-style.sld
```

3. Copy the linked SLD content ([see here](#)) the newly created file and save it with `Strg + o`. You can now close the nano editor with `Strg + X`.

We will now create the style and upload the SLD file we just created.

1. Copy the following block into your terminal and execute it to create a new style object:

```
$ curl \
-v \
-u admin:momo-ws \
-XPOST \
-H "Content-type: text/xml" \
-d "<style>
<name>countries_rest</name>
<filename>countries-style.sld</filename>
</style>" \
http://localhost/geoserver/rest/workspaces/momo-rest/styles
```

2. And again, verify that the response contains the lines

```
HTTP/1.1 201 Created
```

and

```
Location: http://localhost:80/geoserver/rest/workspaces/momo-rest/styles/countries_rest
```

3. Afterwards we can upload the style created above with (Note: Ensure the path to file `countries-style.sld` is correct!):

```
$ curl \
-v \
-u admin:momo-ws \
-XPUT \
-H "Content-type: application/vnd.ogc.sld+xml" \
-d @countries-style.sld \
http://localhost/geoserverrest/workspaces/momo-rest/styles/countries_rest
```

4. This command should complete with:

```
HTTP/1.1 200 OK
```

## Assign a layer style

After we have created the style, we can assign this style to the layer `countries_rest`.

1. Copy and execute the following command in the terminal window:

```
$ curl \
-v \
-u admin:momo-ws \
-XPUT \
-H "Content-type: text/xml" \
-d "<layer>
<defaultStyle>
<name>countries_rest</name>
<workspace>momo-rest</workspace>
</defaultStyle>
</layer>" \
http://localhost/geoserver/rest/layers/momo-rest:countries_rest
```

2. After finished with `HTTP/1.1 200 OK` we can open the [preview page](#) to review the changes made to the layer style.



Scale = 1 : 35M

**countries**

fid	scalerank	featurecla	labelrank	sovereignt	sov_a3	adm0_dif	level	type	admin	adm0_a3	geou_dif	geounit	gu_a
countries.153	0	Admin-0 country	3.0	Mongolia	MNG	0.0	2.0	Sovereign country	Mongolia	MNG	0.0	Mongolia	MNG

*Layer created and styled via the REST API.*

# Updating a layer

Basically we can change every element of catalog by the use of the REST API. In the following example we will change the `countries_rest` layer's default output projection to `EPSG:54009` ([Mollweide projection](#)).

1. Execute the following terminal command to update the layer `countries_rest`. (Note: Every update needs the property `&lt;enabled&ampgttrue&lt;/enabled&ampgt`; otherwise the catalog entry, in this case the layer, will be disabled and not be visible to any user!)

```
$ curl \
-v \
-u admin:momo-ws \
-XPUT \
-H "Content-type: text/xml" \
-d "<featureType>
<enabled>true</enabled>
<srs>EPSG:54009</srs>
<projectionPolicy>REPROJECT_TO_DECLARED</projectionPolicy>
</featureType>" \
http://localhost/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest/featuretypes/countries_rest
```

2. After this step has been confirmed as successfully finished with `HTTP / 1.1 200 OK`, we can then automatically calculate the new native and lat/lon bounding box of the layer by appending the parameter `recalculate=nativebbox,latlonbbox` to the REST URL:

```
$ curl \
-v \
-u admin:momo-ws \
-XPUT \
-H "Content-type: text/xml" \
-d "<featureType>
<enabled>true</enabled>
</featureType>" \
http://localhost/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest/featuretypes/countries_rest?recalc
```

3. Review that the layer has been updated correctly by opening the layer configuration in the GeoServer UI (`Data > Layers`) and have a look at the subsection `Coordinate Reference System` and `Bounding Boxes`, which should contain your requested changes.

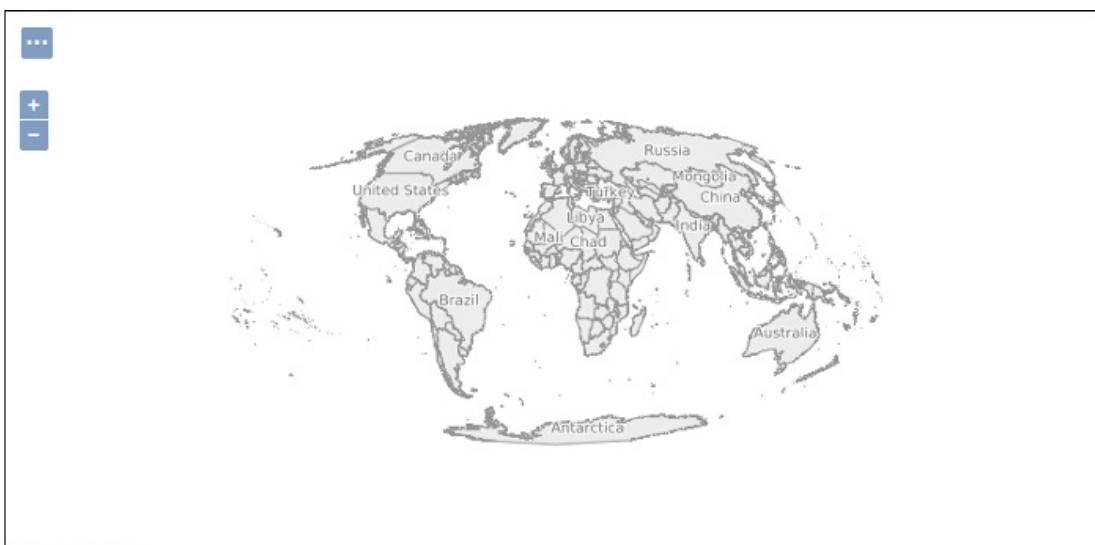
**Coordinate Reference Systems**

<b>Native SRS</b>	EPSG:4326	EPSG:WGS 84...
<b>Declared SRS</b>	EPSG:54009	<a href="#">Find...</a> <a href="#">World_Mollweide...</a>
<b>SRS handling</b>	<a href="#">Reproject native to declared ▾</a>	

**Bounding Boxes**

<b>Native Bounding Box</b>			
Min X	Min Y	Max X	Max Y
-18.203.152,54443	-9.020.047,848073	18.203.154,07225	8.798.621,884479
<a href="#">Compute from data</a>			
<b>Lat/Lon Bounding Box</b>			
Min X	Min Y	Max X	Max Y
-458,36623610465	-90	180	84,5022735595702
<a href="#">Compute from native bounds</a>			

4. Finally have a look at the layer preview page and note, that the default SRS is set to EPSG:54009.



Scale = 1 : 279M  
Click on the map to get feature info

## Remove a resource

You should have noticed that the layer `countries` is available twice now. The first one was created "manually" by the use of the GeoServer UI, the second one via the REST API. Because we don't want to unnecessarily publish a layer twice (and of course to learn how to delete a resource by means of REST), we will delete the `countries_rest` layer by using the HTTP operation `DELETE`.

1. Execute the following command to delete the feature type `countries_rest` and the corresponding layer by appending `recurse=true` to the request:

```
$ curl \
-v \
-u admin:momo-ws \
-XDELETE \
http://localhost/geoserver/rest/workspaces/momo-rest/datastores/db_momo_ws_rest/featuretypes/countries_rest?recurse=true
```

2. After the above command has successfully executed with `HTTP/1.1 200 OK` try to find the layer in the GeoServer Layer configuration page and if anything worked fine, you shouldn't be able to find it

## GeoWebCache

The most common request to GeoServer is to provide an OGC-compliant WMS interface and thus generating maps in raster format. For this reason, caching of these WMS requests may have a decisive influence to the performance of the server and should be carried out on each (productive) system wherever possible. For caching map tiles there is a variety of good open source caching engines available, but here we'll use the GeoServer integrated GeoWebCache (GWC), which acts as a proxy between the client and GeoServer.



*GeoWebCache as proxy, source: [http://geowebcache.org/docs/current/\\_images/how\\_it\\_works.png](http://geowebcache.org/docs/current/_images/how_it_works.png)*

In the following sections we'll initiate all required steps to generate a cache for the layer `momo:countries` :

- [Prerequisites](#)
- [Configure a new gridset](#)
- [Configure a cached layer](#)
- [Generate map tiles.](#)
- [Check cache directory](#)
- [Check cache-headers](#)

## Prerequisites

Before we can start caching a set of layers we need to configure a directory where GWC should save all cached tiles. To accomplish this, please follow these steps:

1. Open the terminal and create the cache directory (Note: You'll be prompted for the admin password):

```
$ sudo mkdir /var/lib/tomcat7/webapps/geoserver/data/gwc
```

2. Ensure GeoServer has read and write access to this directory by changing the ownership to user and group `tomcat7`:

```
$ sudo chown tomcat7:tomcat7 /var/lib/tomcat7/webapps/geoserver/data/gwc
```

3. Open the terminal and copy the following command to open the file `web.xml` in the text editor `gedit`:

```
$ sudo gedit /var/lib/tomcat7/webapps/geoserver/WEB-INF/web.xml
```

4. The following block will advise GWC to store all cached tiles into the directory

`/var/lib/tomcat7/webapps/geoserver/data/gwc`. Insert it at line ~64 in the already opened file.

```
<!-- The GWC data directory-->
<context-param>
  <param-name>GEOWEBCACHE_CACHE_DIR</param-name>
  <param-value>/var/lib/tomcat7/webapps/geoserver/data/gwc</param-value>
</context-param>
```

5. Save the changes and close the text editor.

6. To apply the changes, we need to restart GeoServer. Go to your terminal and run the following command:

```
$ sudo service tomcat7 restart
```

7. Open the newly created directory in the terminal to check there is a file named `geowebcache.xml` only:

```
$ cd /var/lib/tomcat7/webapps/geoserver/data/gwc
```

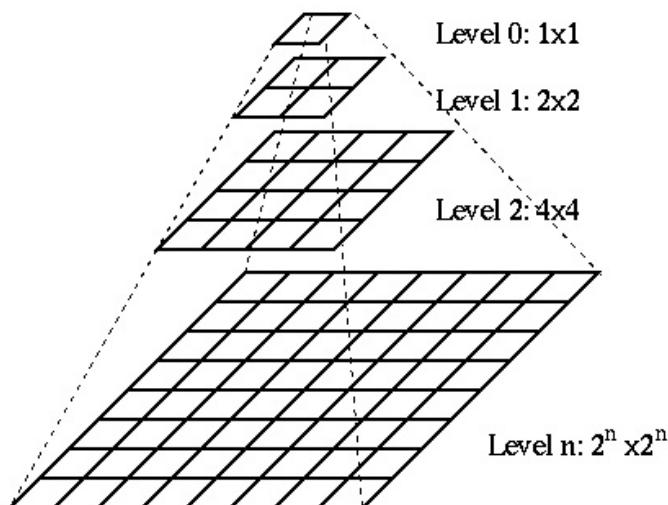
# Tiles and gridsets

## Tiles

GeoWebCache caches images retrieved from a WMS. The smallest unit of image cached is known as a tile. All tiles are assumed to be the same dimensions and are typically square (i.e. 256 pixels by 256 pixels). The tiles are stored in a rectangular grid, indexed by (x,y) coordinates. A z coordinate (zero-indexed) is used to denote the zoom level, resulting in each tile being indexed as a triplet (x,y,z).

## Gridsets

Gridsets and gridsubsets refer to the spatial reference system of the layers served by GeoWebCache. When GeoWebCache makes a request to a WMS, it uses the gridset and gridsubset information to convert its internal tile index to a spatial request that the WMS will understand.



*Composition of a gridset, source: [http://3.bp.blogspot.com/\\_0\\_xIiXP5xuY/S5pEpCjenAI/AAAAAAAACKY/PDKTGZ6vzGI/s1600-h/Image\\_Pyramid.gif](http://3.bp.blogspot.com/_0_xIiXP5xuY/S5pEpCjenAI/AAAAAAAACKY/PDKTGZ6vzGI/s1600-h/Image_Pyramid.gif)*

A gridset is a global definition (i.e. not layer-specific) specifying:

- A spatial reference system.
- A bounding box describing the extent, typically the maximum extent for the above reference system.
- One of either a list of scale denominators, resolutions, or zoom levels.
- The tile dimensions in pixels (constant for all zoom levels).

A gridsubset is a layer-specific definition specifying:

- The gridset for the layer.
- (Optional) The bounding box for that layer (which must be a subset of the extent of the gridSet).
- (Optional) A list of zoom levels (which must be a subset of what is defined in the gridSet).

*Note:* For further instructions have a look at the source of the above explanations, [here](#).

## Configure a new gridset

So, our first step will be to create a new gridset:

1. Go to `Tile Caching`  $\rightarrow$  `Gridsets`

## Gridsets

Manage the available gridsets or create a new one

[Create a new gridset](#)

[Remove selected gridsets](#)

<code>&lt;&lt; &lt;   1   &gt; &gt;&gt;</code> Results 1 to 5 (out of 5 items)						<a href="#">Search</a>
<input type="checkbox"/>	Gridset	CRS	Tile Dimensions	Zoom levels	Disk Usage	
<input type="checkbox"/>	<code>Global/CRS84Scale</code>	EPSG:4326	256 x 256	21	0,0 B	<a href="#">Create a copy</a>
<input type="checkbox"/>	<code>EPSG:4326</code>	EPSG:4326	256 x 256	22	0,0 B	<a href="#">Create a copy</a>
<input type="checkbox"/>	<code>Google/CRS84Quad</code>	EPSG:4326	256 x 256	19	0,0 B	<a href="#">Create a copy</a>
<input type="checkbox"/>	<code>EPSG:900913</code>	EPSG:900913	256 x 256	31	0,0 B	<a href="#">Create a copy</a>
<input type="checkbox"/>	<code>Global/CRS84Pixel</code>	EPSG:4326	256 x 256	18	0,0 B	<a href="#">Create a copy</a>

`<< < | 1 | > >>` Results 1 to 5 (out of 5 items)

2. Click `Create a new gridset` to create a new gridset and use the following options for the creation:

- o `Name:` momo-4326
- o `Coordinate Reference System:` Use the find-button to select `EPSG:4326`
- o `Gridset bounds:` Click `Compute from maximum extent of CRS`
- o `Tile width in pixels:` 512
- o `Tile height in pixels:` 512
- o `Define grids based on:` Select `scale denominators`

3. Click `Add zoom level` to create a new zoom level. Enter the scale `300.000.000` and the name `0`.

4. Once again click `Add zoom level`. You will see that the scale value is automatically cut into halves (`150.000.000`). Just enter the name `1` and repeat this step until you reached a total count of 8 zoom levels. The last scale value should be `2.343.750`.

## Create a new gridset

Define a new gridset for GeoWebCache

Name \*

Description

---

Coordinate Reference System


[EPSG:WGS 84...](#)

Units: °

Meters per unit: 111319.49079327358

Gridset bounds

Min X	Min Y	Max X	Max Y
-180	-90	180	90

[Compute from maximum extent of CRS](#)

Tile width in pixels \*

Tile height in pixels \*

### Tile Matrix Set

Define grids based on:  Resolutions  Scale denominators

Level	Pixel Size	Scale	Name	Tiles
0	0.7545848386603978	1: 300.000.000	0	1 x 1
1	0.3772924193301989	1: 150.000.000	1	2 x 1
2	0.1886462096650995	1: 75.000.000	2	4 x 2
3	0.0943231048325497	1: 37.500.000	3	8 x 4
4	0.0471615524162749	1: 18.750.000	4	15 x 8
5	0.0235807762081374	1: 9.375.000	5	30 x 15
6	0.0117903881040687	1: 4.687.500	6	60 x 30
7	0.0058951940520344	1: 2.343.750	7	120 x 60

[Add zoom level](#)

- Click **Save**.

## Cached layer

In the next step we'll configure the layer `countries` to apply to all needed cache-properties.

1. Go to `Data` > `Layers` and select the `countries` layer.
2. Open the panel `Tile Caching`.

Here we can configure all GWC-dependend properties in a per-layer-basis. The most important configuration parameters are:

- *Create a cached layer for this layer*: Should this layer be cached?
- *Metatiling factors*: Metatile are larger map tiles from which the cached tiles will be cut. The factor in this case indicates the size of the metatiles. A factor of **3x3** means that the screen width of the target tile is increased by a factor of three that results (by a requested tile size of 256px) in an metatile tile size of 768px. Primarily metatiles are needed to prevent duplicate map labels (for example for road layers) in two adjacent tiles.
- *Gutter size in pixels*: Additional frame (in px) to be requested by a tile. Only useful when there are layout problems in the preparation of labels and/or features on the tile edge in conjunction with the use of metatiles.
- *Tile Image Formats*: The standard image format for the tiles.
- *STYLES*: Is there any other style existing for the given layer that should be cached, it must be selected here. In most cases it will be sufficient to set the default layer styles (`LAYER DEFAULT`) only.
- *Gridset*: The gridset defines the grid the stored tiles are indexed and thus defines the spatial index of the individual tiles. The single tile in the rectangular grid is identified by means of a **x**, **y**, **z** coordinate triple. The **x** and **y** coordinates determine the horizontal and vertical position, the **z** coordinate the zoom level. See [previous chapter](#) as well.

## Configure a cached layer

With this in mind, we can configure the layer `countries` as follows:

1. Select the following values:
  - *Create a cached layer for this layer*: checked
  - *Enable tile caching for this layer*: checked
  - *Metatiling factors*: 4 x 4
  - *Gutter size in pixels*: 0
  - *Tile Image Formats*: Check `image/png` only
  - *Expire server cache after n seconds*: 0
  - *Expire client cache after n seconds*: 0
  - *Styles*: Select `LAYER DEFAULT`
  - *Gridset*: Select `momo-4326` in the `Add grid subset` combobox and click the green plus icon. Remove any other preconfigured gridset by clicking the red minus icon.

**Data Publishing Dimensions Tile Caching**

**Tile cache configuration**

Create a cached layer for this layer

Enable tile caching for this layer

Enable In Memory Caching for this Layer.

**BlobStore**

(\*) Default BlobStore ▾

**Metatiling factors**  
4 ▾ tiles wide by 4 ▾ tiles high

**Gutter size in pixels**  
0 ▾

**Tile Image Formats**

- image/gif
- image/jpeg
- image/png
- image/png8

**Expire server cache after n seconds (set to 0 to use source setting)**  
0

**Expire client cache after n seconds (set to 0 to use server setting)**  
0

**Parameter Filters**

**STYLES**

Default Style      Alternate Styles

LAYER DEFAULT ▾  ALL STYLES

Add filter  Choose One ▾

Add Style filter

Available gridsets

Gridset	Published zoom levels	Cached zoom levels	Grid subset bounds
momo-4326	Min ▾ / Max ▾	Min ▾ / Max ▾	Dynamic

Add grid subset: Choose One ▾

**Save** **Cancel**

2. Click **Save**.

# Generate map tiles

Generally speaking, GWC applies two methods for creating cached map tiles:

1. *On-the-fly processing*: If a GWC layer is primarily requested by a client, the appropriate map tiles are rendered and subsequently stored in the GWC data directory. The next client, requesting the same layer on the same location receives a (much faster) response from the cache.
2. *Preprocessing of map tiles*: The tiles of a layer will be preprocessed and stored in a defined bounding box and in defined zoom levels along the given gridset. In contrast to the on-the-fly calculation, this method requires, depending on the available system resources, significantly more computing time, but all clients will receive a direct response from the cache.

With the following steps we'll preprocess the tiles and start the so called `seeding` job.

1. Go to `Tile Caching` > `Tile Layers`.
2. Find the layer `momo:countries` and select `Seed/Truncate`.

## Tile Layers

Manage the cached layers published by the integrated GeoWebCache  
[Add a new cached layer](#)  
[Remove selected cached layers](#)

Type	Layer Name	Disk Quota	Disk Used	BlobStore	Enabled	Preview	Actions
	momo:countries	N/A	N/A			Select One	<a href="#">Seed/Truncate</a>   Empty

3. In the upcoming mask we can configure a GWC-task for seeding the layers `countries`. Here we can use the following configuration:

- *Number of tasks to use*: 04
- *Type of operation*: Reseed - regenerate all tiles (The option `Seed - generate missing tiles` would behave the same here as we haven't any cache present)
- *Grid Set*: momo-4326
- *Format*: image/png
- *Zoom start*: 00
- *Zoom stop*: 07



**GeoWebCache**

List [this Layer tasks ▾](#) (there are no tasks for other Layers)

Kill [all ▾](#) Tasks for Layer 'momo:countries'. [Submit](#)

**List of currently executing tasks:**

- *none*

[Refresh list](#)**Please note:**

- This minimalistic interface does not check for correctness.
- Seeding past zoomlevel 20 is usually not recommended.
- Truncating KML will also truncate all KMZ archives.
- Please check the logs of the container to look for error messages and progress indicators.

Here are the max bounds, if you do not specify bounds these will be used.

- momo-4326: -180.0,-90.0,181.800018310547,84.5022735595703

**Create a new task:**

Number of tasks to use: [04 ▾](#)

Type of operation: [Reseed - regenerate all tiles ▾](#)

Grid Set: [momo-4326 ▾](#)

Format: [Image/png ▾](#)

Zoom start: [00 ▾](#)

Zoom stop: [07 ▾](#)

Modifiable Parameters: [STYLES: momo:countries ▾](#)

Bounding box:      
These are optional, approximate values are fine.

[Submit](#)

4. Click [Submit](#).5. In the same window the section [List of currently executing tasks](#) will be filled with the recent tasks and involves some basic informations about it.[List of currently executing tasks:](#)

<b>Id</b>	<b>Layer</b>	<b>Status</b>	<b>Type</b>	<b>Estimated # of tiles</b>	<b>Tiles completed</b>	<b>Time elapsed</b>	<b>Time remaining</b>	<b>Tasks</b>
1	momo:countries	RUNNING	RESEED	9,738	224	5 seconds	49 seconds	(Task 1 of 4) <a href="#">Kill Task</a>
2	momo:countries	RUNNING	RESEED	9,738	272	5 seconds	40 seconds	(Task 2 of 4) <a href="#">Kill Task</a>
3	momo:countries	RUNNING	RESEED	9,738	240	5 seconds	46 seconds	(Task 3 of 4) <a href="#">Kill Task</a>
4	momo:countries	RUNNING	RESEED	9,738	256	5 seconds	43 seconds	(Task 4 of 4) <a href="#">Kill Task</a>

[Refresh list](#)6. Depending on your system resources the seeding tasks should not cover more than a few minutes. Click [Refresh list](#) to see if the tasks are finished or not.

## Checking the cache directory

Once all tasks are completed, we should verify the content of the cache directory we created above. Given that everything worked fine in the previous step, the cache directory should contain a lot of tiles building up the tile pyramid for the layer `countries`.

1. Open the terminal and navigate to the GWC cache directory for the `countries` layer:

```
$ cd /var/lib/tomcat7/webapps/geoserver/data/gwc/momo_countries
```

2. List the directory contents with:

```
$ ls -l
```

3. Explore that the cache directory is built up by following pattern:

```
momo_countries/ (layername)
|
+-- momo-4326_07/ (gridset name + zoom level)
    |
    +-- 07_03/ (internal notation based on gridset + zoom level)
        |
        +-- 0119_0057.png (tile index)
```

## Checking the cache-headers

Finally we are going to inspect the response send from the GeoServer/GeoWebCache to the client in more detail. As soon as a layer is being cached by GeoWebCache, the response headers of single tile are extended by the following HTTP-headers:

Header	Description
geowebcache-cache-result	If the tile is delivered by the cache, the value is <b>HIT</b> otherwise it's <b>MISS</b> .
geowebcache-crs	The coordinate system of the tile.
geowebcache-gridset	The name of the underlying gridset.
geowebcache-tile-bounds	The bounding box of the tile.
geowebcache-tile-index	The index of the tile (x, y, z) in the gridset.

To check if these headers are set, we need to open the GeoServer user interface again.

1. Go to Tile Caching > Tile Layers .
  2. Find the layer momo:countries and select momo-4326 / png under Preview .

## Tile Layers

Manage the cached layers published by the integrated GeoWebCache

- + Add a new cached layer
  - Remove selected cached layers

<<	<	I	>	>>	Results 1 to 1 (out of 1 matches from 25 items)	countries	
Type	Layer Name	Disk Quota	Disk Used	BlobStore	Enabled	Preview	Actions
	momo:countries	N/A	N/A			Select One ▾ Select One	Seed/Truncate   Empty
<<	<	I	>	>>	Results 1 to 1 (out of 1 matches from 25 items)	momo-4326 / png	

3. In the preview window/tab press F12 to open the browser's **Developer Toolbar**, activate the **Network** tab, select the **Img** subsection and reload the page to record the network activity.



4. Clear the list content with the `clear` button (🚫)
  5. Zoom in to a location of your choice, find a WMS GetMap request in the developer toolbar and select it. Explore the right hand

sided information panel and find the `Response Headers` section. In this you should find the headers looking similar the following ones:

```
geowebcache-cache-result: HIT  
geowebcache-crs: EPSG:4326  
geowebcache-gridset: momo-4326  
geowebcache-tile-bounds: 158.05400771985825, 54.880289022796376, 206.3474373941237, 103.17371869706184  
geowebcache-tile-index: [7, 3, 3]
```

# JavaScript

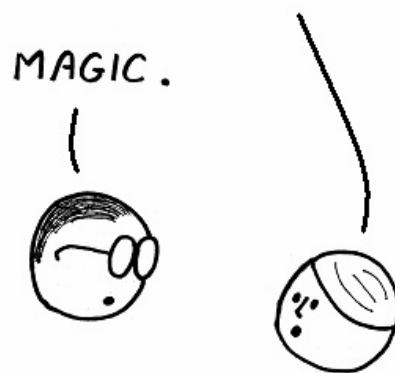
- [Beginners](#)

The beginners section is based on the work done in this [repository](#). Thank you very much!

## Learn Javascript

This book will teach you the basics of programming and Javascript. Whether you are an experienced programmer or not, this book is intended for everyone who wishes to learn the JavaScript programming language.

HOW DOES COMPUTER  
PROGRAMMING WORK ?



Screen

JavaScript (*JS for short*) is the programming language that enables web pages to respond to user interaction beyond the basic level. It was created in 1995, and is today one of the most famous and used programming languages.

# Basics about Programming

In this first chapter, we'll learn the basics of programming and the Javascript language.

Programming means writing code. A book is made up of chapters, paragraphs, sentences, phrases, words and finally punctuation and letters, likewise a program can be broken down into smaller and smaller components. For now, the most important is a statement. A statement is analogous to a sentence in a book. On its own, it has structure and purpose, but without the context of the other statements around it, it isn't that meaningful.

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad [term](#) which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
var hello = "Hello";
var world = "World";

// Message equals "Hello World"
var message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order.

# Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what your code does, thus making it easier for others to understand what your code does.

In Javascript, comments can be written in 2 different ways:

- Line starting with `//` :

```
// This is a comment, it will be ignored by the interpreter
var a = "this is a variable defined in a statement";
```

- Section of code starting with `/*` and ending with `*/`, this method is used for multi-line comments:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
var a = "this is a variable defined in a statement";
```

## Exercise

Mark the editor's contents as a comment

```
Mark me as a comment
or I'll throw an error
```

# Variables

The first step towards really understanding programming is looking back at algebra. If you remember it from school, algebra starts with writing terms such as the following.

$$3 + 5 = 8$$

You start performing calculations when you introduce an unknown, for example x below:

$$3 + x = 8$$

Shifting those around you can determine x:

$$\begin{aligned}x &= 8 - 3 \\-> x &= 5\end{aligned}$$

When you introduce more than one you make your terms more flexible - you are using variables:

$$x + y = 8$$

You can change the values of x and y and the formula can still be true:

$$\begin{aligned}x &= 4 \\y &= 4\end{aligned}$$

or

$$\begin{aligned}x &= 3 \\y &= 5\end{aligned}$$

The same is true for programming languages. In programming, variables are containers for values that change. Variables can hold all kind of values and also the results of computations. Variables have a name and a value separated by an equals sign (=). Variable names can be any letter or word, but bear in mind that there are restrictions from language to language of what you can use, as some words are reserved for other functionality.

Let's check out how it works in Javascript, The following code defines two variables, computes the result of adding the two and defines this result as a value of a third variable.

```
var x = 5;  
var y = 6;  
var result = x + y;
```

# Variable types

Computers are sophisticated and can make use of more complex variables than just numbers. This is where variable types come in. Variables come in several types and different languages support different types.

The most common types are:

- **Numbers**
  - **Float**: a number, like 1.21323, 4, -33.5, 100004 or 0.123
  - **Integer**: a number like 1, 12, -33, 140 but not 1.233
- **String**: a line of text like "boat", "elephant" or "damn, you are tall!"
- **Boolean**: either true or false, but nothing else
- **Arrays**: a collection of values like: 1,2,3,4,'I am bored now'
- **Objects**: a representation of a more complex object
- **null**: a variable that contains null contains no valid Number, String, Boolean, Array, or Object
- **undefined**: the undefined value is obtained when you use an object property that does not exist, or a variable that has been declared, but has no value assigned to it.

JavaScript is a “*loosely typed*” language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

## Exercise

Create a variable named `a` using the keyword `var`.

# Equality

Programmers frequently need to determine the equality of variables in relation to other variables. This is done using an equality operator.

The most basic equality operator is the `==` operator. This operator does everything it can to determine if two variables are equal, even if they are not of the same type.

For example, assume:

```
var foo = 42;
var bar = 42;
var baz = "42";
var qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will also evaluate to `true` despite `foo` and `baz` being different types. Behind the scenes the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.

The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

# Numbers

JavaScript has **only one type of numbers** – 64-bit float point. It's the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value.

In this chapter, we'll learn how to create numbers and perform operations on them (like additions and subtractions).

# Creation

Creating a number is easy, it can be done just like for any other variable type using the `var` keyword.

Numbers can be created from a constant value:

```
// This is a float:  
var a = 1.2;  
  
// This is an integer:  
var b = 10;
```

Or from the value of another variable:

```
var a = 2;  
var b = a;
```

## Exercise

Create a variable `x` which equals `10` and create a variable `y` which equals `a`.

```
var a = 11;
```

# Operators

You can apply mathematic operations to numbers using some basic operators like:

- **Addition:** `c = a + b`
- **Subtraction:** `c = a - b`
- **Multiplication:** `c = a * b`
- **Division:** `c = a / b`

You can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

## Exercise

Create a variable `x` equal to the sum of `a` and `b` divided by `c` and finally multiplied by `d`.

```
var a = 2034547;  
var b = 1.567;  
var c = 6758.768;  
var d = 45084;  
  
var x =
```

# Advanced Operators

Some advanced operators can be used, such as:

- **Modulus (division remainder):** `x = y % 2`
- **Increment:** Given `a = 5`
  - `c = a++`, Results: `c = 5` and `a = 6`
  - `c = ++a`, Results: `c = 6` and `a = 6`
- **Decrement:** Given `a = 5`
  - `c = a--`, Results: `c = 5` and `a = 4`
  - `c = --a`, Results: `c = 4` and `a = 4`

## Exercise

Define a variable `c` as the modulus of the decremented value of `x` by 3.

```
var x = 10;  
var c =
```

# Strings

JavaScript strings share many similarities with string implementations from other high-level languages. They represent text based messages and data.

In this course we will cover the basics. How to create new strings and perform common operations on them.

Here is an example of a string:

```
"Hello World"
```

## **Creation**

You can define strings in JavaScript by enclosing the text in single quotes or double quotes:

```
// Single quotes can be used
var str = 'Our lovely string';

// Double quotes as well
var otherStr = "Another nice string";
```

In Javascript, Strings can contain UTF-8 characters:

"» español English »»»»» العربية português »»»»» русский »»»»»»»";

**Note:** Strings can not be subtracted, multiplied or divided.

## Exercise

Create a variable named `str` set to the value `"abc"`.

## Concatenation

Concatenation involves adding two or more strings together, creating a larger string containing the combined data of those original strings. This is done in JavaScript using the `+` operator.

```
var bigStr = 'Hi ' + 'JS strings are nice ' + 'and ' + 'easy to add';
```

### Exercise

Add up the different names so that the `'fullName'` variable contains John's complete name.

```
var firstName = "John";
var lastName = "Smith";

var fullName =
```

# Length

It's easy in Javascript to know how many characters are in string using the property `.length`.

```
// Just use the property .length  
var size = 'Our lovely string'.length;
```

**Note:** Strings can not be subtracted, multiplied or divided.

## Exercise

Store in the variable named `size` the length of `str`.

```
var str = "Hello World";  
  
var size =
```

## Conditional Logic

A condition is a test for something. Conditions are very important for programming, in several ways:

First of all conditions can be used to ensure that your program works, regardless of what data you throw at it for processing. If you blindly trust data, you'll get into trouble and your programs will fail. If you test that the thing you want to do is possible and has all the required information in the right format, that won't happen, and your program will be a lot more stable. Taking such precautions is also known as programming defensively.

The other thing conditions can do for you is allow for branching. You might have encountered branching diagrams before, for example when filling out a form. Basically, this refers to executing different "branches" (parts) of code, depending on if the condition is met or not.

In this chapter, we'll learn the base of conditional logic in Javascript.

## Condition If

The easiest condition is an if statement and its syntax is `if(condition){ do this ... }`. The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value:

```
var country = 'France';
var weather;
var food;
var currency;

if(country === 'England') {
    weather = 'horrible';
    food = 'filling';
    currency = 'pound sterling';
}

if(country === 'France') {
    weather = 'nice';
    food = 'stunning, but hardly ever vegetarian';
    currency = 'funny, small and colourful';
}

if(country === 'Germany') {
    weather = 'average';
    food = 'wurst thing ever';
    currency = 'funny, small and colourful';
}

var message = 'this is ' + country + ', the weather is ' +
             weather + ', the food is ' + food + ' and the ' +
             'currency is ' + currency;
```

**Note:** Conditions can also be nested.

### Exercise

Fill up the value of `name` to validate the condition.

```
var name =

if (name === "John") {

}
```

## Else

There is also an `else` clause that will be applied when the first condition isn't true. This is very powerful if you want to react to any value, but single out one in particular for special treatment:

```
var umbrellaMandatory;

if(country === 'England'){
    umbrellaMandatory = true;
} else {
    umbrellaMandatory = false;
}
```

The `else` clause can be joined with another `if`. Lets remake the example from the previous article:

```
if(country === 'England') {
    ...
} else if(country === 'France') {
    ...
} else if(country === 'Germany') {
    ...
}
```

### Exercise

Fill up the value of `name` to validate the `else` condition.

```
var name =

if (name === "John") {

} else if (name === "Aaron") {
    // Valid this condition
}
```

# Comparators

Lets now focus on the conditional part:

```
if (country === "France") {
    ...
}
```

The conditional part is the variable `country` followed by the three equal signs (`==`). Three equal signs tests if the variable `country` has both the correct value (`France`) and also the correct type (`String`). You can test conditions with double equal signs, too, however a conditional such as `if (x == 5)` would then return true for both `var x = 5;` and `var x = "5";`. Depending on what your program is doing, this could make quite a difference. It is highly recommended as a best practice that you always compare equality with three equal signs (`==` and `!=`) instead of two (`=` and `!=`).

Other conditional test:

- `x > a` : is `x` bigger than `a`?
- `x < a` : is `x` less than `a`?
- `x <= a` : is `x` less than or equal to `a`?
- `x >= a` : is `x` greater than or equal to `a`?
- `x != a` : is `x` not `a`?
- `x` : does `x` exist?

## Exercise

Add a condition to change the value of `a` to the number 10 if `x` is bigger than 5.

```
var x = 6;
var a = 0;
```

# Logical Comparison

In order to avoid the if-else hassle, simple logical comparisons can be utilised.

```
var topper = (marks > 85) ? "YES" : "NO";
```

In the above example, `?` is a logical operator. The code says that if the value of `marks` is greater than 85 i.e. `marks > 85`, then `topper = YES`; otherwise `topper = NO`. Basically, if the comparison condition proves true, the first argument is accessed and if the comparison condition is false, the second argument is accessed.

## Concatenate conditions

Furthermore you can concatenate different conditions with "or" or "and" statements, to test whether either statement is true, or both are true, respectively.

In JavaScript "or" is written as `||` and "and" is written as `&&`.

Say you want to test if the value of `x` is between 10 and 20—you could do that with a condition stating:

```
if(x > 10 && x < 20) {
    ...
}
```

If you want to make sure that country is either "England" or "Germany" you use:

```
if(country === 'England' || country === 'Germany') {
    ...
}
```

**Note:** Just like operations on numbers, Conditions can be grouped using parenthesis, ex: `if ( (name === "John" || name === "Jennifer") && country === "France" ) .`

### Exercise

Fill up the 2 conditions so that `primaryCategory` equals ``E/J`` only if name equals ``John`` and country is ``England``, and so that `secondaryCategory` equals ``E|J`` only if name equals ``John`` or country is ``England``

```
var name = "John";
var country = "England";
var primaryCategory, secondaryCategory;

if ( /* Fill here */ ) {
    primaryCategory = "E/J";
}
if ( /* Fill here */ ) {
    secondaryCategory = "E|J";
}
```

# Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
var numbers = [1, 1, 2, 3, 5, 8];
```

# Indices

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. Indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets [] are used to signify you are referring to an index of an array.

```
// This is an array of strings
var fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
var banana = fruits[1];
```

## Exercise

Define the variables using the indices of the array

```
var cars = ["Mazda", "Honda", "Chevy", "Ford"]
var honda =
var ford =
var chevy =
var mazda =
```

# Length

Arrays have a property called length, and it's pretty much exactly as it sounds, it's the length of the array.

```
var array = [1 , 2, 3];  
  
// Result: l = 3  
var l = array.length;
```

## Exercise

Define the variable a to be the number value of the length of the array

```
var array = [1, 1, 2, 3, 5, 8];  
var l = array.length;  
var a =
```

# Loops

Loops are repetitive conditions where one variable in the loop changes. Loops are handy, if you want to run the same code over and over again, each time with a different value.

Instead of writing:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

You can write:

```
for (var i=0; i < cars.length; i++) {
    doThing(cars[i]);
}
```

# For Loop

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

```
for(condition; end condition; change){  
    // do it, do it now  
}
```

Lets for example see how to execute the same code ten-times using a `for` loop:

```
for(var i = 0; i < 10; i = i + 1){  
    // do this code ten-times  
}
```

**Note:** `i = i + 1` can be written `i++`.

## Exercise

Using a for-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) from 0 to 99.

```
var message = "";
```

# While Loop

While Loops repetitively execute a block of code as long as a specified condition is true.

```
while(condition){  
    // do it as long as condition is true  
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0, x = "";  
while (i < 5) {  
    x = x + "The number is " + i;  
    i++;  
}
```

The Do/While Loop is a variant of the while loop. This loop will execute the code block once before checking if the condition is true. It then repeats the loop as long as the condition is true:

```
do {  
    // code block to be executed  
} while (condition);
```

**Note:** Be careful to avoid infinite looping if the condition is always true!

## Exercise

Using a while-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) as long as its length (`message.length`) is less than 100.

```
var message = "";
```

## Do...While Loop

The do...while statement creates a loop that executes a specified statement until the test condition evaluates to be false. The condition is evaluated after executing the statement. Syntax for do... while is

```
do{  
    // statement  
}  
while(expression) ;
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;  
do {  
    document.write(i + " ");  
    i++; // incrementing i by 1  
} while (i < 10);
```

*Note:* `i = i + 1` can be written `i++`.

### Exercise

Using a do...while-loop, print numbers between less than 5.

```
var i = 0;
```

# Functions

Functions, are one of the most powerful and essential notions in programming.

Functions like mathematical functions perform transformations, they take input values called **arguments** and **return** an output value.

# Declaring Functions

Functions, like variables, must be declared. Let's declare a function `double` that accepts an **argument** called `x` and **returns** the double of `x`:

```
function double(x) {  
    return 2 * x;  
}
```

*Note:* the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
var double = function(x) {  
    return 2 * x;  
};
```

*Note:* the function above **may not** be referenced before it is defined, just like any other variable.

## Exercise

Declare a function named `triple` that takes an argument and returns its triple.

# Higher Order Functions

Higher order functions are functions that manipulate other functions. For example, a function can take other functions as arguments and/or produce a function as its return value. Such *fancy* functional techniques are powerful constructs available to you in JavaScript and other high-level languages like python, lisp, etc.

We will now create two simple functions, `add_2` and `double`, and a higher order function called `map`. `map` will accept two arguments, `func` and `list` (its declaration will therefore begin `map(func, list)`), and return an array. `func` (the first argument) will be a function that will be applied to each of the elements in the array `list` (the second argument).

```
// Define two simple functions
var add_2 = function(x) {
    return x + 2;
};

var double = function(x) {
    return 2 * x;
};

// map is cool function that accepts 2 arguments:
// func    the function to call
// list    a array of values to call func on
var map = function(func, list) {
    var output=[];           // output list
    for(idx in list) {
        output.push( func(list[idx]) );
    }
    return output;
}

// We use map to apply a function to an entire list
// of inputs to "map" them to a list of corresponding outputs
map(add_2, [5,6,7]) // => [7, 8, 9]
map(double, [5,6,7]) // => [10, 12, 14]
```

The functions in the above example are simple. However, when passed as arguments to other functions, they can be composed in unforeseen ways to build more complex functions.

For example, if we notice that we use the invocations `map(add_2, ...)` and `map(double, ...)` very often in our code, we could decide we want to create two special-purpose list processing functions that have the desired operation baked into them. Using function composition, we could do this as follows:

```
process_add_2 = function(list) {
    return map(add_2, list);
}
process_double = function(list) {
    return map(double, list);
}
process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Now let's create a function called `buildProcessor` that takes a function `func` as input and returns a `func`-processor, that is, a function that applies `func` to each input in list.

```
// a function that generates a list processor that performs
var buildProcessor = function(func) {
  var process_func = function(list) {
    return map(func, list);
  }
  return process_func;
}
// calling buildProcessor returns a function which is called with a list input

// using buildProcessor we could generate the add_2 and double list processors as follows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Let's look at another example. We'll create a function called `buildMultiplier` that takes a number `x` as input and returns a function that multiplies its argument by `x` :

```
var buildMultiplier = function(x) {
  return function(y) {
    return x * y;
  }
}

var double = buildMultiplier(2);
var triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

### Exercise

Define a function named `negate` that takes `add1` as argument and returns a function, that returns the negation of the value returned by `add1`. (Things get a bit more complicated ;))

```
var add1 = function (x) {
  return x + 1;
};

var negate = function(func) {
  // TODO
};

// Should return -6
// Because (5+1) * -1 = -6
negate(add1)(5);
```

# Objects

The primitive types of JavaScript are `true` , `false` , numbers, strings, `null` and `undefined` . **Every other value is an `object` .**

In JavaScript objects contain `propertyName : propertyValue` pairs.

# Creation

There are two ways to create an `object` in JavaScript:

1. literal

```
var object = {};
// Yes, simply a pair of curly braces!
```

*Note:* this is the **recommended** way.

2. and object-oriented

```
var object = new Object();
```

*Note:* it's almost like Java.

# Properties

Object's property is a `propertyName : PropertyValue` pair, where **property name can be only a string**. If it's not a string, it gets casted into a string. You can specify properties **when creating** an object **or later**. There may be zero or more properties separated by commas.

```
var language = {
  name: 'JavaScript',
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: 'Brendan',
    lastName: 'Eich'
  },
  // Yes, objects can be nested!
  getAuthorFullName: function(){
    return this.author.firstName + " " + this.author.lastName;
  }
  // Yes, functions can be values too!
};


```

The following code demonstates how to **get** a property's value.

```
var variable = language.name;
// variable now contains "JavaScript" string.
variable = language['name'];
// The lines above do the same thing. The difference is that the second one lets you use litteraly any string as a prop
variable = language.newProperty;
// variable is now undefined, because we have not assigned this property yet.
```

The following example shows how to **add** a new property **or change** an existing one.

```
language.newProperty = 'new value';
// Now the object has a new property. If the property already exists, its value will be replaced.
language['newProperty'] = 'changed value';
// Once again, you can access properties both ways. The first one (dot notation) is recomended.
```

## Mutable

The difference between objects and primitive values is that **we can change objects**, whereas primitive values are immutable.

```
var myPrimitive = "first value";
myPrimitive = "another value";
// myPrimitive now points to another string.
var myObject = { key: "first value"};
myObject.key = "another value";
// myObject points to the same object.
```

# Reference

Objects are **never copied**. They are passed around by reference.

```
// Imagine I had a pizza
var myPizza = {slices: 5};
// And I shared it with You
var yourPizza = myPizza;
// I eat another slice
myPizza.slices = myPizza.slices - 1;
var numberOfSlicesLeft = yourPizza.slices;
// Now We have 4 slices because myPizza and yourPizza
// reference to the same pizza object.
var a = {}, b = {}, c = {};
// a, b, and c each refer to a
// different empty object
a = b = c = {};
// a, b, and c all refer to
// the same empty object
```

# Prototype

Every object is linked to a prototype object from which it inherits properties.

All objects created from object literals ( {}) are automatically linked to Object.prototype, which is an object that comes standard with JavaScript.

When a JavaScript interpreter (a module in your browser) tries to find a property, which You want to retrieve, like in the following code:

```
var adult = {age: 26},
    retrievedProperty = adult.age;
// The line above
```

First, the interpreter looks through every property the object itself has. For example, `adult` has only one own property — `age`. But besides that one it actually has a few more properties, which were inherited from Object.prototype.

```
var stringRepresentation = adult.toString();
// the variable has value of '[object Object]'
```

`toString` is an Object.prototype's property, which was inherited. It has a value of a function, which returns a string representation of the object. If you want it to return a more meaningful representation, then you can override it. Simply add a new property to the `adult` object.

```
adult.toString = function(){
    return "I'm "+this.age;
}
```

If you call the `toString` function now, the interpreter will find the new property in the object itself and stop.

Thus the interpreter retrieves the first property it will find on the way from the object itself and further through its prototype.

To set your own object as a prototype instead of the default Object.prototype, you can invoke `Object.create` as follows:

```
var child = Object.create(adult);
/* This way of creating objects lets us easily replace the default Object.prototype with the one we want. In this case,
child.age = 8;
/* Previously, child didn't have its own age property, and the interpreter had to look further to the child's prototype.
Now, when we set the child's own age, the interpreter will not go further.
Note: adult's age is still 26. */
var stringRepresentation = child.toString();
// The value is "I'm 8".
/* Note: we have not overridden the child's toString property, thus the adult's method will be invoked. If adult did no
```

child's prototype is `adult`, whose prototype is `Object.prototype`. This sequence of prototypes is called **prototype chain**.

## Delete

`delete` can be used to **remove a property** from an object. It will remove a property from the object if it has one. It will not look further in the prototype chain. Removing a property from an object may allow a property from the prototype chain to shine through:

```
var adult = {age:26},  
    child = Object.create(adult);  
    child.age = 8;  
  
delete child.age;  
/* Remove age property from child, revealing the age of the prototype, because then it is not overriden. */  
var prototypeAge = child.age;  
// 26, because child does not have its own age property.
```

## Enumeration

The `for in` statement can loop over all of the property names in an object. The enumeration will include functions and prototype properties.

```
var fruit = {  
    apple: 2,  
    orange:5,  
    pear:1  
,  
sentence = 'I have ',  
quantity;  
for (kind in fruit){  
    quantity = fruit[kind];  
    sentence += quantity+' '+kind+  
        (quantity==1?'':'s')+  
        ', ';  
}  
// The following line removes the trailing coma.  
sentence = sentence.substr(0,sentence.length-2)+'.';  
// I have 2 apples, 5 oranges, 1 pear.
```

## Global footprint

If you are developing a module, which might be running on a web page, which also runs other modules, then you must beware the variable name overlapping.

Suppose we are developing a counter module:

```
var myCounter = {  
    number : 0,  
    plusPlus : function(){  
        this.number : this.number + 1;  
    },  
    isGreaterThanTen : function(){  
        return this.number > 10;  
    }  
}
```

*Note:* this technique is often used with closures, to make the internal state immutable from the outside.

The module now takes only one variable name — `myCounter`. If any other module on the page makes use of such names like `number` or `isGreaterThanTen` then it's perfectly safe, because we will not override each others values;

# OpenLayers Workshop

Welcome to the **OpenLayers 3 Workshop**. This workshop is designed to give you a comprehensive overview of OpenLayers as a web mapping solution.

## Setup

These instructions assume that you are starting with an `openlayers-workshop.zip` archive from the latest [workshop release](#). In addition, you'll need [Node](#) installed to run a development server for the OpenLayers library.

After extracting the zip, change into the `openlayers-workshop` directory and install some additional dependencies:

```
npm install
```

Now you're ready to start the workshop server. This serves up the workshop documentation in addition to providing a debug loader for the OpenLayers library.

```
npm start
```

This will start a development server where you can read the workshop documentation and work through the exercises:  
<http://terrestris.github.io/momo3-ws/>.

## Overview

This workshop is presented as a set of modules. In each module you will perform a set of tasks designed to achieve a specific goal for that module. Each module builds upon lessons learned in previous modules and is designed to iteratively build up your knowledge base.

The following modules will be covered in this workshop:

- [Basics](#) - Learn how to add a map to a webpage with OpenLayers.
- [Layers and Sources](#) - Learn about layers and sources.
- [Controls and Interactions](#) - Learn about using map controls and interactions.
- [Vector Topics](#) - Explore vector layers in depth.
- [Custom Builds](#) - Create custom builds.

## Basics

- [Creating a map](#)
- [Dissecting your map](#)
- [Useful resources](#)

# Creating a Map

In OpenLayers, a map is a collection of layers and various interactions and controls for dealing with user interaction. A map is generated with three basic ingredients: markup, style declarations, and initialization code.

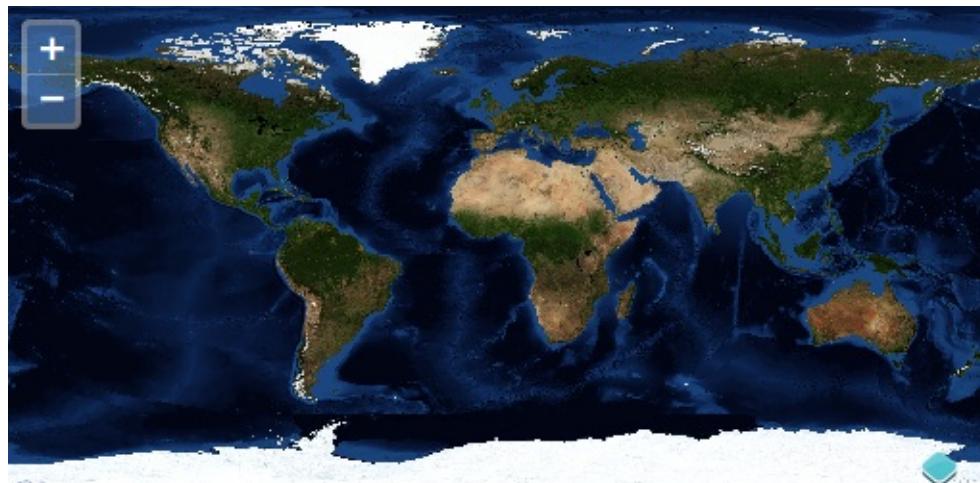
## Working Example

Let's take a look at a fully working example of an OpenLayers 3 map.

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

## Tasks

1. Make sure you've completed the [setup instructions](#) to install dependencies and get the debug server running.
2. Copy the text above into a new file called `map.html`, and save it in the root of the workshop directory.
3. Open the working map in your web browser: <http://terrestris.github.io/momo3-ws//map.html>



*A working map displaying imagery of the world*

Having successfully created our first map, we'll continue by looking more closely at [the parts](#).

# Dissecting Your Map

As demonstrated in the [previous section](#), a map is generated by bringing together markup, style declarations, and initialization code. We'll look at each of these parts in a bit more detail.

## Map Markup

The markup for the map in the [previous example](#) generates a single document element:

```
<div id="map"></div>
```

This `<div>` element will serve as the container for our map viewport. Here we use a `<div>` element, but the container for the viewport can be any block-level element.

In this case, we give the container an `id` attribute so we can reference it as the target of our map.

## Map Style

OpenLayers comes with a default stylesheet that specifies how map-related elements should be styled. We've explicitly included this stylesheet in the `map.html` page (`<link rel="stylesheet" href="/ol.css" type="text/css">`).

OpenLayers doesn't make any guesses about the size of your map. Because of this, following the default stylesheet, we need to include at least one custom style declaration to give the map some room on the page.

```
<link rel="stylesheet" href="/ol.css" type="text/css">
<style>
  #map {
    height: 256px;
    width: 512px;
  }
</style>
```

In this case, we're using the map container's `id` value as a selector, and we specify the width (`512px`) and the height (`256px`) for the map container.

The style declarations are directly included in the `<head>` of our document. In most cases, your map related style declarations will be a part of a larger website theme linked in external stylesheets.

## Map Initialization

The next step in generating your map is to include some initialization code. In our case, we have included a `<script>` element at the bottom of our document `<body>` to do the work:

```
<script>
  var map = new ol.Map({
    target: 'map',
    layers: [
      new ol.layer.Tile({
        source: new ol.source.TileWMS({
          url: 'http://demo.opengeo.org/geoserver/wms',
          params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
        })
      })
    ],
    view: new ol.View({
      projection: 'EPSG:4326',
      center: [0, 0],
      zoom: 0,
      maxResolution: 0.703125
    })
  });
</script>
```

The order of these steps is important. Before our custom script can be executed, the OpenLayers library must be loaded. In our example, the OpenLayers library is loaded in the `&lt;head&gt;` of our document with `&lt;script`

```
src="/loader.js".&gt;&lt;/script&gt; .
```

Similarly, our custom map initialization code (above) cannot run until the document element that serves as the viewport container, in this case `&lt;div id="map"&gt;&lt;/div&gt;`, is ready. By including the initialization code at the end of the document `&lt;body&gt;`, we ensure that the library is loaded and the viewport container is ready before generating our map.

Let's look in more detail at what the map initialization script is doing. Our script creates a new `ol.Map` object with a few config options:

```
target: 'map'
```

We use the viewport container's `id` attribute value to tell the map constructor where to render the map. In this case, we pass the string value `"map"` as the target to the map constructor. This syntax is a shortcut for convenience. We could be more explicit and provide a direct reference to the element (e.g. `document.getElementById('map')`).

The layers config creates a layer to be displayed in our map:

```
layers: [
  new ol.layer.Tile({
    source: new ol.source.TileWMS({
      url: 'http://demo.opengeo.org/geoserver/wms',
      params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
    })
  })
],
```

Don't worry about the syntax here if this part is new to you. Layer creation will be covered in another module. The important part to understand is that our map view is a collection of layers. In order to see a map, we need to include at least one layer.

The final step is defining the view. We specify a projection, a center and a zoom level. We also specify a `maxResolution` to make sure we don't request bounding boxes that GeoWebCache cannot handle.

```
view: new ol.View({
  projection: 'EPSG:4326',
  center: [0, 0],
  zoom: 0,
  maxResolution: 0.703125
})
```

You've successfully dissected your first map! Next let's [learn more](#) about developing with OpenLayers.



# OpenLayers Resources

The OpenLayers library contains a wealth of functionality. Though the developers have worked hard to provide examples of that functionality and have organized the code in a way that allows other experienced developers to find their way around, many users find it a challenge to get started from scratch.

## Learn by Example

New users will most likely find diving into the OpenLayer's example code and experimenting with the library's possible functionality the most useful way to begin.

- <http://openlayers.org/en/master/examples/>

## Browse the Documentation

For further information on specific topics, browse the growing collection of OpenLayers documentation.

- <http://openlayers.org/en/master/doc/quickstart.html>
- <http://openlayers.org/en/master/doc/tutorials>

## Find the API Reference

After understanding the basic components that make-up and control a map, search the API reference documentation for details on method signatures and object properties. If you only want to see the stable part of the API, make sure to check the `Stable Only` checkbox.

- <http://openlayers.org/en/master/apidoc/>

## Join the Community

OpenLayers is supported and maintained by a community of developers and users like you. Whether you have questions to ask or code to contribute, you can get involved by using the `openlayers-3` tag on StackOverflow for usage questions or signing up for the developers mailing list.

- <http://stackoverflow.com/questions/tagged/openlayers-3>
- <https://groups.google.com/forum/#!forum/ol3-dev>

## Reporting issues

For reporting issues it is important to understand the several flavours in which the OpenLayers library is distributed:

- `ol.js` - the script which is built using the Closure Compiler in advanced mode (not human readable)
- `ol-debug.js` - human readable version to be used during development

When you encounter an issue, it is important to report the issue using `ol-debug.js`. Also include the full stack trace which you can find using Web Developer tools such as Chrome's Developer Tools. To test this out we are going to make a mistake in map.html by changing `ol.layer.Tile` into `ol.layer.Image`. The error you will see is: `Uncaught TypeError: undefined is not a function`. If you report this to the mailing list, nobody will know what it means. So first, we are going to change the script tag which points to `ol.js` to point to `ol-debug.js` instead. Reload the page. The debugger will now stop on the error, and we can see the full stack trace.

The screenshot shows the Chrome DevTools debugger. The left pane displays the source code for 'map.html' with a blue bar highlighting the line 'goog.asserts.DEFAULT\_ERROR\_HANDLER = function(e) { throw e; };'. The right pane shows the Call Stack, which is currently paused on an exception of type 'goog.asserts.AssertionError'. The stack trace includes frames from 'ol-debug.js' and 'ol.js'. The Local scope variables panel shows the exception object, with properties <exception>, e, and this. The Global scope is also visible.

```
3557 '}  
3558 goog.inherits(goog.asserts.AssertionError, goog.debug.Error);  
3559  
3560 /** @override */  
3561 goog.asserts.AssertionError.prototype.name = 'AssertionError';  
3562  
3563  
3564 /**  
3565 * The default error handler.  
3566 * @param {!goog.asserts.AssertionError} e The exception to be handled.  
3567 */  
3568 goog.asserts.DEFAULT_ERROR_HANDLER = function(e) { throw e; };  
3569  
3570  
3571 /**  
3572 * The handler responsible for throwing or logging assertion errors.  
3573 * @private {function(!goog.asserts.AssertionError)}  
3574 */  
3575 goog.asserts.errorHandler_ = goog.asserts.DEFAULT_ERROR_HANDLER;  
3576  
3577  
3578 /**  
3579 <
```

*At a breakpoint in the debugger*

## Layers and Sources

- [WMS sources](#)
- Tiled sources
- Proprietary tile providers
- Vector data
- Image vector source

# Web Map Service Layers

When you add a layer to your map, the layer's source is typically responsible for fetching the data to be displayed. The data requested can be either raster or vector data. You can think of raster data as information rendered as an image on the server side. Vector data is delivered as structured information from the server and may be rendered for display on the client (your browser).

There are many different types of services that provide raster map data. This section deals with providers that conform with the OGC (Open Geospatial Consortium, Inc.) [Web Map Service \(WMS\)](#) specification.

## Creating a Layer

We'll start with a fully working map example and modify the layers to get an understanding of how they work.

Let's take a look at the following code:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

## Tasks

1. If you haven't already done so, save the text above as `map.html` in the root of your workshop directory.
2. Open the page in your browser to confirm things work: <http://terrestris.github.io/momo3-ws//map.html>

## The `ol.layer.Tile` Constructor

The `ol.layer.Tile` constructor gets an object literal of type `olx.layer.TileOptions` see:

<http://openlayers.org/en/master/apidoc/ol.layer.Tile.html> In this case we are providing the source key of the options with an

`ol.source.TileWMS`. A human-readable title for the layer can be provided with the title key, but basically any arbitrary name for the key can be used here. In OpenLayers 3 there is a separation between layers and sources, whereas in OpenLayers 2 this was all part of the layer.

`ol.layer.Tile` represents a regular grid of images, `ol.layer.Image` represents a single image. Depending on the layer type, you would use a different source (`ol.source.TileWMS` versus `ol.source.ImageWMS`) as well.

## The `ol.source.TileWMS` Constructor

The `ol.source.TileWMS` constructor has a single argument which is defined by:

<http://openlayers.org/en/master/apidoc/ol.source.TileWMS.html>. The url is the online resource of the WMS service, and params is an object literal with the parameter names and their values. Since the default WMS version is 1.3.0 now in OpenLayers, you might need to provide a lower version in the params if your WMS does not support WMS 1.3.0.

```
layers: [
  new ol.layer.Tile({
    title: 'Global Imagery',
    source: new ol.source.TileWMS({
      url: 'http://demo.opengeo.org/geoserver/wms',
      params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
    })
  })
]
```

## Tasks

1. This same WMS offers a [Natural Earth](#) layer named `'ne:NE1_HR_LC_SR_W_DR'`. Change the value of the `LAYERS` parameter from `'nasa:bluemarble'` to `'ne:NE1_HR_LC_SR_W_DR'`. Your revised `ol.layer.Tile` Constructor should look like:

```
new ol.layer.Tile({
  title: 'Global Imagery',
  source: new ol.source.TileWMS({
    url: 'http://demo.opengeo.org/geoserver/wms',
    params: {LAYERS: 'ne:NE1_HR_LC_SR_W_DR', VERSION: '1.1.1'}
  })
})
```

2. Change your layer and source to have a single image instead of tiles. Look at the following API doc pages for hints:  
<http://openlayers.org/en/master/apidoc/ol.layer.Image.html> and <http://openlayers.org/en/master/apidoc/ol.source.ImageWMS.html>. Use the Network tab of your browser's developer tools to make sure a single image is requested and not 256x256 pixel tiles.



Having worked with dynamically rendered data from a Web Map Service, let's move on to learn about [cached tile services](#).

# Cached Tiles

By default, the Tile layer makes requests for 256 x 256 (pixel) images to fill your map viewport and beyond. As you pan and zoom around your map, more requests for images go out to fill the areas you haven't yet visited. While your browser will cache some requested images, a lot of processing work is typically required for the server to dynamically render images.

Since tiled layers make requests for images on a regular grid, it is possible for the server to cache these image requests and return the cached result next time you (or someone else) visits the same area - resulting in better performance all around.

## `ol.source.XYZ`

The Web Map Service specification allows a lot of flexibility in terms of what a client can request. Without constraints, this makes caching difficult or impossible in practice.

At the opposite extreme, a service might offer tiles only at a fixed set of zoom levels and only for a regular grid. These can be generalized as tiled layers with an XYZ source - you can consider X and Y to indicate the column and row of the grid and Z to represent the zoom level.

## `ol.source.OSM`

The [OpenStreetMap \(OSM\)](#) project is an effort to collect and make freely available map data for the world. OSM provides a few different renderings of their data as cached tile sets. These renderings conform to the basic XYZ grid arrangement and can be used in an OpenLayers map. The `ol.source.osm` layer source accesses OpenStreetMap tiles.

## Tasks

1. Open the `map.html` file from the [previous section](#) in a text editor and change the map initialization code to look like the following:

```
<script>
  var map = new ol.Map({
    target: 'map',
    layers: [
      new ol.layer.Tile({
        source: new ol.source.OSM()
      })
    ],
    view: new ol.View({
      center: ol.proj.fromLonLat([126.97, 37.56]),
      zoom: 9
    }),
    controls: ol.control.defaults({
      attributionOptions: {
        collapsible: false
      }
    });
  });
</script>
```

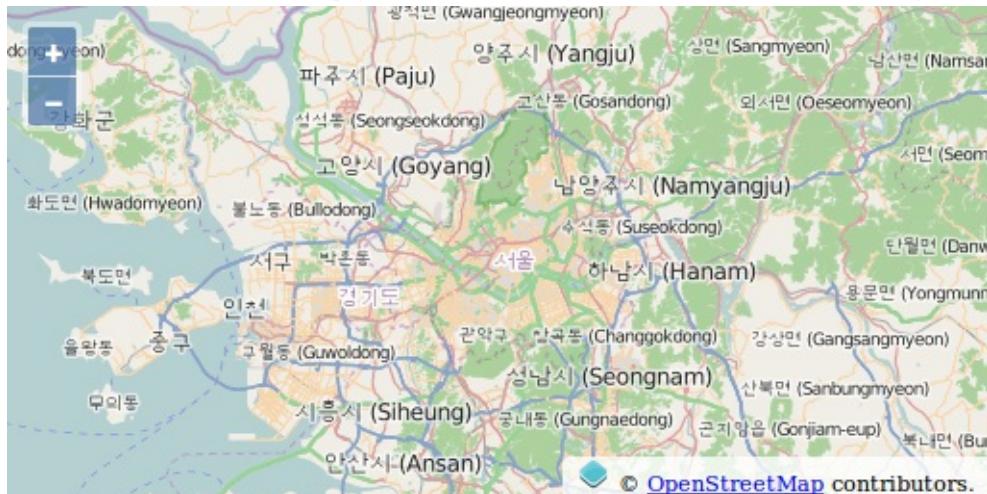
2. In the `&lt;head&gt;` of the same document, add a few style declarations for the layer attribution.

```

<style>
  #map {
    width: 512px;
    height: 256px;
  }
  .ol-attribution a {
    color: black;
  }
</style>

```

3. Save your changes, and refresh the page in your browser: <http://terrestris.github.io/momo3-ws//map.html>



## A Closer Look

### Projections

Review the view definition of the map:

```

view: new ol.View({
  center: ol.proj.fromLonLat([126.97, 37.56]),
  zoom: 9
})

```

Geospatial data can come in any number of coordinate reference systems. One data set might use geographic coordinates (longitude and latitude) in degrees, and another might have coordinates in a local projection with units in meters. A full discussion of coordinate reference systems is beyond the scope of this module, but it is important to understand the basic concept.

OpenLayers 3 needs to know the coordinate system for your data. Internally, this is represented with an `ol.proj.Projection` object but strings can also be supplied.

The OpenStreetMap tiles that we will be using are in a Mercator projection. Because of this, we need to set the initial center using Mercator coordinates. Since it is relatively easy to find out the coordinates for a place of interest in geographic coordinates, we use the `ol.proj.fromLonLat` method to turn geographic coordinates ( `&#39;EPSG:4326&#39;` ) into Mercator coordinates ( `&#39;EPSG:3857&#39;` ).

### Alternative Projections

OpenLayers 3 includes transforms between Geographic ( `&#39;EPSG:4326&#39;` ) and Web Mercator ( `&#39;EPSG:3857&#39;` ) coordinate reference systems. Because of this, we can use the `ol.proj.fromLonLat` function above without any extra work. If you want to work with data in a different projection, you need to include some additional information before using the `ol.proj.*` functions.

For example, if you wanted to work with data in the 'EPSG:21781' coordinate reference system, you would include the following two script tags in your page:

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/proj4js/2.3.6/proj4.js" type="text/javascript"></script>
<script src="http://epsg.io/21781-1753.js" type="text/javascript"></script>
```

Then in your application code, you could register this projection and set its validity extent as follows:

```
// This creates a projection object for the EPSG:21781 projection
// and sets a "validity extent" in that projection object.
var projection = ol.proj.get('EPSG:21781');
projection.setExtent([485869.5728, 76443.1884, 837076.5648, 299941.7864]);
```

The extent information can be looked up at <http://epsg.io/>, using the EPSG code.

## Layer Creation

```
layers: [
  new ol.layer.Tile({
    source: new ol.source.OSM()
  })
],
```

As before, we create a layer and add it to the layers array of our map config object. This time, we accept all the default options for the source.

## Style

```
.ol-attribution a {
  color: black;
}
```

A treatment of map controls is also outside of the scope of this module, but these style declarations give you a sneak preview. By default, an `ol.control.Attribution` control is added to all maps. This lets layer sources display attribution information in the map viewport. The declarations above alter the style of this attribution for our map (notice the Copyright line at the bottom right of the map).

## Attribution Control Configuration

By default the `ol.control.Attribution` adds an `i` (information) button that can be pressed to actually displays the attribution information. To comply to [OpenStreetMap's Terms Of Use](#), and always display the OpenStreetMap attribution information, the following is used in the options object passed to the `ol.Map` constructor:

```
controls: ol.control.defaults({
  attributionOptions: {
    collapsible: false
  }
})
```

This removes the `i` button, and makes the attribution information always visible.

Having mastered layers with publicly available cached tile sets, let's move on to working with [proprietary raster layers](#).

# Proprietary Raster Layers

In previous sections, we displayed layers based on a standards compliant WMS (OGC Web Map Service) and a custom tile cache. Online mapping (or at least the tiled map client) was largely popularized by the availability of proprietary map tile services. OpenLayers provides layer types that work with these proprietary services through their APIs.

In this section, we'll build on the example developed in the [previous section](#) by adding a layer using tiles from Bing.

## Bing!

Let's add a Bing layer.

### Tasks

1. In your `map.html` file, find where the OSM (OpenStreetMap) source is configured and change it into an `ol.source.BingMaps`

```
source: new ol.source.BingMaps({
  imagerySet: 'Road',
  key: 'Ak-dzM4wZjSqTlzveKz5u0d4IQ4bRzVI309GxmkgSVr1ewS6iPSr0vOKhA-CJ1m3'
})
```

*Note - The Bing tiles API requires that you register for an API key to use with your mapping application. The example here uses an API key that you should not use in production. To use the Bing layer in production, register for an API key at <https://www.bingmapsportal.com/>.*

2. Save your changes and reload `map.html` in your browser: <http://terrestris.github.io/momo3-ws//map.html>



## Complete Working Example

Your revised `map.html` file should look something like this:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
      .ol-attribution a {
        color: black;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map" class="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            source: new ol.source.BingMaps({
              imagerySet: 'Road',
              key: 'Ak-dzM4wZjSqTlzeKz5u0d4IQ4bRzVI309GxmkgSVr1ewS6iPSr0vOKhA-CJlm3'
            })
          })
        ],
        view: new ol.View({
          center: ol.proj.fromLonLat([126.97, 37.56]),
          zoom: 9
        })
      });
    </script>
  </body>
</html>
```

# Vector Layers

Vector Layers are represented by `ol.layer.Vector` and handle the client-side display of vector data. Currently OpenLayers 3 supports full vector rendering in the Canvas renderer, but only point geometries in the WebGL renderer.

## Rendering Features Client-Side

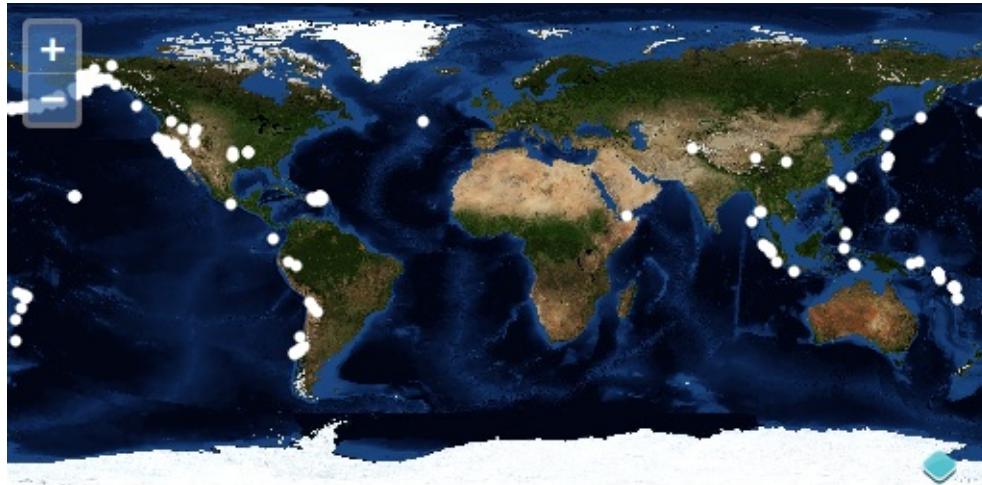
Let's go back to the WMS example to get a basic world map. We'll add some feature data on top of this in a vector layer.

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>
```

## Tasks

1. Open `map.html` in your text editor and copy in the contents of the initial WMS example. Save your changes and confirm that things look good in your browser: <http://terrestris.github.io/momo3-ws//map.html>
2. In your map initialization code add another layer after the Tile layer (paste the following). This adds a new vector layer to your map that requests a set of features stored in GeoJSON:

```
new ol.layer.Vector({
  title: 'Earthquakes',
  source: new ol.source.Vector({
    url: '/data/layers/7day-M2.5.json',
    format: new ol.format.GeoJSON()
  }),
  style: new ol.style.Style({
    image: new ol.style.Circle({
      radius: 3,
      fill: new ol.style.Fill({color: 'white'})
    })
  })
})
```



## A Closer Look

Let's examine that vector layer creation to get an idea of what is going on.

```
new ol.layer.Vector({
  title: 'Earthquakes',
  source: new ol.source.Vector({
    url: '/data/layers/7day-M2.5.json',
    format: new ol.format.GeoJSON()
  }),
  style: new ol.style.Style({
    image: new ol.style.Circle({
      radius: 3,
      fill: new ol.style.Fill({color: 'white'})
    })
  })
})
```

The layer is given the title `'Earthquakes'` and some custom options. In the options object, we've included a `source` of type `ol.source.Vector` which points to a url. We've given the source a `format` that will be used for parsing the data.

**Note** - In the case where you want to style the features based on an attribute, you would use a style function instead of an `ol.style.Style` for the `style` config option of `ol.layer.Vector`.

## Bonus Tasks

1. The white circles on the map represent `ol.Feature` objects on your `ol.layer.Vector` layer. Each of these features has attribute data with `title` and `summary` properties. Register a `'singleclick'` listener on your map that calls `forEachFeatureAtPixel` on the map, and displays earthquake information below the map viewport.
  2. The data for the vector layer comes from an earthquake feed published by the USGS (<http://earthquake.usgs.gov/earthquakes/catalogs/>). See if you can find additional data with spatial information in a format

supported by OpenLayers 3. If you save another document representing spatial data in your `data` directory, you should be able to view it in a vector layer on your map.

## Solutions

As a solution to the first bonus task you can add an `info` div below the map:

```
<div id="info"></div>
```

and add the following JavaScript code to display the title of the clicked feature:

```
map.on('singleclick', function(e) {
  var feature = map.forEachFeatureAtPixel(e.pixel, function(feature) {
    return feature;
  });
  var infoElement = document.getElementById('info');
  infoElement.innerHTML = feature ? feature.get('title') : '';
});
```

## Image Vector

In the previous example using an `ol.layer.Vector` you can see that the features are re-rendered continuously during animated zooming (the size of the point symbolizers remains fixed). With a vector layer, OpenLayers will re-render the source data with each animation frame. This provides consistent rendering of line strokes, point symbolizers, and labels with changes in the view resolution.

An alternative rendering strategy is to avoid re-rendering data during view transitions and instead reposition and scale the rendered output from the previous view state. This can be accomplished by using an `ol.layer.Image` with an `ol.source.ImageVector`. With this combination, "snapshots" of your data are rendered when the view is not animating, and these snapshots are reused during view transitions.

The example below uses an `ol.layer.Image` with an `ol.source.ImageVector`. Though this example only renders a small quantity of data, this combination would be appropriate for applications that render large quantities of relatively static data.

### `ol.source.ImageVector`

Let's go back to the vector layer example to get earthquake data on top of a world map.

```

<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <title>OpenLayers 3 example</title>
    <script src="/loader.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var map = new ol.Map({
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          }),
          new ol.layer.Vector({
            title: 'Earthquakes',
            source: new ol.source.Vector({
              url: '/data/layers/7day-M2.5.json',
              format: new ol.format.GeoJSON()
            }),
            style: new ol.style.Style({
              image: new ol.style.Circle({
                radius: 3,
                fill: new ol.style.Fill({color: 'white'})
              })
            })
          })
        ],
        view: new ol.View({
          projection: 'EPSG:4326',
          center: [0, 0],
          zoom: 0,
          maxResolution: 0.703125
        })
      });
    </script>
  </body>
</html>

```

## Tasks

1. Open `map.html` in your text editor and copy in the contents of the vector example from above. Save your changes and confirm that things look good in your browser: <http://terrestris.github.io/momo3-ws//map.html>
2. Change the vector layer into:

```

new ol.layer.Image({
  title: 'Earthquakes',
  source: new ol.source.ImageVector({
    source: new ol.source.Vector({
      url: '/data/layers/7day-M2.5.json',
      format: new ol.format.GeoJSON()
    }),
    style: new ol.style.Style({
      image: new ol.style.Circle({
        radius: 3,
        fill: new ol.style.Fill({color: 'white'})
      })
    })
  })
})

```

3. Reload <http://terrestris.github.io/momo3-ws//map.html> in the browser *Note* - You will see the same vector data but depicted as an image. This will still enable things like feature detection, but the vector data will be less sharp. So this is essentially a trade-off between performance and quality.

## A Closer Look

Let's examine the layer creation to get an idea of what is going on.

```

new ol.layer.Image({
  title: 'Earthquakes',
  source: new ol.source.ImageVector({
    source: new ol.source.Vector({
      url: '/data/layers/7day-M2.5.json',
      format: new ol.format.GeoJSON()
    }),
    style: new ol.style.Style({
      image: new ol.style.Circle({
        radius: 3,
        fill: new ol.style.Fill({color: 'white'})
      })
    })
  })
})

```

We are using an `ol.layer.Image` instead of an `ol.layer.Vector`. However, we can still use vector data here through `ol.source.ImageVector` that connects to our original `ol.source.Vector` vector source. The style is provided as config of `ol.source.ImageVector` and not on the layer.

## Bonus Tasks

1. Verify that feature detection still works by registering a `'singleclick'` listener on your map that calls `forEachFeatureAtPixel` on the map, and displays earthquake information below the map viewport.

## Controls and interactions

- [Scale line control](#)
- [Select interaction](#)
- [Draw interaction](#)
- [Modify interaction](#)

# Displaying a Scale Line

Another typical widget to display on maps is a scale bar. OpenLayers 3 provides an `ol.control.ScaleLine` for just this.

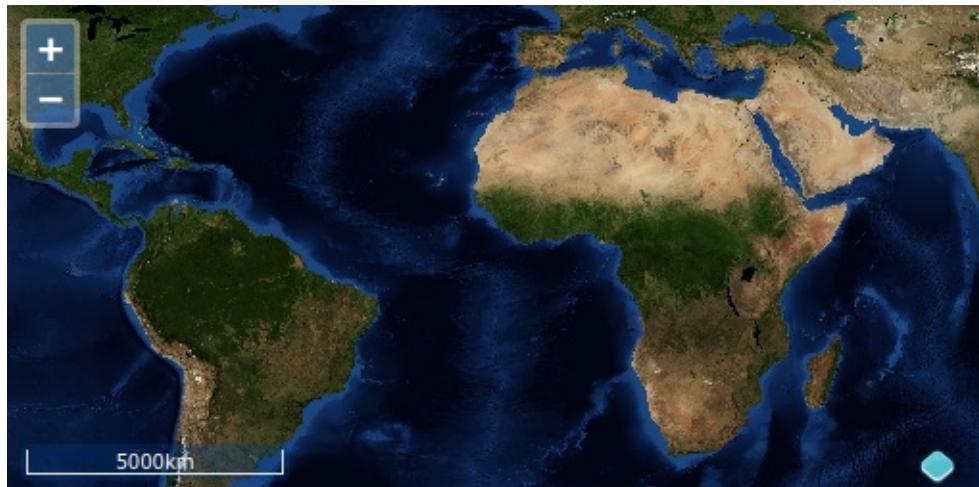
## Creating a ScaleLine Control

### Tasks

1. Open the `map.html` in your text editor.
2. Somewhere in the map config, add the following code to create a new scale line control for your map:

```
controls: ol.control.defaults().extend([
  new ol.control.ScaleLine()
]),
```

3. Save your changes and open `map.html` in your browser: <http://terrestris.github.io/momo3-ws//map.html>



## Moving the ScaleLine Control

You may find the scale bar a bit hard to read over the imagery. There are a few approaches to take in order to improve scale visibility. If you want to keep the control inside the map viewport, you can add some style declarations within the CSS of your document. To test this out, you can include a background color and padding to the scale bar with something like the following:

```
.ol-scale-line {
  background: black;
  padding: 5px;
}
```

However, for the sake of this exercise, let's say you think the map viewport is getting unbearably crowded. To avoid such over-crowding, you can display the scale in a different location. To accomplish this, we need to first create an additional element in our markup and then tell the scale control to render itself within this new element.

### Tasks

1. Create a new block level element in the `<body>` of your page. To make this element easy to refer to, we'll give it an `id` attribute. Insert the following markup somewhere in the `<body>` of your `map.html` page. (Placing the scale element right after the map viewport element `<div id="map"></div>` makes sense.):

```
<div id="scale-line" class="scale-line"></div>
```

2. Now modify the code creating the scale control so that it refers to the `scale-line` element:

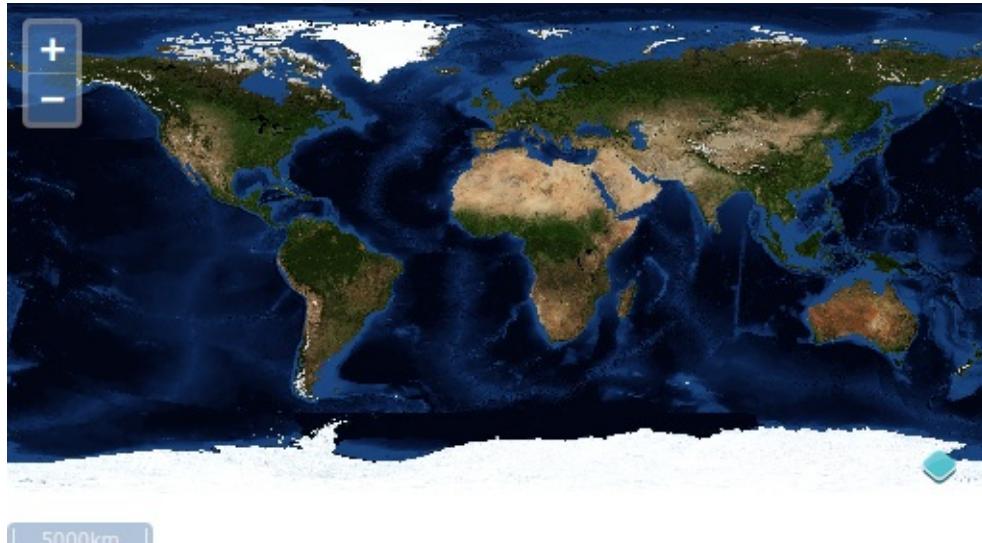
```
controls: ol.control.defaults().extend([
  new ol.control.ScaleLine({className: 'ol-scale-line', target: document.getElementById('scale-line')})
]),
```

3. Save your changes and open `map.html` in your browser: <http://terrestris.github.io/momo3-ws//map.html>

4. "Fix" the position of the control with, for example, the following CSS rules:

```
.scale-line {
  position: absolute;
  top: 350px;
}
.ol-scale-line {
  position: relative;
  bottom: 0px;
  left: 0px;
}
```

5. Now save your changes and view `map.html` again in your browser: <http://terrestris.github.io/momo3-ws//map.html>



*Note* - To create a custom control you can inherit (by using `ol.inherits`) from `ol.control.Control`. To see an example of this check out: <http://openlayers.org/en/master/examples/custom-controls.html>.

## Selecting Features

As we've seen in the layers module, we can pull features as vectors and draw them on top of a base map. One of the advantages of serving vector data is that users can interact with the data. In this example, we create a vector layer where users can select and view feature information.

The previous example demonstrated the use of an `ol.control.Control` on the map. Controls have a visual representation on the map or add DOM elements to the document. An `ol.interaction.Interaction` is responsible for handling user interaction, but typically has no visual representation. This example demonstrates the use of the `ol.interaction.Select` for interacting with features from vector layers.

## Create a Vector Layer and a Select Interaction

### Tasks

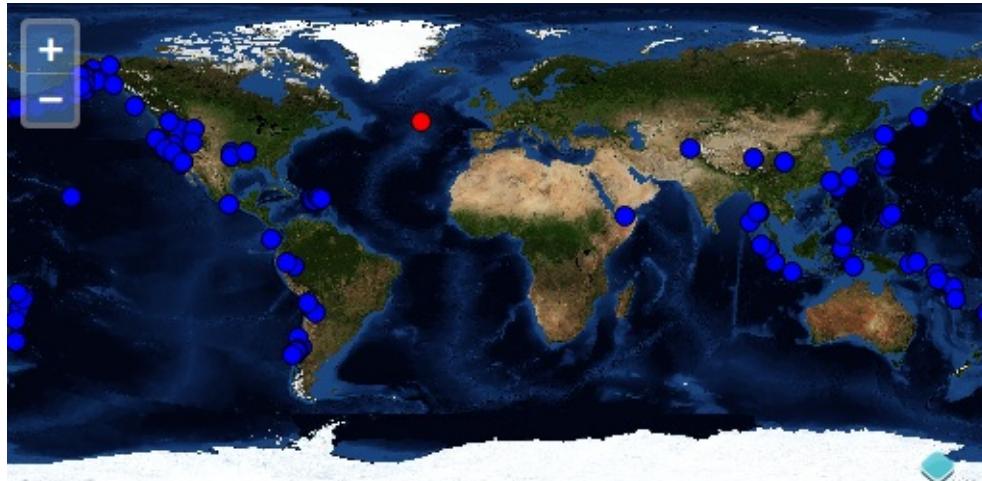
1. Let's start with the vector layer example from a [previous section](#). Open `map.html` in your text editor and make sure it looks something like the following:

```

<!doctype html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/ol.css" type="text/css">
  <style>
    #map {
      height: 256px;
      width: 512px;
    }
  </style>
  <script src="/loader.js" type="text/javascript"></script>
  <title>OpenLayers 3 example</title>
</head>
<body>
  <h1>My Map</h1>
  <div id="map"></div>
  <script type="text/javascript">
    var map = new ol.Map({
      interactions: ol.interaction.defaults().extend([
        new ol.interaction.Select({
          style: new ol.style.Style({
            image: new ol.style.Circle({
              radius: 5,
              fill: new ol.style.Fill({
                color: '#FF0000'
              }),
              stroke: new ol.style.Stroke({
                color: '#000000'
              })
            })
          })
        })
      ]),
      target: 'map',
      layers: [
        new ol.layer.Tile({
          title: 'Global Imagery',
          source: new ol.source.TileWMS({
            url: 'http://demo.opengeo.org/geoserver/wms',
            params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
          })
        }),
        new ol.layer.Vector({
          title: 'Earthquakes',
          source: new ol.source.Vector({
            url: '/data/layers/7day-M2.5.json',
            format: new ol.format.GeoJSON()
          }),
          style: new ol.style.Style({
            image: new ol.style.Circle({
              radius: 5,
              fill: new ol.style.Fill({
                color: '#0000FF'
              }),
              stroke: new ol.style.Stroke({
                color: '#000000'
              })
            })
          })
        })
      ],
      view: new ol.View({
        projection: 'EPSG:4326',
        center: [0, 0],
        zoom: 1
      })
    });
  </script>
</body>
</html>

```

2. Save your changes to `map.html` and open the page in your browser: <http://terrestris.github.io/momo3-ws//map.html>. To see feature selection in action, use the mouse-click to select an earthquake:



## Drawing Features

New features can be drawn by using an `ol.interaction.Draw`. A draw interaction is constructed with a vector source and a geometry type.

## Create a Vector Layer and a Draw Interaction

### Tasks

1. Let's start with the example below. Open `map.html` in your text editor and make sure it looks something like the following:

```

<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var source = new ol.source.Vector({
        url: '/data/layers/7day-M2.5.json',
        format: new ol.format.GeoJSON()
      });
      var draw = new ol.interaction.Draw({
        source: source,
        type: 'Point'
      });
      var map = new ol.Map({
        interactions: ol.interaction.defaults().extend([draw]),
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          }),
          new ol.layer.Vector({
            title: 'Earthquakes',
            source: source,
            style: new ol.style.Style({
              image: new ol.style.Circle({
                radius: 5,
                fill: new ol.style.Fill({
                  color: '#0000FF'
                }),
                stroke: new ol.style.Stroke({
                  color: '#000000'
                })
              })
            })
          ],
          view: new ol.View({
            projection: 'EPSG:4326',
            center: [0, 0],
            zoom: 1
          })
        });
      </script>
    </body>
  </html>

```

2. Save your changes to `map.html` and open the page in your browser: <http://terrestris.github.io/momo3-ws//map.html>. To see drawing of point geometries in action, click in the map to add a new feature:



## Bonus Tasks

1. Create a listener which gets the new feature's X and Y after it is drawn.

# Modifying Features

Modifying features works by using an `ol.interaction.Select` in combination with an `ol.interaction.Modify`. They share a common collection (`ol.Collection`) of features. Features selected with the `ol.interaction.Select` become candidates for modifications with the `ol.interaction.Modify`.

## Create a Vector Layer and a Modify Interaction

### Tasks

1. Let's start with the working example. Open `map.html` in your text editor and make sure it looks something like the following:

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="/ol.css" type="text/css">
    <style>
      #map {
        height: 256px;
        width: 512px;
      }
    </style>
    <script src="/loader.js" type="text/javascript"></script>
    <title>OpenLayers 3 example</title>
  </head>
  <body>
    <h1>My Map</h1>
    <div id="map"></div>
    <script type="text/javascript">
      var source = new ol.source.Vector({
        url: '/data/layers/7day-M2.5.json',
        format: new ol.format.GeoJSON()
      });
      var style = new ol.style.Style({
        image: new ol.style.Circle({
          radius: 7,
          fill: new ol.style.Fill({
            color: [0, 153, 255, 1]
          }),
          stroke: new ol.style.Stroke({
            color: [255, 255, 255, 0.75],
            width: 1.5
          })
        }),
        zIndex: 100000
      });
      var select = new ol.interaction.Select({style: style});
      var modify = new ol.interaction.Modify({
        features: select.getFeatures()
      });
      var map = new ol.Map({
        interactions: ol.interaction.defaults().extend([select, modify]),
        target: 'map',
        layers: [
          new ol.layer.Tile({
            title: 'Global Imagery',
            source: new ol.source.TileWMS({
              url: 'http://demo.opengeo.org/geoserver/wms',
              params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
            })
          }),
          new ol.layer.Vector({
            title: 'Earthquakes',
            source: source,
            style: new ol.style.Style({

```

```

        image: new ol.style.Circle({
          radius: 5,
          fill: new ol.style.Fill({
            color: '#0000FF'
          }),
          stroke: new ol.style.Stroke({
            color: '#000000'
          })
        })
      ],
      view: new ol.View({
        projection: 'EPSG:4326',
        center: [0, 0],
        zoom: 1
      })
    );
  </script>
</body>
</html>

```

- Save your changes to `map.html` and open the page in your browser: <http://terrestris.github.io/momo3-ws//map.html>. To see feature modification in action, use the mouse-click to select an earth quake and then drag to move the point.

## A Closer Look

Let's examine how modifying features works.

```

var style = new ol.style.Style({
  image: new ol.style.Circle({
    radius: 7,
    fill: new ol.style.Fill({
      color: [0, 153, 255, 1]
    }),
    stroke: new ol.style.Stroke({
      color: [255, 255, 255, 0.75],
      width: 1.5
    })
  }),
  zIndex: 100000
});
var select = new ol.interaction.Select({style: style});
var modify = new ol.interaction.Modify({
  features: select.getFeatures()
});

```

We create 2 interactions, an `ol.interaction.Select` to select the features before modifying them, and an `ol.interaction.Modify` to actually modify the geometries. They share the same `ol.Collection` of features. Features selected using `ol.interaction.Modify` become candidates for modification with the `ol.interaction.Modify`. As previously, the `ol.interaction.Select` is configured with a style object, which effectively defines the style used for drawing selected features. When the user clicks in the map again, the feature will be drawn using the layer's style.

## Vector Topics

- An aside on formats
- Styling concepts
- Custom feature styles

# Working with Vector Formats

The base `ol.layer.Vector` constructor provides a fairly flexible layer type. By default, when you create a new vector layer, no assumptions are made about where the features for the layer will come from, since this is the domain of `ol.source.Vector`. Before getting into styling vector features, this section introduces the basics of vector formats.

## `ol.format`

The `ol.format` classes in OpenLayers 3 are responsible for parsing data from the server representing vector features. The format turns raw feature data into `ol.Feature` objects.

Consider the two blocks of data below. Both represent the same `ol.Feature` object (a point in Barcelona, Spain). The first is serialized as [GeoJSON](#) (using the `ol.format.GeoJSON` parser). The second is serialized as KML (OGC Keyhole Markup Language) (using the `ol.format.KML` parser).

### GeoJSON Example

```
{
  "type": "Feature",
  "id": "OpenLayers.Feature.Vector_107",
  "properties": {},
  "geometry": {
    "type": "Point",
    "coordinates": [-104.98, 39.76]
  }
}
```

### KML Example

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
  <Placemark>
    <Point>
      <coordinates>-104.98,39.76</coordinates>
    </Point>
  </Placemark>
</kml>
```

# Understanding Style

When styling HTML elements, you might use CSS like the following:

```
.someClass {
  background-color: blue;
  border-width: 1px;
  border-color: olive;
}
```

The `.someClass` text is a selector (in this case it selects all elements that include the class name `'someClass'`) and the block that follows is a group of named properties and values, otherwise known as style declarations.

## Layer style

A vector layer can have styles. More specifically, a vector layer can be configured with an `ol.style.Style` object, an array of `ol.style.Style` objects, or a function that takes an `ol.Feature` instance and a resolution and returns an array of `ol.style.Style` objects.

Here's an example of a vector layer configured with a static style:

```
var layer = new ol.layer.Vector({
  source: new ol.source.Vector(),
  style: new ol.style.Style({
    // ...
  })
});
```

And here's an example of a vector layer configured with a style function that applies a style to all features that have an attribute named `class` with a value of `'someClass'`:

```
var layer = new ol.layer.Vector({
  source: new ol.source.Vector(),
  style: function(feature, resolution) {
    if (feature.get('class') === 'someClass') {
      // create styles...
      return styles;
    }
  },
});
```

## Symbolizers

The equivalent of a declaration block in CSS is a `symbolizer` in OpenLayers 3 (these are typically instances of `ol.style` classes). To paint polygon features with a blue background and a 1 pixel wide olive stroke, you would use two symbolizers like the following:

```
new ol.style.Style({
  fill: new ol.style.Fill({
    color: 'blue'
  }),
  stroke: new ol.style.Stroke({
    color: 'olive',
    width: 1
  })
});
```

Depending on the geometry type, different symbolizers can be applied. Lines work like polygons, but they cannot have a fill. Points can be styled with `ol.style.Circle` or `ol.style.Icon`. The former is used to render circle shapes, and the latter uses graphics from file (e.g. png images). Here is an example for a style with a circle:

```
new ol.style.Circle({
  radius: 20,
  fill: new ol.style.Fill({
    color: '#ff9900',
    opacity: 0.6
  }),
  stroke: new ol.style.Stroke({
    color: '#ffcc00',
    opacity: 0.4
  })
});
```

## ol.style.Style

An `ol.style.Style` object has 4 keys: `fill`, `image`, `stroke` and `text`. It also has an optional `zIndex` property. The `style` function will return an array of `ol.style.Style` objects.

If you want all features to be colored red except for those that have a `class` attribute with the value of `"someClass"` (and you want those features colored blue with an 1px wide olive stroke), you would create a style function that looked like the following (by the way, it is important to create objects outside of the style function so they can be reused, but for simplicity reasons the objects are created inline in the example below):

```
var primaryStyles = [
  new ol.style.Style({
    fill: new ol.style.Fill({
      color: 'blue'
    }),
    stroke: new ol.style.Stroke({
      color: 'olive',
      width: 1
    })
  })
];
var otherStyle = [new ol.style.Style({
  fill: new ol.style.Fill({
    color: 'red'
  })
});
];
var otherStyles = [
  // define other styles here
]

layer.setStyle(function(feature, resolution) {
  if (feature.get('class') === 'someClass') {
    return primaryStyles;
  } else {
    return otherStyles;
  }
});
```

*Note* - It is important to create the style arrays outside of the actual style function. The style function is called many times during rendering, and you'll get smoother animation if your style functions don't create a lot of garbage.

A feature also has a style config option that can take a function having only resolution as argument. This makes it possible to style individual features (based on resolution).

## Pseudo-classes

CSS allows for pseudo-classes on selectors. These basically limit the application of style declarations based on contexts that are not easily represented in the selector, such as mouse position, neighboring elements, or browser history. In OpenLayers 3, a somewhat similar concept is having a style config option on an `ol.interaction.Select`.

An example is:

```
var select = new ol.interaction.Select({
  style: new ol.style.Style({
    fill: new ol.style.Fill({
      color: 'rgba(255,255,255,0.5)'
    })
  })
});
```

With the basics of styling under your belt, it's time to move on to [styling vector layers](#).

# Styling Vector Layers

1. We'll start with a working example that displays building footprints in a vector layer. Open your text editor and save the following as `map.html` in the root of your workshop directory:

```

<!doctype html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/ol.css" type="text/css">
  <style>
    #map {
      height: 256px;
      width: 512px;
    }
  </style>
  <title>OpenLayers 3 example</title>
  <script src="/loader.js" type="text/javascript"></script>
</head>
<body>
  <h1>My Map</h1>
  <div id="map"></div>
  <script type="text/javascript">
    var map = new ol.Map({
      target: 'map',
      layers: [
        new ol.layer.Tile({
          source: new ol.source.OSM()
        }),
        new ol.layer.Vector({
          title: 'Buildings',
          source: new ol.source.Vector({
            url: '/data/layers/buildings.kml',
            format: new ol.format.KML({
              extractStyles: false
            })
          }),
          style: new ol.style.Style({
            stroke: new ol.style.Stroke({color: 'red', width: 2})
          })
        })
      ],
      view: new ol.View({
        center: ol.proj.fromLonLat([-122.79264450073244, 42.30975194250527]),
        zoom: 16
      })
    });
  </script>
</body>
</html>

```

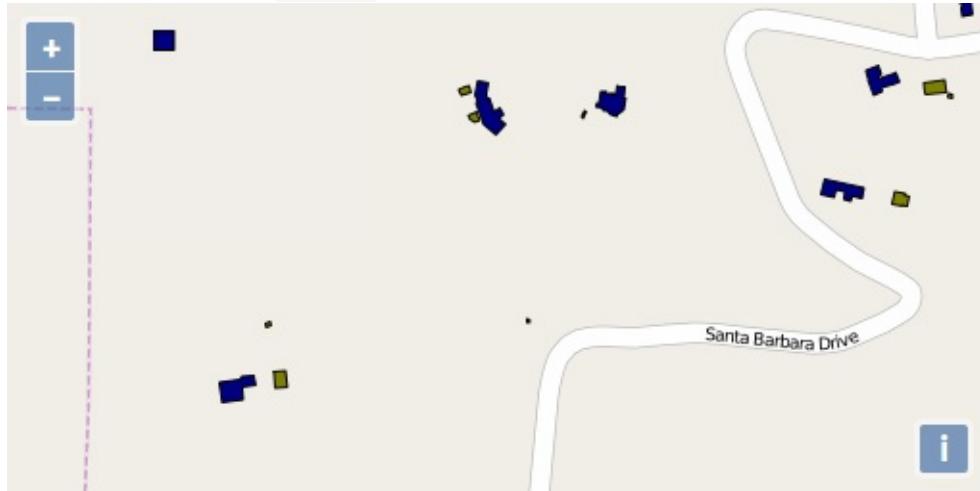
2. Open this `map.html` file in your browser to see buildings with a red outline: <http://terrestris.github.io/momo3-ws/map.html>
3. With a basic understanding of styling in OpenLayers, we can create a style function that displays buildings in different colors based on the size of their footprint. In your map initialization code, add the following two styles arrays and replace the `style` option for the `'Buildings'` layer with the style function below:

```

var defaultStyles = [
  new ol.style.Style({
    fill: new ol.style.Fill({color: 'navy'}),
    stroke: new ol.style.Stroke({color: 'black', width: 1})
  })
];
var smallStyles = [
  new ol.style.Style({
    fill: new ol.style.Fill({color: 'olive'}),
    stroke: new ol.style.Stroke({color: 'black', width: 1})
  })
];
function style(feature, resolution) {
  if (feature.get('shape_area') < 3000) {
    return smallStyles;
  } else {
    return defaultStyles;
  }
}

```

4. Save your changes and open `map.html` in your browser: <http://terrestris.github.io/momo3-ws//map.html>



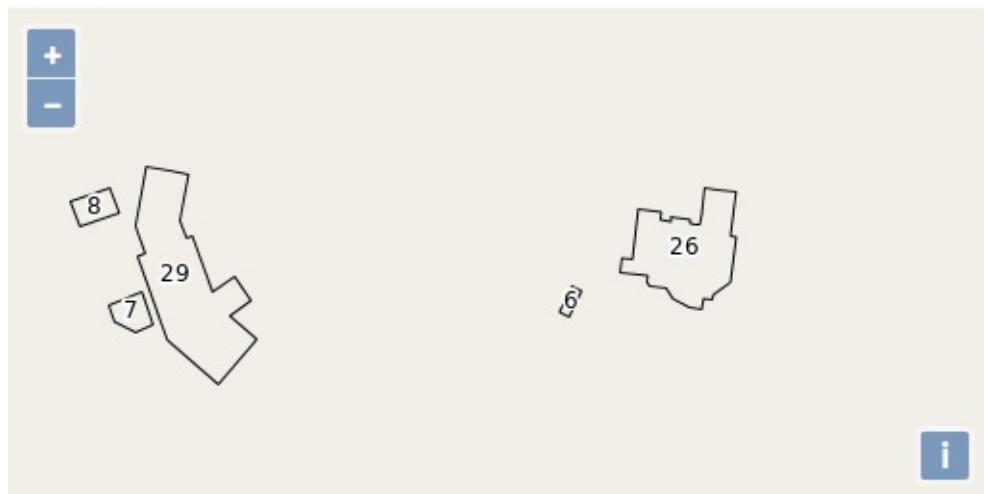
5. Now as a final step, let's add a label to the buildings. For simplicity we're only using a label and a black outline as the style.

```

style: (function() {
  var stroke = new ol.style.Stroke({
    color: 'black'
  });
  var textStroke = new ol.style.Stroke({
    color: '#fff',
    width: 3
  });
  var textFill = new ol.style.Fill({
    color: '#000'
  });
  return function(feature, resolution) {
    return [new ol.style.Style({
      stroke: stroke,
      text: new ol.style.Text({
        font: '12px Calibri,sans-serif',
        text: feature.get('key'),
        fill: textFill,
        stroke: textStroke
      })
    })];
  };
})()

```

6. Save your changes and open `map.html` in your browser: <http://terrestris.github.io/momo3-ws//map.html>



## Custom Builds

- [Concepts](#)
- [Create a custom build](#)

# Understanding custom builds

OpenLayers 3 is a big library providing a lot of functionality. So it is unlikely that an application will need and use all the functionality OpenLayers 3 provides. Custom builds (a.k.a. application-specific builds) are OpenLayers 3 builds with just the functionality your application needs. Custom builds are often much smaller than the full build, so creating custom builds is often a very good idea.

## Requirements

OpenLayers 3 builds are created by using the [Closure Compiler](#). The goal of the Closure Compiler is to compile JavaScript to better JavaScript, that takes less time to download and run faster.

The Closure Compiler is a Java program, so running the Compiler requires a Java Virtual Machine. So before jumping to the next section, and creating a custom build, make sure Java is installed on your machine.

You just need the Java Runtime Environment, which you can download from the [Oracle Java site](#). For example, for Windows, you would download and install `jre-8u60-windows-i586.exe`.

## Build configuration file

Creating a custom build requires writing a build configuration file. The format of build configuration files is JSON. Here is a simple example of a build configuration file:

```
{
  "exports": [
    "ol.Map",
    "ol.View",
    "ol.layer.Tile",
    "ol.source.OSM"
  ],
  "jvm": [],
  "umd": true,
  "compile": {
    "externs": [
      "externs/bingmaps.js",
      "externs/closure-compiler.js",
      "externs/esrijson.js",
      "externs/geojson.js",
      "externs/oli.js",
      "externs/olx.js",
      "externs/proj4js.js",
      "externs/tilejson.js",
      "externs/topojson.js"
    ],
    "define": [
      "goog.array.ASSUME_NATIVE_FUNCTIONS=true",
      "goog.dom.ASSUME_STANDARDS_MODE=true",
      "goog.json.USE_NATIVE_JSON=true"
    ],
    "jscomp_error": [
      "*"
    ],
    "jscomp_off": [
      "useOfGoogBase",
      "unnecessaryCasts",
      "lintChecks"
    ],
    "extra_annotation_name": [
      "api", "observable"
    ],
    "compilation_level": "ADVANCED",
    "warning_level": "VERBOSE",
    "use_types_for_optimization": true,
    "manage_closure_dependencies": true
  }
}
```

The most relevant part of this configuration object is the `exports` array. This array declares the functions/constructors you use in your JavaScript code. For example, the above configuration file is what you'd use for the following JavaScript code:

```
var map = new ol.Map({
  target: 'map',
  layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM()
    })
  ],
  view: new ol.View({
    center: [0, 0],
    zoom: 4
  })
});
```

## Creating custom builds

In this section we're going to create a custom build for the map you created at the [last chapter](#).

1. Start with the `map.html` file you created previously:

```

<!doctype html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/ol.css" type="text/css">
  <style>
    #map {
      height: 256px;
      width: 512px;
    }
  </style>
  <title>OpenLayers 3 example</title>
  <script src="/loader.js" type="text/javascript"></script>
</head>
<body>
  <h1>My Map</h1>
  <div id="map"></div>
  <script type="text/javascript">
    var style = (function() {
      var stroke = new ol.style.Stroke({
        color: 'black'
      });
      var textStroke = new ol.style.Stroke({
        color: '#fff',
        width: 3
      });
      var textFill = new ol.style.Fill({
        color: '#000'
      });
      return function(feature, resolution) {
        return [new ol.style.Style({
          stroke: stroke,
          text: new ol.style.Text({
            font: '12px Calibri,sans-serif',
            text: feature.get('key'),
            fill: textFill,
            stroke: textStroke
          })
        })];
      };
    })();
    var map = new ol.Map({
      target: 'map',
      layers: [
        new ol.layer.Tile({
          source: new ol.source.OSM()
        }),
        new ol.layer.Vector({
          title: 'Buildings',
          source: new ol.source.Vector({
            url: '/data/layers/buildings.kml',
            format: new ol.format.KML({
              extractStyles: false
            })
          }),
          style: style
        })
      ],
      view: new ol.View({
        center: ol.proj.fromLonLat([-122.79264450073244, 42.30975194250527]),
        zoom: 16
      })
    );
  </script>
</body>
</html>

```

2. Create a build configuration file for that map:

```
{
  "exports": [
    "ol.Map",
    "ol.View",
    "ol.format.KML",
    "ol.layer.Tile",
    "ol.layer.Vector",
    "ol.proj.fromLonLat",
    "ol.source.OSM",
    "ol.source.Vector",
    "ol.style.Fill",
    "ol.style.Stroke",
    "ol.style.Style",
    "ol.style.Text"
  ],
  "jvm": [],
  "umd": true,
  "compile": {
    "externs": [
      "externs/bingmaps.js",
      "externs/closure-compiler.js",
      "externs/esrijson.js",
      "externs/geojson.js",
      "externs/oli.js",
      "externs/olx.js",
      "externs/proj4js.js",
      "externs/tilejson.js",
      "externs/topojson.js"
    ],
    "define": [
      "goog.array.ASSUME_NATIVE_FUNCTIONS=true",
      "goog.dom.ASSUME_STANDARDS_MODE=true",
      "goog.json.USE_NATIVE_JSON=true",
      "ol.ENABLE_DOM=false",
      "ol.ENABLE_WEBGL=false",
      "ol.ENABLE_PROJ4JS=false",
      "ol.ENABLE_IMAGE=false",
      "goog.DEBUG=false"
    ],
    "jscomp_error": [
      "*"
    ],
    "jscomp_off": [
      "useOfGoogBase",
      "unnecessaryCasts",
      "lintChecks"
    ],
    "extra_annotation_name": [
      "api", "observable"
    ],
    "compilation_level": "ADVANCED",
    "warning_level": "VERBOSE",
    "use_types_for_optimization": true,
    "manage_closure_dependencies": true
  }
}
```

3. Create the custom build using `OpenLayers`'s `build.js` Node script:

```
$ node node_modules/openlayers/tasks/build.js ol-custom.json ol-custom.js
```

This will generate the `ol-custom.js` custom build at the root of the project.

4. Now change `map.html` to use the custom build (`ol-custom.js`) rather than the development loader.

So change

```
<script src="/loader.js" type="text/javascript"></script>
```

to

```
<script src="/ol-custom.js" type="text/javascript"></script>
```

The page should now load much faster than before!

# ExtJS

- Introduction
- Hello ExtJS
  - include resources
  - Create a window in Ext.onReady()
  - Where to find docs...
- Layouts
  - Start with 1 Panel in a viewport (layout default: 'auto')
    - explain Viewport and 3 configs of Panel
  - Add more panels to the viewport (still auto layout)
    - show that the result isn't what we really wanted
  - go to the layout browser example and pick two which you Linked
  - In the end we need to have the 'border-layout' (with panels in north, center, west, east, south)
- Components
  - start with the final example from Layouts
    - tabpanel in the center
  - Use img component in header
  - use basic form in east
  - use basic grid (local data) in south
  - use basic tree in west
- Next add behaviour
  - events, remote data / AJAX, filtering???

## **GeoExt**

# Glossary

## term

Definition for this term

[3.1.1. Basics](#)    [3.1.1. Basics](#)