Documentation Gamebook

Version: Alpha 0.1.x

Table of contents

License

Installation

1. Install Meteor (https://www.meteor.com/install)
2. Download last release (https://git.informatik.fh-nuernberg.de/BudickDa45997/gamebook-webversion/tags)
3. Unzip it and go into directory of repository with you console. (cd /pathTo/unzippedDir)
4. Start application by typing 'meteor'
5. Visit localhost:3000 with Chrome or another modern browser (IE<11, Safari and Opera do not count as modern browsers).

Tutorials

The two tutorials offer a quick overview about what is possible with this framework. You can play them without any installation by visiting the URL: gamebook.meteor.com
The first tutorial is 'STORY 1' in the menu, the second one is 'STORY 2'.
You can view the code in /book/tutorial_one.js and /book/tutorial_two.js.

While tutorial one is only a short overview over the function of the framework, the second one is a short story about a player destroying a table and getting teleported around. In the end he has to face an armed gladiator and has to fight for his life.

The UI is very simple and created with materialize.css.
Materialize was chosen because of its modern app-like optic. But looking back bootstrap would have been a better choice because it has more functionalities and is used by more frontend-developer. Furthermore materialize.css for meteor is a little bit buggy.

Accessing the backpack is a little bit awkward: The icon to open the backpack is not shown (probably a bug). By clicking on the button with the word 'RUCKSACK' the player empties their hand into the backpack. Right click empties the right hand, left click the left hand. By opening the backpack with the button right of the 'RUCKSACK'-button, a dialog opens with the content of the backpack. By clicking with left or right on items, the player takes the item into the left or right hand.

On the top of the display the last event is shown. On the bottom next to the backpack the statistics of the player are shown. They change like the backpack and the text in real-time.

Basic principles of the framework

A short overview about the parts and mechanics of a story.

Story

An application can contain multiple stories. Which story plays can be decided by the URL or the user session, or any event in the code. In the example the playing story is set in /lib/router.js in Line 62 with the function call:

```
Gamebook.startStory(story);
```

The variable story is set by the URL in a switch-case with this code (this will start the story with name ‘storyOne’)

```
story = Gamebook.getStory('storyOne');
```

The story is a wrapper for different scenes and each scene contains several SceneObjects. A story contains an EnvironmentObject. This EnvironmentObject contains information about the environment of the scene, e.g. the weather.

Scenes

Every story is made of scenes. A scene contains all of its objects and all of it text. For every big text change a scene should be created.

SceneObjects

Do not confuse with the scene object, which is the JavaScript object of a scene. To avoid further confusion we will call objects of the type SceneObject ‘game objects’ from now on.
SceneObjects are objects the player can interact with. An object can be everything. In the text it is represented by a hyperlink. The important thing about SceneObjects is, that they have global and individual properties. Global properties are assigned by the name of the object.
Example:
Every ‘House’ has the same description ‘A building where people live in’. An individual house can have the property ‘Burning’ which gives it a fire damage of 20.

Individual properties are added by rules and effects. A rule is a key-value-datatype. An effect packs a set of rules.
Example:
The effect ‘Burning’ has the rules ‘firedamage’ ‘+20’ and ‘health ‘-20’. The object ‘House’ has this effect. It started with the effect ‘woodenHouse’ with ‘health’ ‘300’. Thus its effective properties are:

- health: 280
- firedamage: 20
- Description: ‘A building where people live in’
-

Calculating rules

When calculating statistics there are two rules:
Integers override: If the value is an integer, it overrides every value that came before. It will be an absolute value and will be manipulated by the rules coming after. If there isn’t any absolute value, the value is zero and will be manipulated.
Strings manipulate: If the value is a string (e.g. ‘+4’ or ‘-4’) the absolute value is manipulated by this value mathematical correct.
Example:

15 '+5' '-4' --- = 16	15 '+3' '-16' '12' --- = 12	'+3' '+16' ‘-15’ --- = 4

Objects

All the object of the gamebook are published via the object Gamebook in /gamebook/lib/gamebook.js.
The reason for this is, that the gamebook-framework should be packed into a meteor-package to make it easily reusable.

Gameobject

story: {}:

Here the active story is stored.

startStory: Function:

Parameter:

- story: Story: The story object that will be set to story and started.

Description:

Sets the story object as story and starts the story.

getStory: Function:

Parameter:

- storyname: String: The name of the story that will be loaded from stories to story.

Description:

Takes a story via the name from stories, and returns it.

stories: {}:

In here every created story is stored.

localStorage: Mongo.Collection:

A mongoDB-collection that that exists on the client only. Exists on the client only.

config:

defaultLanguage: String:

Here the language is set. Per default it is german.

startUiCountdown: Function:

Parameter:

timeInMs: Number: Length of the countdown in milliseconds
steps: Number: Length of every iteration in milliseconds
cb: Function: Function that is executed, when countdown reaches zero.

Description:

Starts a countdown and writes the current value into a Meteor Session variable that can be used in the view.

Return:

killSwitch: This value is used by stopCountdown to stop the countdown.

startSilentCountdown: Function:

Parameter:

timeInMs: Number: Length of the countdown in milliseconds
steps: Number: Length of every iteration in milliseconds
cb: Function: Function that is executed, when countdown reaches zero.

Description:

Starts a countdown, nothing else.

Return:

killSwitch: Number: This value is used by stopCountdown to stop the countdown.

stopCountdown: Function:

Parameter:

killSwitch: Number: The return value of startUiCountdown and startSilentCountdown.

Description:

Stops countdown function.

Return:

Nothing

Effect: Class:

Constructor:

Parameter:

- name: String: Name of the effect, this name is shown in the UI, so you should use the language of your story.
- rules: Array<Rule>: An array of rules who supply the properties for this effect.

Description:

Parameter are stored as member variables of the instance of this object.

getStats: Function:

Description: Returns from rules calculated properties of this effect.

Rule: Class:

Constructor:

Parameter:

- key: String: Name of the value (e.g. Gesundheit/Health) should be in the language of your story, because it is used in view.
- value: String/Number: Value of the key. If it is a String, it is a manipulator, if it is a Number it is an absolute value (for more information look at: Basic principles of the framework -> Calculating rules).

Description:

Parameters are stored as member variables of the instance of this object.

Story: Class:

Constructor:

Description:

Creates the following arrays:

- scenes: Here all the scenes are stored
- effects: All the global effects of the story are stored here (not implemented yet)
- sceneHistory: Every visited scene is stored in here. Last element is the scene the player is in now.

Sets the player object as -1. So we can check if a player is already set.

addScene: Function:

Parameter:

scene: Scene: Scene that will be stored in scenes.

Return

index: Number: The index of the scene in scenes. With this index we can start a scene directly with startScene(index)

createNewPlayer: Function:

Parameter:

player: Player: The player object of the player.

Description:

Tests if a player has already been defined. If there is a player it will send a debug message and overwrite the current player. It can be used to change the idendity. Be aware that the backpack will be overwritten.

startScene: Function:

Parameter:

index: Number: Will start the scene with this index from scenes.

Description:
The scene with the index of the parameter index will be started. If you call a scene that does not exist, the application will crash with a message.

addEffect: Function:
Parameter:
effect: Effect: The effect that will be added to effects

nextScene: Function:
Description:
Will start the scene after the current one. By adding one to the index of the currentscene.
Will throw an error and ignore command if there is no next scene.

previousScene: Function:
Description:
Will start the scene that played before the current scene.
Will throw an error and ignore command if there is no previous scene.

Start(): Function:
Description:
Starts the first scene of the story.

currentScene()
Description:
Returns the object of the current scene.
Return:
scene: Scene: current scene object

getSceneObject: Function:
Parameter:
_id:String: Id of a SceneObject in the current scene.
Description:
Returns a game object from the current scene as SceneObject.
Return:
gameObject: SceneObject: A game object of the current scene with the _id of the parameter.
Returns nothing when game object does not exist and throws an error.

getStats: Function:
Description: Returns from effects calculated properties of this effect.

Scene: Class:

Constructor:
Parameter:
weather: String: The weather that is set for the environment of this scene. The chosen weather should be set in text/weather.js. The game supports only 'rainy' and 'sunny'.
Description:
Creates the following arrays:
- sceneObjects: Here all the scenes are stored
- effects: All the global effects of the story are stored here (not implemented yet)

Creates an environment object and sets the weather via the parameter
Sets the player object as -1. So we can check if a player is already set.
Creates text: String that stores the text without keywords.
Creates onStart: Function, where a function can be defined that is always called when the scene is started.

updateWeather: Function:
Parameter:
weather: String: The weather that is set for the environment of this scene. The chosen weather should be set in text/weather.js. The game supports only 'rainy' and 'sunny'.
Description:
Changes the weather of the scene.

howIsTheWeather: Function:
Parameter:
language: String (optional): Get the text of the weather in the defined language. If not set the gamebook.config.defaultLanguage will be used.
Description:
Adds a short description of the weather to the text of the scene.
addEffect: Function:
Parameter:
effect: Effect: The effect that will be added to effects
addText: Function:
Parameter:
text: String: The text that is added to the scene
Description:
Adds text to the scene. Concatenates the text to the current text of the scene.
overrideText: Function:
Parameter:
text: String: The text that is added to the scene
Description:
Adds text to the scene. Deletes the current text of the scene.
createKeyword: Function:
Parameter:
text: String: The text that depends on the keyword. Must contain the keyword in within two braces ('This [keyword] text.').
Description:
Adds game object to the scene and returns this game object as SceneObject for further manipulation.
Return:
sceneObject: SceneObject: A game object that can be manipulated.
updateText: Function:
Description:
Updates the text in the view. Should be called after the use of addText, overrideText or createKeyword and after every operation that changes the scene (gameobject.take(), gameobject.destroy etc.) Makes changes visible to the user.
findSceneObject: Function:
Parameter:
_id:String: ID of a SceneObject in the scene.
Description:
Returns a game object from the current scene as SceneObject.
Return:
gameObject: SceneObject: A game object of the current scene with the _id of the parameter. Returns -1 when the object was not found.
removeSceneObject: Function
Parameter:
_id:String: ID of a SceneObject in the scene.
Description:
Removes the object with the ID of _id from the scene
Return:
gameObject: Boolean: If object was removed returns true, if it was not found it returns false.

getStats: Function:

Description: Returns from effects calculated properties of this effect.

Objects that are used by the objects in gamebook

These objects are not accessed by the user directly. Except SceneObject which is the return value of createKeyword.

SceneObject: Class:

Constructor:
Parameter:
name: String: The name of the object. The name is/are the word(s) that were in the brackets at createKeyword. This name is used to get global properties for the object.
addEffect: Function:
Description:
Adds an effect to the object
take: Function:
Description:
Removes object from scene and adds it into players backpack.
drop: Function:
Adds current Object to the scene and removes it from the backpack. Not that you cannot drop object from your hands.
destroy: Function:
Description:
Removes object from the scene. Note that you cannot destroy objects in you backpack or your hand.
addEvent: Function:
Parameter:
event: string: name of the event, depends on the UI.
callback: Function: Function that is executed, when event is called.
name: string: Give a default name to this event.
Description:
There is more in chapter Important methods -> Events
addEvents: Function:
event: [string]: Array of different events that trigger one function.
(E.g. : ['left','right'], left and right trigger the same function).
callback: Function: Function that is executed, when event is called.
name: string:
Description:
There is more in chapter Important methods -> Events
checkEvent: Function:
Parameter:
event: String: Check if the event with the name event exists for this event.
checkOverride: Function:
Parameter:
event: String: Check if there is a description for this event.
getStats: Function:
Description:
Returns from effects calculated properties of this effect.
beforeHook: Function:
Description:
Is used internally, do not mess with it. If you want to write your own hook, check the chapter Important methods -> Hooks
afterHook: Function:
Description:

Is used internally, do not mess with it. If you want to write your own hook, check the chapter Important methods -> Hooks

fireEvent: Function:
Parameter:
event: string: Name of the event, that will be fired.
Description:
Will always fire the beforeHook of this object. Then checks if event does exist. If not the function ends here. If the event exists, it will be executed. Then the afterHook is fired.

Player: Class:

Constructor:
Parameter:
Effects: [Effect]: Optional: An array of effects that are used on the player.
destroy: Function:
Description:
Calls onDestroy hook of player.
addToBackpack: Function:
Parameter:
object: SceneObject: Object that is put into backpack.
Description:
Puts object into backpack. When you want to put an SceneObject from the scene into the backpack, call take() on the object and it takes care of everything. Use addToBackpack only if you know what you are doing, or you end with a lot of copies of your object.
addToBackpackFromLeftHand: Function:
Description:
Takes the object from the player's left hand and puts it into the backpack.
addToBackpackFromRightHand: Function:
Description:
Takes the object from the player's right hand and puts it into the backpack.
removeObjectFromBackpack: Function:
Paramter:
_id: The _id of the object that will be removed.
Description:
Deletes object from backpack via _id. Returns true when success, false when object not found.
Return:
Boolean: True if object was removed, false if object had not been found.
getObjectFromBackpack: Function:
Paramter:
_id: The _id of the object that will be removed.
Description:
Deletes object from backpack via _id. Returns true when success, false when object not found.
Return:
Boolean: True if object was removed, false if object had not been found.
takeLeftHand: Function:
Parameter:
object: SceneObject: Object that will be added to the left hand.
takeRightHand: Function:
Parameter:
object: SceneObject: Object that will be added to the left hand.

addEffect: Function:
Description: Adds an effect to the object
getStats: Function:
Description: Returns from effects calculated properties of this effect.
beforeHook: Function:
Description: Is used internally, do not mess with it. If you want to write your own hook, check the chapter Important methods -> Hooks
afterHook: Function:
Description: Is used internally, do not mess with it. If you want to write your own hook, check the chapter Important methods -> Hooks

Environment: Class:

Constructor:
Parameter:
chosenWeather: String: default('random'): Name of the weather, if weather is 'random' everytime the weather is fetched it is chosen randomly from the config.
Description:
Sets weather. Weather has to be defined in gamebook/lib/text/weather.js.
getWeather: Function:
Description:
Returns weather.
Return:
weather: {}: a weather as is defined in gamebook/lib/text/weather.js.
setWeather: Function:
Parameter:
chosenWeather: String: Name of weather. Must be defined in gamebook/lib/text/weather.js.
Description:
Gets weather by name (chosenWeather) from gamebook/lib/text/weather.js and sets it as weather for getWeather() to return.

Important methods

When creating a story there are several methods which will be used regularly.
Best practice: Use methods wherever you can.

Text

There are three ways to add text to a scene.
var newScene = new Gamebook.Scene();

You can concat text to the text of the scene via the scene objects method:
newScene .addScene('This text is added. ');
newScene .addScene('Another text is added. ');
To update the text in the view at runtime (e.g. in an event callback), updateText() has to be called whenever changes are meant to happen.
newScene.updateText();

You can overwrite all preexisting text with
newScene.newScene overrideText('One text to rule them all!
'); //we can use html

A third way is to create a keyword:
var gameobject = newScene.createKeyword('And there is one [Ring] to rule them all.');

This will create a text like this:

One text to rule them all!
And there is one Ring to rule them all.

Keywords

Keywords are text the user can interact with. If they are destroyed or taken by the user, they vanish from the text. The text of a keyword has two parts:

- The keyword
- The surrounding text

The surrounding text gives the author the possibility to describe the surrounding of the keyword. This text will vanish from the scene with the keyword.

To create a keyword in a scene, the author can call the createKeyword method from the scene.
The method returns a gameobject that can be manipulated.

Events

Events are functions that are called by the user in the UI. The allowed names are defined in gamebook/lib/objects/scene_object.next.js. In line 24 in the array called daVinciEvents. These daVinciEvents are implemented in the UI and are the only one that can be called by the user, thus are the only events that can be used.
There are two ways to add an event to a game object.

var exampleGameobject = scene.creatKeyword('This is an [examplekeyword].');

```
exampleGameobject.addEvent('left', function(){
   //do something
}, 'Do something with the left hand');
//or
```

```
exampleGameobject.addEvents(['left', 'right'], function(){
   //do something
}, 'Do something with bot hands');
```

The defined function is executed when the defined event was used on the keyword of exampleGameobject by the user. The last defined function on a keyword overwrites the function that had been defined before.

Hooks

Hooks are function that can be defined by the author. They will be executed on different occasions.

```
SceneObject.beforeHookOverride = function(){
   //this code is executed before an event is fired
}

SceneObject. afterHookOverride = function(){
   //this code is executed after an event had been fired
}

SceneObject. afterDestruction= function(){
   //this code is executed after the object had been destroyed
}

Scene.onStart = function(){
   //this code is executed whenever this scene is started.
   //Beware that if a scene is started multiple time, this function is called every time.
}


Player.beforeHookOverride = function(){
   //this code is executed before an event is fired
}

Player. afterHookOverride = function(){
   //this code is executed after an event had been fired
}

Player. afterDestruction= function(){
   //this code is executed after the player had been destroyed
}
```

Change scene

Scene changes make only sense, when they happen within a callback function (e.g. the function of an event).
The story object take care of the scenes and provides us some useful functions:
startScene: If there is a lot of jumping between scenes, this is your function. The function needs a parameter. This parameter is the index of the scene. The index is the return value of addScene.

```
var story = new Gamebook.Story();
var firstScene = new Gamebook.Scene();

var firstSceneIndex = story.addScene(firstScene);
```

The author can call this scene always via:
Story.startScene(firstSceneIndex);

previousScene: This will always start the scene, that played before the current scene.

nextScene: This will always start the next scene by adding one to the index of the current scene and calling the scene with the index of the result.
In short: The scene added to the story, after the current one is started.
A history of the scenes that have been played until now, are stored in the array story.sceneHistory.

Anything else

can be done by JavaScript. Text can contain html.
There are a lot of undocumented features that can be used. Gamebook.localStorage can be used to persist user data on the client. It works like a mongo collection. You can use JavaScript functions like promt to get user input or you can write your own input fields with html and open then as popup with $('#yourHtmlDivsId).openModal();.
You can use all the functions of meteor (be careful to use client-function like Session.get('anything') within an if(Meteor.isClient){}).

With Great Power Comes Great Responsibility.
Be careful that stories are always created by a trusted source. If Mallory can write a story for all user, you will have a problem and you will cry.

Meteor-specific syntax

There is some syntax that will require some deeper knowledge of Meteor. Here is a little crash course:

Tracker.Dependency

```
this.effectsDep = new Tracker.Dependency();
```

and

```
this.effectsDep.changed();
```

and

```
this.effectsDep.depend();
```

Helps meteor to make these variables reactive which means that changes are directly pushed into the view.

Session-variable:

Those variables are reactive and can only be used on the client. Using them on the server will result into an error.

Session.set('foobar', value); //Sets session-variable named foobar with the value of the variable value.
Session.get('foobar'); //Returns value of foobar.

export class

Export class makes the class global.
You find more information in the Git-Repository of harmony (https://github.com/mquandalle/meteor-harmony).

Best Practice

- Use object methods wherever you can.
- Check if you are on the server/client when you do stuff.
- Don’t let Mallory write you stories.
- Have fun, but not too much.
- Don't eat earwax avoid roasted cabbage.
- Respect the terms of GPL and CC at all time, or else…

Future plans and ideas

These are ideas that could be implemented in the future.

Damage over time

An effect is added each x seconds. For example burning could do a damage of y every x seconds. After a period of time, the burning object is destroyed.

Random values or critical hits

When dealing damage via an effect a random multiplicator is generated. If the number is high enough critical damage is dealt. If it is very low, the strike misses.

Dependencies between keywords and syntax

If we have to keywords that are in the same sentence, we need a new sentence when one keyword is gone.

“There is an [apple] and next to a [sword] on the table.”
The apple get eaten.
“There is a [sword] on the table.”
The sword is taken.
“The table is empty.”

A construct like this is not possible at the moment. To support a construct like this, the author should be possible to write sentences for different states of those objects.