URLCHOPPER TECHNICAL DOCUMENTATION

Table of contents

1. Summary

The application's main goal is to provide a URL shortener service. It is a web application with a single webpage, where the user can type in (or paste) a long URL and after pressing a button a short URL is generated and is written on the page. The application supports URL redirection: the user can click on the generated link, or paste the generated short URL in the browser's address bar to be redirected to the original webpage. The user's previous short URL generations can be viewed on the front page. The user is authenticated with a cookie. An additional page is available from the menu where one can examine the statistics (which long URL is shortened how many times).

2. Technologies used

The application is based on Spring, and Spring MVC. It's dependencies are managed by Maven. The underlying database is MySql, JPA is used for persisting entities with Hibernate as the persistence provider. Logging is managed by SLF4J, which serves as a façade for the Logback logging framework.

The web pages are built with JavaServer Pages (JSP) and the Apache Tiles templating framework. The Twitter Bootstrap front-end framework is added to the project for faster development.

JUnit and EasyMock are the unit testing and mocking frameworks used.

3. Architecture

As mentioned above, the application is based on the Spring MVC framework. The views are scriptless JSPs, and the incoming HTTP requests are served by Spring controllers. These controllers call a service layer, which handles the main business logic (mainly the short URL generation). The service layer uses an ordinary repository which acts as the data access layer. It uses JPA's entity manager to persist and find entities. The domain model consists of three domain classes, which are detailed later in the documentation.

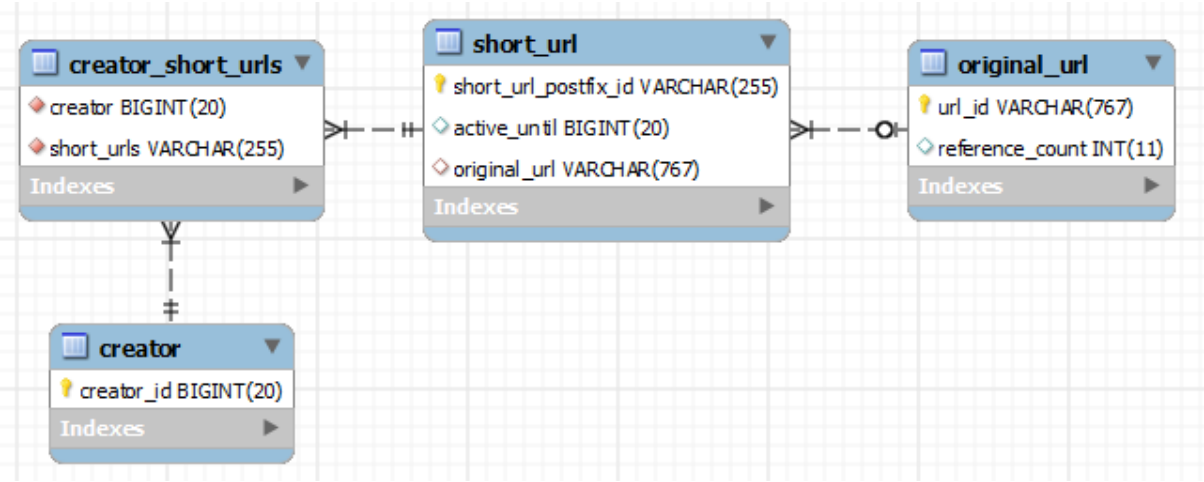
a. Domain model

original_url

Stores an original url value and a reference count. If the generated short Url was removed from the table because its lifespan ended, the reference count can still be read from the database. Records are never removed from this table.

short_url

The expire date and a foreign key to the original_url is stored in this table. When the short URLs lifespan expires, the actual record should be removed from the table (not implemented yet).



creator

This table stores only one data, who created the short URL.

creator_short_URLs – connection table

This table contents two foreign key. One of the foreign keys point to short_url, the other points to creator.

b. Controllers

IndexController

IndexController is responsible for handling every action on the application's main page.

Short URL generation – IndexController.generate()

This controller handles the generation requests after the 'Generate' button is pushed on the view. It gets the long URL as a GET request parameter and checks if the user's id is available in the session (see User handling). If not available, the user's id is set to null. Then it passes these values to the service layer for short URL generation. The returned value is the generated short URL, which is added to the Spring MVC model to be able to display it on the view. The service is autowired to the controller by Spring.

Showing user history – IndexController.index()

If the user is recognized by its cookie value (see User handling) – therefore a user id is present in the session – a short list of its previous short URL generations is shown on the main view just below the input field. It is the IndexController's responsibility to get this data from the service layer. The history service returns a list of short URL DTO-s, which is added to the model and then displayed on the view.

Redirecting short URL requests to the original URL – IndexController.redirect()

The application is able to handle the short URL requests. A typical valid short URL matches the following pattern: http://*hostname*/urlchopper/*{postfix}*, where *{postfix}* is a randomly generated 6 character long identifier. The controller handles every URL request that matches this pattern by getting the postfix from the requested URL as a path variable. It then asks the service layer for the saved original URL associated with the postfix. The returned URL is added to the model as a flash attribute because the user is first redirected to a wait page, and only after 5 seconds to the original URL.

StatisticsController

The controller's only task is to add the statistics list to the model to be able to display it on the statistics view. The statistics contains which original URL was shortened by how many times. The controller gets these information from the service layer before adding it to the model.

c. User handling

The application is able to recognize a previous user without classic authentication. To achieve this, a new cookie is added to the user's browser just before its first generation request. After the cookie is added (and before it is deleted by the user), the user's browser can be recognized based on this value. This behavior is managed through filters and session values.

CookieFilter

This filter runs before every request. It is necessary because this way it doesn't matter which page is loaded first when a new session starts, it is still known which user connected to the application. When the filter first runs, the user id (the application cookie's value) is added to the session if a valid cookie is found in the request. A cookie value is valid, if it can be parsed to long, and if it is found by the user repository as a previously saved user id. The validation is used to prevent application crashes if a user rewrites the cookie in the browser.

The filter returns immediately if the user id is already in the session, so the cookie value is read only once in a session.

If no application cookie was found in the request, this filter does nothing, because we only want to store a new user in the database if he made at least one generation request.

AddCookieFilter

This filter is responsible for adding a new user to the database when the user made its first generation request. If no session value or valid cookie was found in the request, it means that the request came in from a new user. Then a new user is created with an automatically generated id. A new cookie is also created with the generated id as cookie value, and with an application specific key. The newly created user is added to the session too.

d. Service layer

The service layer contains the main business logic of the application. It is used by the controllers, and the service layer uses the repositories to communicate with the database. 2 interfaces and their implementations belong to this layer. The UrlRepository's main task is to generate the new short URL postfixes, and the HistoryService's task is to provide data for the controllers to display the users's previous short URLs.

UrlService

UrlService – generate()

When a user hits the generate button on the webpage, the controller asks the UrlService to generate a new short URL postfix, or return a previous one if the original URL was already shortened before. This method gets the requested original URL and a user id as parameters. The user id is used only if it is not null to update the user's list of short URLs. This is done before the method returns, and when the short URL is already generated.

Three different cases can occur when generating a short URL. The first is when a short URL already exists for the requested original URL. In this case the short URLs lifespan has to be lengthened, the original URL has to be updated by increasing its reference count by one, and the short URL belonging to the original URL has to be returned. In the second case, the requested original URL exists in the database, but there is no short URL belonging to it. It can happen when the short URL is removed from the database because its lifespan has ended (comment: removing short URLs from the database has not been implemented yet). In this case original URL's reference count has to be increased by one, but a new short URL has to be generated. Generation is detailed later. In the third case, the original URL does not exist in the database, so it has to be created along with the short URL. An original URL is created by initializing its reference count to 1, and persisting it to the database.

Unique postfix generation is based on random methods. The postfix is 6 characters long, and consists of digits and lowercase letters. That gives us 36^6 (=2.176.782.336) different options which means that the generation can be completely random. The random generation means that it is possible that an already existing postfix is generated, and the generation has to run again, but its chance is really low. If 1 million active short URL is present in the database at the same time, the chance of generating an already existing one is ~1/2176, the chance of generating 2 existing ones in a row is 1/(2176*2176). This means that the chances of running the generation at least 2 times in a row is basically 0. Furthermore, the postfix length can be modified by overwriting its length in a properties file (config.properties).

UrlService – findOriginalUrlByShortUrl

This method is called when a redirection is made, and the controller has to find the original URL belonging to the short URL in the path variable. The service asks the repository for the ShortURL object, and returns the OriginalURL object that belongs to it.

UrlService – getAllOriginalUrls()

Used by the StatisticsController to return all the OriginalURL objects.

HistoryService

HistoryService – getUserUrls()

This method is called from the controller layer to provide the previous short URLs generated by the user. It gets the user id as an input parameter, and uses the UserRepository to get the corresponding user. It also transforms the ShortURLs belonging to that user to data transfer objects, and returns a list of that DTOs.

e. Repository layer

The repository layer follows the common conventions and grants an interface to create, update, find or remove entities from the database. Every domain has a corresponding repository interface, and a JPA based implementation. JPA's entity manager is injected to these implementations by the Spring container, and the transactions are also managed by Spring. We used annotation based transaction management in the implementations.