Browser Chronicles

A browser-based gamebook platform

A Gamebook

A gamebook[1] is a fiction book, where the reader can influence the the story by making choices.

Usually, after a section of text, different possible actions are presented. Each of them is associated to a number, pointing to the text section corresponding to the player's action (i.e. *talk with the frog — go to page 1234*).

Eventually, the story arrives to an end, and depending on his choices, the player wins or loses.

Browser Chronicles, overview

Browser Chronicles is a game engine that could run different *stories*.

A *story* is organized as a graph, composed by *steps*. Each *step* defines the interaction with the user and could give the opportunity to go to another *step*; in this case, the destination *step* depends on the user input. A special type of *step*, called *end*, informs the user about the result of his path (win/lose).

Required views

Browser Chronicles presents to the user two main views: *play* and *show*, detailed below. These views should be able to read an XML file describing a *story*, and must be implemented using HTML, CSS and AngularJS.

- *play*: interactive view, meant for the end players, that starts displaying the first *step*, letting then the user play the game, by moving from a *step* to the next one according to player's decisions.
- *show*: service view, useful to check the validity of the *story*; it illustrates the complete graph of the *story*, highlighting the winning path composed by the minimum number of *steps*.

[1] in French: *livre dont vous êtes le héros*.

You should create the project by running `yo angular`[2]. Because of a bug of the generator, in order to have everything work properly, you should also run `npm install grunt-karma --save-dev`. Enable then the XML support installing `bower i --save angular-xml`, then configuring it[3].

Protocol details

The XML file root element must be named `story`. Many `step` elements appear inside the root element. Each `step` element needs to record some data: `id`, `type`, and some details related to its specificity.

For example:

```
<story>
  <step id="0" netxStep="3" title="Welcome!"/>
  ...
  <step>
    <id>132</id>
    <type>end</type>
    <win>false</win>
  </step>
</story>
```

As you can guess, our engineers' ideas are not so clear about the format: you'll need to refine this definition into something acceptable and then formalize it with an XML Schema Definition, that should embrace the *step* types present in your *stories*, and be easily extensible for new possible types. This enables the possibility to validate a *story* correctness before loading it on the game engine, and ensure a simple but efficient XML structure.

Step types

A *step*, independently from its `type`, could have a `title` and a `description` (both always optional), that must be displayed to the player, if present. The `type` defines the further interaction requested to (in general) move to another *step*.

---

[2] refer to the project page for further usage info: https://github.com/yeoman/generator-angular, you are encuraged to use the shipped generators to create additional code.
[3] you'll need the HTTP interceptor, more info: https://github.com/johngeorgewright/angular-xml

Multiple choice

This `type` represents the most basic interaction with the user. It will show `title` and `description` as mentioned before, and a number of exclusive options. When the player selects an option and clicks the "next" button, the application updates the view to the *step* corresponding to the player's choice (hint: use the `ids`).

The number of options must be greater than zero, but it could be *one*. In this case, the option label will not be shown, but the "next" button will point directly to the corresponding *step*. This could be used to introduce the *story*, or split the interactions with narrative parts.

Riddle

For this `type`, the `description` will likely consist in some hints. The player should then guess a word (or a sentence) and write it into an *input* box. If the answer does not match with any possible solution, an error message is shown, otherwise the player is sent to the associated next *step*.

The number of accepted answers must be at least one, but potentially we could have different next steps associated to different replies.

End

The only type not offering a next *step*. This type exhibits `win`, a boolean indicating the outcome of the game. The player must be informed on the value of `win` with a big colored text (eg. You win/You lose).

In case of victory, a short sentence should inform the player about the number of *steps* he did and the *minimum number* of *steps* required from the start to a successful end.

A *story* has in general many `end` *steps*, but needs at least one happy ending.

Memory

For this *step* `type`, the player needs to solve a memory game.

An even number of reversed playing cards appears on the screen; the user selects two cards, by clicking on them; if they are identical, they disappear, otherwise their face is hidden again, and the user could select two cards. When

all the pairs have been found, the player could click on the "next" button to go to the next *step*.

This step type should work out-of-the box with a predefined set of cards, but the XML format should enable the customization of card faces. To modulate the difficulty of this `type`, the number of shown card should be customizable (some constraints apply), with the effect to take a subset of the deck or duplicate some couples.

Maze

Alternatively to `memory`, you could implement a 2D maze. The XML will store the number of rows and columns of the puzzle, that will then be generated[4] by the application.

The game starts with the player's piece at the starting point; using the keyboard arrows (bonus: think about touch devices) the users move his piece inside the maze. When the exit is reached, the game stops and the "next" button is activated.

You name it

Neither `memory` nor `maze` inspire you? You can formalize an alternative, write down a short description and send it to us[5].

Evolution — where the bonus points hide

This section presents some possible extensions to Browser Chronicles, that will be evaluated *only* if the requisites explained in "acceptance criteria" are met. Before starting your way to implement any of these, talk to us, so to ensure that your plan is a good plan.

Riddle answer flexibility

Add an option to the `riddle type`, relaxing the string equality constraint to something more reasonable; indeed, in some cases, "close enough" answers should be accepted.

---

[4] how to generate a maze: https://en.wikipedia.org/wiki/Maze_generation_algorithm
[5] as a private message on Piazza

For example, you could calculate the Levenshtein distance[6] between the expected answers and the given one, and take the decision based on the resulting value.

Rich text

Our authors would like to insert rich content on the *stories*, such as t**ex**t fo**rm**at*ting* and images. In particular, HTML fragments should be supported and correctly displayed if inserted in any rendered text attribute.

Persistence

When a player abandons temporary a *story* and then get back to it, he should be able to get back to the same *step* without re-doing all the path. The status could be saved either on the browser or on the server.

Author mode

Enable users to create *stories* without writing XML. An additional view, called *edit*, should be created, containing a GUI showing the *story* graph. Each *step* could be modified by clicking on it and changing its attributes on a form.

This view includes an "export" button to show the XML code (or a "save as" browser window).

Security concerns

Talking about security: how this platform could be exploited? Find a possible way to fix the major security flaws and implement it (yes, you'll need server-side code).

Acceptance criteria

- Ship your application with some example *stories* as to show its potential; these *stories* would be listed by the application, and loaded on demand using an Ajax call (AngularJS `$http`).
- The Web Console[7] is your friend *during development*, but remember to remove debugging code before shipping the application! Moreover,

[6] more info: https://en.wikipedia.org/wiki/Levenshtein_distance
[7] https://developer.mozilla.org/en/docs/Web/API/Console

errors must be handled by explaining the issue to the user: the application must *never* freeze with a red error on console.
- The *play* view loads the *story* corresponding to a parameter given in the URL, and it must implement `multiple choice`, `riddle`, `end` plus one more type: `memory`, `maze` or your proposal (to be accepted).
- The *show* view loads a *story* in the same way, but it shows all the *steps* at once, as a graph (find a JavaScript library to do that); the *steps* `id`s should be rendered as nodes, and the *next step* connections as links. The shortest path between the starting step and a winning end should be highlighted in green. If there is no path from the start to the victory, show a warning.
- For both the *show* view and the `end type`, you'll need to compute the shortest path between the start and a successful end. At least this function must be unit tested. `yo` included for you a JavaScript test suite: Jasmine[8]; `grunt test` will run the tests saved in the directory `test/spec`.
- In case the potential evolutions change substantially the structure (e.g. their presence change considerably the behavior of the system), they should be implemented separately. In any case explain how to enable them on the readme file.

Report

A report describing your efforts is requested. Be professional, focusing on these points:

- Difficulties you encountered during the development and how you overcame them. [½ to 2 pages]
- Extensibility: how easy is it to add a new *step* `type`? Which actions should be taken to adapt your code and the XSD? Give a complete example using `memory` or `maze` (the one you didn't implement). [some code allowed]
- Complexity: calculate (and explain) the complexity of your shortest path finder algorithm. [½ page]
- Tests: enumerate you test cases for the shortest path function, explaining the reason under each case. Add any other test-related information concerning your project. [½ to 2 pages]

[8] documentation on: http://jasmine.github.io/2.4/introduction.html

- Security: the solution, as described by this track, has some severe security issues that allow an expert user to break the game rules. Enumerate the possible flaws and propose structure fixes. [1 page]
- If you implemented some evolutions, describe your choices.

Evaluation

- ⅓ demos: think about the features you want to present,
    - Wednesday: peer-reviews, other developers will evaluate your product;
    - Thursday: customer-reviews, they want to see an MVP, don't let them down!
- ⅓ report: truth, completeness, clearness.
- ⅓ code: see acceptance criteria.

Deliveries

- `yo angular` will prepare your Git repository and ensure these requirements are respected:
    - the application must start with `grunt serve`, the tests with `grunt test`.
    - your source code should live in a subdirectory of your repository, called `app`.
- The `story.xsd` file and some example *stories* must be stored into a subdirectory called `app/stories`.
- Final products and report on Sunday at 23:59 CEST. The expected tag is `vFinal`.