

# Board Game Borrow – Developers Guide

## Table of Contents

<b>INTRODUCTION.....</b>	<b>1</b>
<b>DATABASE .....</b>	<b>1</b>
ENGINE.....	1
DESIGN AND ARCHITECTURE .....	2
<b>TESTING.....</b>	<b>8</b>
<b>STYLING.....</b>	<b>5</b>
CALENDAR .....	5

## Introduction

The aim of this guide is to present the architectural and technical aspects of the application in order to provide for an easier development process for onboarding developers. It also serves as a guide for some practices and ideas that may be reused in this or other applications.

## Database

### Engine

In the initial design document the choice of the database engine fell on PostgreSQL, since it is one of the most powerful database engines, providing for high data integrity and allowing for complex designs. Upon the actual draw-up of the application architecture it became apparent that the database structure in such web app is going to be rather simple, and therefore PostgreSQL would be an over-kill in such situation.

The choice of the database engine then fell on SQLite. Since Rails comes with a built-in support for SQLite, and since it is the default engine when creating a new Rails project, it was a very easy tool to use to jump straight into the development.

Being a web application, BoardGameBorrow is supposed to be very light and efficient. SQLite turned out to provide just the functionality needed – being a self-contained, file-based database it offered to handle data with less constraint and more ease compared to hosted relational databases. That was especially visible in terms of application and database integration – the fact that SQL calls are made to a file directly on the same machine instead of

communicating through an interface (network ports and sockets) lets SQLite to be fast and efficient.

Depending on the degree of usage of the application it might be feasible to switch to a more sophisticated database engine in the future. That may occur in case SQLite becomes overloaded upon going to production. That may be possible due to the fact that write operations are limited when it comes to SQLite, allowing only a single write operation to occur at a time, hence providing a limited throughput. However, that change may have to be backed up by results of load and stress tests on the system, since few hosting providers offer PostgreSQL managed instances.

In case testing proves SQLite is not handling the expected load the application would be withstanding, it is easy to configure BoardGameBorrow to use PostgreSQL engine instead. In case a running PostgreSQL server is present on the machine, it is enough to modify **config/database.yml** file to point to that instance as follows in case of development environment:

```
development:

  adapter: postgresql

  encoding: unicode

  database: bgb_development

  pool: 5

  username: bgb


  password:
```


Make sure to change username and password appropriately within the configuration.


## Design and architecture

The database was designed to operate on three main tables – rentals, games and users. Rentals table has foreign keys from both users (users.id => rentals.user\_id) and games (games.id => rentals.game\_id) tables. The relationship between the tables is such that

- a user may have many rentals, but a rental may have only one user assigned to it (one-to-many)
- a game may have many rentals, but a rental may only concern one game (one-to-many)

users	
 <b>id</b>	INTEGER
<b>email</b>	VARCHAR
<b>encrypted_password</b>	VARCHAR
<b>is_admin</b>	INTEGER
reset_password_token	VARCHAR
reset_password_sent_at	DATETIME
remember_created_at	DATETIME
name	VARCHAR
<b>sign_in_count</b>	INTEGER
current_sign_in_at	DATETIME
last_sign_in_at	DATETIME
current_sign_in_ip	VARCHAR
last_sign_in_ip	VARCHAR
<b>created_at</b>	DATETIME
<b>updated_at</b>	DATETIME

games	
 <b>id</b>	INTEGER
title	VARCHAR
min_players	INTEGER
max_players	INTEGER
min_player_age	INTEGER
playing_time	INTEGER
complexity	FLOAT
location	VARCHAR
link	VARCHAR
<b>created_at</b>	DATETIME
<b>updated_at</b>	DATETIME
rating	FLOAT
expansion_for	INTEGER
is_deleted	INTEGER

rentals	
 <b>id</b>	INTEGER
name	VARCHAR
start_time	DATE
end_time	DATE
game_id	INTEGER
<b>created_at</b>	DATETIME
<b>updated_at</b>	DATETIME
user_id	INTEGER
is_optional	INTEGER
is_vetoed	INTEGER

Additionally, there are a few important fields in each of the tables representing useful from application point of view attributes of each object.

The distinguishing field in the users table is the field called **is\_admin**. It defaults to 0 whenever a user signs up for the service, and this attribute should be changed directly within the database in case the user should be given administrator permissions by executing the following query:

```
update users set is_admin=1 where id={user_id};
```

Users with administrator privileges are allowed to add games to the database and to erase them from there using the GUI interface within the web app.

When a game is deleted it is not erased from the database completely. Instead it is flagged with **is\_deleted** field being set to 1. That occurs in order for the users who have already borrowed the game to have the history of their reservations preserved. Games with **is\_deleted** set to 1 are still visible in the reservations list, but cannot be borrowed.

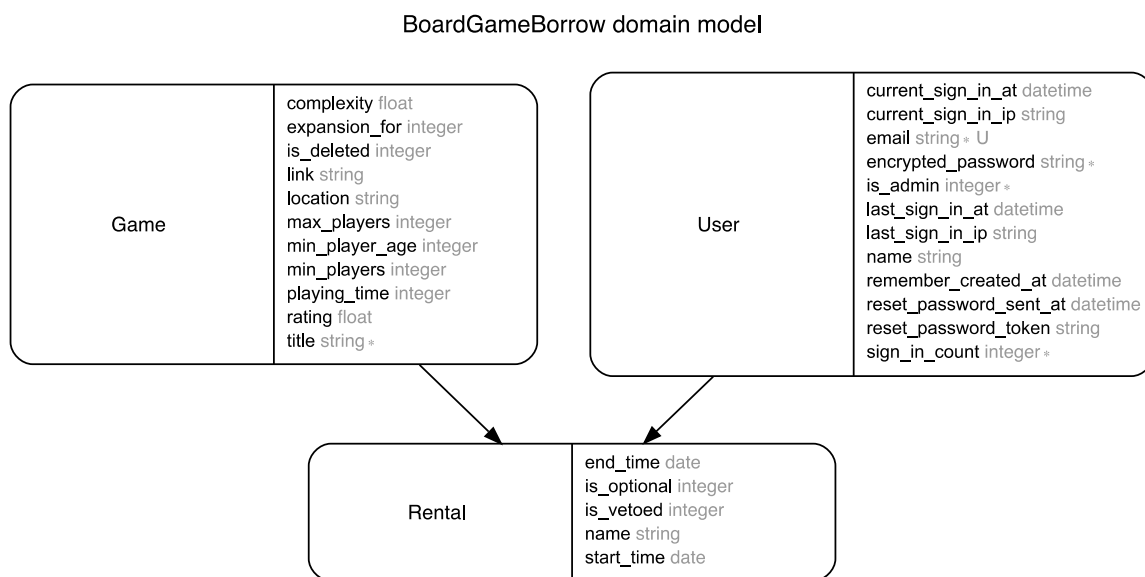
Another valuable field in the games table is the field called **expansion\_for**. That field is the link between the current game and the game it is considered an expansion for. It is populated with the id of the game we want to link the current game to. This field should be utilized in future development when functionality regarding expansions is going to be added.

In terms of rentals table, there are two fields connected to reservations – **is\_optional** and **is\_vetoed**. **is\_optional** field is assigned to 1 in case a reservation is optional. The following rentals should be treated as optional:

- each subsequent reservation on given day
- reservation of the game on the day immediately after the day it was borrowed

Optional reservations may be vetoed by other users in case those users do not have a guaranteed reservation for the same day. In case that happens, **is\_vetoed** field within the reservation row is populated with the id of the user who has vetoed the reservation – based on that id the user that originally rented the game is informed whom that reservation was vetoed by.

The following diagram portrays the relationships between entities within BGB domain model:



## Styling

### Calendar

The calendar is a vital part of the application's user interface, so it was crucial for it to function neatly and be styled in a pleasant way.

The `simple_calendar` gem was used as a basis of the calendar for BGB application. You can familiarize yourself with the source code of the gem in this Git repo:

[https://github.com/excid3/simple\\_calendar](https://github.com/excid3/simple_calendar) .

The month calendar presenting events was chosen as the basis, the events being rentals for the particular game:

```
<%= month_calendar events: @rentals do |date, rentals| %>
  ...
<% end %>
```

The rentals (events) presented within the calendar had to be filtered, since there exist many games in the database, as well as many rentals for each of them, whereas the same base calendar instance is rendered for each of them. The rentals within the game calendar are filtered via a map that is created within the loop from above, as follows:

```
...
<% game_rentals = {}
  rentals.each do |rental|
    game_rentals[rental.game_id] = rental if rental.is_vetoed.nil?
  end %>
<% game_rental = game_rentals[@game.id] %>
...
```

For each date a rental is only displayed in case it belongs to the particular game, which will be visible in the logic described further along.

The default styling of the calendar provided with `simple_calendar.scss` located in the gem's git repo, however, has proven to be not user-friendly enough for our purposes. You may observe the default regular css styling in the image below:

## «March 2013»

Sun	Mon	Tue	Wed	Thu	Fri	Sat
24	25	26	27	28	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24 Test Second	25	26	27	28	29	30
31	1	2	3	4	5	6

**Name**      **Start time**  
 Test Event 2012-12-07 20:08:00 UTC  
 Test      2013-03-24 13:11:00 UTC  
 Second      2013-03-24 12:12:00 UTC

New Event

You may notice that the only way to add events to that calendar is by clicking the button New Event on the right hand side of the image, which is not the best possible way of handling that particular functionality from a user experience point of view.

Naturally, the simplest answer that has been proven to be efficient and stable for styling purposes is bootstrap. The calendar from BoardGameBorrow application uses bootstrap theme called Journal provided by Bootswatch ( <https://bootswatch.com/journal/> ). It is utilized within the app by using **stylesheets/bootstrap.css**, which has been copied from Bootswatch's page.

For the purposes of BGB it was crucial to provide color-coded information regarding availability of the game on specific dates, existing optional and guaranteed reservations, as well as information about the people who have placed the reservations.

Journal theme provided a set of classes that worked perfectly for this purpose. The prettiest way of presenting all of the necessary information within the calendar was to use buttons with specific classes, as well as links for user information. Thus, styling is being chosen on the basis of the conditions that a given rental fulfills – in case there is no rental for that game for a given day, the button displayed is pink and clickable (*class="btn btn-primary"*), in case an optional reservation exists for that game for that day the button displayed is orange and clickable (*class="btn btn-danger"*), in case a guaranteed reservation exists for that day, the button is grey and non-clickable (*class="btn btn-default disabled"*):

```

...
<% if !game_rental.nil? %>
  <% if game_rental.is_optional? %>
    <% cl = "danger" %>
  <% else %>
    <% cl = "default disabled" %>
  <% end %>
<% else %>
  <% cl = "primary" %>
<% end %>

<div><%= link_to date, new_rental_path(date: date, game_id: @game.id), :class =>
"btn btn-#{cl}" %></div>
...

```

If there exists a rental for that game for a given day (no matter if it is guaranteed or optional), the user the rental is linked to is displayed underneath the button:

```

...
<% if !game_rental.nil? %>
  <div><%= link_to game_rental.user.email, game_rental, :class => "" %></div>
<% end %>
...

```

The final product looks similar to the image below:

Previous February 2017 Next

Mon	Tue	Wed	Thu	Fri	Sat	Sun
2017-01-30	2017-01-31	2017-02-01	2017-02-02	2017-02-03 y@romanets.ry	2017-02-04 example@domain.com	2017-02-05
2017-02-06	2017-02-07	2017-02-08	2017-02-09	2017-02-10	2017-02-11	2017-02-12
2017-02-13	2017-02-14	2017-02-15	2017-02-16	2017-02-17	2017-02-18	2017-02-19
2017-02-20	2017-02-21	2017-02-22	2017-02-23	2017-02-24 example@domain.com	2017-02-25 y@romanets.ry	2017-02-26 y@romanets.ry
2017-02-27 y@romanets.ry	2017-02-28	2017-03-01	2017-03-02	2017-03-03	2017-03-04	2017-03-05

Note, that simple\_calendar gem creates the calendar with the use of simple css table, td and th assignments, therefore any bootstrap theme can be utilized in place of Journal by substituting bootstrap.css within stylesheets directory.

The solution outlined above may be reused by anyone who is working with simple\_calendar gem and is looking to customize its look and feel to enhance UX. The logic of filtering out rentals may be reused in any application connected to rentals and reservations.

## Testing

Testing is an integral part of the development process. The more modules appear in the application as it grows, the more tests are going to have to be written. Since some of the application logic depends on the attributes assigned to the user that is currently logged in, a helper class was created to facilitate Devise sign in and sign out functionalities that may be required in order to perform certain operations.

The helper module is located under **test/test\_helper.rb** and is called **SignInHelper**. The following code can be reused in any other application that uses Devise gem for authentication:

```
module SignInHelper
  include Warden::Test::Helpers

  def sign_in(resource_or_scope, resource = nil)
    resource ||= resource_or_scope
    scope = Devise::Mapping.find_scope!(resource_or_scope)
    login_as(resource, scope: scope)
  end

  def sign_out(resource_or_scope)
    scope = Devise::Mapping.find_scope!(resource_or_scope)
    logout(scope)
  end
end
```

Warden Test Helpers provide us the to login or log out on any given request. Devise searches for the devise scope to be used. It is very important to provide Warden's logout() method with the scope argument, so that it only logs out users with given scope. Otherwise all users will be logged out.

In our case, we have included SignInHelper into ActionDispatch::IntegrationTest class, since that is the class our tests extend.

```
class ActionDispatch::IntegrationTest
  include SignInHelper
```



```
end
```

The test files themselves need to include the following line:

```
require 'test_helper'
```

as well as perform `sign_in` action on the user within the `setup` block, so that the requirements for application logic are fulfilled:

```
setup do
  ...
  @user = users(:three)
  sign_in @user
end
```

`SignInHelper` can be reused both in testing as well as in regular application logic.