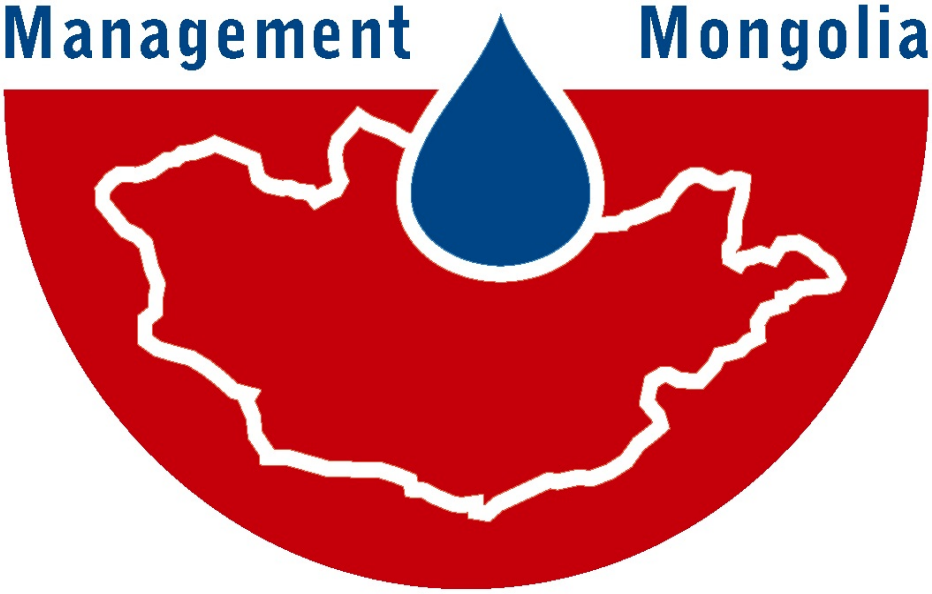
MoMo

Integrated Water Resources Management Mongolia



MoMo3 Workshop

2016-02-22 – 2016-02-26

Table of Contents

Welcome to the workshop

Introduction to core technologies behind the MoMo geoportal

Ulan Bator, Mongolia

A 5-day workshop from 2016-02-22 to 2016-02-26

Sources

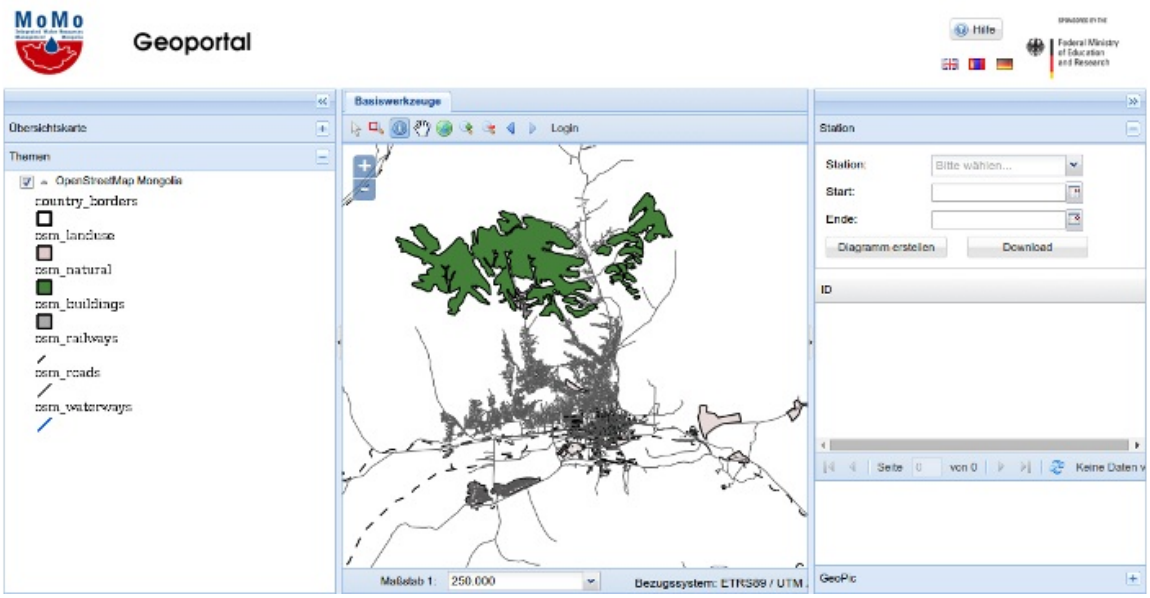
- Workshop URL
- Download workshop (ZIP)
- Download workshop (PDF)
- Download workshop (EPUB)
- Git repository on github

Authors

- Marc Jansen (jansen@terrestris.de)
- Daniel Koch (koch@terrestris.de)

Overview

This workshop is intended to introduce certain key software packages behind the current and upcoming MoMo GeoPortal.



The current version of the soon-to-be-updated MoMo Geoportal

In this overview section we'll learn about the following topics

- Prerequisites
- References
- Setup
- Data

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:

> In case you didn't receive a bootable USB with the above system, you can grab an image under the following URL: http://files.terrestris.de/momo-master.tar.gz. This file is roughly 2.2 GB big, and once it is decompressed it will have a size of 16GB.

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.

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`
  - Password: `momo`
- GeoServer:
  - User: `admin`
  - Password: `geoserver`
- PostgreSQL:
  - User: `momo`
  - Password: `momo`

Useful paths

- Your home directory: `/home/momo`
- Workshop directory: `/home/momo/materials`
- Tomcat webapp directory: `/opt/tomcat/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 {ARCHIVE_FILE_NAME}.tar.gz {DIRECTORY_TO_ARCHIVE}`
- Extract an archive: `$ tar -xvzf {ARCHIVE_FILE_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 start|stop|restart tomcat`

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
  - 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 packges to latest version

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

- In case you didn't receive a bootable USB with the above system, you can grab an image under the following URL: http://files.terrestris.de/momo-master.tar.gz This file is roughly 2.2 GB big, and once it is decompressed it will have a size of 16GB.
- 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 us 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)

> 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

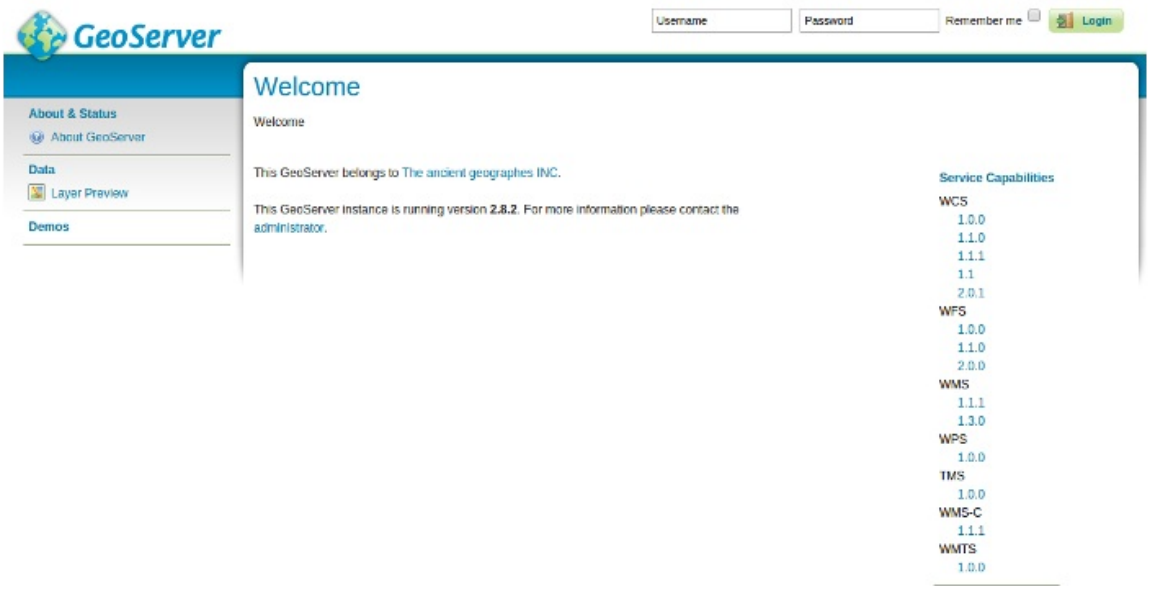
GeoServer includes a web-based administration interface. Most GeoServer configuration can be done through this interface, without the need to edit configuration files by hand or use an API. This section will give a brief overview to the web interface. Subsequent sections will use the web interface in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



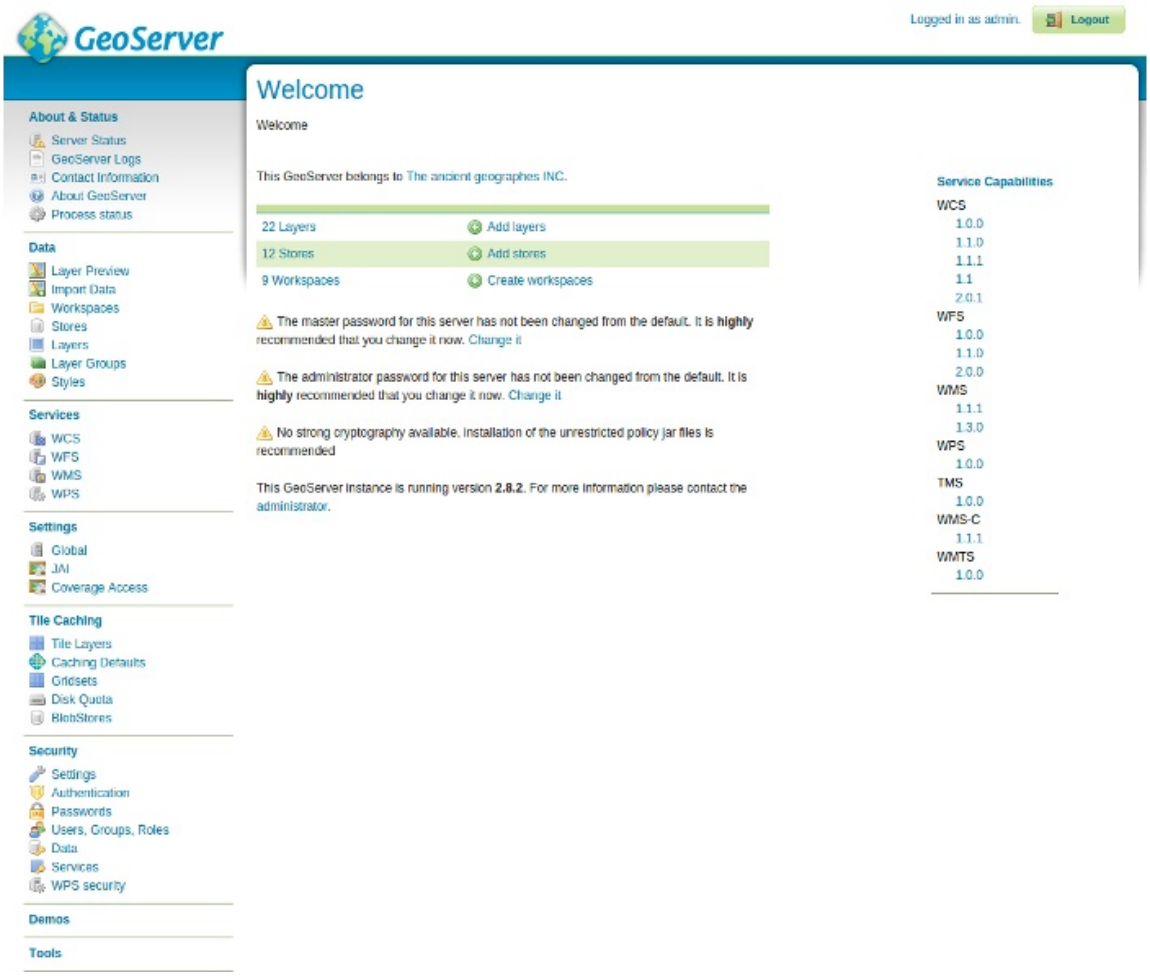
GeoServer welcome page

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.



GeoServer welcome page

After the login, 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.

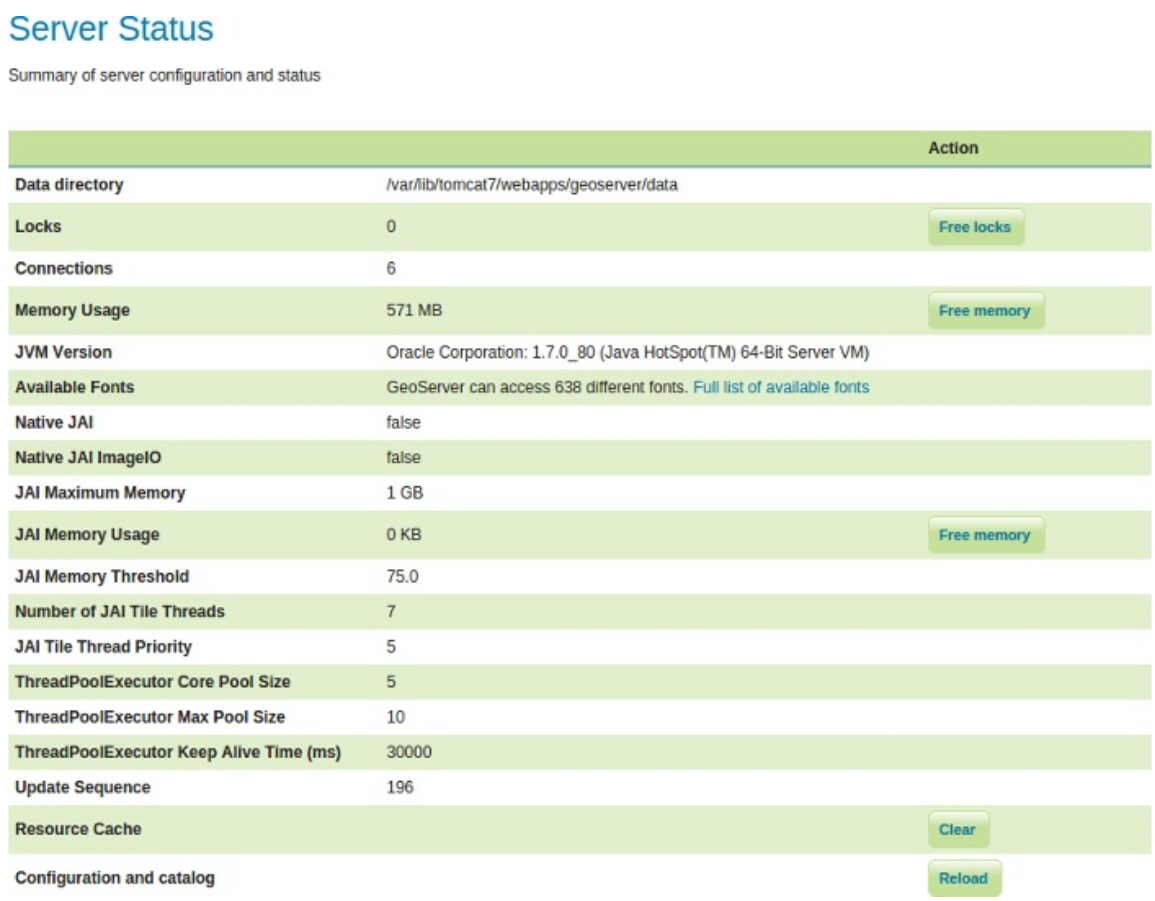
About and Status

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	Free locks
Connections	6	
Memory Usage	571 MB	Free memory
JVM Version	Oracle Corporation: 1.7.0_80 (Java HotSpot(TM) 64-Bit Server VM)	
Available Fonts	GeoServer can access 638 different fonts. Full list of available fonts	
Native JAI	false	
Native JAI ImageIO	false	
JAI Maximum Memory	1 GB	
JAI Memory Usage	0 KB	Free memory
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		Clear
Configuration and catalog		Reload

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.

> 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).

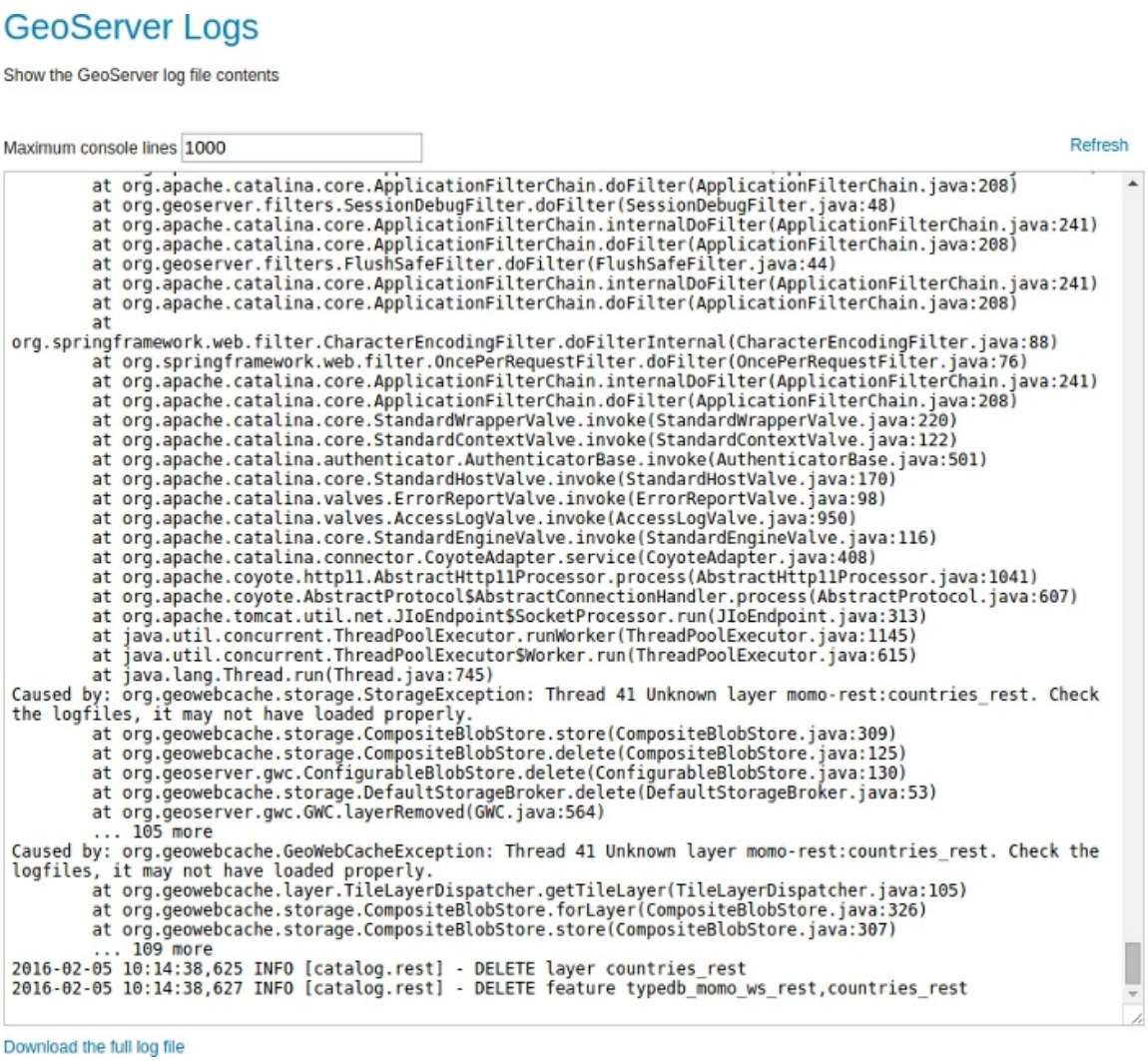
Option	Description
Data directory	The absolute path to your data directory.
Locks	A WFS has the ability to lock features to prevent more than one person from updating the feature at one time. If data is locked, edits can be performed by a single WFS editor. When the edits are posted, the locks are released 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 `Free locks` 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 `Free Memory` 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 `Free Memory` 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 `Clear` 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 `${GEOSERVER_DATA_DIR}/user_projections/epsg.properties` .
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 `Reload` button will force GeoServer to reload all of its configuration from disk.

Exercise

- 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`.



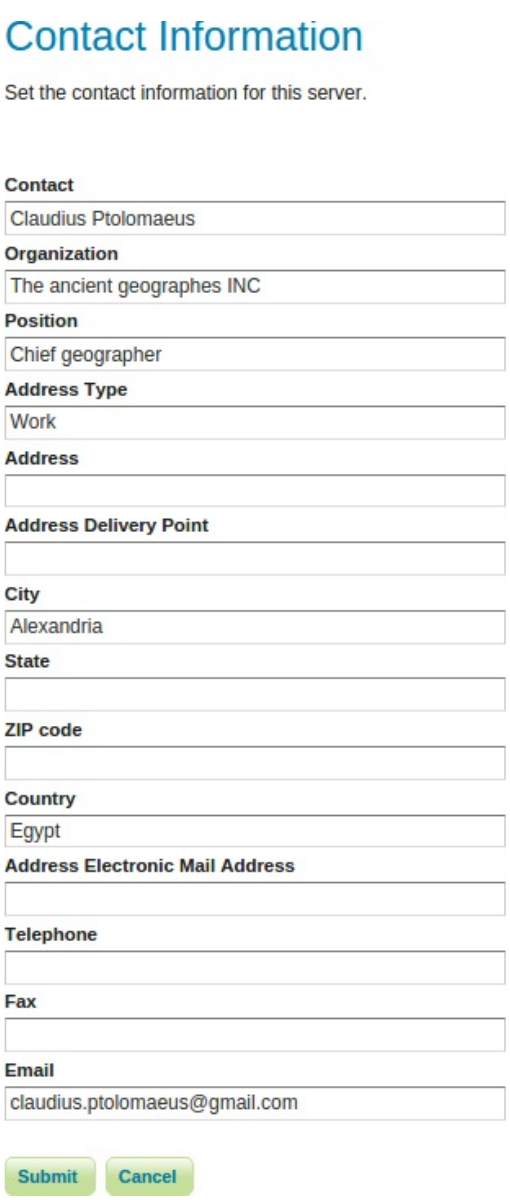
Logging page.

Exercise

- 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 page.

Exercise

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

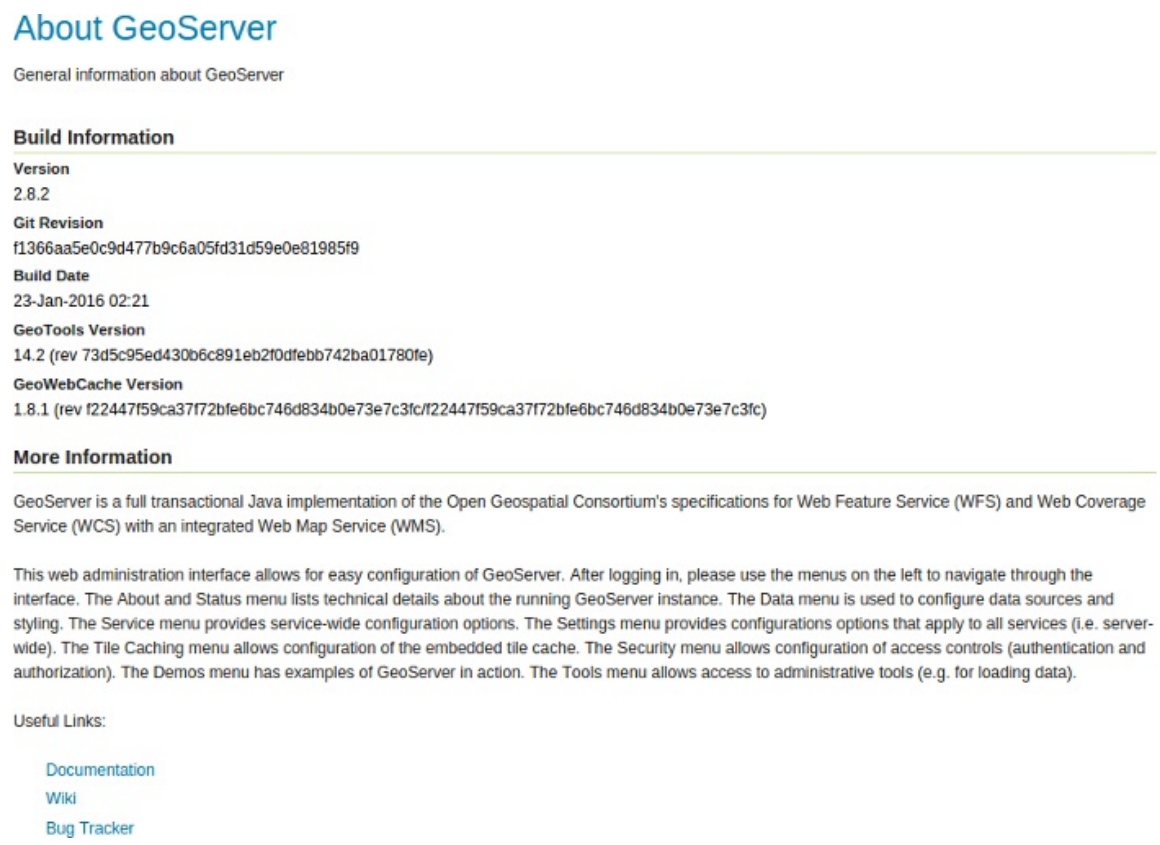
  http://localhost/geoserver/ows?SERVICE=WMS&REQUEST=GetCapabilities&VERSION=1.1.0

> 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 espacially 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 page.

Exercise

- 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

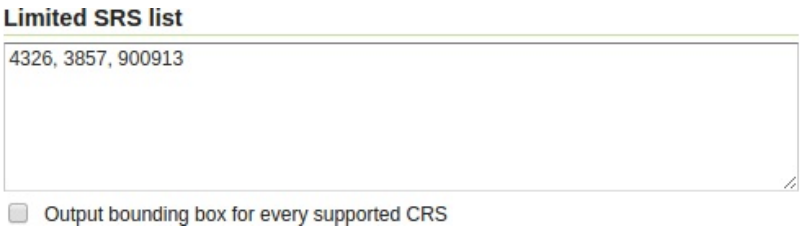
The default GetCapabilities document contains a comprehensive list of all available spatial reference systems (SRS) of your GeoServer WMS server instance. Generally it is not needed, that you list all supported systems as you typically want to publish data in an limited list of projections only. Furthermore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

Exercise

- Open the following URL in your browser to open the GetCapabilites document of your GeoServer instance: GetCapabilities
- 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.

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



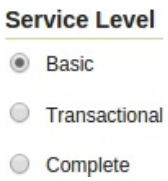
- Click `Submit`.
- Reopen the GetCapabilites document and you will note, that the document is reduced by a huge amount of lines.

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.

Exercise

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



- Click `Submit`.
- 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.

Exercise

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

- 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.

Exercise

- Go to `Settings` ❭ `Global`.
- Set `Number of Decimals` to 2.

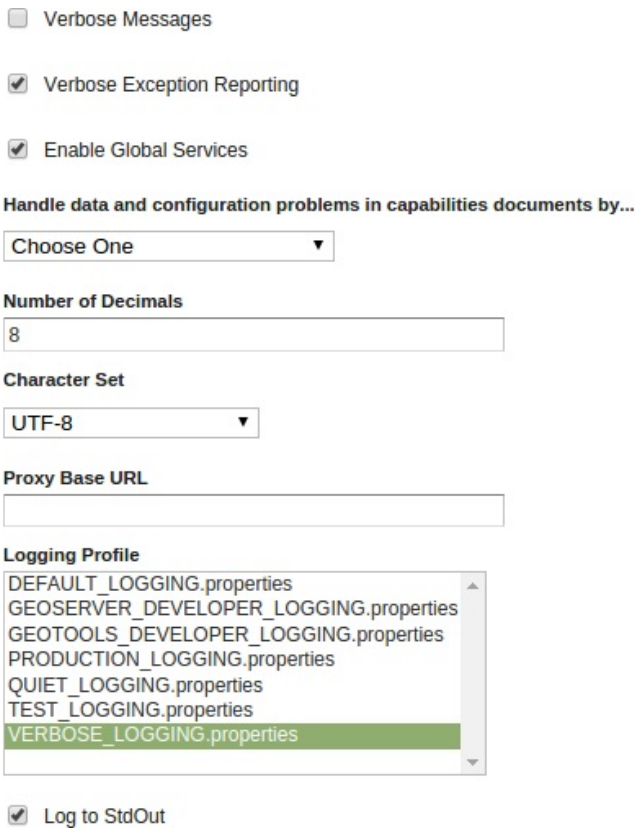
- Click `Submit`.

Change logging level

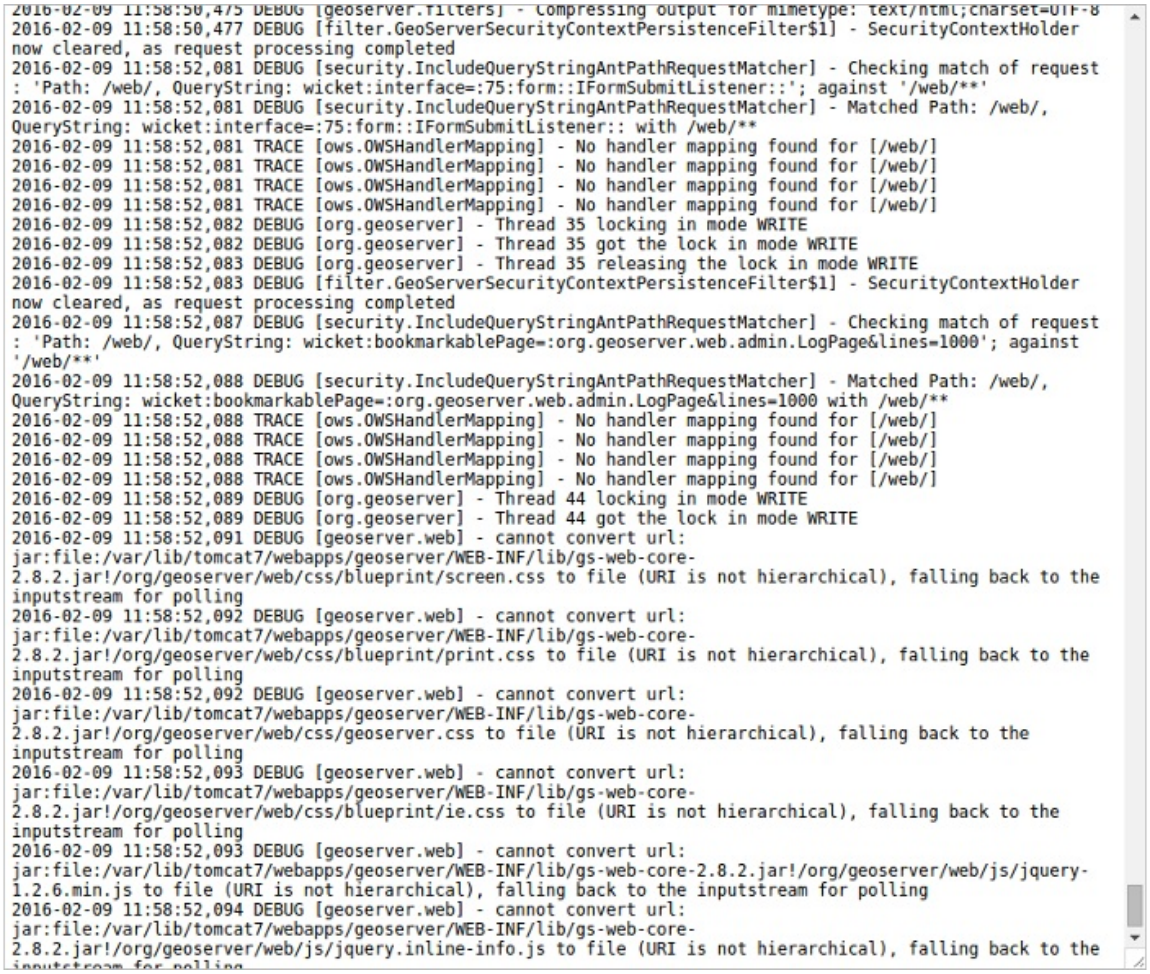
At some point in your production (or development) usage of GeoServer you'll encounter a problem where you'll need to get further and detailed informations to find the cause of the problem. In this case you can increase the level of logging. The following steps will guide you how to set the logging level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

Exercise

- Go to `Settings` ❭ `Global`.
- Select the check box beside `Verbose Exception Reporting` to return service exceptions with full Java stack traces.
- 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.



- Click `Submit`.
- 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.
- 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).



- 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.).

Exercise

To change the master password follow these steps:

- Go to `Security` ❭ `Passwords`.
- Select `Change password`.
- 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!)
- Click `Change Password`.

Change admin user password

Exercise

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

- Go to `Security` ❭ `Users, Groups, Roles`.
- 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.
- Select the user `admin` by clicking on its username.
- 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!)

- 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

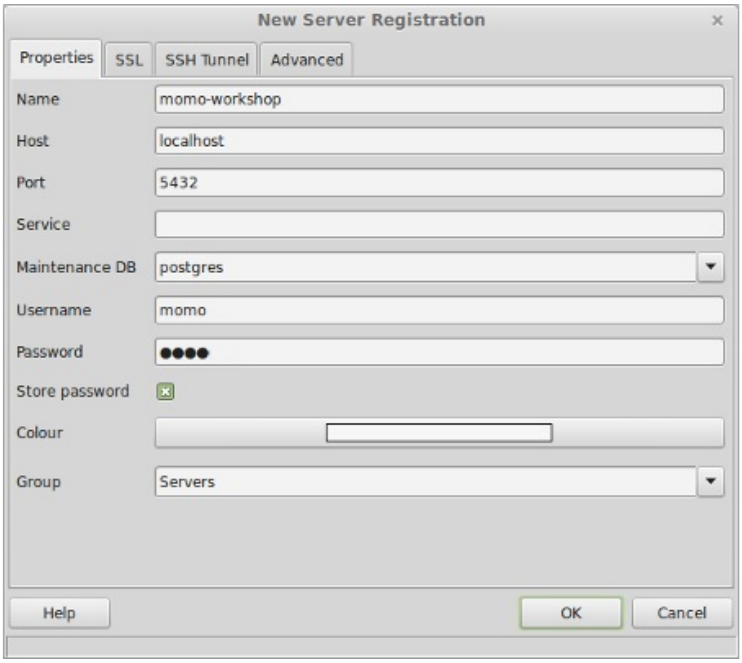
Before we can start importing and reading data in this modules, we have to create a new database on the (already installed) PostgreSQL database server as data source for new layers. Thus we will use the administration tool `pgAdmin III`. Let's start by opening it:

- Click `Menu` in the lower left corner and search for `pgadmin`
- 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:

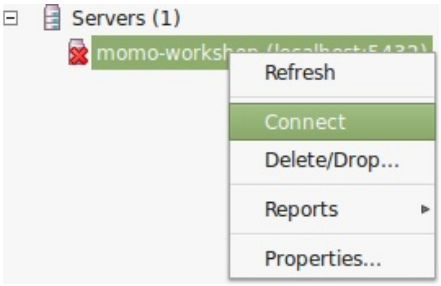
- Create a new server connection by selecting `File` ❭ `Add Server` in the top menu bar and enter the following:
  - *Name:* momo-workshop
  - *Host:* localhost
  - *Port:* 5432
  - *Username:* momo
  - *Password:* momo
  - *Store password:* checked



- 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).



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

- 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.
- Copy the following SQL block into the SQL-Query window:

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

- Click `Execute query` () in the upper toolbar to run the query.
- 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.
- 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.

- 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!
- Copy the following SQL block into the SQL-Query window to create a new schema named `geodata`:

```
CREATE SCHEMA geodata
  AUTHORIZATION momo;
```

- Click `Execute query` () to run the query.
- 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`.

- 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;
```

- Click `Execute query` to run the query.
- 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).

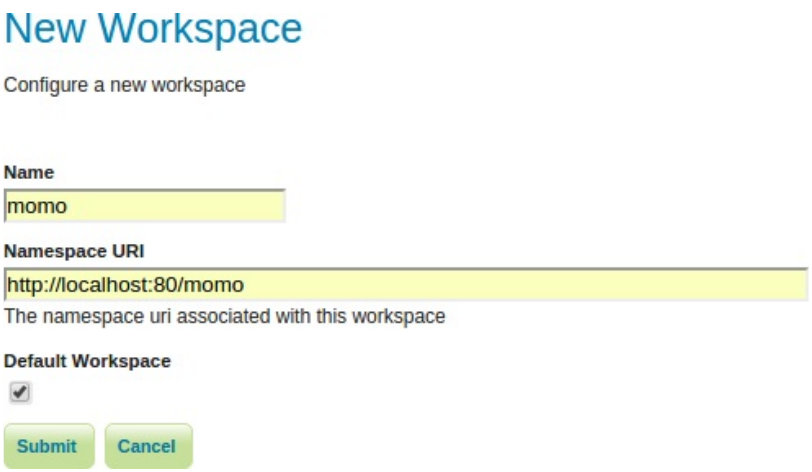
Stores and layers must all have an associated workspace. Styles may optionally be associated with a workspace, but can also be global.

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`.

- Navigate to `Data` ❭ `Workspaces`.
- Click `Add new workspace` and enter the following:
  - *Name:* momo
  - *Namespace URI:* http://localhost:80/momo
  - *Default Workspace:* checked
- Click `Submit`



Add new workspace

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

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

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

```
$ cd ~/materials
```

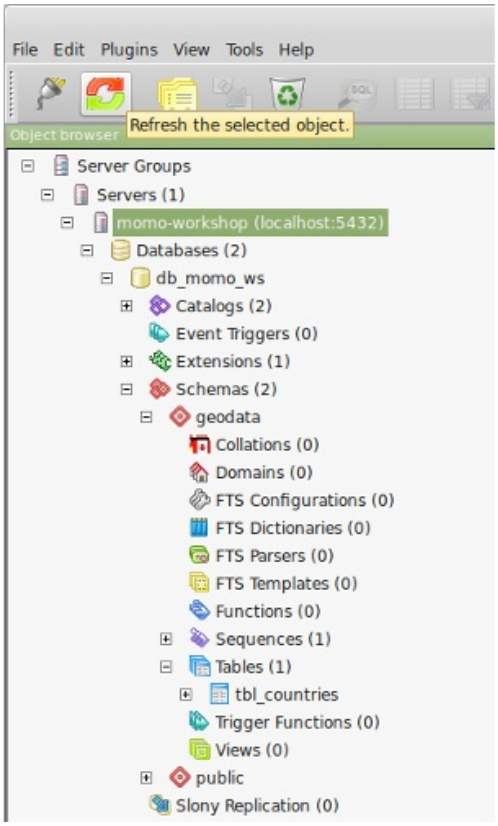
- List all directories in the current folder with:

```
$ ls -l
```

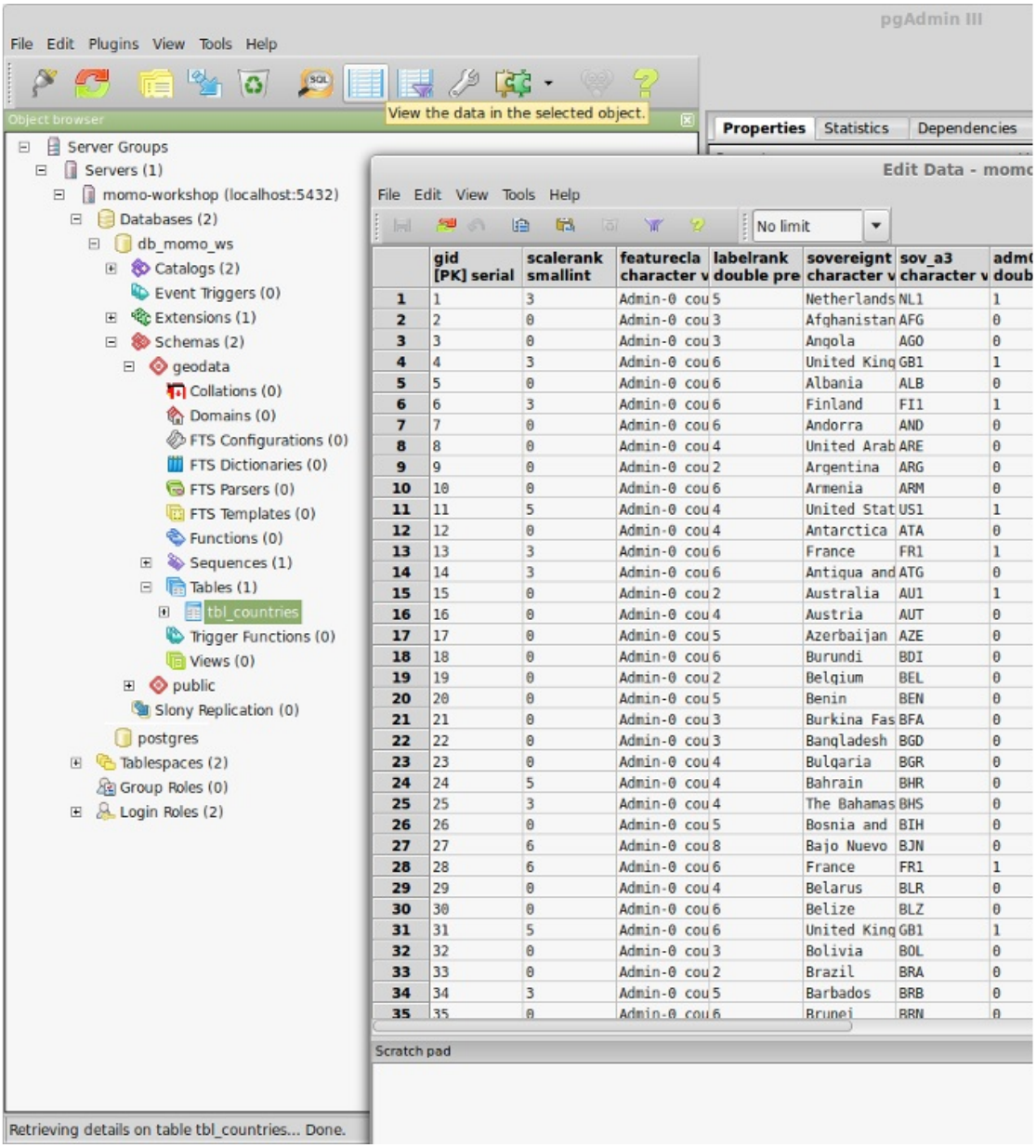
- You should be able to see the folders `natural_earth` and `osm_mongolia`. If you inspect them further on, you will see that each folder contains a wide list of single shapefiles, but for the moment we want to use a small portion of them to import into the database only. Therefore we are going to start with importing a shapefile involving worldwide country polygons.
- Navigate to directory `materials/natural_earth/10m_cultural` and check if you can find a file named `ne_10m_admin_0_countries.shp`.
- 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 \
  -W \
  -d db_momo_ws
```

- 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:



- 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:



- 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.

- Navigate to `Data` ❭ `Stores`.
- Click `Add new Store`.
- Click `PostGIS - PostGIS Database`
- Set the Workspace to `momo` if it isn't set already.
- Configure the new store as follows:
  - *Data Source Name:* db_momo_ws
  - *Enabled:* checked
  - *dbtype:* postgis
  - *host:* localhost
  - *port:* 5432
  - *database:* db_momo_ws
  - *schema:* geodata
  - *user:* momo
  - *passwd:* momo
- Click `Save`

Add new store

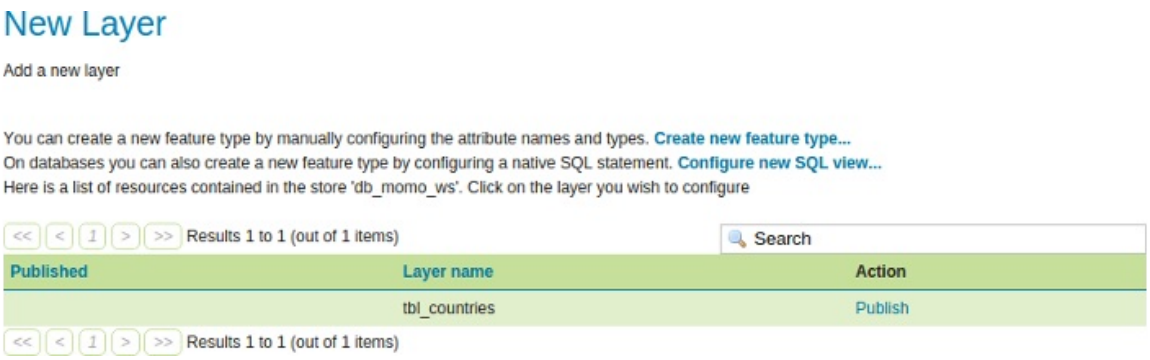
Layer

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`.

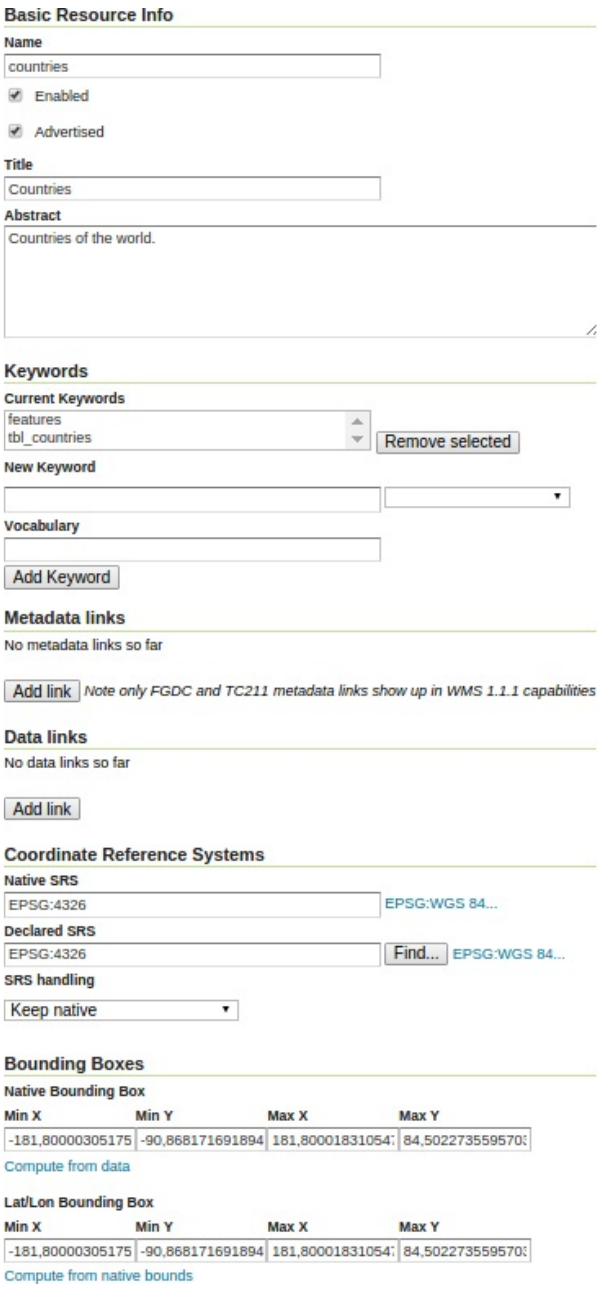


Publish a layer

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.

- 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
- Let GeoServer calculate the bounds of the data by clicking `Compute from data`.
- Convert the native bounds to the Lat/Lon Bounding Box by clicking `Compute from native bounds`.
- Click `Save`.

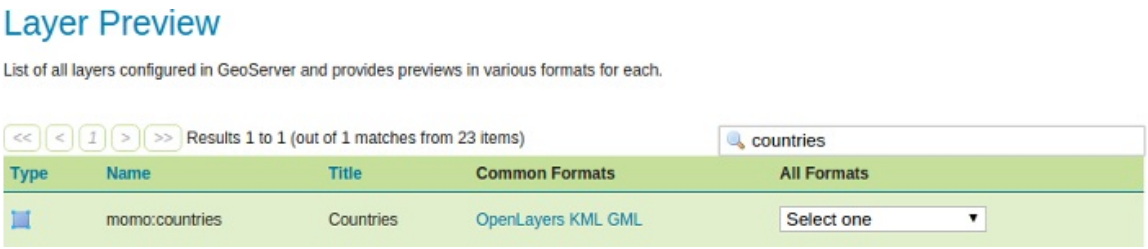


Publish a new layer

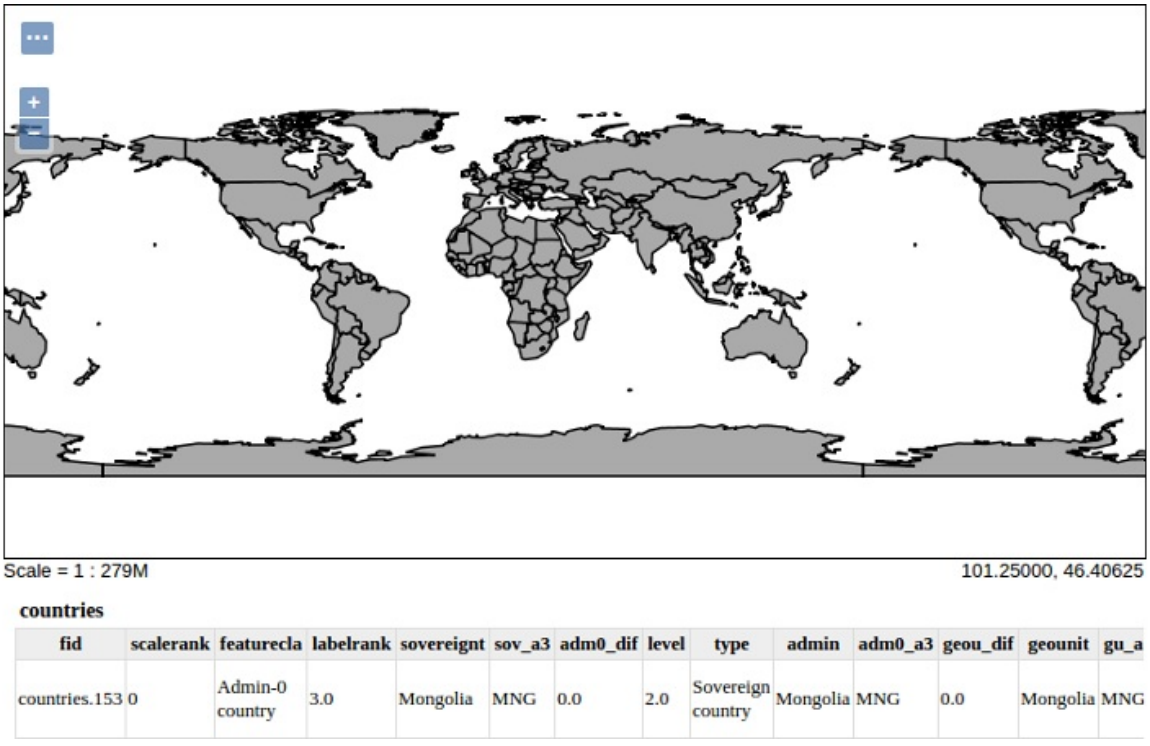
Previewing a layer

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

- Navigate to `Data` ❭ `Layer Preview`
- Search for `countries`.
- Click `OpenLayers`.



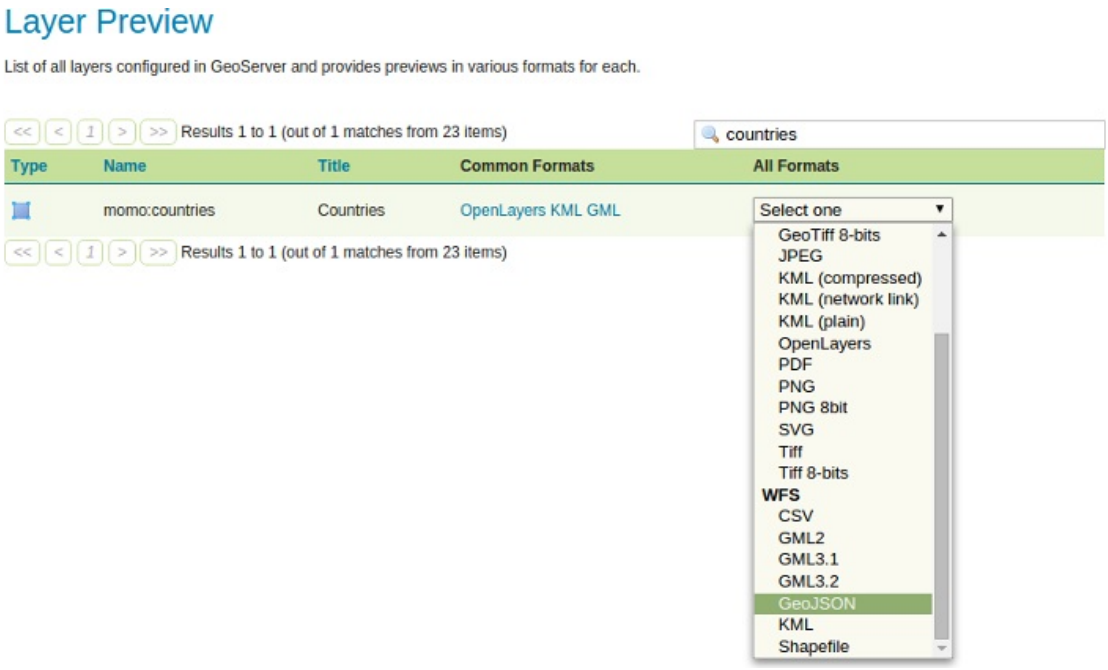
Layer Preview



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.

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



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,
            "sovereignt": "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

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

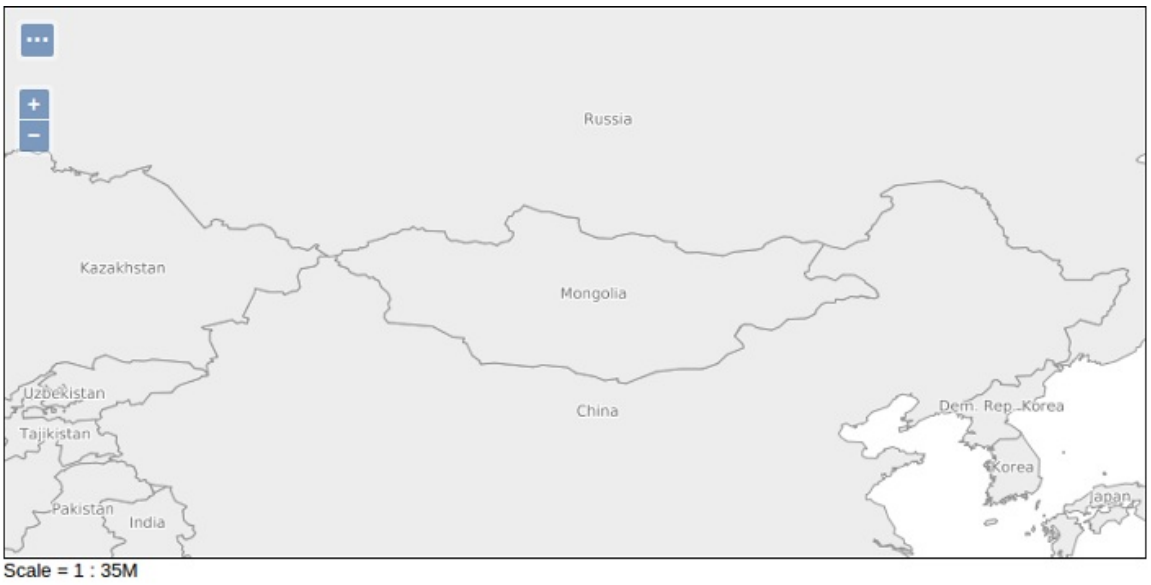
```
<?xml version="1.0" encoding="UTF-8"?>
<sld:StyledLayerDescriptor
xmlns="http://www.opengis.net/sld"
xmlns:sld="http://www.opengis.net/sld"
xmlns:ogc="http://www.opengis.net/ogc"
xmlns:gml="http://www.opengis.net/gml"
version="1.0.0">
<sld:NamedLayer>
  <sld:Name>countries</sld:Name>
  <sld:UserStyle>
      <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:Halo>
                      <sld:Radius>1</sld:Radius>
                      <sld:Fill>
                          <CssParameter name="fill">#FFFFFF</CssParameter>
                      </sld:Fill>
                  </sld:Halo>
                  <sld:Fill>
                      <CssParameter name="fill">#707070</CssParameter>
                  </sld:Fill>
              </sld:TextSymbolizer>
          </sld:Rule>
      </sld:FeatureTypeStyle>
  </sld:UserStyle>
</sld:NamedLayer>
</sld:StyledLayerDescriptor>
```

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

- Go to tab `Publishing`.

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

- Click `Save`.
- 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

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 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).

Out 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.

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

```
$  mkdir OB_LR_pyramid/
```

- 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/ /opt/tomcat/webapps/geoserver/data/data/
```

- Navigate to the data directory:

```
$ cd /opt/tomcat/webapps/geoserver/data/data/
```

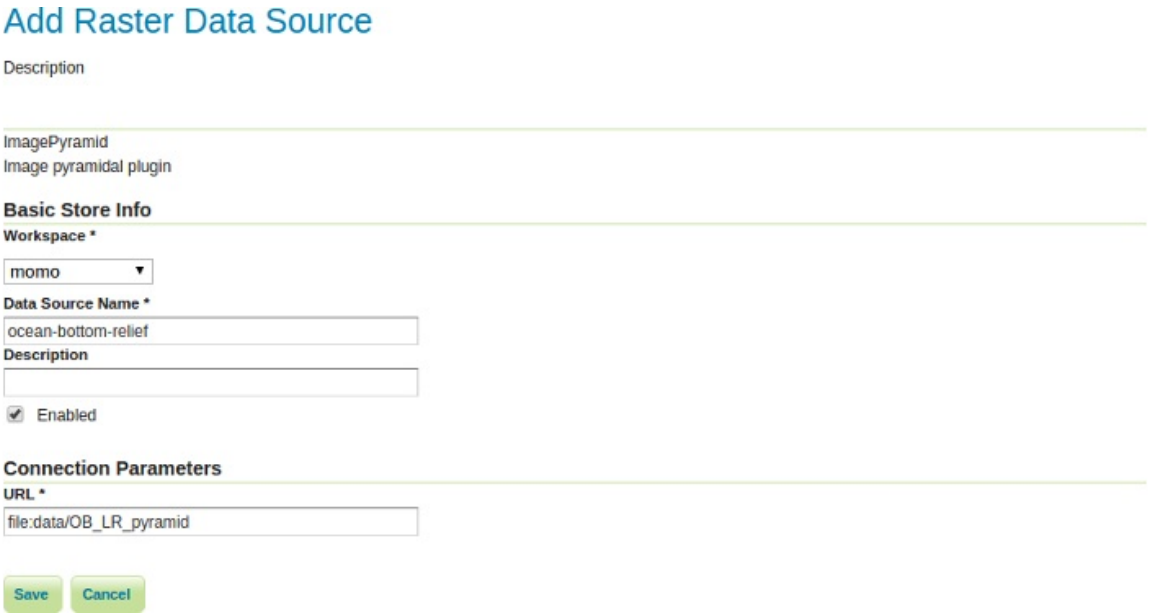
- Assign read and write access to the `tomcat` user:

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

Creating a new store

- Go to `Data` ❭ `Stores` ❭ `Add a new Store`
- Select `ImagePyramid`

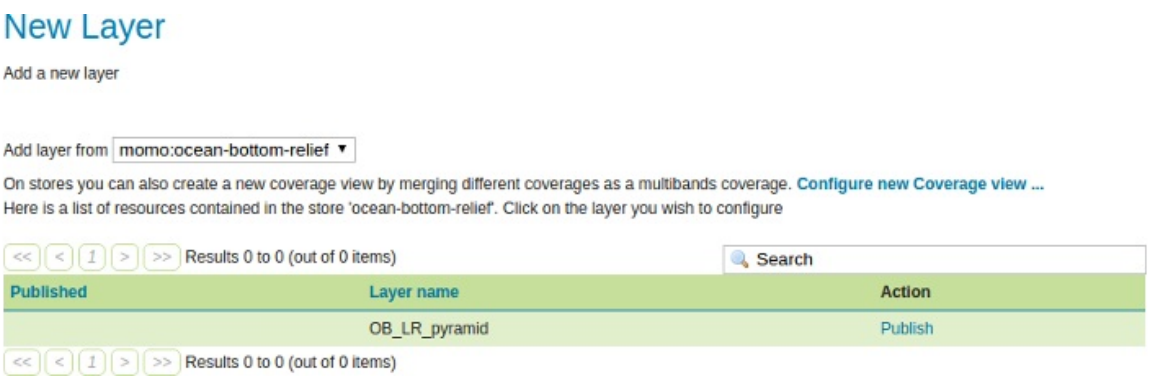
- Create the new store as follows:
  - *Workspace:* momo
  - *Data Source name:* ocean-bottom-relief
  - *Enabled:* checked
  - *URL:* `file:data/OB_LR_pyramid`



- 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`.

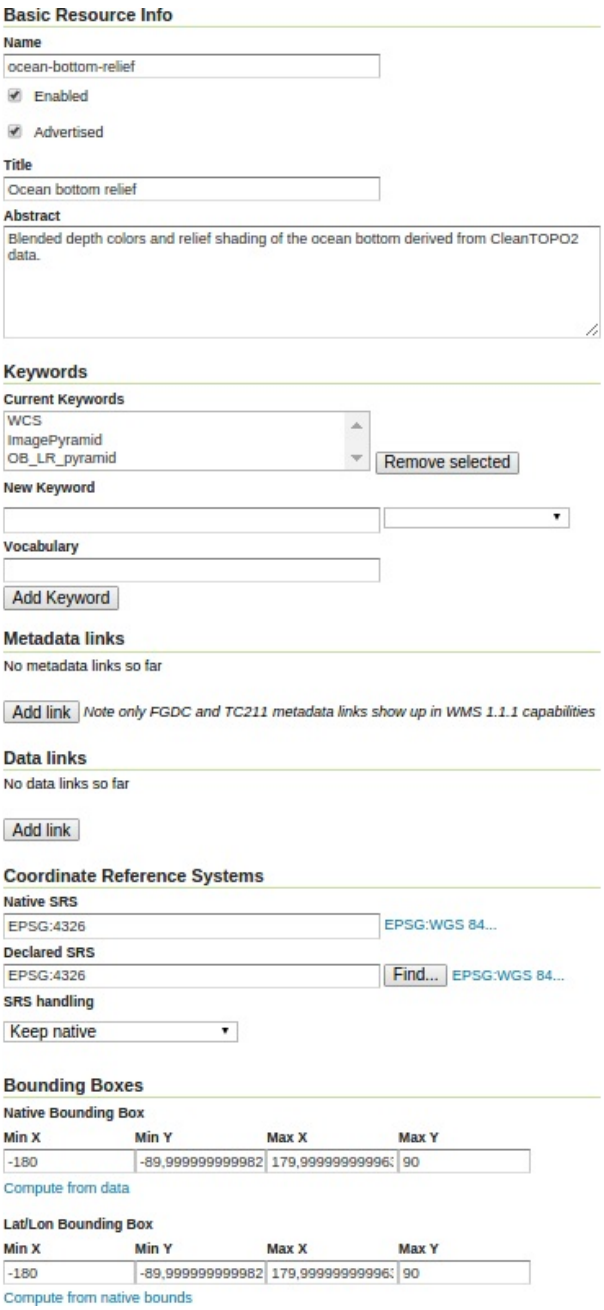


Configure ImagePyramid store

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.

- 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
- Let GeoServer calculate the bounds of the data by clicking `Compute from data`.
- Convert the native bounds to the Lat/Lon Bounding Box by clicking `Compute from native bounds`.

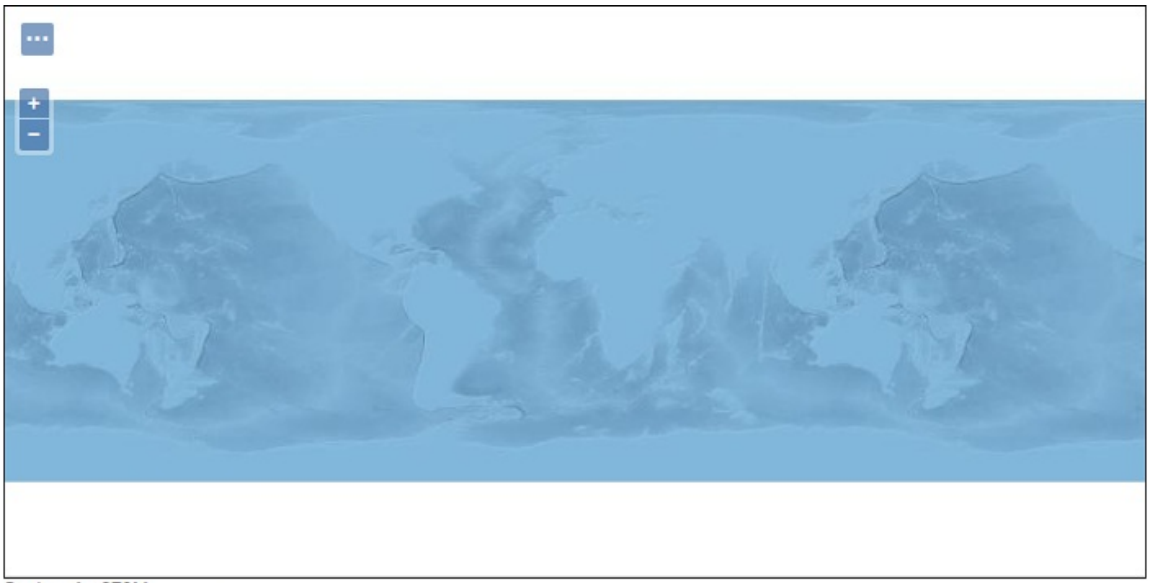

Basic Resource Info
Name
ocean-bottom-relief
Enabled
Advertised
Title
Ocean bottom relief
Abstract
Blended depth colors and relief shading of the ocean bottom derived from CleanTOPO2 data.
Keywords
Current Keywords
WCS
ImagePyramid
OB_LR_pyramid
Remove selected
New Keyword
Vocabulary
Add Keyword
Metadata links
No metadata links so far
Add link
Note only FGDC and TC211 metadata links show up in WMS 1.1.1 capabilities
Data links
No data links so far
Add link
Coordinate Reference Systems
Native SRS
EPSG:4326
EPSG:WGS 84...
Declared SRS
EPSG:4326
Find...
EPSG:WGS 84...
SRS handling
Keep native
Bounding Boxes
Native Bounding Box
Min X
Min Y
Max X
Max Y
-180
-89,999999999982
179,99999999996
90
Compute from data
Lat/Lon Bounding Box
Min X
Min Y
Max X
Max Y
-180
-89,999999999982
179,99999999996
90
Compute from native bounds

- 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

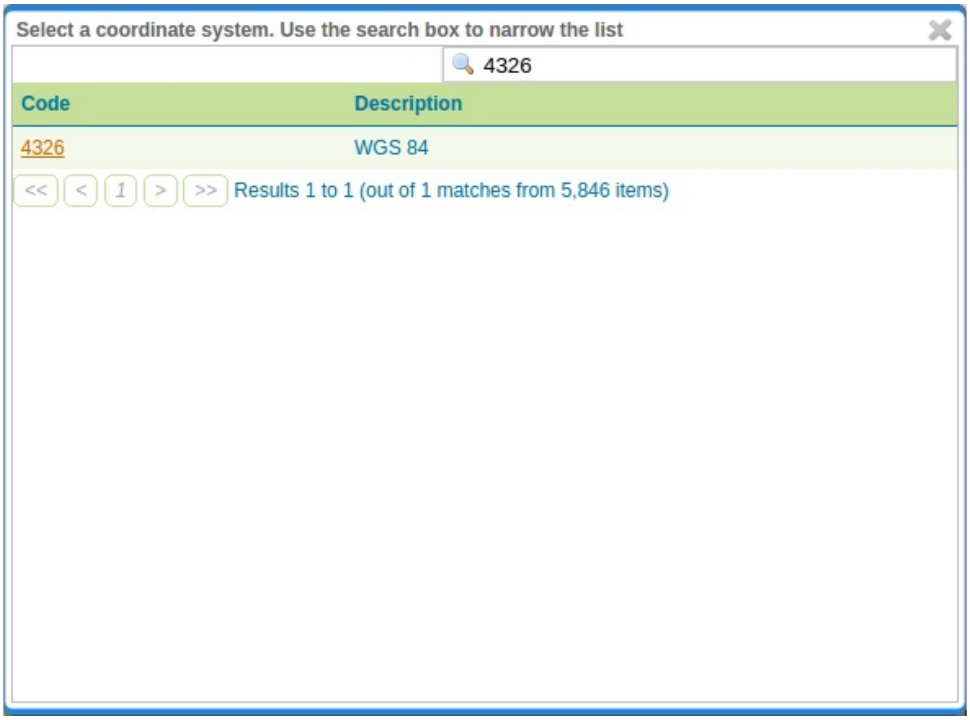
A layer group, as its name suggests, is a collection of layers. A layer group makes it possible to request multiple layers with a single WMS request. A layer group contains information about the layers that comprise the layer group, the order in which they are rendered, the projection, associated styles, 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.

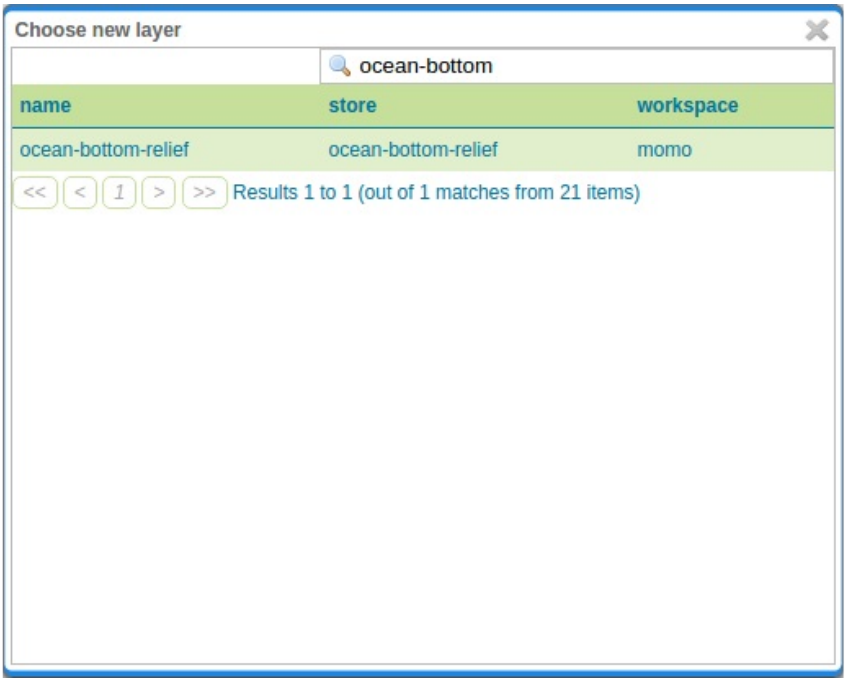
Create grouped layer

Use countries and ocean bottom relief

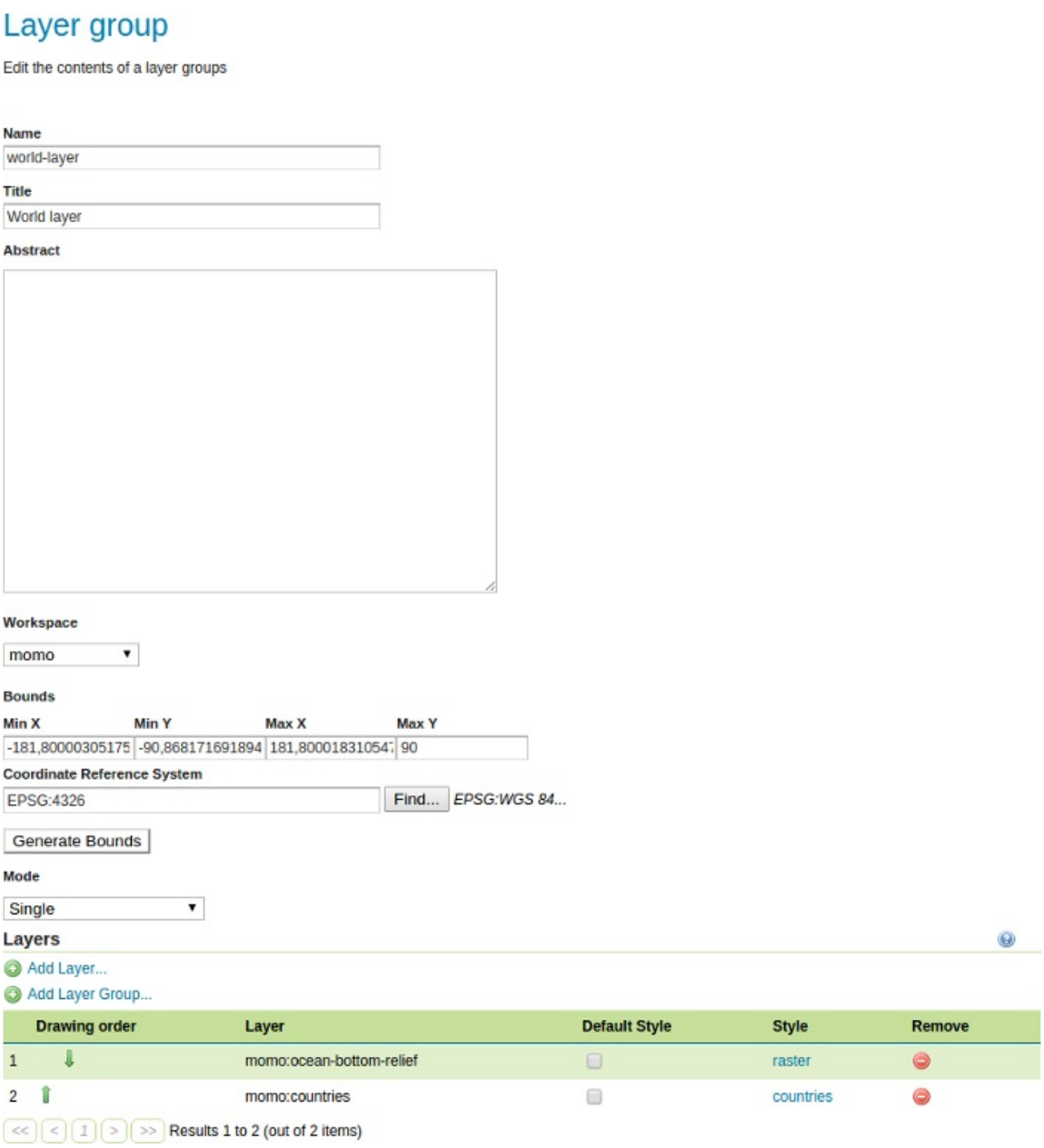
- Go to `Data` ❭ `Layer Groups` ❭ `Add new layer group`.
- Create a new layer group as follows:
  - *Name:* world-layer
  - *Title:* World layer
  - *Workspace:* momo
- Find and select `EPSG:4326` under `Coordinate Reference System`.



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



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

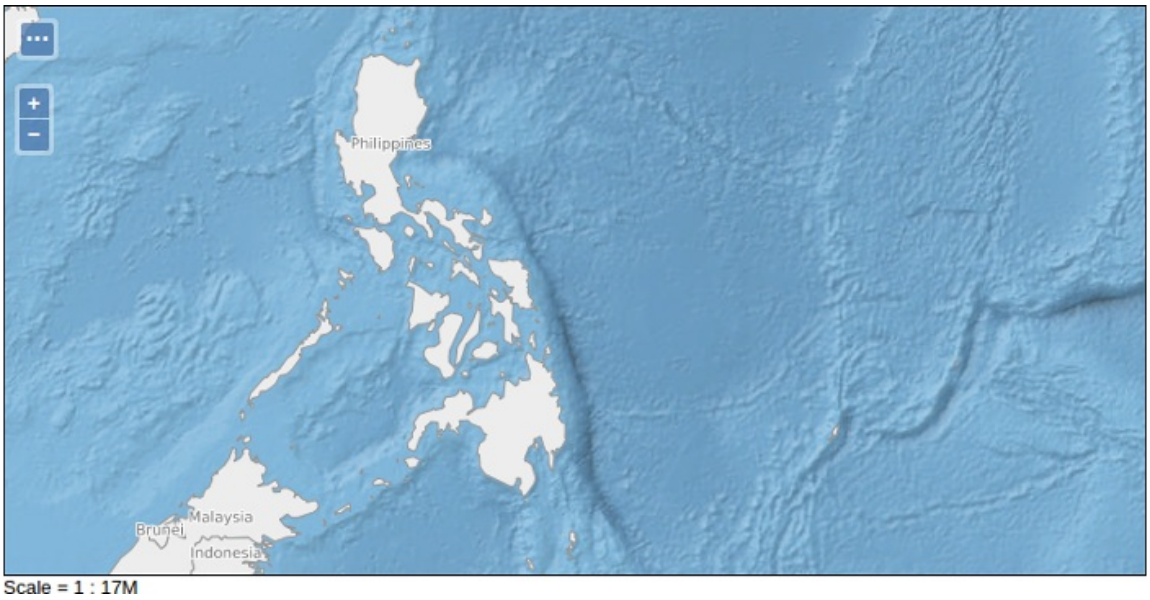

Layer group
Edit the contents of a layer groups
Name
world-layer
Title
World layer
Abstract
Workspace
momo
Bounds
Min X
Min Y
Max X
Max Y
-181,80000305175
-90,868171691894
181,80001831054
90
Coordinate Reference System
EPSG:4326
Find...
EPSG:WGS 84...
Generate Bounds
Mode
Single
Layers
Add Layer...
Add Layer Group...
Drawing order
Layer
Default Style
Style
Remove
1
momo:ocean-bottom-relief
raster
2
momo:countries
countries
Results 1 to 2 (out of 2 items)

- 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`.

- Navigate to `Data` ❭ `Layer Preview`
- Search for `world-layer`.
- Click `OpenLayers`.



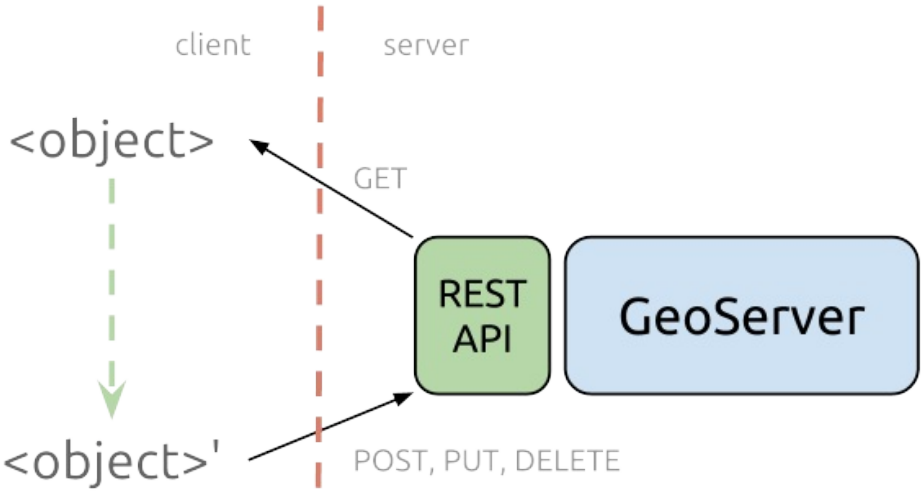
The group layer centered to the Philippines

GeoServer Advanced

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

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

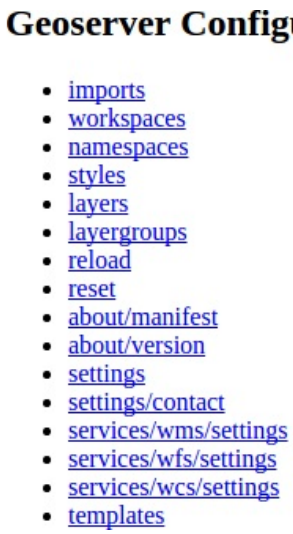
In this module we will learn how to read out the GeoServer configuration via the REST API.

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.

- 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
  ```



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 `http://localhost/geoserver/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.

- 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
  ```

- 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.

- 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`.
- 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.
- 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).

- 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</user>
          <passwd>momo</passwd>
          <dbtype>postgis</dbtype>
        </connectionParameters>
      </dataStore>" \
  http://localhost/geoserver/rest/workspaces/momo-rest/datastores
```

- 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
```

- 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.

- 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
```

- 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_
```

- 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.

- Open the terminal and navigate to your home directory with:

```
$ cd ~
```

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

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

- Copy the linked SLD content (see here) the newly created file and save it with `Ctrl + O`. You can now close the nano editor with `Ctrl + X`.

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

- 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
```

- 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
```

- 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
```

- 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`.

- 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
```

- 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).

- Execute the following terminal command to update the layer `countries_rest` . (Note: Every update needs the property `<enabled>true</enabled>`

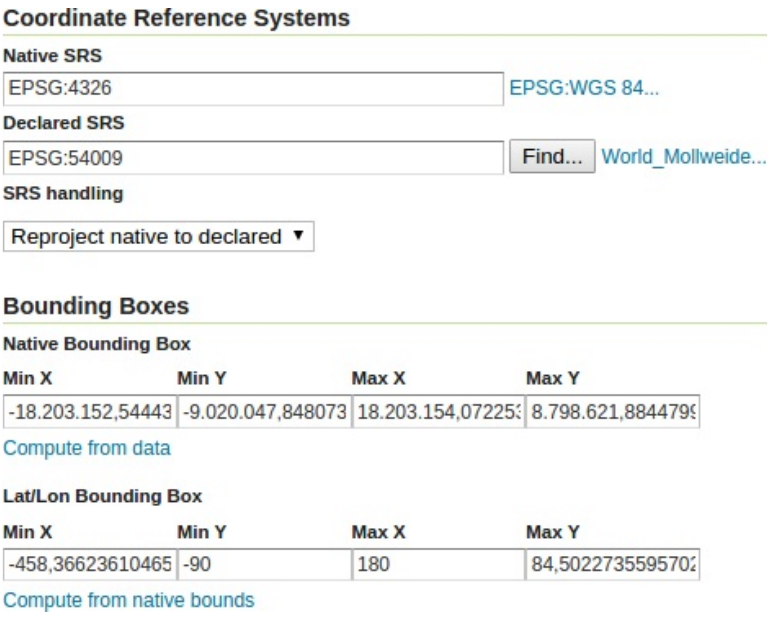
  otherwise the catalog entry, in this case the layer, will be disabled and not be visible to any user!)

```
$ 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
```

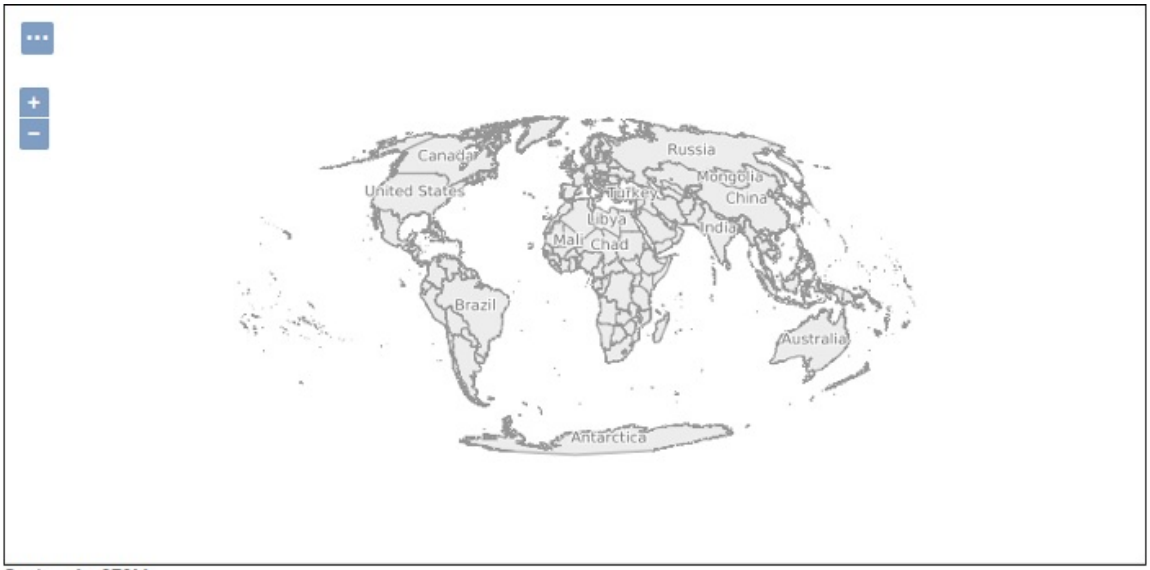
- After this step has been confirmed as successfully finished with `HTTP / 1.1 200 OK` , we can then automatically calculate the new native and lat/lon bounding box of the layer by appending the parameter `recalculate=nativebbox,latlonbbox` to the REST URL:

```
$ 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?rec
```

- 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.



- 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

Layer in EPSG:54009.

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`.

- 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?rec
```

- 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

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:

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

```
$ sudo mkdir /opt/tomcat/webapps/geoserver/data/gwc
```

- Ensure GeoServer has read and write access to this directory by changig the ownership to user and group `tomcat` :

```
$ sudo chown tomcat:tomcat /opt/tomcat/webapps/geoserver/data/gwc
```

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

```
$ sudo gedit /opt/tomcat/webapps/geoserver/WEB-INF/web.xml
```

- The following block will advise GWC to store all cached tiles into the directory `/opt/tomcat/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>/opt/tomcat/webapps/geoserver/data/gwc</param-value>
</context-param>
```

- Save the changes and close the text editor.
- To apply the changes, we need to restart GeoServer. Go to your terminal and run the following command:

```
$ sudo restart tomcat
```

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

```
$ cd /opt/tomcat/webapps/geoserver/data/gwc
```

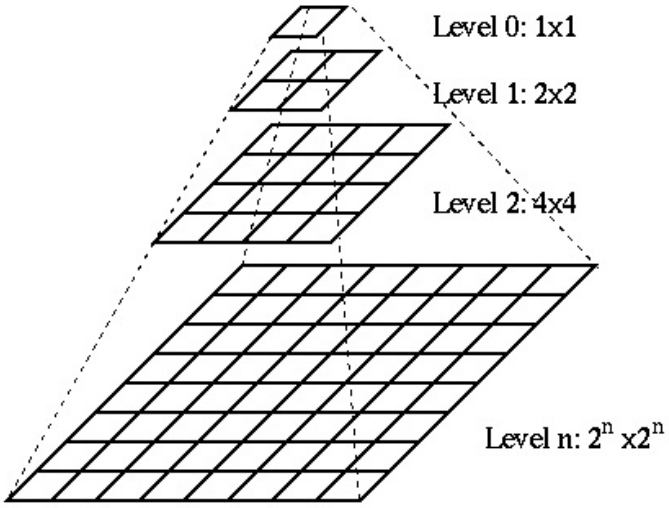
Tiles and gridsets

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/AAAAAAAAAKY/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:

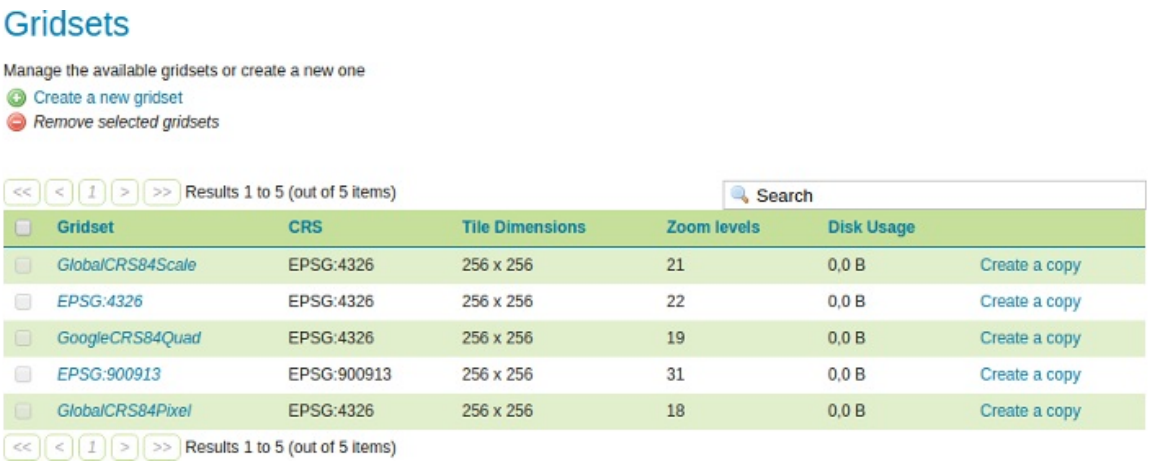
- The gridset for the layer.
- (Optional) The bounding box for that layer (which must be a subset of the extent of the gridSet).
- (Optional) A list of zoom levels (which must be a subset of what is defined in the gridSet).

> For further instructions have a look at the source of the above explanations, here.

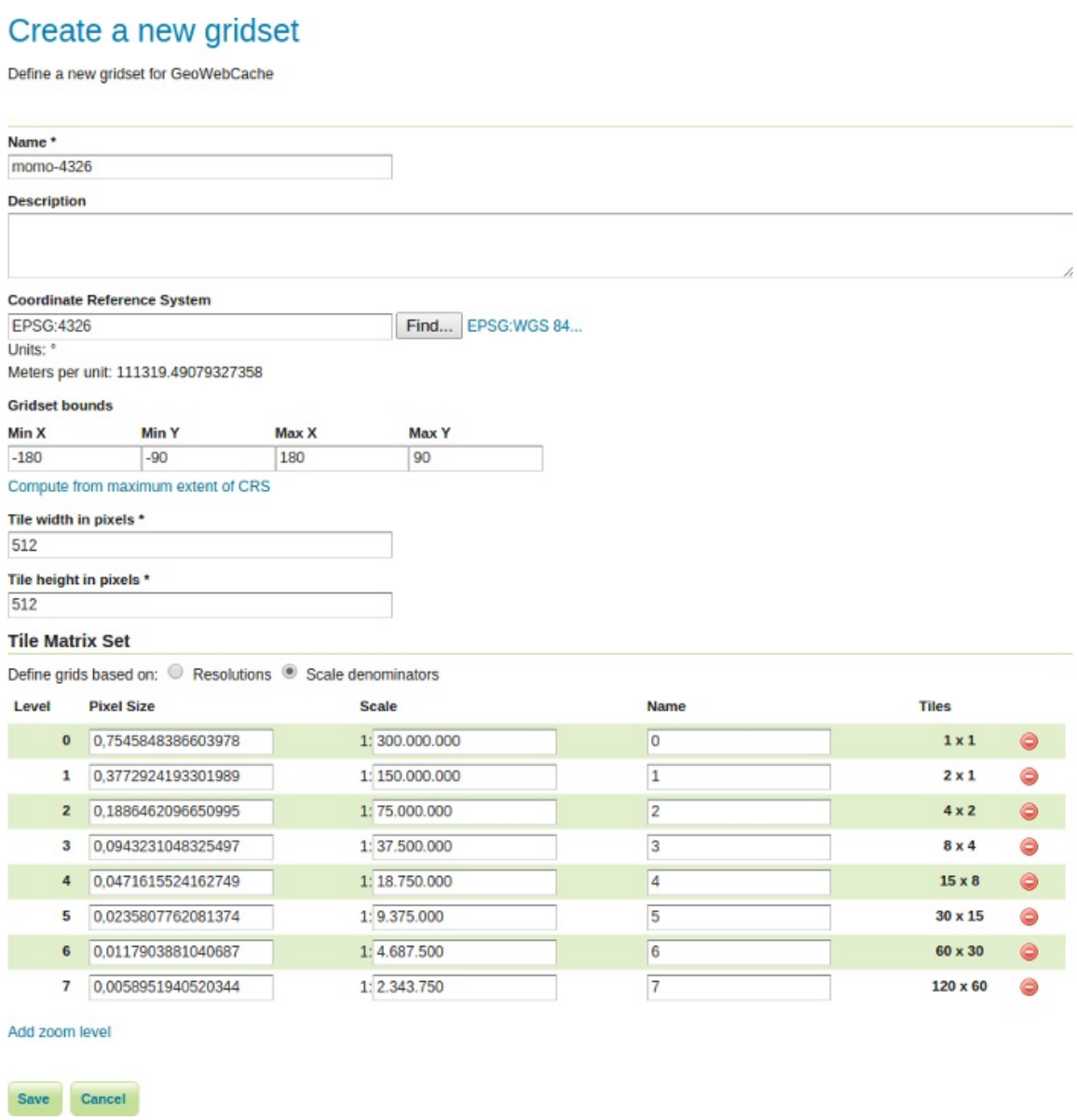
Configure a new gridset

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

- Go to `Tile Caching` ❭ `Gridsets`



- Click `Create a new gridset` to create a new gridset and use the following options for the creation:
  - *Name:* momo-4326
  - *Coordinate Reference System:* Use the find-button to select `EPSG:4326`
  - *Gridset bounds:* Click `Compute from maximum extent of CRS`
  - *Tile width in pixels:* 512
  - *Tile height in pixels:* 512
  - *Define grids based on:* Select `Scale denominators`
- Click `Add zoom level` to create a new zoom level. Enter the scale `300.000.000` and the name `0`.
- 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 *
momo-4326
Description
Coordinate Reference System
EPSG:4326
Find...
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 *
512
Tile height in pixels *
512
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
Save
Cancel

- Click `Save`.

Cached layer

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

- Go to `Data` ❭ `Layers` and select the `countries` layer.
- Open the panel `Tile Caching`.

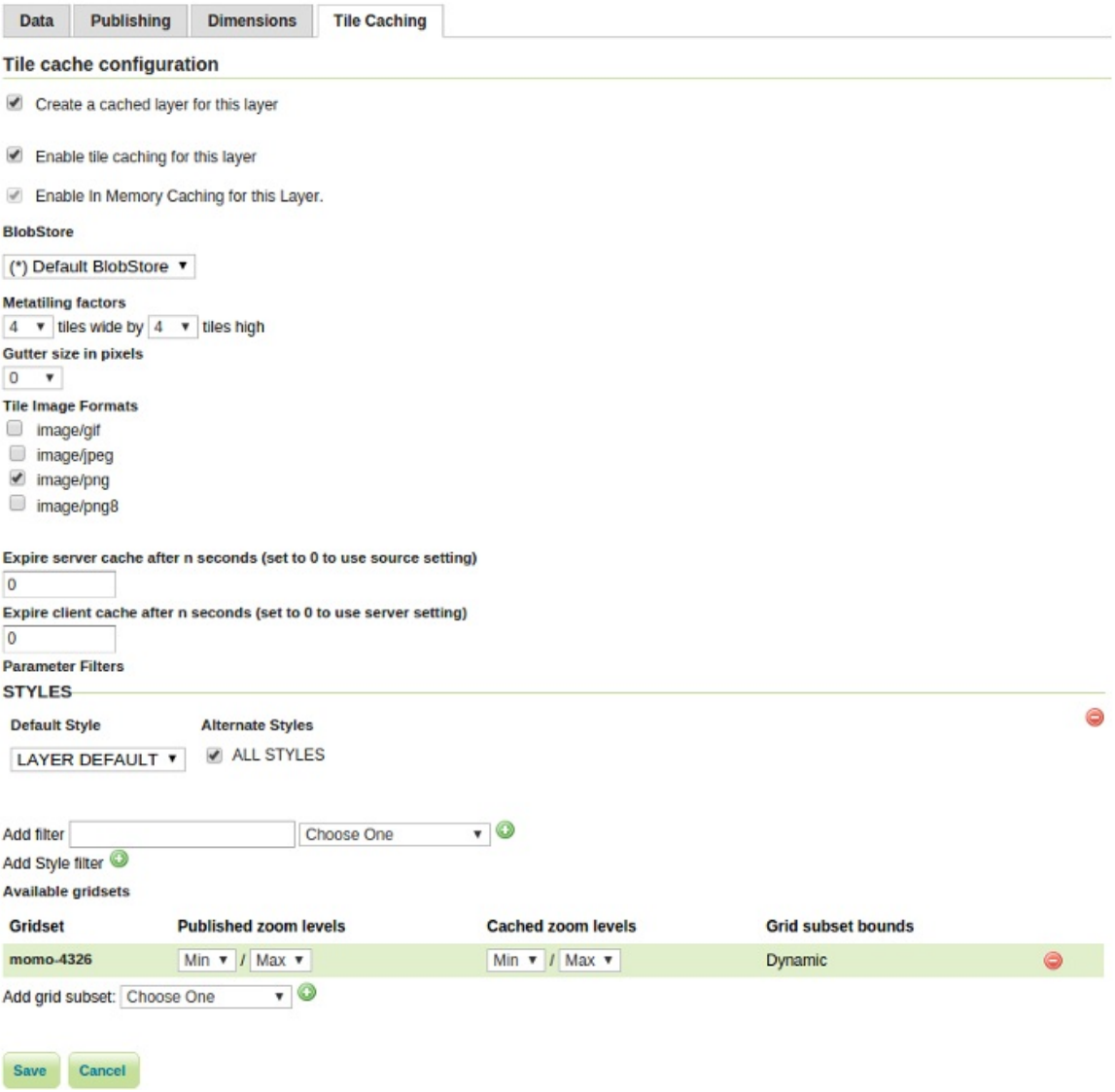
Here we can configure all GWC-dependend properties in a per-layer-basis. The most important configuration parameters are:

- *Create a cached layer for this layer*: Should this layer be cached?
- *Metatiling factors:* Metatiles 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:

- 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

- Click `Save`.

Generate map tiles

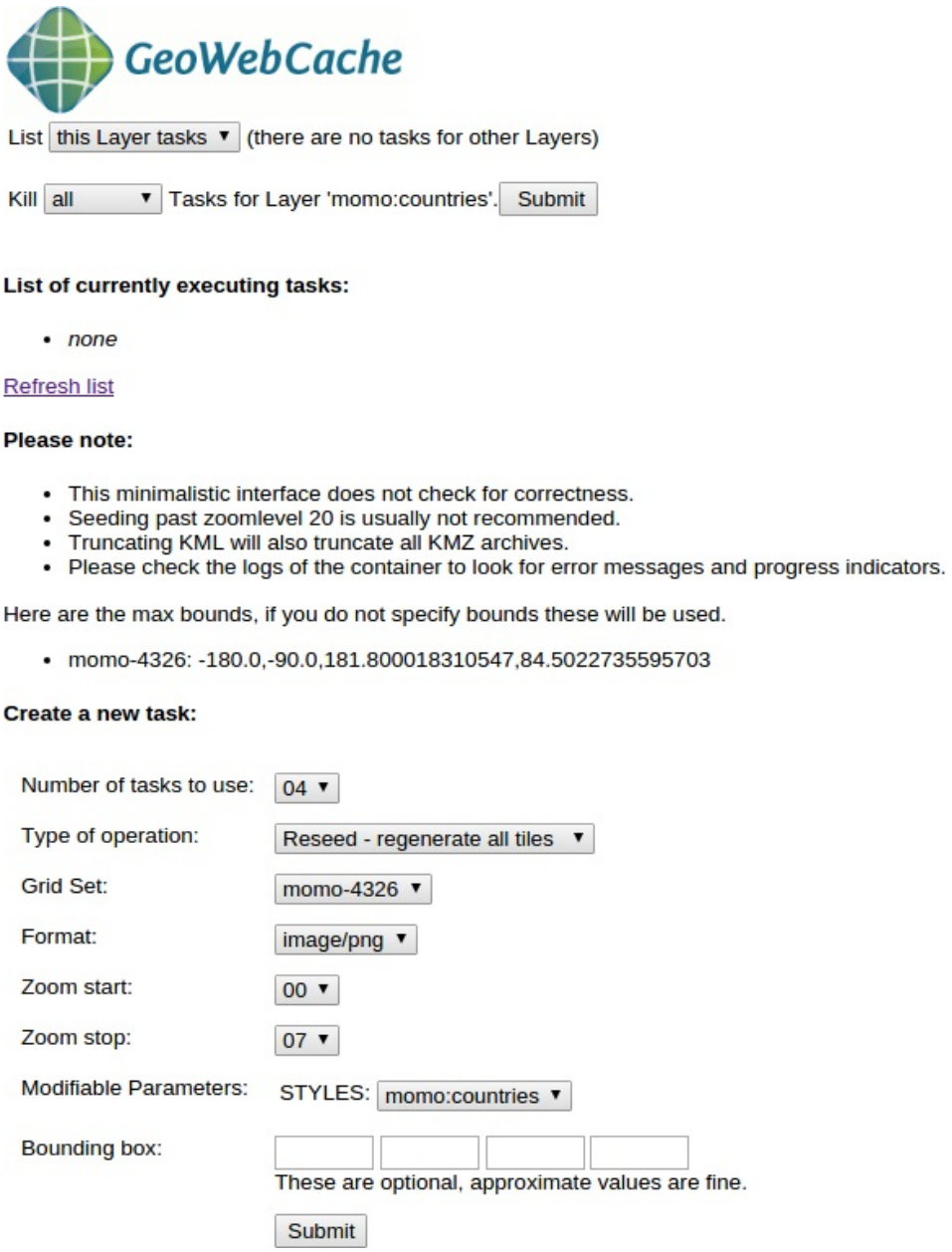
Generally speaking, GWC applies two methods for creating cached map tiles:

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.

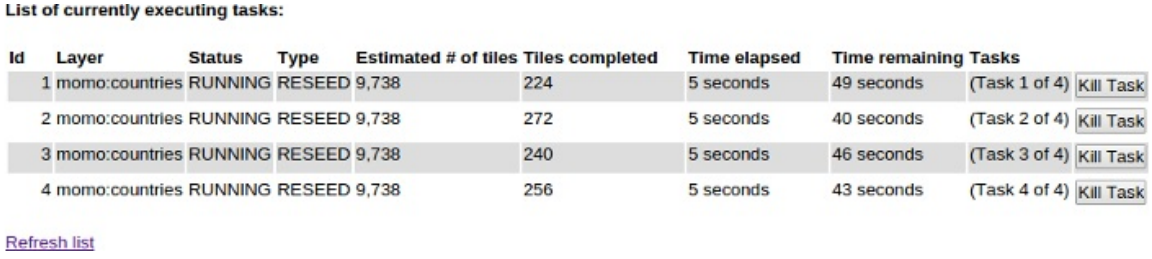
With the following steps we'll preprocess the tiles and start the so called `Seeding` job.

- Go to `Tile Caching` ❭ `Tile Layers` .
- Find the layer `momo:countries` and select `Seed/Truncate` .

- 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



- Click `Submit`.
- 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.



- 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`.

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

```
$ cd /opt/tomcat/webapps/geoserver/data/gwc/momo_countries
```

- List the directory contents with:

```
$ ls -l
```

- 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

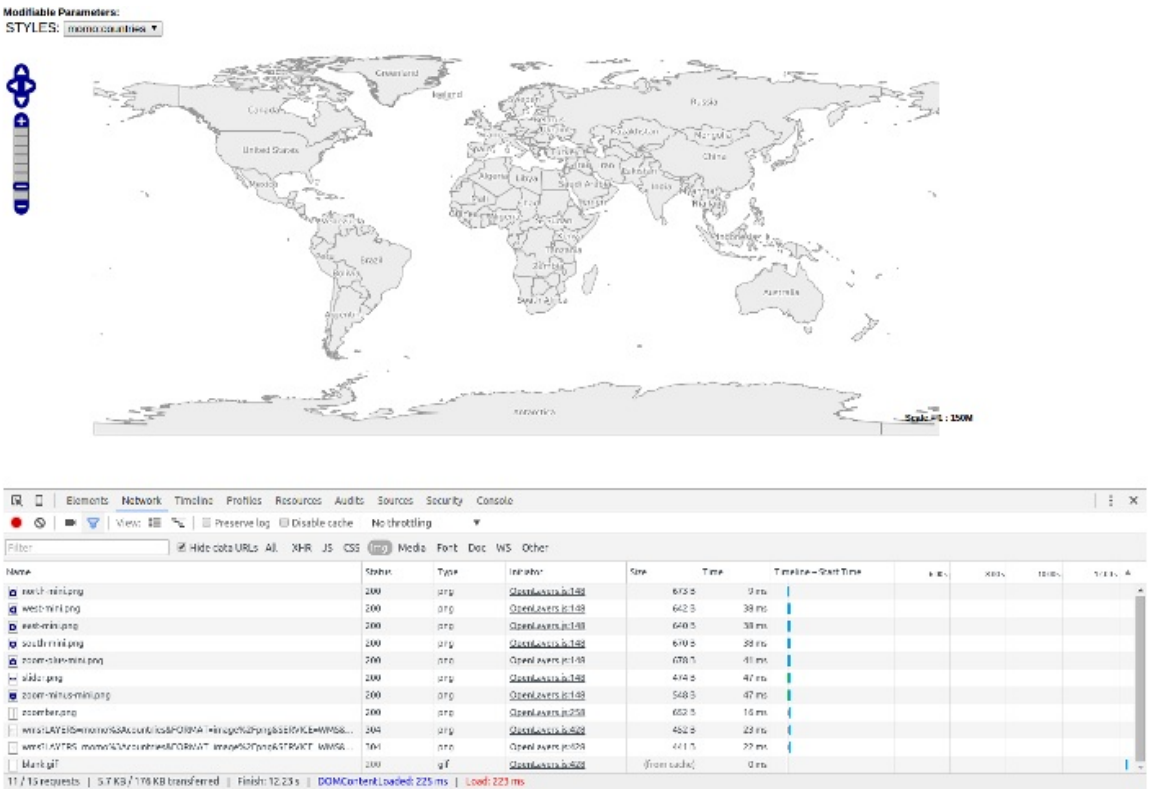
Finally we are going to inspect the response send from the GeoServer/GeoWebCache to the client in more detail. As soon as a layer is being cached by GeoWebCache, the response headers of single tile are extended by the following HTTP-headers:

Header	Description
geowebcache-cache-result	If the tile is delivered by the cache, the value is **HIT** otherwise it's **MISS**.
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:

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

- In the preview window/tab press `F12` to open the browsers **Developer Toolbar**, activate the `Network` tab, select the `Img` subsection and reload the page to record the network activity.

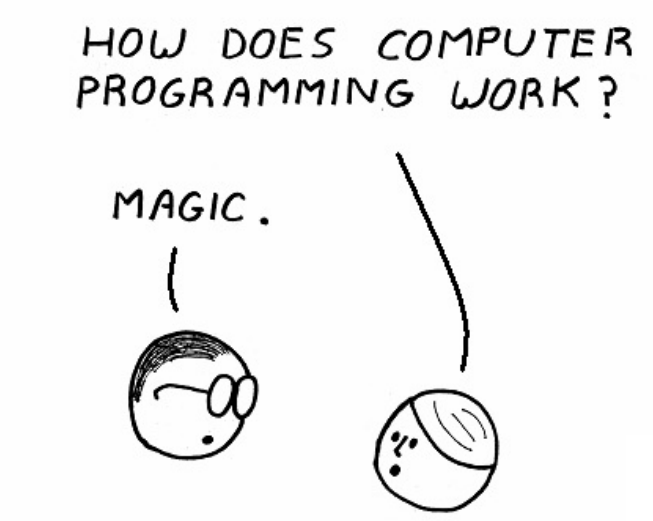


- Clear the list content with the `Clear` button (🚫)
- Zoom in to a location of your choice, find a WMS GetMap request in the developer toolbar and select it. Explore the right hand sided information panel and find the `Response Headers` section. In this you should find the headers looking similar the 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]
```

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.



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:

```
x = 8 - 3
-> x = 5
```

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:

```
x = 4
y = 4
```

or

```
x = 3
y = 5
```

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 substracted, 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 = &quot;5&quot;;` . 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 &gt; a` : is x bigger than a?
- `x &lt; a` : is x less than a?
- `x &lt;= a` : is x less than or equal to a?
- `x &gt;=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 &gt; 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 `&amp;&amp;`.

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, Condtions can be grouped using parenthesis, ex: `if ( (name === &quot;John&quot; || name === &quot;Jennifer&quot;) &amp;&amp; country === &quot;France&quot;)`.

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
}
```

> : `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);
```

> : `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!
```

> this is the **recomended** way.

2. and object-oriented

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

> 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 pro
    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;
    }
}
```

> 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

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=&quot;stylesheet&quot; href=&quot;/ol.css&quot; type=&quot;text/css&quot;>`).

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 `<head>` of our document with `<script src=&quot;/loader.js&quot;></script>`.

Similarly, our custom map initialization code (above) cannot run until the document element that serves as the viewport container, in this case `<div id=&quot;map&quot;></div>`, is ready. By including the initialization code at the end of the document `<body>`, 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 `&quot;map&quot;` 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(&quot;map&quot;)`).

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.

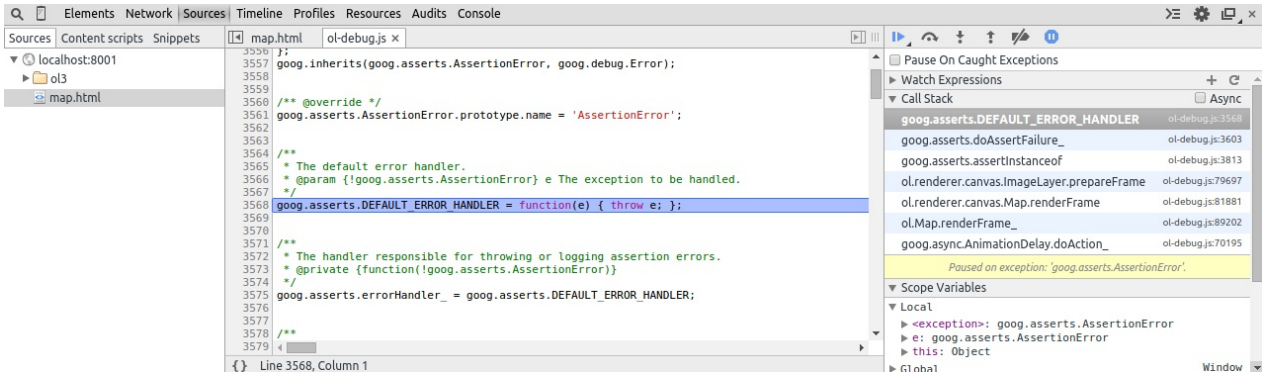
- http://stackoverflow.com/questions/tagged/openlayers-3
- https://groups.google.com/forum/#!forum/ol3-dev

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.



At a breakpoint in the debugger

Layers and Sources

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.

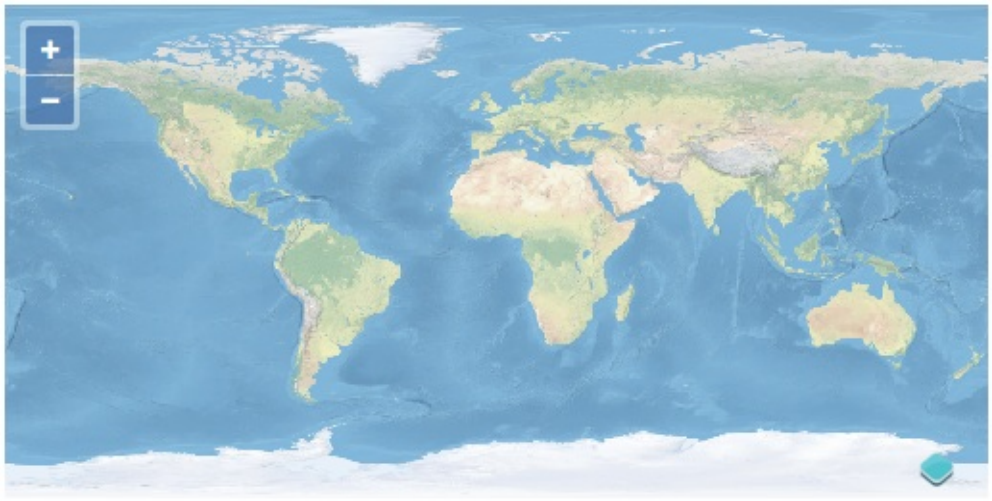
```
layers: [
    new ol.layer.Tile({
      title: 'Global Imagery',
      source: new ol.source.TileWMS({
        url: 'http://demo.opengeo.org/geoserver/wms',
        params: {LAYERS: 'nasa:bluemarble', VERSION: '1.1.1'}
      })
    })
  ]
```

Tasks

1. This same WMS offers a Natural Earth layer named `&#39;ne:NE1_HR_LC_SR_W_DR&#39;`. Change the value of the `LAYERS` parameter from `&#39;nasa:bluemarble&#39;` to `&#39;ne:NE1_HR_LC_SR_W_DR&#39;`. 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 `<head>` 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 `&#39;EPSG:21781&#39;` 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: '<Your Bing Maps Key Here>'
    })
   ```

   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: '<Your Bing Maps Key Here>'
            })
          })
        ],
        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

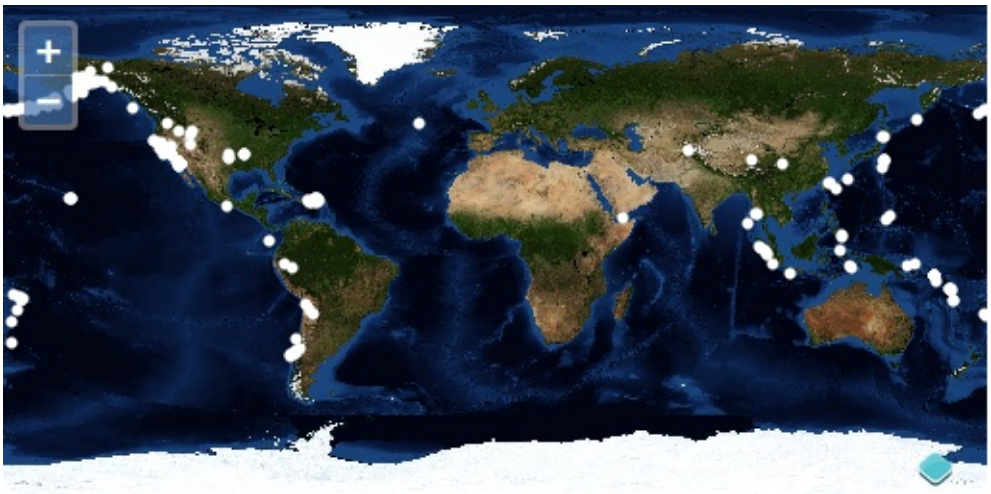
Let's go back to the WMS example to get a basic world map. We'll add some feature data on top of this in a vector layer.

```
<!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 `&#39;Earthquakes&#39;` 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 `&#39;singleclick&#39;` 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 `&#39;singleclick&#39;` listener on your map that calls `forEachFeatureAtPixel` on the map, and displays earthquake information below the map viewport.

Controls and interactions

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.

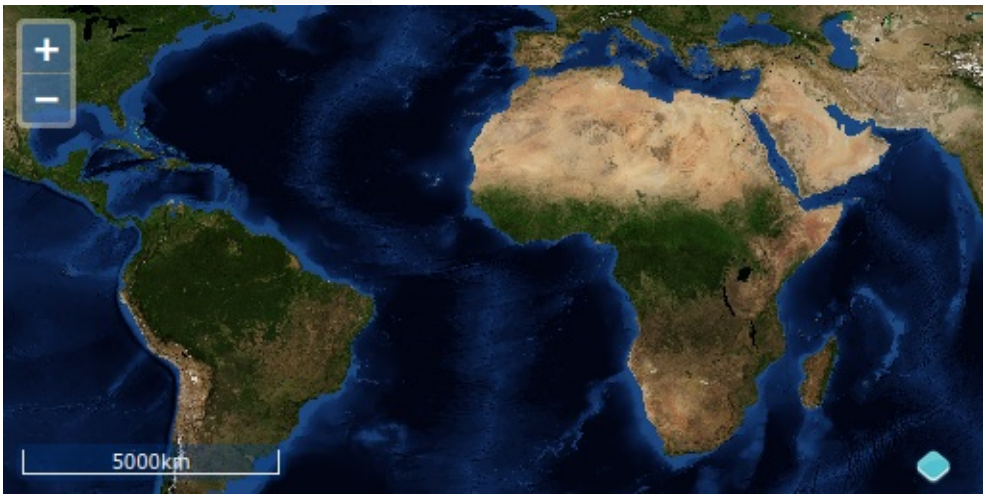
Creating a ScaleLine Control

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=&quot;map&quot;></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.

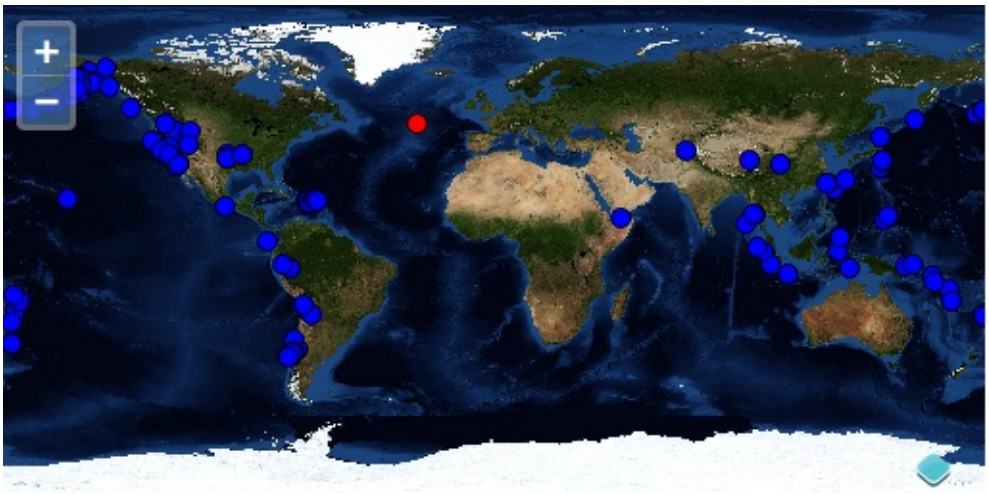
Create a Vector Layer and a Select Interaction

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.

Solutions

Here is a solution for the first bonus task. In it we register an event listener for the `drawend` event of the `ol.interaction.Draw`. This method logs the feature's X and Y to the developer console:

```
draw.on('drawend', function(evt){
  var feature = evt.feature;
  var p = feature.getGeometry();
  console.log(p.getCoordinates());
});
```

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>
```

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 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

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 `&#39;someClass&#39;`) 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 `&#39;someClass&#39;` :

```
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 `&quot;someClass&quot;` (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

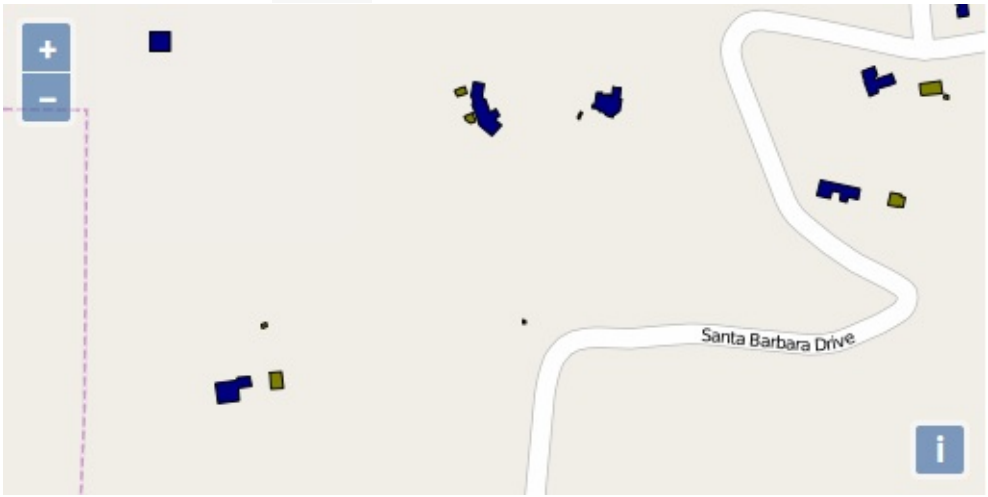
1. We'll start with a working example that displays building footprints in a vector layer. Open your text editor and save the following as `map.html` in the root of your workshop directory:

```
<!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 `&#39;Buildings&#39;` 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;
   }
 }
```

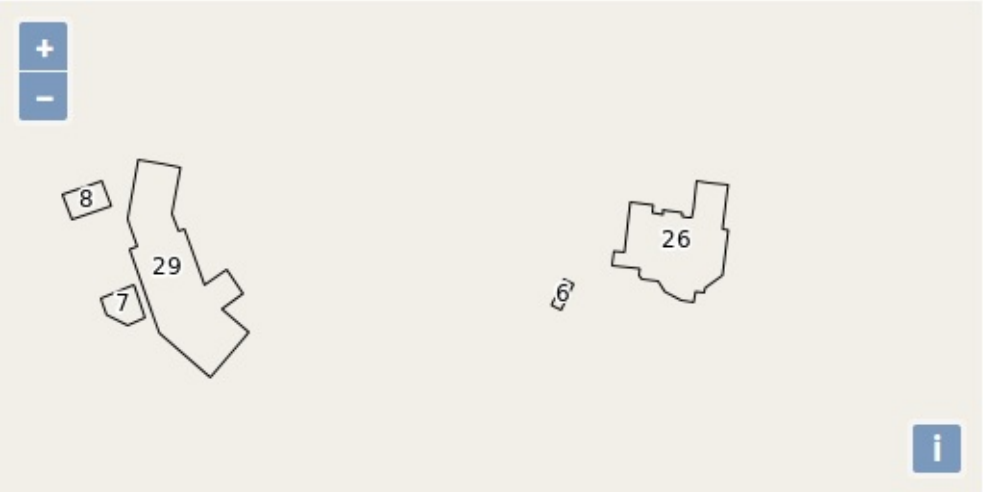
4. Save your changes and open `map.html` in your browser: http://terrestris.github.io/momo3-ws//map.html



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


8
29
7
26
6

Custom Builds

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 that 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 downoad 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 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!

Ext JS Workshop

Welcome to the **Ext JS Workshop**. This workshop is designed to deliver you a first insight into the JavaScript framework Ext JS for developing web applications. As this workshop is further intended for beginners we'll mainly focus on the core concepts and components delivered by Ext JS by simple tasks. Hence we'll learn how to include the framework into a basic HTML page, how to use the `Viewport`, what user interface components Ext JS provides to us and how to programmatically interact with these components.

These goals are subdivided into the following sets of modules:

- Introduction to Ext JS
- Basics
- Layouts
- Components
- Data
- Events

Let's start with the introduction to Ext JS!

Introduction

What is Ext JS?

Ext JS is a JavaScript based application framework for developing interactive cross-platform applications and is a product of Sencha.

Useful resources

API documentation

The quantity and quality of the ExtJS API documentation is outstanding.

- http://docs.sencha.com/extjs/6.0/6.0.0-classic/

The docs provide a list of all classes on the left and details once you click on any class. In order to understand and make use of ExtJS, it is crucial to fully grasp the documentation.

Examples

The examples for the ExtJS framework can be found here:

http://examples.sencha.com/extjs/6.0.0/examples/

As with the API documentation, you may at first be overwhelmed at the sheer masses of examples. It is nonetheless very useful to click through some of them, as the show how to combine the classes of the framework into small working applications.

Other

- If you want to quickly check some class, you can e.g. use the Sencha Fiddle website.
- Specific questions (and answers) can be browsed in the Sencha Forums.

Workshop setup

The workshop requires a minimum of preliminary work. Please follow the next steps to set up your workshop environment.

Prepare workshop folder

In this workshop we're going to learn the basics of Ext JS by the use of simple consecutive exercises, where we're going to create (and save) HTML files on your local computer. In order to have a comparable setup, at first we'll create an appropriate workshop folder, where to put these files in.

- Open the terminal and navigate to your home folder with:

```
$ cd ~
```

- Create a new folder named `ext-workshop` with:

```
$ mkdir ext-workshop
```

- Enter the newly created directory with:

```
$ cd ext-workshop
```

Some examples will need some additional files. Download these files by performing the following steps:

- Reopen the terminal and download the workshop files into your `ext-workshop` folder with:

```
$ wget http://terrestris.github.io/momo3-ws/en/extjs/materials.tar.gz
```

- Extract the downloaded archive file with (this will create a folder named `materials` in your workshop directory):

```
$ tar -xvzf materials.tar.gz
```

Prepare simple workshop HTTP server

For the first exercises we won't have any need for serving our code snippets via the HTTP protocol, but in the later parts we're going to use technologies that require a local HTML server. A very simple way to serve a content of a specific directory over a web server is to use the SimpleHTTPServer module provided by Python. In the next few steps we're going to start the `SimpleHTTPServer` in the workshop directory where all upcoming exercise files will be served over the HTTP protocol.

- (optional) Open the terminal and navigate to your workshop folder with:

```
$ cd ~/ext-workshop
```

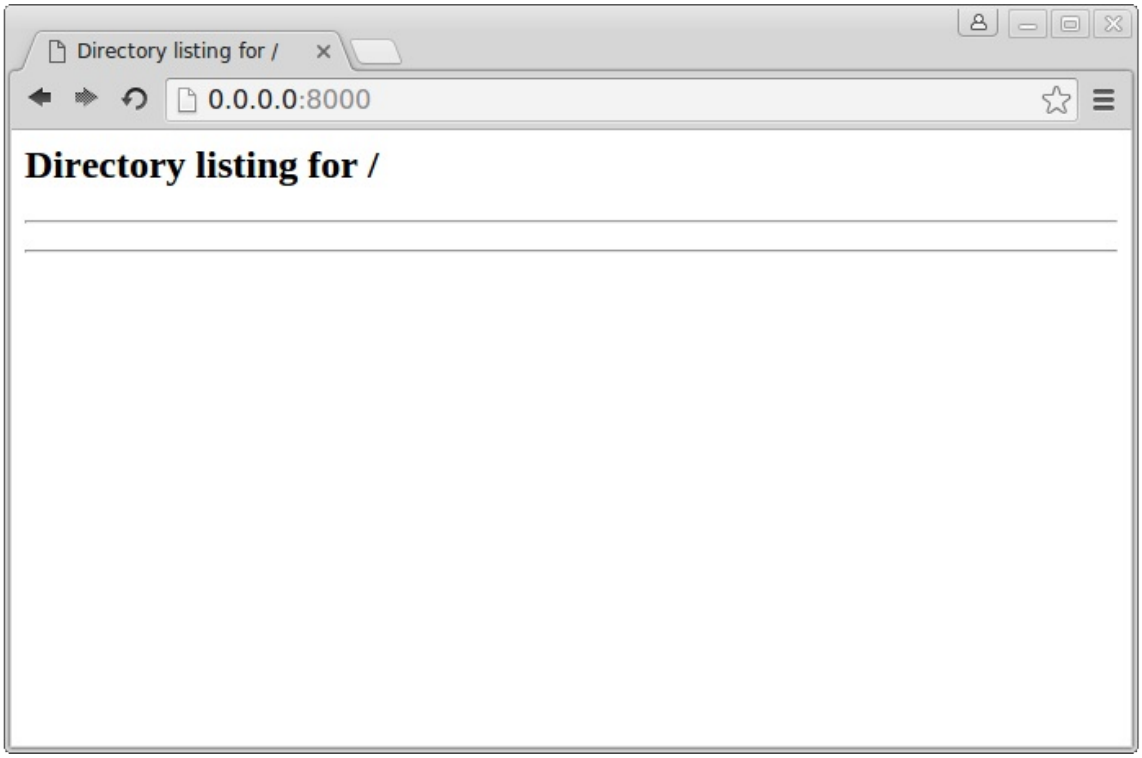
- Run the `SimpleHTTPServer` with:

```
$ python -m SimpleHTTPServer
```

This should give you the following output meaning that the files of the current directory are served through port number **8000**.

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

- Finally open a web browser and navigate to `http://0.0.0.0:8000` which should give you a listing of all available files in the served web directory.



The running SimpleHTTPServer

> If you want to quit the server, you can either simply quit the terminal or press `Ctrl + C`.

Basics

Now that our development environment is ready, it's time to get started.

In this chapter we'll learn how to:

- Create a simple HTML file and how to embed Ext JS
- Create your first Ext JS component
- Create and configure the Ext JS Viewport

Basic HTML file with Ext JS

We'll start the workshop with creating a simple HTML page and embed the framework into it.

Create basic HTML page

Basically we'll work with a single HTML file we'll extend gradually within each section only. Our initial file will thereby only contain the basic HTML template showing a heading. Let's create the file by the use of the (highly recommended) text editor `atom`.

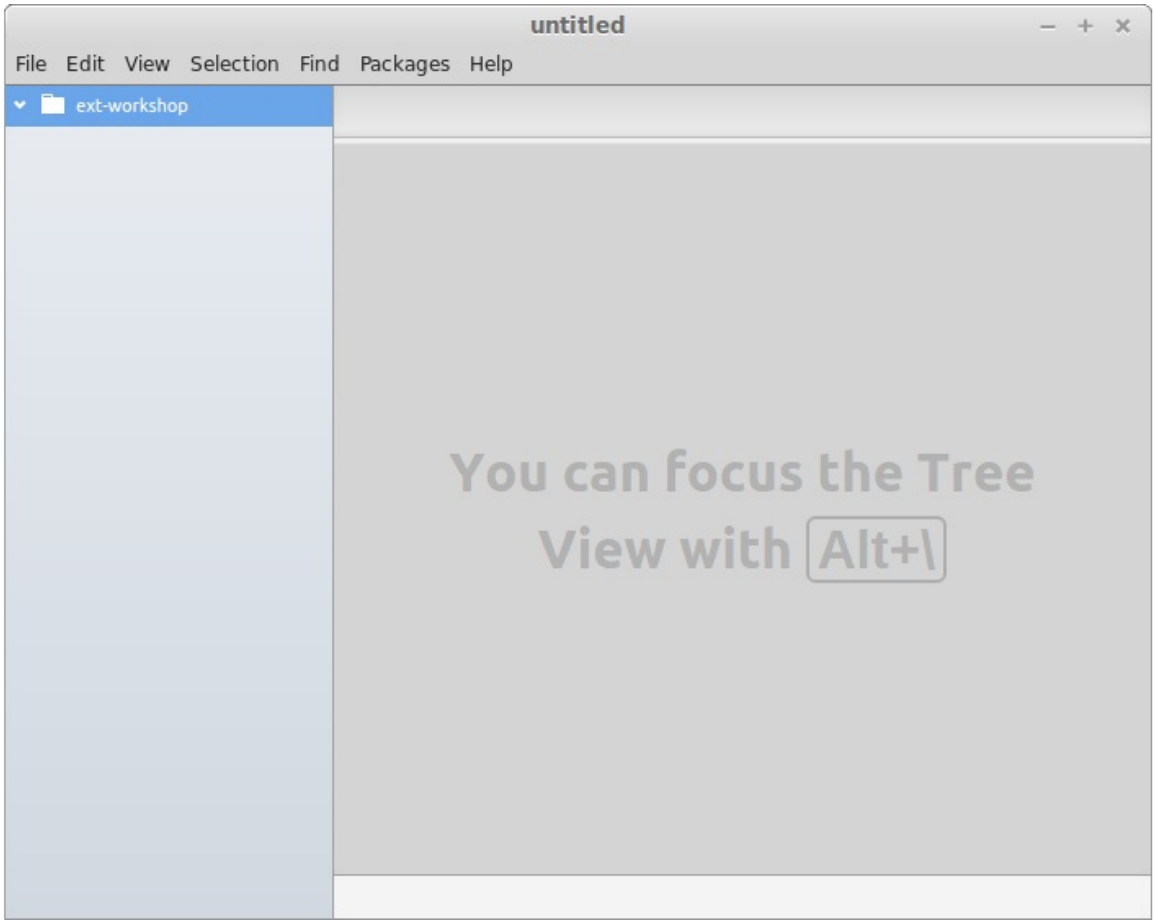
Exercise

- Open the terminal and navigate to your workshop directory (if not already done) with:

```
$ cd ~/ext-workshop
```

- Start the `atom` editor with the `ext-workshop` directory as project:

```
$ atom .
```



Start view atom editor.

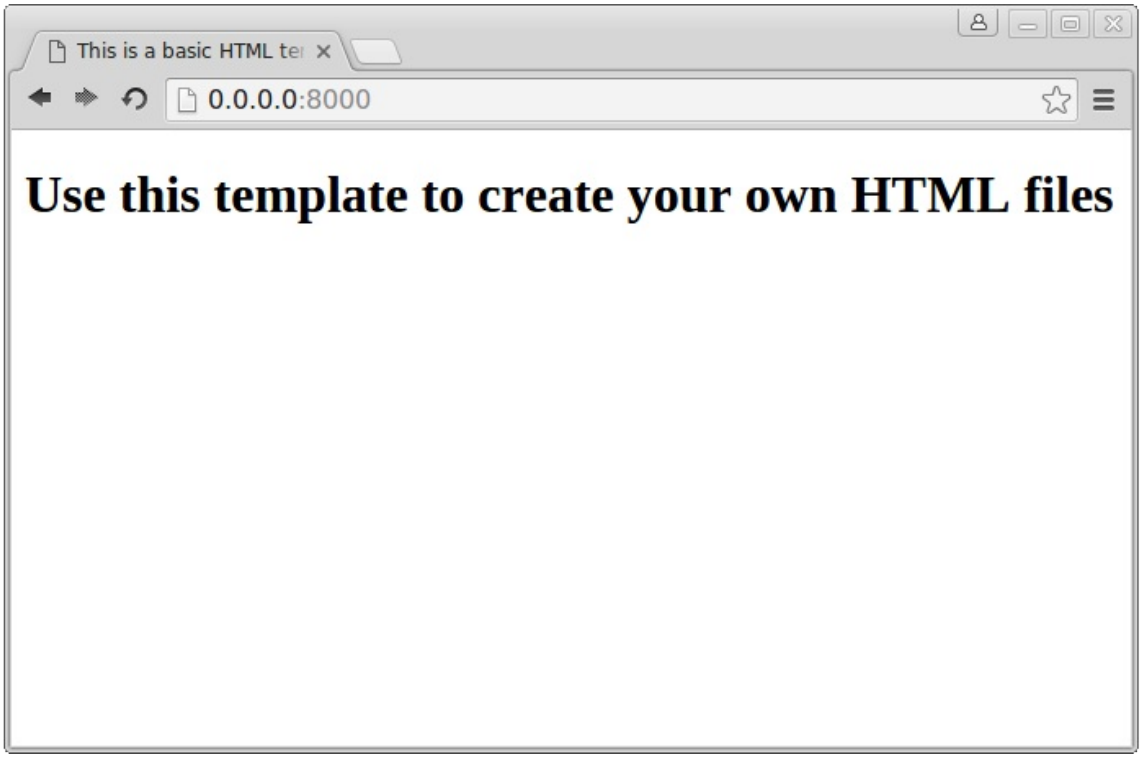
Having `atom` opened we can create a new file by opening the context-menu on the project folder `ext-workshop` and selecting `New File`.

- Create a new file named `index.html` in the exercise directory and copy the content of the following basic HTML template into it

> basic-template.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>This is a basic HTML template</title>
    </head>
    <body>
        <h1>Use this template to create your own HTML files</h1>
    </body>
</html>
```

- Reopen the browser and (re-)load the URL http://0.0.0.0:8000 to see the changes take effect:



The basic HTML page.

Include Ext JS

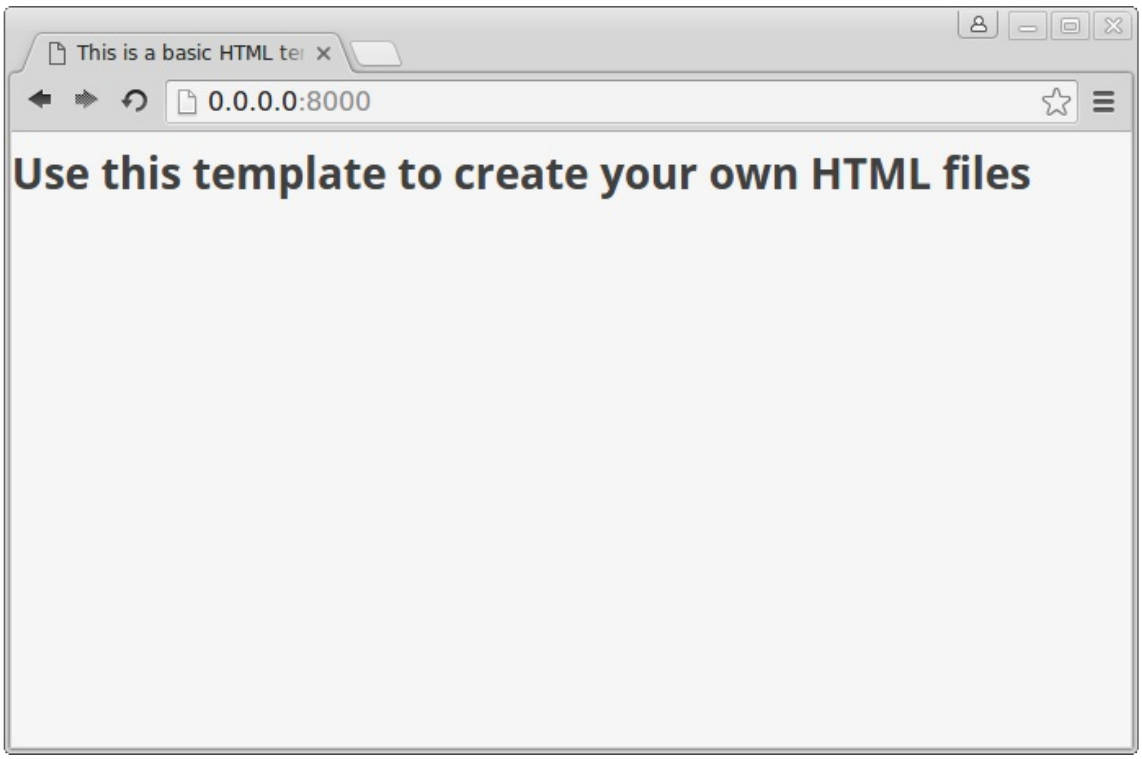
In the next step we'll insert two important lines into the `index.html` that will automatically include the *full* ExtJS library into our basic HTML template. The Ext JS code itself is also available online via `cdnjs`, so we don't necessarily have to download the framework code to our local machine first. As you will see in the next few steps, the (productive version of the) framework consists of two files: Both a `css` (*Cascading Style Sheets*) and a `JS` (*JavaScript*) file:

Exercise

- Include the external files inside the end of your `<head>` element of your `index.html`:
  > include-ext-cdnjs.html

```
<!-- include a CSS stylesheet -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/theme-tri
<!-- include an external JavaScript file -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
```

- Again, reload the URL in the browser and keep track of the changes:



The basic HTML page after including Ext JS.

> For this workshop it's satisfying to include the full builds of the framework and to always load them in the `head`. This technique allows us to basically forget about these resources for the course of the workshop. For a production website you would probably load the files in a different manor, and you would rather not load the versions of the libraries which contain everything. But the creation of specific versions of the base libraries that only include what your application actually needs, is way beyond the scope of this workshop.

Ext JS is here!

Having the last module finished, we have a simple HTML page without any real use of the Ext JS framework present. We'll continue by creating a very simple Ext JS component to verify the framework is ready to work with.

Exercise

- (Re-)open your `index.html` and replace the `<h1>` element block within the `<body>` tag with the following `<script>` block:

  > open-window.html

```
<script type="text/javascript">
    Ext.onReady(function() {

        Ext.create('Ext.window.Window', {
            title: 'Hello',
            height: 200,
            width: 400,
            layout: 'fit',
            bodyPadding: 15,
            constrain: true,
            html: 'Ext JS is here!'
        }).show();

    });
</script>
```

- Reload the page in the browser and look what's going to happen:

Hello Ext JS.

So, what have we done to create this simple Ext JS window?

The contents of the `<script type="text/javascript">` tag will be interpreted as JavaScript by the browser and any JavaScript code in it'll be run as soon as the interpreter sees it. In the next line we are finally going to *really* work with Ext JS. `Ext` is the global namespace that encapsulates all classes, singletons and methods provided by the framework. By calling it on the root scope (as the global singleton object), we have access to the global methods provided by Ext JS. Here we execute the method `onReady()` which has an anonymous function as argument. This function is being processed as soon as the document is ready (but before the document's `onload` listener and before images are loaded).

> As already mentioned, the Ext JS API documentation is quite substantial and really helpful while developing applications. Please take your time to get familiar with the documentation and start by inspecting the docs for the method `onReady()` used above by following this link.

In the anonymous function we pass to the `Ext.onReady()` method we execute - once again on the Ext global object - the method `create()`. With the help of this method we instantiate a Ext JS class (to be more specific: a subclass of `Ext.Base`) by its full class name. Here we create the class `Ext.window.Window`, which, as you may noticed, is a floating, resizable and draggable window containing simple HTML text as content. Every component has a individual set of configuration parameters (e.g. `title`), which are passed to the `create()` method as the second parameter (and bunched in an object). And over again: See the documentation for a full list of all available `Configs` for the window class.

Ext JS Viewport

In contrast to other JavaScript frameworks (e.g. jQuery) Ext JS is typically used to serve as an integral framework that is used to build feature-rich single page applications (SPA) and not as some kind of an "utility" or "helper" framework for isolated challenges. Thus you would generally not use Ext JS to integrate a single window (as created in the former section) in your existing webapplication.

When developing an Ext JS application, (one of) the main components you're dealing with, is the `Ext.container.Viewport`. The `Viewport` class represents a specialized container that automatically resizes itself to the size of the `document body` and therefore the viewable application area. Further on it automatically resizes due to resizing the browser window and will perform sizing and positioning on its child components as you can add other Ext JS UI components and containers to it. How the positioning inside the `Viewport` takes place is thereby configurable by a so called `Layout` (see next chapter).

> Given that the `Viewport` sizes to browser window, it's reasonable to have a **single** viewport per Ext JS application only.

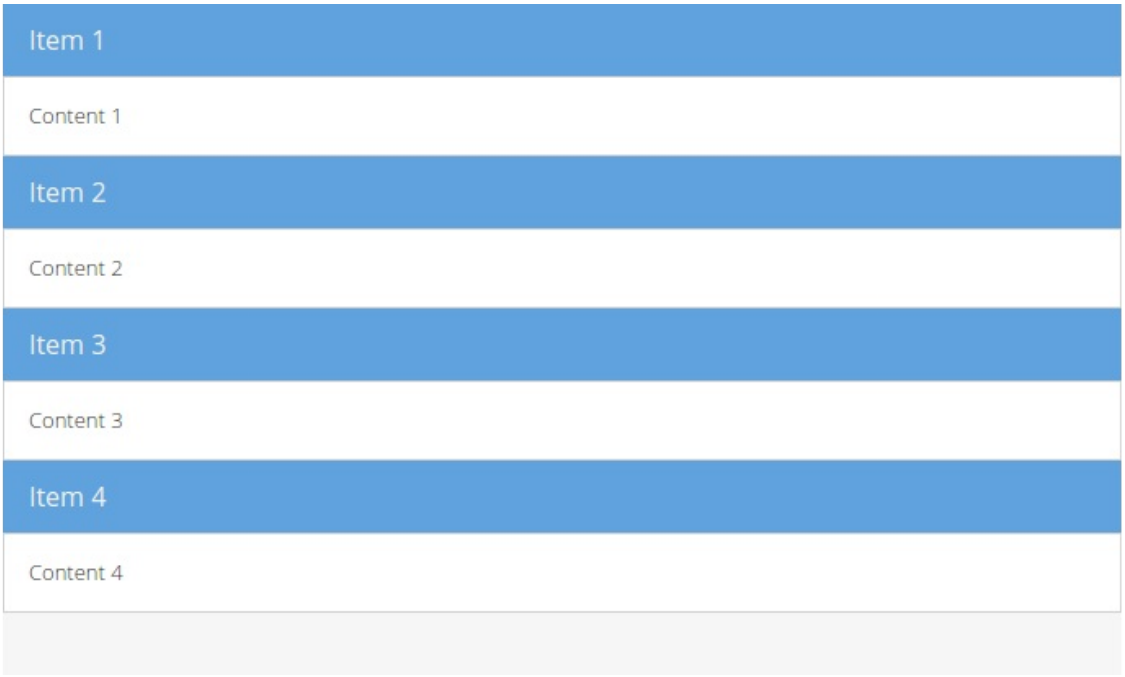
In this section we're going create a simple viewport containing a set of nested child components. This viewport will then act as the basic template for any further exercise in this workshop.

Exercise

- (Re-)open your `index.html` and replace the code creating the `Ext.window.Window` component with the following snippet
  > viewport-simple.js

```
Ext.create('Ext.container.Viewport', {
    defaults: {
        bodyPadding: 15
    },
    items: [{
        title: 'Item 1',
        html: 'Content 1'
    }, {
        title: 'Item 2',
        html: 'Content 2'
    }, {
        title: 'Item 3',
        html: 'Content 3'
    }, {
        title: 'Item 4',
        html: 'Content 4'
    }]
});
```

- Reload the application page in the browser and you'll notice the first elementary indications of a simple full-screen webapplication (Try to resize the browser window!):



Simple viewport.

In the above example we used the method `Ext.create()` to instantiate the component `Ext.container.Viewport` very similar to the previous example. The result is a stack of four components composed of a `title` and a `html` value. Per `defaults` we declare that each direct child component in the `Viewport` - configured as `item` - should be rendered with a `bodyPadding` of 15 pixels additionally.

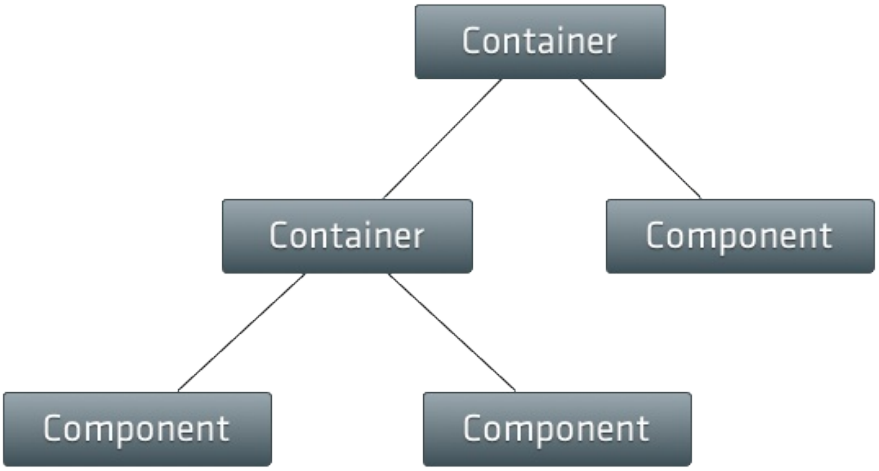
Exercise

- Use the Ext JS API documentation for the `Viewport` class (here) to answer this question:

  > I created an Ext JS viewport with a couple of items without any specific configurations except `title` and `html`. At his juncture the `html` parameter includes a plenty of text content. Strangely I could not see any scrollbars, if the browser window gets resized. What is wrong here?

Layouts

An Ext JS application (UI) is always made up of single components based on the base class `Ext.Components`. The `Viewport` class we have seen in the previous chapter is some kind of a special type of a component as it may contain other components. Once Ext JS components are inserted into a superior container, its layout properties must be defined (according to the requirement by your application).



Component architecture, source: https://docs.sencha.com/extjs/6.0/core_concepts/images/component_architecture.png

The `Layout` tells this superior container (e.g. the `Viewport`) how to properly arrange its child components (e.g. `Panel`) in sizing and positioning. As you may correctly note, the recent example we were using hasn't any specific layout set, but could be rendered in the browser. This happens because the default layout for all containers is the layout type `Auto` and this layout does not specify any special positioning or sizing rules for child elements. It simply renders the child items as normal block elements in the DOM.

Generally the layout of a `Container` has the be set via the `layout` configuration attribute. In most cases it's satisfactory to set the name of the requested layout as a simple string (e.g. `'auto'`) only, but there are layouts available where a full object, specifying the layout options in more detail, are allowed. Furthermore several layouts hold particular attributes related to the child components of the container specifying e.g. it's inner position or size.

In this section we're going to have a quick look to some of the predefined layouts Ext JS provides to us. Here we focus on the following layouts:

> The descriptions given in the upcoming subsections are based on the API documentation.

- Column
- HBox
- VBox
- Accordion
- Table
- Border

For a full list of all layouts have a look at the API documentation or the Kitchen Sink.

Column

The `Column` layout is the layout style of choice for creating structural layouts in a multi-column format where the width of each column can be specified as a percentage or fixed width, but the height is allowed to vary based on the content.

The layout does not have any direct `config` options, but it does support a specific `config` property of `columnWidth` that can be included in the `config` of any panel added to it. The layout will use the `columnWidth` (if present) or `width` of each panel during layout to determine how to size each panel. If `width` or `columnWidth` is not specified for a given panel, its width will default to the panel's `width` (or auto).

The `width` property is always evaluated as pixels and must be a number greater than or equal to 1. The `columnWidth` property is always evaluated as a percentage and must be a decimal value greater than 0 and less than 1 (e.g. .25).

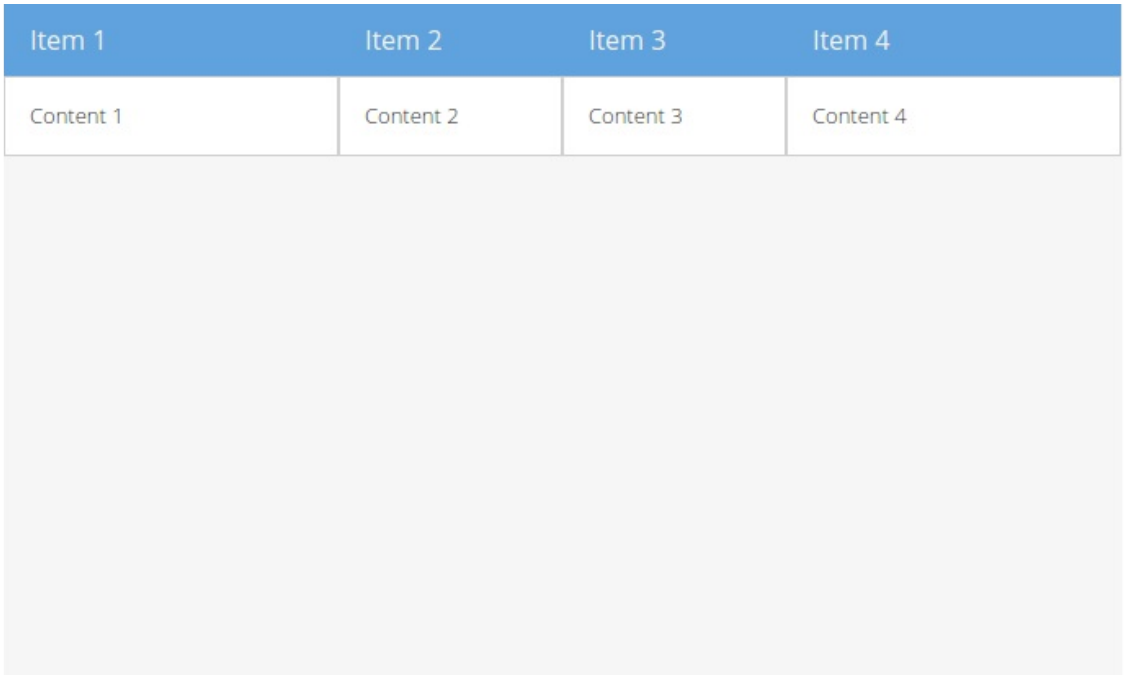
Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:

  > layout-column.js

```
Ext.create('Ext.container.Viewport', {
    layout: 'column',
    defaults: {
        bodyPadding: 15,
    },
    items: [{
        title: 'Item 1',
        columnWidth: 0.3,
        html: 'Content 1'
    }, {
        title: 'Item 2',
        columnWidth: 0.2,
        html: 'Content 2'
    }, {
        title: 'Item 3',
        columnWidth: 0.2,
        html: 'Content 3'
    }, {
        title: 'Item 4',
        columnWidth: 0.3,
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:



Column layout.

HBox

The `HBox` layouts arranges items horizontally across the container. This layout optionally divides available horizontal space between child items containing a numeric `flex` configuration.

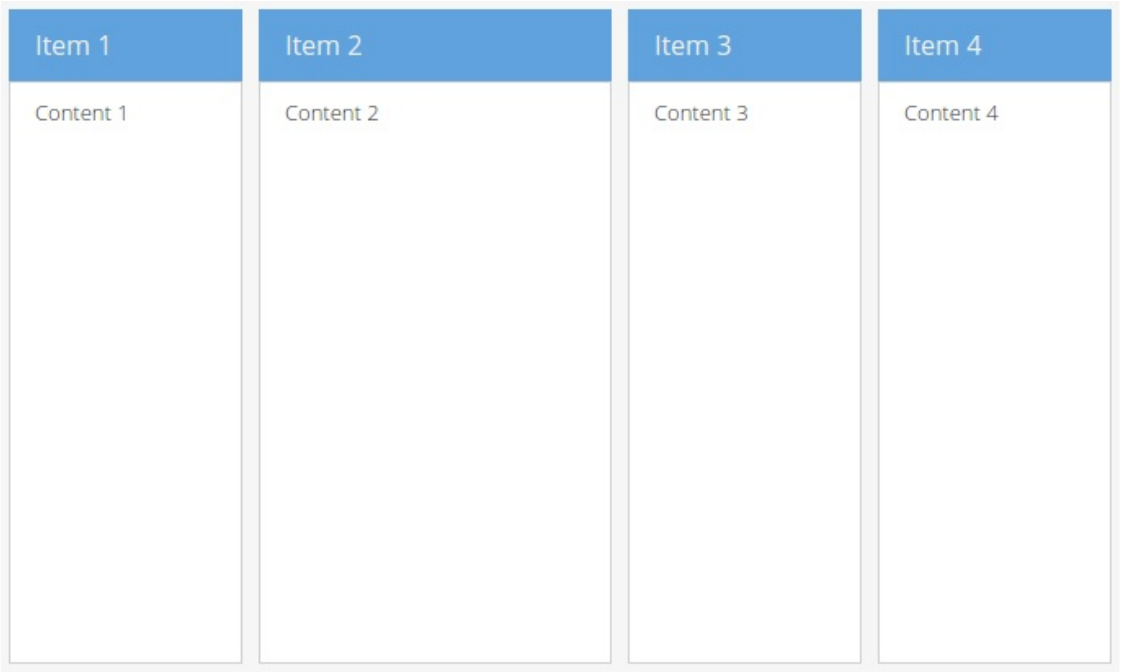
This layout may also be used to set the heights of child items by configuring it with the `align` option. Additionally you can specify how the child items of the container are packed together by setting the `pack` option.

Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:
  > layout-hbox.js

```
Ext.create('Ext.container.Viewport', {
    layout: {
        type: 'hbox',
        pack: 'start',
        align: 'stretch'
    },
    defaults: {
        bodyPadding: '10 15',
        margin: 5,
        flex: 1,
    },
    items: [{
        title: 'Item 1',
        html: 'Content 1'
    }, {
        title: 'Item 2',
        flex: 1.5,
        html: 'Content 2'
    }, {
        title: 'Item 3',
        html: 'Content 3'
    }, {
        title: 'Item 4',
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:


Item 1
Content 1
Item 2
Content 2
Item 3
Content 3
Item 4
Content 4

HBox layout.

VBox

The `VBox` layouts arranges items vertically across the container. This layout optionally divides available vertical space between child items containing a numeric `flex` configuration.

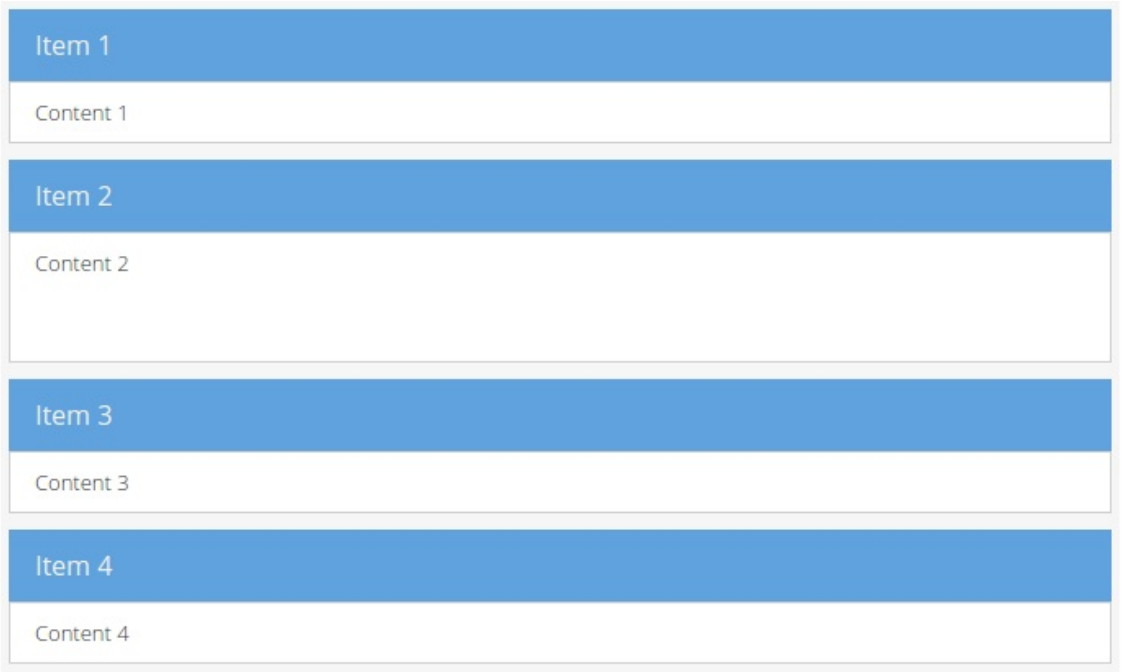
This layout may also be used to set the widths of child items by configuring it with the `align` option. Additionally you can specify how the child items of the container are packed together by setting the `pack` option.

Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:
  > layout-vbox.js

```
Ext.create('Ext.container.Viewport', {
    layout: {
        type: 'vbox',
        pack: 'start',
        align: 'stretch'
    },
    defaults: {
        bodyPadding: '10 15',
        margin: 5,
        flex: 1,
    },
    items: [{
        title: 'Item 1',
        html: 'Content 1'
    }, {
        title: 'Item 2',
        flex: 1.5,
        html: 'Content 2'
    }, {
        title: 'Item 3',
        html: 'Content 3'
    }, {
        title: 'Item 4',
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:


Item 1
Content 1
Item 2
Content 2
Item 3
Content 3
Item 4
Content 4

VBox layout.

Accordion

The `Accordion` layout is a layout that manages multiple Panels in an expandable accordion style such that by default only one panel can be expanded at any given time (set `multi` -config to have more open). Each Panel has built-in support for expanding and collapsing.

> Only panels and all subclasses of `Ext.panel.Panel` may be used in an accordion layout container.

Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:
  > layout-accordion.js

```
Ext.create('Ext.container.Viewport', {
    layout: 'accordion',
    defaults: {
        bodyPadding: 15,
        border: false
    },
    items: [{
        title: 'Item 1',
        html: 'Content 1'
    }, {
        title: 'Item 2',
        html: 'Content 2'
    }, {
        title: 'Item 3',
        html: 'Content 3'
    }, {
        title: 'Item 4',
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:

Accordion layout.

Table

The `Table` layout allows you to easily render content into an HTML table. The total number of columns can be specified, and `rowspan` and `colspan` can be used to create complex layouts within the table.

In the case of `Table` layout, the only valid layout config properties are `columns` and `tableAttrs`. However, the items added to a layout can supply the config properties `rowspan` (the number of rows that the spanned cell needs to cover), `colspan` (the number of cells that the cell should replace) and `cellCls` (a CSS class name added to the table cell containing the item).

The basic concept of building up a `Table` layout is conceptually very similar to building up a standard HTML table. You simply add each panel (or "cell") that you want to include along with any span attributes specified as the special config properties of `rowspan` and `colspan` which work exactly like their HTML counterparts. Rather than explicitly creating and nesting rows and columns as you would in HTML, you simply specify the total column count in the layout config and start adding panels in their natural order from left to right, top to bottom. The layout will automatically figure out, based on the column count, rowspans and colspans, how to position each panel within the table.

> Just like with HTML tables, your `rowspans` and `colspans` must add up correctly in your overall layout or you'll end up with missing and/or extra cells!

Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:

  > layout-table.js

```
Ext.create('Ext.container.Viewport', {
    layout: {
        type: 'table',
        columns: 3,
        tableAttrs: {
            style: {
                width: '100%'
            }
        }
    },
    defaults: {
        bodyPadding: 15,
    },
    items: [{
        title: 'Item 1',
        rowspan: 1,
        colspan: 1,
        html: 'Content 1'
    }, {
        title: 'Item 2',
        rowspan: 1,
        colspan: 1,
        html: 'Content 2'
    }, {
        title: 'Item 3',
        rowspan: 1,
        colspan: 1,
        html: 'Content 3'
    }, {
        title: 'Item 4',
        colspan: 2,
        rowspan: 1,
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:

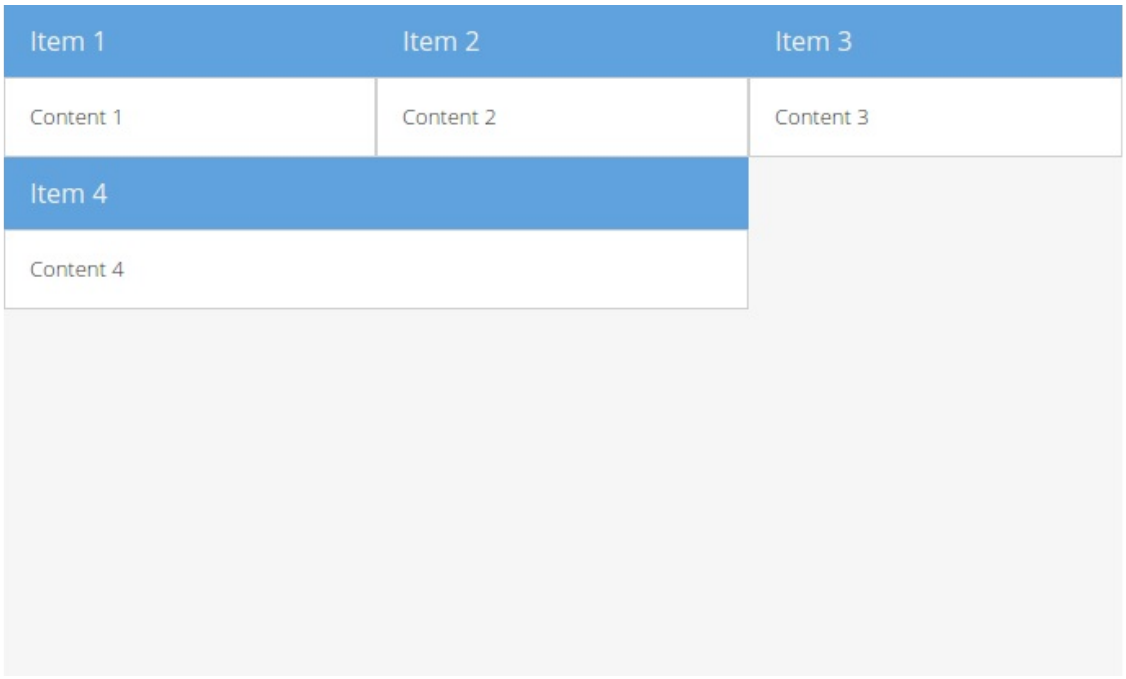


Table layout.

Border

The `Border` layout is a multi-pane, application-oriented UI layout style that supports multiple nested panels, automatic bars between `regions` and built-in expanding and collapsing of `regions`.

When using this layout note, that any container using the border layout must have a child item with `region:'center'`. This child item in the center region will always be resized to fill the remaining space not used by the other regions in the layout. Any child items with a region of `west` or `east` may be configured with either an initial `width`, `flex` or an initial percentage `width` value. Any child items with a region of `north` or `south` may be configured with either an initial `height`, `flex` value or an initial percentage `height` value.

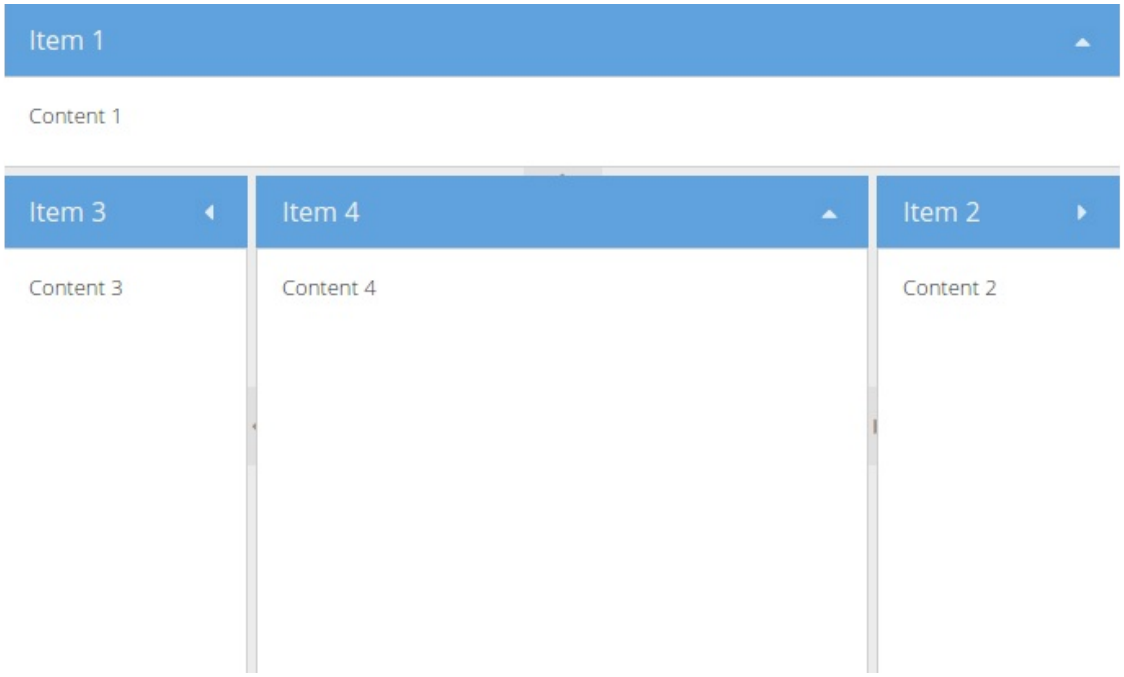
Exercise

- (Re-)open your `index.html` and update the code creating the `Ext.container.Viewport` component to match the following snippet:

  > layout-border.js

```
Ext.create('Ext.container.Viewport', {
    layout: 'border',
    defaults: {
        bodyPadding: 15,
        collapsible: true,
        split: true
    },
    items: [{
        title: 'Item 1',
        region: 'north',
        height: 100,
        html: 'Content 1'
    }, {
        title: 'Item 2',
        region: 'east',
        width: 150,
        html: 'Content 2'
    }, {
        title: 'Item 3',
        region: 'west',
        width: 150,
        html: 'Content 3'
    }, {
        title: 'Item 4',
        region: 'center',
        html: 'Content 4'
    }]
});
```

- Reload the page in the browser and take a look at the result:



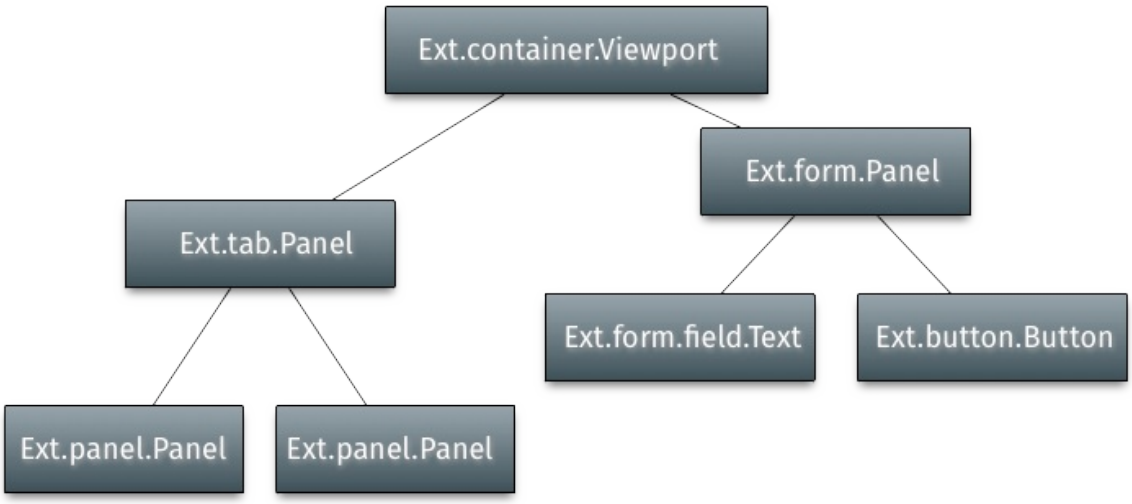
Border layout.

Components

Components are referred to the Ext JS class `Ext.Component` which is the base class for all components. Generally speaking a component itself is a predefined Ext JS compatible module composed of HTML, CSS and JavaScript. In the former exercises we already met the components `Ext.window.Window`, `Ext.container.Viewport` and `Ext.panel.Panel` (whereas the latter not explicit, but it's the default component in the `Viewport` container).

Every Component has a shorthand name called `xtype`. The xtype is especially useful if you want to render your application lazily, that means rendering your components at the time they're getting meaningful for your application, e.g. creating an error message at the time an error occurred. In the upcoming examples we'll use the `xtype` to create components.

A typical application's component hierarchy starts with a viewport at the top, which has other containers and/or components nested within it.



The component hierarchy, source: https://docs.sencha.com/extjs/6.0/core_concepts/images/component_heirarchy_5.png

In the upcoming section we're going to inspect some components that might be useful for any Ext JS application your're going to develop in the future:

> The descriptions given in the upcoming subsections are based on the API documentation.

- Panel
- Image
- Form
- Tree
- Grid

Panel

A `Panel` is a container designed for building structured blocks for application oriented user interfaces. Panels are, by their inheritance from `Ext.container.Container`, capable of being configured with a layout (see previous chapter) and containing child components. Panels also provide built-in collapsible, expandable and closable behavior and can be easily dropped into any container or layout, whereas the layout and rendering is completely managed by the framework.

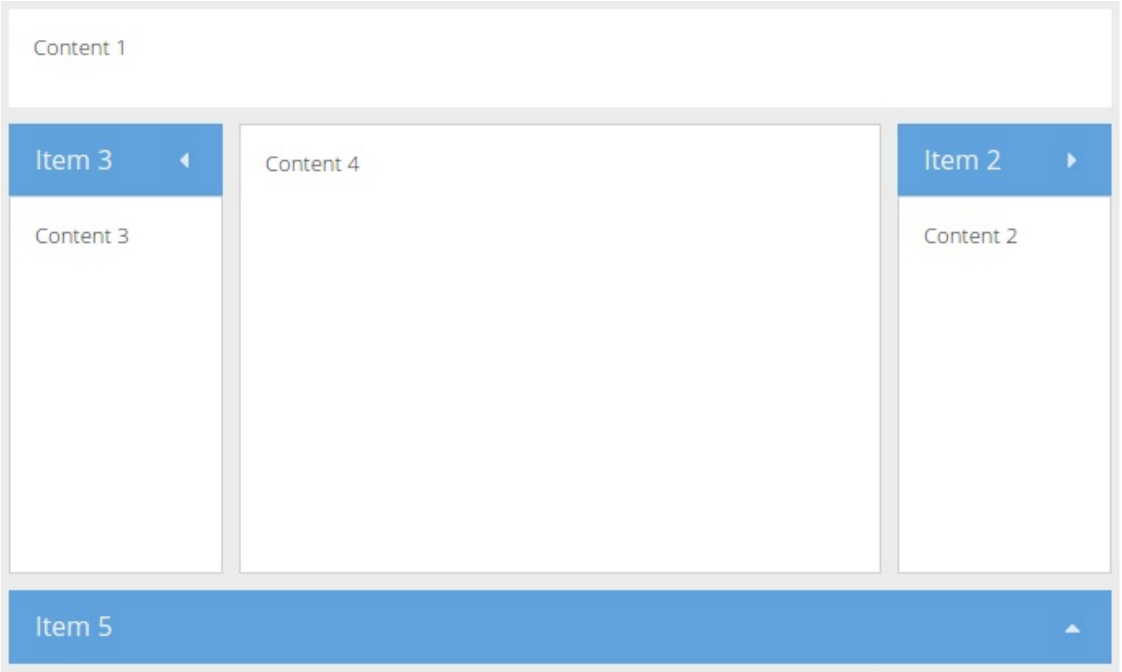
In most applications the panel is one of the most often used components. In the next exercise we'll extent our existing viewport by only a few configurations and will see, that we have worked with panels (even we haven't specified it) yet.

Exercise

- (Re-)open your `index.html` and extend the `Ext.container.Viewport` to match the following snippet:

  component-panel.js

```
Ext.create('Ext.container.Viewport', {
    layout: 'border',
    defaults: {
        xtype: 'panel',
        bodyPadding: 15,
        collapsible: true,
        split: false,
        margin: 5
    },
    items: [{
        region: 'north',
        collapsible: false,
        height: 60,
        border: false,
        html: 'Content 1'
    }, {
        title: 'Item 2',
        region: 'east',
        width: '20%',
        html: 'Content 2'
    }, {
        title: 'Item 3',
        region: 'west',
        width: '20%',
        html: 'Content 3'
    }, {
        region: 'center',
        collapsible: false,
        html: 'Content 4'
    }, {
        title: 'Item 5',
        region: 'south',
        maxHeight: 350,
        collapsed: true,
        html: 'Content 5'
    }]
});
```



Advanced Border layout.

As you may notice, it's hardly to spot any viewable difference to our previous example. But have a look at the `defaults` attribute set in the viewport. It contains a new key named `xtype` (remember: it's the shorthand name for a component) with the value `panel`. Thus every direct child in the viewport will be instantiated as a `panel`.

Advanced panel configuration

A Panel may also contain bottom and top toolbars, along with separate header, footer and body sections.

Exercise

- (Re-)open your `index.html` and extend the panel rendered in the viewports center by the following snippet:
  > component-panel-toolbar.js

```
tbar: [{
    xtype: 'button',
    text: 'Button 1',
    iconCls: 'fa fa-repeat'
}]
```

- Reload the page in the browser and take a look at the result:

Panel toolbar.

Nested components

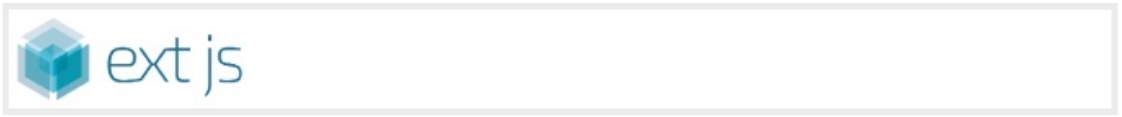
The `Ext.Img` component can be used to insert an image into the Ext JS handled lifecycle. For example the class makes it easy to change the source of the image container.

Exercise

- (Re-)open your `index.html` and extend the `Ext.container.Viewport` items by the following snippet:
  > component-image.js

```
{
    region: 'north',
    collapsible: false,
    height: 60,
    border: false,
    bodyPadding: 5,
    items: [{
        xtype: 'image',
        src: './materials/ext-logo.png',
        height: 50
    }]
}
```

- Reload the page in the browser and take a look at the result:



Nested component image in a panel.

Form Fields

The `Ext.form.Panel` presents a subclass of the panel and is especially useful for building user interaction web forms and for saving and loading remote data. Usually you combine a form panel with subclasses inherited from the `Ext.field.Field` class. In the following example we'll get to know some of the most important fields one would use in a form (listed with xtypes and links to the API documentation):

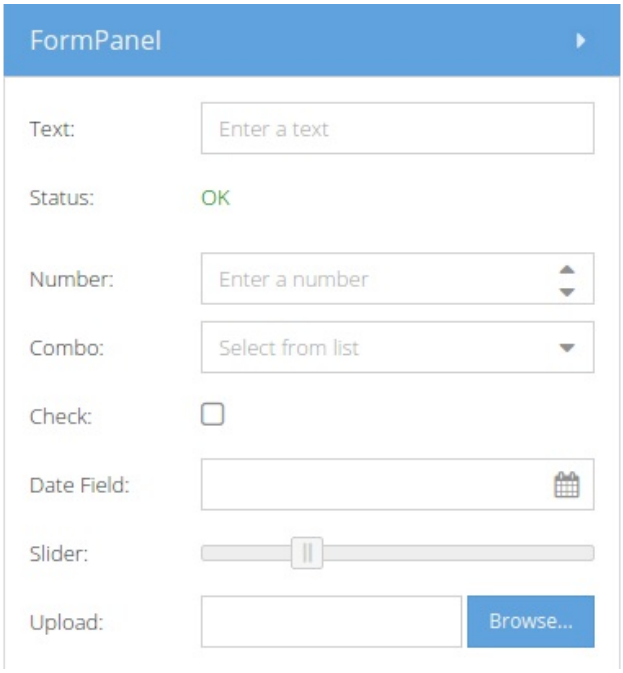
- textfield
- displayfield
- numberfield
- combobox
- checkbox
- datepicker
- slider
- filefield
- button

Exercise

- (Re-)open your `index.html` and extend the panel in the viewport's east region by the following snippet:
  > component-form-fields.js

```
{
    xtype: 'form',
    title: 'FormPanel',
    region: 'east',
    width: '20%',
    autoScroll: true,
    defaults: {
        anchor: '100%'
    },
    items: [{
        xtype: 'textfield',
        name: 'text',
        fieldLabel: 'Text',
        emptyText: 'Enter a text'
    }, {
        xtype: 'displayfield',
        name: 'status',
        fieldLabel: 'Status',
        value: '<span style="color:green;">OK</span>'
    }, {
        xtype: 'numberfield',
        name: 'number',
        fieldLabel: 'Number',
        emptyText: 'Enter a number',
        minValue: 0,
        maxValue: 99
    }, {
        xtype: 'combo',
        name: 'combo',
        fieldLabel: 'Combo',
        emptyText: 'Select from list',
        minValue: 0,
        maxValue: 99,
        store: [
            'Entry 1',
            'Entry 2',
            'Entry 3'
        ]
    }, {
        xtype: 'checkbox',
        name: 'check',
        fieldLabel: 'Check'
    }, {
        xtype: 'datefield',
        name: 'dateField',
        fieldLabel: 'Date Field'
    }, {
        xtype: 'slider',
        name: 'slider',
        fieldLabel: 'Slider',
        minValue: 0,
        maxValue: 100,
        value: 25
    }, {
        xtype: 'filefield',
        name: 'upload',
        fieldLabel: 'Upload'
    }]
}
```

- Reload the page in the browser and take a look at the result:



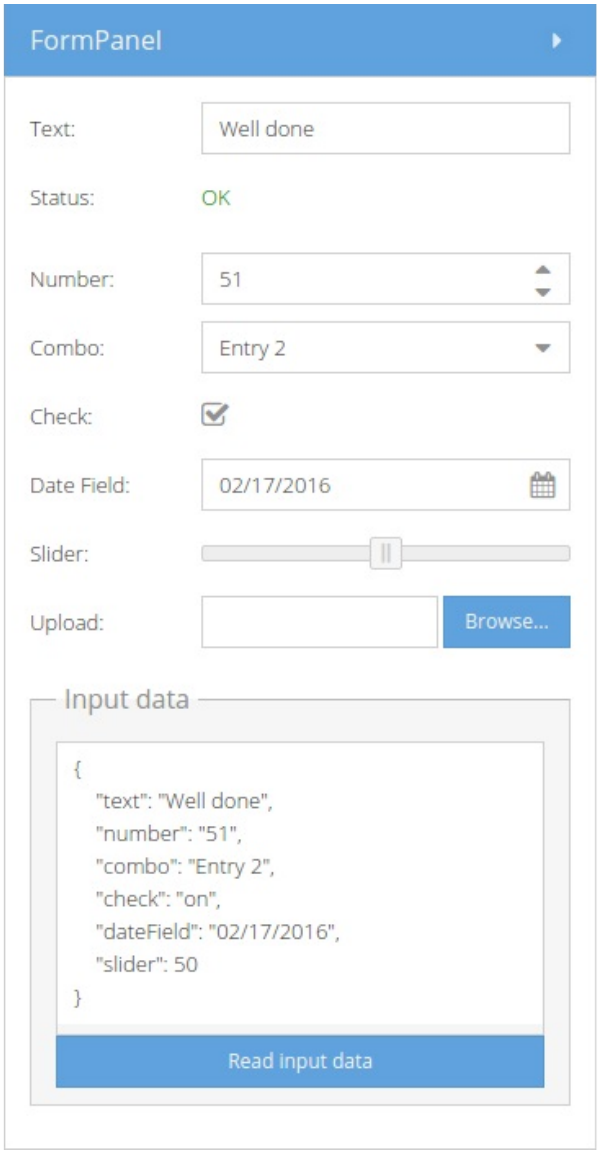
Nested component image in a panel.

As stated above, the form is very useful if you want to systematically read out values given by the user and to work with them afterwards, e.g. sending the values to a server endpoint. In the next example we're going to create another useful form component, the `Ext.form.FieldSet` class. The fieldset is a specialized container for grouping fields. We we'll now create a fieldset with a textarea and a button (ignore the `handler` method for the moment, we'll explain events and component querying later on).

- Add the following fieldset to the lower end of the form field we declared above:

```
{
  xtype: 'fieldset',
  title: 'Input data',
  layout: 'fit',
  items: [{
      xtype: 'textarea',
      height: 180,
      isFormField: false
  }, {
      xtype: 'button',
      text: 'Read input data',
      handler: function(btn) {
          var form = btn.up('form'),
              textArea = form.down('textarea');
          textArea.setValue(
              JSON.stringify(
                  form.getValues(), null, 4
              )
          );
      }
  }]
}
```

- Reload the page in the browser, enter some custom values in the form field and press the button `Read input data`:



Nested component image in a panel.

Trees

The `Ext.tree.Panel` class provides a tree-structured UI representation of tree-structured data. A treepanel must be bound to a `Ext.data.TreeStore` (the Ext JS data package including stores will be handled in the next chapter).

Exercise

- (Re-)open your `index.html` and extend the `Ext.container.Viewport` items by the following snippet:

  > component-treepanel.js

```
{
    xtype: 'treepanel',
    width: '20%',
    bodyPadding: 0,
    title: 'TreePanel',
    region: 'west',
    rootVisible: false,
    store: {
        data: {
            text: 'Root',
            children: [{
                text: 'Child 1',
                leaf: true
            }, {
                text: 'Child 2',
                leaf: true
            }, {
                text: 'Child 3',
                leaf: true
            }, {
                text: 'Child 4',
                children: [{
                    text: 'GrandChild 1',
                    leaf: true
                }, {
                    text: 'GrandChild 2',
                    leaf: true
                }]
            }]
        }
    }
}
```

- Reload the page in the browser and take a look at the result:

Treepanel with dummy items.

Grid

The `Ext.grid.Panel` class is of avail for showing up (large amounts) of tabular data. The required properties of a gridpanel are a store and a column definition.

Exercise

- (Re-)open your `index.html` and extend the `Ext.container.Viewport` items by the following snippet:

  > component-grid.js

```
{
    xtype: 'gridpanel',
    title: 'GridPanel',
    region: 'south',
    bodyPadding: 0,
    maxHeight: 350,
    collapsed: true,
    columns: [{
        text: 'First name',
        dataIndex: 'firstName',
        flex: 1
    }, {
        text: 'Last name',
        dataIndex: 'lastName',
        flex: 1
    }, {
        text: 'Instruments',
        dataIndex: 'instruments',
        flex: 1
    }],
    store: {
        data: [{
            firstName: 'Angus',
            lastName: 'Young',
            instruments: 'Guitar'
        }, {
            firstName: 'Cliff',
            lastName: 'Williams',
            instruments: 'Bass guitar, vocals'
        }, {
            firstName: 'Brian',
            lastName: 'Johnson',
            instruments: 'Vocals'
        }, {
            firstName: 'Stevie',
            lastName: 'Young',
            instruments: 'Guitar, vocals'
        }, {
            firstName: 'Chris',
            lastName: 'Slade',
            instruments: 'Drums, percussion'
        }]
    }
}
```

- Reload the page in the browser and take a look at the result:

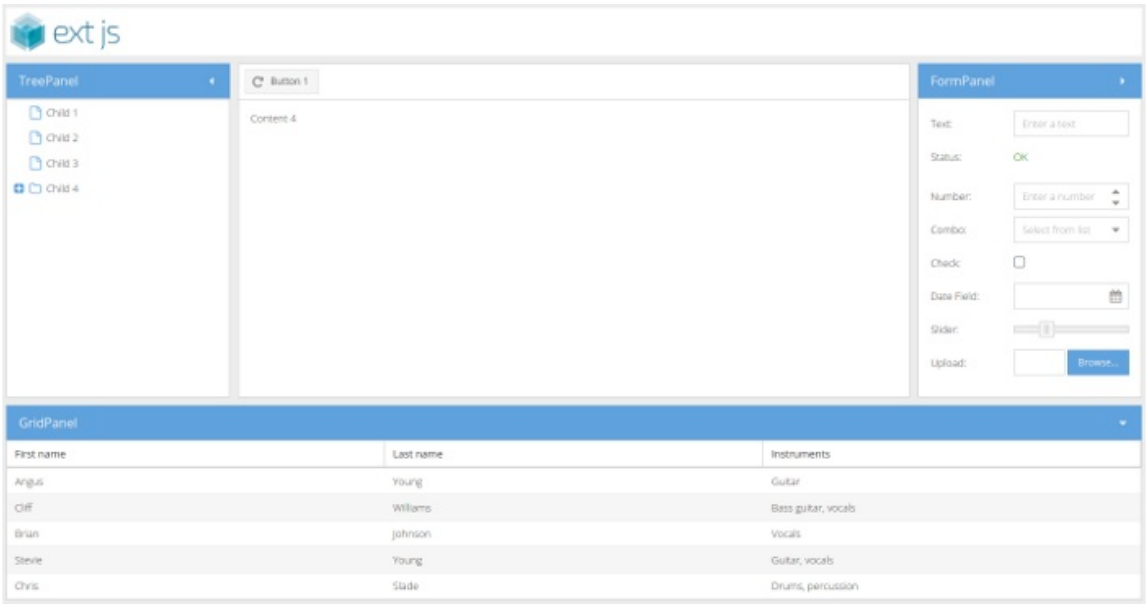
GridPanel

First name	Last name	Instruments
Angus	Young	Guitar
Cliff	Williams	Bass guitar, vocals
Brian	Johnson	Vocals
Stevie	Young	Guitar, vocals
Chris	Slade	Drums, percussion

Gridpanel with some inline data.

Final output

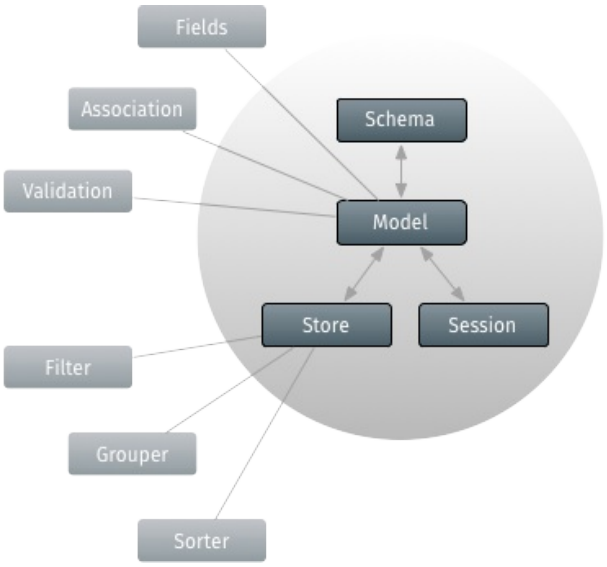
Finally your Ext JS application should look similar to this:



Final application layout.

Data

Now we've became acquainted with one of most important Ext JS visual components, we're going to learn how one could load remote data into the existing webapplication *without* reloading the page itself. For example this might be interested for you if you want to implement a paging functionality to the grid. In this context we'll get in touch with the Ext JS data package that is responsible for loading (and saving) all of the data in the application. The package consists of multiple classes, but there are three that are more important than all the others: The `Ext.data.Model`, `Ext.data.Store` and `Ext.data.proxy.Proxy` (sub-)classes.



The Ext JS data package, source: http://docs.sencha.com/extjs/6.0/core_concepts/images/data-model.png

In the forthcoming exercises we're going to recreate a gridpanel in the center of our border layout that'll contain an `Ext.data.Store` reading remote data with the use of an `Ext.data.proxy.Ajax` proxy. The store will be associated with an `Ext.data.Model`.

Preparation

Let's start this section by creating another gridpanel in the center of our border layout. Here we're going to replace the existing the panel with the gridpanel by keeping the toolbar. The toolbar (and its button) will be needed in the next module. The `columns` definition will stay unaffected in comparison to the gridpanel we created in the recent exercise.

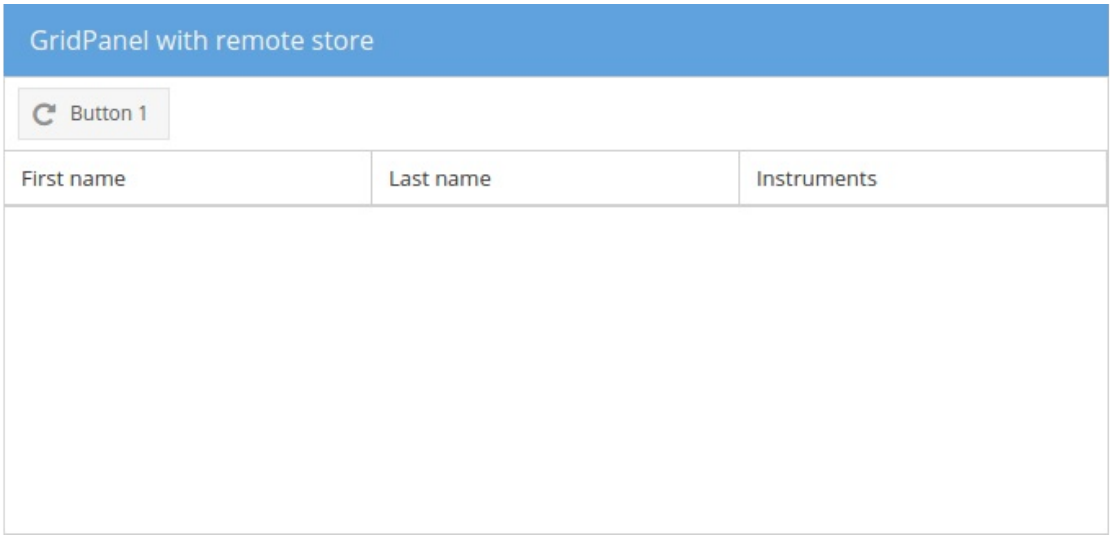
Exercise

- (Re-)open your `index.html` and find the declaration of the center region.
- Replace the panel in the border layouts center with a grid, but leave the toolbar and set the store to null (for the moment).

  > data-grid.js

```
{
    xtype: 'gridpanel',
    title: 'GridPanel with remote store',
    region: 'center',
    collapsible: false,
    bodyPadding: 0,
    columns: [{
        text: 'First name',
        dataIndex: 'firstName',
        flex: 1
    }, {
        text: 'Last name',
        dataIndex: 'lastName',
        flex: 1
    }, {
        text: 'Instruments',
        dataIndex: 'instruments',
        flex: 1
    }],
    store: null,
    tbar: [{
        xtype: 'button',
        text: 'Button 1',
        iconCls: 'fa fa-repeat'
    }]
}
```

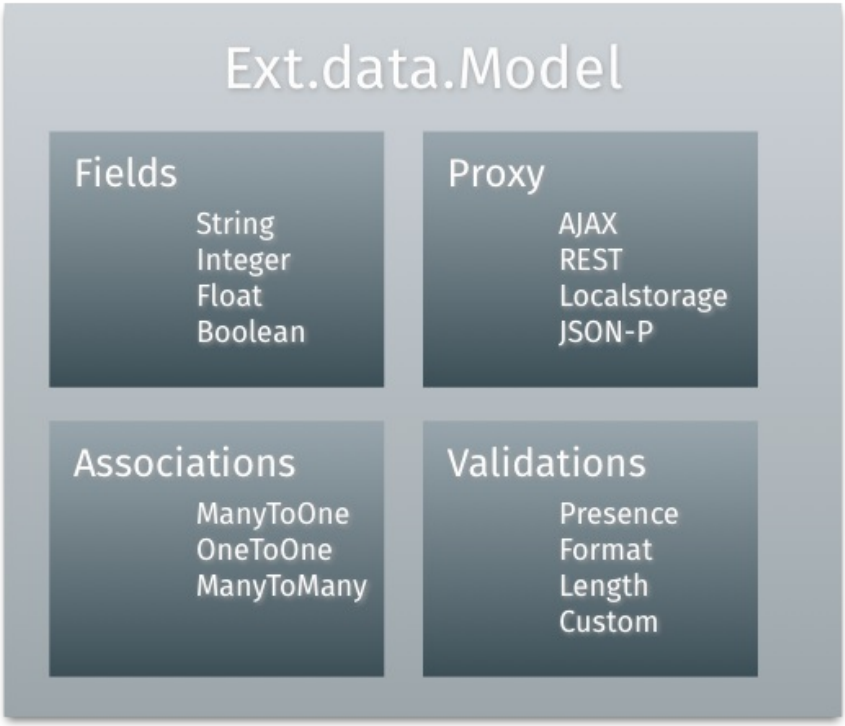
- Reload the page in the browser and verify the empty gridpanel in the layouts center:



The new gridpanel.

Model

The core of the data package is the `Ext.data.Model` class. A Model or Entity represents some object that your application manages, e.g. the (former) members of a rock band. Models are used by stores, which are in turn used by many of the data-bound components in Ext JS. The most significant parts (or properties) of a model are `Fields` (they handle the members of a model), `Proxies` (they handle the loading and saving of model data), `Validations` (they handle validation of the data, e.g. if a field has not-null value) and `Associations` (they handle the relations and linkages to other model instances).



Parts of the Ext JS model class, source: http://docs.sencha.com/extjs/6.0/core_concepts/images/model-breakdown.png

In this exercise we'll build up a simple model, that'll contain some (string) fields and simple validation for input data (ensuring that all fields have a value). As we only have this single model we don't want to model any associations. Please refer to the API documentation for further details. You both have the possibility to assign the proxy in the model or the store (using that model). Both ways do have advantages depending on your application setup: If you set the proxy in the model it allows you to load and save instances of this model without the need of a store and multiple stores could use the same model. In contrast defining the proxy in the store it allows you to use the same data model in multiple stores, even if the stores will load their data from different sources. In this exercise we're going to set the proxy in the store (without any specific reason).

Exercise

- (Re-)open your `index.html` and insert the following code *before* the instantiation of the viewport (line ~15) to create a new model called `FormerMembers`:

  > data-model.js

```
Ext.define('FormerMembers', {
    extend: 'Ext.data.Model',
    fields: [{
        name: 'firstName',
        type: 'string'
    }, {
        name: 'lastName',
        type: 'string'
    }, {
        name: 'instruments',
        type: 'string'
    }],
    validators: {
        firstName: 'presence',
        lastName: 'presence',
        instruments: 'presence'
    }
});
```

Proxy and store

A store takes care of the client side caching of model objects and can be configured to load data via a proxy. They provide different functions for accessing the underlying model instances (e.g. sorting, filtering or querying). To load and save instances a store uses a proxy. Generally speaking the data source can be either local (client proxy) or remote (server proxy), whereas the client proxies load/save their data locally and the server proxies load/save their data by sending requests to a remote server.

In the following exercise we'll create a new `Ext.data.Store` reading data with a `Ext.data.proxy.Ajax` from a remote source (The remote source is the python server our simple webapplication lives on, but be aware that it could be any other remote source).

Exercise

- (Re-)open your `index.html` and insert the following code *after* the instantiation of the model `FormerMembers` (line ~34) to create the store.

  > data-store.js

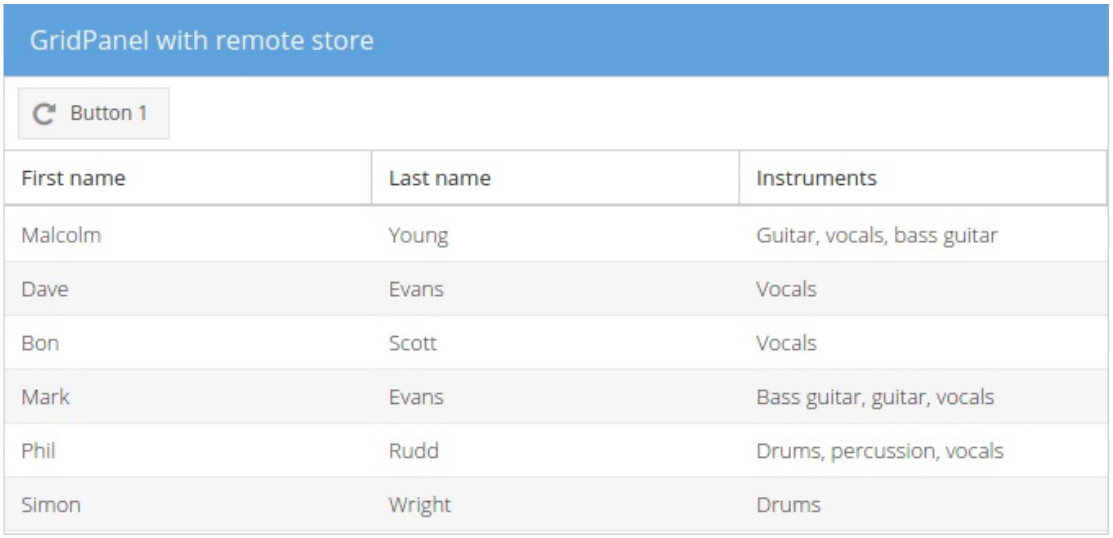
```
Ext.create('Ext.data.Store', {
    autoLoad: true,
    storeId: 'formerMembers',
    model: 'FormerMembers',
    proxy: {
        type: 'ajax',
        url: './materials/former-members.json',
        reader: {
            type: 'json',
            rootProperty: 'data'
        }
    }
});
```

If you would reload the page now, you wouldn't be able to see any changes in the centered gridpanel as the newly created store isn't regarded as being existent to the grid.

- Find the declaration of the gridpanel and update the `store` property to:

```
   store: 'formerMembers'
```

- Finally reload the page and view the results.



Grid with remote data loaded.

Events

In Ext JS `events` are signals from a class that are `fired` if anything happened to a class. Events are fired globally in the Ext namespace so that every class can `listen` to those. In our Ext JS application we can use these events to code specific reponses that will be executed if a certain event is being fired. Nearly all Ext JS components and classes fire different kinds of events at their lifecycle. For example, each class inherited from `Ext.Component` fires the event `added` after the component had been added to a container. We can listen for that event by configuring a simple `listeners` object.

In this section we'll get to know three events fired by different components to get a basic idea about the potential behind `events` and `listeners`.

- Event click
- Event afterrender
- Event change

Event `click`

The `click` event is being fired when e.g. a button or a menu entry is clicked.

In the following exercise we'll use the `click` event of the button rendered to the toolbar in the centered panel to load the remote data if the button is clicked by the user.

Exercise

- (Re-)open your `index.html` and find the instantiation of the store `formerMembers` (line ~29) we introduced in the former module.
- Set `autoLoad: false` in the store.

After reloading the page you should notice that the grid contains no data anymore. Can you explain why?

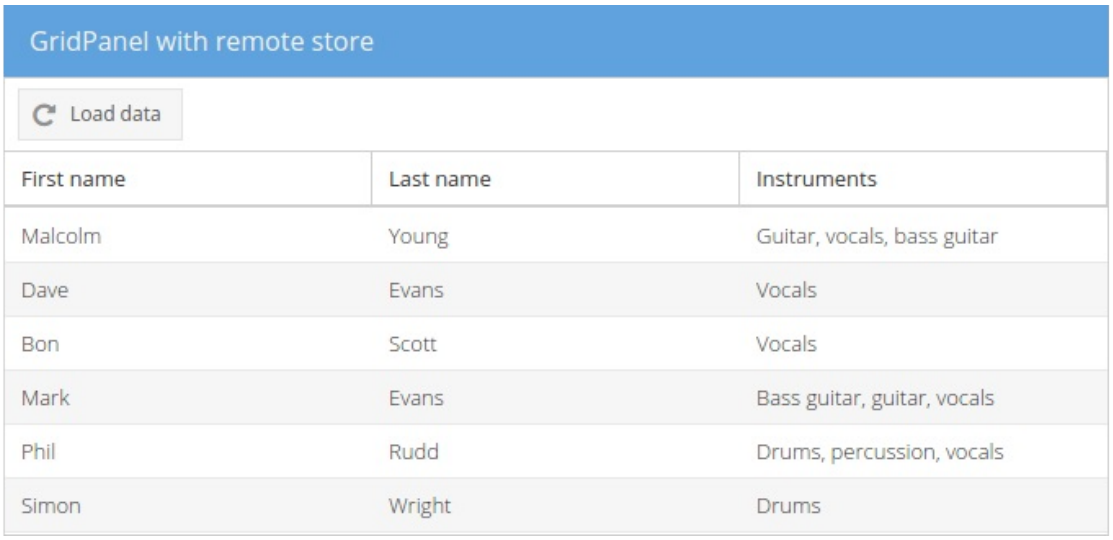
In the next few steps we're going to fetch back our missing data by reusing the already existing button in the toolbar as a `Load data` button:

- Find the button declaration within the panel in the center region and rename it accordingly (`text: 'Load data'`).
- Register a new listener to the `click` -event and pass an anonymous function to it. This function will be called if the `click` event is fired by the button class:

  > event-click.js

```
listeners: {
    click: function(btn) {
        var gridpanel = btn.up('gridpanel');
        gridpanel.getStore().load();
    }
}
```

- Again, reload the page in the browser and click the updated `Load data` button.



Load remote data `onClick`.

Dissecting the example

Let's have a more detailed look at the function we passed to the `click` listener:

```
function(btn) {...}
```

Every event can be fired with optional arguments passed to the listener. Here, our anonymous handler function receives the argument `btn`, whereat the variable `bt` holds a reference to the button instance firing the `click` event ("the clicked button").

```
var gridpanel = this.up('gridpanel');
```

Remember, we are dealing with hierarchically structured components. Ext JS (internally) registers all instantiated components in its `Ext.ComponentManager`. Within this manager we can navigate across and search the application component composition. The explanation of the manager and the corresponding `Ext.ComponentQuery` singleton is far beyond the goals of this workshop, but it's very recommended to have a look at the very detailed documentation. Long story short: Each component provides us the method `up()` and each container the methods `up()`, `down()` and `query()` to simply navigate across the component hierarchy by the use of very simple filter expressions. The easiest way one can think of is build a filter that returns *the first xtype in the lower/upper hierarchy level* of a given component. Having this in mind, the upper method will return the first component of xtype `gridpanel` in the upper direction based on the pressed button.

```
gridpanel.getStore().load();
```

Now we got the gridpanel, we can access the underlying store by using the `getStore()` method and directly execute the method `load()` to load any local or remote data associated with the store.

Event `afterrender`

The `afterrender` event is being fired after a component is finally rendered (to the DOM) and is very often used if you want to make sure, your listener function is called *after* the component is rendered.

In the following exercise we'll register another `listener` to the button `Load data` that will show up a minimalistic message box (`Ext.toast`) to the user after the button has been rendered.

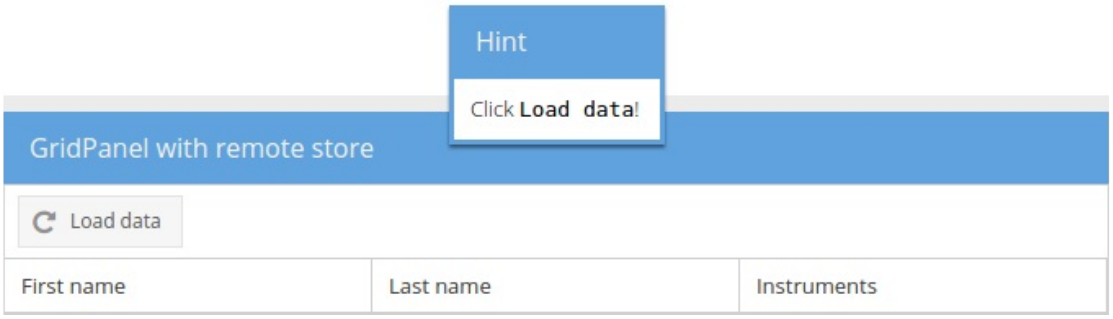
Exercise

- (Re-)open your `index.html` and find the button declaration within the panel in the center region and we used in the former exercise.
- Register a listener for the event `afterrender` by appending the following code block to its `listeners` array:

  event-afterrender.js

```
afterrender: function(cmp) {
    Ext.toast({
        html: 'Click <code>Load data</code>!',
        title: 'Hint',
        align: 't',
    });
}
```

- And again, reload the page in the browser and you will see the toast.



A simple Ext.toast.

Dissecting the example

Let's have a more detailed look at the function we passed to the `afterrender` listener:

```
function(cmp) {...}
```

As already illustrated, events can hold extra arguments which will be received in the listener functions and similiar to the `click` event, the `afterrender` event passes a reference to the rendered component to the anonymous function. The variable `cmp` therefore refers to the button itself (but is not used in our function).

```
Ext.toast({
  html: 'Click <code>Load data</code>!',
  title: 'Hint',
  align: 't',
});
```

The `Ext.toast` class provides a lightweight, auto-dismissing pop-up notifications and is configurable by the use of a configuration object. Here we set both the `html` and `title` keys for a simple message and title as well as the `align` key for specifing the alignment of the toast message to the top of its anchor (the viewport).

Event `change`

The `change` event is being fired when e.g. a file input field's value has changed.

In this (final) exercise we'll add a new textfield to the gridpanels toolbar involving a filter function that is called on every change made by the user to the textfield.

Exercise

- (Re-)open your `index.html` and find the toolbar declaration inside the gridpanel rendered to the center region.

At first we will make use of the class `Ext.toolbar.Fill` to add a non-rendering placeholder item to the toolbar whereby all following items will be aligned to the right of the toolbar. (Other useful toolbar items you may interested in are `tbseparator` and `tbspacer`.)

- Add the placeholder to the end of the toolbar by inserting the following declaration:

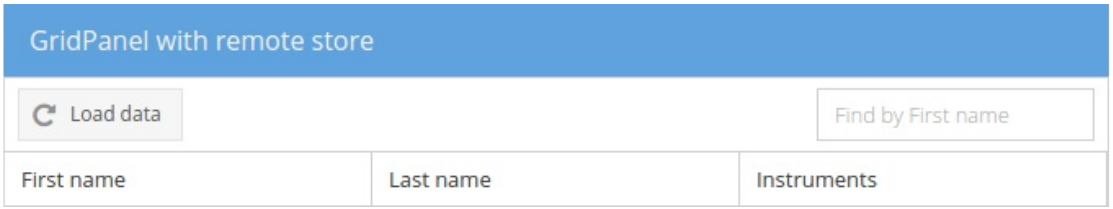
```
{
  xtype: 'tbfill'
}
```

- Next we'll add a textfield with the `change` listener to the toolbar:

> event-change.js

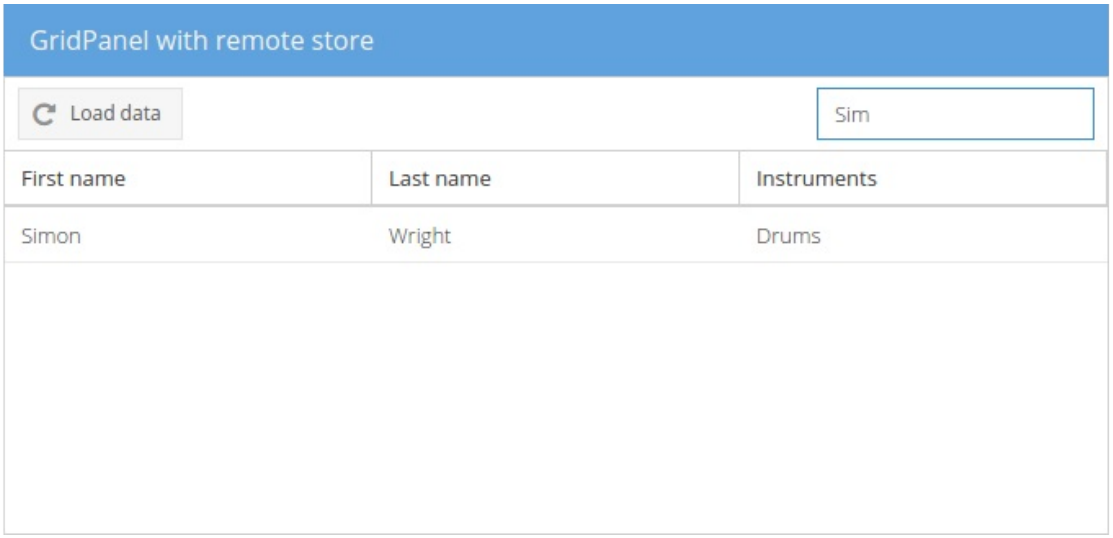
```
{
    xtype: 'textfield',
    emptyText: 'Find by First name',
    listeners: {
        change: function(field, newValue, oldValue) {
            field.up('gridpanel').getStore().filter({
                property: 'firstName',
                value: newValue || '',
                anyMatch: true,
                caseSensitive: false
            });
        }
    }
}
```

- Reload the page in the browser and you should notice a new textfield in the upper right of the centered gridpanel.



Filter textfield.

- Try out the newly created filter by loading data into the grid and changing the textfields value.



Filter textfield.

Dissecting the example

Let's have a more detailed look at the function we passed to the `change` listener:

```
function(field, newValue, oldValue) {...}
```

If the value of a field is changed, our anonymous function is being called with passed arguments `field`, `newValue` and `oldValue`. `field` holds a reference to the textfield itself, `newValue` the changed/prevailing and `oldValue` the original/preceding value.

```
field.up('gridpanel').getStore()
```

Again, we use the methods `up()` to get a reference to the gridpanel and `getStore()` to get the associated store (see event click for the detailed information).

```
.filter({...})
```

With the store in hand we can access all methods provided by the (instantiated) `Ext.data.Store` class. As we want to filter the store by a particular value given in the textfield, we can make use of the method `filter()`. This method filters the data in the store by one or more fields and can be configured with a detailed filter configuration (from `Ext.util.Filter` class).

```
{
  property: 'firstName',
  value: newValue || '',
  anyMatch: true,
  caseSensitive: false
}
```

The given object represents a filter that is applied to the `filter()` function and is defined to filter the `property` (that is the field in the model to filter on) `'firstName'`. The `value` to filter with is the `newValue` given by the event or an empty string (`''`) if the passed value is falsy (`false`, `0`, `''` `null`, `undefined` or `NaN`). Setting `anyMatch` to `true` configures the filter to match the value characters at any position in the store's value and by having `caseSensitive` set to `false` we ignore exact case matching.

GeoExt3 workshop

Welcome to the GeoExt3 workshop, in which you'll learn how to use GeoExt3 in your ExtJS applications.

This workshop is composed out of several modules, which usually are done in order.

- Before diving into programming, the metainformation chapter has a lot of information about this workshop, the intended audience and how to best create solutions for the workshop.
- Afterwards we'll work through the first steps chapter, in which you learn about Openlayers, ExtJS and GeoExt3.
- Now that we know these libraries and frameworks, we are ready to use the `GeoExt.component.Map` in the map chapter.
- Your map may very well contain a lot of layers, these can be managed with a layer tree.
- The next chapter is all about vector features, you'll create a grid that is synced with a vector layer in the map.
- The last chapter shows other aspects of GeoExt components like popups, an embeddable overview map, the utility classes to talk to MapFish print servers and other parts of GeoExt.

Metainformation

In this chapter we'll provide you with some metainformation about this workshop.

In order to efficiently work through the workshop, you are advised to read the following parts:

- General information about the workshop
- The target audience of the workshop
- The goals we try to achieve in the workshop
- How to setup your development environment so that you can work on the workshop tasks efficiently
- Some final notes (e.g. about the chosen structure)

Let's start with some general information about the workshop.

About

Authors

This workshop was created by the following individuals:

- Marc Jansen
- Daniel Koch

Contribute

The workshop repository can be found at https://github.com/geoext/geoext3-ws.

We look forward to external contributors. If you found an issue or have an idea of how to improve the workshop, just open an issue here.

Used libraries

During the workshop you will work with the following JavaScript libraries or frameworks:

- OpenLayers (v3.13.0): http://openlayers.org/
- ExtJS (v6.0.0, GPL): https://www.sencha.com/products/extjs/, download
- GeoExt3 (c326b01c pre-v3.0.0): http://geoext.github.io/geoext3/

Copyright

License

Target audience

This workshop is targeted at developers that want to try out the GeoExt library.

GeoExt is based on OpenLayers and ExtJS, so a bit of background in these JavaScript libraries / frameworks doesn't hurt. It is not necessarily needed to have a deep understanding of the base libraries, though. All examples or tasks usually highlight the key aspects that make everything work together.

Some basic familiarity with JavaScript is assumed, but again, we will not dive to deep into the language, so basically any interested person will be able to understand what is going on.

In order to accomplish everything in the workshop, some OGC services (such as WMS and WFS) will be used. We should provide you with enough written background so you'll grasp the core of the tasks.

Still unsure if you can work through this material?

We strive to make this workshop as understandable as possible, so please try it out! If you fail or experience problems, just tell us so: We are really looking forward to getting feedback.

Goals

These are the goals we want to reach with this workshop:

- Learn the basics of GeoExt
  - theoretically
  - practically
- Learn how to work with the API-docs
- Learn about other places where to find help for GeoExt
- Get to know certain core classes of GeoExt
- Iteratively create an application
- Learn how to debug in case of errors

Development environment

Required software

In order to complete this workshop, you will need the following software:

- A text editor, for example Atom or some other editor in which you feel comfortable.
- A browser, to read the workshop instructions and open up the tasks you will have to accomplish.
- Node.JS, so that you can run the workshop examples. Node.js will also install `npm`, which we will use to install workshop dependencies and to serve the workshop slides as HTML. If you are on linux, we have made excellent experiences with `nvm` to install various versions of Node.js.

Preparation steps

- Download the latest workshop-contens from this URL:
  - https://github.com/geoext/geoext3-ws/archive/master.zip
- Extract the zip-archive into a directory of your choice.
- You should find the following files and directories in the `geoext3-ws-master`-folder:
  - `LICENSE.md`
  - `package.json`
  - `README.md`
  - `src/`
- Install the dependencies of the workshop via `npm install`.
- Here are some example steps for a Linux-system:

```
# create a directory gx-ws …
mkdir -p ~/gx-ws
# … go there …
cd ~/gx-ws
# … grab the zip-archive …
wget https://github.com/geoext/geoext3-ws/archive/master.zip
# … unzip the archive …
unzip master.zip
# … change into the extracted folder
cd geoext3-ws-master
# Install dependencies via npm
npm install
```

Starting the workshop

- Issue the following command in the directory `geoext3-ws-master` from above:

```
npm start
```

- This should give you an output like below:

```
Live reload server started on port: 35729
Press CTRL+C to quit ...

info: loading book configuration....OK
info: load plugin gitbook-plugin-image-captions ....OK
info: load plugin gitbook-plugin-highlight ....OK
info: load plugin gitbook-plugin-search ....OK
info: load plugin gitbook-plugin-sharing ....OK
info: load plugin gitbook-plugin-fontsettings ....OK
info: load plugin gitbook-plugin-livereload ....OK
info: >> 6 plugins loaded
info: start generation with website generator
info: clean website generatorOK
info: generation is finished

Starting server ...
Serving book on http://localhost:4000
```

- If instead you see some error like below, the workshop is likely already running on your system or some other application is blocking the ports `35729` and `4000`:

```
... Uhoh. Got error listen EADDRINUSE :::35729 ...
Error: listen EADDRINUSE :::35729
  at Object.exports._errnoException (util.js:870:11)
  at exports._exceptionWithHostPort (util.js:893:20)
  at Server._listen2 (net.js:1237:14)
  at listen (net.js:1273:10)
  at Server.listen (net.js:1369:5)
  at Server.listen (/home/jansen/.gitbook/versions/2.6.7/node_modules/tiny-lr/lib/server.js:164:15)
  at Promise.apply (/home/jansen/.gitbook/versions/2.6.7/node_modules/q/q.js:1078:26)
  at Promise.promise.promiseDispatch (/home/jansen/.gitbook/versions/2.6.7/node_modules/q/q.js:741:41)
  at /home/jansen/.gitbook/versions/2.6.7/node_modules/q/q.js:1304:14
  at flush (/home/jansen/.gitbook/versions/2.6.7/node_modules/q/q.js:108:17)

You already have a server listening on 35729
You should stop it and try again.
```

Stopping the workshop

- Simply hit `Ctrl-C` in the terminal where you started the workshop, e.g. `~/gx-ws/geoext3-ws-master`.

Notes

In this section we want to keep certain notes about this workshop. It is **not a necessary requirement** to read this section if you just start with GeoExt.

In case you have questions about *why* we have structured the workshop as we have done it, please continue reading.

Q: Why not as Ext.application? Why no MVC?

> Why didn't you create the examples as `Ext.application()`? An why aren't you using the MVC pattern?

We recommend the use of `Ext.application` / `Ext.app.Application` and the use of the MVC or MVVM pattern for real-world applications. For this introductory workshop we think this would complicate stuff more than necessary.

Q: Why not with help of `Sencha Cmd`?

> Why don't you use the `Sencha Cmd` for the workshop?

We use the `sencha` tool quite often in our daily work, but found that the additional burden of installation steps would be distracting for the main focus of this workshop.

First steps

Now that we know all the required metainformation and have set up our development environment, it is time to get started.

In this chapter we will learn 5 things:

1. Where to save our exercise HTML files.
2. How to include OpenLayers in our exercise.
3. How to include ExtJS in our exercise.
4. How to include GeoExt in our exercise.
5. Where to look for more documentation.

Hello exercise

Throughout this workshop, you will encounter various tasks, that you should accomplish. Most of the time you will be asked to edit an HTML or JavaScript-file and see if the result is as intended.

In order to have comparable results, you are advised to save your HTML and any additional files inside of the `src/exercise/` -folder. If you followed the instructions for setting up the development environment, this folder will be located at:

`~/gx-ws/geoext3-ws-master/src/exercise`.

If you e.g. store a file named `map.html` inside this directory, and you are serving the workshop as recommended, than this file can be accessed via the following URL:

/map.html

Shall we tackle our first tiny excercise? Ok then, here we go:

Exercises

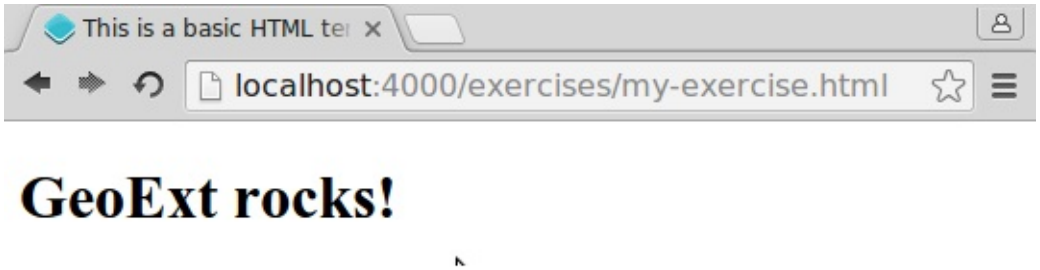
- Create a `my-exercise.html` HTML-file in the `src/exercise/` -folder, open it with your text editor and fill it with the template HTML from below:
  > template.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>This is a basic HTML template</title>
    </head>
    <body>
        <h1>Use this template to create your own HTML files</h1>
    </body>
</html>
```

- See if your file is available in a browser under the following URL: /my-exercise.html
- In the body of the HTML change the content of the first `<h1>` -element to read: `GeoExt rocks!`
- Check if any changes to the HTML file are reflected in your browser. Reload the URL /my-exercise.html

If everything worked, you should see something like in the following images.

Our first HTML-page



Indeed, it does!

> **Please note:**
>
> In case you added more files (e.g.for upcoming tasks) to the `src/exercise/` folder and they are *not* instantly available under the URL /filename.html
>
> … then you have to stop and start the fileserving again. See the notes on starting / stopping (Hint: `Ctrl-C` or `npm start`)

Hello OpenLayers

Ok, we can create and edit HTML-files, and we can see the changes in our browser because all files in `src/exercises/` are always available under .

Let's see how we can include OpenLayers in our page so that we can start to use it. In order to do so, we need to include a CSS and a JavaScript file.

Exercises

- See if you find a folder `lib/ol/` inside of the `src/exercise/` -folder. It should contain two files: `ol.js` and `ol.css`
- Create a new `ol-example.html` from the basic template

> template.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>This is a basic HTML template</title>
    </head>
    <body>
        <h1>Use this template to create your own HTML files</h1>
    </body>
</html>
```

- Change `ol-example.html` to include both files in the `<head>`. Use the below templates to include a CSS and a JavaScript file.

> include-js-css.html

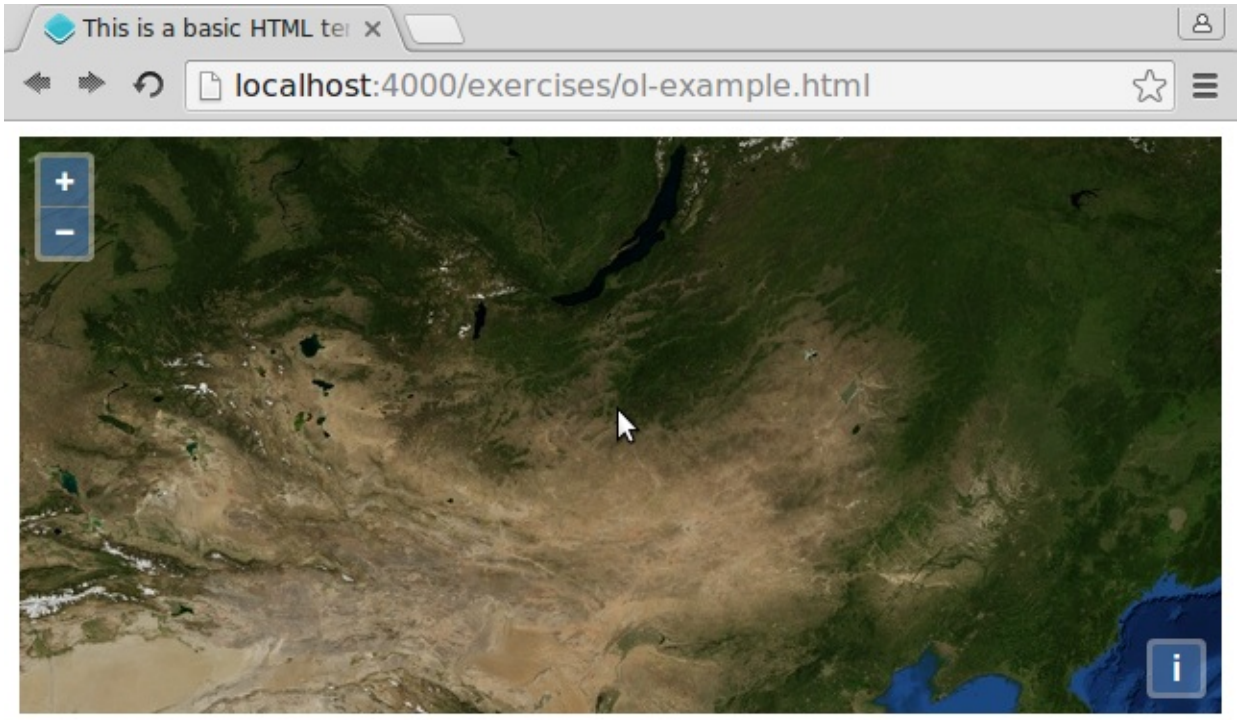
```
<!-- include a CSS stylesheet -->
<link rel="stylesheet" href="path/to/file.css" type="text/css">

<!-- include an external JavaScript file -->
<script src="path/to/file.js" type="text/javascript"></script>
```

- Verify that /ol-example.html loads your file.
- In the `<body>` of the file, add the following HTML-fragment, which includes a tiny bit of JavaScript:

> simple-map.html

```
<div id="map" style="height: 600px"></div>
<script type="text/javascript">
  var map = new ol.Map({
    target: 'map',
    layers: [
      new ol.layer.Tile({
        source: new ol.source.MapQuest({layer: 'sat'})
      })
    ],
    view: new ol.View({
      center: ol.proj.fromLonLat([106.92, 47.92]),
      zoom: 4
    })
  });
</script>
```

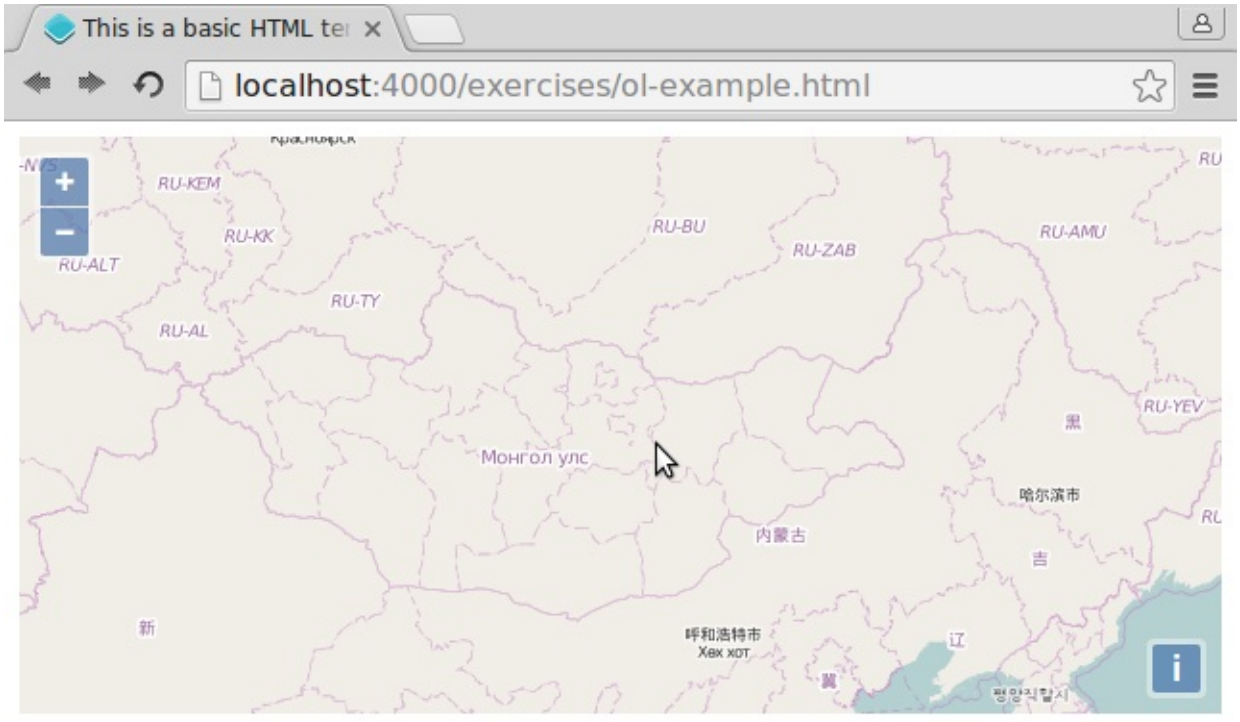
- When you now reload the /ol-example.html URL, you should see an OpenLayers map centered on Ulan Bator:



A very basic OpenLayers maa

- To verify we are really looking at Ulan Bator, just change the layers to now consist an OpenStreetMap layer, which e.g. has labels and a country outline. Use the following JavaScript snippet at the appropriate place:

```
new ol.layer.Tile({
  source: new ol.source.OSM()
})
```



Say "hi" to the OSM layer

Hello ExtJS

Before we can learn how to use GeoExt, we need to see if we can use ExtJS in our page.

Again we'll need to include two resources in a HTML page to be able to use ExtJS: And again it is a CSS and a JavaScript file.

Exercises

- Create a new `ext-example.html` from the basic template

> template.html

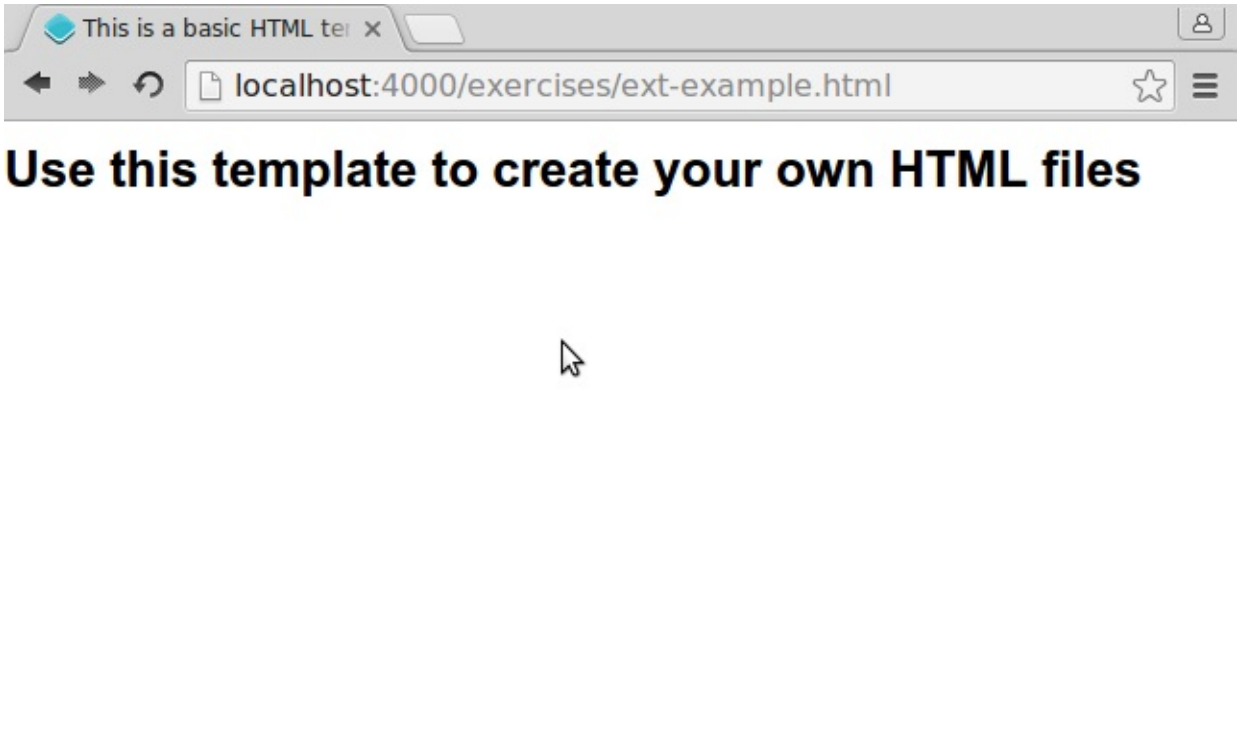
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>This is a basic HTML template</title>
    </head>
    <body>
        <h1>Use this template to create your own HTML files</h1>
    </body>
</html>
```

- Change `ex-example.html` to include the following two files:
  - https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-crisp/resources/theme-crisp-all.css
  - https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js

> include-js-css.html

```
<!-- include a CSS stylesheet -->
<link rel="stylesheet" href="path/to/file.css" type="text/css">

<!-- include an external JavaScript file -->
<script src="path/to/file.js" type="text/javascript"></script>
```

- Verify that /ext-example.html loads your file.
- Does your basic page look like the one in the following image? Why does the font look so different?

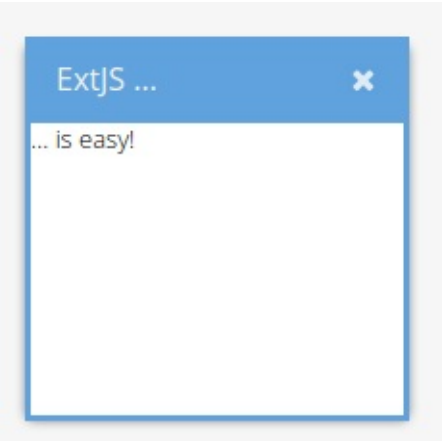


The template-HTML with the ExtJS resources included

- In order to see if everything was included successfully, let's instantiate an ExtJS class. Please copy and paste the following into the `<body>` of the test-file:

```
<script>
Ext.onReady(function(){
    var win = Ext.create('Ext.window.Window', {
        width: 200,
        height: 200,
        title: 'ExtJS …',
        html: '… is easy!'
    });
    win.show();
});
</script>
```

- You should see an `Ext.window.Window` like below:



ExtJS is easy

Hello GeoExt

Now that we know how to use OpenLayers and ExtJS, it's time to join these libraries. Enter **GeoExt**!

We'll start with the result of the last exercise, which was a basic HTML file that included the resources to ExtJS.

Exercises

- Copy the following HTML into a file `hello-geoext.html` in the `exercises` -directory:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>This is a basic HTML template</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/t
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
        Ext.onReady(function(){
            var win = Ext.create('Ext.window.Window', {
                width: 200,
                height: 200,
                title: 'ExtJS …',
                html: '… is easy!'
            });
            win.show();
        });
        </script>
    </body>
</html>
```

- Add the CSS and JavaScript for OpenLayers:

```
<link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
<script src="./lib/ol/ol.css" type="text/javascript"></script>
```

- Add the JavaScript for GeoExt: `https://geoext.github.io/geoext3/master/GeoExt.js`
- Most GeoExt components don't need special CSS. If you use the `Popup` -components, you may want to include the following CSS file: `http://geoext.github.io/geoext3/master/resources/css/gx-popup.css`
- Verify that /hello-geoext.html loads in your browser

Adding our first GeoExt component

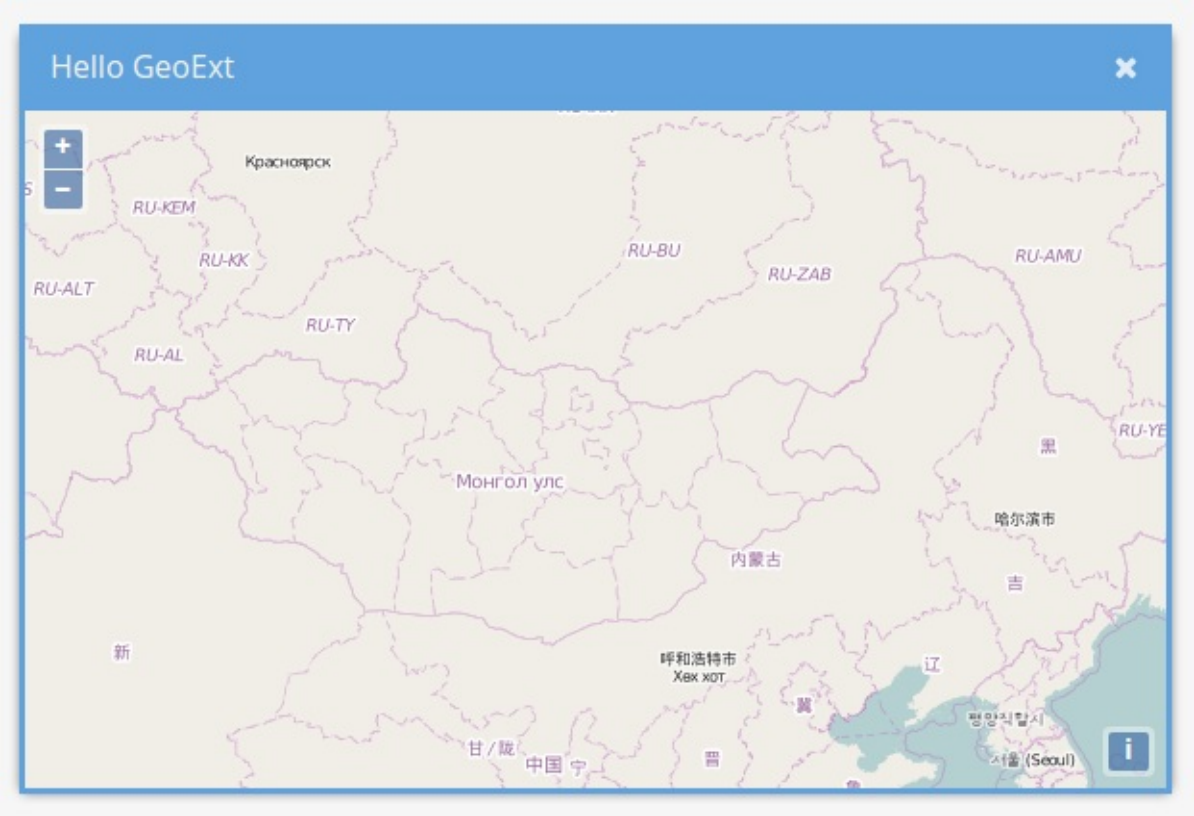
Exercises

- We are now going to create an instance of `GeoExt.component.Map` and render it in the window we created in the previous example.
- In order to do that:
  - Remove the `html: '…is easy!'` from the `Ext.window.Window` configuration object
  - And add the following lines instead:

```
// in the config object:
layout: 'fit',
items: [
    Ext.create('GeoExt.component.Map', {
        map: new ol.Map({
            target: 'map',
            layers: [
                new ol.layer.Tile({
                    source: new ol.source.OSM()
                })
            ],
            view: new ol.View({
                center: ol.proj.fromLonLat([106.92, 47.92]),
                zoom: 4
            })
        })
    })
]
```

- Next: Please change the title of the window and make is slightly bigger.

If your example looks like the one below, everything is set up correctly!



Hello GeoExt!

Useful resources

In order to efficiently work with all the libraries used throughout the workshop, you'll need know some other resources:

OpenLayers

Homepage: http://openlayers.org

API documentation

Use th API docs of OpenLayers to know about available classes and their properties:

- http://openlayers.org/en/v3.13.1/apidoc/

The left side of the API-docs is a list of all objects in OpenLayers. If you click on the name of an OpenLayers class there, you see the API of the object.

In the subpages you can find a list of all properties, methods and events the class provides. You should make yourself familiar with how to navigate the API docs.

Examples

Many people learn best when they see the parts of a library in action. OpenLayers has a vast amount of published online examples, which mostly focus on one aspect of the library.

- http://openlayers.org/en/v3.13.1/examples/

Browse the examples and learn how to find one that provides the information you need.

Other

- OpenLayers also has published a workshop at http://openlayers.org/workshop/.
- Common problems that may arise when using OpenLayers 3 are explained in the Frequently Asked Questions (FAQ).
- For specific questions one can ask on stackoverflow using the tag 'openlayers-3'.

ExtJS

API documentation

The quantity and quality of the ExtJS API documentation is outstanding.

- http://docs.sencha.com/extjs/6.0/6.0.0-classic/

The docs provide a list of all classes on the left and details once you click on any class. In order to understand and make use of ExtJS, it is crucial to fully grasp the documentation.

Examples

The examples for the ExtJS framework can be found here:

- http://examples.sencha.com/extjs/6.0.0/examples/

As with the API documentation, you may at first be overwhelmed at the sheer masses of examples. It is nonetheless very useful to click through some of them, as the show how to combine the classes of the framework into small working applications.

Other

- If you want to quickly check some class, you can e.g. use the Sencha Fiddle website.
- Specific questions (and answers) can be browsed in the Sencha Formums.

GeoExt

Homepage: https://geoext.github.io/geoext3/

API documentation

The GeoExt API documentation (generated with the same software as the ExtJS one) can be found here:

- http://geoext.github.io/geoext3/master/docs/

If you know your way around in The ExtJS documentation, you will easily understand the GeoExt one.

There is also a version of the API of GeoExt, which includes all the classes from the ExtJS framework:

- http://geoext.github.io/geoext3/master/docs-w-ext/

Examples

The (few) examples of the GeoExt library can be accessed from the homepage: https://geoext.github.io/geoext3/

Summary

This chapter gently introduced you to OpenLayers, ExtJS and GeoExt. You have learned…

- … how and where to create exercise files.
- … how to include the resources of OpenLayers.
- … how to include the resources of ExtJS.
- … how to include the resources of GeoExt.
- … where to find more information online.

The next chapter will focus on the `GeoExt.component.Map` which we have seen briefly in the first example we created.

Map

This chapter will introduce you to one of the most central components in GeoExt: the `GeoExt.component.Map`.

Will work through the following parts to learn about this component:

- Creating a basic example
- Dissecting the parts of the example
- Explore configuration variants

Basic example

We want to have a look at a fully working example first.

Exercises

- Create a new file `map.html` in the `src/exercise` -directory.
- Paste the following html-code into the file you have just created:

```
<!DOCTYPE html>
<html>
  <head>
      <meta charset="utf-8">
      <title>Exercise | GeoExt Workshop</title>
      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resour
      <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script
      <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
      <script src="./lib/ol/ol.js" type="text/javascript"></script>
      <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
  </head>
  <body>
      <script>
var map;
Ext.onReady(function(){
  // 1) OpenLayers
  //
  // Create an instance of an OpenLayers map:
  map = new ol.Map({
      layers: [
          new ol.layer.Tile({
              source: new ol.source.OSM()
          })
      ],
      view: new ol.View({
          center: ol.proj.fromLonLat( [106.92, 47.92] ),
          zoom: 12
      })
  });

  // 2) GeoExt
  //
  // Create an instance of the GeoExt map component with that map:
  var mapComponent = Ext.create('GeoExt.component.Map', {
      map: map
  });

  // 3) Ext JS
  //
  // Create a viewport
  var vp = Ext.create('Ext.container.Viewport', {
      layout: 'fit',
      items: mapComponent
  });
});
      </script>
  </body>
</html>
```

- Verify that /map.html loads in your browser and looks like the picture below.

A map component in a fullscreen viewport

We will now dissect the example and explain what each part does.

Dissecting the example

Let's look at the parts of the HTML page.

The HTML skeleton

The HTML of the page looks as follows:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/t
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
        </script>
    </body>
</html>
```

HTML5 DOCTYPE

The first line in this document is the `doctype` of the HTML. By specifying…

```
<!DOCTYPE html>
```

…we declare that the HTML file shall be handled as an HTML5 document. We recommend the usage of this doctype to force browsers into fixed rules of rendering the page. This eventually also reduces inconsistencies of the behaviour of the page in various browsers.

Declaration of the character set

In order to tell the browser that we have encoded our file as UTF-8, we add a `<meta>` -tag to the `&lt;head&gt;` of the document:

```
<head>
    <meta charset="utf-8">
</head>
```

This way we can be relatively sure that all the characters we enter into the document (e.g. German umlauts like `ä`, `ö` or `ü`; or `Улаанбаатар`) are correctly displayed when viewing the site.

CSS and JavaScript resources

Also in the `<head>` of the document we load external JavaScript and CSS files, so we can use our needed libraries later.

```
<head>
    <link rel="stylesheet" href="URL-or-relative-path-to-file" type="text/css">
    <script src="URL-or-relative-path-to-file" type="text/javascript"></script>
</head>
```

For this workshop it will be enough to always include the full builds of the library; and to always load them in the `<head>`. This technique allows us to basically forget about these resources for the course of the workshop. For a production website you would probably load the files in a different manor, and you would rather not load the versions of the libraries which contain everything. The creation of specific versions of the base libraries that only include what your application actually needs, is way beyond the scope of this workshop.

`<script>` -tag in the `<body>`

Our body of the HTML file is really, really, *really* minimalistic:

```
<body>
    <script>
    </script>
</body>
```

We only include one `&lt;script&gt;` -tag that will contain all the JavaScript that we need to create our map. The contents of this tag will be interpreted as JavaScript, and the code will be run as soon as the browser sees it.

JavaScript code for the map

All our code to create the full screen map lives in the `<script>` -tag in the HTML `<body>`.

Let's go through all the lines in there.

A variable named `map`

The first line in the example reads:

```
var map;
```

This creates a global variable named map, which (at this point) has the value `undefined`. Later on we will store our instance of the `ol.Map` in that variable. We have made it global to allow for easier debugging (e.g. in the developer tools of your browser). For the workshop it is OK to create a lot of global variables for stuff you want to examine later on; in production sites it usually frowned upon.

Passing a function to `Ext.onReady`

The next line reads:

```
Ext.onReady(function(){
    // some other lines we do not care about now
});
```

These lines pass an anonymous (e.g. unnamed) function to the method `Ext.onReady`. This method will execute the passed function as soon as the Document is ready, e.g. External resources have loaded and the DOM (Document Object Model) of the page is ready to be manipulated.

Behind the curtains, when we create instances of some Ext classes, they will eventually need to modify the DOM. In order to run into problems when such changes happen to early (remember, all code in the `<script>` tag is executed as soon as it is being read), we wrap the real code to actually create ExtJS components into a function. We then simply tell ExtJS to delay the real work to a later time, when everything is ready.

Let's have a look at the parts inside this function.

Creating an `ol.Map`

First we want to create an instance of an `ol.Map`:

```
map = new ol.Map({
    layers: [
        new ol.layer.Tile({
            source: new ol.source.OSM()
        })
    ],
    view: new ol.View({
        center: ol.proj.fromLonLat( [106.92, 47.92] ),
        zoom: 12
    })
});
```

These lines create an OpenLayers map and configure it with a view that is centered on Ulan Bator and that has one layer showing pre-rendered tiles from the OpenStreetMap project.

You should already be slightly familiar with OpenLayers and can basically use any map that works without GeoExt.

Since we did not write `var map = …`, the assignment will happen to the global variable map, that we declared in the first line of the `script`-tag. You can easily debug the OpenLayers map this way.

Creating a `GeoExt.component.Map`

Next we use the method `Ext.create` with two arguments: the name of the class to create, and a configuration object with properties for the instance.

```
var mapComponent = Ext.create('GeoExt.component.Map', {
    map: map
});
```

In plain English this line could read

> Please create an instance of the class `GeoExt.component.Map` and ensure that it is configured with the OpenLayers map I have stored in the variable `map`. Once you have done that, please store this instance in a variable `mapComponent`.

After these lines have executed, we now have two variables, one holding the plain OpenLayers map (`map`), and one that is named `mapComponent` which contains an instance of a GeoExt class; and this instance knows about the OpenLayers map.

Creating a `Ext.Viewport`

The final four lines in the block read:

```
var vp = Ext.create('Ext.container.Viewport', {
    layout: 'fit',
    items: mapComponent
});
```

Again we use `Ext.create` to build an instance of a class, this time of the `Ext.container.Viewport` class. From the ExtJS API docs:

> A specialized container representing the viewable application area (the browser viewport).
>
> The Viewport renders itself to the document body, and automatically sizes itself to the size of the browser viewport and manages window resizing. There may only be one Viewport created in a page.

(source)

This viewport will be as big as the browser viewport. All it's children (configured via the `items`-key) will be layed out according to the `fit`-layout. This layout ensures that the child component (in our case the `mapComponent`) will be as big as the viewport itself.

Try to resize your browser window and see that the viewport (and the containing map component) always fill out the full area of the browser window.

Next steps

Let's look at various variants to configure the three parts of our map.

Configuration variants

This chapter looks at our possibilities to customize the appearance and behaviour of the map.

Configuring aspects of OpenLayers

As you have seen, we have simply created an instance of `ol.Map` and passed it to the `GeoExt.component.Map`. If we configure the `ol.Map` differently, the changes should be reflected in the final application.

Exercises

Change the following aspects of the OpenLayers map:

- Set a different map center.
- Initially zoom to another region.
- Add more layers to the map. Try the layers from these WMS capabilities for example:
  - http://ows.terrestris.de/osm/service?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetCapabilities
  - http://ows.terrestris.de/osm-gray/service?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetCapabilities
- Add another control. Try these, for example:
  - `ol.control.ScaleLine`
  - `ol.control.MousePosition`

Configuring aspects of ExtJS

Change the following aspects of the extJS components:

- Use another layout for the viewport. Just remember that you probably need to change two places:
  - The `layout` config of the viewport.
  - And depending on the chosen layout, children (our map-component) may need new properties.
- Wrap the `GeoExt.component.Map` in a panel with a title.

Configuring aspects of GeoExt

Of course you can also change aspects directly via GeoExt:

- Set the center of the map, but this time with GeoExt.
- Add a layer with GeoExt.

Summary

This chapter has shown you how to use the `GeoExt.component.Map`. We first started with a working example, that was the dissected in great detail. You have then learned how to configure your application by changing properties of OpenLayers-, ExtJS- and GeoExt-objects.

The next chapter will now introduce a component to deal with the possibly hierarchical structure of layers in the tree: we want to add a layer-tree.

Layer tree

Now that we have a map-component in an ExtJS layout, we naturally want to add more layers to the map. But when we have more than one layer in the OpenLayers map, we also may want to include some component to handle the individual visibility and the order of layers in the map. As OpenLayers does not provide a control to influence these properties, we have to create our own component.

GeoExt wants to help here by providing the necessary parts to create a Tree of the layers in the map-component.

Let's try to add a layer tree.

Prepare layout

The previous chapter started from the following template, which we now want to recreate.

Exercises

- Please create a file `map.html` in the `src/erxercises` directory and paste the following:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/t
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){
    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 12
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'fit',
        items: mapComponent
    });
});
        </script>
    </body>
</html>
```

- We want to change the layout of the viewport as follows:

```
var vp = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [
        mapComponent
    ]
});
```

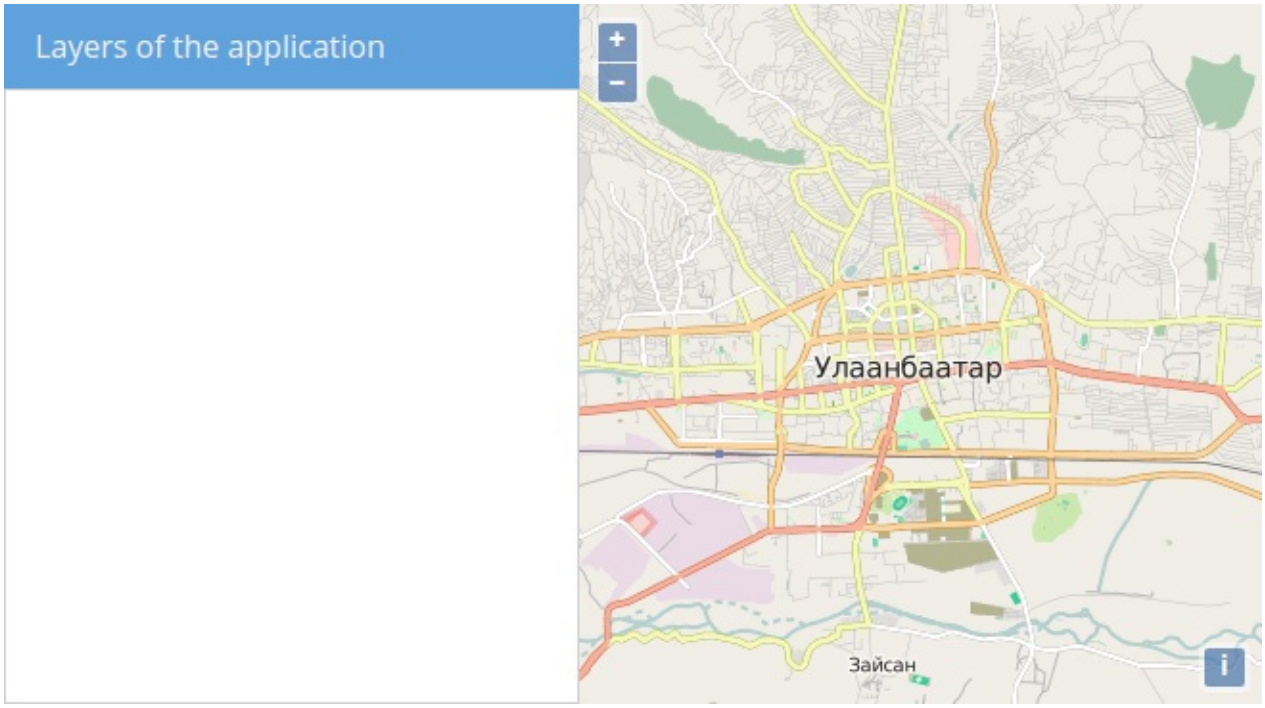
- In order to be usable in a border-layout, one child component needs to have the `region` -property set to `center` :

```
var mapComponent = Ext.create('GeoExt.component.Map', {
    map: map,
    region: 'center'
});
```

- If you apply the above changes, your application should render again in the browser, but since we only have one component in the border-layout, you'll not notice a visual difference.
- Let's first add a placeholder panel where we want to add the layer tree:

```
var layerTreePanel = Ext.create('Ext.panel.Panel', {
    title: 'Layers of the application',
    width: 300,
    region: 'west'
});
// … this panel also needs to be added to the viewport
var vp = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [
        mapComponent,
        layerTreePanel
    ]
});
```

- Your application should now look like the following:



Our placeholder panel in the viewport

Create a TreePanel

We now have the layout prepared and simply need to switch the contents of the west-panel.

Since we want to use a tree to eventually control the layers of the map, we'll use an `Ext.tree.Panel` instead of the simple panel.

Exercises

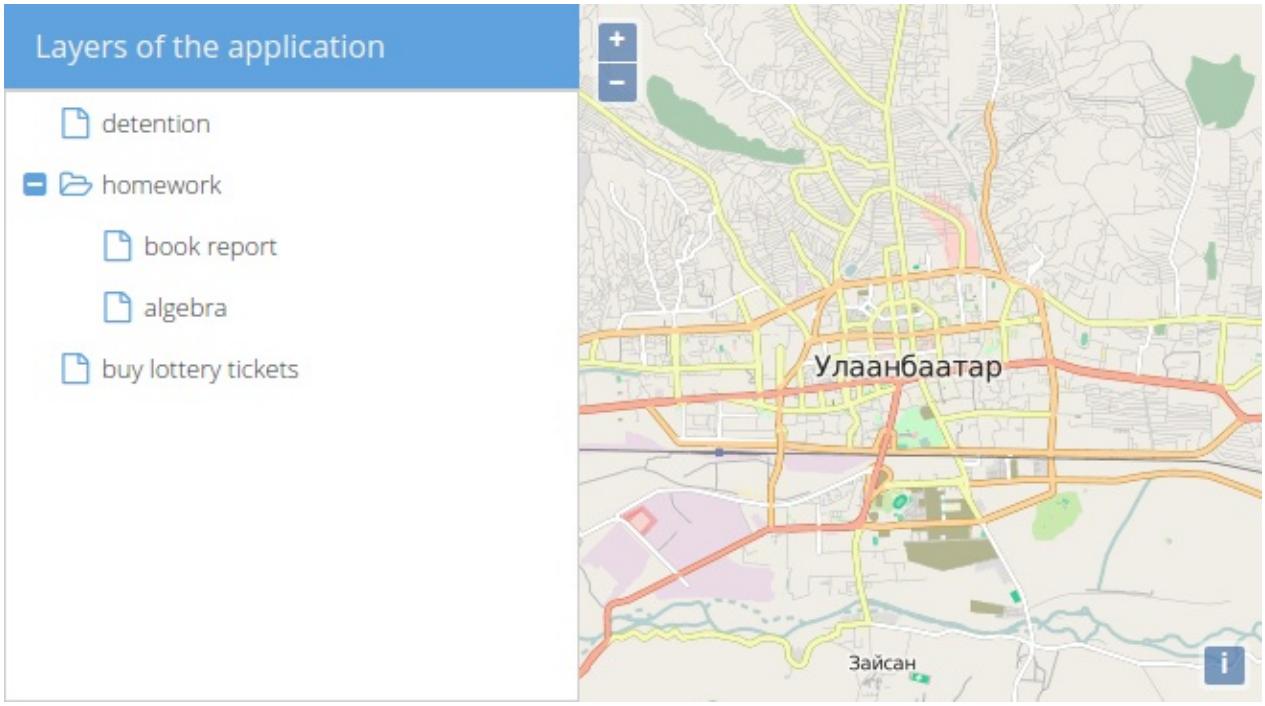
- Next, we'll switch out the `Ext.panel.Panel` against a dedicated `Ext.tree.Panel`. If we look at the documentation for the tree-panel, you'll see a very basic example, which you please add to the viewport instead of our placeholder.
- The example from the above page looks like this:

```
var store = Ext.create('Ext.data.TreeStore', {
    root: {
        expanded: true,
        children: [
            { text: 'detention', leaf: true },
            { text: 'homework', expanded: true, children: [
                { text: 'book report', leaf: true },
                { text: 'algebra', leaf: true}
            ] },
            { text: 'buy lottery tickets', leaf: true }
        ]
    }
});

Ext.create('Ext.tree.Panel', {
    title: 'Simple Tree',
    width: 200,
    height: 150,
    store: store,
    rootVisible: false,
    renderTo: Ext.getBody()
});
```

- Try to understand what each line of the above code does and see which lines you need to change or remove, so that you can use the tree in our layout.
- 

Hints

- Some hints (in case you have trouble getting it to work)
  - The store — as complicated as it looks at first — can be left as is, you don't need to change something here.
  - The return value of the `Ext.create(&#39;Ext.tree.Panel&#39;, /**/)` call is currently ignored. You should try to save it in a variable (probably the one from our basic setup `layerTreePanel`).
  - The height of the tree-panel is unnecessary, we want to put the panel in the west region, which has full height by default. Remove the `height`-property.
  - The `renderTo`-configuration of the tree-panel is also fine for the ExtJS standalone example, but bad for our combination setup. In our case, the viewport takes care of where to actually render the tree. Remove the `renderTo`-property.
- The final result should look like this:



The copy and pasted Ext-example in our viewport

-

Solution

- For reference, here is the full code of the store, tree and viewport that lead to the above picture:

```
var store = Ext.create('Ext.data.TreeStore', {
    root: {
        expanded: true,
        children: [
            { text: 'detention', leaf: true },
            { text: 'homework', expanded: true, children: [
                { text: 'book report', leaf: true },
                { text: 'algebra', leaf: true}
            ] },
            { text: 'buy lottery tickets', leaf: true }
        ]
    }
});

var layerTreePanel = Ext.create('Ext.panel.Panel', {
    title: 'Layers of the application',
    width: 300,
    region: 'west',
    store: store,
    rootVisible: false
});

var vp = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [
        mapComponent,
        layerTreePanel
    ]
});
```

Assign LayersTree store

Instead of the hard-coded hierarchical list of things in the tree, we now want to link the tree with our map.

In order to do this, we need to change the store that is used for the tree. Instead of the all-purpose `Ext.data.TreeStore`, we'll use the special GeoExt class `GeoExt.data.store.LayersTree`

Exercises

- If you haven't done already, set up a file called `map.html` in the `src/exercises` directory and paste the following contents:

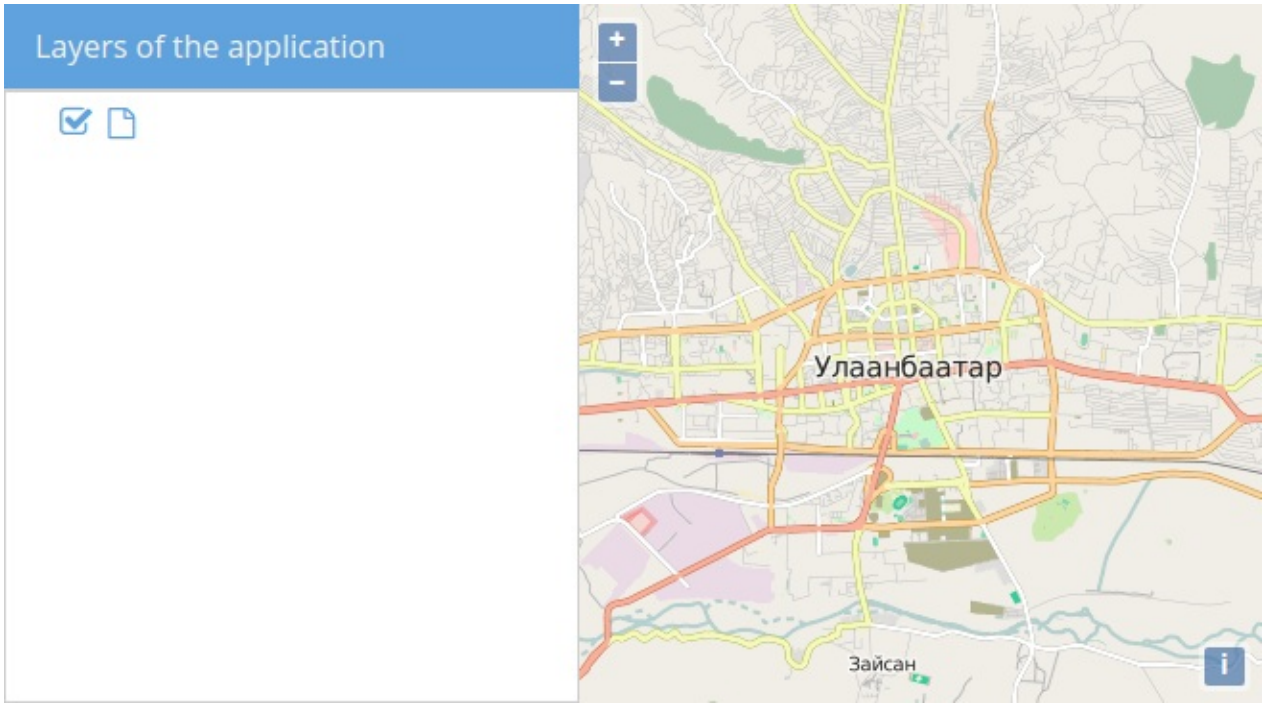
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){
    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 12
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map,
        region: 'center'
    });

    var store = Ext.create('Ext.data.TreeStore', {
        root: {
            expanded: true,
            children: [
                { text: 'detention', leaf: true },
                { text: 'homework', expanded: true, children: [
                    { text: 'book report', leaf: true },
                    { text: 'algebra', leaf: true}
                ] },
                { text: 'buy lottery tickets', leaf: true }
            ]
        }
    });

    var layerTreePanel = Ext.create('Ext.tree.Panel', {
        title: 'Layers of the application',
```

```
        width: 300,
        region: 'west',
        store: store,
        rootVisible: false
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'border',
        items: [
            mapComponent,
            layerTreePanel
        ]
    });
});
        </script>
    </body>
</html>
```

- Make yourself familiar with the `GeoExt.data.store.LayersTree` class by studying the following API-docs: http://geoext.github.io/geoext3/master/docs/#!/api/GeoExt.data.store.LayersTree
- Create an instance of the `GeoExt.data.store.LayersTree` class and pass it the following configuration object:

```
{
    layerGroup: /* the top level layer group of the map */
}
```

- Study the API docs of `ol.Map` to get the appropriate `LayerGroup`: http://openlayers.org/en/v3.13.1/apidoc/ol.Map.html
- If everything works fine, you should see a tree with one (currently unlabeled) leaf. Next to the leaf you find a checkbox, that reflects the overall visibility of the layer.

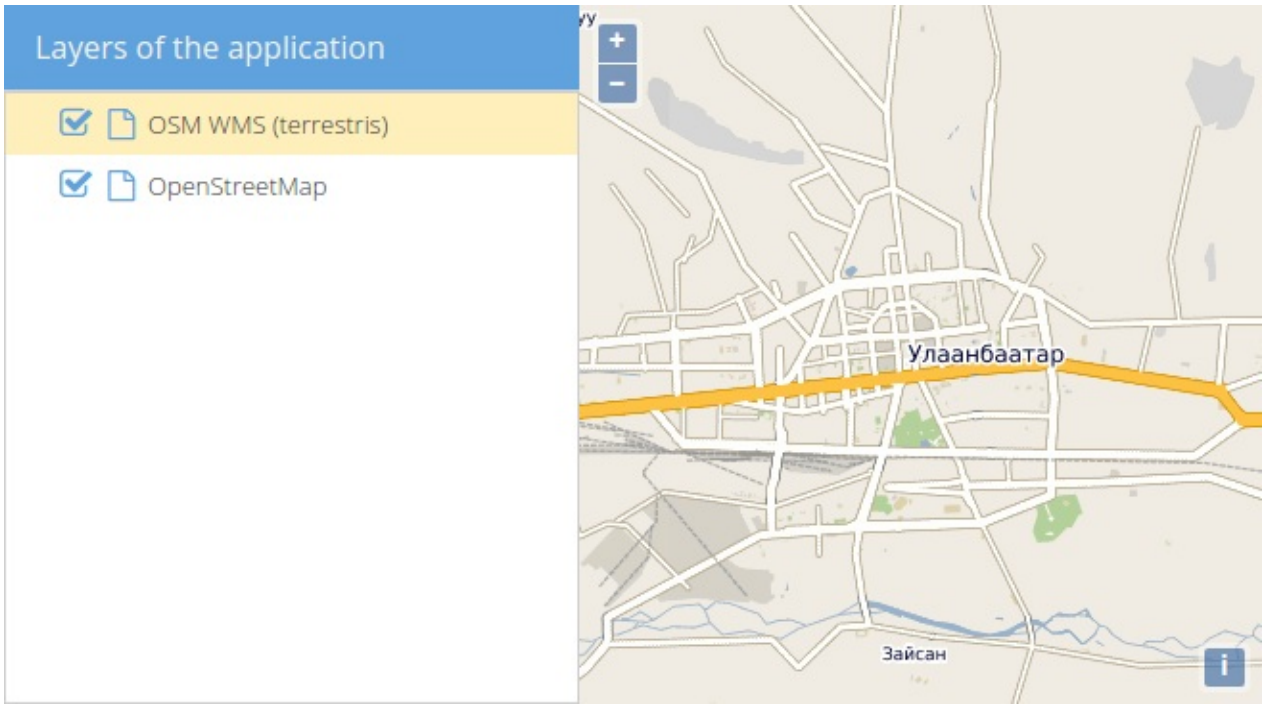


The working but currently unlabeled tree

- Study the `GeoExt.data.store.LayersTree` and find out why there is no label next to the tree-element.
- Add more layers to the map and see if they all appear in the map and in the tree. Take e.g. the following WMS:

```
url: http://ows.terrestris.de/osm/service
layers: OSM-WMS
```

- Read the documentation for the Ext class `Ext.tree.plugin.TreeViewDragDrop`. What happens if you add this plugin to the tree?
- Your application should now e.g. look like this:



The tree in the application

-

Solution

- For reference, here are code-snippets for the relevant parts of the code:

```
// layers should have a property for their name (configurable)
new ol.layer.Tile({
    source: new ol.source.OSM(),
    name: 'OpenStreetMap'
});

// Creating an appropriate treestore
var treeStore = Ext.create('GeoExt.data.store.LayersTree', {
    layerGroup: map.getLayerGroup()
});

// Use the store in the tree and also load plugin
var layerTreePanel = Ext.create('Ext.tree.Panel', {
    title: 'Layers of the application',
    width: 300,
    region: 'west',
    store: treeStore,
    rootVisible: false,
    viewConfig: {
        plugins: { ptype: 'treeviewdragdrop' }
    }
});
```

Summary

This chapter taught you how to make use of the class `GeoExt.data.store.LayersTree` to create a tree showing the layers of the application.

The tree-panel correctly reorders the layer order in the map if the order changes (via drag and drop) in the tree. All tree-leafs have a checkbox to control the visibility of the connected layer.

Feature grid

In this chapter we want to add a grid component to the application, which shows a row for every feature of a vector layer. We also want to add basic interaction between the grid of features and the map.

We will start in the usual way by setting up an ExtJS placeholder component which we will then gradually be enhanced or replaced.

Head over to the prepare layout chapter, in which we'll add another visual component to our application.

Prepare layout

We want to add a grid panel to our basic map application now.

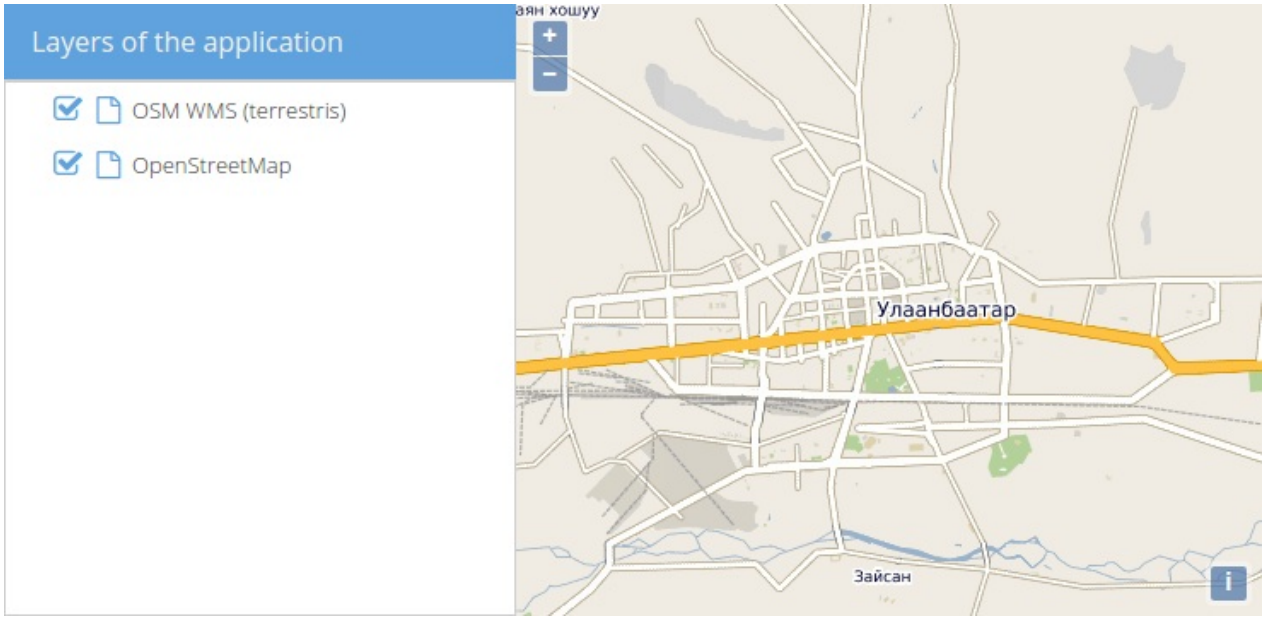
Exercises

- Prepare the `map.html` file to contain the following code. This is basically the result of the previous chapters:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/t
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){
    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM(),
                name: 'OpenStreetMap'
            }),
            new ol.layer.Tile({
                source: new ol.source.TileWMS({
                    url: 'http://ows.terrestris.de/osm/service',
                    params: {
                        layers: 'OSM-WMS'
                    }
                }),
                name: 'OSM WMS (terrestris)'
            })
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 12
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map,
        region: 'center'
    });

    var treeStore = Ext.create('GeoExt.data.store.LayersTree', {
        layerGroup: map.getLayerGroup()
    });

    var layerTreePanel = Ext.create('Ext.tree.Panel', {
        title: 'Layers of the application',
        width: 300,
        region: 'west',
        store: treeStore,
```

```
        rootVisible: false,
        viewConfig: {
            plugins: { ptype: 'treeviewdragdrop' }
        }
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'border',
        items: [
            mapComponent,
            layerTreePanel
        ]
    });
});
        </script>
    </body>
</html>
```

- If you open this file in a browser (/map.html), the application should look like in the following image:



Our starting point

- We want to have a grid in the south, so let's start with the basic example from the ExtJS Grid documentation:

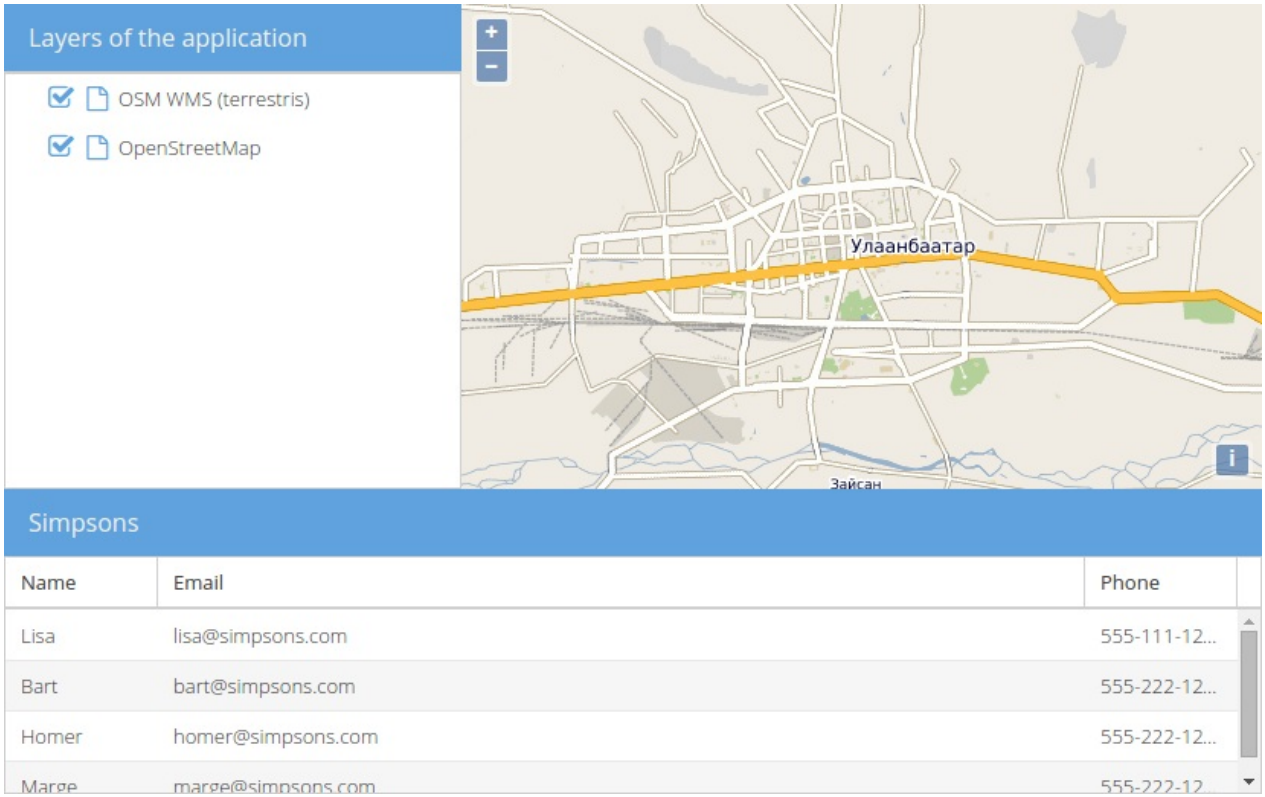
```
Ext.create('Ext.data.Store', {
    storeId: 'simpsonsStore',
    fields:[ 'name', 'email', 'phone'],
    data: [
        { name: 'Lisa', email: 'lisa@simpsons.com', phone: '555-111-1224' },
        { name: 'Bart', email: 'bart@simpsons.com', phone: '555-222-1234' },
        { name: 'Homer', email: 'homer@simpsons.com', phone: '555-222-1244' },
        { name: 'Marge', email: 'marge@simpsons.com', phone: '555-222-1254' }
    ]
});

Ext.create('Ext.grid.Panel', {
    title: 'Simpsons',
    store: Ext.data.StoreManager.lookup('simpsonsStore'),
    columns: [
        { text: 'Name', dataIndex: 'name' },
        { text: 'Email', dataIndex: 'email', flex: 1 },
        { text: 'Phone', dataIndex: 'phone' }
    ],
    height: 200,
    width: 400,
    renderTo: Ext.getBody()
});
```

- Instead of using a `storeId` and then later `Ext.data.StoreManager.lookup('simpsonsStore')`, we will simply use a variable to be able to reference the store. Since we will put the panel in our border layout, we do not need the `renderTo` and `width` properties. Don't forget to assign the `region: south`. We'll also save the panel in a variable. Your code should look roughly like the following:
- 

Hint

```
var featureStore = Ext.create('Ext.data.Store', {
    fields:[ 'name', 'email', 'phone'],
    data: [
        { name: 'Lisa', email: 'lisa@simpsons.com', phone: '555-111-1224' },
        { name: 'Bart', email: 'bart@simpsons.com', phone: '555-222-1234' },
        { name: 'Homer', email: 'homer@simpsons.com', phone: '555-222-1244' },
        { name: 'Marge', email: 'marge@simpsons.com', phone: '555-222-1254' }
    ]
});

var featurePanel = Ext.create('Ext.grid.Panel', {
    title: 'Simpsons',
    store: featureStore,
    columns: [
        { text: 'Name', dataIndex: 'name' },
        { text: 'Email', dataIndex: 'email', flex: 1 },
        { text: 'Phone', dataIndex: 'phone' }
    ],
    height: 200,
    region: 'south'
});
```

- Once we have added the `featurePanel` to the viewport, our application should look like in the following image:



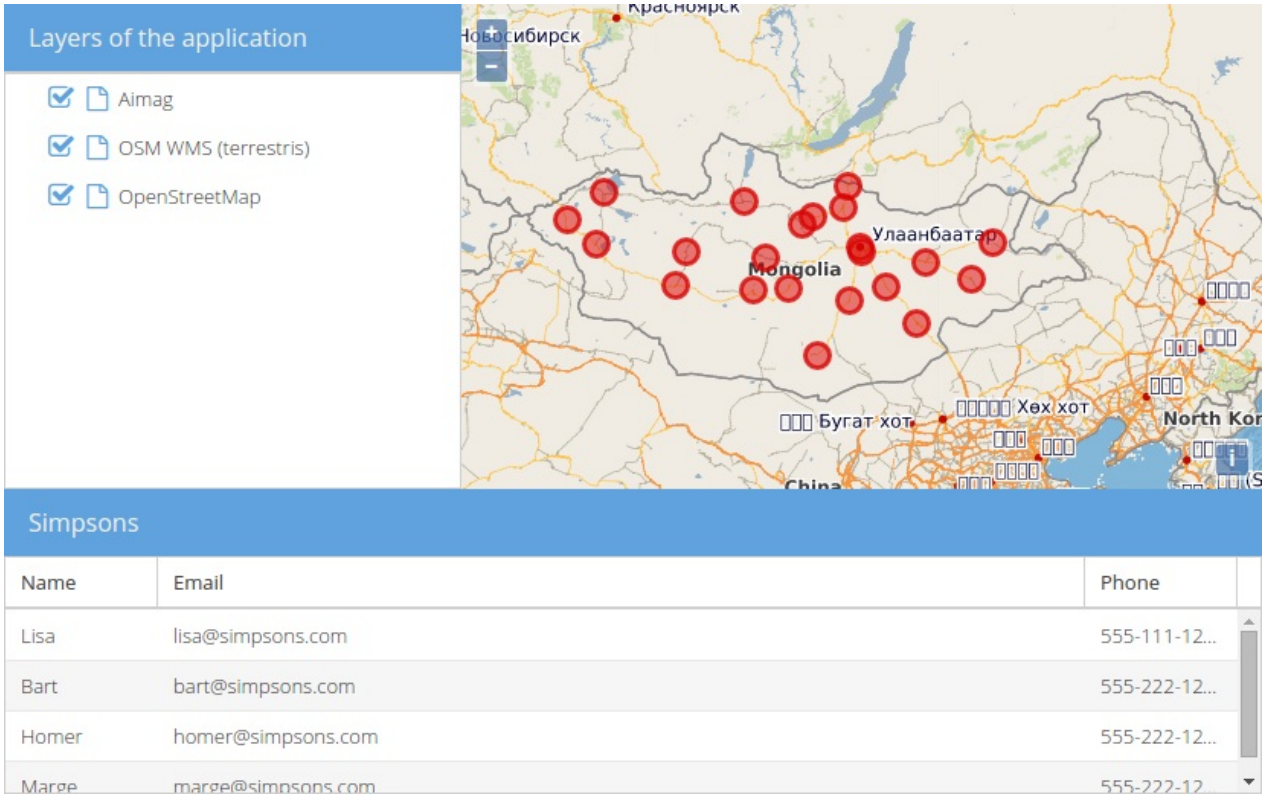
The prepared ExtJS layout

- Of course we also want to have a vector layer in the map, whose features we want in the grid later.
- Please create a new `ol.layer.Vector`, that has a `ol.source.GeoJSON` configured and loads the local data in `src/exercises/data/aimag-centers.json`. Please style the points with red circles. Please also zoom the map a little bit further out; zoom level `4` should be fine.
-

Hint

```
var redStyle = new ol.style.Style({
    image: circle = new ol.style.Circle({
        fill: new ol.style.Fill({
            color: 'rgba(220, 0, 0, 0.5)'
        }),
        stroke: new ol.style.Stroke({
            color: 'rgba(220, 0, 0, 0.8)',
            width: 3
        }),
        radius: 8
    })
})
var vectorLayer = new ol.layer.Vector({
    source: new ol.source.Vector({
        url: 'data/aimag-centers.json',
        format: new ol.format.GeoJSON()
    }),
    name: 'Aimag',
    style: redStyle
});
```

- Our application should now look like in the following image:



Our map now also shows the 'Aimag'

Create a feature grid

Now it's time to change the grid to no longer show static data from The Simpsons, but instead one row for every feature of the vector layer.

Exercises

- Please set up `src/map.html` to contain the following lines:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){
    var redStyle = new ol.style.Style({
        image: circle = new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(220, 0, 0, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(220, 0, 0, 0.8)',
                width: 3
            }),
            radius: 8
        })
    })
    var vectorLayer = new ol.layer.Vector({
        source: new ol.source.Vector({
            url: 'data/aimag-centers.json',
            format: new ol.format.GeoJSON()
        }),
        name: 'Aimag',
        style: redStyle
    });
    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM(),
                name: 'OpenStreetMap'
            }),
            new ol.layer.Tile({
                source: new ol.source.TileWMS({
                    url: 'http://ows.terrestris.de/osm/service',
                    params: {
                        layers: 'OSM-WMS'
                    }
                }),
                name: 'OSM WMS (terrestris)'
            }),
            vectorLayer
        ],
        view: new ol.View({
```

```
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 4
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map,
        region: 'center'
    });

    var treeStore = Ext.create('GeoExt.data.store.LayersTree', {
        layerGroup: map.getLayerGroup()
    });

    var layerTreePanel = Ext.create('Ext.tree.Panel', {
        title: 'Layers of the application',
        width: 300,
        region: 'west',
        store: treeStore,
        rootVisible: false,
        viewConfig: {
            plugins: { ptype: 'treeviewdragdrop' }
        }
    });

    var featureStore = Ext.create('Ext.data.Store', {
        fields:[ 'name', 'email', 'phone'],
        data: [
            { name: 'Lisa', email: 'lisa@simpsons.com', phone: '555-111-1224' },
            { name: 'Bart', email: 'bart@simpsons.com', phone: '555-222-1234' },
            { name: 'Homer', email: 'homer@simpsons.com', phone: '555-222-1244' },
            { name: 'Marge', email: 'marge@simpsons.com', phone: '555-222-1254' }
        ]
    });

    var featurePanel = Ext.create('Ext.grid.Panel', {
        title: 'Simpsons',
        store: featureStore,
        columns: [
            { text: 'Name', dataIndex: 'name' },
            { text: 'Email', dataIndex: 'email', flex: 1 },
            { text: 'Phone', dataIndex: 'phone' }
        ],
        height: 200,
        region: 'south'
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'border',
        items: [
            mapComponent,
            layerTreePanel,
            featurePanel
        ]
    });
});
        </script>
    </body>
</html>
```

- Instead of a generic `Ext.data.Store`, use a `GeoExt.data.store.Features`. Look up the API documentation on http://geoext.github.io/geoext3/master/docs for further details.
- Make sure you reference your vector layer and the map to work on when configuring the store.

- 

Hint

```
var featureStore = Ext.create('GeoExt.data.store.Features', {
    layer: vectorLayer,
    map: map
});
```

- Next we need to configure the `columns` of the `Ext.grid.Panel`. Look up the attributes of the GeoJSON and change the appropriate configuration in the config object for the `Ext.grid.Panel`.
- 

Hint

```
// E.g.
columns: [
    {text: 'Name', dataIndex: 'NAME', flex: 3},
    {text: 'Population', dataIndex: 'POP', flex: 1},
    {text: 'Id', dataIndex: 'AIMAG_ID', flex: 1}
]
```

- Additionally we can use the `GeoExt.grid.column.Symbolizer` class of GeoExt to include the styling of the feature in the grid. Add the following line to your columns definition:

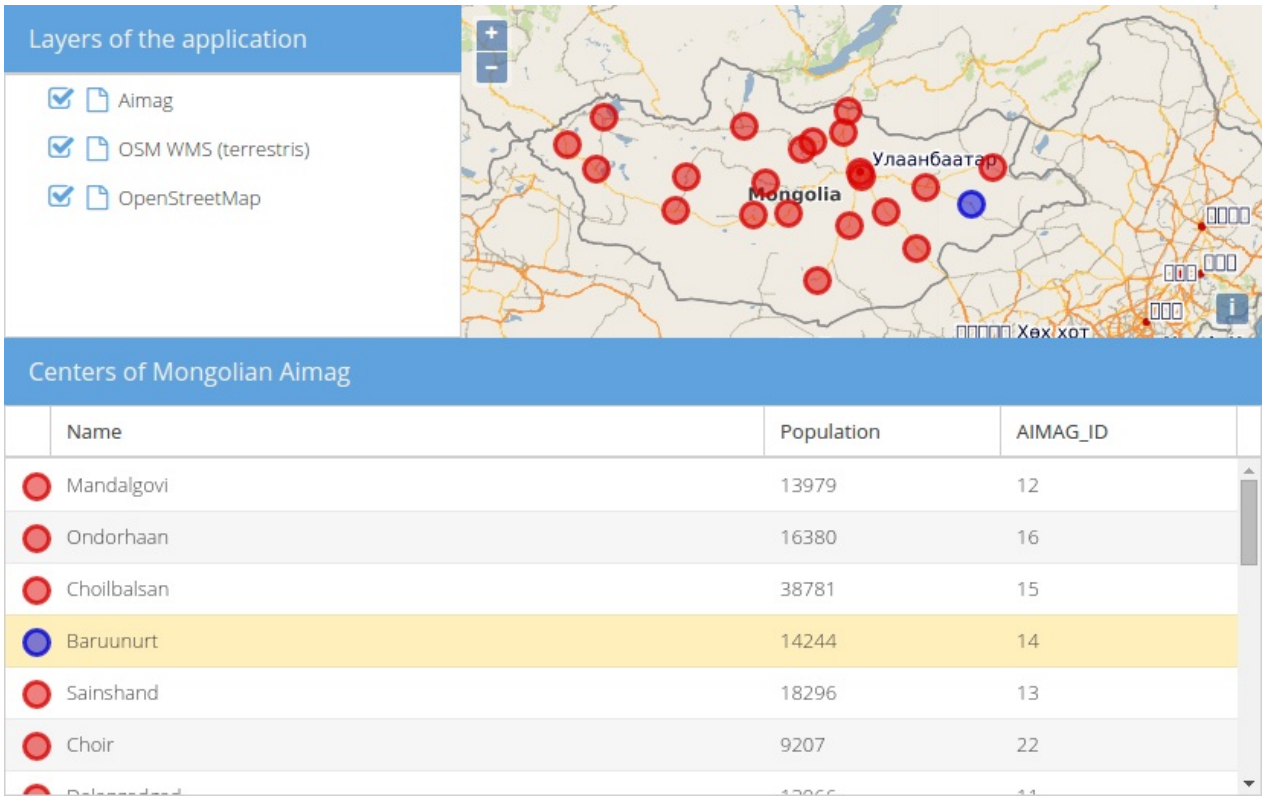
```
{xtype: 'gx_symbolizercolumn', width: 30}
```

- When a row is selected in the grid, it is visually highlighted. Wouldn't it be nice if the feature on the map would also have a different style once its associated row is selected?
- Assign a `selectionchange` listener on the grid and ensure that the correct feature is highlighted in the map. Hint: Create a new style and in the callback reset the style for every feature and reassign the new style to the to the currently selected feature. Use `console.log(arguments)` to see what you have been passed and how you can get the feature from the passed arguments.
- **Bonus:** Once the feature has a different style on the map, it would be nice if we could see that style in the grid, right? Change the `selectionchange` listener to also update the grid once the style of the feature has changed.
- 

Hint

```
var featureGrid = Ext.create('Ext.grid.Panel', {
    store: featureStore,
    region: 'south',
    title: 'Centers of Mongolian Aimag',
    columns: [
        {xtype: 'gx_symbolizercolumn', width: 30},
        {text: 'Name', dataIndex: 'NAME', flex: 3},
        {text: 'Population', dataIndex: 'POP', flex: 1},
        {text: 'AIMAG_ID', dataIndex: 'AIMAG_ID', flex: 1}
    ],
    listeners: {
        selectionchange: function(sm, selected) {
            // reset all selections
            featureStore.each(function(rec) {
                rec.getFeature().setStyle(null);
            });
            // highlight grid selection in map
            Ext.each(selected, function(rec) {
                rec.getFeature().setStyle(blueStyle);
            });
            // update the grid rendering of the geometry
            sm.view.refresh();
        }
    },
    height: 300
});
```

- Your application should now roughly look like depicted below:



The application with a feature grid

Summary

This chapter has taught you a lot:

- We learned how to create and use the `GeoExt.data.store.Features` class and configure it with a layer and a map.
- We learned that such stores can be an in place replacement for `Ext.data.Store` , e.g. to display the contained data in a grid.
- We learned about `GeoExt.grid.column.Symbolizer` that can be used to render feature symbolizers in grids.
- We also have seen how convenient it is to use event listeners to update the visual representation of your map (the `selectionchange` listener).

The next and final chapter will briefly introduce you to some other GeoExt components.

Popups, Overview & more

GeoExt has some more components and classes that we didn't touch so far. This chapter wants to show some other aspects of GeoExt. However, we cannot go through every class in detail and will only barely touch all the possibilities that GeoExt provides.

We will enhance our current application with two concrete usages of GeoExt functionality: an embedded overview-map and popups for hovered locations.

Furthermore we want to give a short theoretical outlook for possible enhancements of your applications. You can then explore these on your own.

Let's start with adding popups for hovered coordinates, shall we?

Popups

In this chapter we want to add short informative popups on the map. We will open the popup when the mouse lasts for a certain amount of time at a specific location. Inside the popup we show the formatted coordinates of the hover-location. To do this we will make use of a GeoExt event on the `GeoExt.component.Map` and of the class `GeoExt.component.Popup`.

Exercises

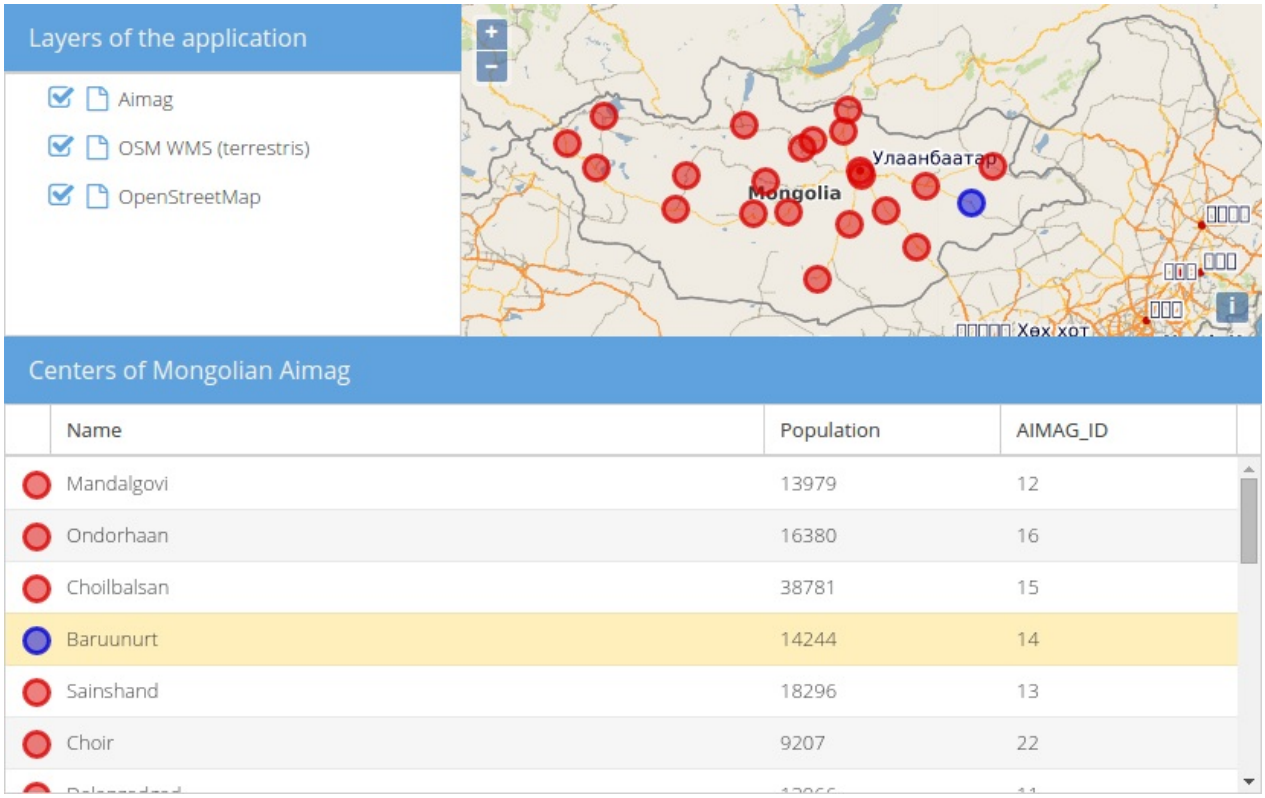
- Prepare the `map.html` file to contain the following code. This is basically the result of the previous chapters:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/t
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){

    var redStyle = new ol.style.Style({
        image: circle = new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(220, 0, 0, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(220, 0, 0, 0.8)',
                width: 3
            }),
            radius: 8
        })
    });
    var blueStyle = new ol.style.Style({
        image: circle = new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(0, 0, 220, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(0, 0, 220, 0.8)',
                width: 3
            }),
            radius: 8
        })
    })


    var vectorLayer = new ol.layer.Vector({
        source: new ol.source.Vector({
            url: 'data/aimag-centers.json',
            format: new ol.format.GeoJSON()
        }),
        name: 'Aimag',
        style: redStyle
    });

    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
```

```
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM(),
                name: 'OpenStreetMap'
            }),
            new ol.layer.Tile({
                source: new ol.source.TileWMS({
                    url: 'http://ows.terrestris.de/osm/service',
                    params: {
                        layers: 'OSM-WMS'
                    }
                }),
                name: 'OSM WMS (terrestris)'
            }),
            vectorLayer
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 4
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map,
        region: 'center'
    });

    var treeStore = Ext.create('GeoExt.data.store.LayersTree', {
        layerGroup: map.getLayerGroup()
    });

    var layerTreePanel = Ext.create('Ext.tree.Panel', {
        title: 'Layers of the application',
        width: 300,
        region: 'west',
        store: treeStore,
        rootVisible: false,
        viewConfig: {
            plugins: { ptype: 'treeviewdragdrop' }
        }
    });

    var featureStore = Ext.create('GeoExt.data.store.Features', {
        layer: vectorLayer,
        map: map
    });
    var featureGrid = Ext.create('Ext.grid.Panel', {
        store: featureStore,
        region: 'south',
        title: 'Centers of Mongolian Aimag',
        columns: [
            {xtype: 'gx_symbolizercolumn', width: 30},
            {text: 'Name', dataIndex: 'NAME', flex: 3},
            {text: 'Population', dataIndex: 'POP', flex: 1},
            {text: 'AIMAG_ID', dataIndex: 'AIMAG_ID', flex: 1}
        ],
        listeners: {
            selectionchange: function(sm, selected) {
                // reset all selections
                featureStore.each(function(rec) {
                    rec.getFeature().setStyle(null);
                });
                // highlight grid selection in map
                Ext.each(selected, function(rec) {
                    rec.getFeature().setStyle(blueStyle);
                });
                // update the grid rendering of the geometry
                sm.view.refresh();
            }
```

```
        },
        height: 300
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'border',
        items: [
            mapComponent,
            layerTreePanel,
            featureGrid
        ]
    });
});
        </script>
    </body>
</html>
```

- If you open this file in a browser (/map.html), the application should look like in the following image:



Our starting point

- For popups to look right, we need some CSS. Include the following in the `<head>` of the page:

```
<link rel="stylesheet" href="http://geoext.github.io/geoext3/master/resources/css/gx-popup.css" type="text/css">
<style>
.gx-popup p {
    padding: 5px 5px 0 5px;
    border-radius: 7px;
    background-color: rgba(255,255,255,0.85);
    border: 3px solid white;
    margin: 0;
    text-align: center;
}
</style>
```

- Configure the existing `GeoExt.component.Map` with `pointerRest: true`. Only if this configuration is `true`, the map-component will emit the `pointerrest` & `pointerrestout` events.
- Read the documentation for `pointerrest` & `pointerrestout`
- Register an event-listener on `pointerrest` that logs the hovered coordinate. Use the OpenLayers utility methods `ol.coordinate.toStringHDMS` and `ol.proj.transform` on the `evt.coordinate` to format the coordinate.
-

Hint

```
var mapComponent = Ext.create('GeoExt.component.Map', {
    map: map,
    region: 'center',
    pointerRest: true
});
mapComponent.on('pointerrest', function(evt) {
    var coordinate = evt.coordinate;
    var lonlat = ol.proj.transform(coordinate, 'EPSG:3857', 'EPSG:4326')
    var hdms = ol.coordinate.toStringHDMS(lonlat);
    console.log(hdms);
});
```

- Once you have the above logging of the coordinate in place, we can now create a popup.
- Instantiate the class `GeoExt.component.Popup` and configure it with your map. You should also provide a `width`. Store the popup in an accessible variable; `var popup` is a good choice.
- Inside of the configured callback on `pointerrest` we now want to update the HTML of the popup (method `setHtml`) and reposition it at the coordinate (method `position`, pass the coordinates in the view projection). Finally, `show` the popup.
-

Hint

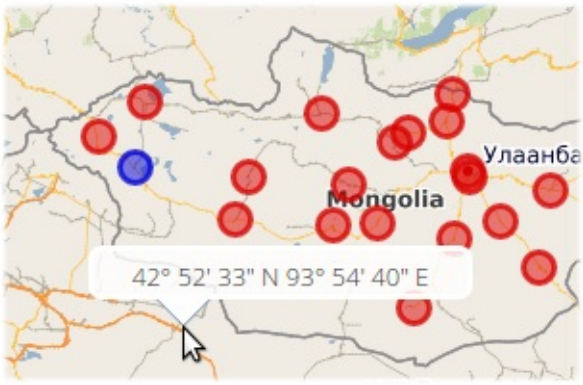
```
mapComponent.on('pointerrest', function(evt) {
    var coordinate = evt.coordinate;
    var lonlat = ol.proj.transform(coordinate, 'EPSG:3857', 'EPSG:4326')
    var hdms = ol.coordinate.toStringHDMS(lonlat);
    popup.setHtml('<p>' + hdms + '</p>');
    popup.position(coordinate);
    popup.show();
});
```

- You may notice that the popup stays in place if the mouse leaves the map viewport, which is undesired in most of the cases. Use the event `pointerrestout` to `hide` the popup whenever the mouse leaves the map.
-

Hint

```
mapComponent.on('pointerrestout', popup.hide, popup);
```

- Congratulations, you can now happily hover anywhere on the map and be greeted with the hovered coordinate:



A popup for a hovered location

Overview map

Especially when zoomed in, it can be hard to understand the extent of the mappanel. Overview-maps, which show the extent of the main map on a smaller scale can be very useful then. GeoExt comes with a useful component to create overviews: `GeoExt.component.OverviewMap`.

Exercises

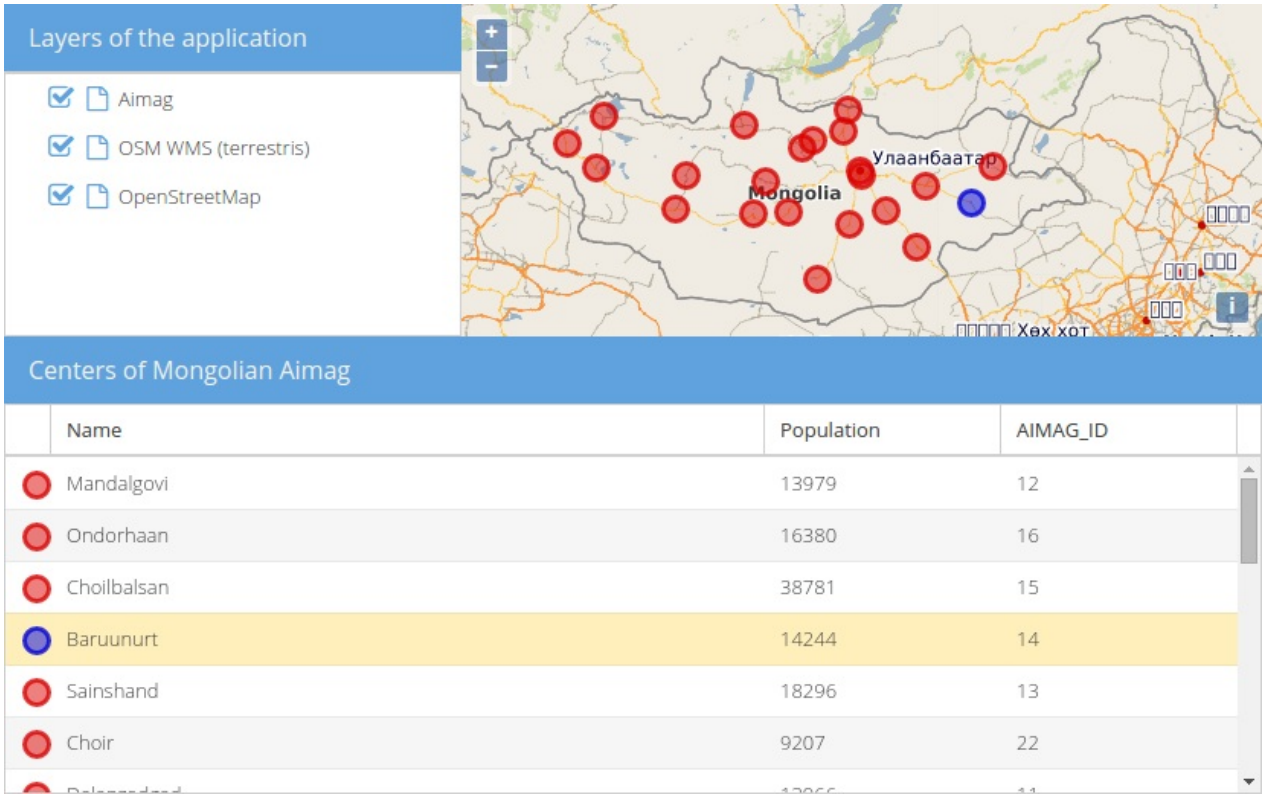
- We'll start again with the code of `map.html` from the previous sections. It's already some lines long:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Exercise | GeoExt Workshop</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/classic/theme-triton/resources/
        <script src="https://cdnjs.cloudflare.com/ajax/libs/extjs/6.0.0/ext-all.js" type="text/javascript"></script>
        <link rel="stylesheet" href="./lib/ol/ol.css" type="text/css">
        <link rel="stylesheet" href="http://geoext.github.io/geoext3/master/resources/css/gx-popup.css" type="text/css">
        <script src="./lib/ol/ol.js" type="text/javascript"></script>
        <script src="https://geoext.github.io/geoext3/master/GeoExt.js" type="text/javascript"></script>
        <style>
.gx-popup p {
    padding: 5px 5px 0 5px;
    border-radius: 7px;
    background-color: rgba(255,255,255,0.85);
    border: 3px solid white;
    margin: 0;
    text-align: center;
}
        </style>
    </head>
    <body>
        <script>
var map;
Ext.onReady(function(){

    var redStyle = new ol.style.Style({
        image: circle = new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(220, 0, 0, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(220, 0, 0, 0.8)',
                width: 3
            }),
            radius: 8
        })
    });
    var blueStyle = new ol.style.Style({
        image: circle = new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(0, 0, 220, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(0, 0, 220, 0.8)',
                width: 3
            }),
            radius: 8
        })
    })


    var vectorLayer = new ol.layer.Vector({
        source: new ol.source.Vector({
```

```
            url: 'data/aimag-centers.json',
            format: new ol.format.GeoJSON()
        }),
        name: 'Aimag',
        style: redStyle
    });

    // 1) OpenLayers
    //
    // Create an instance of an OpenLayers map:
    map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM(),
                name: 'OpenStreetMap'
            }),
            new ol.layer.Tile({
                source: new ol.source.TileWMS({
                    url: 'http://ows.terrestris.de/osm/service',
                    params: {
                        layers: 'OSM-WMS'
                    }
                }),
                name: 'OSM WMS (terrestris)'
            }),
            vectorLayer
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat( [106.92, 47.92] ),
            zoom: 4
        })
    });

    // 2) GeoExt
    //
    // Create an instance of the GeoExt map component with that map:
    var mapComponent = Ext.create('GeoExt.component.Map', {
        map: map,
        region: 'center',
        pointerRest: true,
        pointerRestInterval: 750,
        pointerRestPixelTolerance: 5
    });

    var popup = Ext.create('GeoExt.component.Popup', {
        map: map,
        width: 200
    });

    // Add a pointerrest handler to the map component to render the popup.
    mapComponent.on('pointerrest', function(evt) {
        var coordinate = evt.coordinate;
        var lonlat = ol.proj.transform(coordinate, 'EPSG:3857', 'EPSG:4326')
        var hdms = ol.coordinate.toStringHDMS(lonlat);
        popup.setHtml('<p>' + hdms + '</p>');
        popup.position(coordinate);
        popup.show();
    });

    // hide the popup once it isn't on the map any longer
    mapComponent.on('pointerrestout', popup.hide, popup);

    var treeStore = Ext.create('GeoExt.data.store.LayersTree', {
        layerGroup: map.getLayerGroup()
    });

    var layerTreePanel = Ext.create('Ext.tree.Panel', {
        title: 'Layers of the application',
        width: 300,
        region: 'west',
        store: treeStore,
        rootVisible: false,
```

```
        viewConfig: {
            plugins: { ptype: 'treeviewdragdrop' }
        }
    });

    var featureStore = Ext.create('GeoExt.data.store.Features', {
        layer: vectorLayer,
        map: map
    });
    var featureGrid = Ext.create('Ext.grid.Panel', {
        store: featureStore,
        region: 'south',
        title: 'Centers of Mongolian Aimag',
        columns: [
            {xtype: 'gx_symbolizercolumn', width: 30},
            {text: 'Name', dataIndex: 'NAME', flex: 3},
            {text: 'Population', dataIndex: 'POP', flex: 1},
            {text: 'AIMAG_ID', dataIndex: 'AIMAG_ID', flex: 1}
        ],
        listeners: {
            selectionchange: function(sm, selected) {
                // reset all selections
                featureStore.each(function(rec) {
                    rec.getFeature().setStyle(null);
                });
                // highlight grid selection in map
                Ext.each(selected, function(rec) {
                    rec.getFeature().setStyle(blueStyle);
                });
                // update the grid rendering of the geometry
                sm.view.refresh();
            }
        },
        height: 300
    });

    // 3) Ext JS
    //
    // Create a viewport
    var vp = Ext.create('Ext.container.Viewport', {
        layout: 'border',
        items: [
            mapComponent,
            layerTreePanel,
            featureGrid
        ]
    });
});
        </script>
    </body>
</html>
```

- If you open this file in a browser (/map.html), the application should look like in the following image, but you should also be able to see popups when hovering over a map location:



Our starting point

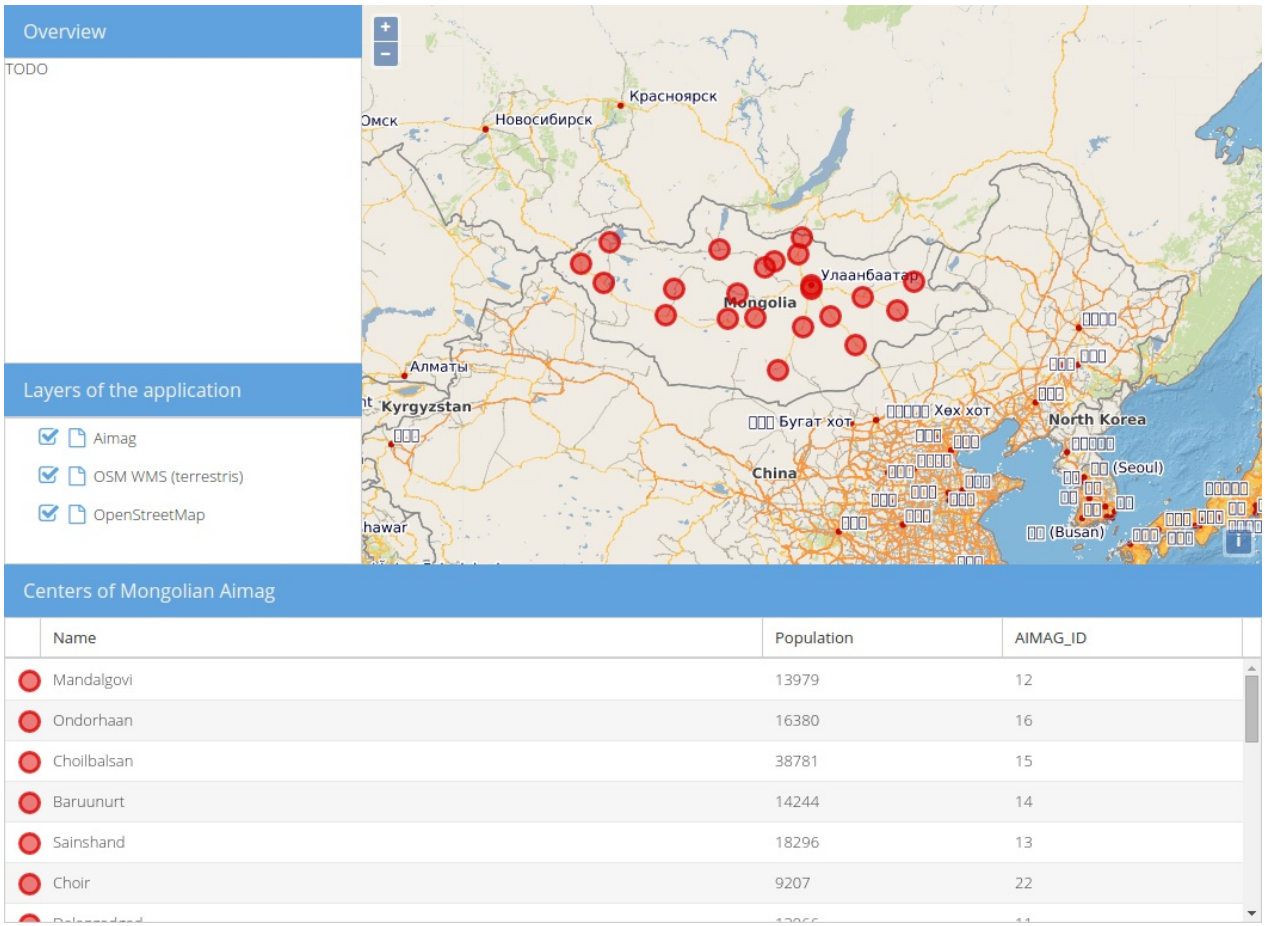
- We want the overview map to live in the top-left corner of our application, right above the layer tree. For this we will — as usual — first prepare the layout before we use the GeoExt component.
- Create a new panel that we will eventually replace with the overview, but don't add it anywhere yet:

```
var overviewPanel = Ext.create('Ext.panel.Panel', {
    title: 'Overview',
    layout: 'fit',
    html: 'TODO',
    height: 300,
    width: 300
});
```

- Instead of assigning the `region: 'west'` to the layer tree panel, we'll create a new `container` with the `vbox` -layout and pass that to the `items` of the `Ext.container.Viewport` :

```
var vp = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [
        mapComponent,
        // below is the new wrapping container:
        {
            xtype: 'container',
            region: 'west',
            layout: 'vbox',
            collapsible: true,
            items: [
                overviewPanel,
                layerTreePanel
            ]
        },
        featureGrid
    ]
});
```

- If we specify `flex: 1` for the `layerTreePanel` (the `region` -property is no longer needed), your application should look like this:



The prepared layout

- Now it is time to use `GeoExt.component.OverviewMap` : Create an instance of this class and read the related API-docs.
- Configure the `overviewPanel` with the created overview instead of `html: 'TODO'` (via `items`).
- You may want to have another layer in the overview. How about this WMS?

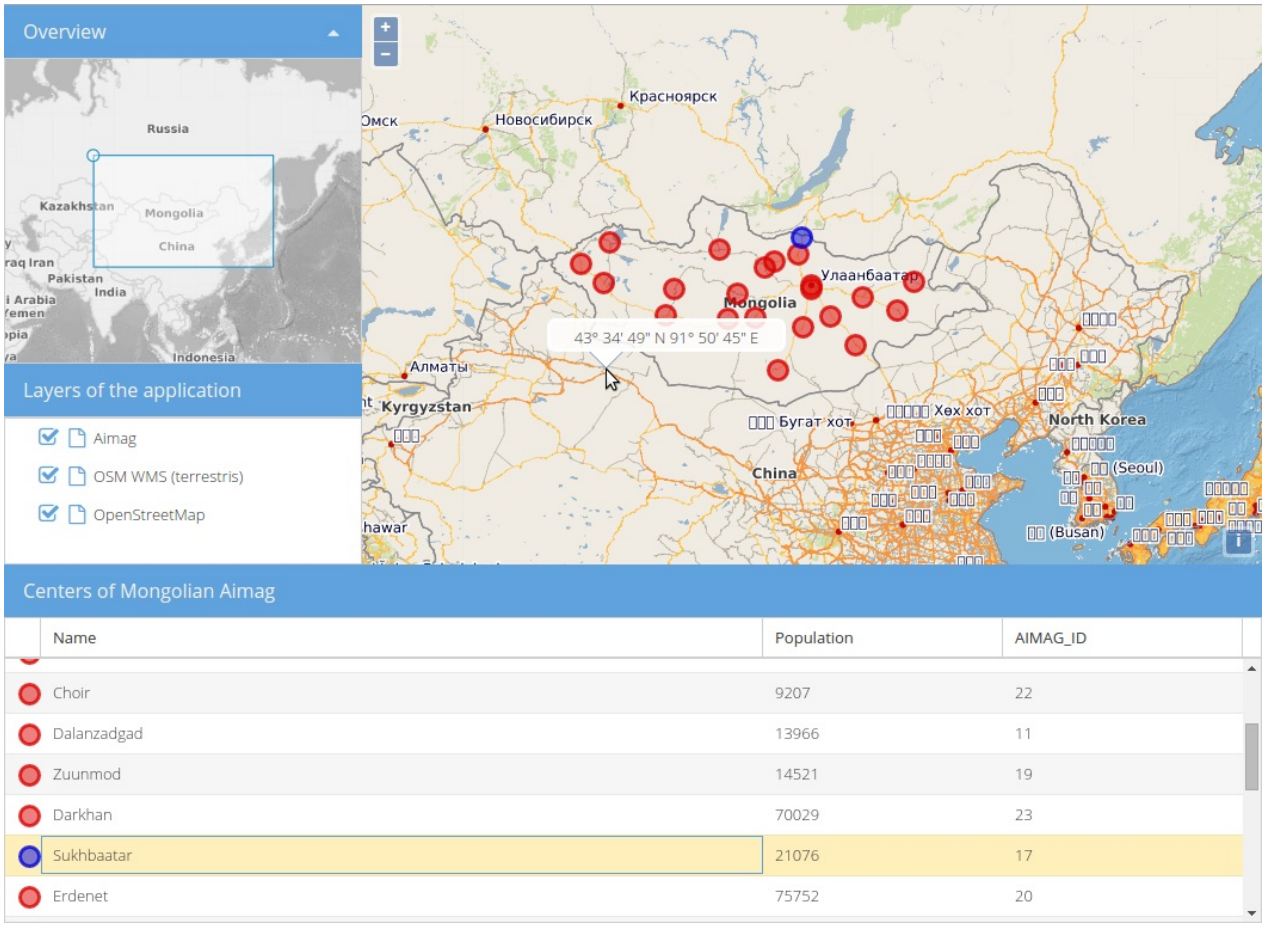
```
URL: http://ows.terrestris.de/osm-gray/service
Layers: OSM-WMS
```

-

Hint

```
var overview = Ext.create('GeoExt.component.OverviewMap', {
    parentMap: map,
    layers: [
        new ol.layer.Tile({
            source: new ol.source.TileWMS({
                url: 'http://ows.terrestris.de/osm-gray/service',
                params: {
                    layers: 'OSM-WMS'
                }
            }),
            opacity: 0.8
        })
    ]
});
var overviewPanel = Ext.create('Ext.panel.Panel', {
    title: 'Overview',
    layout: 'fit',
    items: overview,
    height: 300,
    width: 300,
    collapsible: true
});
```

- If everything went well, you should see an application like below:



The final application

Other & summary

Congratulations! You created quite an application with just around 200 lines of JavaScript; and that includes plenty of comments and whitespace.

GeoExt still has more to offer.

We couldn't talk about legends in the tree, the super useful `GeoExt.OlObject` -class or the `PrintProvider` which allows you to serialize your map to a format understandable by the superb Mapfish Print Servlet (v3).

Make sure to checkout all examples, the API documentation (also available with ExtJS-classes) and the code on github.

The source for this workshop is also on github. If you find an error or outdated section, just open an issue or — even better — provide us with a pull request.

We hope you like what you have learned.

Synopsis

Congratulations! You've successfully finished the workshop "Introduction to core technologies behind the MoMo geoportal" 🎉!