Project Description for Web Software Development

2015-2016 (CSE-C3210) Course Project

Last Modified: 2015-11-27

A quick step-by-step introduction to the project

The projects are carried out in groups of **three**. Piazza can be used to search groups and there is a special post especially for this. You can post advertisements or read and reply to advertisements of others. When creating an advertisement, it might be good to tell:

- about your ambitions (e.g. would you like to get full points or just pass)
- possible time/place constraints (i.e. when and how often can you meet)
- about yourself
- and somehow describe the group mates you are searching

After you have found your group, you should:

1. **Setup the repository:** Someone in the group creates a project in Niksula's GitLab and adds all team members to it. In addition, **wsd-agent** needs to be added to the project as reporter and project needs to be set to private (same steps as in Exercise round 1). *The group is considered **registered** when all members of the group have been added to it and wsd-agent is added as a reporter.*
2. **Write and commit the project plan:** Write your project plan, then commit and push that as the readme file of your project. 'That is a file called `README.md` in the root of your project. Plain text goes well, but you can also use markdown. If you want to use images in your project description (e.g. database schema, layout sketches) you can add those to the project as well and link them in *markdown*.
3. **Create an issue:** After you have your project plan finished, create an issue in GitLab where you request your plan to be reviewed and assign the issue to **wsd-agent**.
4. **Implement:** Start implementing your project after your project plan is accepted (i.e. your issue has been commented and closed by wsd-agent). Your application will eventually be deployed to Heroku, so you may also want to read the Heroku Django tutorial before starting your implementation work. It should be noted that because Heroku provides only limited monthly resources for free, development and testing should be first performed locally (not on Heroku's remote servers). Otherwise, you may run out of your quota when you are supposed to deploy your project and when the course staff would grade what has been deployed.
    a. **Note** that if you get feedback on some specific aspect of your project plan and do not react accordingly in the final project, points will be deducted.
5. **Deploy** your project to Heroku. Someone from your group needs to register to Heroku.

There will be a demo on Heroku on one of the lectures in spring.

6. **Create a new issue** to your project in GitLab where you tell that the work is finished and assign that to to the person that gave you feedback on the project plan.

Deadlines

- Group registration - 13.12.2015 midnight
- Project Plan - 20.12.2015 midnight
- Final submission - 20.2.2016 midnight (end of period III)
- Project demonstrations - few weeks after submission, announced when space has been reserved

Group registrations and project plans are reviewed continuously as they come in and we encourage you start as soon as you can.

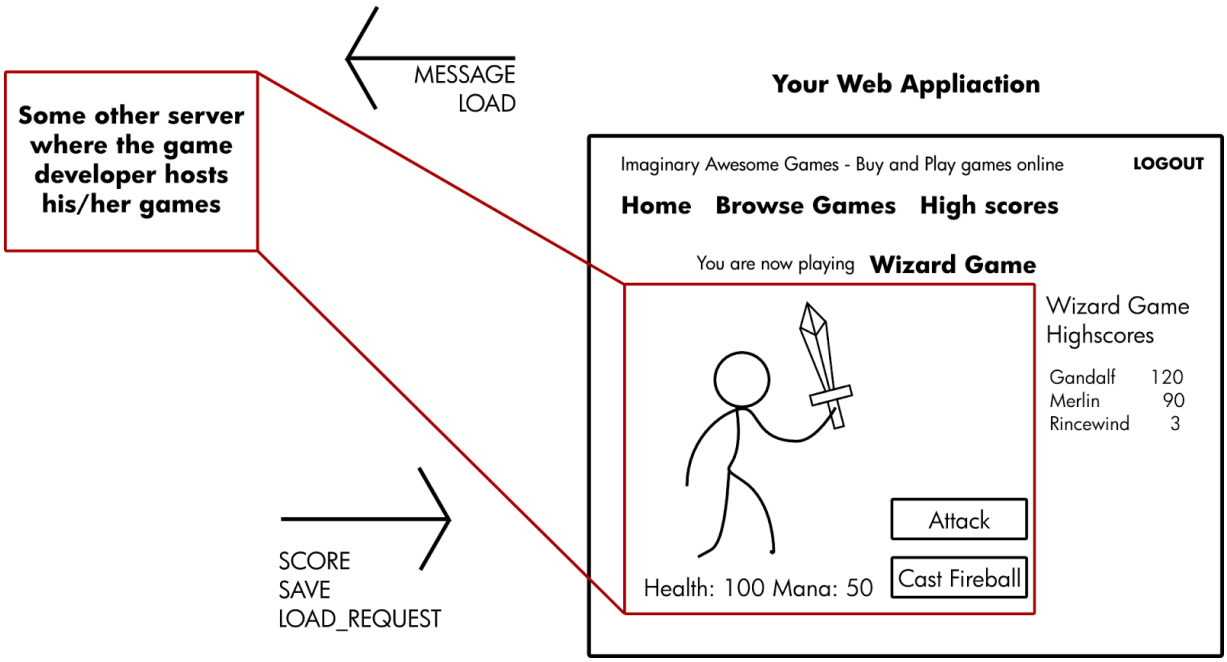
Project Topic and Requirements

The topic for the course project for CSE-C3210 2015-2016 is an **online game store for JavaScript games**. The service has two types of users: players and developers. Developers can add their games to the service and set a price for it. Players can buy games on the platform and then play purchased games online.

At the end, you deploy the project to Heroku. Game developers are not supposed to upload the JavaScript files of a game to the server. Instead, games are added by providing an URL to an HTML file that contains all assets and JavaScript links to JavaScript files.

The project has to be coded using the Django framework and has to include both client and server side code. On the client side, we strongly suggest using jQuery as a JavaScript library throughout the project. Other JavaScript libraries are allowed but not supported by the staff. Use of CSS libraries such as Bootstrap is allowed and recommended. Don't make your project a single-page-app.

You should implement both server and client side code mostly by yourself and not compose the service from various third party Django apps. The motivation of this is that before using third party applications, it is good to understand the basics and be able to write simple services just by yourselves.



Picture 1. *Rough outline of the service (including interaction with the game). Note that this is just an example. You are free to decide on the style and layout.*

Game Developer Information

The game developers have an inventory of games that are on sale on the site. New games are added to the inventory by giving a link to an URL of the game, which is an HTML file that should be displayed in an iframe to the player. The platform must support a simple message system, the games will send message events to the parent document informing on the score of the game. The platform should have a global high score for each game, where top scores for each game is displayed.

Similarly to submitting high scores, the service should accept save/load game state messages. The save message can contain arbitrary JSON formatted data containing a state of a game and save it to the database. Similarly, if there is a saved game state for that game and that particular player, the service should send a message to the game, so the state can be loaded and the player can continue playing from their previous state.

Sample Messages/Document the protocol

Game and the game service communicate with `window.postMessage`. All the messages must contain a `messageType` attribute, which can be one of six things:

- **SCORE**
  - sent from the game to the service, informing of a new score submission
  - the message must also contain `score` attribute
- **SAVE**
  - Sent from the game to the service, the service should now store the sent game state
  - the message must also contain `gameState` attribute, containing game specific state information
  - You can assume that the `gameState` is serializable by JSON.stringify

- **`LOAD_REQUEST`**
  - Sent from the game to the service, requesting that a game state (if there is one saved) is sent from the service to the game
  - The service will either respond with `LOAD` or `ERROR`
- **`LOAD`**
  - Sent from the service to the game
  - Must contain `gameState` attribute, which has the game specific state to be loaded
- **`ERROR`**
  - Sent from the service to the game
  - Must contain `info` attribute, which contains textual information to be relayed to the user on what went wrong
- **`SETTING`**
  - Sent from the game to the service once the game finishes loading
  - Must contain `options` attribute, which tells game specific configuration to the service. This is mainly used to adjust the layout in the service by providing a desired resolution in *pixels*, see examples for details.

Example message SCORE from game to the service

```
var message = {
      messageType: "SCORE",
      score: 500.0 // Float
};
```

Example message SAVE from game to the service

```
var message =  {
      messageType: "SAVE",
      gameState: {
            playerItems: [
                  "Sword",
                   "Wizard Hat"
            ],
            score: 506.0 // Float
};
```

Example message LOAD_REQUEST from game to the service

```
var message = {
      messageType: "LOAD_REQUEST"
};
```

Example message LOAD from service to the game

```
var message = {
      messageType: "LOAD",
      gameState: {
            playerItems: [
                  "Sword",
                   "Wizard Hat"
            ],
            score: 506.0 // Float
```

```
	}
};
```

Example message ERROR from service to the game

```
var message =  {
	messageType: "ERROR",
	info: "Gamestate could not be loaded"
};
```

Example message SETTING from service to the game

```
var message =  {
	messageType: "SETTING",
	options: {
		"width": 400, //Integer
		"height": 300 //Integer
	}
};
```

An example “game” is available and served as HTTP (http://webcourse.cs.hut.fi/example_game.html) and also in the wsd2015 repository (wsd2015/examples/example_game.html). It has the features that your service should support (sends player's score and handles incoming messages). It additionally supports saving and loading of game state.

Payment Service

Players pay for the games they want to play by using an external Simple Payments service. This service mimics an online bank payment service. Documentation and usage examples can be found on the Simple Payments site: http://payments.webcourse.niksula.hut.fi/

Note About Security

The simple messaging system used by the **game** platform can easily be tampered with JavaScript (e.g. to achieve higher scores on games/modify the saved game state). You do not need to implement any measures to protect against this.

Also plain HTTP is used with **payment service**. (we haven’t bought a certificate for it)

Besides these exceptions you should aim to make your project secure. Student projects have often had problems with authorization and authentication or they could have allowed the user to e.g. tamper with data coming back from the payment service or do script injections to items shown to other users.

Think about how you (or someone else) could attack your software... We will :)

Note About *Software*

The course is a web **software** development course. Every group member has to contribute in the software - meaning that it is not enough to concentrate on HTML/CSS only. Ideally everyone should be given a possibility to try and learn all the topics. Remember also that the course project is a learning opportunity, so remember to leave your comfort zone and try new things. If there is a guru in you group, do pair programming with them and gain a skill that will benefit you in the future.

Generic requirements

- Valid CSS and HTML (use a validator... we will)
    - We also encourage to use linting services for checking JavaScript code (http://jshint.com/ and http://www.jslint.com/)
- The service should work on modern browsers, especially Firefox or Chrome which conform to standards rather well
- Code should be commented well enough
- Although in real life you might use an existing django app for some parts of your project, beware of “externalizing” all aspects of the project. We will only grade the part you wrote.

Signs of Quality

- Reusability
- Modularity
- Versatile use of Django’s features
- Sensible URL scheme (do not create a ‘single page app’)
- Security (basic security can be circumvented, e.g. acquiring other user’s session/credentials, privilege escalation, injection attacks etc. **TEST ENOUGH**)
- Going beyond the basics
- Crash & idiot proof (**TEST ENOUGH**)

Functional Requirements and Grading

Please note that in order to get max points, the solution needs to be well done, not just something resembling a solution. Where minimum points have been listed, you will get minimum points for a working solution that fulfills requirements. In general, working solution is worth half of the maximum points in order to get full points the implemented feature needs to be of excellent quality.

Mandatory Requirements

Minimum functional requirements

- Register as a player and developer
- As a developer: add games to their inventory, see list of game sales
- As a player: buy games, play games, see game high scores and record their score to it

Authentication (mandatory, 100-200 points):

- Login, logout and register (both as player or developer). Email validation is not required for the minimum points but is required to get more than 100 points. For dealing with email in Django see https://docs.djangoproject.com/en/1.8/topics/email/#email-backends *You do not*

need to configure a real SMTP-server, using Django's Console Backend is enough for full points.
- Use Django auth

Basic player functionalities (mandatory, 100-300 points):
- Buy games, payment is handled by a mockup payment service: http://payments.webcourse.niksula.hut.fi/
- Play games. See also game/service interaction
- Security restrictions, e.g. player is only allowed to play the games they've purchased
- Also consider how your players will find games (are they in a category, is there a search functionality?)

Basic developer functionalities (mandatory 100-200 points):
- Add a game (URL) and set price for that game and manage that game (remove, modify)
- Basic game inventory and sales statistics (how many of the developers' games have been bought and when)
- Security restrictions, e.g. developers are only allowed to modify/add/etc. their own games, developer can only add games to their own inventory, etc.

Game/service interaction (mandatory 100-200 points):
- When player has finished playing a game (or presses submit score), the game sends a postMessage to the parent window containing the current score. This score must be recorded to the player's scores and to the global high score list for that game. See section on Game Developer Information for details.
- Messages from service to the game must be implemented as well

Quality of Work (mandatory 0-100 points)
- Quality of code (structure of the application, comments)
- Purposeful use of framework (Don't-Repeat-Yourself principle, Model-View-Template separation of concerns)
- User experience (styling, interaction)
- Meaningful testing

Non-functional requirements (mandatory 0-200 points)
- Project plan (part of final grading, max. 50 points)
- Overall documentation, demo, teamwork, and project management as seen from the history of your GitLab project (and possible other sources that you submit in your final report)

More Features

Save/load and resolution feature (0-100 points):
- The service supports saving and loading for games with the simple message protocol described in Game Developer Information

3rd party login (0-100 points)
- Allow OpenID, Gmail or Facebook login to your system. This is the only feature where you are supposed to use third party Django apps in your service.

RESTful API (0-100 points)

- Design and Implement some RESTful API to the service
- E.g. showing available games, high scores, showing sales for game developers (remember authentication)

Own game (0-100 points)

- Develop a simple game in JavaScript that communicates with the service (at least high score, save, load)
- Note that it does not need to be the greatest game ever, going a little beyond the given example test game is enough.
- Also note, that you can only get points for one game that you develop.
- You can host the game as a static file in your Django project or elsewhere. But you should include it in your repository.
- *Do not start this before you have a working implementation of the service*

Mobile Friendly (0-50 points)

- Attention is paid to usability on both traditional computers and mobile devices (smart phones/tablets)
- It works with devices with varying screen width and is usable with touch based devices (see e.g. http://en.wikipedia.org/wiki/Responsive_web_design)

Social media sharing (0-50 points)

- Enable sharing games in some social media site (Facebook, Twitter, Google+, etc.)
- Focus on the metadata, so that the shared game is "advertised" well (e.g. instead of just containing a link to the service, the shared items should have a sensible description and an image)

Requirements for Different Stages

Project Plan

General description of what you are doing and how you are doing that (what kinds of views, models are needed), how they relate to each other, and what is the implementation order and timetable. We offer one sample of project plan structure.

- What features you plan to implement?
- Are there some extra features not listed in the project description what you plan to implement?
- For each feature, how do you plan to implement it?
- Adding information on how you plan on working on the project is recommended as well (will you meet face-to-face regularly, will use some project management tools, etc.)

Final Submission

When submitting your project, 1) create an issue and assign that to **to the same person that gave you feedback on the original project plan** 2) add the following information to your Readme.md (you can also link to a separate report from your readme):

- Your names and student IDs
- What features you implemented and how much points you would like to give to yourself from those? Where do you feel that you were successful and where you had most problems. Give

sufficient details, this will influence the non-functional points awarded.
- How did you divide the work between the team members - who did what?
- Instructions how to use your application and **link to Heroku** where it is deployed.
    - If a specific account/password (e.g. game developer) is required to try out and test some aspects of the work, please provide the details.

Project Demonstrations

- 30 min demo with course personnel
- whole group required to be present
- booking system will be available after the exam week in February 2016.

Grading

In the grading of previously listed features, correctness, security, coding style, usability, documentation, and your own tests will all be considered. Generally all members of the group receive the same grade from the project but exceptions can be made if the workload isn't distributed roughly equally. Exceptions can also be suggested by the group if someone has done more than their fair share of the work.

Tentative grade limits:

- Grade +1 900 points (minimum to pass the project)
- Grade +2 1100 points
- Grade +3 1400 points

The exact grade limits will be decided after all the projects have been delivered. Also notice that there are 1700 points in total.

Disclaimer

By granting us access to your project you also acknowledge that the contents of the project can be used for research and improving the project. Any personally identifying information will not be published.