SWIMv2 Project

Design document

Nicolò Andronio

Andrea Brancaleoni

Table of contents

RASD modifications

Filtering search results by location is no longer an available feature.

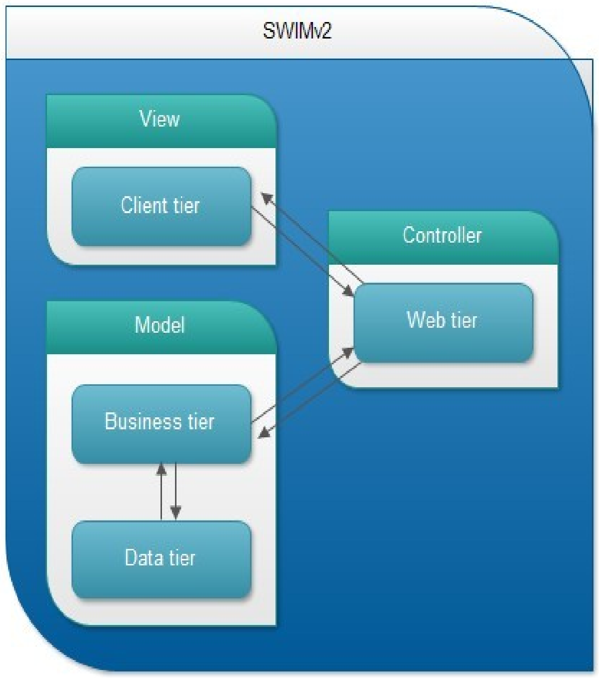
General design

1.1. System architecture

The project is to be realized as a 4-tier web application. The division in tiers roughly reflects this schema:

- Client tier: users will access the service site through the web within a managed interface, i.e. a web browser. They will receive view data under the form of html pages transferred using standard http protocols. In addition to markup and style code, the application will possibly send chunks of executable scripts included in these pages (JavaScript). Such scripts could aid in high-level front-end data validation. The client should therefore considered relatively “fat”.
- Web tier: handles http requests, routes them to appropriate controllers, keeps user sessions and checks for access rules compliance. This layer mediates the interaction between the caller (client) and the callee (business function) and is also concerned with error and exception handling. It is the first and most external component of the server architecture.
- Business tier: contains business logic, such as management functionalities and data manipulation routines. It constitutes the core of the application and is responsible for its behavior. Implementation of this tier is the most important task of this project and greatly influences the quality of service metrics exposed outside.
- Data tier: it is responsible for data persistence, integrity and management. This layer is mainly composed by software parts that are generally independent from our implementation (file system manager and dbms). However, the interface between third-party services and business logic is part of the actual implementation.

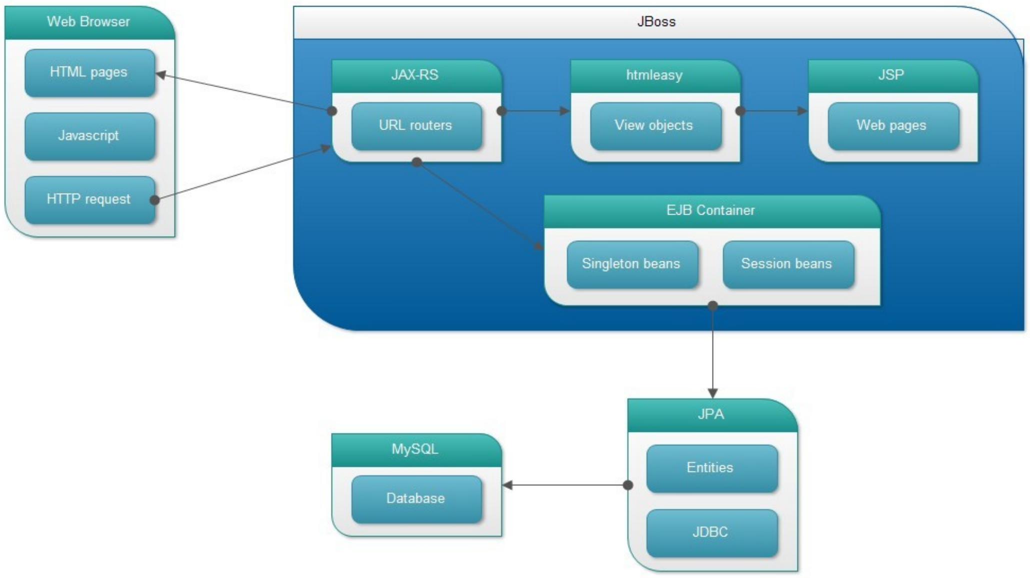
A design choice that directly descends from this separation, which comes pretty natural to enterprise web applications (especially if developed with high-level technology such as Java), is the usage of a common, widely-known pattern: MVC. In our case, the model comprehends both data and business tiers, while the web tiers makes up for the controller and client tier for the view. The only allowed interactions are view-controller and controller-model. The view can see part of the model but cannot directly interact with it. The model is totally oblivious of the rest of the application.



1.2. Technology specification

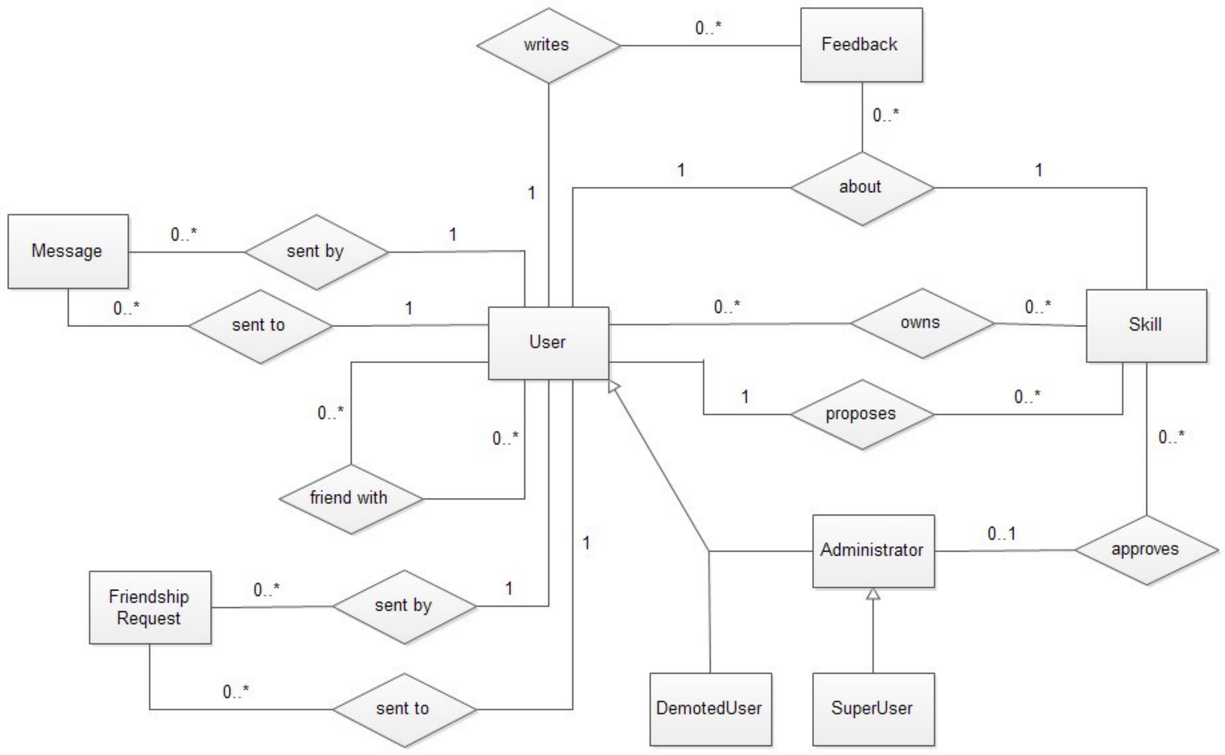
The identified tiers constitute our main design choice for this project. Now we need to spot out the subsystem required to build these tiers and iteratively refine their specification to create the final, detailed, implementation architecture. The system is designed as a RESTful service, accessed through a view framework.
Here are its high-level general systems, sorted from bottom to top with respect to the dependency hierarchy:

- Database: although the technology we intend to use greatly simplifies the usage of dbms through abstraction, the underlying service has to be present. In particular, the application will use a MySql installation.
- Entities: in order to elevate from dbms low level to language level, an entity structure is needed. In the first part of the model, entities are defined as normal POJO classes, properly annotated with JPA annotations. The application needs concepts to represent users, skills, messages, feed-backs, friendship requests and relations between them.
- Functionalities: each functionality outlined in the paragraph 1.3 of the Requirement Analysis and Specification Document must be implemented, following the conditions imposed by early use cases descriptions (section 2 of the same document). These functionalities should be divided into groups and exposed as methods by some interface. Specific implementations are delegated to EJB session beans.
- Controllers: these classes must map each http request to a proper interface method and handle exceptions in such methods, also defining the chosen template for front-end visualization. They are to be implemented via JAX-RS technology, using annotations to define paths for each method invocation.
- Web pages: define html templates using java server pages. They can be returned by JAX-RS methods through htmleasy integration.
- Web server: the infrastructure that holds everything on the server side is a java web server. Our choice is JBoss.



Data tier

2.1. Entity-Relationship diagram



The above diagram shows the entity structure used by the web application. For lack of space, attributes have been omitted but will be specified by later diagrams.

2.1.1. User

Represents a correctly registered and logged user of the system. Users can:

- Be friends with many other users (possibly zero);
- Send friendship requests;
- Receive friendship requests;
- Own skills;
- Propose skills;
- Write feedbacks;
- Receive feedbacks about own skills;
- Send messages;
- Receive messages.

2.1.2. Administrator

Administrators are users. They can additionally approve proposed skills.

2.1.3. Skill
A skill is a generic ability in doing something.

- Every skill can be owned by any number of users;
- Each skill has been proposed by exactly one user;
- A skill may or may not be approved, but in the latter case, the administrator who approves the skill is unique;
- Skills can be mentioned in feedbacks.

2.1.4. Feedback
Represents a rated comment about a specific user’s skill. A feedback is posted by exactly one user. Each feedback refers only to a (one) specific skill of a (one) specific user.

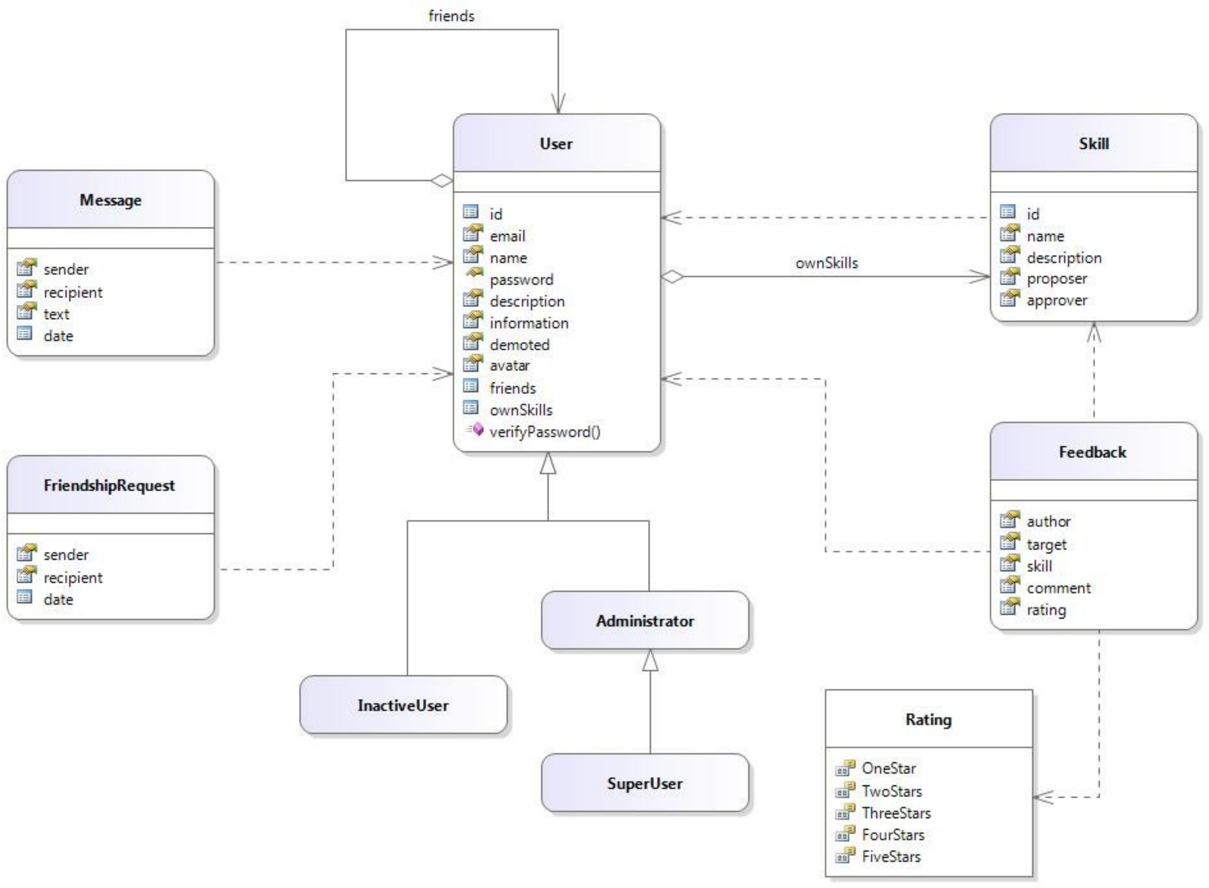
2.1.5. Message
It’s a simple chunk of text sent by an user to another one at a certain date and time. Messages cannot be sent to multiple users: each message has exactly one sender and one target.

2.1.6. FriendshipRequest
A request of friendship sent by an user to another user. Requests can target only exactly one user at a time.

2.1.7. SuperUser
It’s an administrator with superior privileges. However, no other specific relation is established by this entity.

2.1.8. DemotedUser
A former administrator that has been demoted by a SuperUser for bad behavior. Since the inheritance type is disjoint, no demoted user can be administrator again.

2.2. Entities class diagram



This diagram represents the outline of @Entity annotated classes in the model package. For brevity, instead of writing private fields, setters and getters, each attribute has been condensed into one field with an explaining icon. The hand symbol indicates that the field has a setter, while the window-like rectangle specifies that the field has a getter.
There are only two strong entities: User and Skill. These posses an auto generated integer primary key. Other entities (Message, Feedback and FriendshipRequest) are weak: their primary key is composed by the set of attributes that refer to strong entities.

- PK(Message) = {sender, recipient}
- PK(Feedback) = {author, target, skill}
- PK(FriendshipRequest) = {sender, recipient}

2.2.1. User

User's attributes are:

- email: string, validated with regular expressions;
- name: string, non-empty, max 255 characters long;
- password: this field can only be set via setter that accepts a string. To check if a password matches the user's password, use the verifyPassword method;
- description: string, max 255 characters long, can be empty;
- information: string, can be empty;
- demoted: boolean, can be changed by the system or by super users;
- avatar: byte array that stores image data, can be null. Avatar image format, size and weight can be restricted;
- friends: collection of Users, can be empty;
- ownSkills: collection of Skills, can be empty.

2.2.2. InactiveUser

This class does not exist in the ER diagram: it has no relation with any other entity. However, we will need this table to store data about users who began the registration process without yet completing it. Ideally, these entities either are persisted for a short time or they represent spammers. For this reason, they should ideally be cleared at specific intervals.

2.2.3. Skill

Skill's attributes are:

- name: string, non-empty, max 255 characters long;
- description: string, preferably non-empty;
- proposer: the user that proposed it in first place;
- approver: the administrator who approved it, may be null (if the skill hasn't been approved yet).

2.2.4. Feedback

Feedback's attributes are:

- author: user who posted the feedback;
- target: user whom the feedback is about;
- skill: skill which the feedback is about;
- comment: string, non-empty;
- rating: enumerated, between OneStar and FiveStars.

2.2.5. Message

Message's attributes are:

- sender: user who sent the message;
- recipient: user whom the message is addressed to;
- text: string, non-empty;
- date: date of the message, it's automatically set on creation.

2.2.6. FriendshipRequest
FriendshipRequest's attributes are:

- sender: user who sent the request;
- recipient: user whom the request is addressed to;
- date: date of request, it's automatically set on creation.

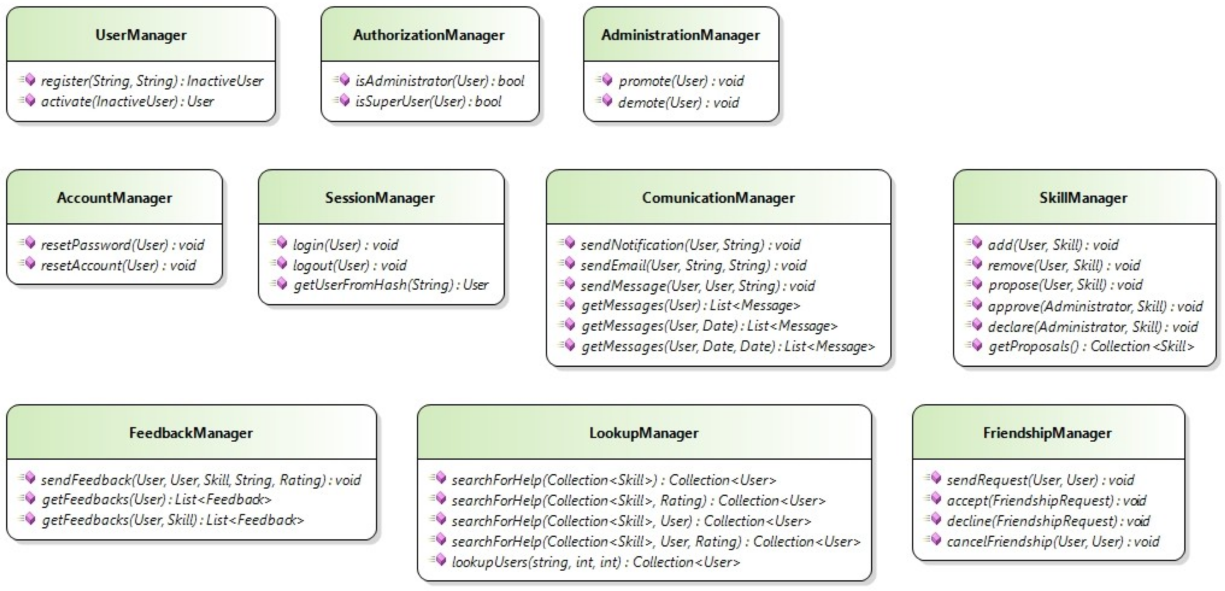
2.2.7. Administrator
Derived from User, it exposes no new attributes.

2.2.8. SuperUser
Derived from Administrator, it exposes no new attributes.

2.3. Logical view
While the usual database design process requires that ER is further refined through a logical view phase, in this case we don't need such step. In fact, the JPA abstraction layer allows us to stop the process at the entity level: logical and physical details will be dealt with by the underlying libraries. However, we can still take some decisions on the low level implementation. In particular, there is a special note for how inheritance is managed.
All the classes exposed above are annotated with @Entity. Only three of those are derived classes. JPA has several different methods for realizing generalization in physical database tables. The default policy is SingleTable, where the entity is mapped to a single database table and a special attribute is added to discriminate between base and derived classes. The actual used policy in our implementation will be this, since Administrator, InactiveUser and SuperUser do not expose any new attribute.

Business tier

3.1. Business class diagram


UserManager
register(String, String) : InactiveUser
activate(InactiveUser) : User
AuthorizationManager
isAdministrator(User) : bool
isSuperUser(User) : bool
AdministrationManager
promote(User) : void
demote(User) : void
AccountManager
resetPassword(User) : void
resetAccount(User) : void
SessionManager
login(User) : void
logout(User) : void
getUserFromHash(String) : User
ComunicationManager
sendNotification(User, String) : void
sendEmail(User, String, String) : void
sendMessage(User, User, String) : void
getMessages(User) : List<Message>
getMessages(User, Date) : List<Message>
getMessages(User, Date, Date) : List<Message>
SkillManager
add(User, Skill) : void
remove(User, Skill) : void
propose(User, Skill) : void
approve(Administrator, Skill) : void
declare(Administrator, Skill) : void
getProposals() : Collection<Skill>
FeedbackManager
sendFeedback(User, User, Skill, String, Rating) : void
getFeedbacks(User) : List<Feedback>
getFeedbacks(User, Skill) : List<Feedback>
LookupManager
searchForHelp(Collection<Skill>) : Collection<User>
searchForHelp(Collection<Skill>, Rating) : Collection<User>
searchForHelp(Collection<Skill>, User) : Collection<User>
searchForHelp(Collection<Skill>, User, Rating) : Collection<User>
lookupUsers(string, int, int) : Collection<User>
FriendshipManager
sendRequest(User, User) : void
accept(FriendshipRequest) : void
decline(FriendshipRequest) : void
cancelFriendship(User, User) : void

These interfaces constitute the general contract between business classes and controllers. Their actual implementation is delegated to appropriate EJB session beans. At runtime, these beans will live in the EJB container and thus exploit its features: in particular, through the entity persistence manager, they will be able to permanently affect the underlying database without knowing its structure or provider, thanks to the Entity abstraction.
Almost all the interfaces will be implemented by stateless session beans, with the exception of SessionManager, which will maintain an internal record of all logged users: this service will be exploited by controllers to know which user is requesting which method by reading request parameters, and eventually denying such access.
We feel that the methods outline is pretty straightforward, so we won't comment on what they are actually doing, since their names are meaningful. However, for readability, we'll list the methods of the diagram in the following paragraph, omitting the parameter types.

3.1.1. UserManager

- register(email, password)
- activate(inactiveUser)

3.1.2. AuthorizationManager

- isAdministrator(user)
- isSuperUser(user)

3.1.3. AdministratorManager

- promote(user)
- demote(user)

3.1.4. AccountManager

- resetPassword(user)
- resetAccount(user)

3.1.5. SessionManager

- login(user)
- logout(user)
- getUserFromHash(hash)

3.1.6. ComunicationManager

- sendNotification(user, notification)
- sendEmail(user, subject, text)
- sendMessage(sender, receiver, text)
- getMessages(sender)
- getMessages(sender, receiver)
- getMessages(sender, since)
- getMessages(sender, receiver, since)
- getMessages(sender, since, until)
- getMessages(sender, receiver, since, until)

3.1.7. SkillManager

- add(user, skill)
- remove(user, skill)
- propose(proposer, skill)
- approve(approver, skill)
- declare(approver, skill)
- getProposals()

3.1.8. FeedbackManager

- sendFeedback(sender, receiver, skill, rating, comment)
- getFeedbacks(target)
- getFeedbacks(target, skill)

3.1.9. LookupManager

- searchForHelp(skills)
- searchForHelp(skills, minRating)
- searchForHelp(skills, user) [searches only for user's friends]
- searchForHelp(skills, user, minRating)
- lookupUsers(partialName, startResultIndex, resultsCount)

3.1.10. FriendshipManager

- sendRequest(sender, receiver)
- accept(request)
- decline(request)
- cancelFriendship(user, friend)

Web tier

4.1. Routes

The controller layer of the application is constituted by JAX-RS resources, classes and methods mapped to URLs and http requests by means of annotations. These controllers receive requests and decide how to serve them. In particular, they:

- manage access rules, by discerning if an user has the rights of invoking a certain functionality, and if not by replying with appropriate error codes;
- handle exceptions thrown in by model business methods and provides a proper redirection to explaining error pages with proper response codes.

Here follows a list of routes, grouped by resource class:

4.1.1. LookupResource

Provides functionalities for looking up users and searching for help.

- GET /users ? {name}
  Gets a list of all users. If {name} is specified, includes in the list only those users that contain the parameter in their name.
- GET /help ? {skills}, {minRating}, {friends}
  Gets a list of users that can provide help for a given set of {skills} (pipe-separated ids). If {minRating} is specified, users are filtered out as to include only those results that has a minimum average feedback of minRating in at least on skill. If {friends} is specified, only friends of the requesting user are included.

4.1.2. SuperUserResource

Provides super-user specific administration features.

- PUT /users/{id}/password
  Changes the password of a given user.
- PUT /users/{id}
  Changes the profile of a given user.
- DELETE /users/{id}
  Deletes an user from the system.
- DELETE /admins/{id}
  Demotes an administrator.

4.1.3. AdministrationResource

Provides administrator specific tasks.

- PUT /admins/{id}
  Promotes an user to administrator.

4.1.4. ProfileResource

Provides functionalities for editing and retrieving profiles data.

- GET /self/
  Gets own profile.
- GET /users/{id}/
  Gets the profile of a given user.
- GET /users/{id}/avatar
  Downloads the avatar of a given user, as image.
- PUT /self/
  Edit own profile.
- PUT /self/avatar
  Upload own profile image.

4.1.5. SkillResource

Provides functionalities for editing skills.

- GET /self/skills
  Gets the list of own skills.
- GET /users/{id}/skills
  Gets the list of skills of a given user.
- GET /skills/proposals [admin]
  Gets a list of unapproved skills proposals.
- PUT /self/skills/{id}
  Adds a given skill to own profile.
- DELETE /skills/proposals/{sid} ? accepted [admin]
  Deletes a given skill proposal, accepting it or not depending on the passed parameter. If appected is not specified, it defaults to accepted=false.
- POST /skills/proposals ? {name}, {description}
  Proposes a new skill with given name and description.
- DELETE /skills/{id} [super]
  Deletes a skill from the database. Only available to super users.
- DELETE /self/skills/{id}
  Removes a skill from own profile.
- DELETE /users/{id}/skills/{sid} [super]
  Removes a skill from a given user’s profile. Only available to super users.

4.1.6. MessageResource

Provides message features access.

- GET /self/messages ? {sender}, {since}, {to}
  Gets a lists of all messages sent and received. If {sender} is specified, filters messages by the name of sender of received messages. If {since} and/or {to} are specified, only lists messages from date/time {since} to date/time {to}. Default value for {since} is unix epoch; default value for {to} is the timestamp at which the listing function is called.
- POST /users/{id}/messages
  Send a message to a given user.

4.1.7. FriendshipResource
Provides means for sending, accepting or declining friendship requests.

- GET /users/{id}/friends
  Gets a list of a given user's friends.
- GET /self/requests
  Gets a list of incoming friendship requests for the calling user.
- PUT /users/{id}/request
  Sends a friendship request to a given user.
- DELETE /self/requests/{id} ? accepted
  Deletes a given friendship request, accepting it or not depending on the {accepted} parameter. Default (if not specified) is false.
- DELETE /self/friends/{id}
  Removes an user from the list of own friends.

4.1.8. FeedbackResource
Provides functionalities for managing feedbacks.

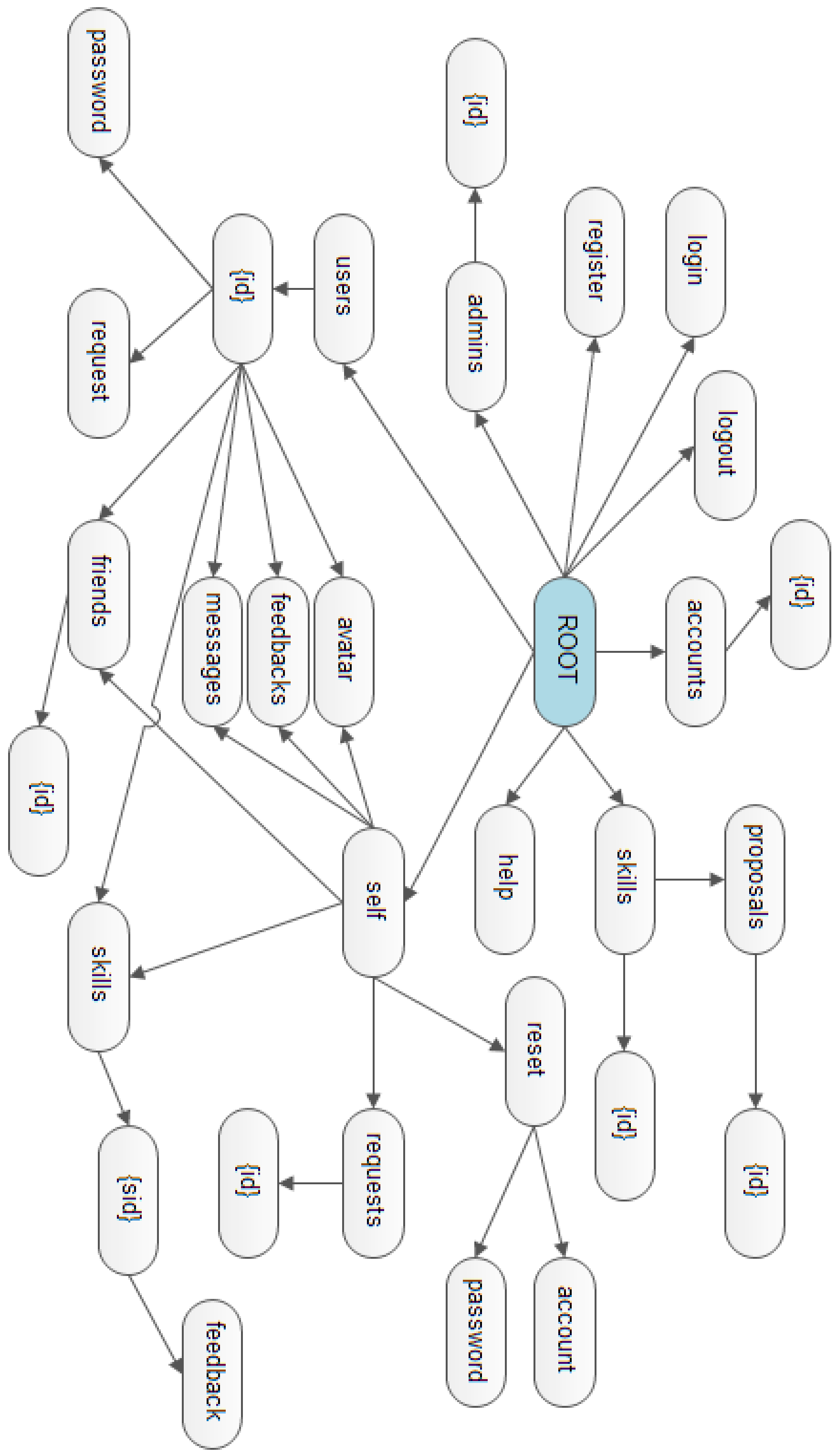
- GET /self/feedbacks ? {skill}
  Gets a list of all received feedbacks. If {skill} is specified, only includes feedback for the given skill.
- GET /users/{id}/feedbacks ? {skill}
  Gets a list of all feedbacks for a given user. As above, if {skill} is specified, only feedbacks about that skill are included.
- POST /users/{id}/skills/{sid}/feedback
  Sends a feedback about a skill to a given user.

4.1.9. SessionResource
Provides functionalities for managing an user session.

- GET /logout
  Logs out from the site.
- POST /login
  Logs in the site.
- POST /register
  Sends a registration request.
- PUT /accounts/{id} [super]
  Manually activates an account. Only available to super users.

4.1.10. AccountResource
Provides credentials retrieval services.

- POST /self/reset/password ? {email}
  Resets own password and sends it via email.
- POST /self/reset/account ? {name}
  Resets own password, sends it via email and displays the email associated to the given user name.


ROOT
accounts
{id}
logout
login
register
admins
{id}
users
{id}
password
request
friends
{id}
messages
feedbacks
avatar
skills
{sid}
feedback
self
requests
{id}
reset
password
account
help
skills
{id}
proposals
{id}

4.2. Session persistence

An user session starts at the login and ends at the logout, or when a timeout expires. Sessions are memorized in a stateful bean and kept alive through cookies. The cookie contains a unique hash generated at the login that identifies the user session. From that hash it is possible to get the associated user and thus determine if any request is valid or not.

Client tier

5.1. Controller interaction

JAX-RS controller can abstract return types by generally returning an object and then specifying a mime type to which the object will be serialized. Common formats are text/html, application/xml and application/json. However, we'd like to exploit java server pages for templating and brief server side scripting for customizing views and messaging. We can achieve this result with htmleasy, a simple html renderer micro-framework that works best with mvc. In this way, a router handler method can simply return a View object or be annotated with @ViewWith annotation, redirecting to the correct jsp page.

5.2. JSP pages

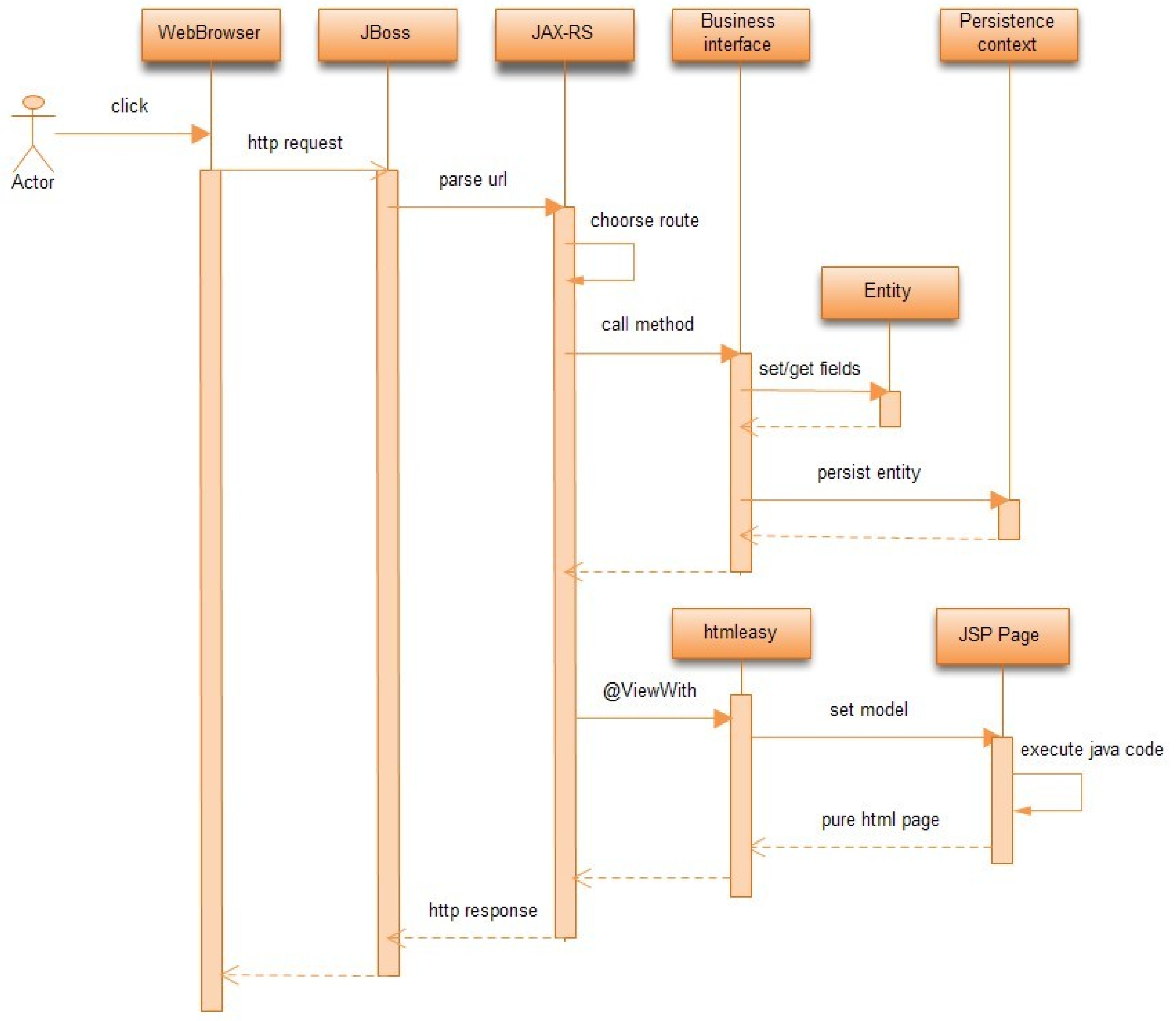
Here is a minimal list of pages to be implemented:

- Home page
- Error page
- Notification page
- View user profile (includes form for sending messages, feedbacks and friendship requests)
- Edit user profile
- View own messages
- View own feedbacks
- Search for help
- View search results
- Search within user base (only look up users)
- View users lookup results (includes form for promoting/demoting, sending friendship requests, activating accounts, deleting accounts, depending on who performs the search)

High level diagrams

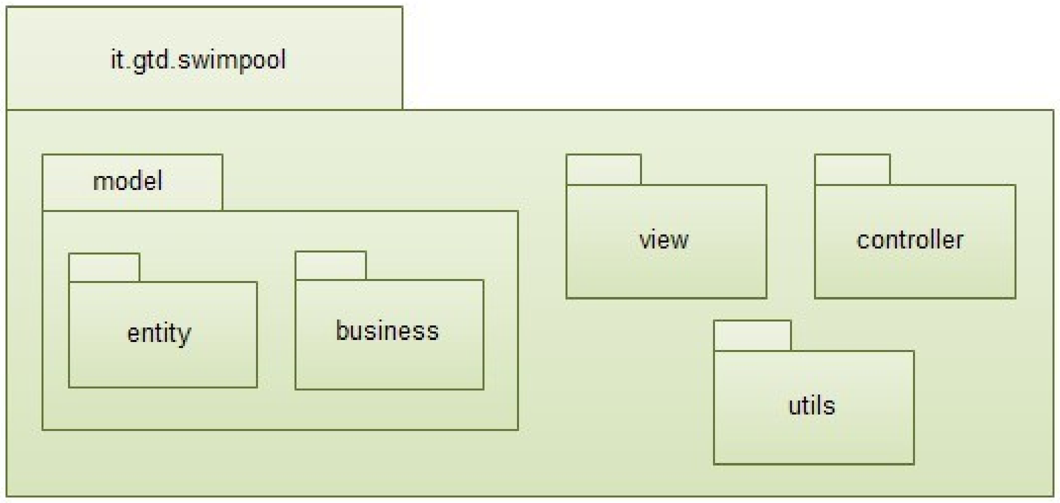
6.1. System sequence diagram

Since the system ideally behaves always in the same manner, it is useless to craft a sequence diagram for each particular case, because that would yield many almost identical diagrams. Therefore, we decided to attach only one diagram that exemplifies the lifecycle of a client request: how it is received, processed and responded to.



6.2. Package diagram

The package division is very simple. The main package holds the name of the project and group. Then, there is one package (that constitute a separate project) for each high level layer, plus one for utility classes. Model contains either entities and business logic.



Also the dependency structure is very simple:

- model.entity does not have dependencies (except, at most, utils)
- model.business depends on model.entity (and possibly on utils)
- controller depends on model
- view depends on controller