

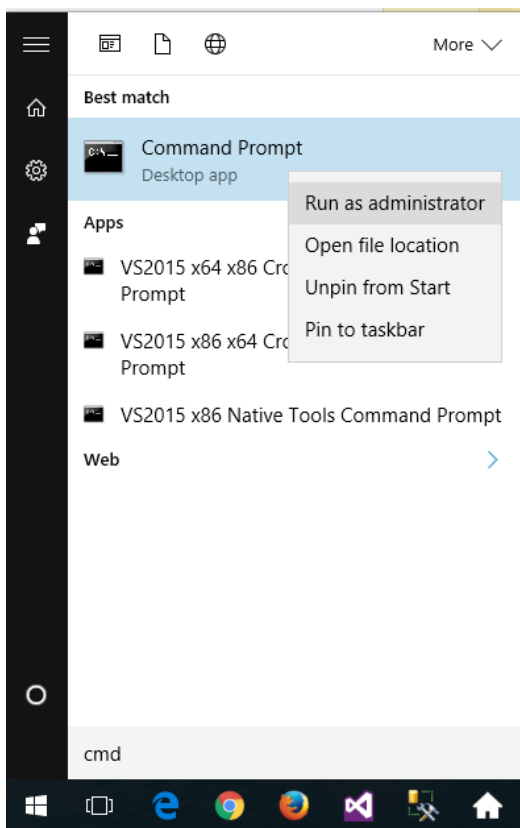
StoryScript tutorial

1 *Setting up Visual Studio Code*

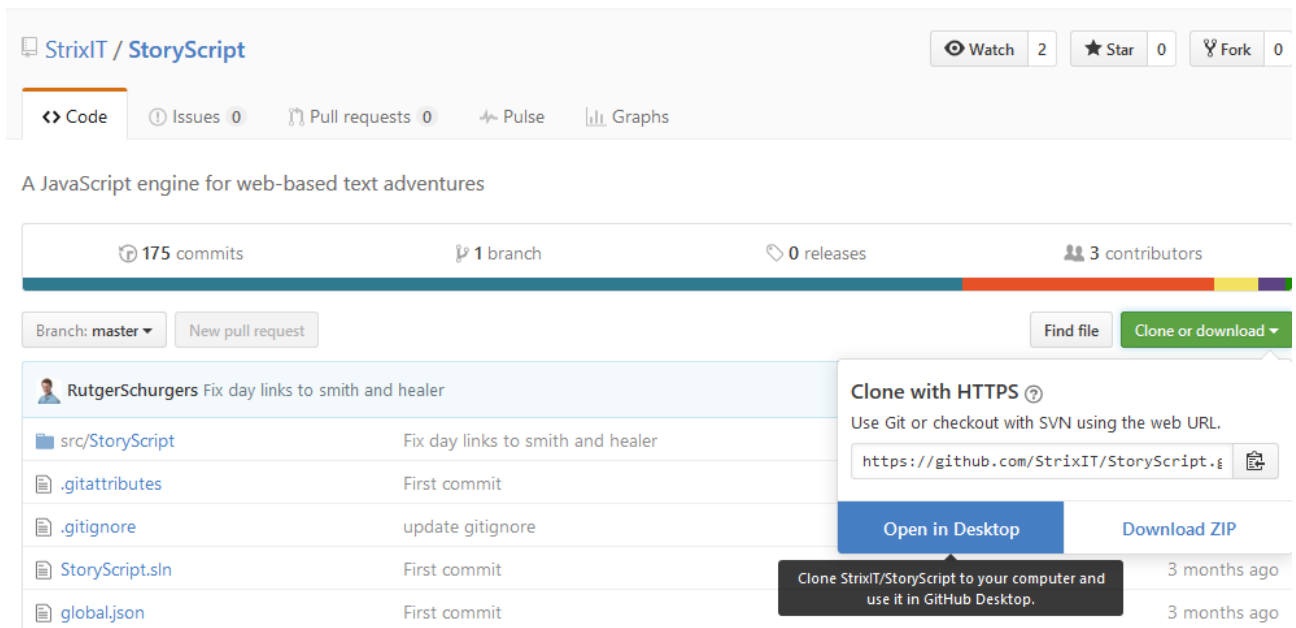
To work with StoryScript effectively, you should use an editor that supports you in creating your game components like HTML and TypeScript files. A good choice is **Visual Studio Code**. This is a product related to the full-fledged Visual Studio, but much more lightweight and, more important, also available on operating systems other than Microsoft Windows. Of course, you can also use the full Visual Studio if you want.

This tutorial will assume you are using Visual Studio Code, and it will help you set it up for Windows. The first step is to download and install it. Go to <https://code.visualstudio.com/download> and download and install the program (the default settings are fine if you are unsure what to choose when you are presented with options).

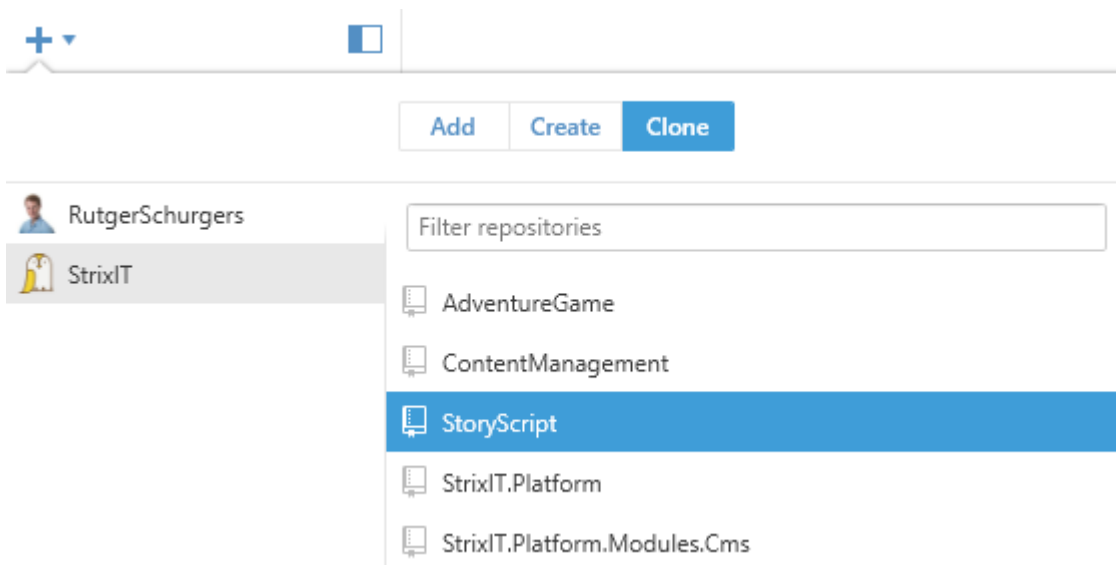
Once done, you will also need to install **NodeJs** to be able to build and run your game. Download it from <https://nodejs.org/en/> and run the installer. Once done, run the following command in the command prompt: **npm install -g gulp**. To access the command prompt, click the windows button in the lower left corner of your desktop and type '**cmd**' to have it appear. Right-click it and choose '**Run as administrator**':



Now that you have your development environment set up, it is time to get the StoryScript code from GitHub. Navigate to <https://github.com/StrixIT/StoryScript>, click '**clone or download**' and click the '**Open in Desktop**' button:

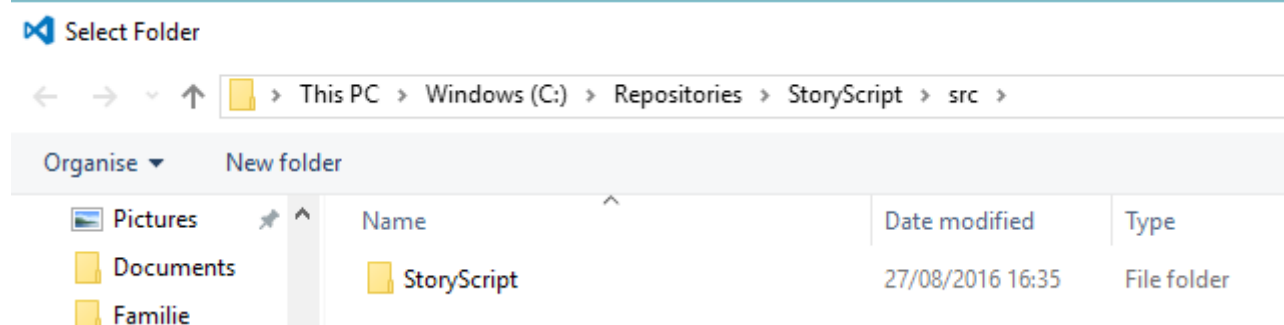
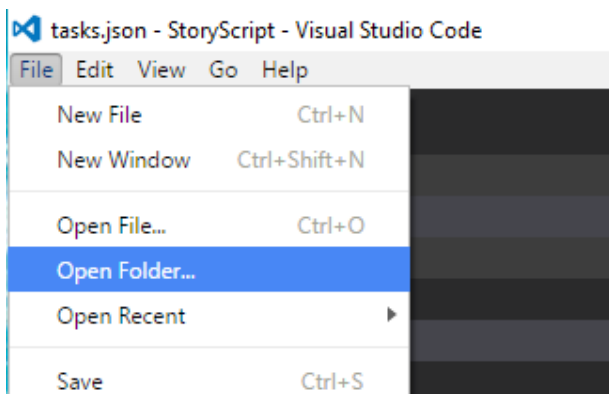


You will be redirected to the GitHub desktop website. Download and install GitHub desktop and start the program. Use the '+' in the top left corner to go and select the StoryScript repository and clone it to your computer:

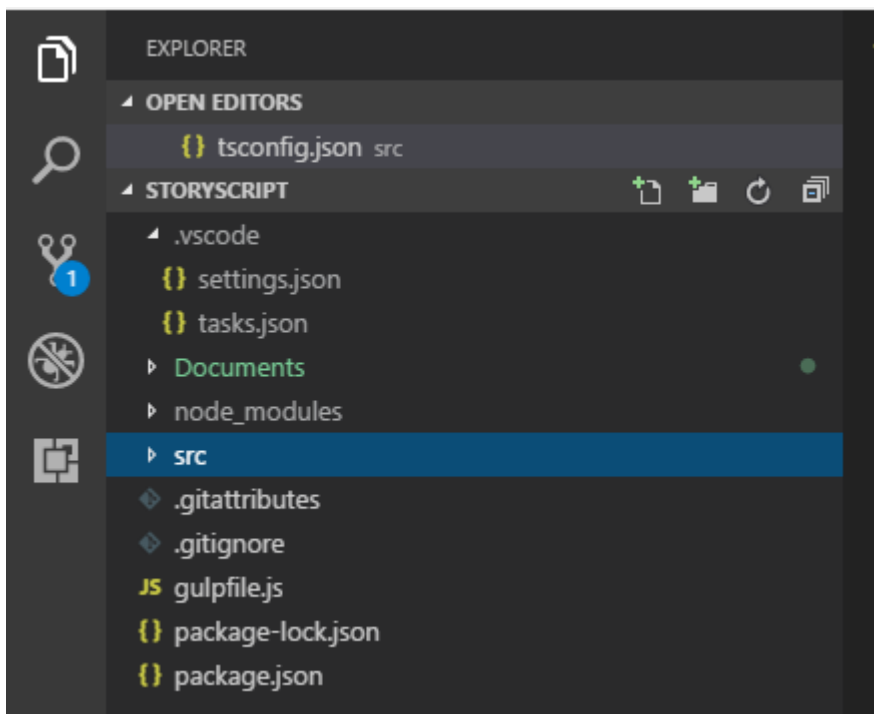


Select a folder to clone to and press ok. Once cloning is done, you are ready to start Visual Studio Code.

From within Code, open the folder you cloned StoryScript to via File → Open folder:



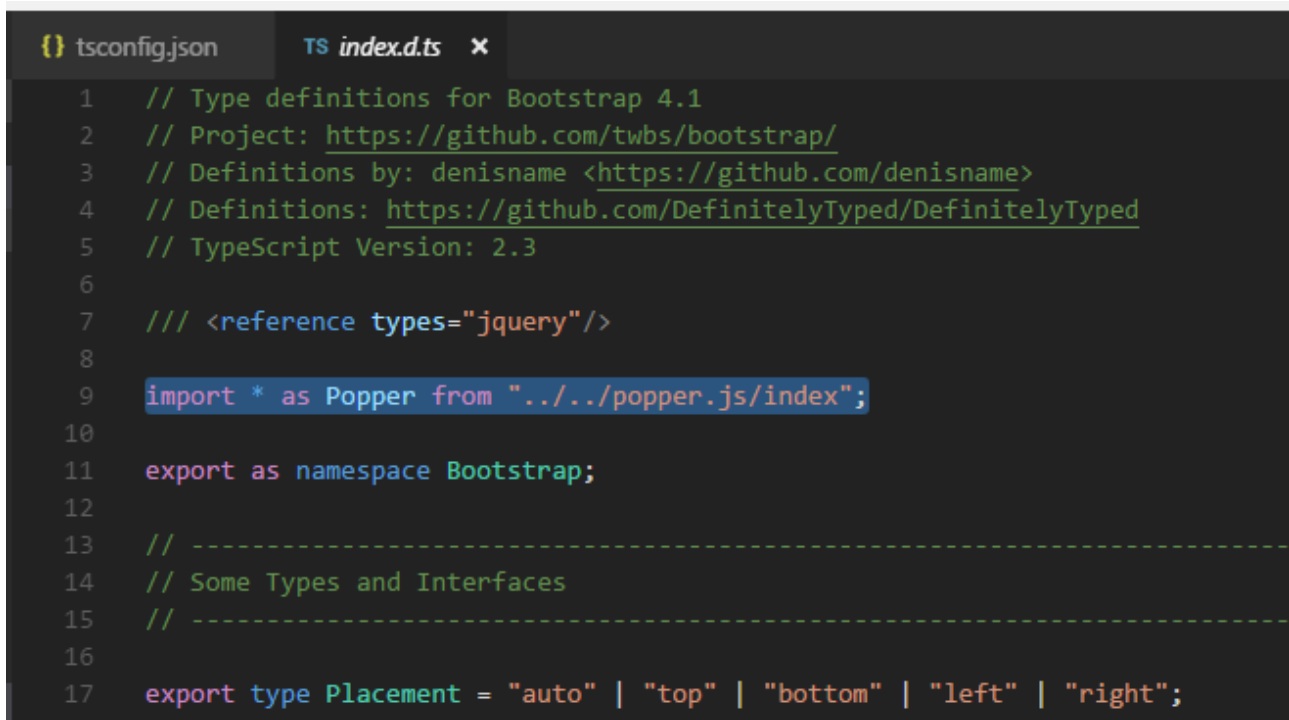
You should see the StoryScript folder open in your explorer on the left:



Great, now just three more steps before you are all set up! Open the integrated development console by pressing CONTROL and `` at the same time. Then:

1. Install all the required packages by typing '**npm install**' (enter). Wait until the installation is done before continuing.
2. You need to install a web server to be able to test and play your game. Use the command '**npm install -g lite-server**' to install lite-server.

3. Finally, correct for a bug in the Bootstrap typings. Open the file '**index.d.ts**' in the folder '**node_modules\@types\bootstrap**' from within the StoryScript folder. Change the line '**import * as Popper from "popper.js/index";**' to '**import * as Popper from "../popper.js/index";**':



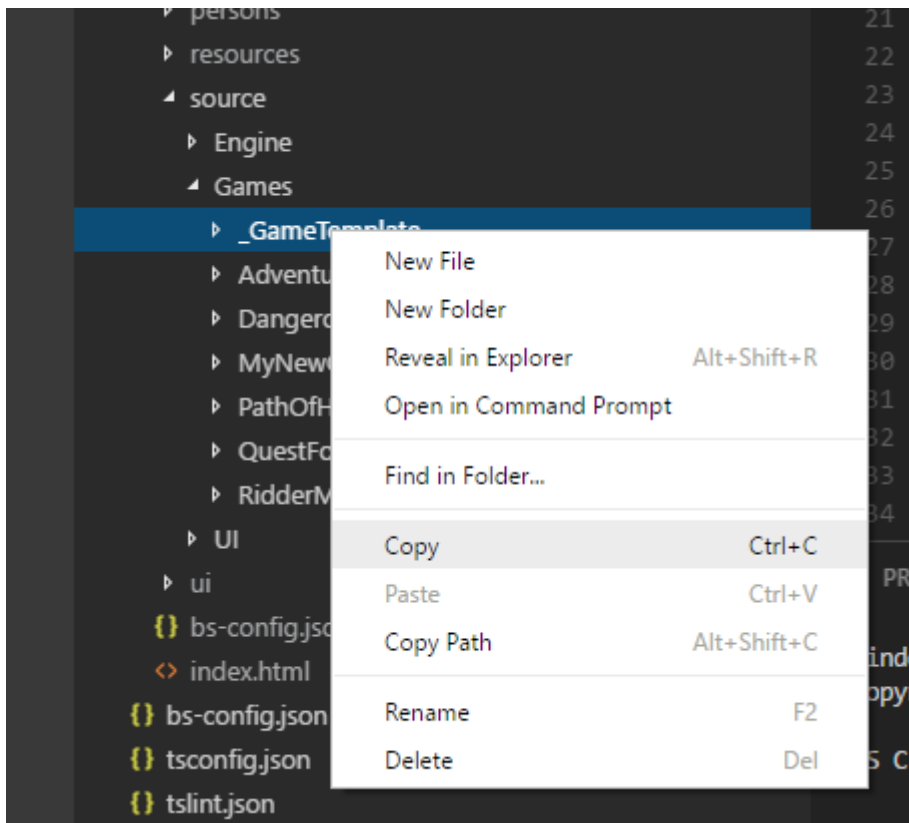
```
tsconfig.json TS index.d.ts x
1 // Type definitions for Bootstrap 4.1
2 // Project: https://github.com/twbs/bootstrap/
3 // Definitions by: denisname <https://github.com/denisname>
4 // Definitions: https://github.com/DefinitelyTyped/DefinitelyTyped
5 // TypeScript Version: 2.3
6
7 /// <reference types="jquery" />
8
9 import * as Popper from "../popper.js/index";
10
11 export as namespace Bootstrap;
12
13 // -----
14 // Some Types and Interfaces
15 // -----
16
17 export type Placement = "auto" | "top" | "bottom" | "left" | "right";
```

Now, you can finally start working on your game!

2 Creating a new, empty game

To start working on a new StoryScript game, first choose a name for it, without spaces or special characters. Let's use **MyNewGame** as an example.

Navigate to the **Games** folder, right-click the **_GameTemplate** folder and choose copy:



Right-click the **Games** folder and choose paste. Then rename the **_GameTemplate – Copy** folder that was created to your game name, so **MyNewGame**.

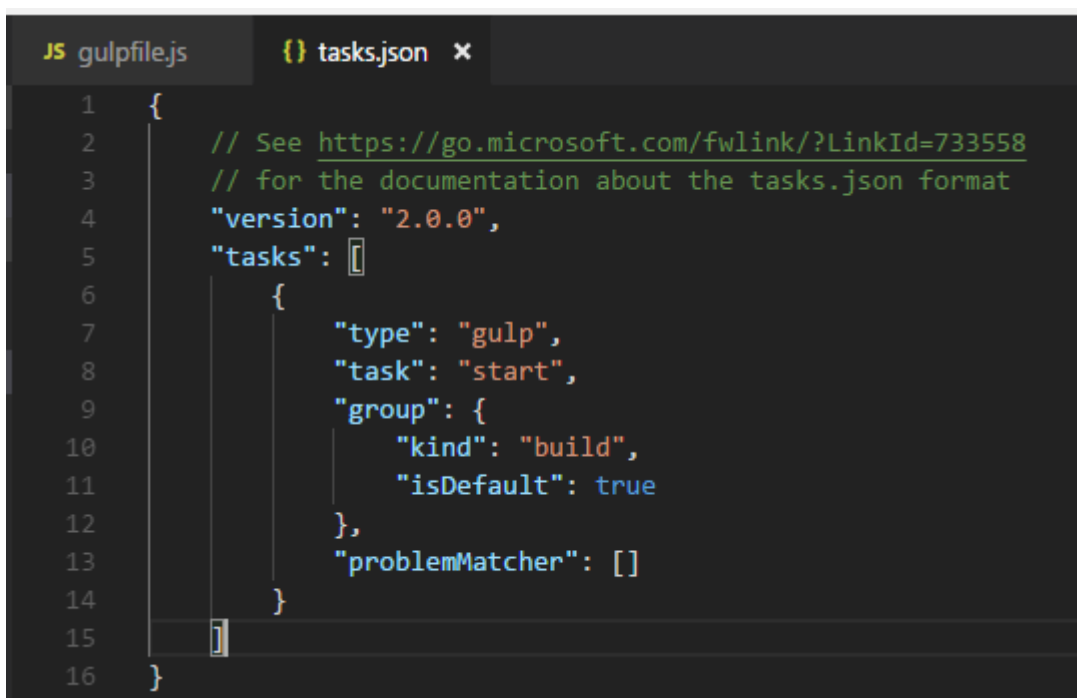
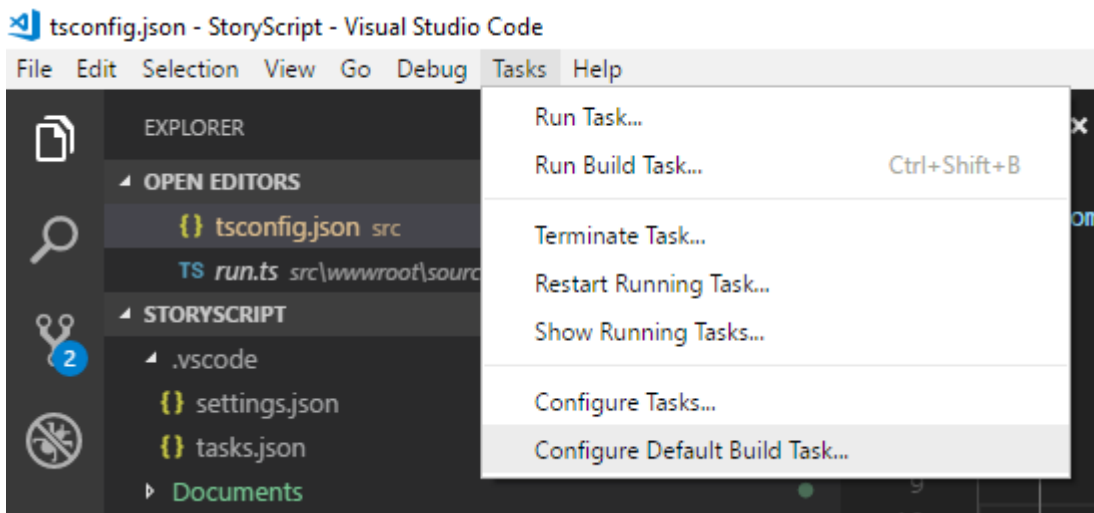
The next step is to move all files to your game namespace. Open all **.ts** files in your new folder and change the first line of all files from **'module GameTemplate'** to **'module MyNewGame'**. Note that in Visual Studio (the full version) the **'Start.ts'** file is hidden under the **'Start.html'**. Click the triangle in front of the **.html** file to display it.

In the **run.ts** file, also change the string **'GameTemplate'** to **'MyNewGame'**.

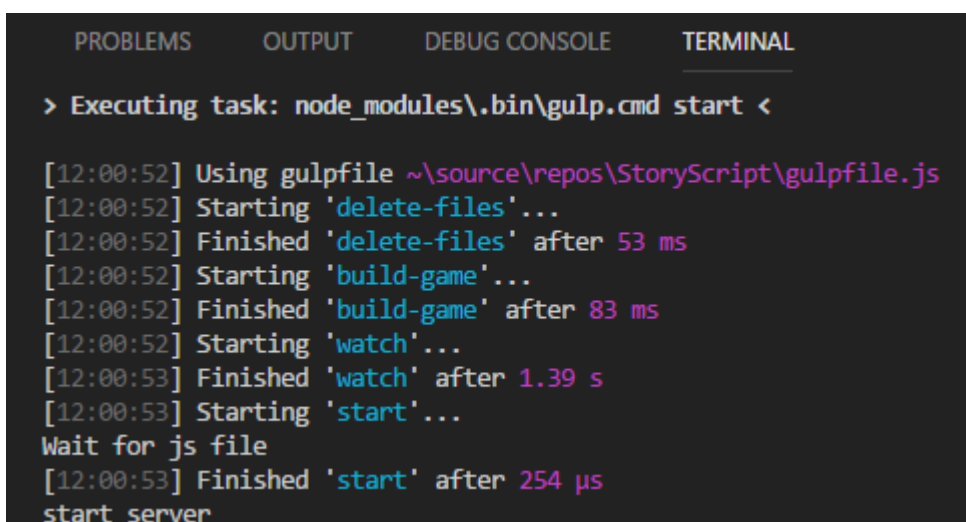
Finally, open the **tsconfig.json** file in the **src/Games** folder of the StoryScript project and change the line **'./_GameTemplate/**/*.ts'** to **'./MyNewGame/**/*.ts'**;



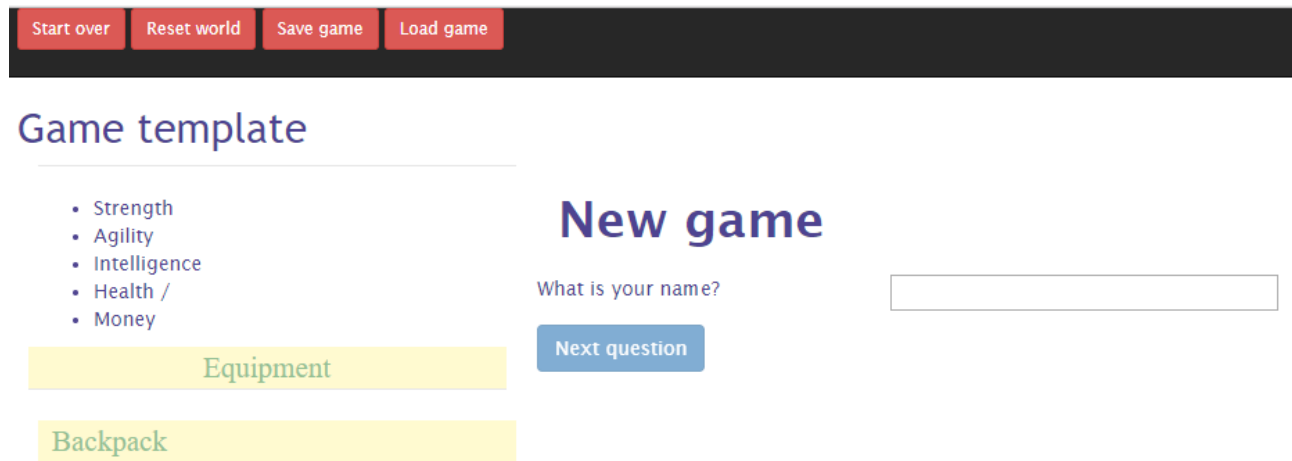
You should now be able to build and run your game. Configure the **'start'** task as your default build task using the **Configure Tasks** from the **Tasks** menu:



Then, press '**CONTROL-SHIFT-B**' to build and run the game. You should see this output in the integrated terminal:



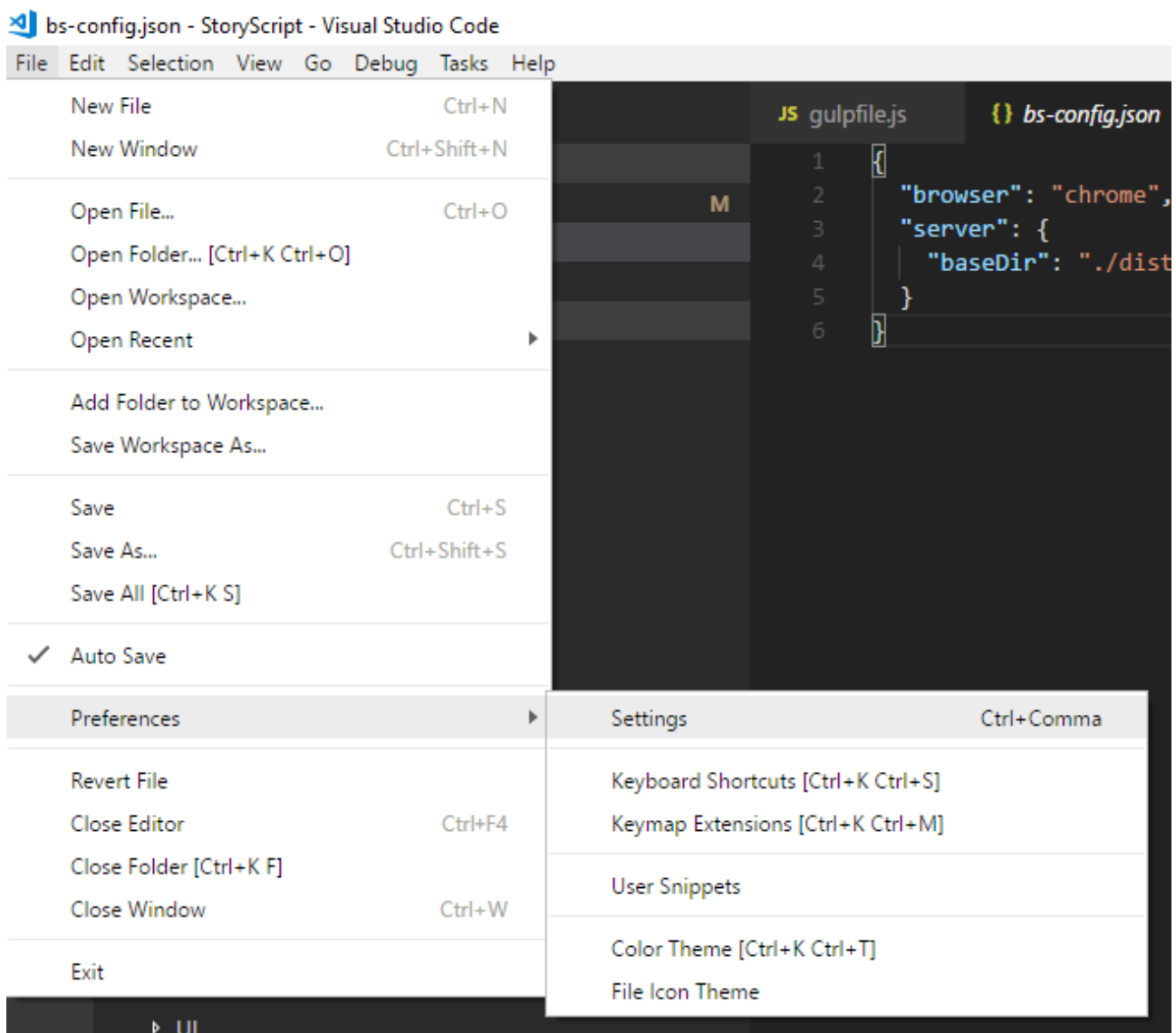
Google Chrome should launch, showing you something like this:



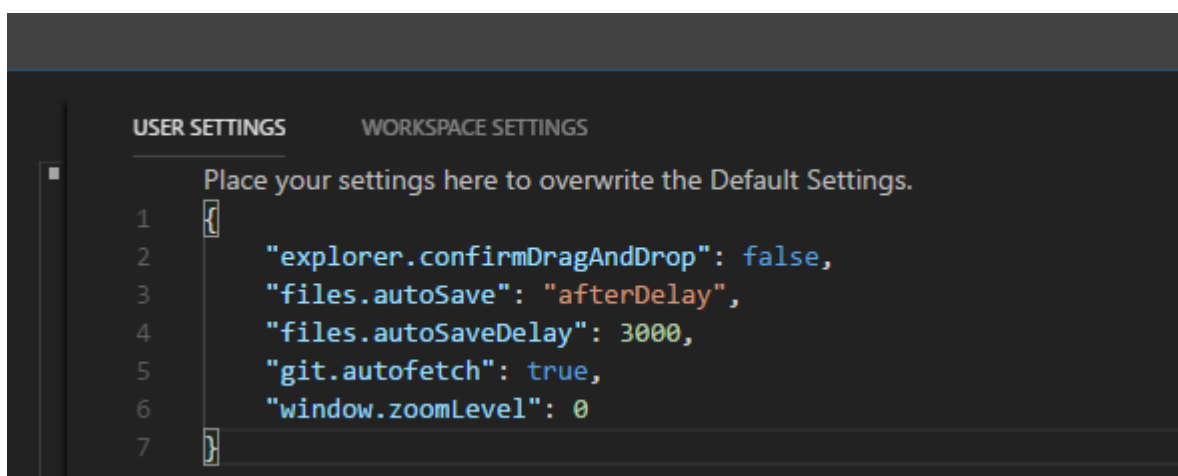
If you don't have Chrome, you can change the browser used in the '**bsconfig.json**' file in the project root folder:

```
JS gulpfile.js  {} bs-config.json x  {} tasks.json
1  {
2    "browser": "chrome",
3    "server": {
4      "baseDir": "../dist"
5    }
6  }
```

Congratulations, you are ready to begin creating your new game! Note that your browser will refresh whenever you save changes to your files. You can enable automatic saving of files via the settings menu:



Change the contents of your 'User Settings' to match the screenshot below. You can of course change the delay to suit your preference:



Now, whenever a file is added, changed or deleted, the browser will refresh showing you the latest version of **PART OF** your game. **REMEMBER** that the browser reloading will not completely reset your game. That is by design. You don't want to have your game reset and all progress lost on

each browser reload. However, you do want to be able to develop your game quickly and see the changes you make as soon as possible. That is why StoryScript has a compromise between saving game state and development speed. Amongst others, all interface texts changed, pictures added and descriptions and conversations changed will show immediately after the browser reloads. Things like new destinations and enemies added to a location will not. However, there is an easy way to load the latest version of your game world, resetting it to its pristine state. To do this, just press the 'Reset World' button in the top navigation bar:



Your character will remain at the location he was before the world was reset and will not be reset. If you want to reset your character you'll have to start over.

3 Define your hero

Next, you want to let the player create a hero with whom to play the game. To determine what such a hero looks like, open the **character.ts** file in your game's folder. You can add properties where it says so. Say we want to measure our hero's **strength**, **agility** and **intelligence**, which start at 1 and max out at 10 (we do not do anything to enforce that yet). We then add:

```
TS character.ts x
1 namespace MyNewGame {
2     export class Character implements StoryScript.ICharacter {
3         name: string = "";
4         score: number = 0;
5         currency: number = 0;
6         level: number = 1;
7         hitpoints: number = 10;
8         currentHitpoints: number = 10;
9
10        // Add character properties here.
11        strength: number = 1;
12        agility: number = 1;
13        intelligence: number = 1;
14
15        items: StoryScript.ICollection<IItem> = [];
16    }
```

Next, we need to choose the equipment slots that we will use. You can enable these slots (the list can be reviewed in the StoryScript folder, character.ts file):

head, body, hands, leftHand, leftRing, rightHand, rightRing, legs, feet.

Let's use **head, body, left** and **right hands** and **feet** only. We define this by populating the character's equipment object like this:

```

14
15     items: StoryScript.ICollection<IItem> = [];
16
17     equipment: {
18         head: IItem,
19         body: IItem,
20         leftHand: IItem,
21         rightHand: IItem,
22         feet: IItem,
23     };
24
25     constructor() {

```

Now, we want the player to make some choices when creating the hero that will determine his or her starting stats. For this we need to define the character creation process. Open the **rules.ts** file in your game's folder and find the line that says “Add the character creation steps here”. We will ask the player what name he wants to give to his character and ask two questions, showing the second question only after he answered the first. To do that, we'll create three steps (if we wanted to show both questions at the same time, we would define only two steps, one for the name and one containing both questions). The required code is listed below:

```

getCreateCharacterSheet = (): StoryScript.ICreateCharacter => {
    return {
        steps: [
            {
                attributes: [
                    {
                        question: 'What is your name?',
                        entries: [
                            {
                                attribute: 'name'
                            }
                        ]
                    }
                ]
            },
            {
                questions: [
                    {
                        question: 'As a child, you were always...',
                        entries: [
                            {
                                text: 'strong in fights',
                                value: 'strength',
                                bonus: 1
                            },
                            {
                                text: 'a fast runner',
                                value: 'agility',
                                bonus: 1
                            },
                            {
                                text: 'a curious reader',
                                value: 'intelligence',
                                bonus: 1
                            }
                        ]
                    }
                ]
            }
        ]
    },

```

```

    {
      questions: [
        {
          question: 'When time came to become an apprentice, you chose
to...',
          entries: [
            {
              text: 'become a guard',
              value: 'strength',
              bonus: 1
            },
            {
              text: 'learn about locks',
              value: 'agility',
              bonus: 1
            },
            {
              text: 'go to magic school',
              value: 'intelligence',
              bonus: 1
            }
          ]
        }
      ]
    }
  ];
}

```

Go back to your browser, which should have refreshed now. After entering a name, you should see the first question with the three options you defined:

Start over
Reset world
Save game
Load game

Game template

- Strength
- Agility
- Intelligence
- Health /
- Money

Equipment

Backpack

New game

As a child, you were always...

Next question

strong in fights
strong in fights
a fast runner
a curious reader

When you're done creating your character, you'll want to display some vital statistics in the character sheet on the left-hand side of the screen. You can specify which character attributes to list in the rules. Let's add the three attributes we defined. Health and money will be displayed by default:

```
namespace MyNewGame {
  export class Rules implements StoryScript.IRules {
    getSheetAttributes = () => {
      return [
        'strength',
        'agility',
        'intelligence'
      ];
    };

    getCreateCharacterSheet = (): StoryScript.ICreateCharacter => {
      return {

```

When you finished character creation, you should now see something like this:

Rutger

- Strength 1
- Agility 2
- Intelligence 2
- Health 10 / 10
- Money 0

Equipment

4 Personalize the interface texts

Now is a good time to change some of the interface texts you are seeing. For example, the display name of your game is still '**Game template**', and maybe we want to show '**Create your character**' instead of '**New game**'. There are three ways in which you can change the interface, and changing just the texts is easy and done via the **customTexts.ts** file in your game folder. The other ways are changing the game styling and customizing display templates, both of which are discussed later in this document.

Open your **customTexts.ts** file, place your cursor one line below where it says 'Add your custom texts here' and press **CONTROL-SPACE**. You will now see a list of all the texts that you can personalize:

```
1 namespace MyNewGame {
2   export class CustomTexts {
3     texts: StoryScript.IInterfaceTexts = {
4       // Add your custom texts here.
5     }
6   }
7 }
8 }
```

actions?

amulet?

attack?

back?

backpack?

body?

closeModal?

combatTitle?

combatWin?

combinations?

completeLevelUp?

congratulations?

(property) StoryScript.IInterfaceText
s.actions: string

Change the game name and new game texts by adding the following code:

```
namespace MyNewGame {
  export class CustomTexts {
    texts: StoryScript.IInterfaceTexts = {
      // Add your custom texts here.
      gameName: 'My new game',
      newGame: 'Create your character'
    }
  }
}
```

Your browser should now display the new texts.

5 Locations

Change a location's name and description

When we create our character, there's nothing for us to do right now. Our adventure window shows very little:

You are here

Start

Your adventure starts here!

Destinations

On the ground

Messages

First, you can change the text displayed under 'You are here' by modifying your currently only location, Start. Open the locations folder under your game folder and you will see a **Start.html** file and **start.ts** file underneath.

To change the 'Start' text, open the **start.ts** file and change the name property in the file, for example to 'Home'. The description of the location ('Your adventure starts here') is in the **Start.html** file. In that file, you can see a **<description>** tag containing the text. Let's change it to 'You are at home. Let's get started.'

Your browser should update and show:

You are here

Home

You are at home. Let's get started.

Destinations

On the ground

Messages

Multiple descriptions for varying circumstances

A nice feature is that you can have multiple descriptions in a location's .html file that you can choose from while running the game. As an example, let's make it so that the location description changes depending on the time of day, with a text for daytime and a text for night time. To do this, you need to repeat the **<description>** tag and add an attribute to the second tag that you will use to select it, for example 'night'. Our **Start.html** will then look like this:

```
<> Start.html x
1  <description name="day">
2      <p>
3          You are at home. You can hear the birds singing in the garden.
4      </p>
5  </description>
6  <description name="night">
7      <p>
8          You are at home. the frogs are croaking in the pond.
9      </p>
10 </description>
```

To actually show the night time description between 6 p.m. And 6 a.m., we need to create a selector in our location's .ts file. Modify your **start.ts** like this:

```

1 namespace MyNewGame.Locations {
2     export function Start(): ILocation {
3         return {
4             name: 'Home',
5             descriptionSelector: (game: IGame) => {
6                 var date = new Date();
7                 var hour = date.getHours();
8
9                 if (hour <= 6 || hour >= 18) {
10                     return 'night';
11                 }
12
13                 return 'day';
14             },

```

Now, depending on your local time, you see either:

You are here

Home

You are at home. You can hear the birds singing in the garden.

Or:

You are here

Home

You are at home. the frogs are croaking in the pond.

Adding new locations

Now that we are happy with our start location, let's allow the player to go somewhere. We'll add to new locations, the hero's garden behind his home and the dirt road in front of it. The easiest way to do so is just to copy **Start.html** twice and rename the copies. The result should look like this:

```

└─ locations
  ├── <> DirtRoad.html
  ├── TS DirtRoad.ts
  ├── <> Garden.html
  ├── TS Garden.ts
  ├── <> Start.html
  └── TS start.ts

```

Open both the new .ts files, change the name properties and remove the description selectors. Also change the function names to match the file names (not required, you can select another name as long as it does not have spaces or special characters). Your **Garden.ts** should look like this:

```
Start.html TS Garden.ts x
1 namespace MyNewGame.Locations {
2     export function Garden(): StoryScript.ILocation {
3         return {
4             name: 'Garden'
5         }
6     }
7 }
```

And your **DirtRoad.ts** similar, with a different name and function name of course.

Next, change your .html files to add some proper descriptions. We use just one description for now, so please remove the name attribute from the description tag. My **Garden.html** looks like this:

```
Start.html TS Garden.ts Garden.html x
1 <description>
2     <p>
3         You are in your garden. There's a little shed at the back. On the right, a little pond.
4     </p>
5 </description>
```

Linking locations

Allright, now we have three locations but no way to reach the new locations yet. To connect locations, open a location's .ts file and add a destinations property. We'll do so in the **Start.ts** file. The player should be able to go to the garden or the dirt road from the start location.

Start by adding a destinations array and a first destination object. A destination has a text (this is shown in the interface under the destinations header) and a target, the actual destination. Note that when you create the target and you typed Locations and entered the dot, visual studio will show you all your locations so you can easily pick the one you want:

```
Start.html TS Garden.ts TS start.ts
1 namespace MyNewGame.Locations {
2     export function Start(): ILocation {
3         return {
4             name: 'Home',
5             descriptionSelector: (game: IGame) => {
6                 var date = new Date();
7                 var hour = date.getHours();
8
9                 if (hour <= 6 || hour >= 18) {
10                     return 'night';
11                 }
12
13                 return 'day';
14             },
15             destinations: [
16                 {
17                     name: 'To the garden',
18                     target: Locations.
19                 }
20             ]
21         }
22     }
23 }
```

gameName

Basement
Bedroom
DirtRoad
Garden
Start

function MyNewGame.Locations.Basement
(): StoryScript.ILocation

When you added both destinations, your destinations code looks something like this:

```
15 destinations: []
16
17 {
18   name: 'To the garden',
19   target: Locations.Garden
20 },
21 {
22   name: 'Out the front door',
23   target: Locations.DirtRoad
24 }
```

When your browser reloaded, click '**Reset world**'. Then you should see this:

You are here

Home

You are at home. You can hear the birds singing in the garden.

Destinations

To the garden

Out the front door

On the ground

Messages

Clicking one of the buttons should take you to that location. Note that now you are stuck because you have not defined the navigation the other way around. Do that, click 'Start over' and walk around your three-stage world.

6 Items

Going out there without decent gear will cut any adventuring career short. Let's add some basic items for our hero to pick up at his home. Add two new files to the **items** folder in your game folder. Add **sword.ts** and **leatherBoots.ts**. They should look like this:

```
Start.html TS Garden.ts TS start.ts TS sword.ts x
1 namespace MyNewGame.Items {
2   export function Sword(): IItem {
3     return {
4       name: 'Sword',
5       damage: '3',
6       equipmentType: StoryScript.EquipmentType.RightHand
7     }
8   }
9 }
```

```
Start.html TS Garden.ts TS start.ts TS sword.ts TS leatherBoots.ts x
1 namespace MyNewGame.Items {
2   export function LeatherBoots(): IItem {
3     return {
4       name: 'Leather boots',
5       defense: 1,
6       equipmentType: StoryScript.EquipmentType.Feet
7     }
8   }
9 }
```

Ok, now that we have the items, we need to make them available. Add them both to the starting location by modifying your **Start.ts**, like so:

```
},
{
  name: 'Out the front door',
  target: Locations.DirtRoad
},
],
items: [
  Items.Sword,
  Items.LeatherBoots
],
```

Click 'Reset world'. When you go to your home, you should see:

You are here

Home

You are at home. You can hear the birds singing in the garden.

Destinations

To the garden

Back: Out the front door

On the ground

- Sword
- Leather gloves

Messages

You can pick up these items by clicking them, and you can then equip them or drop them again.

7 Events

Let's now enhance our walking around experience a bit by adding an event to our garden. Events are anything you can think of that happen once, when the player first enters a new location or leaves first leaves one. We'll do something very simple and just write a message to the log. Modify your **Garden.ts** like this:

```
return {  
  name: 'Garden',  
  destinations: [  
    {  
      name: 'Enter your home',  
      target: Locations.Start,  
    }  
  ],  
  enterEvents: [  
    (game) => {  
      game.logToActionLog('You see a squirrel running off.');    }  
  ],  
}
```

Reset the game and go to the garden. The first time you go there (and first time only), you should see the message appearing under 'Messages':

On the ground

Messages

You see a squirrel running off.

This is of course rather trivial, but events can be used for powerful stuff as you have the full game API at your disposal in the event function. Check out the API documentation once you feel comfortable enough with the StoryScript concepts described in this tutorial.

8 Enemies

Let's give the hero an opportunity to be heroic by adding a bandit to the dirt road which he can fight. First, add a new **bandit.ts** file to the enemies folder in your game folder. It should look like this:



```
1 namespace MyNewGame.Enemies {
2   export function Bandit(): IEnemy {
3     return {
4       name: 'Bandit',
5       hitpoints: 10,
6       attack: '1d6',
7       items: [
8         Items.Sword
9       ]
10    }
11  }
12 }
```

Note that I gave the bandit a sword, just like the one our hero can pick up at home. When an enemy is defeated, the items he carries are dropped on the floor in that location.

Now, we need to add the bandit to the dirt road location. Modify **DirtRoad.ts** like this:

```

1 namespace MyNewGame.Locations {
2     export function DirtRoad(): StoryScript.ILocation {
3         return {
4             name: 'Dirt road',
5             destinations: [
6                 {
7                     name: 'Enter your home',
8                     target: Locations.Start
9                 }
10            ],
11            enemies: [
12                Enemies.Bandit
13            ],

```

Reset the game. You should see the bandit as you enter the dirt road:

You are here

Dirt road

The little dirt road in front of your house leads to the main road to town. In the other direction, it ends in the woods.

Encounters

You face these foes:

- Bandit

Start combat

Messages

9 Creating a combat system

When you enter combat and click the Attack Bandit button, nothing happens. That's because there is no default combat system in StoryScript, as combat rules as well as how heroes, enemies and items are defined can vary wildly. You will have to program such a system yourself, which requires some skill with JavaScript and TypeScript.

For now, let's just add a very simple system to experiment with. Open your **rules.ts** and find the fight method, shown below:

```

fight = (game: IGame, enemy: ICompiledEnemy, retaliate?: boolean) => {
    var self = this;
    retaliate = retaliate == undefined ? true : retaliate;

    // Implement character attack here.

    if (retaliate) {
        game.currentLocation.activeEnemies.filter((enemy: ICompiledEnemy) => {
            // Implement monster attack here
        });
    }
}

```

Modify it like this:

```

fight = (game: IGame, enemy: ICompiledEnemy) => {
    var self = this;
    var damage = game.helpers.rollDice('1d6') + game.character.strength + game.helpers.calculateBonus(game.character, 'damage');
    game.logToCombatLog('You do ' + damage + ' damage to the ' + enemy.name + '!');
    enemy.hitpoints -= damage;

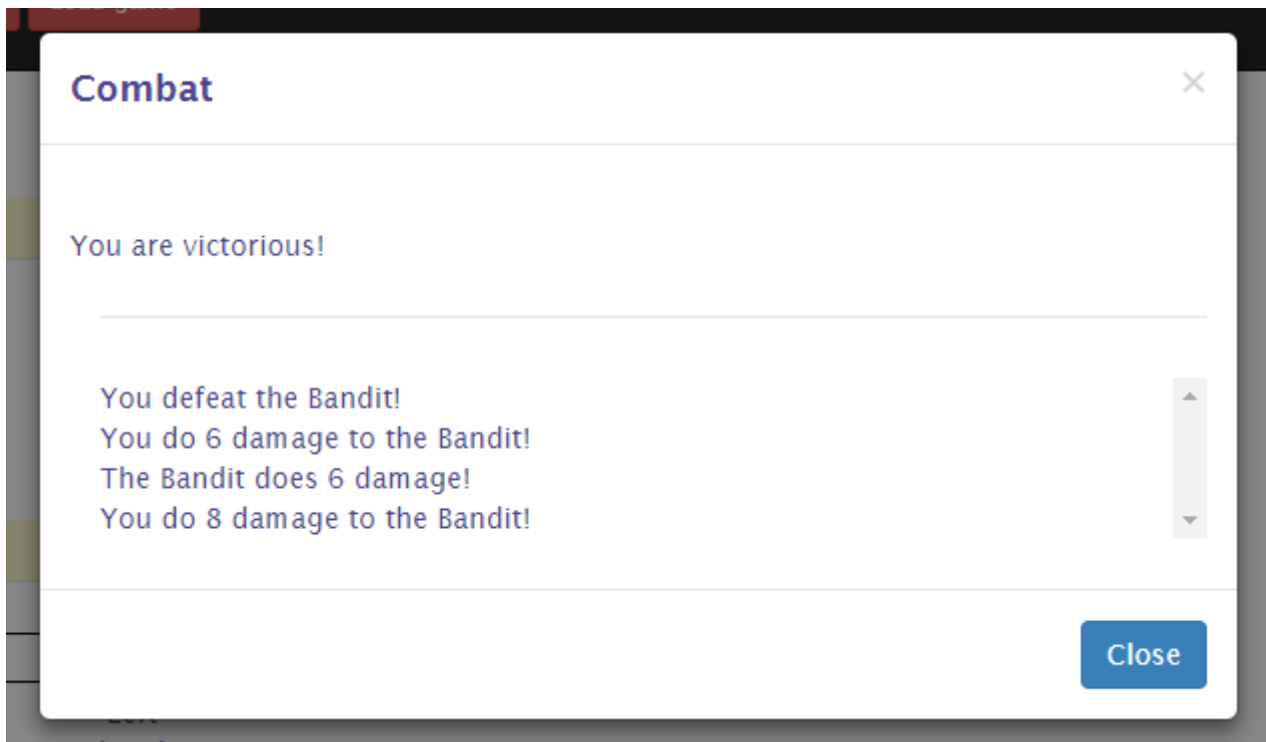
    if (enemy.hitpoints <= 0) {
        game.logToCombatLog('You defeat the ' + enemy.name + '!');
    }

    game.currentLocation.activeEnemies.filter((enemy: ICompiledEnemy) => { return enemy.hitpoints > 0; }).forEach(function (enemy) {
        var damage = game.helpers.rollDice(enemy.attack) + game.helpers.calculateBonus(enemy, 'damage');
        game.logToCombatLog('The ' + enemy.name + ' does ' + damage + ' damage!');
        game.character.currentHitpoints -= damage;
    });
}

```

This combat system is very simple. It just rolls a six-sided die for the hero and adds his strength and weapons damage bonus to the result. It writes a message to show the damage done, and subtracts the result from the enemy's hitpoints. When the enemy has 0 or less hitpoints, another message is written and the method exits, returning true for winning combat. If not, the enemy attacks the player in the same way.

Reset and click Attack bandit. You can't lose yet, so after a few clicks the bandit might be defeated and you see something like this:



When you click 'Close', notice that the Bandit's sword is now lying on the ground:

You are here

Dirt road

The little dirt road in front of your house leads to the main road to town. In the other direction, it ends in the woods.

Destinations

Enter your home

On the ground

- Sword

Messages

You might also lose, in which case you'll see something like this:

You lost...

You have failed your quest!

Your score: 0

Try again

10 Actions and CombatActions

For the game locations to become interesting, they should offer more than only enemies and events. There should be things to do, buttons to push. That's where actions come in. When you define actions, these will be available at the location after all enemies have been defeated. Examples of actions you might want to add are search, to find more items when the character is alert enough, opening chests, etc.

Let's add two actions to the garden, one to search the shed and one to look in the pond. Add this code to **Garden.ts**:

```
actions: [
  {
    text: 'Search the Shed',
    execute: (game) => {
      // Add a new destination.
      game.currentLocation.destinations.push({
        name: 'Enter the basement',
        target: Locations.Basement
      });
    },
  },
  {
    text: 'Look in the pond',
    execute: (game: IGame) => {
      game.logToLocationLog('The pond is shallow. There are frogs and snails in there, but nothing of interest.');
```

Note that I had to add the Basement location to be able to add the new destination. If you are following along, add that location too.

Reset and you should see the following in the garden:

You are here

Garden

You are in your garden. There's a little shed at the back. On the right, a little pond.

Actions

Search the Shed

Look in the pond

Destinations

Back: Enter your home

When you press the Search button, a new destination will be added and the Search button removed. Look in the pond will remove the action and write to the location description. If you want to keep the button when it is clicked, have the action's execute method return *true* (check out the API).

Combat actions are like actions but are available during combat only, when regular actions are not available. Let's allow the hero to run back home when he is afraid to stand up to the bandit. Add this code to **DirtRoad.ts**:

```
combatActions: [  
  {  
    text: 'Run back inside',  
    execute: (game: IGame) => {  
      game.changeLocation('Start');  
      game.logToActionLog(`You storm back into your house and slam the door behind  
you. You where lucky... this time!`);  
      return true;  
    }  
  }  
]
```

When you go out the door to face the bandit, you should see the option to run back inside. When you take it, you will be told you ran away:

You are here

Home

You are at home. the frogs are croaking in the pond.

Destinations

To the garden

Back: Out the front door

On the ground

- Sword
- Leather boots

Messages

You storm back into your house and slam the door behind you. You where lucky... this time!

11 Doors, gates, rivers and other barriers

To make the world more interesting and locations less easy to reach, you can add barriers that impede the hero's progress. Barriers can be anything from doors, locked gates, rivers, chasms or even leaving the atmosphere of a planet.

Barriers are added to destinations, so that it is not simply a matter of moving from one location to the next by pressing a button anymore. As an example, let's add a trap door as a barrier between the garden and the shed basement. Let's leave it unlocked for now, we'll add a required key later.

Open the **Garden.ts** file and add the following barrier code to the Basement destination:

```
barrier: {
  name: 'Wooden trap door',
  actions: [
    {
      name: 'Inspect',
      action: (game: IGame) => {
        game.logToLocationLog('The trap door looks old but still strong due to
steel reinforcements. It is locked.');
```

```
        game.logToLocationLog('You open the trap door. A wooden staircase
leads down into the darkness.');
```

```
    })
  }
]
}
```

When you run your game and search the shed, the Basement destination should be greyed out because it is blocked by the trap door:

Garden

You are in your garden. There's a little shed at the back. On the right, a little pond.

Actions

Look in the pond

Destinations

Wooden trap door

Inspect ▼

Back: Enter your home

Enter the basement

On the ground

Clicking on the trap door will trigger the action selected in the dropdown. When you leave it at 'Inspect', a message will be written to the location description and the Inspect action will be removed. When the Open action is chosen, the barrier will be removed, including any actions remaining.

Notice that one of the barrier actions uses a StoryScript action, the Open action. You can either define your own method to handle a barrier action or use a default one. This is a far more general concept that will be explained in a bit more detail in chapter 15 on the API. For now, notice that you use a StoryScript action and pass it a method that is to be executed (in this case) AFTER performing the default action of opening the door (removing the barrier).

After opening the trap door, the barrier is gone and you can access the basement destination:

You are here

Garden

You are in your garden. There's a little shed at the back. On the right, a little pond.

The trap door looks old but still strong due to steel reinforcements. It is not locked.
You open the trap door. A wooden staircase leads down into the darkness.

Actions

Look in the pond

Destinations

Enter your home

Enter the basement

Right. Barriers really make the world more interesting and interactive, but we can add some excitement by requiring the player to acquire the means to pass the barrier before letting him through. In the case of the trap door, this is a key, but you could also require a boat to pass a river or a rocket to travel to another planet, to name just a few possibilities. Let's see how to add that trap door key.

First, add a new item and call it Basement Key (file `basementKey.ts`):

```
TS Garden.ts TS basementKey.ts TS bandits.ts TS DirtRoad.ts TS trade.ts
1 namespace MyNewGame.Items {
2   export function BasementKey(): StoryScript.IKey {
3     return {
4       name: 'Basement key',
5       open: {
6         name: 'Open',
7         action: StoryScript.Actions.OpenWithKey((game: IGame, destination: StoryScript.IDestination) => {
8           game.logToLocationLog('You open the trap door. A wooden staircase leads down into the darkness.');
```

Notice that we again use a standard StoryScript action, `OpenWithKey`. We pass it the same callback we used for the open action on the barrier.

Now, remove the Open action on the trap door. Instead, define the key required. Replace the barrier code with:

```
barrier: {
  key: Items.BasementKey,
  name: 'Wooden trap door',
  actions: [
    {
      name: 'Inspect',
```

```

        action: (game: IGame) => {
            game.logToLocationLog('The trap door looks old but still strong due to
steel reinforcements. It is locked.');
```

```

        }
    }
}
]
}
```

A note on actions added during runtime

In the example above, I added an inspect action at run-time. Because of the way StoryScript handles saving the game state, please be aware that actions added during run-time will be treated slightly different from actions that are present at design-time. Make sure that any actions you add once the game is running are self-contained, meaning they should only reference arguments passed into them or use variables defined in them, and not anything else!

When you run the game, only the inspect action is available on the trap door until you have the basement key in your possession. Let's bring a few things together and give the key to the bandit we added to the dirt road location, so the player will have to defeat him in order to enter the basement:



The screenshot shows a code editor with three tabs: 'TS Garden.ts', 'TS bandit.ts' (which is active), and 'TS DirtRoad.ts'. The code in 'TS bandit.ts' defines a namespace 'MyNewGame.Enemies' and an exported function 'Bandit(): IEnemy'. The function returns an object with the following properties: 'name' set to 'Bandit', 'hitpoints' set to 10, 'attack' set to '1d6', and 'items' set to an array containing 'Items.Sword' and 'Items.BasementKey'. The code is as follows:

```

1  namespace MyNewGame.Enemies {
2      export function Bandit(): IEnemy {
3          return {
4              name: 'Bandit',
5              hitpoints: 10,
6              attack: '1d6',
7              items: [
8                  Items.Sword,
9                  Items.BasementKey
10             ]
11          }
12      }
13  }
```

Play the game, defeat the bandit, pick up the key he drops and go to the garden. With the key in your hand, you should now be able to open the trap door again:

- Agility 1
- Intelligence 1
- Strength 3
- Health 6 / 10
- Money 5

Equipment

Head		
Right hand	Body	Left hand
Sword		
Feet		
Leather boots		

Backpack

- Basement key

Drop

You are here

Garden

You are in your garden. There's a little shed at the back

Actions

Look in the pond

Destinations

Wooden trap door

Inspect

Inspect

Open

Back: Enter your home

On the ground

When you open the door, you get to keep the key. If you want to create a use-once key, modify the key like this:

```

TS createCharacter.ts TS Garden.ts TS basementKey.ts x TS DirtRoad.ts TS trade.ts
1 namespace MyNewGame.Items {
2   export function BasementKey(): StoryScript.IKey {
3     return {
4       name: 'Basement key',
5       keepAfterUse: false,
6       open: {
7         name: 'Open',
8         action: StoryScript.Actions.OpenWithKey((game: IGame, destination: StoryScript.IDestination) => {
9           game.logToLocationLog('You open the trap door. A wooden staircase leads down into the darkness.');

```

12 Persons

Not everything moving you meet along the way needs to be hostile to your character. You can also add persons, people (or something else) that you can talk to and/or trade with. You might anger these persons, at which point they can become enemies if you allow this in your game.

We'll add a friend to the game, who is present in your living room. Create a new folder 'persons' under your game folder and add two files to it, friend.html and friend.ts. The html file is very important for persons, as you'll define the conversations you have with them here.

Add this code to the friend.ts file:

```
TS Garden.ts    TS basementKey.ts    TS Friend.ts  X
1  namespace MyNewGame.Persons {
2      export function Friend(): IPerson {
3          return {
4              name: 'Joe',
5              hitpoints: 10,
6              attack: '1d6',
7              currency: 10,
8              conversation: {
9              }
10         }
11     }
12 }
```

Although the conversation property is now an empty object, it is important that it is there nonetheless as the presence of this property will determine whether or not the engine will try to load the conversation file.

Now that we created the friend, add him to the living room by adding him in the start.ts file:

```
TS Garden.ts    TS basementKey.ts    TS Friend.ts    TS start.ts  X
1  namespace MyNewGame.Locations {
2      export function Start(): ILocation {
3          return {
4              name: 'Home',
5              descriptionSelector: (game: IGame) => { ...
14          },
15          destinations: [ ...
28          ],
29          persons: [
30              Persons.Friend
31          ]
32      }
33  }
34 }
```

Build and run the game. In the living room, you should now see your friend:

Encounters

Talk to Joe

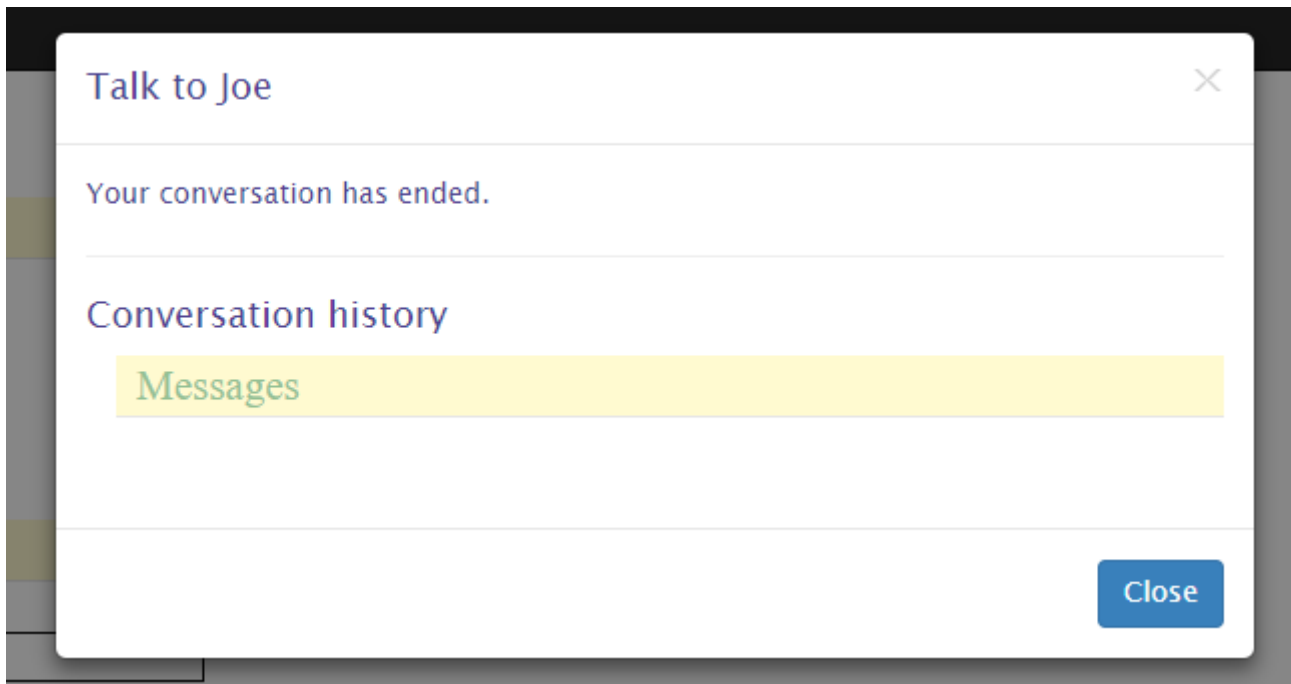
Attack Joe!

You are here

Home

If you don't want to be able to attack Joe, set the `canAttack` property to `false`. The attack button will then disappear.

Try talk to Joe. As you haven't given him any lines yet, he has not much to say:



Making Joe a bit more of a conversationalist is the topic of the next chapter. You can interact with persons in more ways, by trading with them or doing assignments (quests) for them. These will be covered in later chapters as well.

13 Conversations

So we want to be able to chat with Joe a bit. His lines and the replies the player character has available are specified in the `friend.html` file. Let's start with something simple and add this code to the file:

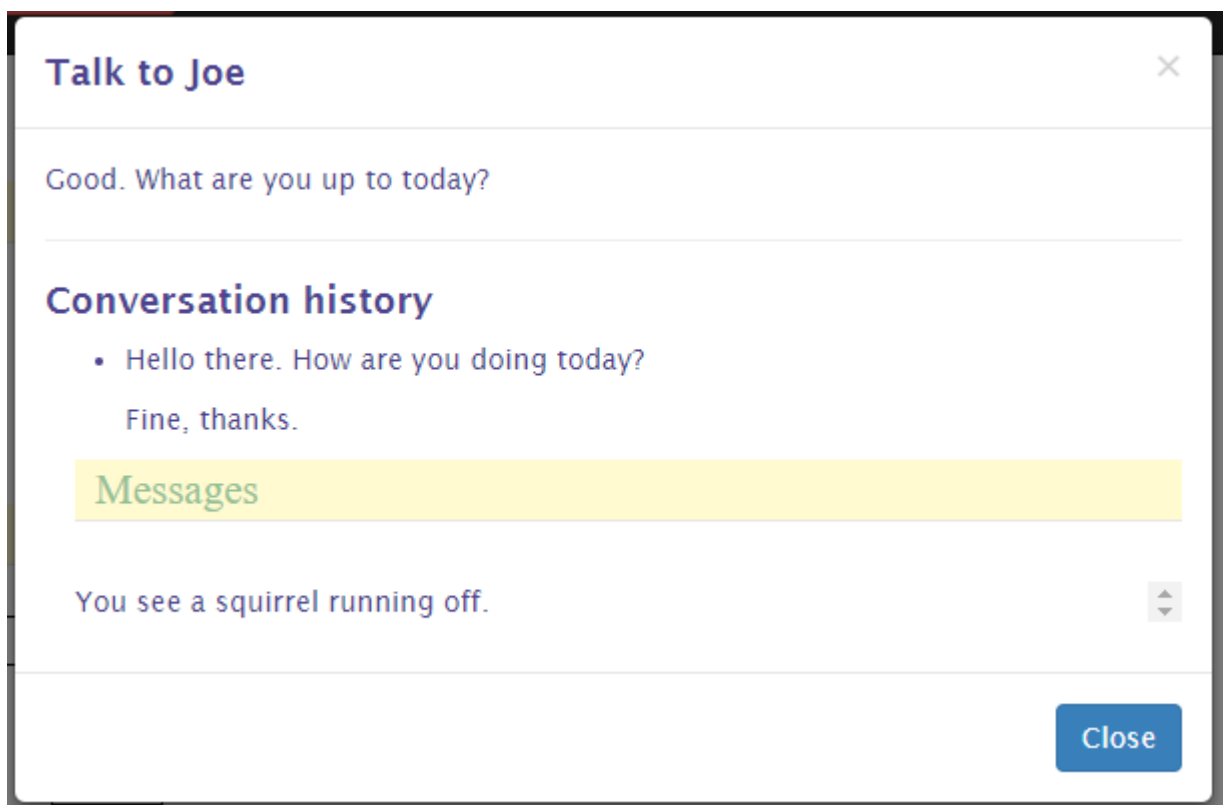
```
TS Garden.ts    TS basementKey.ts    TS Friend.ts    <> Friend.h
1  <conversation>
2  |   <node name="hello">
3  |     <p>
4  |       Hello there. How are you doing today?
5  |     </p>
6  |     <replies>
7  |       <reply node="fine">
8  |         Fine, thanks.
9  |       </reply>
10 |     </replies>
11 |   </node>
12 |   <node name="fine">
13 |     <p>
14 |       Good. What are you up to today?
15 |     </p>
16 |   </node>
17 </conversation>
```


A conversation is made up of nodes. Within a node, you can use all the HTML you want to create a colourful discussion. This html are the lines of the person. You create replies that the player character can use as a response by adding the special `<replies>` tag and adding child `<reply>` elements for each available reply.

You can see that there are two nodes in this conversation, and they both have a name. This is very important, as that name is used to link the nodes to each other to create a conversation flow. As you first speak to a person, the engine has a number of ways to determine the node to start with. As we have specified nothing special, the first node, named 'hello', will be used.

So, when you first talk to Joe, he'll ask how you are doing. You can see there is just one reply available right now, saying 'Fine, thanks'. Also, you can see that this reply has an attribute called 'node'. This is used to move to a new node when using that reply. You can see that the name of the node to go to is specified, 'fine'. So selecting this reply will make Joe go on with his lines in the 'fine' node.

Talk to Joe again, choosing the only reply options available right now. The conversation should go like this:



You see that the words exchanged are logged, so you can always review what's been said in case you forget.

You can add in a reply that is available to everything a person has to say by specifying a default response, which will allow you to end the conversation a bit less abruptly:

```
TS Friend.ts  <> Friend.html x  TS conversation.ts
1  <conversation>
2    <default-reply>
3      Never mind. See you later.
4    </default-reply>
5    <node name="hello">
```

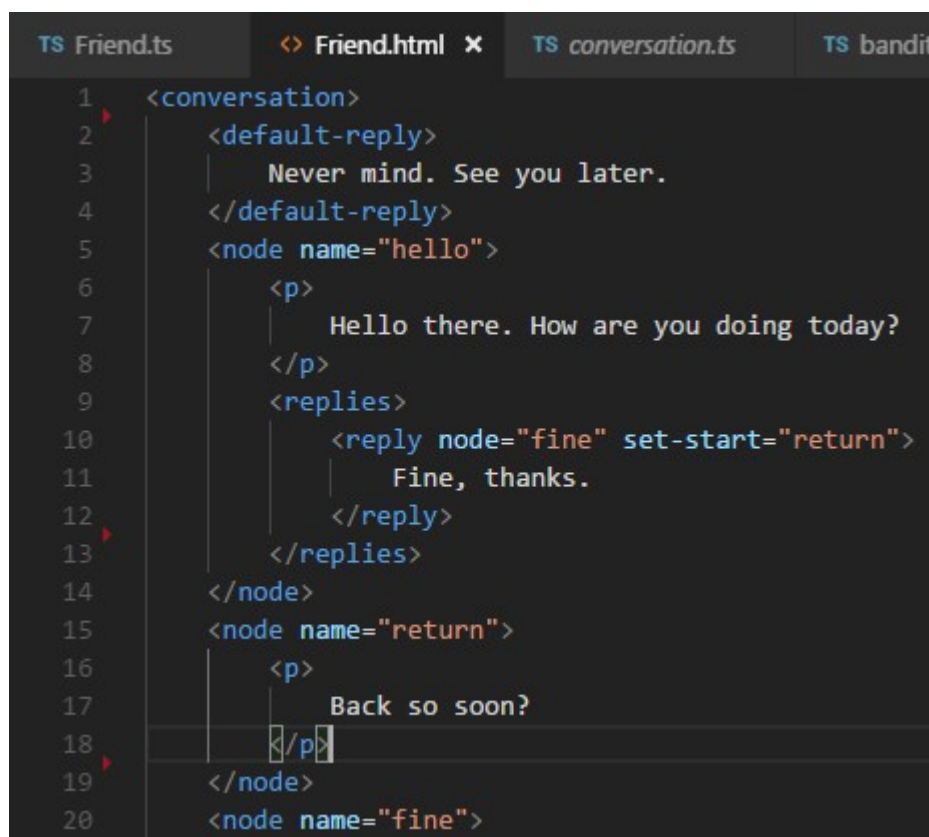
Great, some interactivity! You can expand on this conversation by adding additional nodes and replies linking to these nodes to get a basic conversation going. That's nice, but it wouldn't be great if the conversation is actually influenced by the things going on in the world around you?

Well, of course we have some options to do this (otherwise it would not make for much of an interactive story). They are:

- Setting a new start node, so the conversation does not start all over again when you re-visit your friend.
- Making replies conditionally available. For example, you could require the character to be witty enough in order to make some remarks.
- Triggering an action when you reply in a certain way.
- Linking your replies to quests.

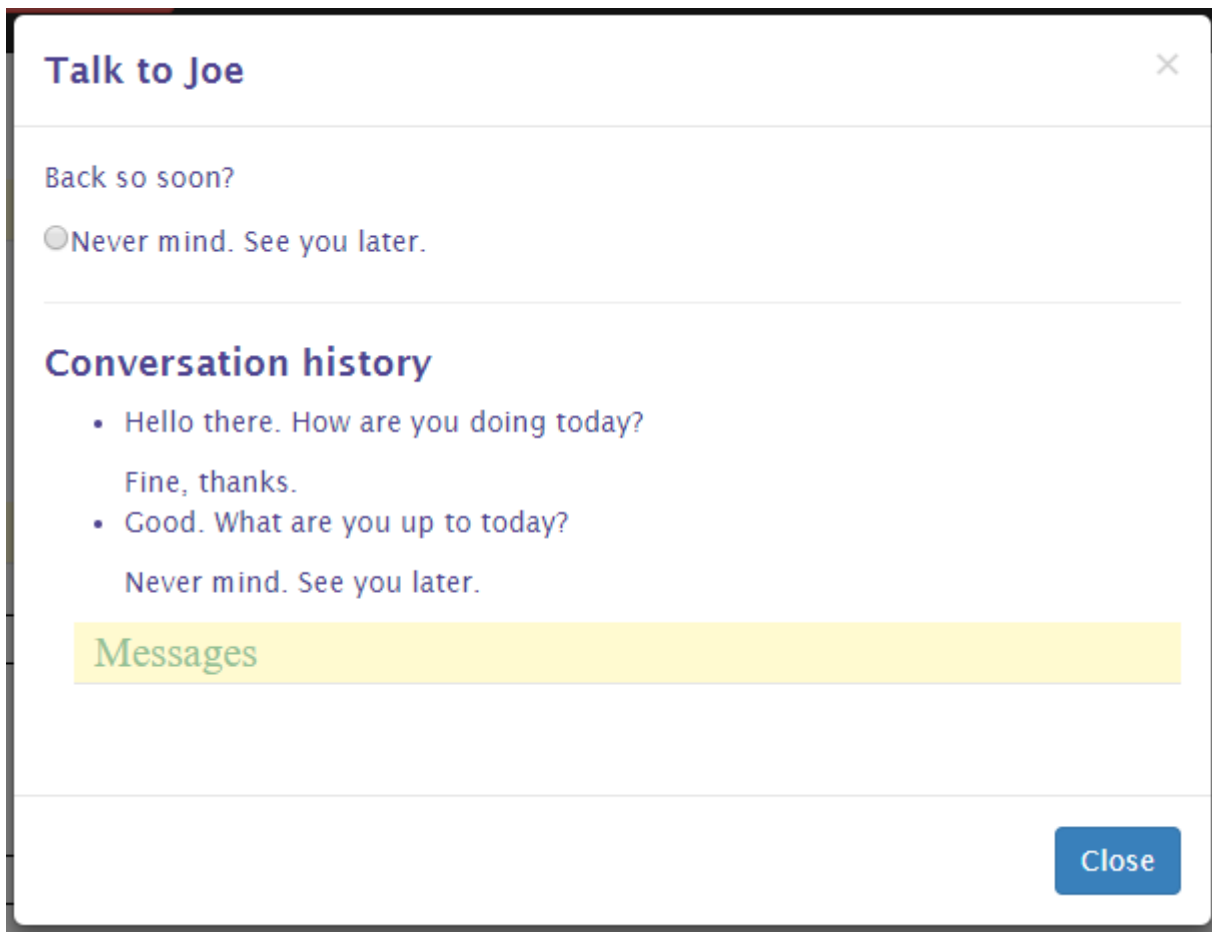
How quests work with conversations is covered in the chapter on quests. Here, we'll go on to look at the first three options.

Let's first change what Joe has to say when we talk to him again to make him a bit less of an automaton. We do this by adding the 'set-start' attribute to the first reply and adding a new node for his new lines like this:



```
1 <conversation>
2   <default-reply>
3     Never mind. See you later.
4   </default-reply>
5   <node name="hello">
6     <p>
7       Hello there. How are you doing today?
8     </p>
9     <replies>
10      <reply node="fine" set-start="return">
11        Fine, thanks.
12      </reply>
13    </replies>
14  </node>
15  <node name="return">
16    <p>
17      Back so soon?
18    </p>
19  </node>
20  <node name="fine">
```

When you now return to Joe, he should have something new in store:



Conditionally available replies

Having replies available only when certain conditions are met is when conversations are getting more interesting and more attuned to what goes on in the game world. For this, you can set the 'requires' attribute on a reply node. You can require a number of things:

- That the player character meet an attribute threshold, e.g. he should be smart enough to talk about relativity. The syntax for this is: **requires="[attribute]=[value]"**, e.g. **requires="intelligence=3"**.
- That the player has a special item with him. Use **requires="item=[itemId]"**, e.g. **requires="item=sword"** for this.
- That the player visited a certain location: **requires="location=[locationId]"**, e.g. **requires="location=garden"**.
- That the player has started, meets the requirements to complete or completed a quest. See the chapter on quests for using these requirements.

We'll add an example of the first three requirements to our conversation:


```

<p>...</p>
<replies>
  <reply node="fine" set-start="return">
    Fine, thanks.
  </reply>
  <reply node="workout" requires="strength=3">
    Great. I have just completed my workout.
  </reply>
</replies>
</node>
<node name="return">
  <p>...</p>
  <replies>
    <reply node="garden" requires="location=garden">
      I just walked through the garden.
    </reply>
    <reply node="key" requires="item=basementkey">
      I found a key.
    </reply>
  </replies>
</node>
<node>...</node>
<node name="workout">
  <p>
    I can see that.
  </p>
</node>
<node name="garden">
  <p>
    Did you see the hedgehog?
  </p>
</node>
<node name="key">
  <p>
    Hm. I don't know what lock that's for.
  </p>
</node>

```

For your character to meet the strength requirement, you should select the first answer to both questions when creating a character. Go to the garden to talk to Joe about its inhabitants.

Try these requirements by building and running the game. Replies that you do not yet qualify for should be hidden from you. If you want them to show but unselectable, set the flag on the conversation in the .ts file like this:

```

conversation: {
  showUnavailableReplies: true
}

```

Triggering actions on replies

The last option of conversations to talk about in this chapter is triggering actions on selecting a specific reply. For this to work, you need to use the 'trigger' attribute with the name of the action to trigger on the reply in the .html file. The actual action you specify in the action collection on the conversation element in the .ts file.

Let's create an example. There is a hedgehog in the garden, but unless Joe told you about it you will

not see it. To make this work, we add the trigger first in the .html file:

```
<conversation>
  <default-reply>...</default-reply>
  <node>...</node>
  <node>...</node>
  <node>...</node>
  <node>...</node>
  <node name="garden">
    <p>
      Did you see the hedgehog?
    </p>
    <replies>
      <reply trigger="addHedgehog">
        No I didn't. I'll pay attention next time.
      </reply>
    </replies>
  </node>
</node>...</node>
```

Second, we specify what the addHedgehog function does in the .ts file:

```
conversation: {
  actions: {
    'addHedgehog': (game, person) => {
      var garden = game.locations.get(Locations.Garden);
      garden.hasVisited = false;
      garden.events = [];

      garden.events.push((game: IGame) => {
        game.logToLocationLog('Ah! There is the hedgehog Joe was talking about.');
```

(method) StoryScript.IGame.logToLocationLog(message: string): void

Note that I use a few tricks here to make this work. Events are run the first time the player visits a location, so I reset the hasVisited flag on the Garden location first. Then I clear the events list, because the squirrel ran off the first time you visited the garden and it has not returned yet. Then, I add the new event to the Garden, writing a simple message to the location log.

Now, when you visit the garden, talk to Joe about it and return to the garden, you should see the hedgehog.

14 Trade and storage

As you build a world to explore and interact with, adding item stores is something you'll want to do sooner or later. Add a locker here, a chest there, and a store in the centre of a small village. Both storage and trade in StoryScript are handled by trade, where storage is a special form of trading where prices are zero and items can thus be moved to and from storage without changing anything else.

Trade can be added in two forms: either on a person (see chapter 12 on persons) or on a location. The syntax is the same, and the way it works as well but for a few small differences.

As an example, we'll add a personal closet to the bedroom, which has some items you can pick up there. You can also put items back in. This will demonstrate the basics of trading.

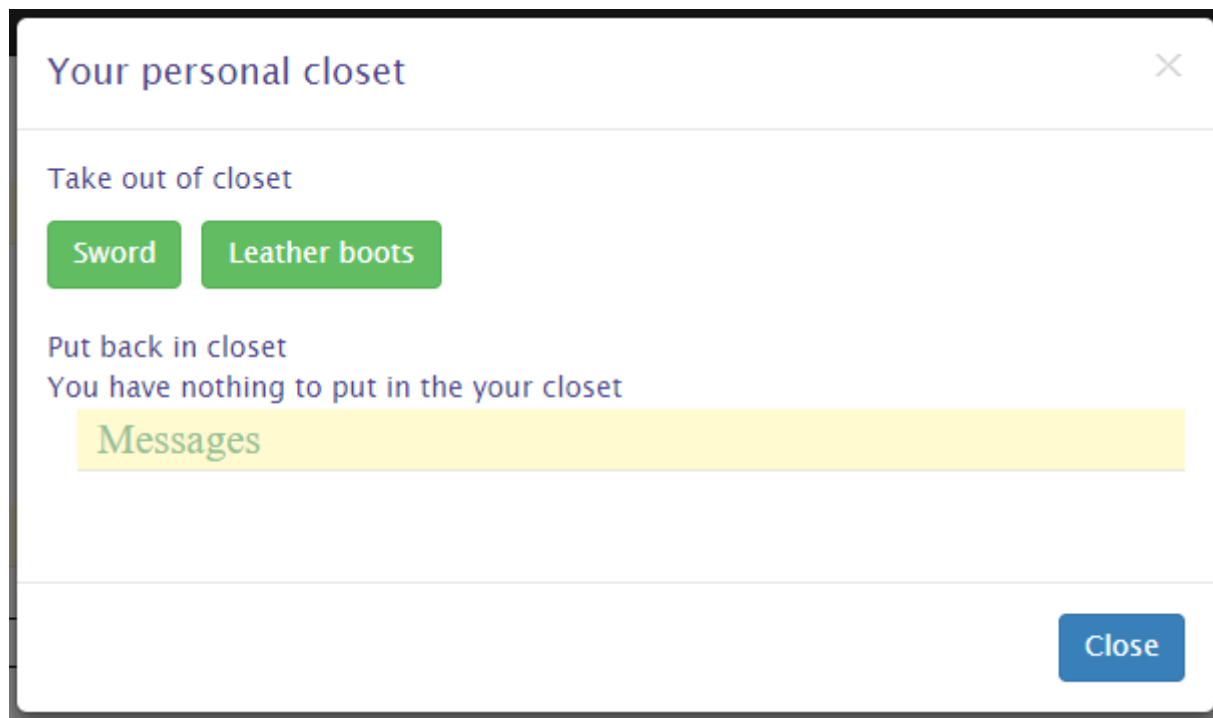
Open the bedroom.ts file and add the following code:


```

    },
    trade: {
      title: 'Your personal closet',
      description: 'Do you want to take something out of your closet or put it back in?',
      buy: {
        description: 'Take out of closet',
        emptyText: 'The closet is empty',
        itemSelector: (game: IGame, item: IItem) => {
          return item.value != undefined;
        },
        maxItems: 5,
        priceModifier: 0
      },
      sell: {
        description: 'Put back in closet',
        emptyText: 'You have nothing to put in the your closet',
        itemSelector: (game: IGame, item: IItem) => {
          return item.value != undefined;
        },
        maxItems: 5,
        priceModifier: (game: IGame) => {
          return 0;
        }
      }
    }
  },

```

Build the game and run it. Go to the bedroom. You should see a button with the title specified in the trade code, and when you press it you should see the trade screen:



Ok, let's explore the options that are not obvious. First, you have both a buy and a sell side to the trade. This is defined from the perspective of the user, so the sell part is about the player 'selling' stuff to the trade object (in this case just putting his gear in the closet). Apart from the maxItems property, which determines the number of items available for buying or selling during one visit, Both buy and sell have two important properties:

- itemSelector. This is a function that returns the items that can be bought from the trader or sold to him, her or it. In this example, the items available for taking from the closet are all items in the game that have a value property. That's why the sword and the leather boots show up and not the basement key. The buy item selector by default will select from all the

items you defined for your game. The sell item selector selects items only from your character's inventory, and will not include equipped items.

- **PriceModifier:** this can either be a number or a function that returns a number that will be multiplied by the item's value. For example, if you want the player to pay twice the value of an item to a trader in order to buy it, the number should be 2. In this example, the number on both sides is 0 because we want to put things in and take them out of the closet without paying any money.

Apart from a storage object or a store, you can also put the trade code on a person in order to be able to trade with that person. Let's enable trading with Joe by adding some code to the friend.js file:

```
    ],
    currency: 10,
    trade: {
      buy: {
        description: 'I\'m willing to part with these items...',
        emptyText: 'I have nothing left to sell to you...',
        itemSelector: (game: IGame, item: IItem) => {
          return item.value != undefined;
        },
        maxItems: 5
      },
      sell: {
        description: 'These items look good, I\'d like to buy them from you',
        emptyText: 'You have nothing left that I\'m interested in',
        itemSelector: (game: IGame, item: IItem) => {
          return item.value != undefined;
        },
        maxItems: 5
      }
    },
    conversation: {
      actions: [
```

As you can see, I omitted the priceModifier here. That means Joe will buy and sell items for precisely their value.

Build and run, now you should be able to trade with Joe:

Encounters

Talk to Joe

Trade with Joe

Attack Joe!

Some last things of note on trading are:

- **currency.** This will tell how much currency the trader has available for trading. **NOTE:** When trading with a person, the person's currency value is used. Traders can buy only as long as they have the money to pay for items that you have to offer. When you buy from them, the money you pay them will replenish their coffers.
- **InitCollection:** this is a function that is called before creating the list of items the trader has for sale. It returns true or false. If true is returned, the list of items for sale is refreshed. If

not, the items that were on sale earlier will stay. If, for example, you want the trader's stock to be replenished based on some event, use this function to check whether the event has occurred and then return true for the list to be reinitialized.

- `ownItemsOnly`: this flag determines whether the `itemSelector` function will be applied to the items available in the game or the items in the trader's inventory. This way, you can allow the trader to sell only from his own gear and nothing else.

15 **Quests**

In any kind of story, the main character has to run errands or do favours. In StoryScript, these can be created as quests.

Let's add a little something we can do for Joe. Joe has misplaced his personal journal and he would really like to have it back. You can help him find it. Create a new folder 'quests' in your game's folder and add a file called `journal.ts` to it. Add this code:


```

script (tsconfig project)
- {} MyNewGame.Quests

module MyNewGame.Quests {
  export function Journal(): StoryScript.IQuest {
    return {
      name: "Find Joe's journal",
      status: (game, quest, done) => {
        return 'Jou have ' + (done ? '' : 'not ') + 'found Joe\'s journal' + (done ? '!' : ' yet. ');
      },
      start: (game, quest, person) => {
      },
      checkDone: (game, quest) => {
        return quest.completed || game.character.items.get(Items.Journal) != null;
      },
      complete: (game, quest, person) => {
        var ring = game.character.items.get(Items.Journal);
        game.character.items.remove(ring);
        game.character.currency += 5;
      }
    }
  }
}

```

You can see that there are a number of elements to this. Apart from the quest name, there is an action you can run as soon as the quest starts. It is not used in this example. Further, there is a status property, which can be either a static text or a function returning some text, which tells what the current status of the quest is. Here, it'll format a message telling you whether or not you found Joe's journal.

Next, there is a function called to check whether the quest requirements are met called checkDone. It should return either true or false, depending on whether the player meets the quest requirements or not.

Finally, the complete function will trigger when the quest is completed and the reward claimed. Joe will give the player some cash when the journal is returned to him.

For this quest, I added a really simple quest item:

```

script (tsconfig project)
- {} MyNewGame

module MyNewGame.Items {
  export function Journal(): IItem {
    return {
      name: 'Joe\'s journal',
      equipmentType: StoryScript.EquipmentType.Miscellaneous,
    }
  }
}

```

We have the elements that we need, now we need to wire them together. The simplest way to do this is to extend the conversation you can have with Joe with a few nodes and replies. First add a new reply when you come to see Joe for a second time:

```

</node>...</node>
<node name="return">
  <p>
    Back so soon?
  </p>
  <replies>
    <reply>...</reply>
    <reply>...</reply>
    <reply node="lostjournal">
      I noticed you seem a bit upset.
    </reply>
  </replies>
</node>

```

Then add the new nodes that are needed for the quest to unfold:

```

<node>...</node>
<node name="lostjournal">
  <p>
    I guess I am. I can't seem to find my personal journal. It is very important to me. Can you help me find it?
  </p>
  <replies>
    <reply node="pleasefindit" quest-start="Journal" set-start="foundjournal">
      Of course. I'll return it as soon as I see it.
    </reply>
  </replies>
</node>
<node name="pleasefindit">
  <p>
    I hope you have better luck than I searching.
  </p>
</node>
<node name="foundjournal">
  <p>
    Have you found my journal?
  </p>
  <replies>
    <reply>
      Not yet, sorry.
    </reply>
    <reply requires="quest-done=Journal" quest-complete="Journal">
      Yes I have. Here it is!
    </reply>
  </replies>
</node>
</conversation>

```

As you see, I used a few new reply attributes here, `quest-start` and `quest-complete`. These will trigger the start and complete functions of the quest whose name is specified, so the Journal quest in this case. Also, I added one of the quest-related `requires` attributes, in order not to show the reply that you found the journal until you actually have it in your possession. You can also use the `quest-start` and `quest-complete` requirements to show replies only when you started or completed a specific quest.

Now, we need add the quest to Joe in the `friend.ts` file:

```

garden.events.push((game: IGame) => {
  game.logToLocationLog('Ah! There is the hedgehog Joe was talking about.');
```

```

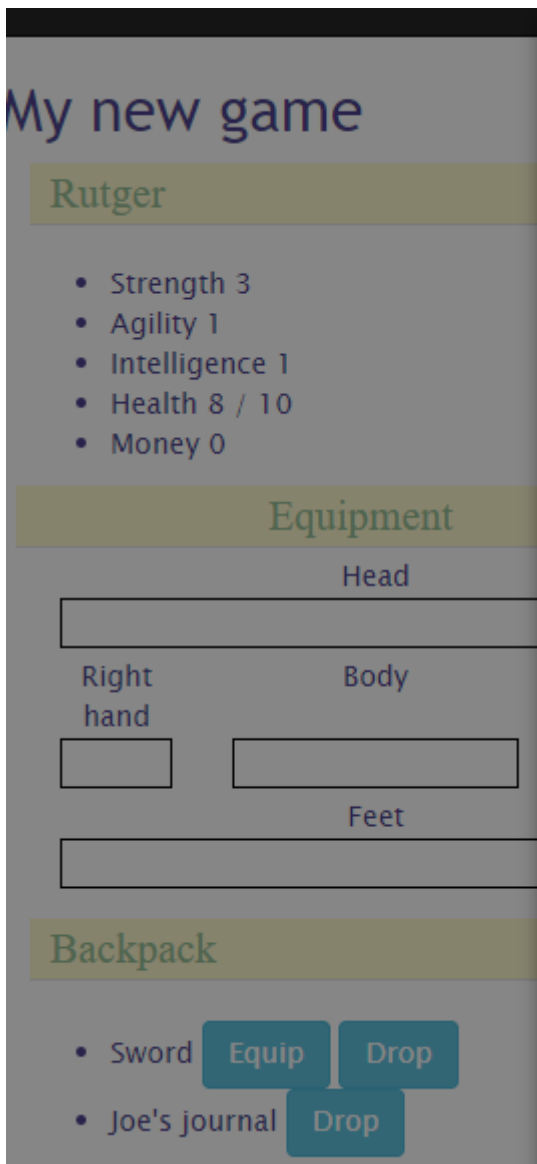
});
}
},
quests: [
  Quests.Journal
]
}

```

Ok, so Joe has a quest for us and we can talk to him to give it to us. You can test all of that by building and running the game. But we still need to put his journal somewhere in the game world for the player to find it and complete the quest. Let's add it to the basement location:

```
module MyNewGame.Locations {  
  export function Basement(): StoryScript.ILocation {  
    return {  
      name: 'Basement',  
      destinations: [  
        {  
          text: 'To the garden',  
          target: Locations.Garden  
        }  
      ],  
      items: [  
        Items.Journal  
      ]  
    }  
  }  
}
```

Now we have a nice quest ready for the player. He first has to talk to his friend and agree to help him. Then, he has to find the journal which is in the basement. To get in the basement, he needs a key. This key is in the possession of the bandit which he needs to defeat. Play the quest and return the journal to Joe. You should see something like this when you return to him:



Talk to Joe

Have you found my journal?

- ☐ Not yet, sorry.
- ☐ Yes I have. Here it is!
- ☐ Never mind. See you later.

Conversation history

- Hello there. How are you doing today?
Fine, thanks.
- Good. What are you up to today?
Never mind. See you later.
- Back so soon?
I noticed you seem a bit upset.
- I guess I am. I can't seem to find my personal journal. Can you help me find it?
Of course. I'll return it as soon as I see it.
- I hope you have better luck than I searching.
Never mind. See you later.
- Have you found my journal?
Not yet, sorry.
- Have you found my journal?

Give Joe his journal and you'll earn 5, erm, money...

Note that when you need to track progress on one of your quests, you can use the quest progress object. It is not typed, so you can add anything to it that you need. You can also check in code whether a quest is done by checking its completed flag.

16 Adding media

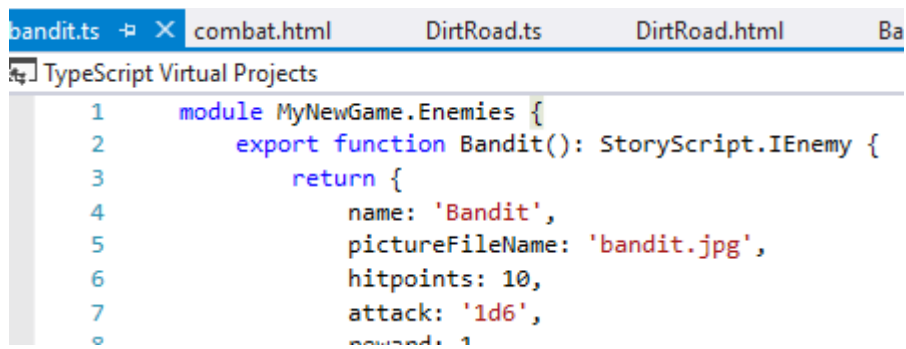
As StoryScript uses HTML and JavaScript for its interface, you can use any media that will work with these technologies in your game. For descriptions, which are plain HTML files, it is easiest. For example, find yourself a picture you want to add and place it in the resources folder of your game folder. Then, in your **Start.html**, include the following HTML-tag:

```

```

Build and refresh, and when your hero is at home you should see your picture shown.

You can also add images to enemies and items, though item pictures are not displayed yet. To add an image to the bandit, open the **Bandit.ts** file and add the pictureFileName property like this:

A screenshot of a code editor with a tab bar at the top showing 'bandit.ts', 'combat.html', 'DirtRoad.ts', 'DirtRoad.html', and 'Ba'. The 'bandit.ts' tab is active. Below the tab bar, the text 'TypeScript Virtual Projects' is visible. The code in the editor is as follows:

```
1 module MyNewGame.Enemies {  
2   export function Bandit(): StoryScript.IEnemy {  
3     return {  
4       name: 'Bandit',  
5       pictureFileName: 'bandit.jpg',  
6       hitpoints: 10,  
7       attack: '1d6',  
8       reward: 1  
9     }  
10  }
```

The **bandit.jpg** file needs to be present in the resources folder as well. Adding images to items works in the same way.

Build, refresh and reset and go out on the road. The bad guy should now have a face!

17 API

TODO: document features unless this can be done well in code.