Design Document

|  |  |
| --- | --- |
| Application | Support Ticket Project |
| Author | João Marcelo Oliveira de Souza – joaomarcelo9393@gmail.com |
| Goal | Crossover Recruitment Applicant |
| Date | January 2017 |

Goal

Create a design document containing the following

1. High level requirement analysis
2. High level presentation of the data model
3. Architecture diagrams describing the composition and working of the system, explaining the component interaction and process, control and data flows.
4. Explain the breakdown of the system into components with technical implementation details of each component along with the design patterns involved and with reasons that justify your choices.
5. Use both visual elements (diagrams) and text descriptions to maximize the amount of information conveyed while keeping the document as compact as possible

Requirement Analysis

This section presents functional and non-functional requirements given by the client as-is and further analysis.

# Functional Requirements

The system allows customers to be able to place support request and the support agents to process the requests. The system implements the following specifications.

1. For customers
   1. A web portal to create support requests and to view status of previous requests.
2. For support agents
   1. A web portal to find and process support requests.
   2. One report, on requests data, with all tickets closed in the last one month; Should be PDF exportable.

Other functional requirements

1. Users should be able to authenticate normally
2. An admin user should be able to manage other users and any other system objects.

Assume any functional details required to achieve the above requirements based on logic and your experience.

# Other Technical and Non-functional Requirements

The following list of technical specifications must be adhered to

1. Use MySQL server, Ruby on Rails on back end, Bootstrap and a JS framework on front end. Use latest versions if possible.
2. On back-end : REST API, db migrations, ActiveRecord, rspec tests
3. On front-end : SPA pattern web-app, with responsive design, using a JS framework from either Angular/Ember/React/Knockout/Backbone, testing with either Qunit/Jasmine/Karma.
4. Enough coverage of proofing automated tests on both back and front end (coverage > 30%).
5. Secure the applications and services.

# Requirement Analysis

The Functional Requirements given need more detail. Some assumptions will be made to cover the necessary details, as the Functional Requirements section states at the end: “*Assume any functional details required to achieve the above requirements based on logic and your experience*”. The necessary assumptions are shown next as additional requirements.

### Additional Functional Requirements

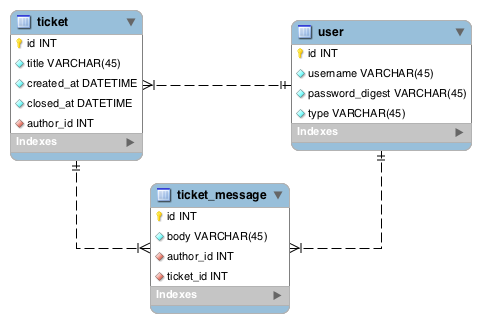
1. There are three roles of users: customers, support agents and admin.
2. All users are authenticated through username and password.
3. For customers and support agents
   1. Can provide additional messages to a support request while it is OPEN.
   2. Can NOT edit messages once submitted – to avoid misunderstandings in the support procedure.
4. For support agents
   1. Can close and re-open support requests.
5. For admins
   1. Can do everything that a support agent can.
   2. Can delete support requests – must confirm his intention.
6. REST API
   1. Performs authentication control to limit access to legitimate users
   2. Performs authorization control to limit the permissions of each user role
7. Web-app
   1. For all users:
      1. Authentication page
      2. List of support requests, ordered by most recent activity first
      3. Support request detail page
         1. List of messages with authors.
         2. When request is open, the page contains a form to provide additional information.
   2. For customers:
      1. Form to create new support request.
   3. For support agents:
      1. Controls to close and re-open support requests.
   4. For admins:
      1. Everything a support agent has access to.
      2. Controls to delete support requests with confirmation prompt.

Data Model

For convenience, some naming has been adapted in the data model:

|  |  |
| --- | --- |
| Original name | Model name |
| Customer, support agent, admin | User, with type attribute |
| Support request | Ticket |
| Support request message | Ticket message |

The data model is represented in the following diagram. Some attributes and names may change slightly in the actual implementation to conform to application details.



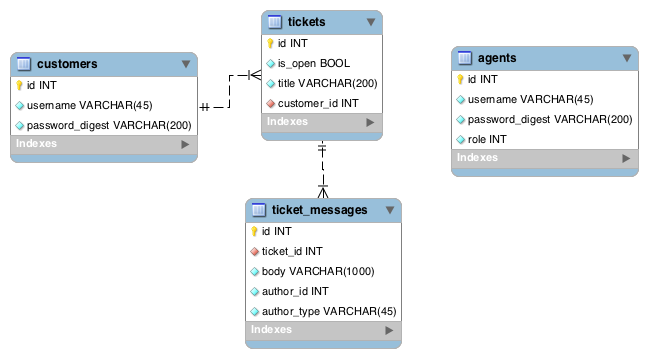
Final Data Model

# Optimizations

1. Speed: all primary keys are indexed.
2. Storage limit: all primary keys are unsigned.
3. Integrity: all tables have primary keys and all references are foreign keys.
4. Consistency: columns are not-*null*, except where *null* is needed.
5. Flexibility: other user roles can be created anytime by setting a new type attribute.
6. Single Table Inheritance: user roles can be represented by different application classes.
7. Simplicity: performing authorization and processing on user records is straight-forward
8. Convenience: many Ruby on Rails libraries rely on a *users* table by default to perform tasks automatically.

# Special Considerations

One possible drawback of this model is having customers in the same table as agents and admins. In the other hand, in order to split these users into separate entities, the *ticket\_message* entity would need a polymorphic relation to the author and thus be susceptible to loss of referential integrity, as shown in the following diagram. The other benefits pointed previously are also lost.



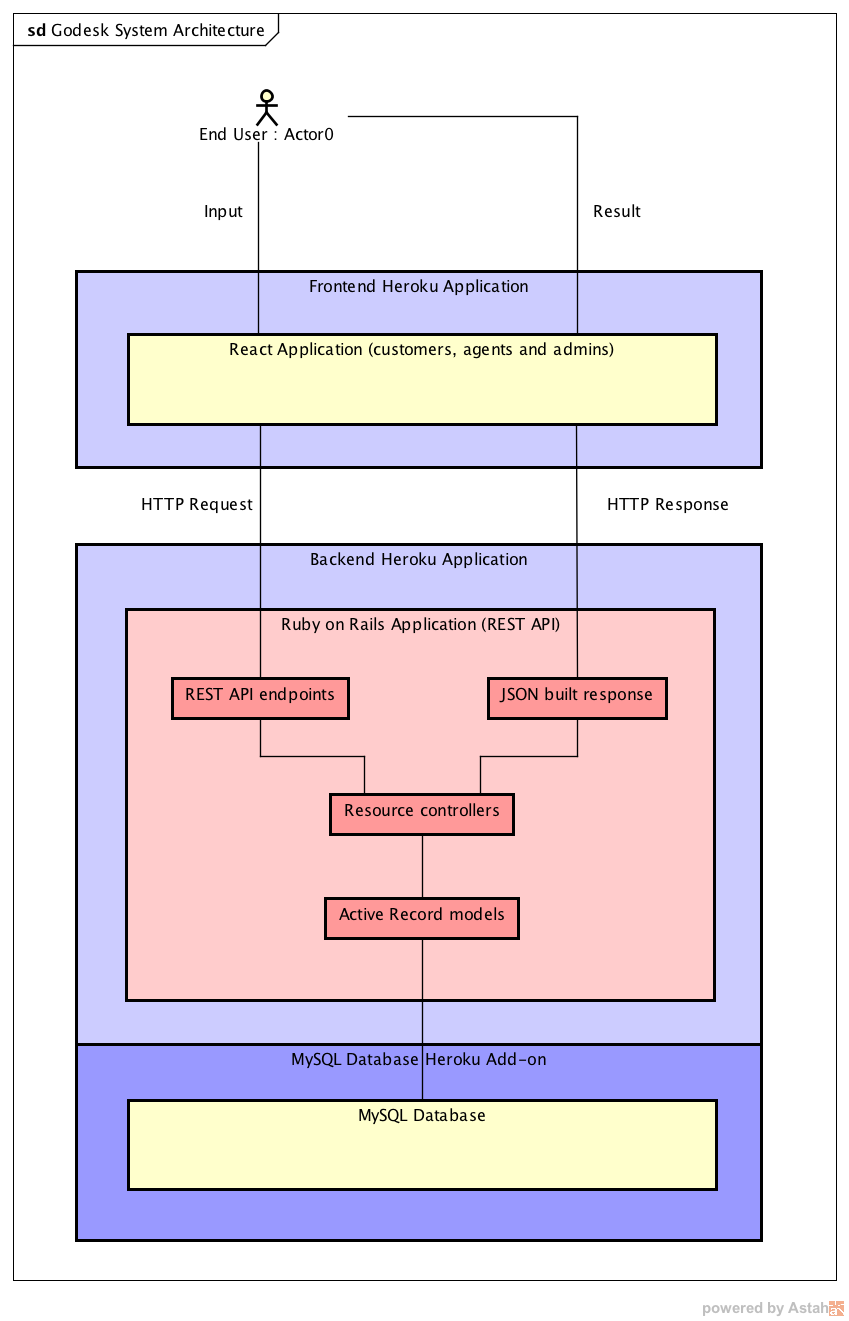
Data Model with polymorphic relation in ticket\_messages

In the chosen model, by having all users in a single table, the main issue to address is securing that a user cannot change its own role by changing the *type* value. That can be easily achieved through authorization rules on the API – only an admin user can manage user roles.

Performance is not an issue on the *users* table because querying on indexed primary keys is fast even on millions of records. Record limit is also not an issue because the MySQL auto-incremental primary key can hold up to billions of values with an unsigned INT type and more if necessary with a BIGINT type.

Architecture

The system is composed of two applications (frontend and backend) and one database. It is deployed to Heroku as demonstrated in the following diagram.



It is possible to deploy this architecture on the Heroku free tier. Each application can be scaled independently.

The React Application is the user interface to the system. It displays support tickets and other data in the browser, handles input, requests new data when necessary and submits user data.

The Ruby on Rails Application bridges the user application to the database using MVC pattern:

1. Receive user requests through RESTful endpoints.
2. Authorize and process requests at the **controller** layer.
3. Performs validations and persistence at the Active Record **model** layer.
4. Render JSON responses at the **view** layer.

Using MySQL Database as a Heroku Add-on enables quick setup of multiple isolated staging environments without effort through Heroku Pipelines, but the application is also flexible enough to use any other service if necessary as long as the database has a public endpoint and a known password.