Project report

Open Data – Road accidents visualization

Noah Cohen, Tomáš Pouzar

Outline

Introduction

Our project, called SmashIT is a simple open data visualization website focused on road accidents in Italy during years from 2001 until 2015. Specifically, we display the number of cars involved in accidents and some basic graphs using this data, all of which are available on dati.istat.it.
The first thing that visitor to our website notice is a huge, easy to read map of Italy showing data for each region of Italy separately. Below the map users can find a simple slider thanks to which can easily change a data set.

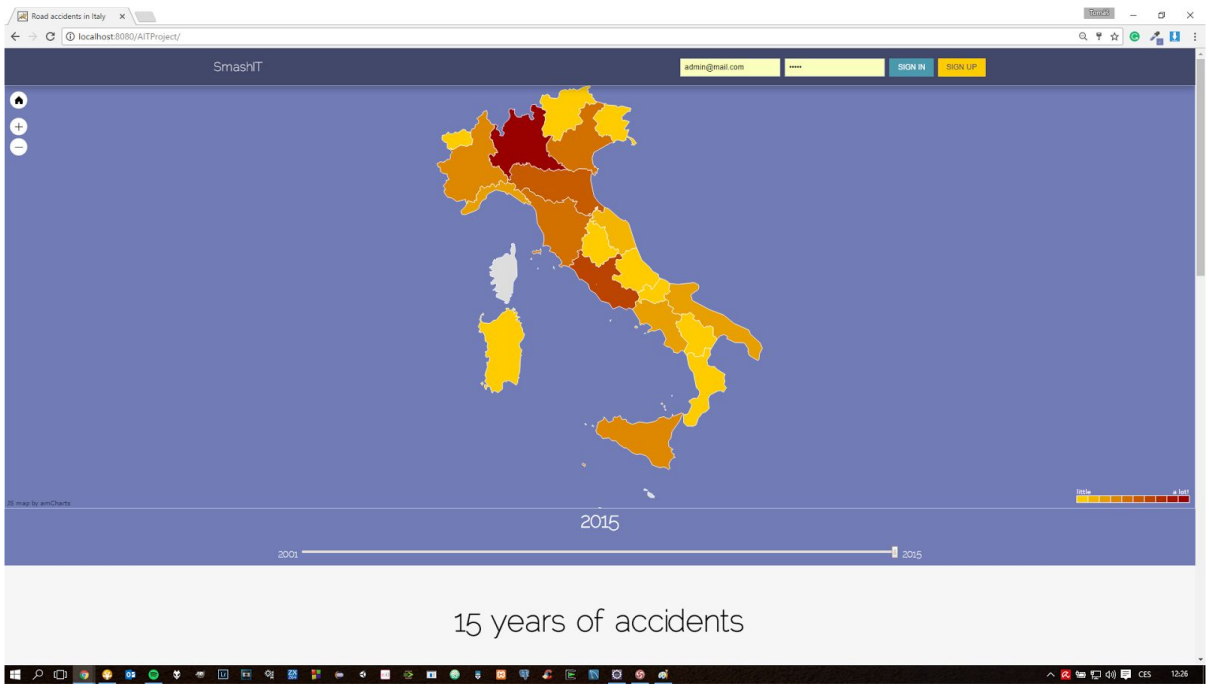

Figure 1.1: Introduction

A huge title, 15 years of accidents, below this map encourages newcomers to scroll down and explore data in more detail. Down on this page, 4 charts can be found in the meantime, which show data from entire 15 years in total. The first column chart, Top 5, shows 5 regions with the highest number of cars involved. A chart to the right of the first one, Evolution, shows how the number of accidents has been gradually reduced over the course of fifteen years. Below these two charts, another pair of can be found. The first one, 5 safest, is a radar chart showing regions with least amount of cars involved. The very last one, Summary, shows a simple pie chart, as the title suggests, including all regions and their share in accidents. All of these charts are downloadable and we also provide an opportunity for logged-in users to add these charts into a simple shopping cart and download .pdf file including charts in the shopping cart.

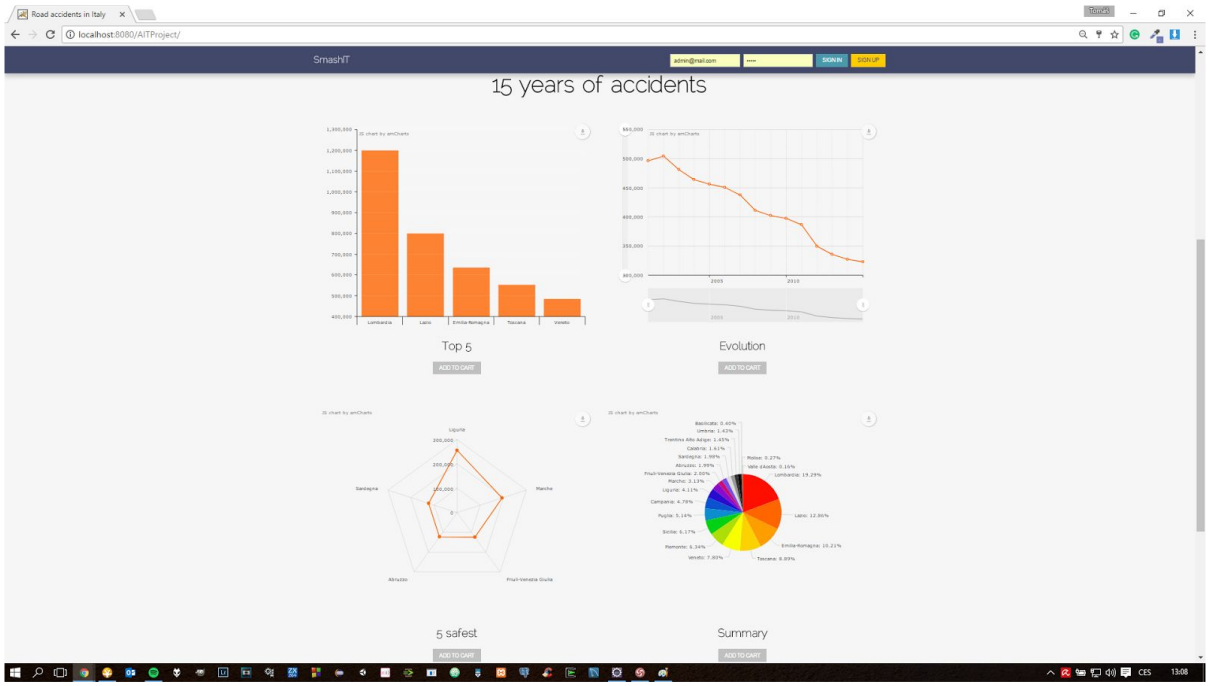

Figure 1.2: 15 years of accidents

At the bottom of the website, there is a few paragraphs briefly describing our project, including a few lines about the data set, Advanced Internet Technologies course and its professors and about us, authors of this project. There is also a button allowing all visitors to download a .csv file of all our data.

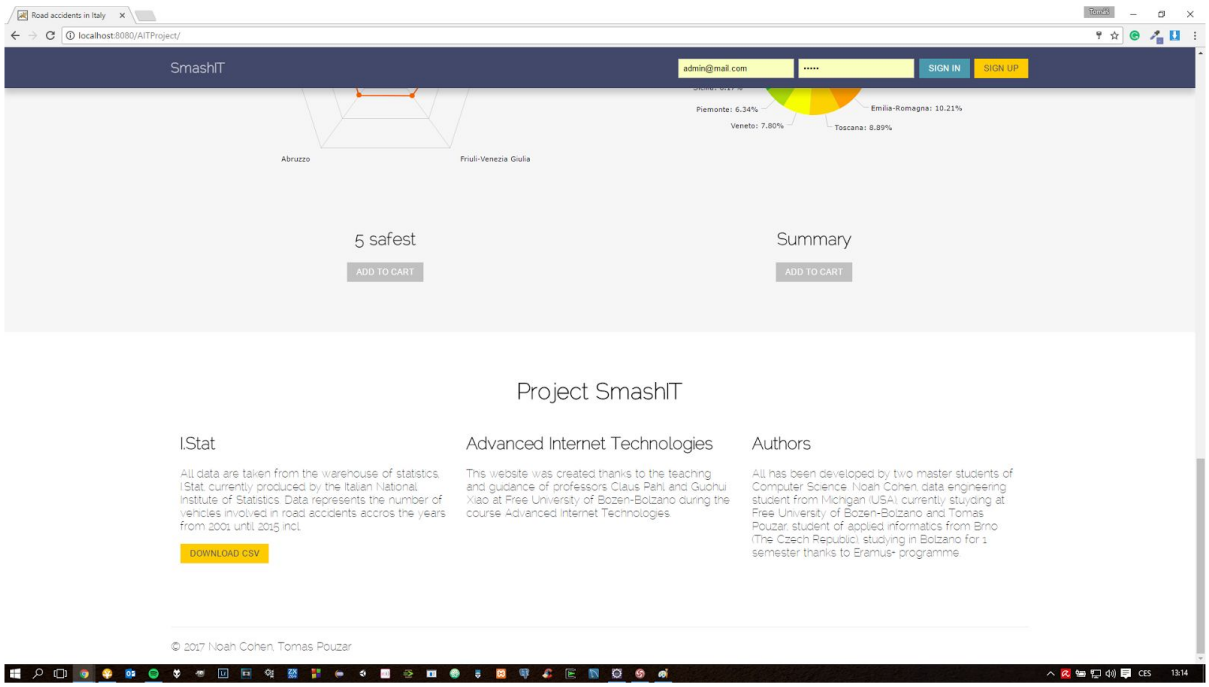

Figure 1.3: Footer

As mentioned before, there is also a possibility to create and manage accounts. Being logged-in on our website provides access to a shopping cart and the possibility of creating and downloading .pdf. Once logged-in, users can also change their registration information whenever they want.

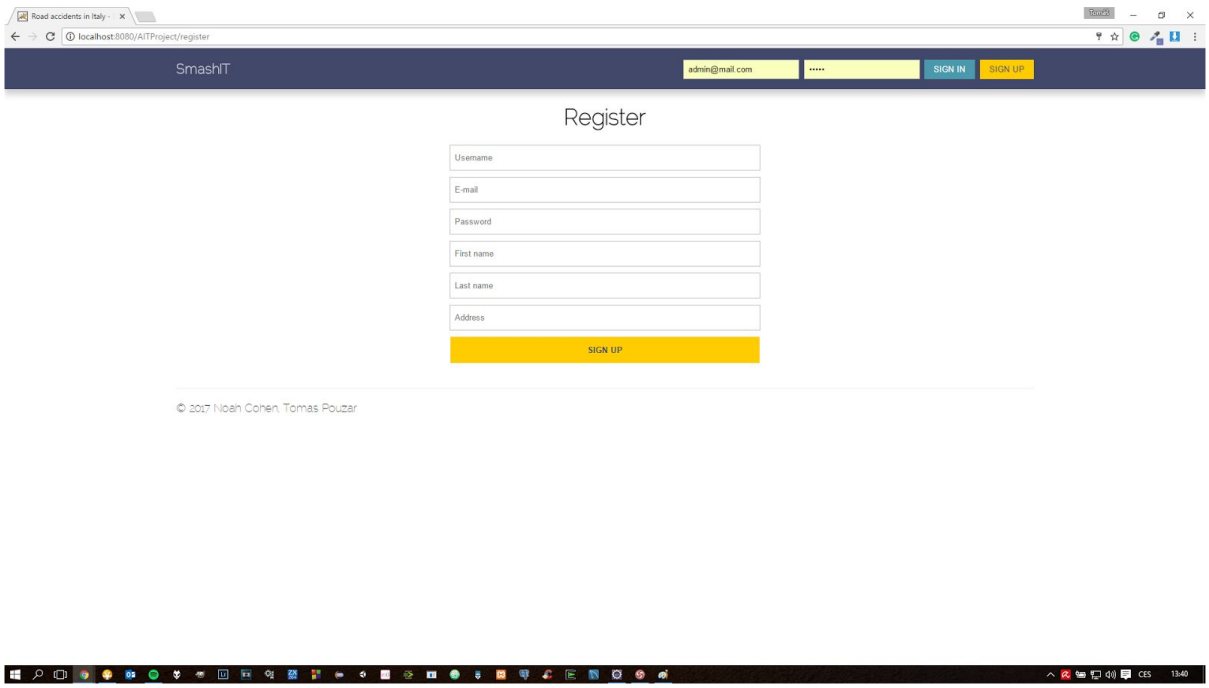

Figure 1.4: Registration form

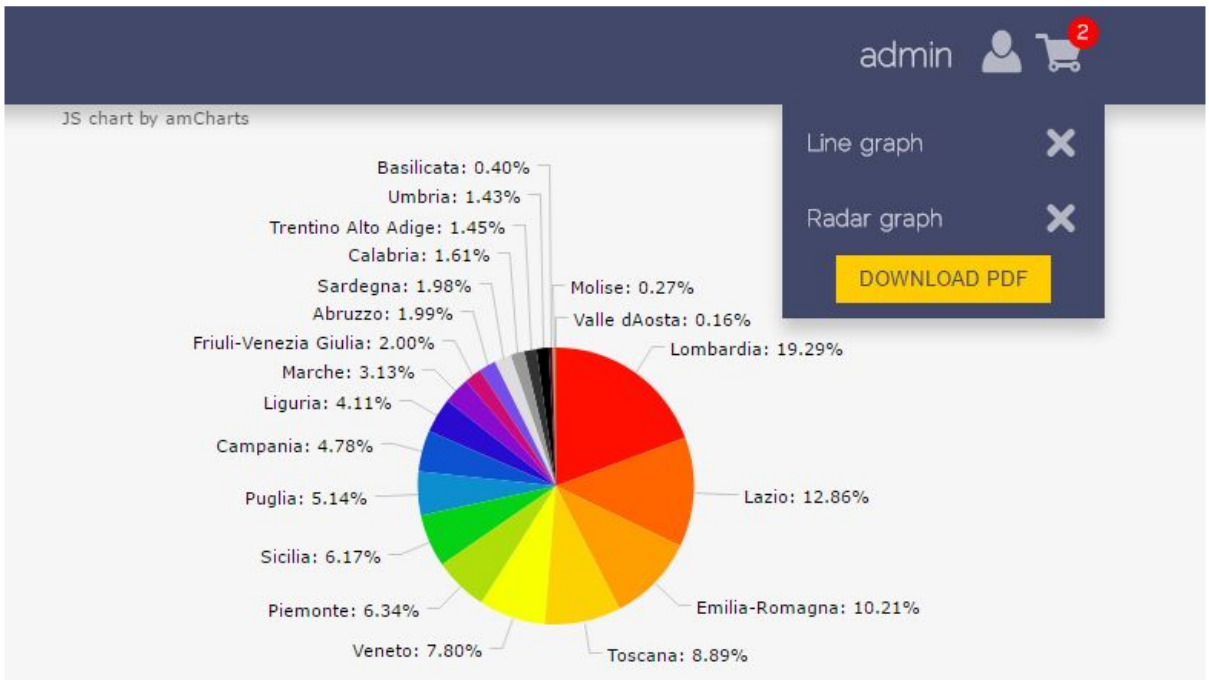

Figure 1.5: Shopping cart preview

The website is of course completely responsive on all possible devices which was tested using Responsinator.

Figure 1.6: Example of responsiveness

Architecture

The project is a standard Java servlet application running so far only locally on Tomcat server using local Postgres database to store any necessary data. Thus, the server side of this application is written mostly in Java code and SQL for communication with database. Frontend is based on JavaServer Pages using HTML, CSS, JavaScript, jQuery and Amcharts libraries for map and charts.
To implement transparent and scalable system we got inspired by MVC model widely used in website development and divided the system into 5 separate groups:

1. **Database** used just for data storage.
2. **Java models**, passive objects to be manipulated with or send to client.
3. **Java controllers** that receives requests from servlets or JavaServer pages and manipulates with database.
4. **Servlets** handle requests from client-side, hand over them to controllers and send results back to client.
5. **Views** consisting of several JSP files, CSS documents JavaScripts.

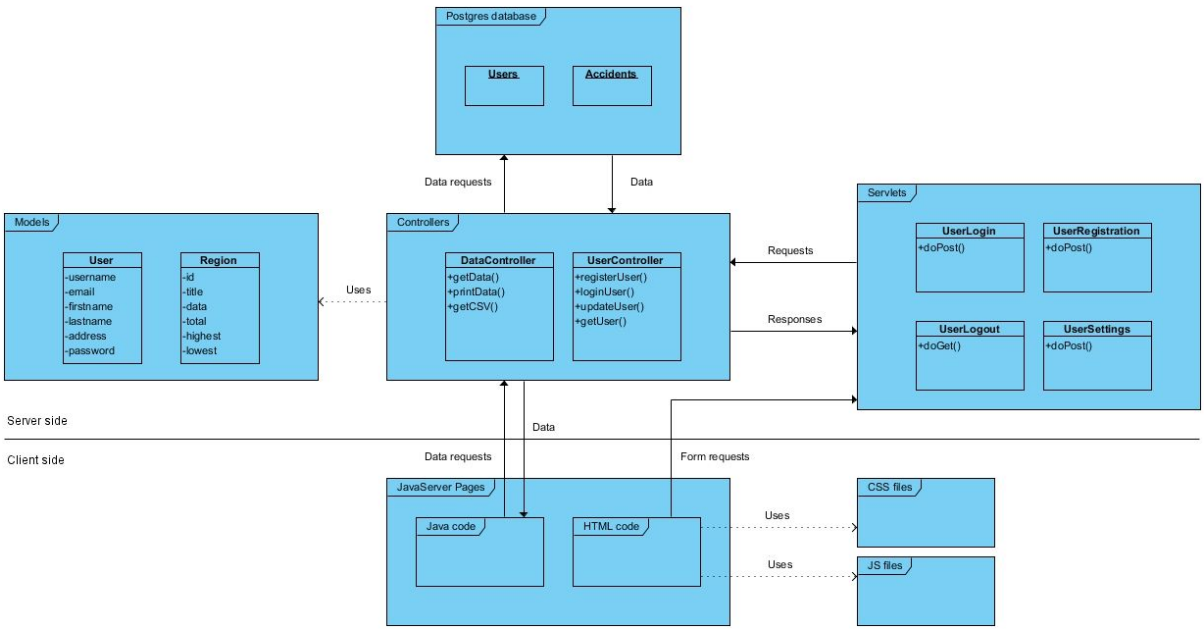

Figure 2.1: Overall architecture of our system

Database model

As mentioned before, we use a Postgres database running on a local server. At this server, there is a scheme for our project containing two tables, one called “users” for registered users, the other one “accidents” for storing data about accidents.
At first we have downloaded all data from dati.istat.it in form of csv which we have imported into our database.

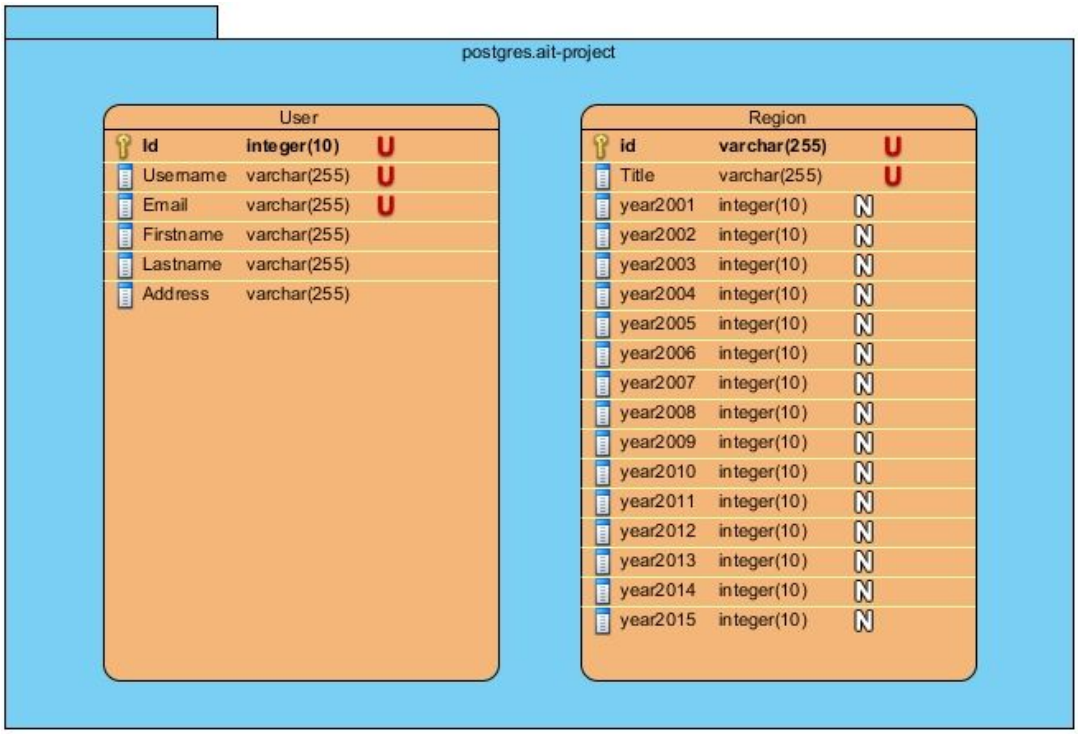

Figure 2.2: Entity diagram

Java models

We have two passive models for storing data from database.

1. **User:** The first one, mirrors data in database and doesn't provide any more functionality but getter and setter methods.

2. **Region:** Provides more information then just pure data from database. Every object of this class represents a real region in Italy, its number of accidents across the years and some additional data like the total number of cars involved in accidents in all 15 years. Each region also contains an information of highest and lowest number of all years to be used in data visualization. These additional information are calculated in the constructor of this class.

All properties of both classes are accessible by getter methods.

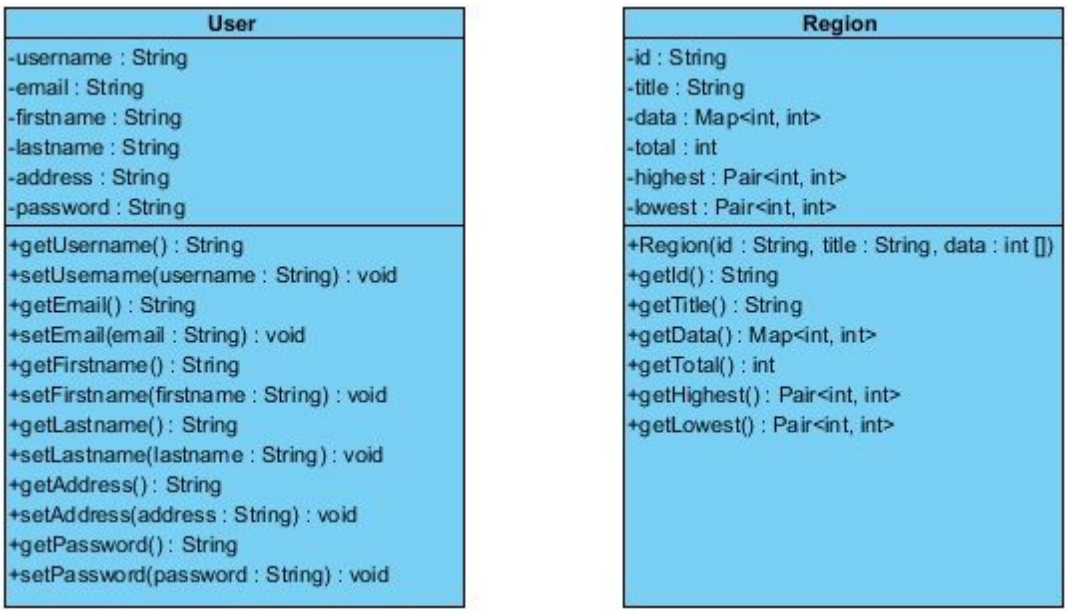

Figure 2.3: Class diagram of our models

Controllers

Both of our Java controllers are static classes and serve as a connection between client's requests and database.

1. **UserController**
    - **registerUser:** Receives data from servlet and inserts a new row into the database if username or email doesn't already exist. In that case on of exceptions is thrown (either DuplicateUsernameException or DuplicateEmailException)
    - **loginUser:** Receives an email and password from servlet, validates the input using database and if it's correct, sends back User model containing all user information.
    - **updateUser:** Receives new user information from servlet and updates the database. If it succeeds, returns User object back to servlet.
    - **getUser:** Simply returns User object according to provided email.

2. **DataController**
    - **getData:** Simply returns data from database in a form of a JSON string. There are several variations of this method eg.: return data in ascending, descending form and others.
    - **getCSV:** Returns all data in a CSV string ready to be put in a file and downloaded.

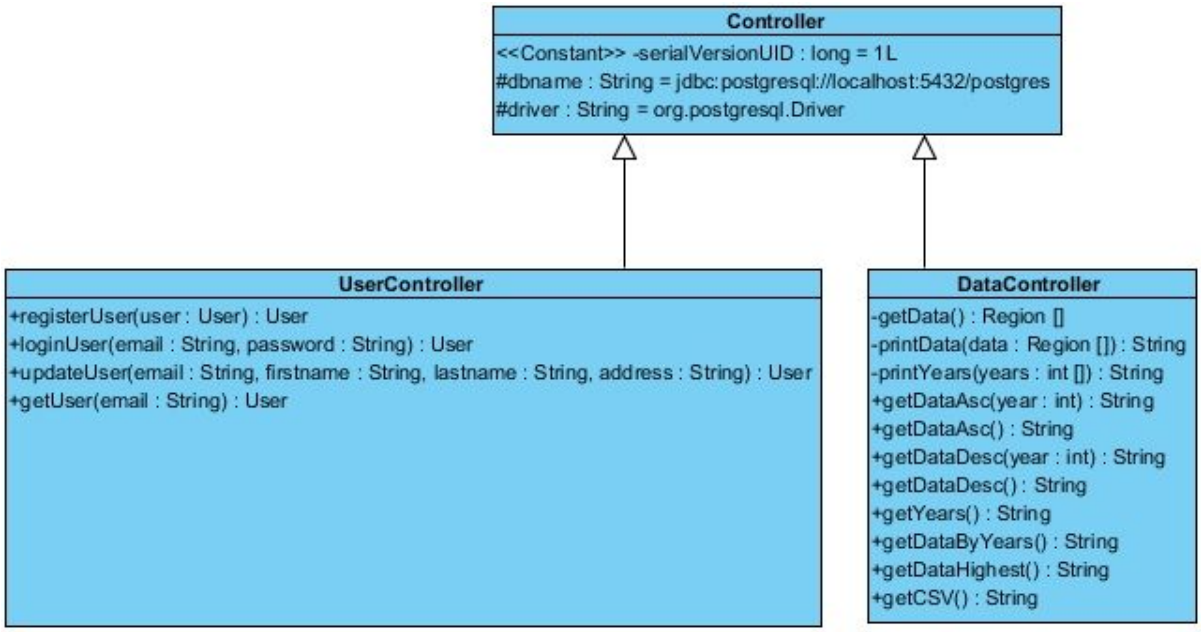

Figure 2.4: Controllers

Servlets

All of our servlets provide one post or get method and they are called after submitting one of the forms.

1. **UserLogin:** Is called when user submits the login form. User input (email, password) is handed to UserController and according to its response, servlet either logs user in (sets session's attribute "user" to User object) or displays an error message.
2. **UserRegistration:** Is called when user submits the registration form, hands over the input values to UserController and either logs user in, if the registration process succeeds, or displays an error message
3. **UserSettings:** Is called when user submits the settings form and again, gives all data to UserController that updates the database or returns an error message.
4. **UserLogout:** When user hits the logout button, this servlet just annulate session's attributes and redirects user back to homepage.

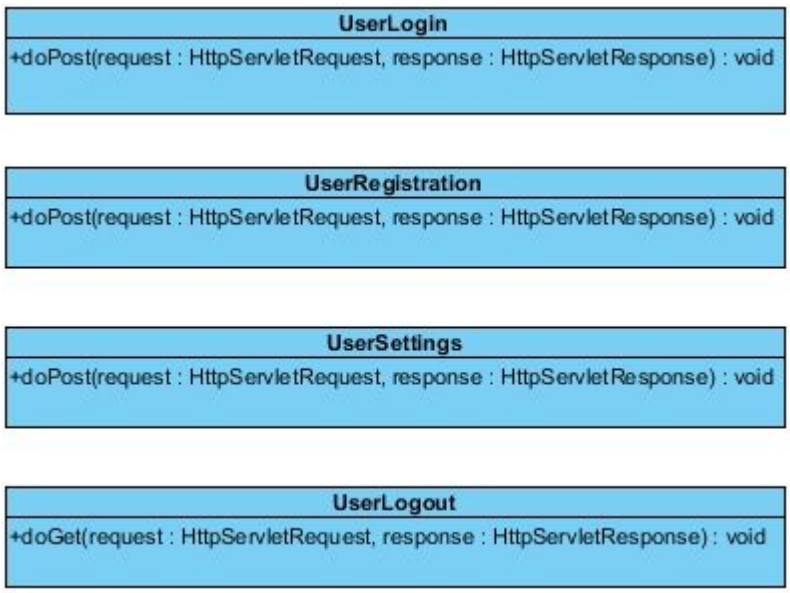

Figure 2.5: Servlets handling client's requests

System Security

One of many things that are still left to do on this project is to implement security measures. There are several threats or recommended steps that we would like to address here in the end of our project report.

1. **Manage SSL:** Since there is a registration and login form, we obviously transfer a lot of sensitive data like names, addresses and mostly passwords. In order to avoid eavesdropping of this information we will make all communication between client and server encrypted, therefore we replace insecure http protocol with https protocol. To achieve this, we will get a valid certificate signed by trusted authority (StartSSL). For testing purposes, it's also possible to create our own certificate and sign it by ourselves, although such certificate won't be recognised by browsers as safe.
   Once we have the certificate, we will configure the server to recognise it and force our application to use https protocol instead of the insecure one.
2. **Prevent code injection:** By code injection we mean SQL injection that can be used to bypass login validation or manipulate with database in general. To prevent such kind of attacks it is usually enough not to accept any SQL queries by plain url request and when it comes to creating SQL queries using parameters received from client, we will always use Java's PreparedStatement that make SQL injection practically impossible.
3. **Make cross-site scripting useless:** Another kind of attack takes in advantage the possibility of modifying and running code on the client side. Consequences of such attacks can be disabling client-side validation of form inputs or manipulating with price when it comes to shopping cart.
   To make it impossible, on top of client-side validation we will put also server-side validation of all important form inputs. If we ever charge any of our services, the final price of our products will by always calculated on the server-side independently on the client-side.
4. **Avoid exposure of sensitive data:** One of the most important thing is not to store any sensitive data, mostly passwords just in plain text. We have to store password of our users for login validation in database. To make them unreadable but still usable for validation is to use one of one-way hash functions (e.g. SHA-1). To make this system even more secure, we will use salted version of hash function which adds a specific string and the password together before the hashing itself.
   Even when the database will get exposed, with this technique, the data will be useless.
5. **Handle brute force attacks:** There are several ways how to prevent brute force attacks. At first, we will demand strong password from users and captcha validation just before login. After several unsuccessful login attempts there will also be a short time out before next possible login which will increase with every wrong password.