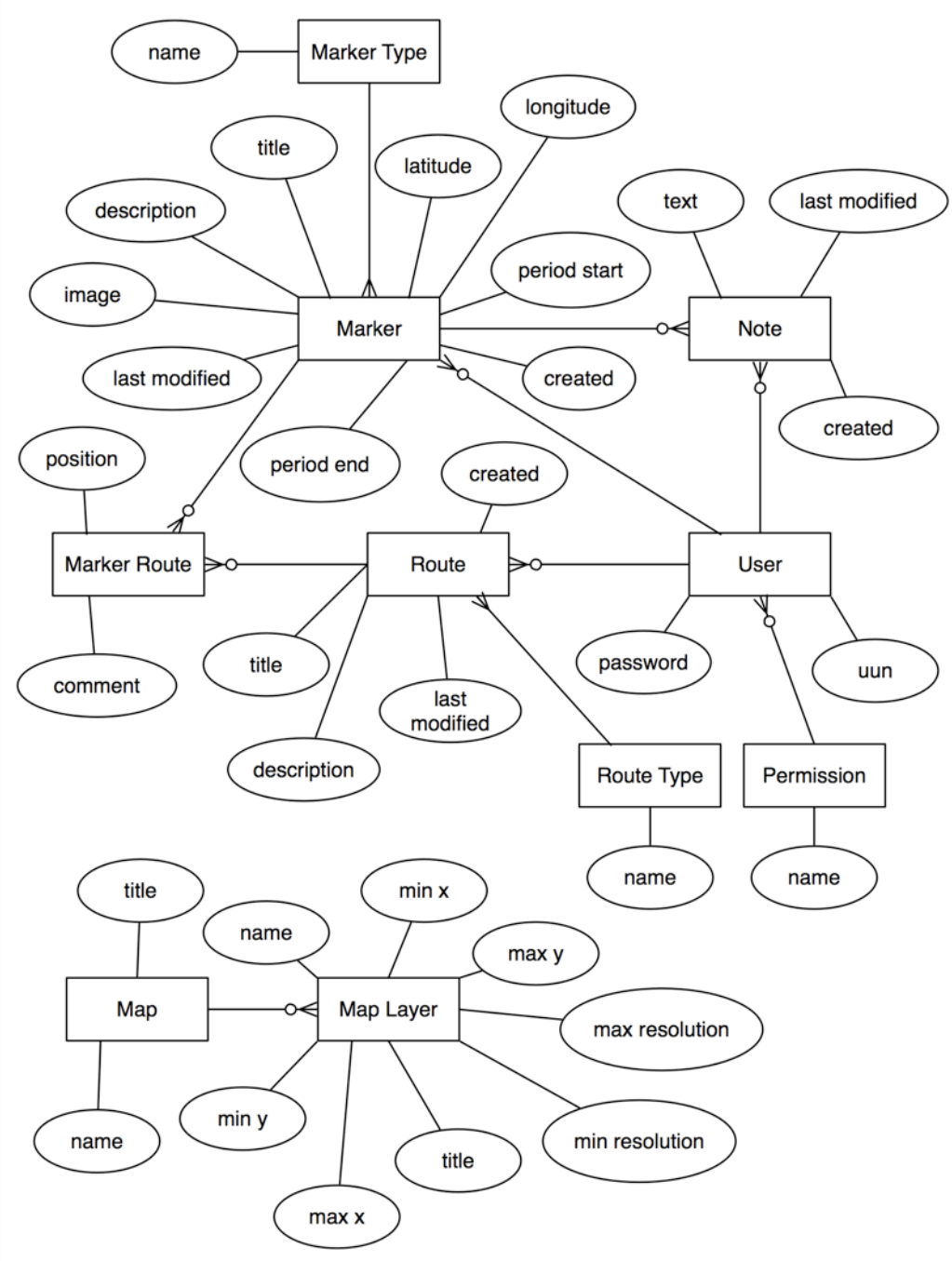
Walking Through Time

Technical Report

Database

Overview



Relational Database Model

Map & Map Layer

The historical maps' master data is stored in two database tables, i.e. Map and Map Layer. A map only has a name and a display title, whereas Map Layers are more complex. Besides name and display title they also contain a minimum and maximum resolution attribute defining a map layer's individual display range. Additionally the map layer's boundaries are determined by the coordinates of the map layer's lower left corner (min x & min y) and upper right corner (max x, max y).

Marker

The marker table stores places of interest on the map. In addition to the basic data like title, description, location (as latitude and longitude), etc. an image URL, a time period, and notes. Time period provides the possibility to indicate when the marked place of interest was e.g. built, demolished, etc.). Notes are kept in a separate table and store users' comments on markers.

Route

A route represents a predefined walk someone could take leading him/her around places of interest. Thus, a route is made up of markers connected in a specific order.

Java

Overview

Walking through Time is a pure web application designed for but not restricted to iPhones with 3.0 firmware. The application is location aware; location data can be requested from the devices GPS and made available to the browser. The application runs on any device and browser supporting JavaScript. In order to provide a richer user experience the browser has to implement W3Cs GeoLocation.

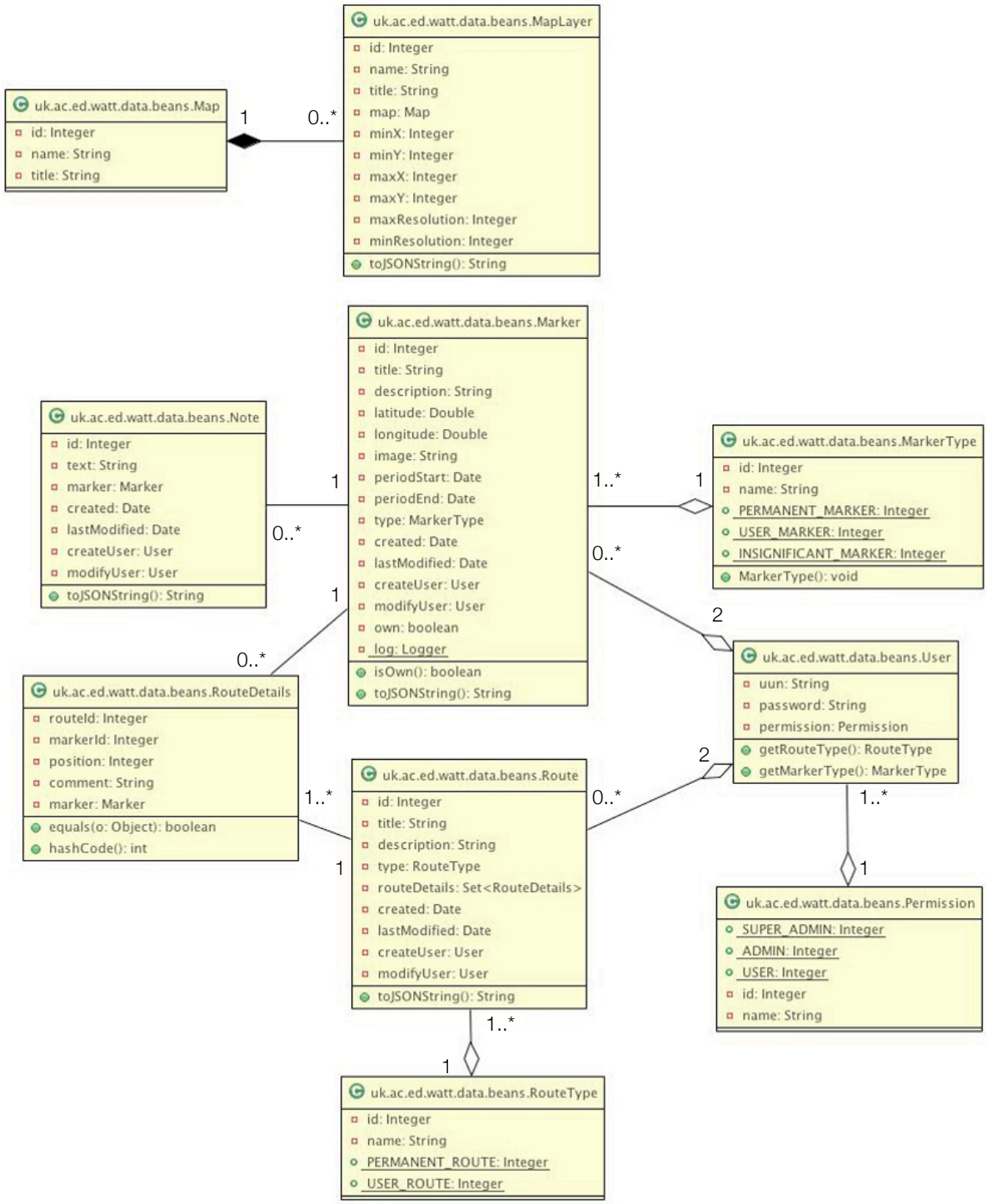
The application was implemented in Java, using Spring and JavaServer Faces (JSF) in combination with Facelets.

Spring (http://www.springsource.org) provides a consistent means of configuring and managing objects as well as their lifecycles.

The JSF web application framework was chosen because it simplifies development and design of server side enterprise Java applications. JSF eases the definition of navigation flows within a web application and enables the developer to compose user interfaces with standard UIcomponents.

Object Relational Mapping

Hibernate is used to enable database independence and highly efficient relational persistence. It allows for creating persistent classes by providing a mapping between the relational database and the object oriented domain model.

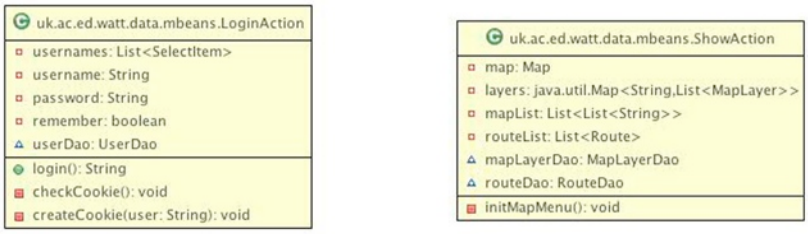


Class diagram reflecting object relational mapping

Managed Beans

Walking through Time consists of two JSF pages that are backed by the following managed Beans:

- Login Action
  handles user authentication and session creation.
- Show Action
  responsible for retrieving navigation content (i.e. map master data, routes)

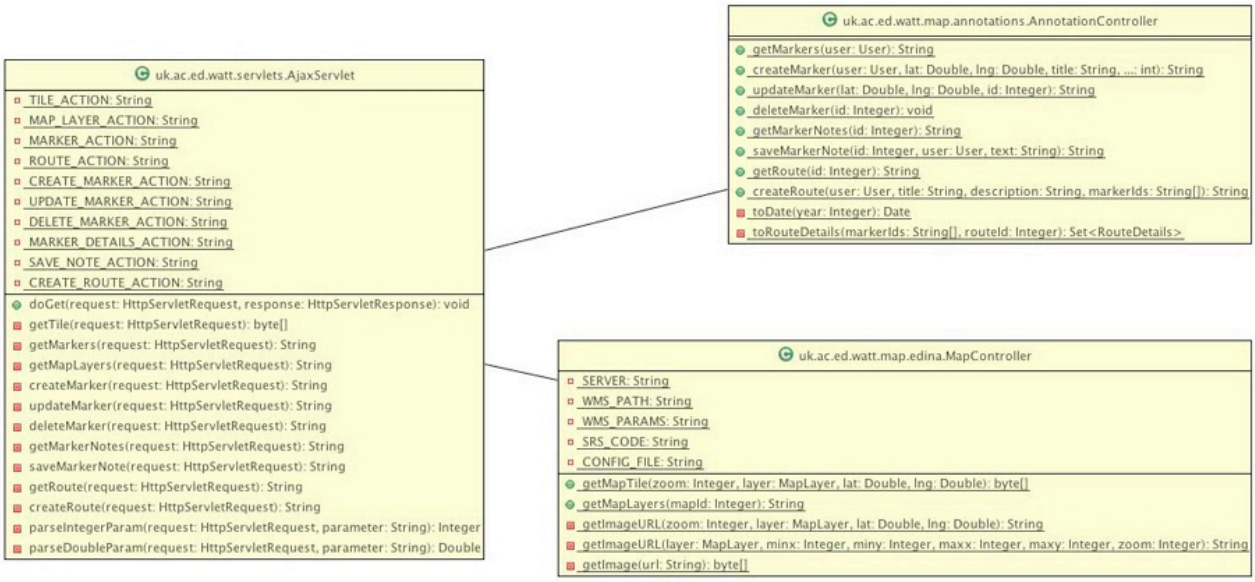


Managed Beans

Application Logic

In order to create a highly responsive application, AJAX is used for communicating between server and client whenever possible. This means that, apart from the login page, there exists only one other page with all the main functionality.
The AjaxServlet serves as an interface for all client-side Ajax requests and dispatches control to one of the following controller classes:

- Map Controller:
  handles all incoming requests for map tiles and map layers.
  Since all historic maps are made available by a Web Mapping Server the application acts as a proxy between this server and the front end.
  When Google Maps are overlaid the historic maps are requested as a number of tiles differing in their centre coordinates. On the Server the centres' latitudes and longitudes are then converted to national grid coordinates in order to pass the requests on to a Web Mapping Server. The received map images are then passed straight through to the front end.
- Annotation Controller:
  handles all incoming requests that deal with markers, notes, or routes.
  Processing these requests always involves communication to the database for storing and retrieving markers, notes or routes as well as updating and deleting markers.
  Objects are sent to and from the server in JavaScript Object Notation (JSON).



AjaxServlet and Controller Classes

JavaScript

Overview

As mentioned before Javascript and Ajax are used in order to enhance the application's response time and reduce traffic to/ from the server.

Rapid development was one of the major project aims. For this reason the Google Maps API has been utilized since it provides basic and extensible map functionality like navigation, zooming, overlaying maps with markers etc. Without such an API it would not have been possible to implement WATT in such a short time-frame.

The starting point for rapid development was a pre-existing map API so that basic functionality like navigation, zooming etc. had not to be implemented from scratch. Therefore Google Maps API was used. It provides a great deal of useful features including map tiling, all kinds of markers, polylines,... and is flexible enough to allow for customisation.
Hence, most of the application's functionality is based on Google Maps API as well the Prototype JavaScript framework.

Similar to the database and Java design also the JavaScript logic builds on the following objects:

- Map
- Map Layer
- Marker
- Route.

Map & Map Layer Functions

Most important functions regarding maps and map layers:

function loadMap() {...}
First of all the Google Maps API has to be initialized and loaded and events have to be registered.

function setUserPosition() {...}
On load the application is trying to retrieve the device's geo location in order to mark it with a blue dot on the map. If no geo location can be retrieved from the device, EDINA serves as the default location.

function changeMap(mapTitle, id, mapId) {...}
This function overlays the Google map with the specified historic map.

function showQuickMenu() {...}
If the user selects a historic map, a quick menu will be generated which allows for convenient navigation between maps of different time periods. (Namely, all map layers corresponding to the current map set.)

function getMapLayers(id) {...}
This function for getting all available layers of the current map set is required to generate the quick menu on the screen.

function changeHeaders() {...}
If the map overlay changes, the head line which contains the map layer's title has to be updated.

function addEvents(gMap) {...}
The zoom level changed event is registered in order to update the user's position.

function createPositionMarker() {...}
The marker (blue dot) that marks the user's position on the map is created in accordance to the current zoom level.

function refreshPosition() {...}
There is a button on the screen that allows to re-centre the map to the user's current position.

function toggleGoogle() {...}
In order to show/hide the historic map's underlying Google map the opacity of a plain, white layer between those two maps is changed.

function removeLayer(mapTitle, id) {...}
If a historic map is being displayed, it can be removed thus bringing the user back to the modern time's view.

function toggleFollowUser() {...}
This function enables/disables user position updating. If enabled, the user position is monitored and the user position marker updated on the map.

Marker Functions

Each Marker contains three different display types for different purposes. These types differ in appearance and behaviour.

- GMarker: A GMarker is a basic marker mainly for marking a place of interest.
- EMarker: If the user decides to edit an existing marker or create a new marker an EMarker is used. It's red, draggable and can be deleted or repositioned.
- RMarker: RMarkers become visible when the user chooses to create a route. These markers look like GMarkers but change their appearance to red pins with black letters on them when clicked, so that the user can keep track of the marker's order within the route.

Most important functions regarding markers:

function toggleMarkers() {...}
Markers can be displayed on the map or hidden again.

function convertMarkers(xhr) {...}
After requesting markers from the server an array of markers (in JavaScript Object Notation) has to be converted into an array of marker objects.

function editMarker(id) {...}
If the user wants to edit a marker its type changes to an EMarker.

function createMarker() {...}
After the user has completed a corresponding form and decided to create a newly defined marker, all marker details are sent to the server for actual creation.

function updateMarker(id) {...}
This function updates the coordinates of a repositioned marker.

function positionMarker() {...}
After the user has defined a new marker's details their validity is checked and a draggable marker is put on the map for positioning.

function discardEMarker() {...}
When positioning a new marker the user might decide to discard it instead of saving it.

function deleteMarker(id) {...}
A marker can be deleted by the user who owns it.

function setMore(id) {...}
If there is additional information available for a marker, the user has the option to view more details. The detail page is prepared (and this function called) when the user clicks a marker's 'More...' button.

function saveMarkerNote(id) {...}
Users are provided with the possibility to share their thoughts/knowledge etc. by commenting on markers.

function getMarkerNotes(id) {...}
Request the latest ten user comments for a specific marker.

Route Functions

A Route is basically a polyline connecting markers in a certain order.

Most important functions regarding routes:

function getRoute(id) {...}
Retrieves specified route from the server.

function convertRoute(xhr) {...}
Once the server returned the requested route (in JavaScript Object Notation) it has to be convert to displayable markers connected via a polyline.

function rmRoute() {...}
Hides currently displayed routes.

function defineRoute() {...}

In the course of defining a new route the user has to select the route's markers and complete the corresponding marker creation form. Therefore the previously mentioned RMarkers are displayed on the map.

function createRoute() {...}

When the user finished defining a new route, all details are sent to the server for actually creating the route.