

Software Architecture Document

● 1. Use Cases

The project is a question/answer website. The most significant use cases are described in the following diagram:



There are three personas in the system:

The **Anonymous User** can view all the questions and answers on the site. He is also able to register himself.

The **Authenticated User** has more rights. In addition to be able to log in and to view all the questions and answers, he can also ask new questions, provide answers, ask for clarification with a comment, search for users or questions and upvote or downvote a question or an answer. In addition he can also edit his user profile and update it with valuable information so other users can get to know him and his fields of expertise better. In case he decides to leave the site he can anonymize his profile, which will leave all his questions, answers and comments but will replace him as an author with an "anonymous user". He can also completely delete his profile and everything he has ever posted, leaving no trace behind.

The **Moderator** can, in addition to everything an authenticated user can do, edit the profiles of all the users of our page. He can also decide to block a user, in case this user

was infringing our policies, where he basically returns to the status of an anonymous user, where he can't post anything.

●2. Logical View

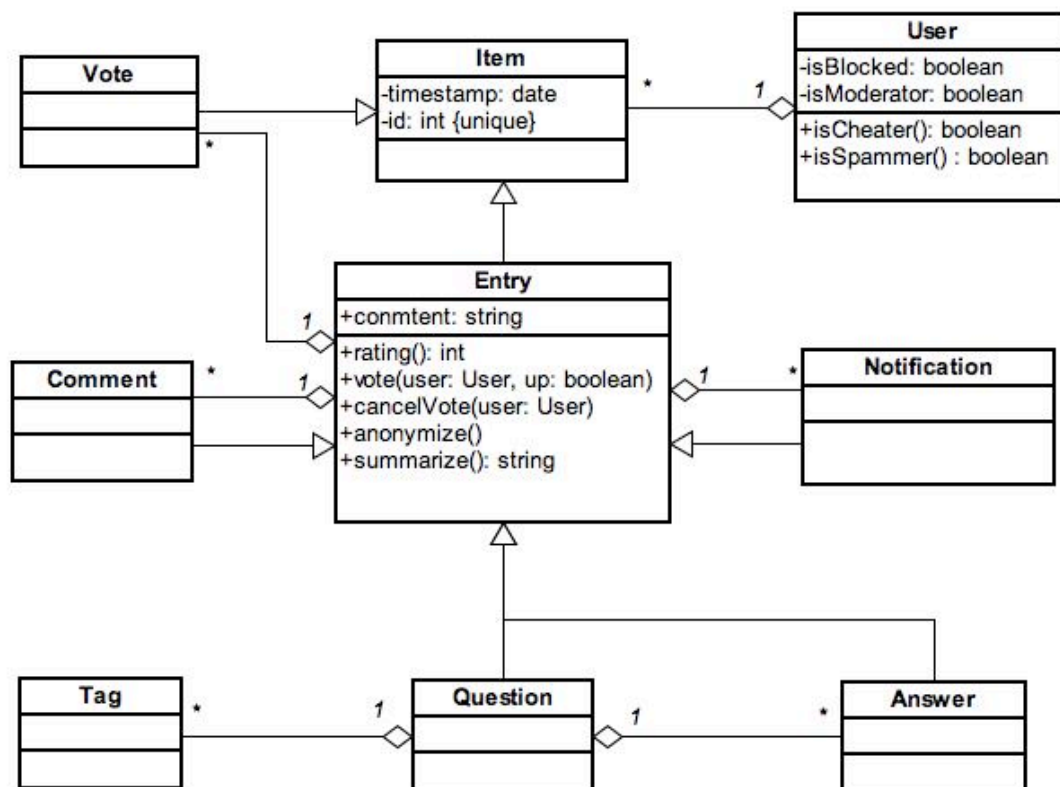
The project relies on the play framework, which enforces a traditional MVC pattern. The play framework promotes a stateless design and a rich domain model.

- Domain model classes are POJO (plain old java object).
- Controller classes must extend the Controller class of play framework
- Views are HTML template with scripts inside

The model is split into two main parts: The model with all data in the application, and the database, which is used to access the data globally.

●2.1 Domain Model

The UML class diagram for the main entities in the domain model:



Item

The Item class signifies that an object belongs to the content of the page. It's the common superclass that ensures a unique id number for all objects, provides access to all instances over their primary key. A creation timestamp is available, too. An item is owned by a user, and a user knows the list of his items – the relationship is bi-directional.

Entry

The Entry provides the common functionality of all contents of the page – they can be rated (by voting them up or down), are created by a user and can provide a short version of themselves in order to keep the interface clean. All Entries can in principle be voted up, but in practice, only questions and answers need this feature.

This is the base class for all entities in the application that the user can edit (such as Questions, Answer, Comment) and also Notifications which are generated automatically but are also textual.

Question

This is the central point of the application and is a subclass of Entry. In addition to it, Questions can be tagged by the owner and answered by other users. So a Question has several tags and several answers.

Answer

It is a subclass from Entry as well. Every answer belongs to one Question.

Comment

A comment is a subclass from Entry. It can't be voted but can be “liked” by other users instead. Every comment belongs to an Entry (in practice it can only belong to a Question or an Answer).

Vote

A vote is owned by a user and extends then Item and references another Entry.

Notification

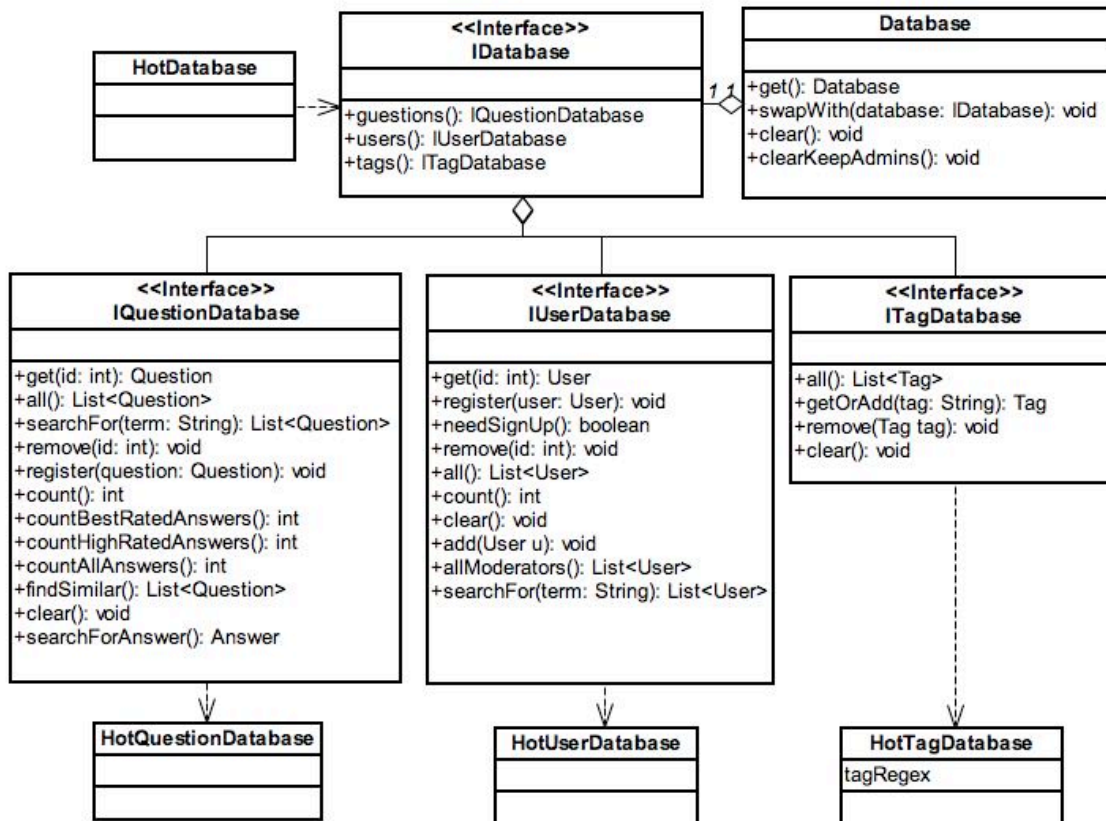
A notification is sent to a user to inform him that something happened to an entry they are watching. It describes what happened and to which entry.

User

A user is web visitor of the application, which registered in the system. The User knows all the items he wrote and can be a moderator or a normal user. A User can be blocked by a moderator in which case most of the actions cannot be performed any longer. A user object keeps track of when he last did a search or a posted something, as some action are only possible after a certain delay.

●2.2 Database

The Database



Database

The IDatabase interface abstracts the persistence strategy. Currently only one implementation exists, which is an in-memory database. The persistent objects of the domain model are then further organized into three database segments depending whether they related primary to question, user or tags. These segments can be access via the Database class which acts as a global singleton that can be used locate and search persistent objects.

●2.3 Controllers

Controllers in the play framework are stateless and consist of static methods. Their role is to handle web request sent from the client and forward to the corresponding view after having updated the model accordingly (MCV pattern). There is very little interaction between controllers, so no UML is provided.

The controllers handle also data validation from the client, and provide the view with the appropriate feedback message if necessary. Validation is done using the Validation class provided in the play framework, while feedback are passed to the view through the flash variable provided in the parent class, e.g. “flash.succes(...)” or “flash.error(...)”.

Application

This controller handles all nonspecific, non-authenticated access to the site. Or in short: what isn't important enough to get an own controller.

CAnswer/CQuestion/CUser

Most of the logic resides in these three controllers, which interact with the corresponding domain objects. Most actions require authentication.

Search

Handles different search requests by passing them to the DB.

●2.4 Background job

The play framework supports background jobs. One job (Bootstrap) runs when the system is started and load the base data in memory. Another one (CleanUpJobs) runs periodically and performs some data-intensive action that cannot be done in real-time.

●2.5 Testing

Session

With play framework the design is essentially stateless. The only information in the session is the logged in user. For ease of mocking, access to the logged in user is abstracted with `ISession` so that the session can be mocked in unit test.

Session and SessionMock

One is the implementation that forwards to the play framework feature and returned the logged in user, the other one is a mock that can be used in unit test.

SysInfo

Similarly to the session, the access to system information such as the current time is abstracted. The `SysInfo` class is globally accessible and returns the implementation to use. This can be seen as a kind of service locator.

SystemInformation and SystemInformationMock

These are the two implementations that can be used to access the system information. One returns the real data from the environment; the other one is a mock that can be used for unit testing.

● 2.6 Security

The application leverages the built-in security of the play framework using the `Secure` class and the `@With` annotation. The hook in the framework that performs the authentication is the `Security` class which implements the method `authenticate`.

●3. *Development View*

The Application is based on the play framework, a framework that enables the rapid development of java web applications. The framework imposes the file structure, and promotes convention over configuration.

Application code

The code of the application is organized in the following structure:

- `./app/`(includes everything which is used for the running app)
- `./app/controllers` (the interface between view and model, allows to use the model, it translates user actions into changes in the model)
- `./app/models` (contains the modeling of the data structure)
- `./app/models/helpers` (contains classes with functionality that don't match the responsibility of other classes)
- `./app/models/database`(An interface for data management and maybe in the future persistency)
- `./app/models/database/importers` (importers for external data to the database)
- `./app/view` (contains the templates for the view)
- `./test/tests/`(contains the different tests for the model)
- `./test/mocks/`(contains the mock classes used by the tests)

Configuration files

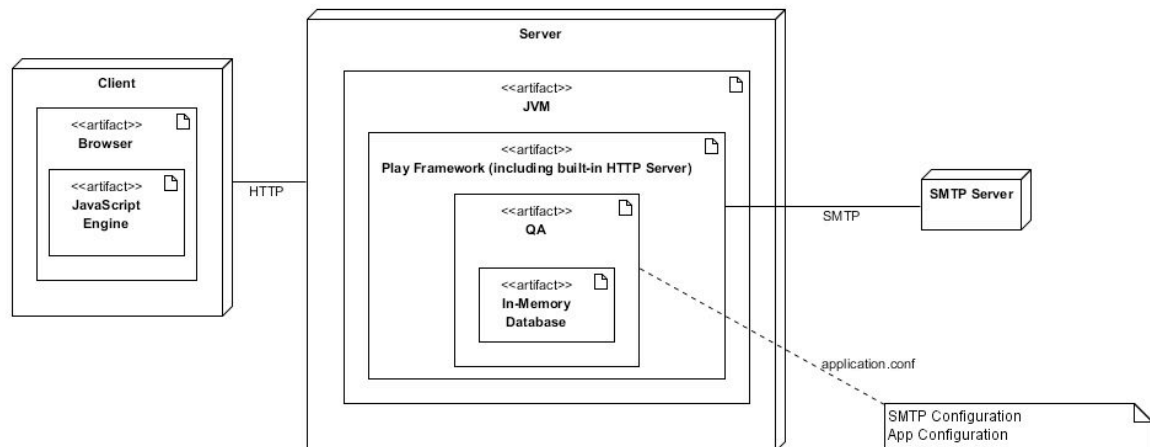
The regular configuration files of play! framework are used:

- `./conf/application.conf` (is key-value configuration files with play-specific settings or application-specific settings)
- `./conf/routes` (maps the URL in the browser to the corresponding action in the controllers, and describe rewrite rules to have readable URLs)
- `./conf/initial-data.yml` (contains initial data, loaded on application startup mainly for testing.)

External library and modules

The play framework has an extensible module system. Only the Secure module is used for this application, which is normally located under `${play.path}/modules/secure`.

●4. Deployment View



Client Browser: The client needs a browser and a working internet connection to access the remote web application over HTTP. In order to use all of the Javascript functions the browser must have a (not too old) Javascript engine and Javascript must be enabled in the browser settings.

Web Server: The client connects to the server via HTTP. On the server there needs to be a running Java Virtual Machine, as the Play! Framework is written entirely in Java.

Then of course the Play! Framework itself must be installed. The Play! Framework provides a built-in HTTP Server that can be used. Otherwise it is also possible to use a standalone web server like apache, which must be installed and configured properly. This would also require some adaptations in the configuration of the QA Application itself.

QA is the name of the app that is run within the Play! Framework. This can either be done directly (you have a directory with all the files on the server and you use the *play run QA* command to start the application and the web server) or by deploying the application as a .war file (web archive). The result will be the same thing. The QA app stores a lot of relevant configuration data in a file called application.conf. There are different configurations for e.g. an SMTP Server or the Application itself that need to be configured correctly.

The application uses an In-Memory Database to store all the data that the application needs, including but not limited to all the questions, answers, comments, user profiles, votes, etc. Once the server is shut down, all the data is lost.

Mail Server: This server is needed to send out confirmation e-mails to newly registered users. The Play! Framework can connect to a mail server through SMTP. The connection settings must be specified in application.conf.

The Mail Server machine obviously needs to have a running and properly configured SMTP Server software (e.g. sendmail, postfix, etc.).