Contents

Part 1. Getting started

.1 Setting up NodeJS, Visual Studio Code and Git

Install the required software

To get started creating games with StoryScript, there are a couple of tools you need to install and configure. First of all, you'll need **NodeJs** to be able to build and run your game. Download it from **https://nodejs.org** and run the installer.

To work with StoryScript effectively, you should use an editor that supports you in creating your game components like HTML and TypeScript files. Also, out-of-the-box support for Git is a great bonus. 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).

Get the StoryScript code

Now that you have your development environment set up, it is time to get the StoryScript code from GitHub. You can do this directly from Visual Studio Code. Start the program, open the command palette (**CONTROL** + **SHIFT** + **P**), select the **Git: Clone** command, pass in the StoryScript URL (https://github.com/StrixIT/StoryScript) and press **ENTER**:
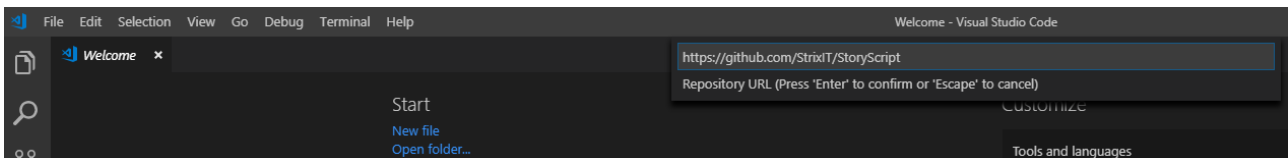
If you get an error that the command is not found, you need to install Git for windows. You can download it from https://git-scm.com/download/win. There are a number of options to choose from during the install, you should be fine using the defaults selected. Once you're done, close and re-open Visual Studio Code and try again.

Select a folder on your computer or create a new one to clone StoryScript to:

3
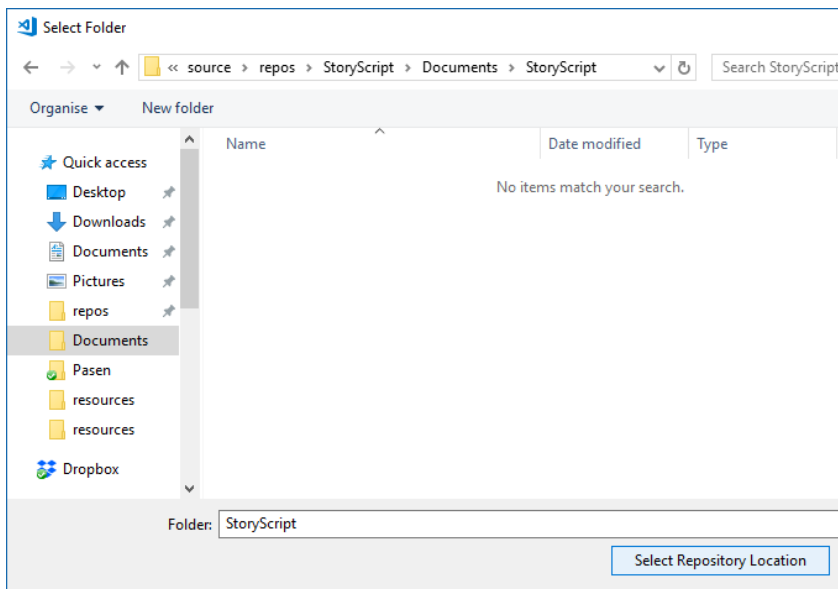
You should see a message in the lower right corner that a clone is in progress:
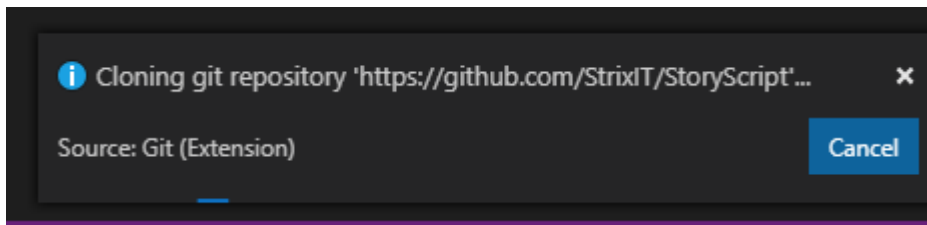
When done, opt to open the new repository:
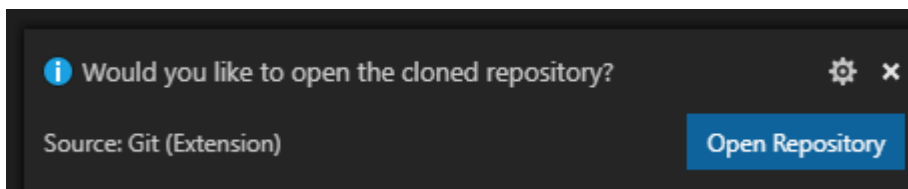
You should see the StoryScript folder open in your explorer on the left:

Getting the StoryScript code snippets

You'll have a much easier time working with StoryScript when you take advantage of the code snippets that are available. Copy the **StoryScriptSnippets.code-snippets** file from the **StoryScript\CodeSnippets** folder to the **StoryScript.vscode** folder:
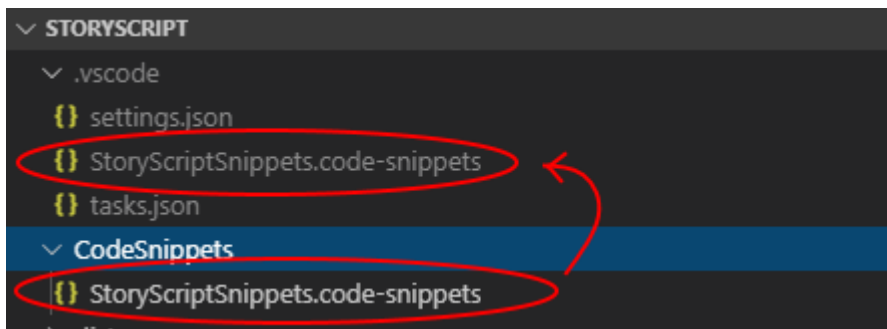
Now, whenever you start typing in a .ts, .js or .html file, you can quickly set up locations, items, barriers etc. by starting with ss (for StoryScript) and then the piece of code you need, e.g. ssLocation. Press CTRL-SPACE to get autocomplete to show if it doesn't automatically, then press ENTER when the right snippet is highlighted:
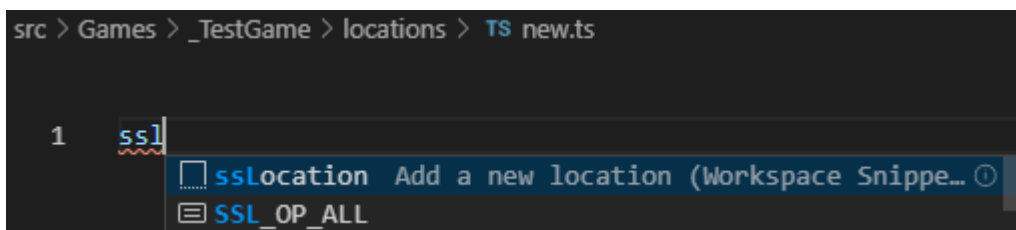
To see what snippets are available, you can just open the snippets file and search all the lines with **"prefix:"** (use CTRL-SHIFT-F for a convenient search):
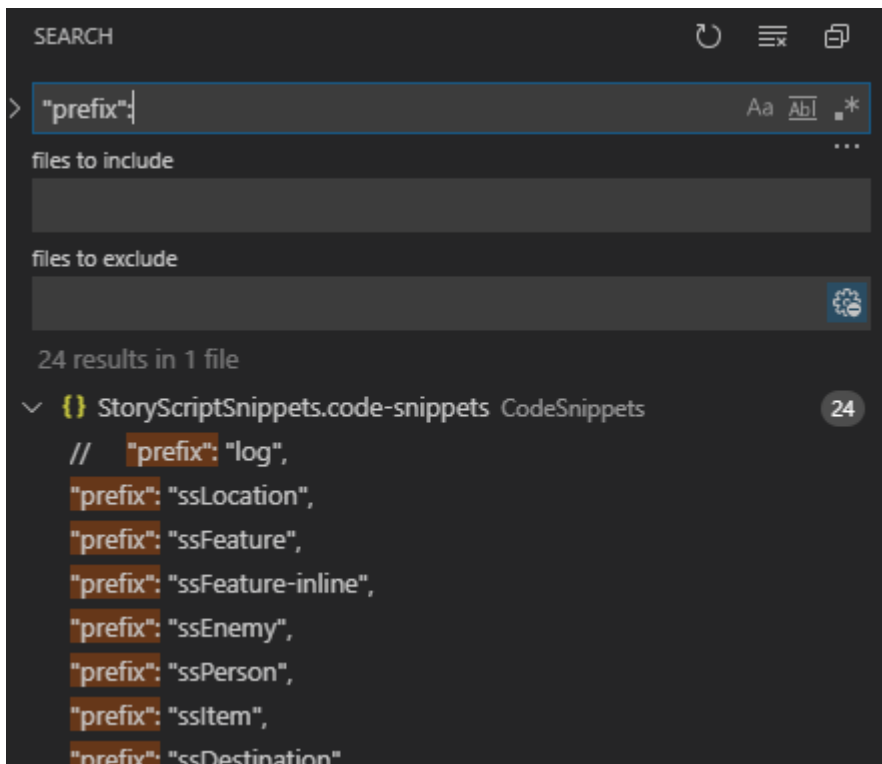
Some snippets can be used to quickly create new entities like locations, but there are also console commands that make this even easier (behind the scenes, these are using the same snippets). These commands will be discussed when working on the tutorial games.

Create your own StoryScript branch

Now that you have a copy of the master branch of StoryScript, you need to create a branch of your own to work on. Again from the command palette, select **Git: Create Branch**, type a name for your branch and press enter:
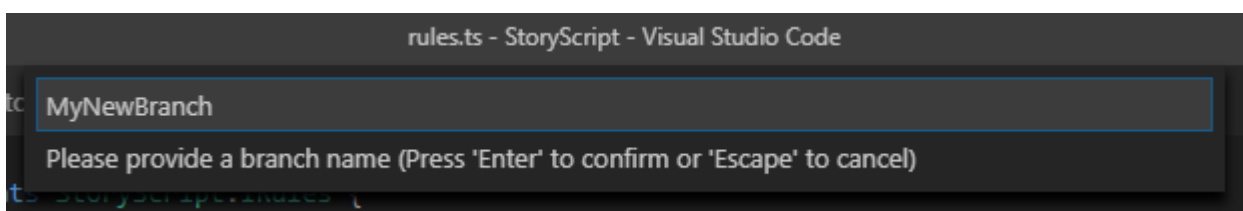
When asked, select the master (not the origin/master) branch as the reference for your new branch (it could be that this step is skipped):

Your new branch should be created and you should switch to it automatically. You can see the branch you're on in the lower left corner:

If asked whether to run git fetch automatically, you can answer both yes or no. I prefer no and do it manually.

You can publish your branch if you have a GitHub account using the **Git: Publish Branch** command. You'll be asked for your GitHub credentials when you run the command, after which your branch is added. You can then store your code online. From now on, you can use Git to control your StoryScript sources and update the StoryScript engine to the latest version. More on that at the end of this chapter.

Great, now just one more step before you are all set up! Open the integrated development console by pressing **CONTROL** + `` ` ``. Then install all the required packages by typing '**npm install**' (enter). Wait until the installation is done.

Now, you can finally start working on your game. You can skip to chapter 2 now, though it is worth reading through the remaining parts of this chapter to be better prepared.

Working with the example games

You will be creating several example games in this tutorial. You can find the completed versions of these on the Examples branch. Use that branch to see them in action, or to copy code instead of typing it in yourself when working on the tutorials:
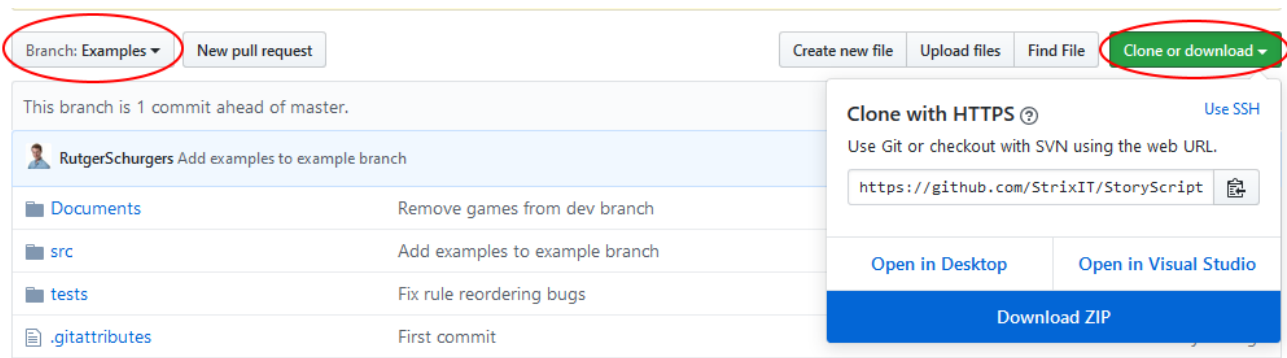
You can also find the examples running online at the StoryScript website, https://storyscript.azurewebsites.net/, and you can download the sources of the games from there as well:
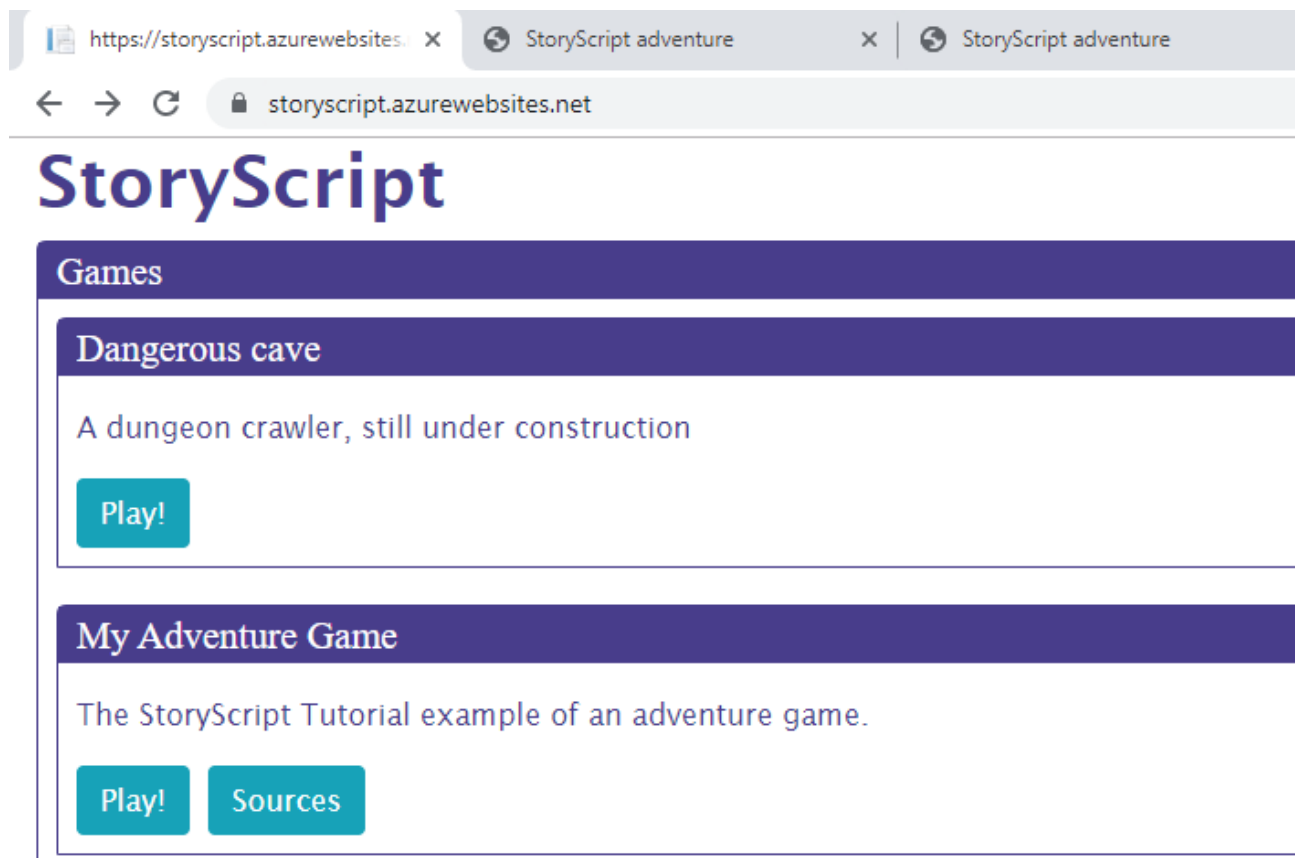
At the beginning of each chapter, a link is included that you can use to see the game in action.

Troubleshooting StoryScript

During game creation, as with any software development, things will go wrong. Here are a few important tips to help you find out what's going on so you can correct problems yourself or report problems for fixing:

1. When your build is failing, make sure you run **npm-install** to be sure all the required packages are available (also see **Saving your code to GitHub and updating StoryScript**). When you get a new version of StoryScript, it is possible new packages were added that

8

you'll need to build the game.

2. Use the developer tools of your browser while working on your game. Some problems that StoryScript can detect (like locations with no destinations and locations without html files for them) will be logged to the browser console. In Chrome, you can press **F12** or use the right-hand menu to open them:
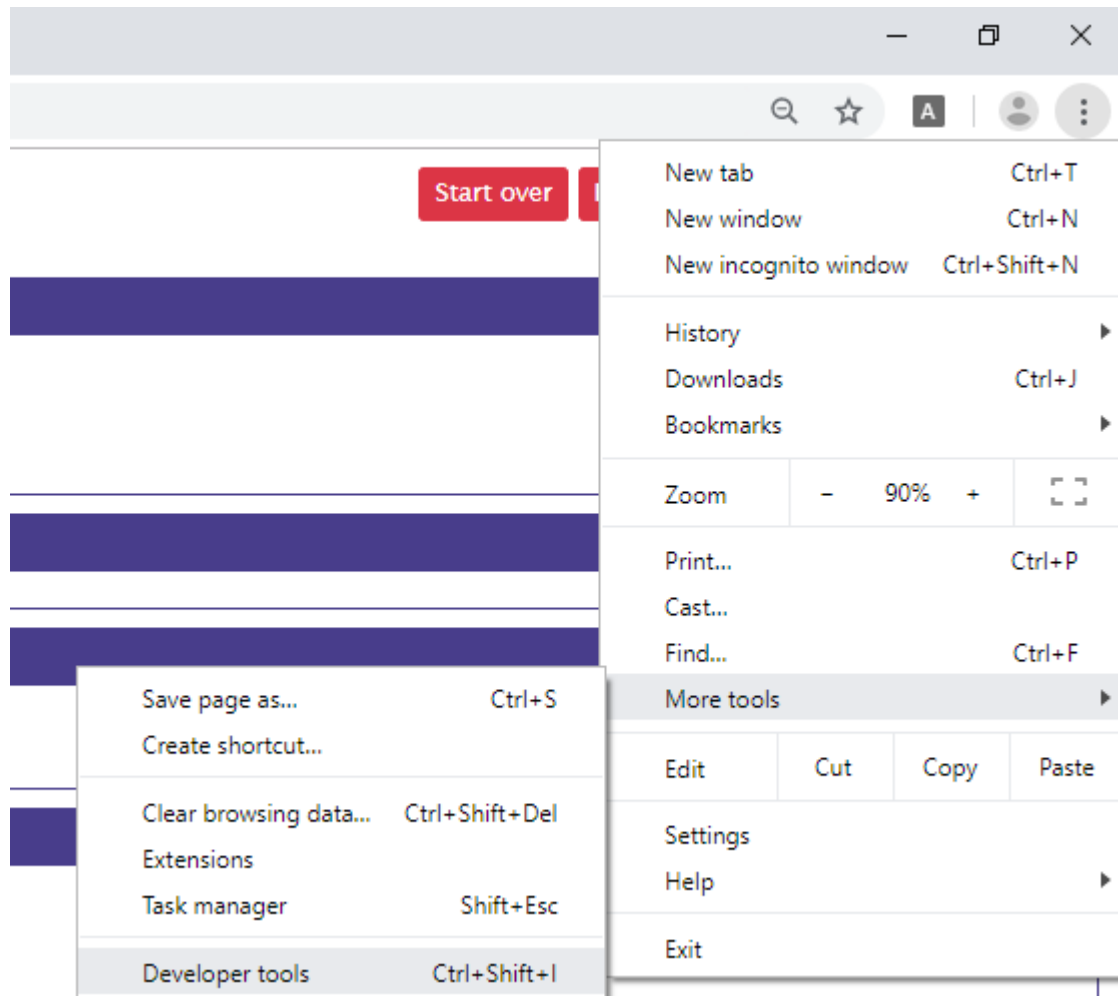
The tools will open at the bottom of the browser window:
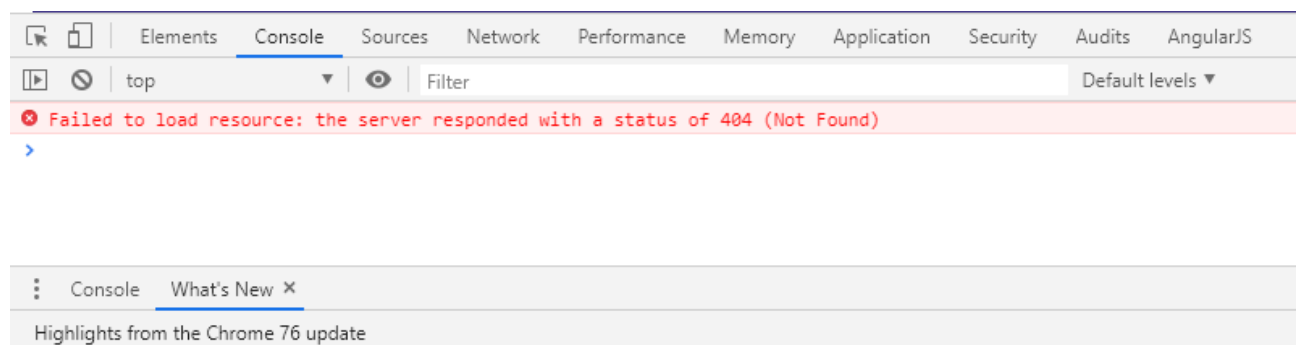
You can undock them and move them to a separate window if you want.

3. When the game fails to load, try clearing the local storage of your browser using the developer tools:
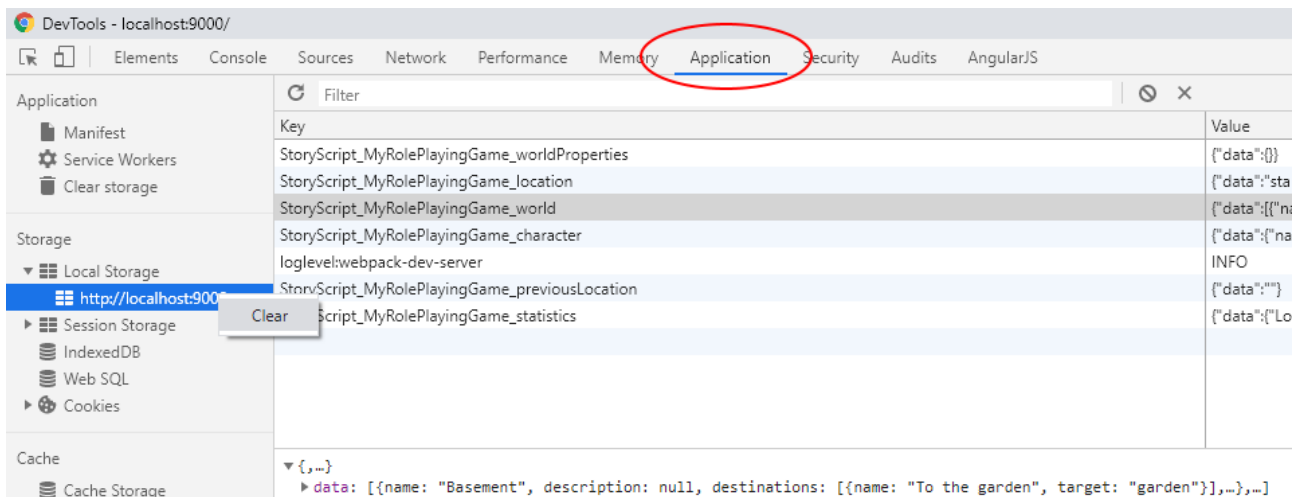
4. Don't change the properties listed here from code. As they are always refreshed from html source files for a better editing experience. Changes done to them through code won't be saved:

   a. Any properties called 'description' or 'descriptions' (on locations, enemies, etc.).

   b. Any properties on conversation nodes.

Saving your code to GitHub and updating StoryScript

When you have made a number of changes, you can save them to GitHub so they are stored not only on your computer. In the left menu, go to Source control, select the items you want to save by highlighting them (click one, then hold **SHIFT** and click another to select a list), right click and select Stage changes:
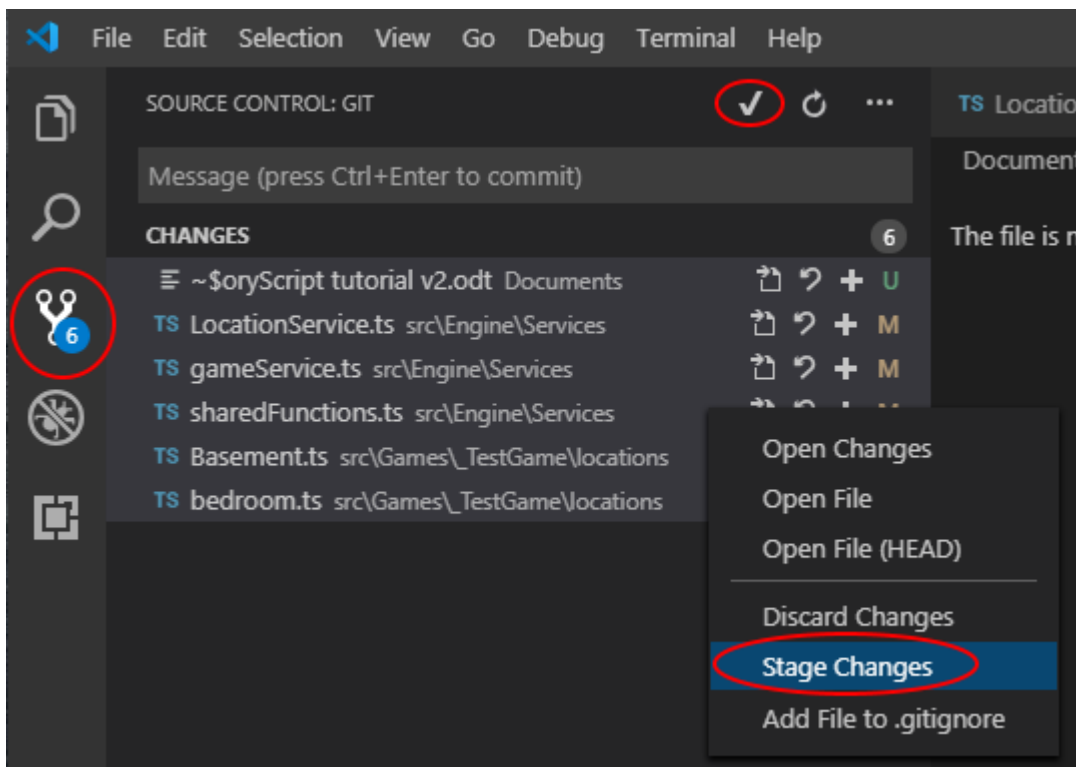
Or you can just click the checkmark at the top and select yes to automatically stage all files. Once the files have been stage, add a commit message and choose Commit:
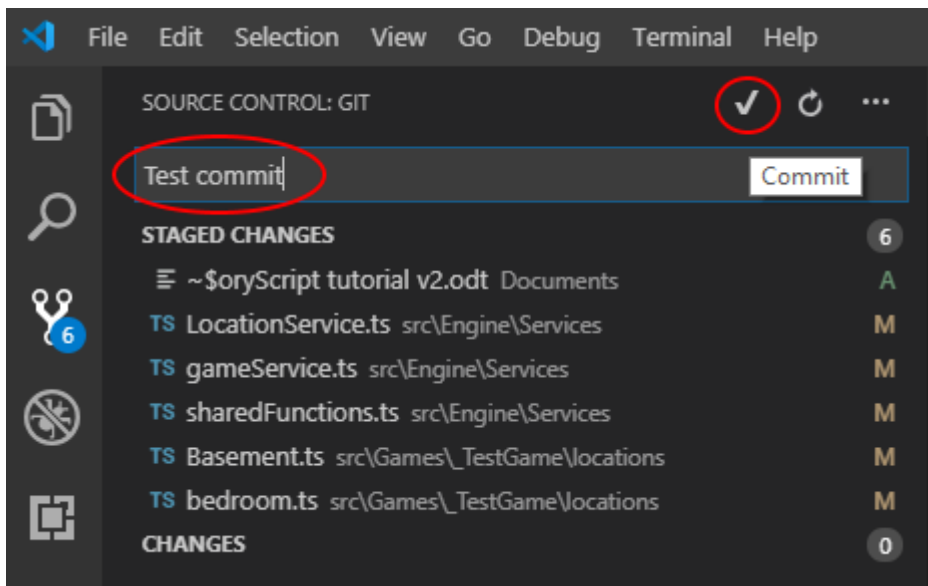
After that, click the revolving arrows next to your branch name in the bottom left of the screen to push your committed changes to your online branch:
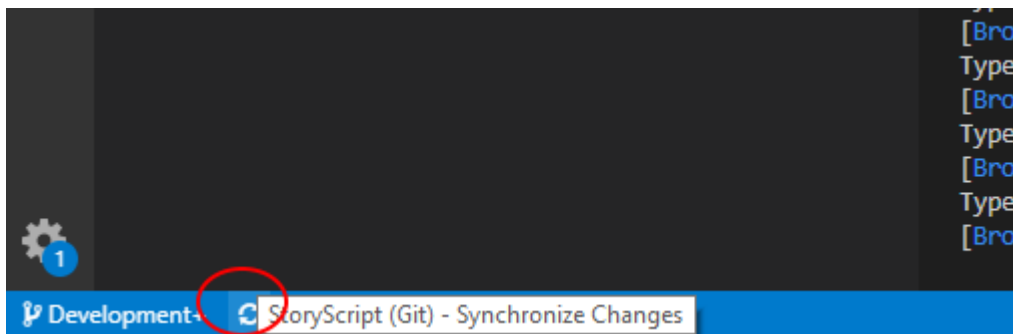
When a new version of StoryScript has been released, you can update your branch doing the following. Click your branch and switch to your local master branch by selecting master. Then synchronize the changes with the new version online. Go back to your own branch and press **CTRL-SHIFT-P** to bring up the Command palette and select the Git: merge branch command:
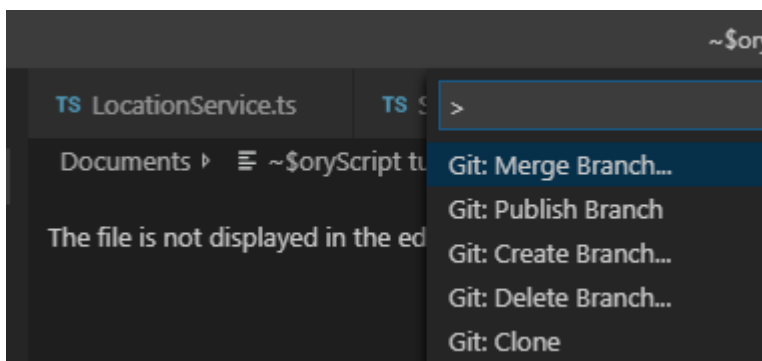
In the list shown, select the master branch:
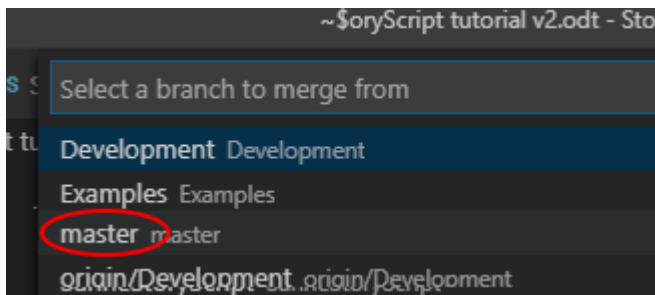
Then, you need to synchronize once more to make your own, local branch up to date with the latest StoryScript code. Having done that, run npm install in the terminal to get any package additions or updates that may be needed:

Now you should be able to run your game using the latest StoryScript code. If the build fails, it could be that breaking changes were introduced. Please check the GitHub readme in case this happens.

Decide what type of game you want to build

StoryScript started out as a simple engine for interactive stories, which basically are stories in which you as a reader make choices from time to time. From there, it expanded to support much more. You can now also create role playing games and adventure games with it.

StoryScript comes with several screen components that make up your game. Not all components make sense for all types of games. For example, in an interactive story you wouldn't need an equipment screen, and probably can also do without a backpack and most other additions needed for creating games that are more than stories with choices. It is easiest to get started by focusing on just the components you need. You can always add more components later when they are required after all.

In this tutorial, we'll be creating a few example games, starting with a simple interactive story. This will introduce some critical concepts that you need to understand for creating any StoryScript game. Having done that, we'll move on to creating first adventure and then role playing games.

.2   Creating a new game

To start working on a new StoryScript game, first choose a name for it, without spaces or special characters. Let's use **MyInteractiveStory** as an example. You can find the example online at https://storyscript.azurewebsites.net/games/myinteractivestory/index.html, the sources are available

for download:
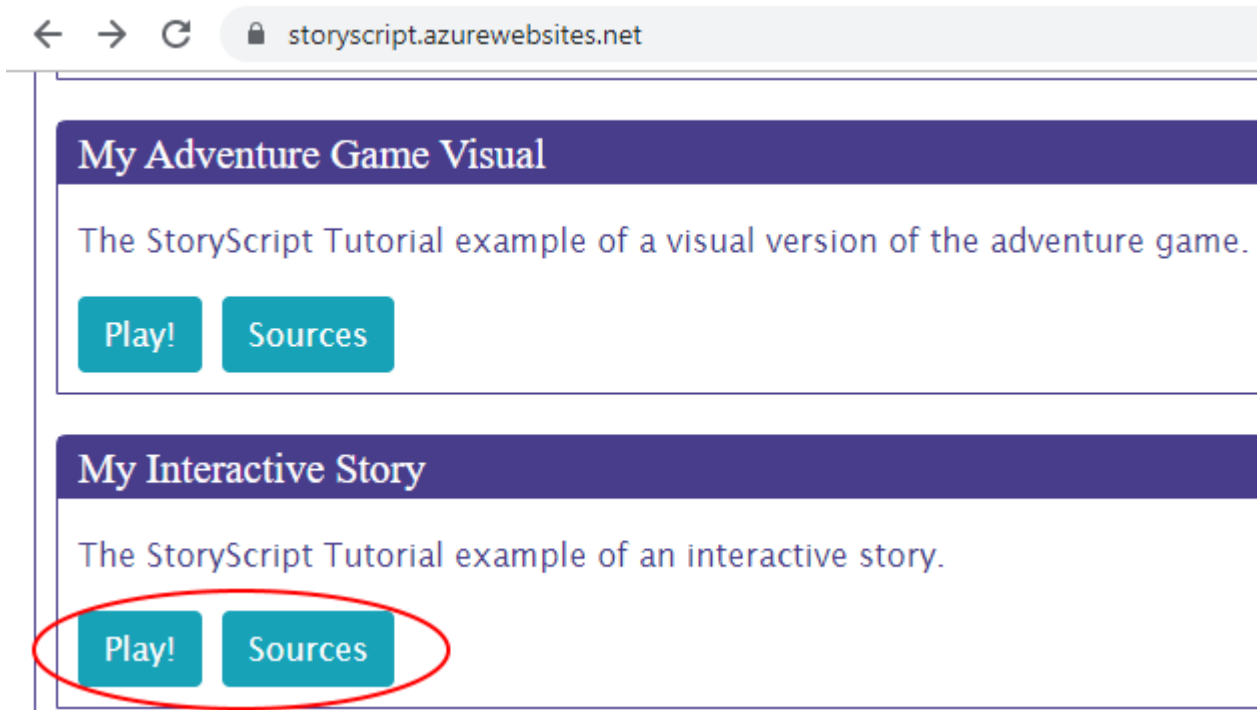
Open a new console window (**CONTROL** + "`'.**) and **type npm run create-game MyInteractiveStory**, like this:
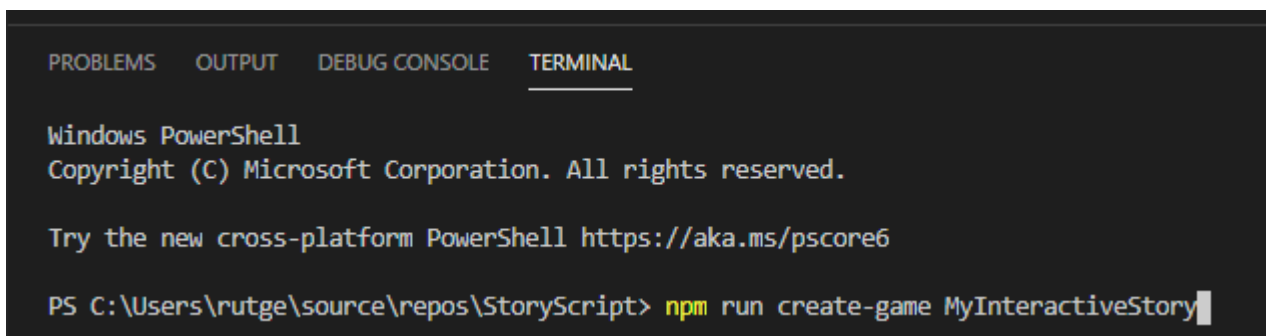
This will create a new folder with your game's contents in the src/Games folder:
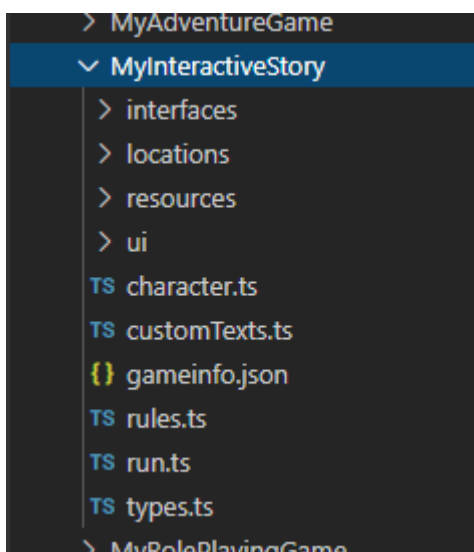
You can now run your new game. In the console, type **npm start** to launch the build of your game:

Chrome will open and wait for the build to complete. When it has, you'll see something like this:
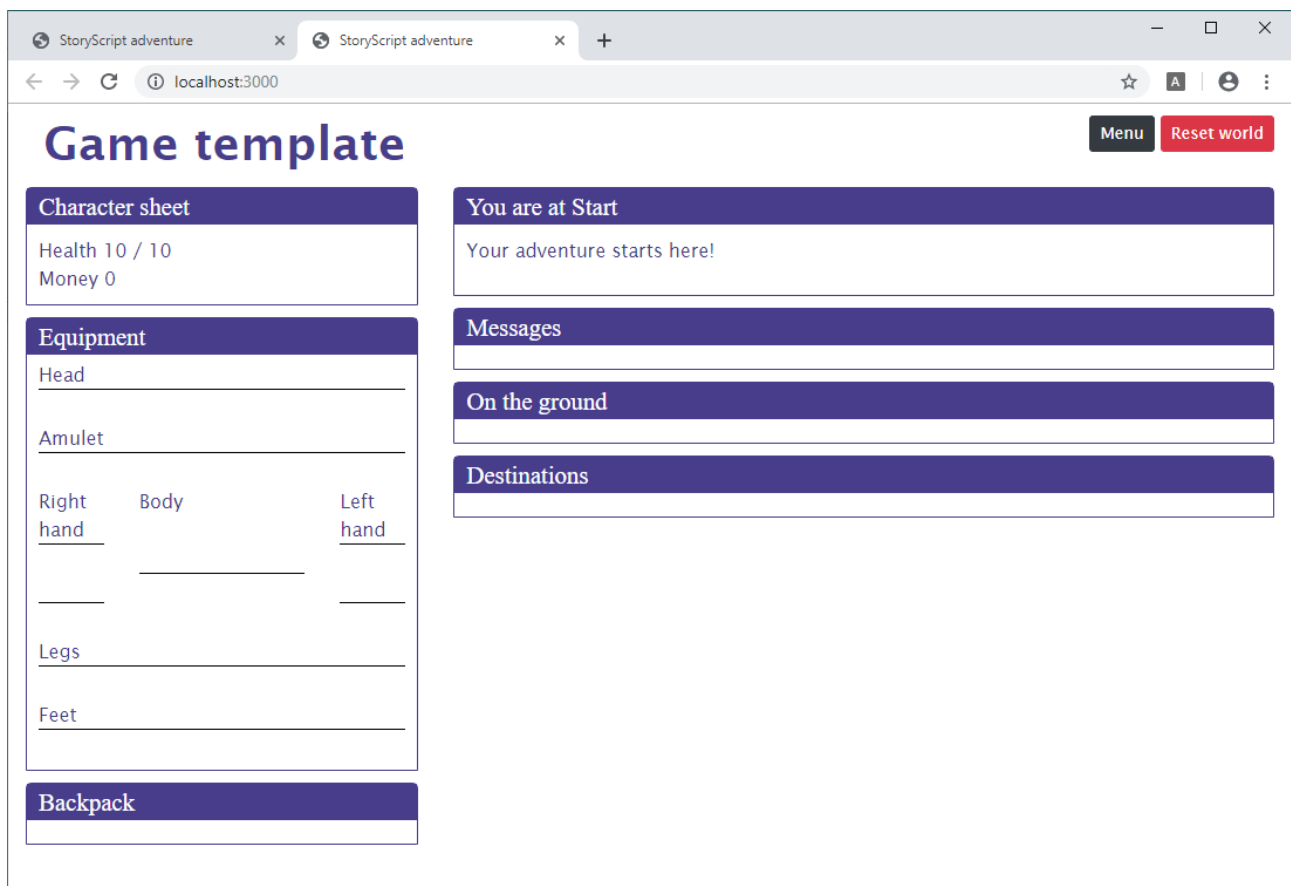
If you don't have Chrome, you can change the browser used in the webpack.development.js' file:

```js
JS webpack.development.js ✕

webpack > JS webpack.development.js > ...
  1    const merge = require('webpack-merge');
  2    const common = require('./webpack.common.js');
  3    const path = require('path');
  4
  5    module.exports = merge(common, {
  6        devtool: 'source-map',
  7        watch: true,
  8        devServer: {
  9            contentBase: path.join(__dirname, "../dist/"),
 10            port: 9000,
 11            hot: true,
 12            open: 'chrome'
 13        }
 14    });          You, 8 days ago • Webpack configs for dev and p
```

You can use 'FireFox', 'Edge' didn't work last time I tried. If you want to use Edge or for some reason your browser doesn't open, you can always start your browser manually and go to http://localhost:9000.

That said, Chrome works really well for StoryScript development, it is worth getting it. I'll be using it in this tutorial.

Congratulations, you are ready to begin creating your new game! Note that your browser will refresh whenever you save changes to your files (for css, the changes will be applied even without a refresh). You can enable automatic saving of files via the settings menu:
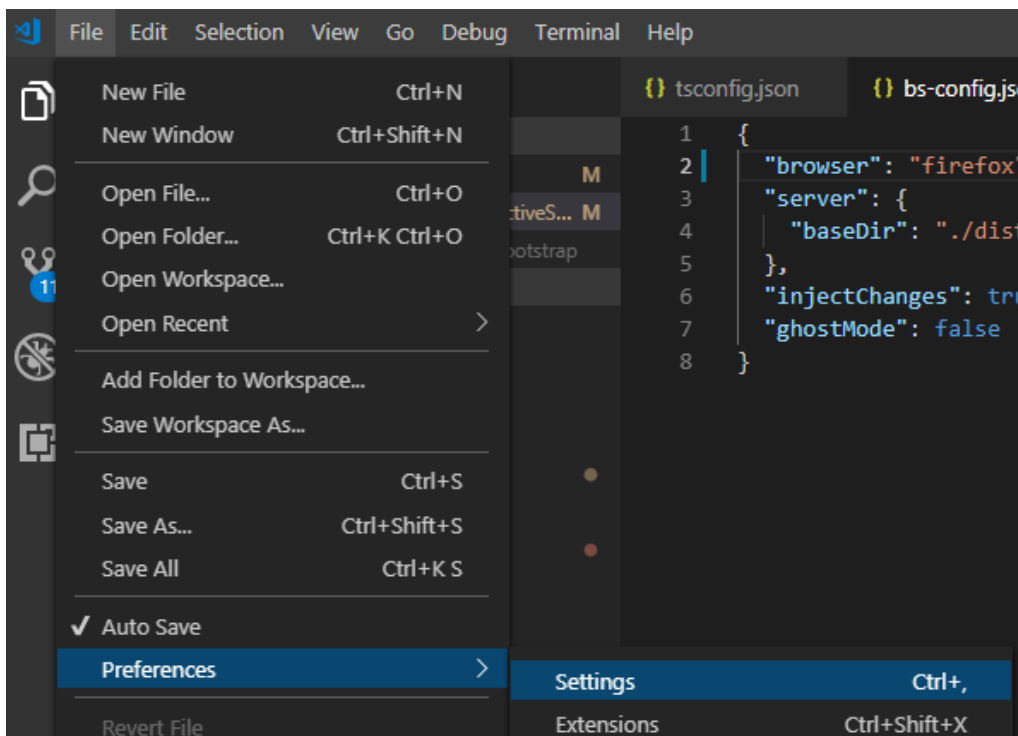
Change the contents of your '**User Settings**' to match the screenshot below. You can of course change the **Auto Save Delay** to suit your preference:
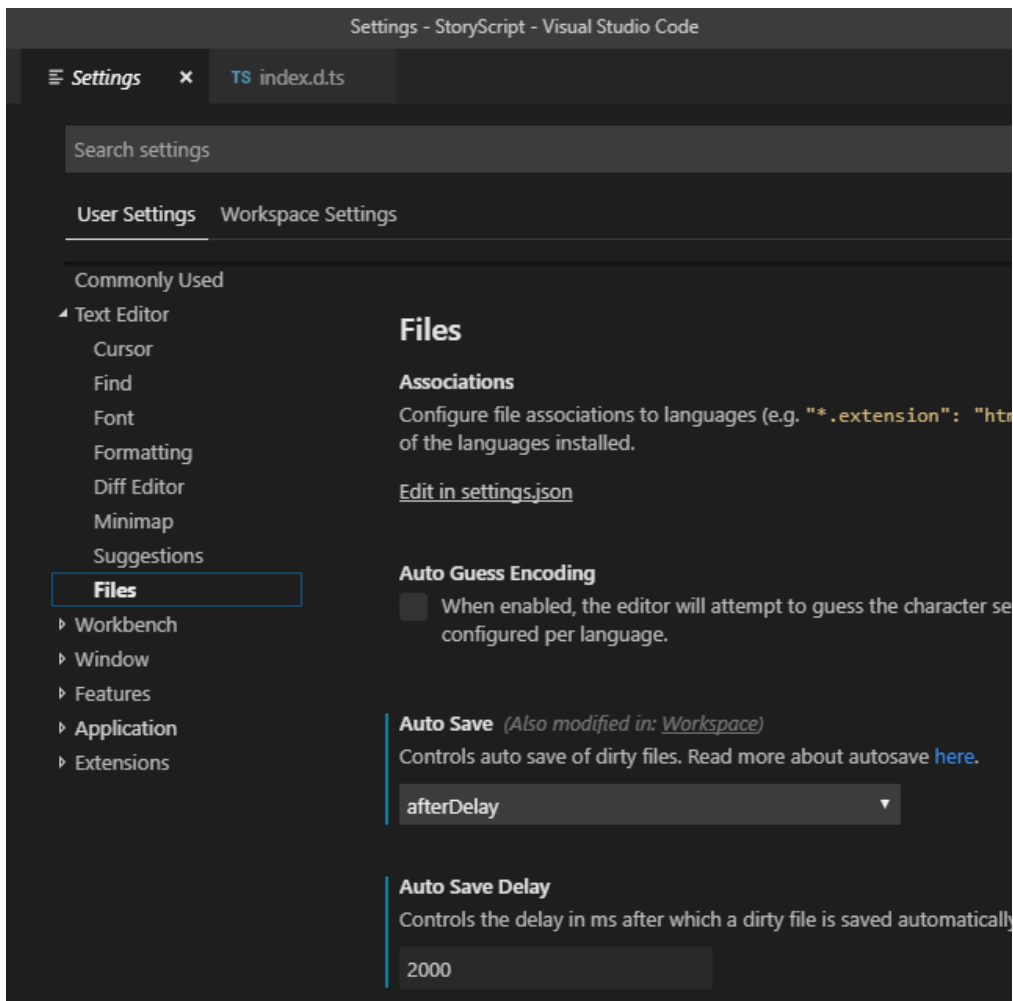
IMPORTANT NOTE: Now, whenever a file is added, changed or deleted, the browser will refresh. All changes to **css**, **html** and **resource** files should be reflected immediately in the browser. Changes you make to **ts** files are **NOT**. This has to do with the way StoryScript saves the state of your game between browser refreshes. It is one thing to display new text, styling or pictures right away, but resetting the game state on each refresh would be very annoying. That would mean that e.g. enemies and items are reset on each refresh. For changes made to your **.ts** files to show up, you need to press the **Reset world** button. From now on, when your browser refresh doesn't show you your latest changes, please press the Reset world button to make them appear:
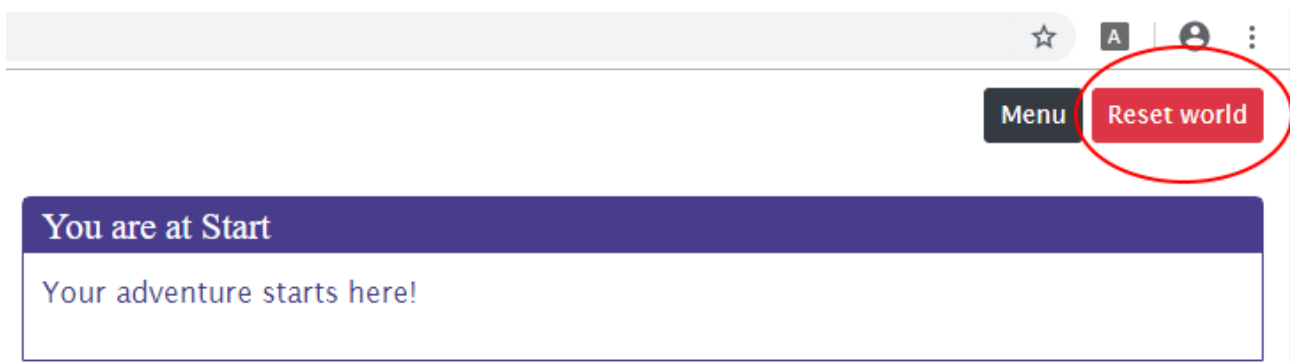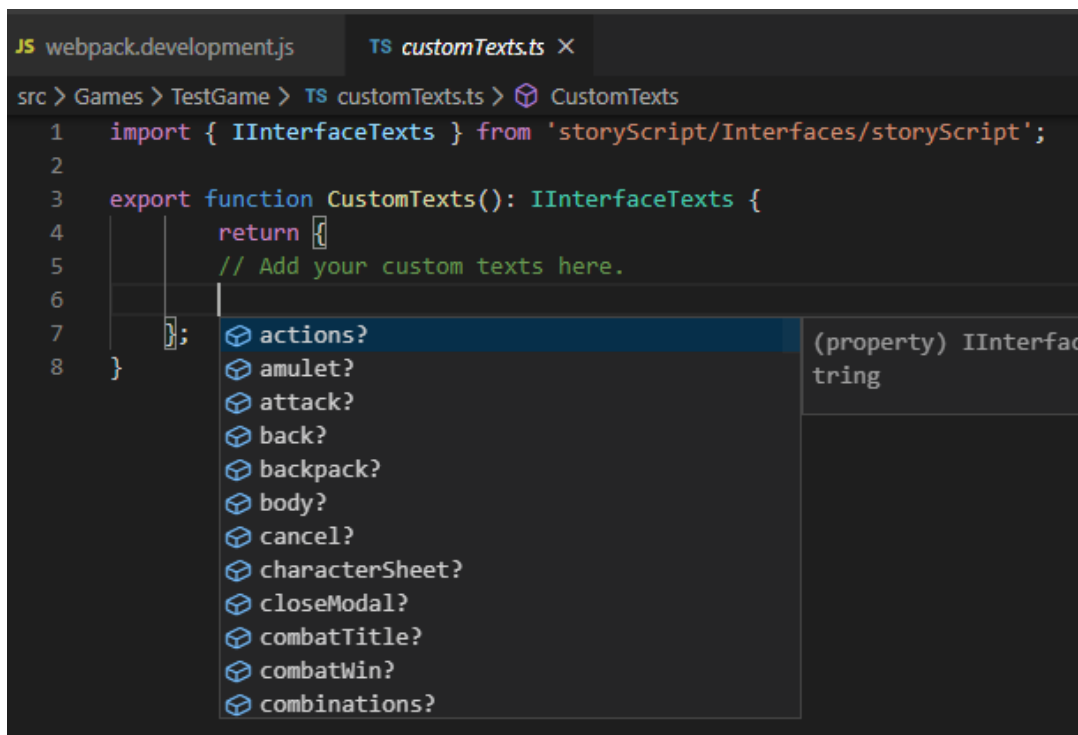
The state of your game is reset and all changes should show up.

.3   Customizing your user interface

With the structure of your game in place, let's start by changing the UI a bit to make it more specific to our game. I'll be showing you some basic ways in which you can do this using custom texts, html, css and resources. As StoryScript leverages basic browser technologies, you can use anything available in html and css to make it feel, look and sound as you want. So you can change font type, but also show pictures or play video.

We'll begin by changing 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 '**Chapter Start**' instead of '**You are at Start**' (Start is actually the name of the first chapter, we can give it a different name later).
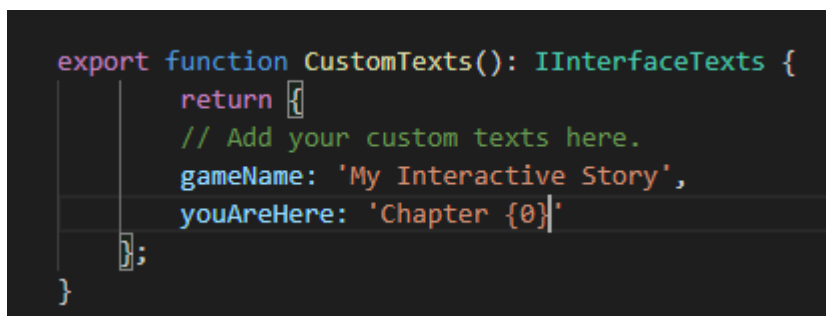
To change UI texts, go to the **customTexts.ts** file in your game folder. Place your cursor one line below where it says 'Add your custom texts here' and press **CONTROL-SPACE**. You will now see a list of all the texts that you can personalize:

```
JS webpack.development.js        TS customTexts.ts ×

src > Games > TestGame > TS customTexts.ts > 🔷 CustomTexts
   1    import { IInterfaceTexts } from 'storyScript/Interfaces/storyScript';
   2
   3    export function CustomTexts(): IInterfaceTexts {
   4         return {
   5         // Add your custom texts here.
   6         |
   7    };    🔷 actions?              (property) IInterfac
   8    }         🔷 amulet?           tring
                  🔷 attack?
                  🔷 back?
                  🔷 backpack?
                  🔷 body?
                  🔷 cancel?
                  🔷 characterSheet?
                  🔷 closeModal?
                  🔷 combatTitle?
                  🔷 combatWin?
                  🔷 combinations?
```

Change the game name and chapter texts by adding the following code. Note the placeholder for the chapter name:

```
export function CustomTexts(): IInterfaceTexts {
     return {
     // Add your custom texts here.
     gameName: 'My Interactive Story',
     youAreHere: 'Chapter {0}'
};
}
```

Your browser should now display the new texts.
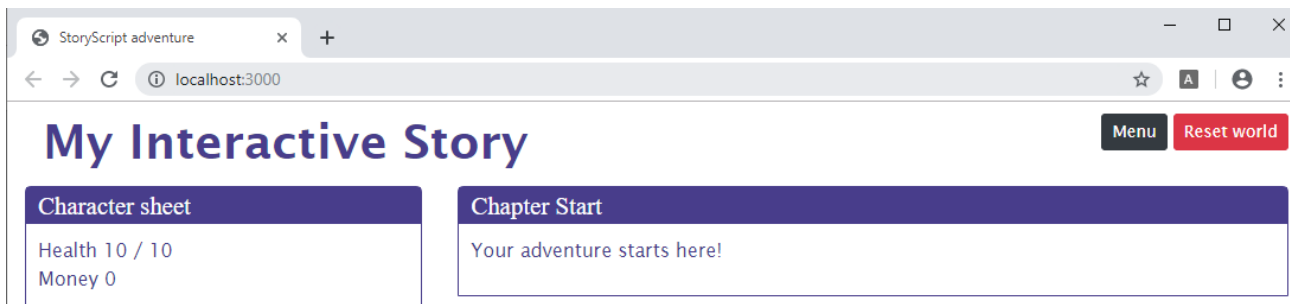
As a second step, let's remove the interface elements we don't need. You do this by copying the **MainComponent.html** file from the **src/UI/Components/Main** folder to your game's ui **components** folder:
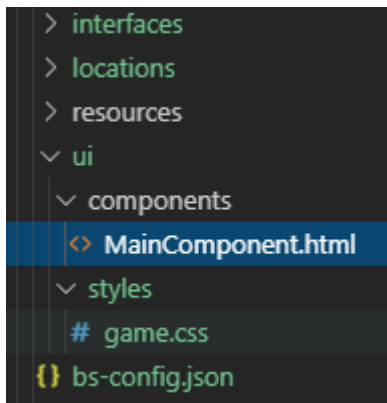
Open the file and change it to this:

```
1   <div>
2       <navigation></navigation>
3       <div class="container-fluid body-content">
4           <sound></sound>
5           <div class="row">
6               <div id="character-container" *ngIf="game.state === 'Play'" id="character-contain
7               </div>
8               <div id="location-container" [ngClass]="{ 'col-8': game.state === 'Play' && showC
9                   <div *ngIf="!game.state">
10                      {{ texts.loading }}
11                  </div>
12                  <div *ngIf="game.state == 'Play'">
13                      <location-text></location-text>
14                      <exploration></exploration>
15                  </div>
16              </div>
17          </div>
18      </div>
19  </div>
```
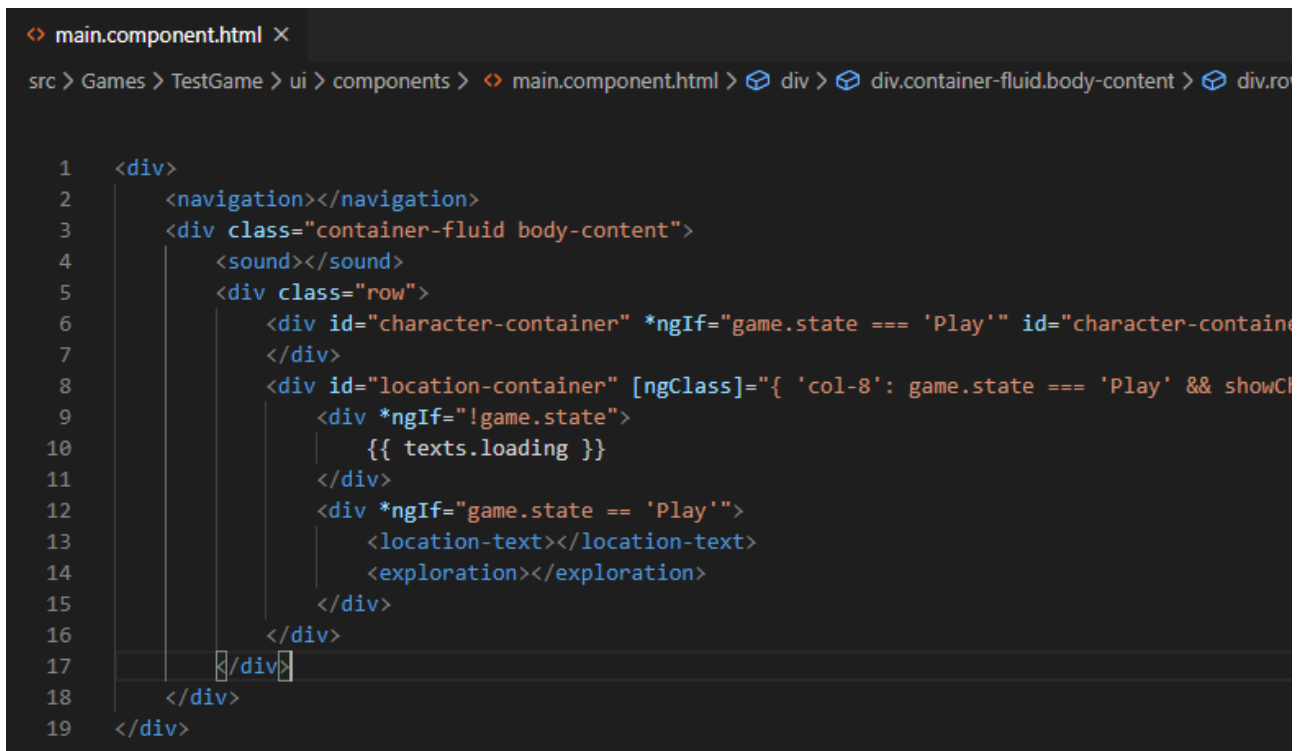
When you compare it to the original, you'll see that I removed all components (like **<backpack>** and **<enemy>**) that we won't be using. If you want to start using them later, you can always put them back in. Your new UI should look like this:

18

With that done, let's do some simple styling changes to illustrate how you can use CSS and image resources to change your game's appearance. Open the '**game.css**' file in the ui folder and first make these additions:


```css
.back-label {
    display: none;
}

#exploration-destinations > .box-title {
    display: none;
}

#location {
    margin-bottom: 0px;
}

#exploration-destinations {
    border-top: none;
    border-radius: 0;
}
```

With this, we hide the header 'destinations' and the space between the blocks to make them look like one block with a divider:

Those are the additions we'll keep. But let's also make some more to get an idea of what you can do with css, like below:

```
{} package.json      # game.css  ×    <> Start.html      TS customTexts.ts
 12    }
 13
 14    body {
 15        background: url('/resources/fecec72f.png');
 16    }
 17
 18    h1 {
 19        color: ■white;
 20    }
 21
 22    .box-container {
 23        background-color: ■white;
 24    }
 25
 26    .box-title {
 27        background-color: ■green;
 28    }
```

I added a background I randomly generated using http://bg.siteorigin.com/ to the **resources** folder in my game directory and referenced it in the css file for use as background picture. I also changed some colors. Your browser will show you the changes without a refresh:
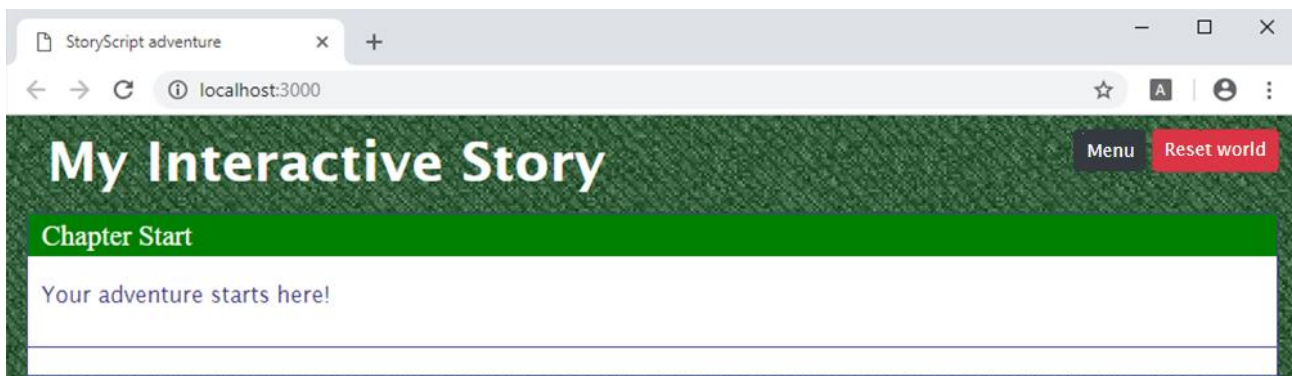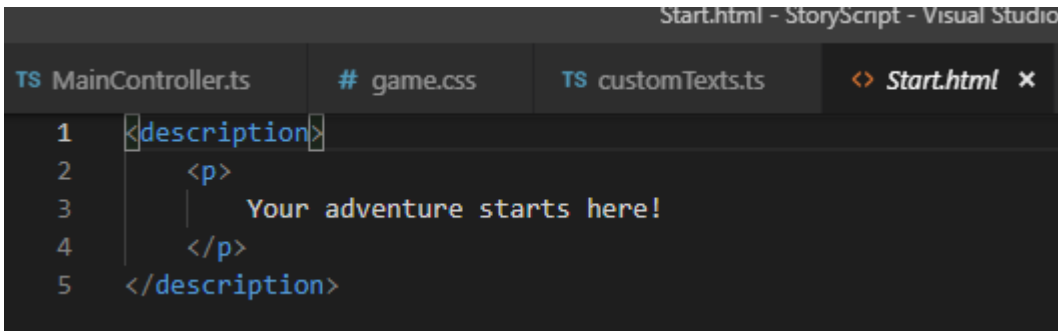
Enough on this for now. I'll reset the last styling changes for the rest of the tutorial.

Part 2. An interactive story
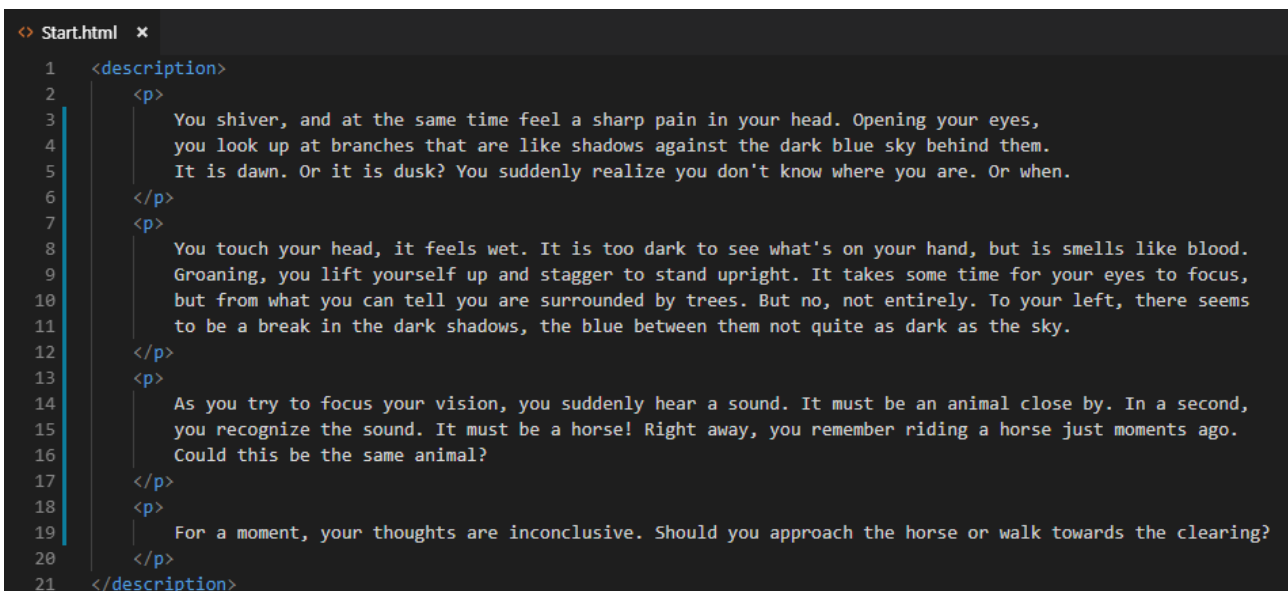
.4   Adding text and choices

Now that we have our UI set up, let's write the first chapter of our interactive novel and give the reader some choices. Just as with the UI, you can use everything a browser supports in your novel as you'll be using HTML markup to structure your content. But let's first focus on simple text. Open up the **Start.html** in the **locations** folder. It has the text you currently see:

Let's change it to contain a few paragraphs:

When you save, your browser refreshes to show you your new text:
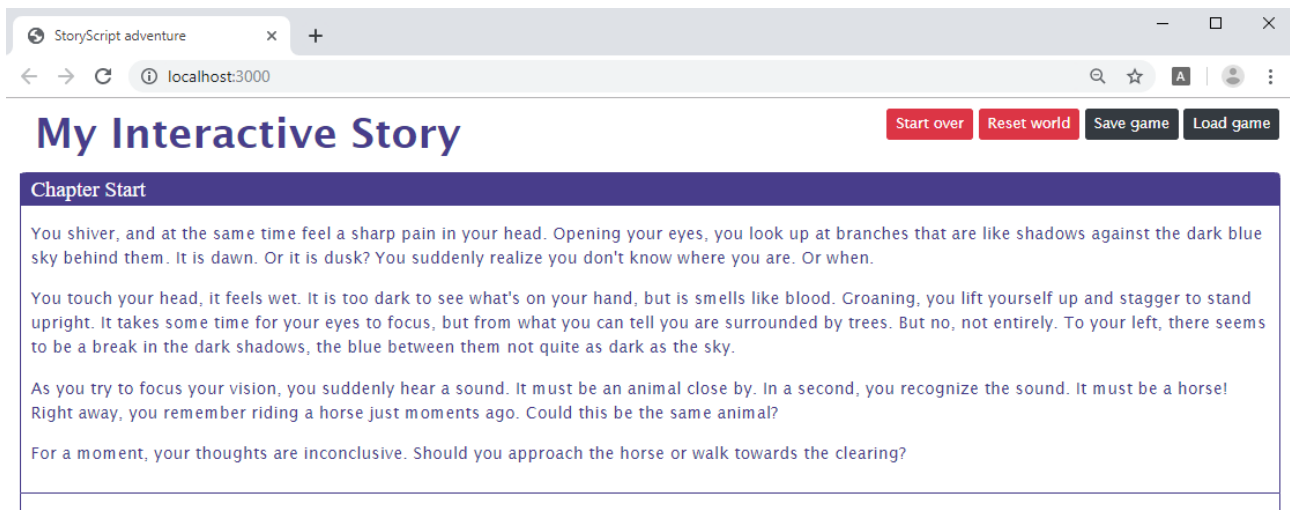
Right. Now we want to change our chapter name, as Start isn't very appealing. For this we'll change the name property of the location in the **Start.ts** file:

```
 5
 6    export function Start() {
 7        return Location({
 8            name: 'A forest clearing',
 9            description: description,
 0            destinations: [
```

Press **reset world** when the browser reloaded. You should now see:

With the opening chapter name and text in place, we want the reader to be able to make a choice. For this, we add destinations to our ts files. Each choice is represented by a destination, which leads to a new part of your story (a new combination of a ts and html file). Currently, the story tells about a choice between two alternatives, so we need to add in two destinations in our **Start.ts** file. Place the cursor inside the square destinations brackets and use the **ssDestination** snippet twice to add two destinations, like this:

```
    name: 'A forest clearing',
    description: description,
    destinations: [
        {
            name: 'Walk towards the clearing',
            target: null,
        },
        {
            name: 'Approach the horse',
            target: null,
        }
    ],
```

As soon as you add destinations this way, your browser (after pressing reset world) will show them to you although at first they are unavailable:

As you didn't add either a **clearing** or a **horse** location yet, there is nothing to continue reading so nothing to switch to. As soon as you add either of these, the corresponding destination button will be activated. Let's create a new location called Horse. There are two ways to do this, I'll show you the manual way first and then the more automated one.

To add a location by hand, start by adding both a **horse.ts** and **horse.html** file to the locations folder:
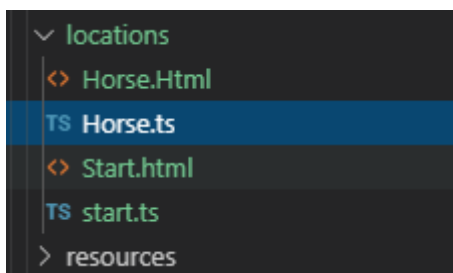
Open the **horse.ts** file. We'll use a code snippet to create the code we need for a location. Start typing **ssl** and intellisense should show you the location snippet (if it doesn't, press CTRL-SPACE):
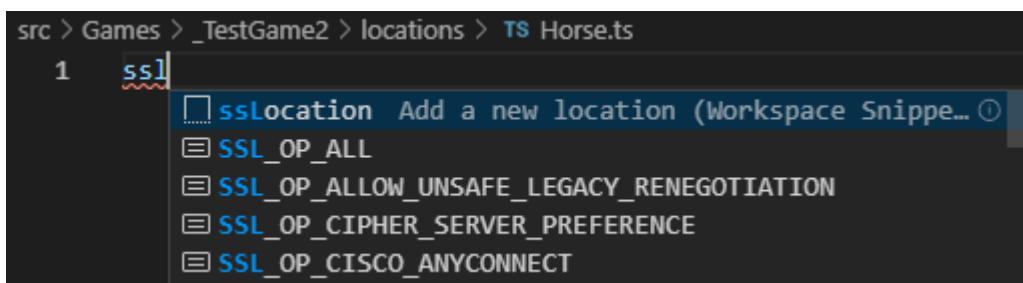
Press enter to activate the snippet. You now have the basic code set up that you need for a location. Go over to the **horse.html** file and type **ssd** to use the description snippet:
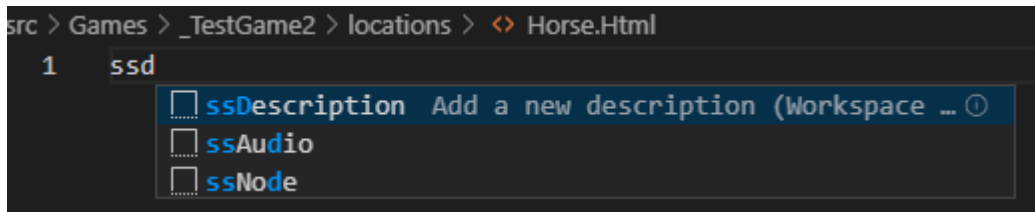
To add the two files with the basic setup automatically, you can use the console command **npm run create-location horse**:
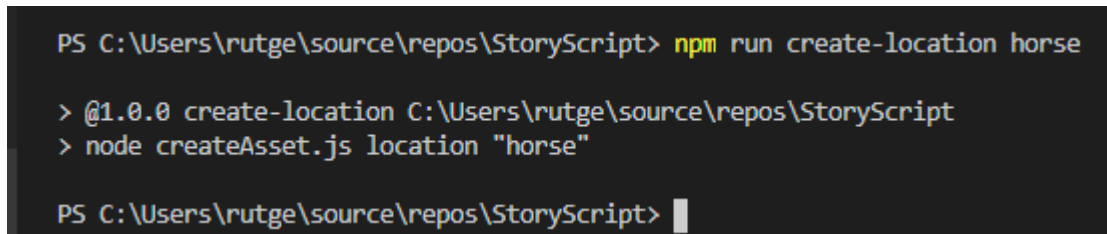
With that, the horse location is set up and you can link the it to the start location in the **Start.ts** file like this:
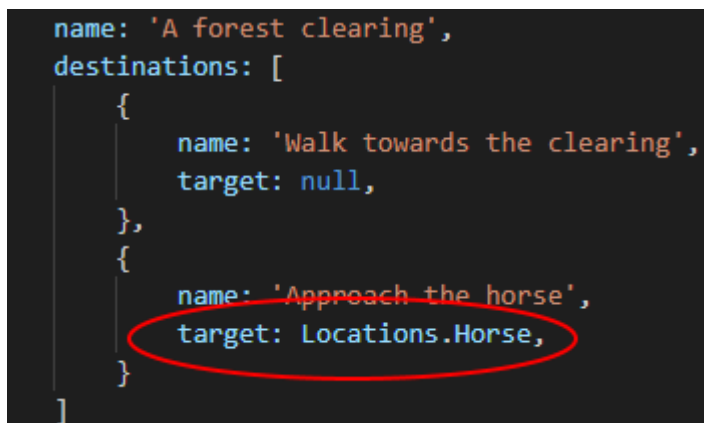
When you reset the world you'll see the horse destination activated:
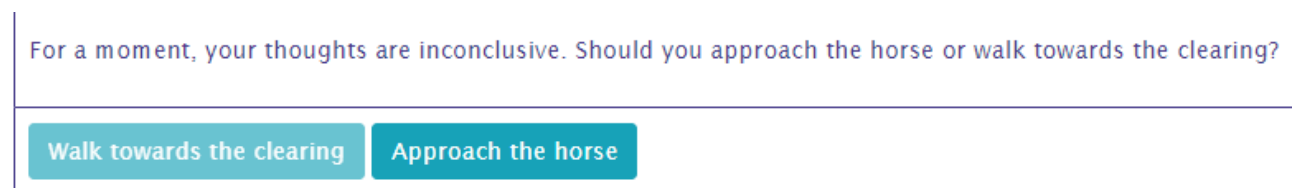
Add a piece of story to the **Horse.html** and click the button. You now see the new piece of the story displayed:

My Interactive Story

Start over | Reset world | Save game | Load game

Chapter A horse

You take a few steps towards the horse, but then trip over what is perhaps a root and fall towards the animal. With effort, you manage to grab a bridle. The horse is disturbed and pulls back it's head, putting you back on your feet with force. There is a flash of pain in your back, but before you really notice it you make out a number of red eyes between the dark trees.

Notice that the only way to get back to the starting point is to press Start over. With only two locations, that's ok, but when you have a bit more of them it would be convenient to automatically have a back button added. That's just for testing of course, whether the player should actually be able to return is something you need to decide and add yourself. But when testing, you can enable the automatic back button by adding this setting in your **rules.ts** file:

```
export function Rules(): IRules {
    return {
        setup: {
            autoBackButton: true,
```

When you start over and go to your second location, you should now see:

Chapter A horse

You take a few steps towards the horse, but then trip
The horse is disturbed and pulls back it's head, puttir
notice it you make out a number of red eyes between

Back to A forest clearing
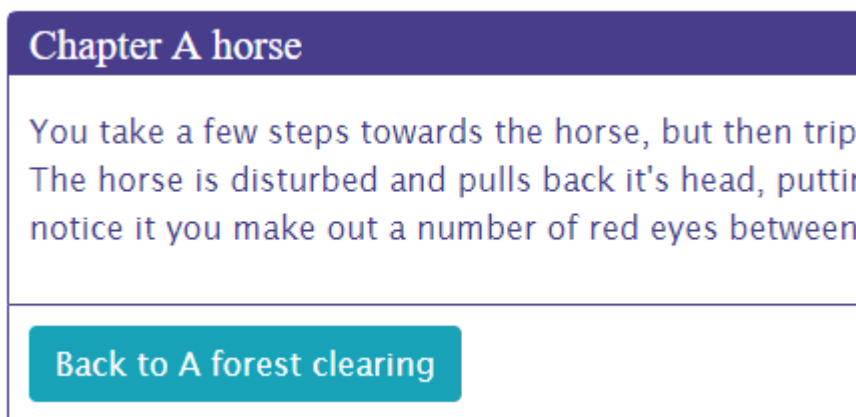
You can also use the save game and load game buttons in the menu to store a game state and go right back to it. In an interactive story that may be not that interesting, but when you create more advanced game types this is definitely a nice thing to have.

.5   Adding media

When you have a story to tell, you can write it in StoryScript. But often, you may want to use more than just text. In books, pictures can help bring the story to life. And when you have the web browser at your disposal, you can also use audio and video. All of this is entirely optional, and creating media of these types is an art you may not master (I know I don't). But when you can create your own pictures or music, or when you have someone working with you that can, you can use those in StoryScript.

Basically, it is as simple as including pictures, audio and video in your descriptions with the usual HTML tags (**<img>, <audio>** and **<video>**). StoryScript will help to start playing them automatically, once or repeatedly (useful for background music). As an example, let's add a picture and some background music to our first chapter. I'm adding a picture and audio, as well as aligning the picture to the text, changing the **start.html** file like this like this:

When your browser refreshes, you should now see this and hear the music play:
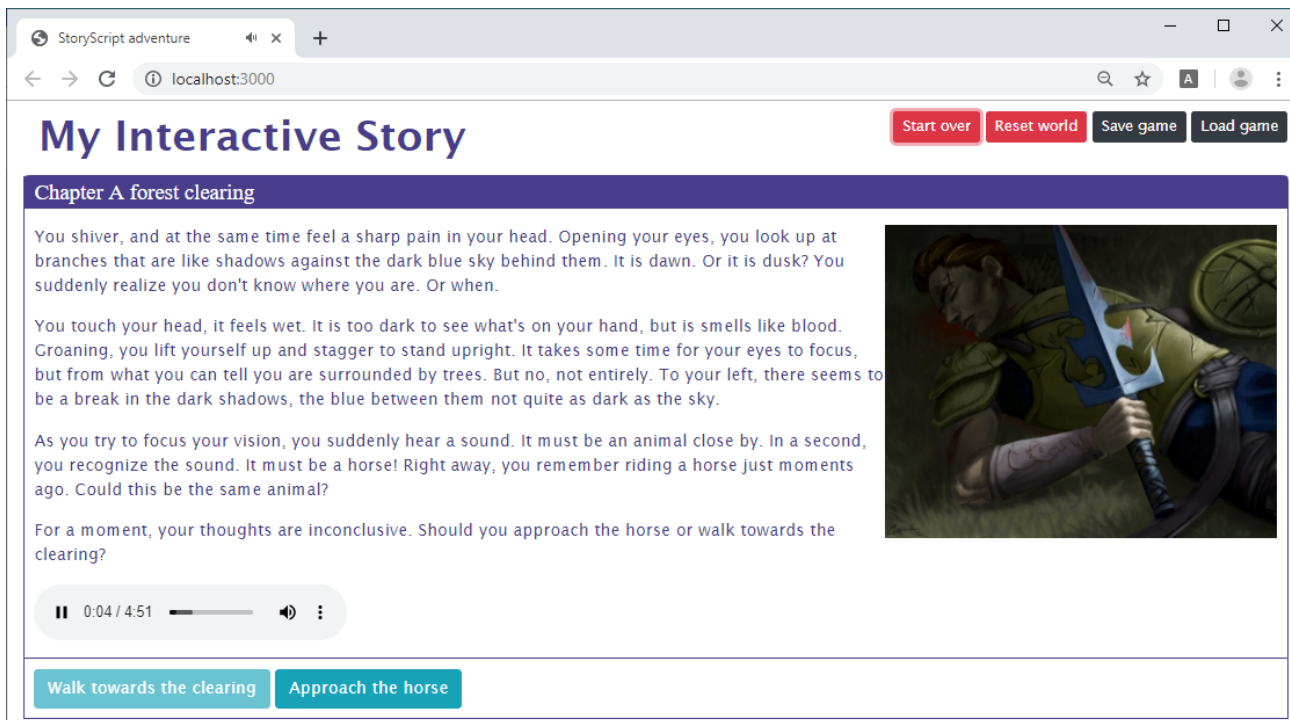
You can hide the audio controls if you want by just removing the controls attribute from the audio tag.

Note that now you've added some music for this one location, but you can also play music during game play, when the menu is active, etc. StoryScript has a few Game states (e.g. character creation and play) and Play states (e.g. menu and combat) for which you can play special pieces of music. To use this feature, configure a play list in your **rules.ts**:

```
export function Rules(): IRules {
    return {
        setup: {
            autoBackButton: true,
            playList: [
                [GameState.Play, 'play.mp3'],
                [PlayState.Menu, 'menu.mp3']
```

You also need the **<sound>** component in your **MainComponent.html** file. Add it:

```
<div>
    <navigation></navigation>
    <div class="container-fluid body-content">
        <sound></sound>
        <div class="row">
            <div id="character-container" *ngIf="ga
            </div>
            <div id="location-container" [ngClass]=
                <div *ngIf="!game.state">
```

When you now play the game, you should hear the play music. When you open the menu, the menu music is started.

Part 3. Adventure gaming

.6   *Using combinations*

In the early days of adventure gaming, classics such as King's Quest and Monkey Island defined the point & click adventure in which you wander the game world solving riddles by combining items, using them on characters or objects etc. Riddles like these can be created in StoryScript using combinations.

Working with combinations will require you to get your feet wet with some TypeScript programming. TypeScript is a superset of JavaScript, but that is not that important here. I've tried to make the StoryScript programming interface (Application Programming Interface or API) as simple to use as possible. That said, all programming requires some getting used to. If you stick with it, though, it gives you a lot of power to create unique games.

First, stop your interactive story game if it is still running by putting the focus in the console and pressing **CONTROL-C** and then **Y**:

You can find the example online at
https://storyscript.azurewebsites.net/games/myadventuregame/index.html . Create a new game called **MyAdventureGame** using the **npm run create-game** command as before. Run it using **npm start**. Your new game should look like this:

Ok, now let's again change the interface a bit to show just the parts we need for our adventure game. Copy over **MainComponent.html**, to the game's **ui/components** folder (you can also use css to hide parts of the interface if you want, but if you're not using whole blocks you might as well remove them).

Back to the Main component. We don't need the equipment screen, we'll only be using the backpack. We also don't want the On the ground, Destinations and Messages blocks. To remove them, change it like this:

```
1  <div>
2      <navigation></navigation>
3      <div class="container-fluid body-content">
4          <sound></sound>
5          <div class="row">
6              <div id="character-container" *ngIf="game.state === 'Pl
7                  <backpack></backpack>
8              </div>
9              <div id="location-container" [ngClass]="{ 'col-8': game
0                  <div *ngIf="!game.state">
1                      {{ texts.loading }}
2                  </div>
3                  <div *ngIf="game.state == 'Play'">
4                      <location-text></location-text>
5                  </div>
6              </div>
7          </div>
8          <div class="row" *ngIf="game.state === 'Play'">
9              <div class="col-12">
0                  <combination></combination>
1              </div>
2          </div>
3      </div>
4  </div>          You, 18 days ago • Add adventure game example
```

While you are at it, change the gameName in the **customTexts.ts** file to 'My adventure game'. When the browser refreshes, we should have a nice, clean page:

My adventure game

Menu   Reset world

Backpack

You are at Start
Your adventure starts here!
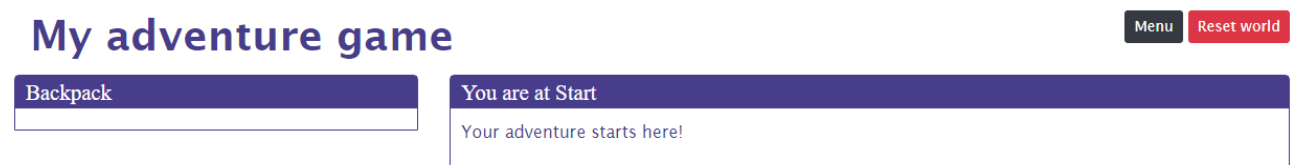
Ok, we'll work with these elements for now. Time to start defining our combinations. Defining what combinations should be available in the game is done in your **rules.ts** file. Let's define 'Walk', 'Use', 'Touch' and 'Look at' for this example. First, create a new **constants.ts** file with these contents, so we define our combinations in one place:

```
tart.ts        TS constants.ts ×    TS horse.ts      <>

src > Games > MyAdventureGame > TS constants.ts > ⁏⁏ C
        You, 18 days ago | 1 author (You)
   1    export class Constants {
   2        static WALK: string = 'Walk';
   3        static USE: string = 'Use';
   4        static TOUCH: string = 'Touch';
   5        static LOOKAT: string = 'Look';
   6    }       You, 18 days ago • Add adventure
```

We can use these definitions to create the combinations in our **rules.ts** file like this. You can use the **ssCombinationAction** snippet to add a new combination action:

```
setup: {
    getCombinationActions: (): ICombinationAction[] => {
        return [
            {
                text: Constants.WALK,
                preposition: 'to',
                requiresTool: false
            },
            {
                text: Constants.USE,
                preposition: 'on'
            },
            {
                text: Constants.TOUCH,
                requiresTool: false
            },
            {
                text: Constants.LOOKAT,
                preposition: 'at',
                requiresTool: false,
                failText: (game, target, tool): string => {
                    return 'You look at the ' + target.name + '. There is nothing special about it';
                }
            }
        ];
    }
}
```

There are a few things to note here:

- You can see that the combinations have an action text (e.g. '**Use**', '**Look**') and an optional preposition ('**on**' or '**at**'). In StoryScript, these will be used to create combinations such as **'Use pen on paper'** or **'Look at gate'**.

- As some combinations require two parts (we'll call these parts the **tool** and the **target**, e.g. **Use** requires both but **Look** does not (of course, you could also use binoculars to get a better look, but you get the idea)) you should specify when one does not require a tool.

- You can specify a default text displayed when a combination is tried that doesn't work in the **customTexts.ts** file. There are two templates you can override, one for combinations requiring a tool and one for those who don't. The numbers between the curly brackets are placeholders, and will be replaced at runtime:

noCombination: "You {2} the {0} {3} the {1}. Nothing happens."

1. The 'tool' for the combination
2. The 'target' for the combination
3. The combination name (Use, Look)
4. The preposition (on, at)

For example: "You use the pen on the paper".

noCombinationNoTarget: "You {1} {2} the {0}. Nothing happens."

1. The 'target' for the combination
2. The combination name (Use, Look)
3. The preposition (on, at)

For example: "You look at the gate".

If you want something more specific, you can specify a fail text per combination, as shown above. You can be even more specific when you know what combination is tried on what object, which will be discussed in a bit.

With your combinations defined, they should show up in your browser:
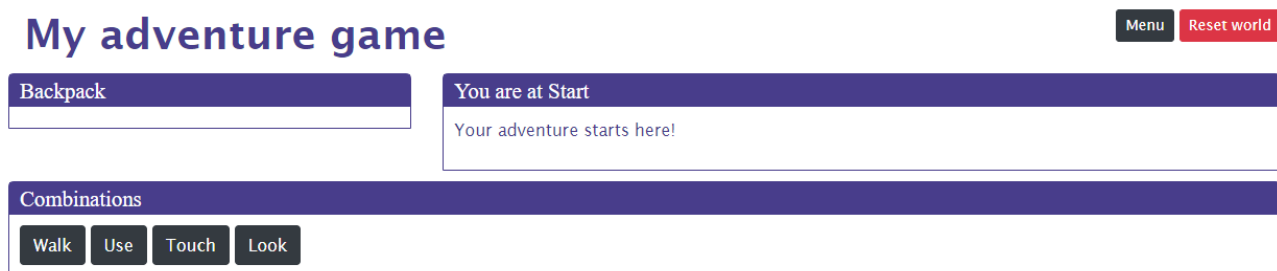
For our combinations to do anything, we need tools and targets for them. These can be any of the following StoryScript entities:

- Features
- Items
- Enemies and Persons
- Barriers

In this part of the tutorial, we'll focus on features and items. Enemies, persons and barriers will be discussed in part 4.

In StoryScript, you can work with combinations in two ways, which you can mix if you want to. The first is text-based, using text descriptions for your locations and features. You can also go picture-based, which means you use one or more pictures to make your world come to life. We'll start with a text example, and then show how you can build the example using pictures as well.

To create a location players can interact with, we'll create some features. Features are noteworthy elements of a location that a player can interact with. Let's start with a fountain as an example, one found in a forest clearing. Go to your **Start.html** and change the description element as shown below. Add the feature element in between the text, you can use the **ssFeature** snippet to do so more quickly:

```
{} tsconfig.json      <> Start.html   ×

src > Games > MyAdventureGame > locations > <> Start.html > ⬡ description
   1   <description>
   2       <p>
   3           You stand at the edge of a forest clearing. Tall trees border it on all sides,
   4           with thick undergrowth making the clearing hard to get to. In the center of the
   5 |         clearing is <feature name="fountain">a fountain with a six-sided base</feature>.
   6       </p>
   7   </description>
```

Second, open the **Start.ts** file and change it like this. To add a feature in a location, you can use the **ssFature-inline** snippet (the idea of features is that they are specific to one location so you can declare them inline, unlike e.g. items that are not (usually) bound to one location. You can also create stand-alone features if you want to use them somewhere else as well or to organize your code, that will be shown later):

```
export function Start() {
    return Location({
        name: 'Start',
        description: description,
        features: [
            {
                name: 'Fountain',
                combinations: {
                    combine: [
                        {
                            combinationType: Constants.LOOKAT,
                            match: (game, target, tool): string => {
                                return 'You look at the fountain water. It is very clear and reflects the forest.';
                            }
                        },
                    ]
                }
            }
        ]
    });
}
```

When your browser refreshes, press **Reset world** or use the menu's **Start over** to see the fountain feature. Press the Look combination button and then the fountain feature to see your match text displayed:
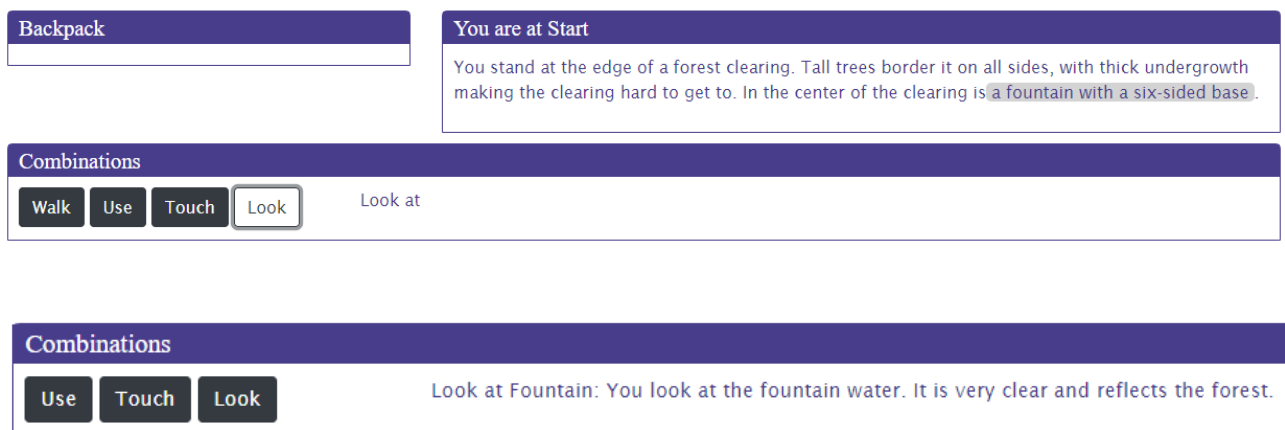
When instead of Look you use Touch, you should see the default fail text as you haven't specified any combination for the fountain feature and Touch:

32

Touch Fountain: You Touch the Fountain. Nothing happens.
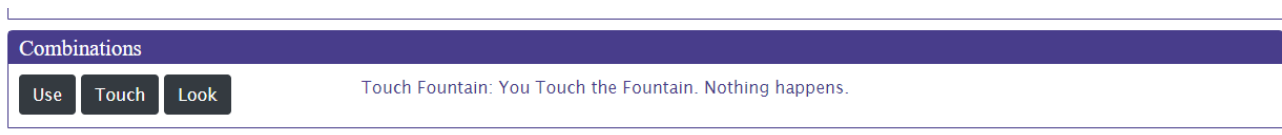
Ok, let's make the Touch action do something as well. When the player walks towards the fountain and touches the water, he'll hear a soft muttering coming from the undergrowth at the edge of the clearing. This should give him the opportunity to go and check out that spot.

For this to work, we need to add a couple of things. Start with adding a new location called Passage using the **npm run create-location passage** command. Give it a name:
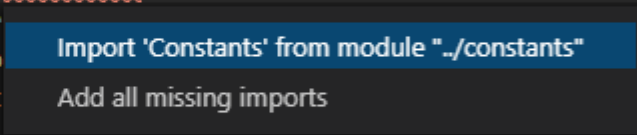
```typescript
export function Passage() {
    return Location({
        name: 'A passage in the undergrowth',
        description: description,
```

Also add a code file for a new feature, which will be the passage. This time, we'll use a stand-alone feature to demonstrate how that works. Use the **npm run create-feature corridor** snippet to create an empty feature and add the code below, using the **ssCombine** snippet to add the combine entry:

```typescript
import { IGame, Feature } from '../types';

export function Corridor() {
    return Feature({
        name: 'Corridor',
        description: 'A passage through the undergrowth',
        combinations: {
            combine: [
                {
                    combinationType: Constants.WALK,
                    match: (game, target, tool): string => {
                        game.changeLocation(Passage);
                        return 'You crawl through the passage.';
                    }
                },
            ]
        }
    });
}
```

Note that Visual Studio Code is reporting two errors here. We need to import the definitions of Constants and Passage. Do so by putting the cursor on them and pressing **CONTROL + '.'**:

33

```
            description: 'A passage through the undergrowth',
            combinations: {
                combine: [
                    {
                        combinationType: Constants.WALK,
                        match: (game, tar
                            game.changeLo    Import 'Constants' from module "../constants"
                            return 'You c    Add all missing imports
                        }
                    },
                ]
            }
        });
```

Select the import and press **Enter**. The red line should go away.

With the new location added, we can make the feature to travel to it available when the Touch combination on the fountain is triggered with a bit more code to the **Start.ts** file:

```
        combinationType: Constants.LOOKAT,
        match: (game, target, tool): string => {
            return 'You look at the fountain water. It is very clear and reflects the forest.';
        }
    },
    {
        combinationType: Constants.TOUCH,
        match: (game, target, tool): string => {
            if (!game.currentLocation.features.get('passage')) {
                game.currentLocation.features.push(Features.Passage);
                return `You walk towards the fountain and touch the fountain water.
                 It is a little cold. When you pull back your hand, you hear a soft
                 muttering. It is coming from a small passage in the undergrowth.`;
            }
            else {
                return 'The fountain water is pleasant to the touch.';
            }
        }
    },
```

Note how I used the backtick ('`') to break up the long text across multiple lines. As you see, the fountain inline feature is becoming big, so in order to organize your code you can also make a stand-alone feature out of it. Create a new code file called **fountain.ts**, move the feature code there and then add the feature to the start location like this:

```
    return Location({
        name: 'Start',
        features: [
            Features.Fountain()
        ]
    });
```

Resetting the world and touching the fountain, you should see this:

34

Touching the fountain, the player unlocked a new feature! In the code, I also made sure that nothing happens the second time you touch the water using the **if/else** block. If you touch the water again, you'll just see a message displayed.

With these simple interactions that do not use a tool in place, let's create a combination with the **Use** command to show an example of a combination that does use a tool. As a simple illustration, let's create a new feature for the **Passage** location with a Look combination that will give the player an item.

First, add an item using the **npm run create-item flask** command:

You can see that for items, an html file is generated as well. You can put a description for the item in there to have more options for formatting than simple text. Now we have a flask item in the game that we can use.

Modify the **Passage.html** file like this:
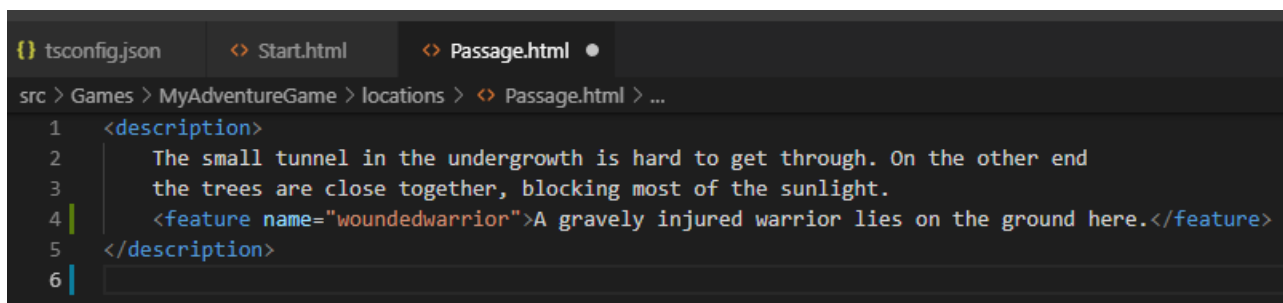
Then change the **Passage.ts** location file like this:

```
name: 'A passage in the undergrowth',
destinations: [
    {
        name: 'Back to the fountain',
        target: Locations.Start
    }
],
features: [
    {
        name: 'Wounded warrior',
        combinations: {
            failText: 'That won\'t help him.',
            combine: [
                {
                    combinationType: Constants.LOOKAT,
                    match: (game, target, tool): string => {
                        if (!game.character.items.get(Items.Flask)) {
                            game.character.items.push(Items.Flask);
                            return `Looking at the warrior, you see a flask on his belt.
                                    carefully, you remove it.`;
                        }
                        else {
                            return 'You see nothing else that might help.';
                        }
                    }
                }
            ]
        }
    }
]
```

Note that I added a destination to be able to return to the fountain from here. Reset your world and go to the Passage location. Look at the warrior. You should now receive a flask:

Backpack	You are at A passage in the undergrowth
Flask	The small tunnel in the undergrowth is hard to get through. On the other end the trees are close together, blocking most of the sunlight. A gravely injured warrior lies on the ground here.

Combinations	
Walk Use Touch Look	Look at Wounded warrior: Looking at the warrior, you see a flask on his belt. carefully, you remove it.

Great, we now have an item to use. We'll let the player fill the flask with fountain water to give to the warrior. Add a new **water** item for the fountain water and call it **Fountain water**:

```
export function Water() {
    return Item({
        name: 'Fountain water',
        equipmentType: EquipmentType.Miscellaneous,
        combinations: {
```

Then, add a new combination to the **fountain.ts** feature file:

```
        },
        {
            combinationType: Constants.USE,
            tool: Items.Flask,
            match: (game, target, tool): string => {
                var flask = game.character.items.get(Items.Flask);

                if (flask) {
                    game.character.items.remove(flask);
                    game.character.items.push(Items.Water);
                    return `You fill the flask with the clear fountain water.`;
                }
                else {
                    return 'The fountain water is pleasant to the touch.';
                }
            }
        }
    }
]
```

As we're now at the Passage location with no way to return to start, we also need to add an additional feature to be able to travel back. Add it to the **passage.ts** file like this:

```
},
{
    name: 'Passage back',
    combinations: {
        combine: [
            {
                combinationType: Constants.WALK,
                match: (game, target, tool): string => {
                    game.changeLocation(Locations.Start);
                    return 'You crawl back to the fountain.';
                }
            }
        ]
    }
}
```

And to the passage.html like this:

```
1    <description>
2        <p>
3            The small tunnel in the undergrowth is hard to get through. On the other end
4            the trees are close together, blocking most of the sunlight.
5            <feature name="woundedwarrior">A gravely injured warrior lies on the ground here.</feature>
6            <feature name="passageback">The way back is hard to see</feature>
7        </p>
8    </description>
```

Use '**Walk**' on the passage back to get back to the fountain. Try the '**Use**' combination with the flask and the fountain. You should see the Flask item replaced by a Fountain water item:

Backpack	You are at Start
Fountain water	You stand at the edge of a forest clearing. Tall trees border it on all sides, with thick undergrowth making the clearing hard to get to. In the center of the clearing is a fountain with a six-sided base.
	A passage through the undergrowth

Combinations

Walk	Use	Touch	Look	Use Flask on Fountain: You fill the flask with the clear fountain water.

Note that the order in which you select the tool and a target for combinations that require both matters in most cases. Only when using two items together will a combination be resolved irrespective of whether the tool or the target was selected first. Let's demonstrate this by adding one more item to wrap up before moving on to using combinations in a visual way.

Add a new **herbs** item and add this code:

```
4
5    export function Herbs() {
6        return Item({
7            name: 'Herbs',
8            equipmentType: EquipmentType.Miscellaneous,
9            combinations: {
0                combine: [
1                    {
2                        combinationType: Constants.TOUCH,
3                        match: (game, target, tool): ICombinationMatchResult => {
4                            game.character.items.push(Herbs);
5                            return {
6                                text: 'You collect the herbs.',
7                                removeTarget: true
8                            };
9                        }
0                    },
1                ]
2            }
3        });
4    }
```
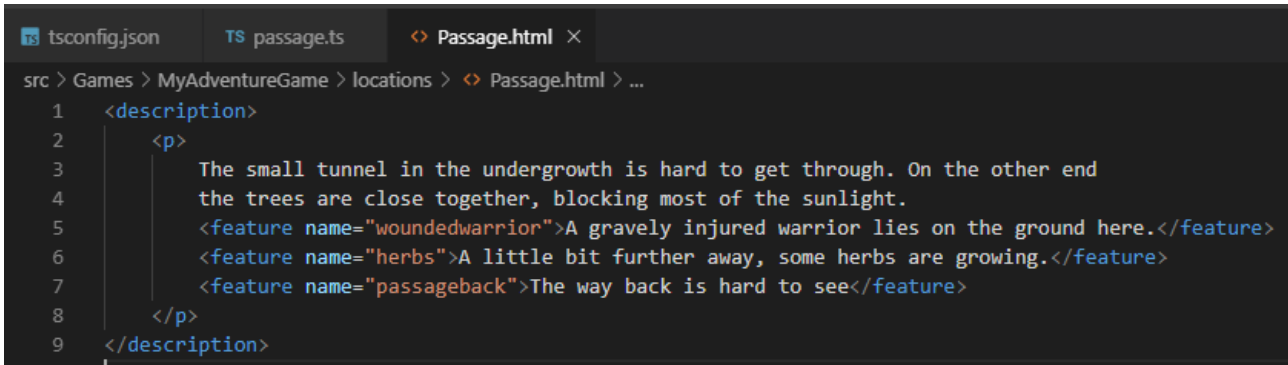
Fix any imports you might be missing. Note that I'm specifying that the feature should be removed when a successful touch combination match is made.

Now, add one more item, **healingPotion**:
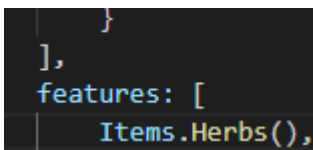
```
export function HealingPotion() {
    return Item({
        name: 'Healing potion',          You, 18 days
        description: description,
        equipmentType: EquipmentType.Miscellaneous
    });
}
```

Add the herbs to the **Passage.html** location like this:

```
TS tsconfig.json      TS passage.ts      <> Passage.html ×

src > Games > MyAdventureGame > locations > <> Passage.html > ...
  1    <description>
  2        <p>
  3            The small tunnel in the undergrowth is hard to get through. On the other end
  4            the trees are close together, blocking most of the sunlight.
  5            <feature name="woundedwarrior">A gravely injured warrior lies on the ground here.</feature>
  6            <feature name="herbs">A little bit further away, some herbs are growing.</feature>
  7            <feature name="passageback">The way back is hard to see</feature>
  8        </p>
  9    </description>
```

Then, add the herbs item to the **Passage.ts** as a new feature:

```
        }
    ],
    features: [
        Items.Herbs(),
```

Note the brackets after the item name, these are needed to instantiate a new herbs item for the location (there is a long technical history why you need them here and not when e.g. adding an item using push. You can also use the brackets when you're pushing a new item, but it doesn't make sense when removing one).

Then, add this combination to the **water.ts** file:

```
name: 'Fountain water',
equipmentType: StoryScript.EquipmentType.Miscellaneous,
combinations: {
    failText: 'You can\'t use the water like that',
    combine: [
        {
            combinationType: Constants.USE,
            tool: Items.Herbs,
            match: (game, target, tool): string => {
                game.character.items.remove(Items.Water);
                game.character.items.remove(Items.Herbs);
                game.character.items.push(Items.HealingPotion);
                return `You cut the herbs into small pieces and add them to the water.
                    This potion should help to heal wounds.`;
            }
        }
    ]
}
```

Start over. When you now go to the Passage location, you'll find the herbs there. Collect them using the Touch combination. When you do that the feature disappears, and you get the Herbs item:

Backpack
Fountain water

You are at A passage in the undergrowth
The small tunnel in the undergrowth is hard to get through. On the other end the trees are close together, blocking most of the sunlight. A gravely injured warrior lies on the ground here. A little bit further away, some herbs are growing. The way back is hard to see

Combinations	
Walk Use Touch Look	Touch

Backpack
Fountain water
Herbs

You are at A passage in the undergrowth
The small tunnel in the undergrowth is hard to get through. On the other end the trees are close together, blocking most of the sunlight. A gravely injured warrior lies on the ground here. The way back is hard to see

Combinations	
Walk Use Touch Look	Touch Herbs: You collect the herbs.

Get the flask, fill it with water and then use the Herbs on the Fountain water to create the Healing potion. You can try both ways, the order shouldn't matter:

Backpack
Healing potion

You are at Start
You stand at the edge of a forest clearing. Tall trees border it on all sides, with thick undergrowth making the clearing hard to get to. In the center of the clearing is a fountain with a six-sided base .
A passage through the undergrowth

Combinations	
Walk Use Touch Look	Use Fountain water on Herbs: You cut the herbs into small pieces and add them to the water. This potion should help to heal wounds.

.7   *Visual combinations*

With the elements of tutorial of chapter 6 in place, we can now look at how you can work with a more visual representation of your world using combinations. The example of what we'll be creating is at https://storyscript.azurewebsites.net/games/myadventuregamevisual/index.html.

We'll just add to the game we build in chapter 6 and show you how easy it is to make the game more visual once you have all the elements in place. Go to the **start.html** file and add a **visual-features** tag, like this (note that not all the area code is shown for readability):

```
        </p>
    </description>
    <visual-features img="fountain.jpg">
        <area name="fountain" coords="292,218,289,259,269,261,229,292,228,321,262,340,364,342,46
    </visual-features>        You, 18 days ago • Add adventure game example
```

The **fountain.jpg** I add to the resources folder. Go to your **maincomponent.html** file and change it like this:

```
    <div *ngIf="!game.state">
        {{ texts.loading }}
    </div>
    <div *ngIf="game.state == 'Play'">
        <!-- <location-text></location-text> -->
        <location-visual></location-visual>
    </div>
</div>
```

My browser now shows this:

To make the image interactive, I use the ImageMap html functionality. I can create an image map using a website like https://www.image-map.net/. Just upload an image and create some areas for your picture:
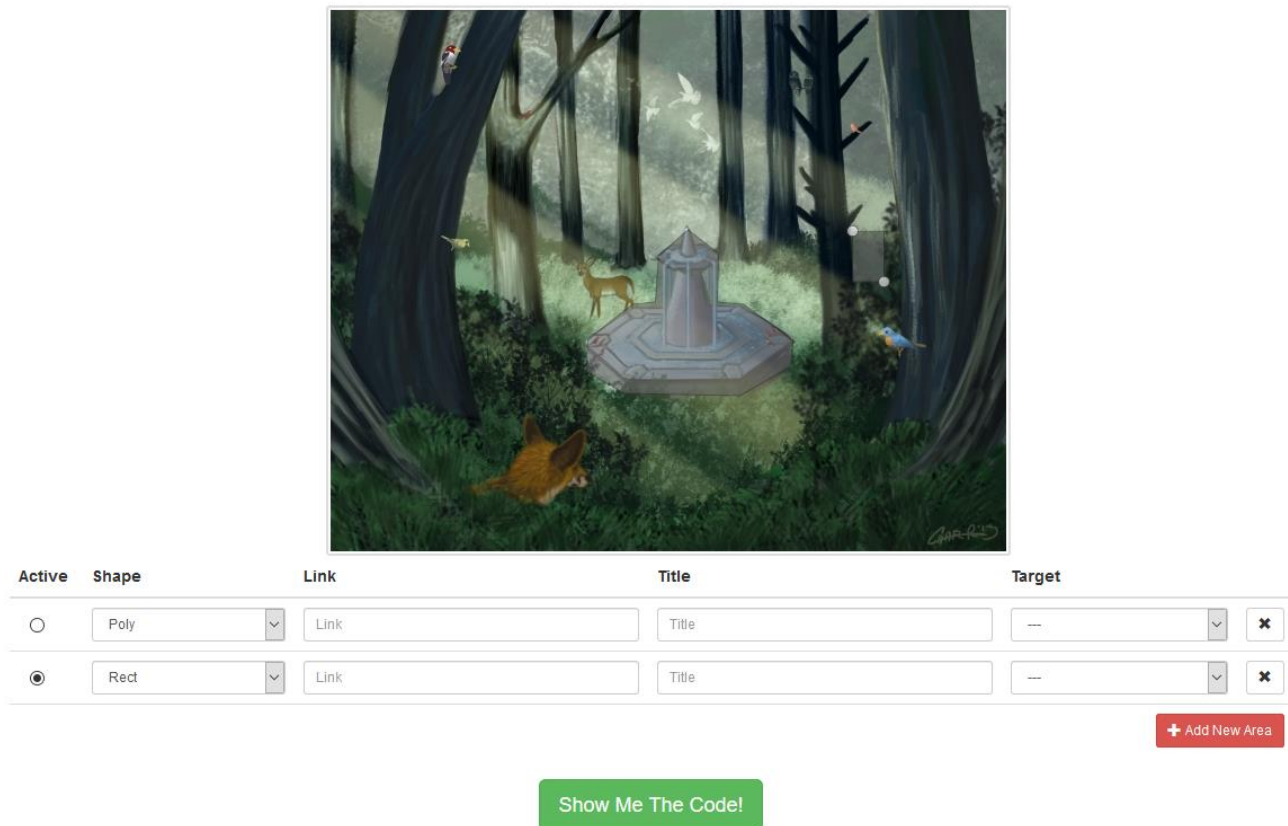
For now, I created two, one for the fountain and one for the passage through the undergrowth. Press the 'Show Me The Code!' button, right click the code shown and select all. Copy the code to a temporary location (e.g. notepad), and use the fountain polygon to set up the first visual feature in the start.html file like this (replace the existing contents):


```
src ▸ Games ▸ MyAdventureGameVisual ▸ locations ▸ <> Start.html ▸ 🔷 visual-features
  1    <visual-features img="fountain.jpg">
  2        <area name="fountain" coords="292,218,289,259,269,261,229,292,
  3    </visual-features>
```

In your browser, you should now see a pointer when you move the mouse over the areas you defined for your picture. And you should be able to interact with the fountain again, looking at it and touching it.

We used the **Touch** combination to add a new feature to the location. Let's change that a bit by making the second area we defined do something. Open the stand-alone feature **corridor.ts**, and add the **coords** and **shape** properties using the coordinates and type of the second, rectangular area we created with the image map tool:

```
export function Corridor() {
    return Feature({
        name: 'Corridor',
        description: 'A passage through the undergrowth',
        coords: '492,241,464,196',
        shape: 'rect',
        combinations: {
            combine: [
```

Refresh your browser and restart. At first, the passage isn't there. Only after you touched the fountain is it added and can you use the Walk combination to travel through the passage to the next location.

Let's make the Passage location visual as well. Add the code below to the **passage.html** file, using the **ssVisualFeatures** and the **ssArea** snippets:

```
{} tsconfig.json        <> Passage.html  ×

src ▸ Games ▸ MyAdventureGameVisual ▸ locations ▸ <> Passage.html ▸ 🔷 visual-features
  1   <visual-features img="fallenhero.jpg">
  2       <area name="woundedwarrior" coords="350,96,378,137,416,182,438,253,519,191,469,137" shape="poly">
  3       <area name="herbs" coords="265,440,385,479" shape="rect">
  4       <area name="passageback" coords="2,292,93,478" shape="rect">
  5   </visual-features>
```

Also add a picture for the **herbs.ts**:

```
export function Herbs() {
    return Item({
        name: 'Herbs',            You, 18 days ago • Add
        picture: 'herbs.png',
        equipmentType: EquipmentType.Miscellaneous,
        combinations: {
            combine: [
```

We now show the warrior on the ground with the herbs close to him. As before, you can pick up the herbs, they will disappear once collected:

From now on, you can cycle between the text and visual mode of the adventure by changing the active location component in your **maincomponent.html** file. You can also use both at the same time if you can think of a use.

Part 4. Role playing games
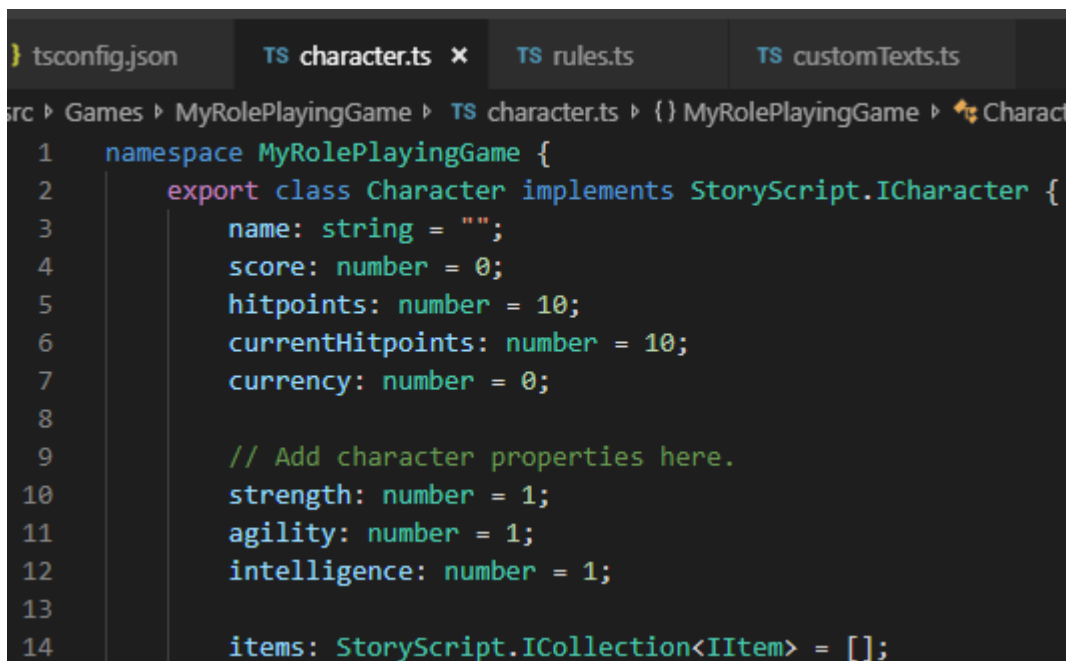
Now that you have a basic grasp of the first two types of games you can create with StoryScript, it is time to dive deeper and develop a basic understanding of the full power it has to offer. In terms of story and interaction, there is nothing more extensive to build than a Role Playing Game (RPG), in which the player interacts with the story world and its inhabitants and developing new skills and growing more powerful through these experiences.

Central to the RPG is the player character. There are plenty of RPGs in which you can assemble a party of characters, but in StoryScript (at least for now) there is one hero for the player to guide during his or her quest. So, we'll start by creating that hero.

Before you move on, create a new game like before so you can make a fresh start. Call it '**MyRolePlayingGame**' (also change the game name in the **customTexts.ts** file). You can find the example online at https://storyscript.azurewebsites.net/games/myroleplayinggame/index.html.

.8   Define your hero

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:

```
} tsconfig.json      TS character.ts  ×      TS rules.ts         TS customTexts.ts

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

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:

```typescript
    items: StoryScript.ICollection<IItem> = [];

    equipment: {
        head: IItem,
        body: IItem,
        leftHand: IItem,
        rightHand: IItem,
        feet: IItem,
    };

    constructor() {
        this.equipment = {
            head: null,
            body: null,
            leftHand: null,
            rightHand: null,
            feet: null
        }
    }
}
```

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). Find the getCreateCharacterSheet method and change it to this, then add the three steps listed below inside the steps square brackets. To do this quickly, you can use the **ssAttributes**, **ssAttribute** and **ssAttributeEntry** snippets to set up attribute steps and **ssQuestions**, **ssQuestion** and **ssQuestionEntry** snippets for question steps:

```typescript
getCreateCharacterSheet = (): StoryScript.ICreateCharacter => {
    return {
        steps: [

        ]
    };
}
```

```typescript
{
    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
                }
            ]
        }
    ]
}
```
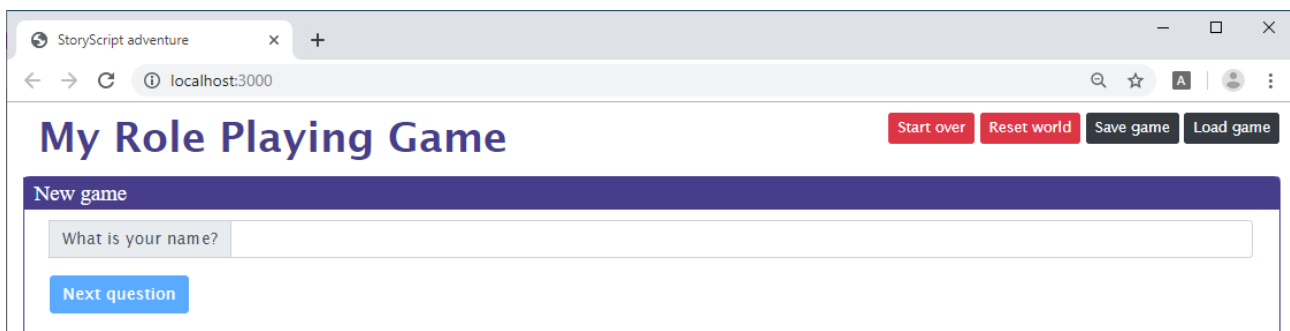
Note that in this example we're using questions to determine our character attribute values, adding bonuses depending on the answers chosen. Another way to set attributes is to use attributes. Here, we use an attribute only for name and we don't specify the minimum, maximum, or value properties. This allows the player to add a free text answer. If you want to set attributes using number inputs, you should also specify the min, max and value properties up front. Additionally, on the attributes property you can specify the **numberOfPointsToDistribute**. You can use this when you add attribute entries (**ssAttributeEntry** snippet) for the attributes in your game. The player will then be able to distribute points up to this number, starting from the value you specify for the attribute up front.

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:

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:

```ts
namespace MyRolePlayingGame {
    export class Rules implements StoryScript.IRules {
        getCombinationActions = () => {
            return [
                // Add combination action names here if you want to use this feature.
            ];
        }

        getSheetAttributes = () => {
            return [
                'strength',
                'agility',
                'intelligence'
            ]
        };

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

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

48

.9 Locations

There's much more you can do with locations than we've seen so far building the other games.

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' (remember to use the **ssDescription** snippet to add an additional description quickly). Our **Start.html** will then look like this:

```html
<description name="day">
    <p>
        You are at home. You can hear the birds singing in the garden.
    </p>
</description>
<description name="night">
    <p>
        You are at home. the frogs are croaking in the pond.
    </p>
</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:

```typescript
namespace MyRolePlayingGame.Locations {
    export function Start() {
        return Location({
            name: 'Home',
            descriptionSelector: (game: IGame) => {
                var date = new Date();
                var hour = date.getHours();

                if (hour <= 6 || hour >= 18) {
                    return 'night';
                }

                return 'day';
            },
        });
    }
}
```

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

You are at Home

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

Or:

You are at 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 two new locations, the hero's garden behind his home and the dirt road in front of it. You can do this by copying the Start location files, renaming them and changing the code inside, or create new files and add the minimum code needed (see below). NOTE: when copying, the file watcher can crash. If that happens the console will show a message like" *The terminal process terminated with exit code: 1*". If that happens, simply complete your changes and restart using **CTRL-SHIFT-B**. The result should look like this:

```
▲ locations
   <> DirtRoad.html
   TS DirtRoad.ts
   <> Garden.html
   TS Garden.ts
   <> Start.html
   TS start.ts
```

50
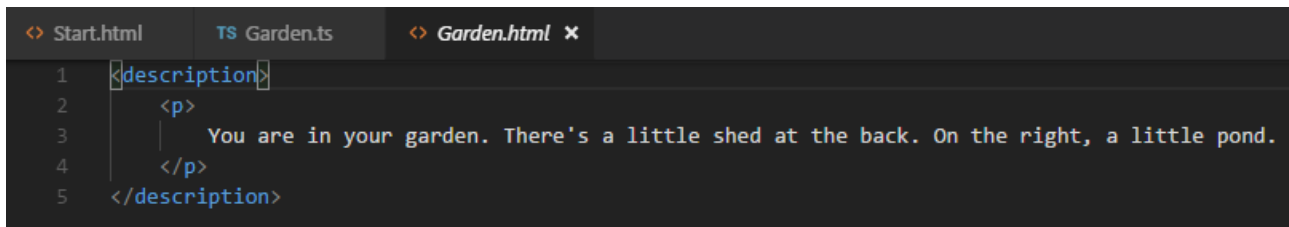
Note that you can should change the function names to match the file names for clarity (not required, you can select another name as long as it does not have spaces or special characters). E.g., your **Garden.ts** should look like this:

```
{} tsconfig.json        TS rules.ts        TS Garden.ts  ✕
1    namespace MyRolePlayingGame.Locations {
2        export function Garden() {
3            return Location({
4                name: 'Garden'
5            });
6        }
7    }
```

Next, change your .html files to add some proper descriptions. We use just one description for now. My **Garden.html** looks like this:

```
<> Start.html        TS Garden.ts        <> Garden.html  ✕
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 first destination to the destinations array by using the **ssDestination** snippet. 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:

```
{} tsconfig.json    TS rules.ts    TS Garden.ts    TS DirtRoad.ts    TS start.ts    ●    <> Garden.html    <> Dirt

1    namespace MyRolePlayingGame.Locations {
2        export function Start() {
3            return Location({
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                   }                        🔷 DirtRoad  function MyRolePlayingGame.Locations.D… ⓘ
20               ],                           🔷 Garden
21           });                             🔷 Start
22       }
23   }
```

When you added both destinations, your destinations code looks something like this (note that you do NOT use brackets after the location name, you're not instantiating a new location, just referring to one):

```
destinations: [
    {
        name: 'To the garden',
        target: Locations.Garden
    },
    {
        name: 'Out the front door',
        target: Locations.DirtRoad
    }
],
```

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

You are at Home

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

On the ground

Destinations
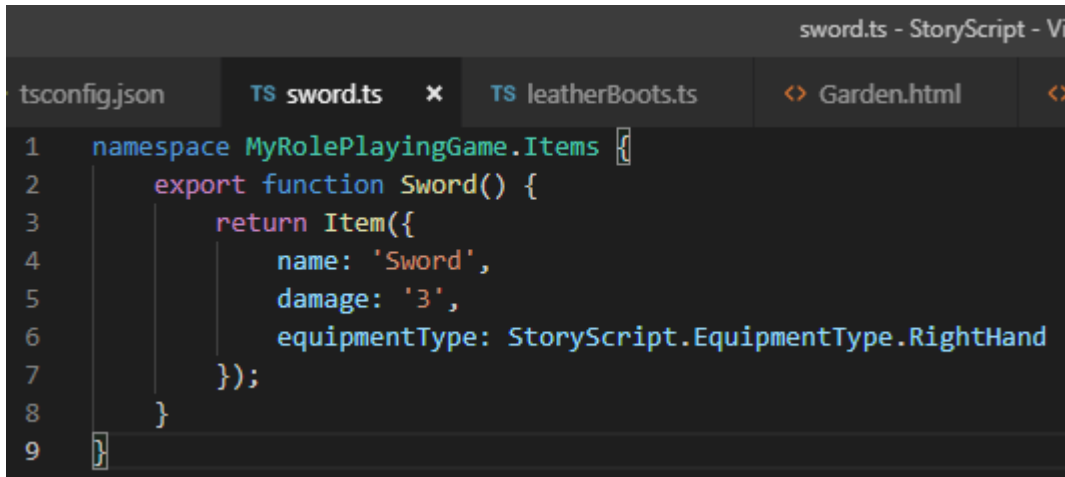
[To the garden] [Out the front door]

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

52

around your three-stage world.

.10 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:
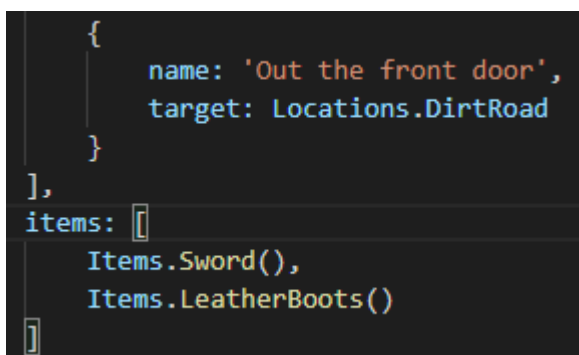
```typescript
namespace MyRolePlayingGame.Items {
    export function Sword() {
        return Item({
            name: 'Sword',
            damage: '3',
            equipmentType: StoryScript.EquipmentType.RightHand
        });
    }
}
```

```typescript
namespace MyRolePlayingGame.Items {
    export function LeatherBoots() {
        return Item({
            name: 'Leather boots',
            defense: 1,
            equipmentType: StoryScript.EquipmentType.Feet
        });
    }
}
```

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:

```typescript
        {
            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 at Home

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

On the ground

Leather boots
Sword

Destinations

To the garden   Out the front door

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

.11 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, using the **ssFunction** snippet to add an event function:

```
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':

Feet

Messages

You see a squirrel running off.

54

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.

.12 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. Use the **ssEnemy** snippet. It should look like this:

```
namespace MyRolePlayingGame.Enemies {
    export function Bandit() {
        return Enemy({
            name: 'Bandit',
            hitpoints: 10,
            attack: '1d6',
            items: [
                Items.Sword(),
            ]
        });
    }
}
```

Note two things. First, 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. Second, the **'attack'** property is not a property available by default in StoryScript. All entities, like locations and enemies, have some default properties you can use, but you can also easily add your own to suit your game.

A note about discovering the StoryScript API

As this is the first time we're adding a new property to one of the building blocks of StoryScript, it is worthwhile explaining how you can see all the properties and functions available out of the box. You already worked with Intellisense extensively, showing in the suggestions properties and classes available that match what you're typing. There is also another way in which you can discover what StoryScript has out of the box and what it's API looks like. That is by using F12 to view type definitions.

In the picture above, placing your cursor over the Enemy on the third line and pressing F12 will take you to the definition of the enemy class for your game:

```
export function Enemy(entity: IEnemy): IEnemy {
    return StoryScript.Enemy(entity);
}

export interface IEnemy extends StoryScript.IEnemy {
    // Add game-specific enemy properties here
}
```

From here, From there, you can do the same to go into the StoryScript.IEnemy definition, the

interface describing all the default properties of an enemy:

```
interface IEnemy extends IFeature {
    /**
     * The details about this enemy as displayed to the player
     * will be used to set this property at run-time.
     */
    description?: string;
    /**
     * The health of the enemy.
     */
    hitpoints: number;
    /**
     * The amount of credits the enemy has, in whatever form.
```

Ok, now that we know how to inspect the default properties and we can see there is no property for attack, let's add a new property **'attack'** that is required on all the enemies in the game. Open the **enemy.ts** file in the interfaces folder and change it like this:

```
{} tsconfig.json      TS enemy.ts   ×   TS DirtRoad.ts

src ▸ Games ▸ MyRolePlayingGame ▸ interfaces ▸ TS enemy.ts ▸ {} MyRolePlayingGame
   1    namespace MyRolePlayingGame {
   2        export function Enemy(entity: IEnemy): IEnemy {
   3            return StoryScript.Enemy(entity);
   4        }
   5
   6        export interface IEnemy extends StoryScript.IEnemy {
   7            // Add game-specific enemy properties here
   8            attack: string;
   9        }
  10    }
```

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

```
    name: 'Dirt road',
    destinations: [
        {
            name: 'Enter your home',
            target: Locations.Start
        }
    ],
    enemies: [
        Enemies.Bandit()
    ]
```

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

You are at 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

.13 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: IEnemy, retaliate?: boolean) => {
    var self = this;
    retaliate = retaliate == undefined ? true : retaliate;

    // Implement character attack here.

    if (retaliate) {
        game.currentLocation.activeEnemies.filter((enemy: IEnemy)
            // Implement monster attack here
        });
    }
}
```

Modify it like this:

```
fight = (game: IGame, enemy: IEnemy) => {
    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: IEnemy) => { 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

57

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

.14 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**, using the **ssAction** snippet:

```
        ],
        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 at Garden

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

On the ground

Actions

Search the Shed   Look in the pond

Destinations

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** (you can use the same **ssAction** snippet):

```
11              enemies: [
12                  Enemies.Bandit
13              ],
14              combatActions: [
15                  {
16                      text: 'Run back inside',
17                      execute: (game: IGame) => {
18                          game.changeLocation('Start');
19                          game.logToActionLog(`You storm back into your house and slam the
20                                                door behind you. You where lucky... this time!`);
21                          return true;
22                      }
23                  }
24              ]
25          }
```

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:

.15 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. Use the **ssBarrier** snippet to add the barrier and then the **ssAction** snippet to add the actions:

60

```
name: 'Enter the basement',
target: Locations.Basement,
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.');
            }
        },
        {
            name: 'Open',
            action: StoryScript.Actions.Open((game, destination) => {
                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:
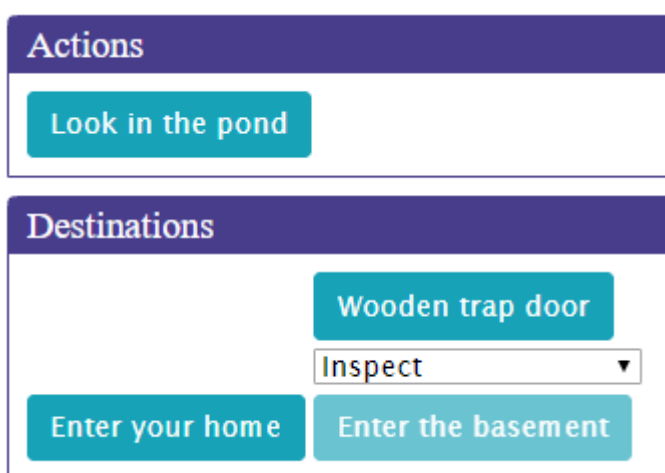
Clicking on the trap door will trigger the action selected in the dropdown. When you leave it at 'Inspect', a message will be written to the location description and the Inspect action will be removed. When the Open action is chosen, the barrier will be removed, including any actions remaining.

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 the chapter 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:

Right. Barriers really make the world more interesting and interactive, but we can add some more 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 key and call it Basement Key (file basementKey.ts, use the **ssKey** snippet):

```typescript
namespace MyRolePlayingGame.Items {
    export function BasementKey() {
        return Key({
            name: 'Basement key',
            open: {
                name: 'Open',
                action: StoryScript.Actions.OpenWithKey((game: IGame, destionation: StoryScript.IDestination) => {
                    game.logToLocationLog('You open the trap door. A wooden staircase leads down into the darkness.');
                })
            },
            equipmentType: StoryScript.EquipmentType.Miscellaneous
        });
    }
}
```

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:

```typescript
name: 'Enter the basement',
target: Locations.Basement,
barrier: {
    name: 'Wooden trap door',
    key: Items.BasementKey,
    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:

```
namespace MyRolePlayingGame.Enemies {
    export function Bandit() {
        return Enemy({
            name: 'Bandit',
            hitpoints: 10,
            attack: '1d6',
            items: [
                Items.Sword(),
                Items.BasementKey()
            ]
        });
    }
}
```

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:

Rutger

Strength 3
Agility 1
Intelligence 1
Health 7 / 10
Money 0

Equipment

Head

Right   Body   Left
hand   hand
Sword
Feet
Leather boots

Backpack

Sword Equip Drop

Basement key Drop

You are at Garden

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

On the ground

Actions

Look in the pond

Destinations

Wooden trap door

Inspect ▾
Inspect
Enter your home Open

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:

```
namespace MyNewGame.Items {
    export function BasementKey(): StoryScript.IKey {
        return {
            name: 'Basement key',
            keepAfterUse: false,
            open: {
                name: 'Open',
                action: StoryScript.Actions.OpenWithKey((game: IGame, destionation: StoryScript.IDestination) => {
                    game.logToLocationLog('You open the trap door. A wooden staircase leads down into the darkness.');
                })
            },
            equipmentType: StoryScript.EquipmentType.Miscellaneous
        }
    }
}
```

.16 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.
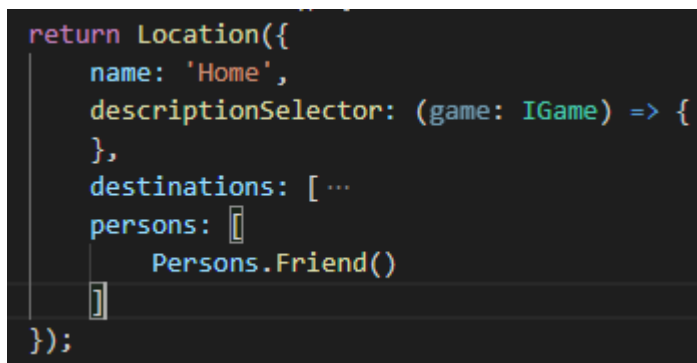
Add this code to the **friend.ts** file, using the **ssPerson** snippet:

```
namespace MyNewGame.Persons {
    export function Friend(): IPerson {
        return {
            name: 'Joe',
            hitpoints: 10,
            attack: '1d6',
            currency: 10,
            conversation: {
            }
        }
    }
}
```
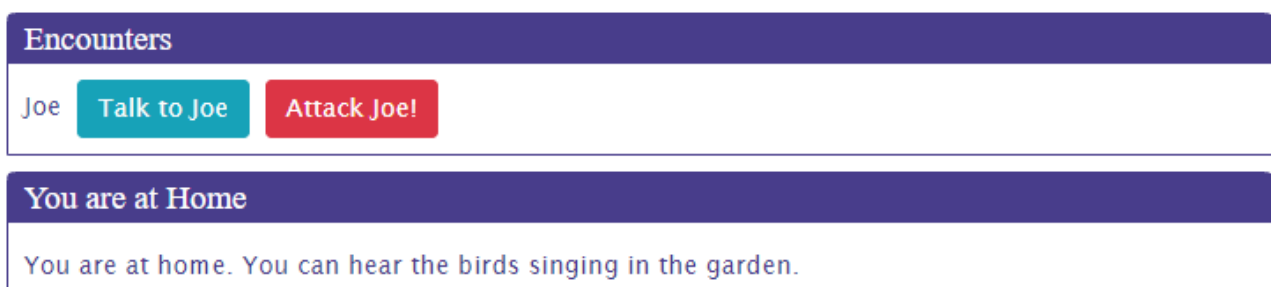
Although the conversation property is now an empty object, it is important that it is there nonetheless as the presence of this property will determine whether or not the engine will try to load the conversation file.

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

```
return Location({
    name: 'Home',
    descriptionSelector: (game: IGame) => {
    },
    destinations: [ ···
    persons: [
        Persons.Friend()
    ]
});
```

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

Encounters		
Joe	Talk to Joe	Attack Joe!

You are at Home

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

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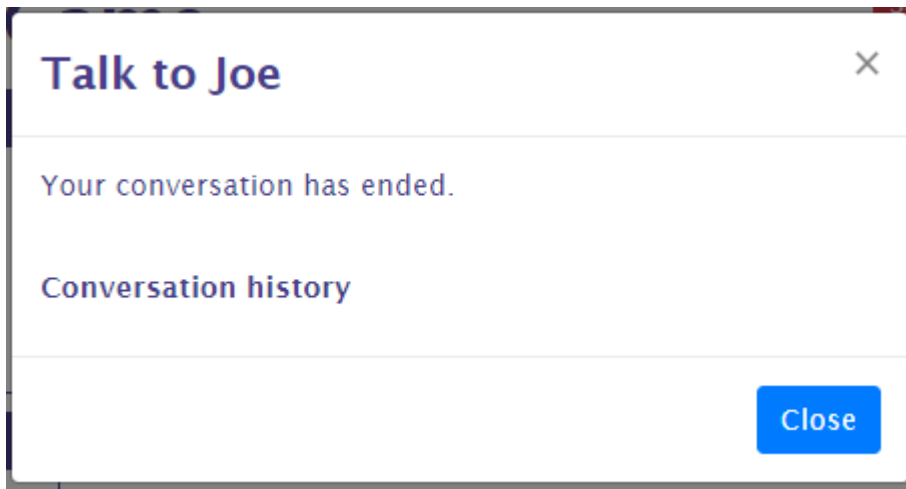
Making Joe a bit more of a conversationalist is the topic of the next chapter. You can interact with persons in more ways, by trading with them or doing assignments (quests) for them. These will be covered in later chapters as well.

.17 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. Use the **ssConversation**, **ssNode** and **ssReply** snippets to do this:

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>**

65

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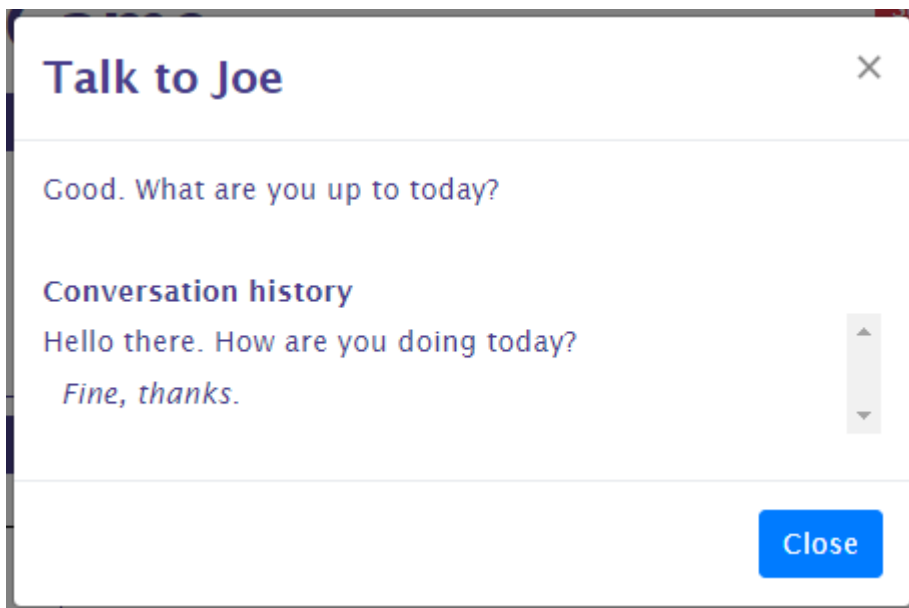
You see that the words exchanged are logged, so you can always review what's been said in case you forget.

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:
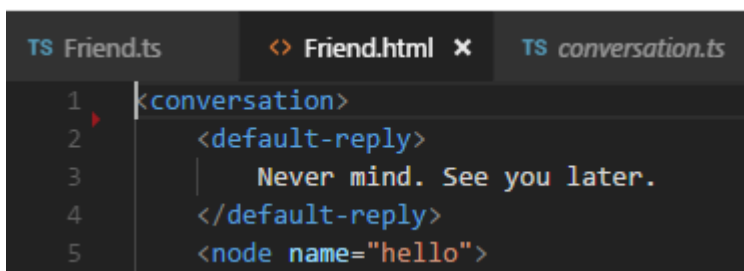
Great, some interactivity! You can expand on this conversation by adding additional nodes and replies linking to these nodes to get a basic conversation going. That's nice, but it wouldn't it be great if the conversation is actually influenced by the things going on in the world around you?
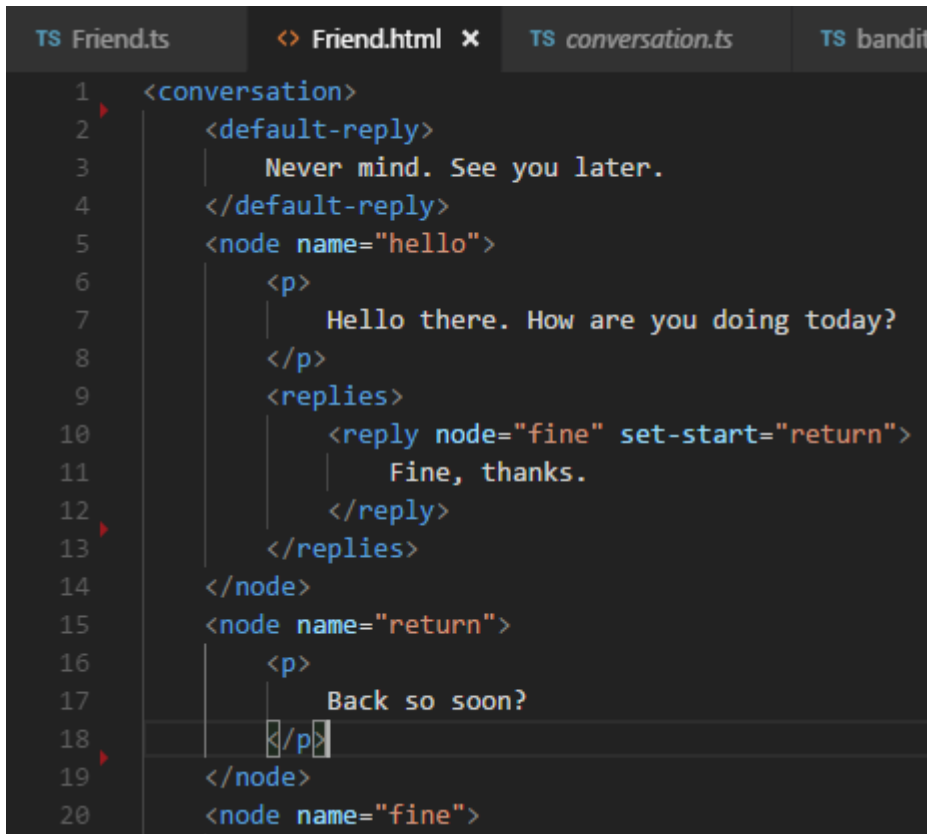
Well, of course we have some options to do this (otherwise it would not make for much of an interactive story). They are:

- 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:

```html
<conversation>
    <default-reply>
        Never mind. See you later.
    </default-reply>
    <node name="hello">
        <p>
            Hello there. How are you doing today?
        </p>
        <replies>
            <reply node="fine" set-start="return">
                Fine, thanks.
            </reply>
        </replies>
    </node>
    <node name="return">
        <p>
            Back so soon?
        </p>
    </node>
    <node name="fine">
```

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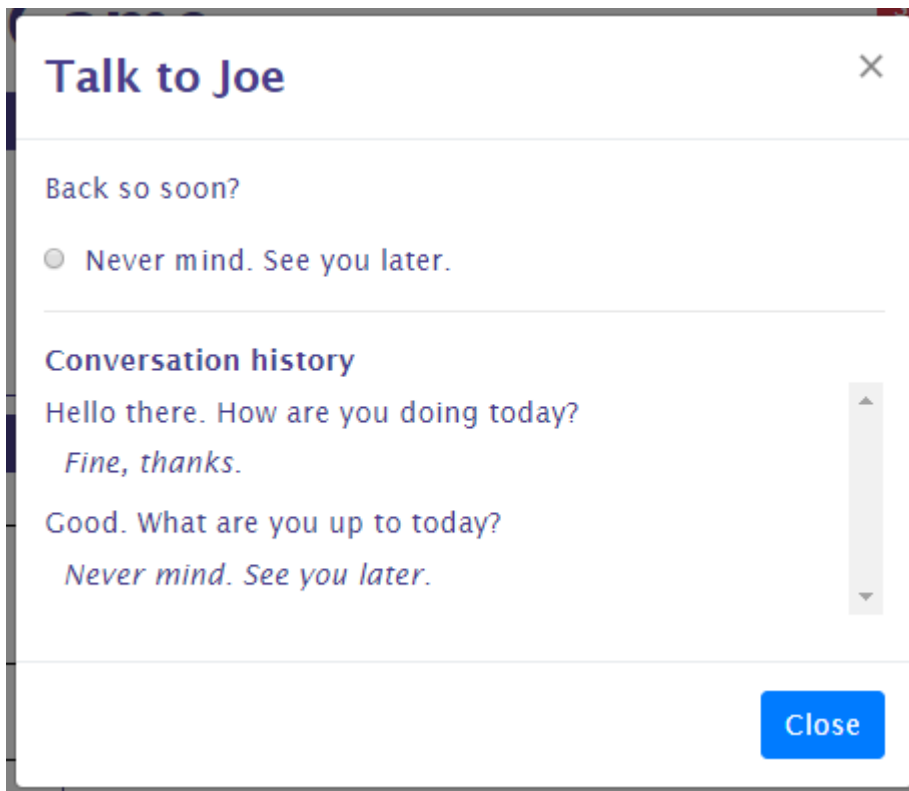
Conditionally available replies

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 several 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:

```html
 5          </default-reply>
 6          <node name="hello">
 7 ⊞          <p> ⋯
 9            </p>
10            <replies>
11                <reply node="fine" set-start="return">
12                    Fine, thanks.
13                </reply>
14                <reply node="workout" requires="strength=3">
15                    Great. I have just completed my workout.
16                </reply>
17            </replies>
18          </node>
19          <node name="return">
20 ⊞          <p> ⋯
22            </p>
23            <replies>
24                <reply node="garden" requires="location=garden">
25                    I just walked through the garden.
26                </reply>
27                <reply node="key" requires="item=basementkey">
28                    I found a key.
29                </reply>
30            </replies>
31          </node>
32          <node name="workout">
33            <p>
34                I can see that.
35            </p>
36          </node>
37          <node name="garden">
38            <p>
39                Did you see the hedgehog?
40            </p>
41          </node>
42          <node name="key">
43            <p>
44                Hm. I don't know what lock that's for.
45            </p>
46          </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. 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:
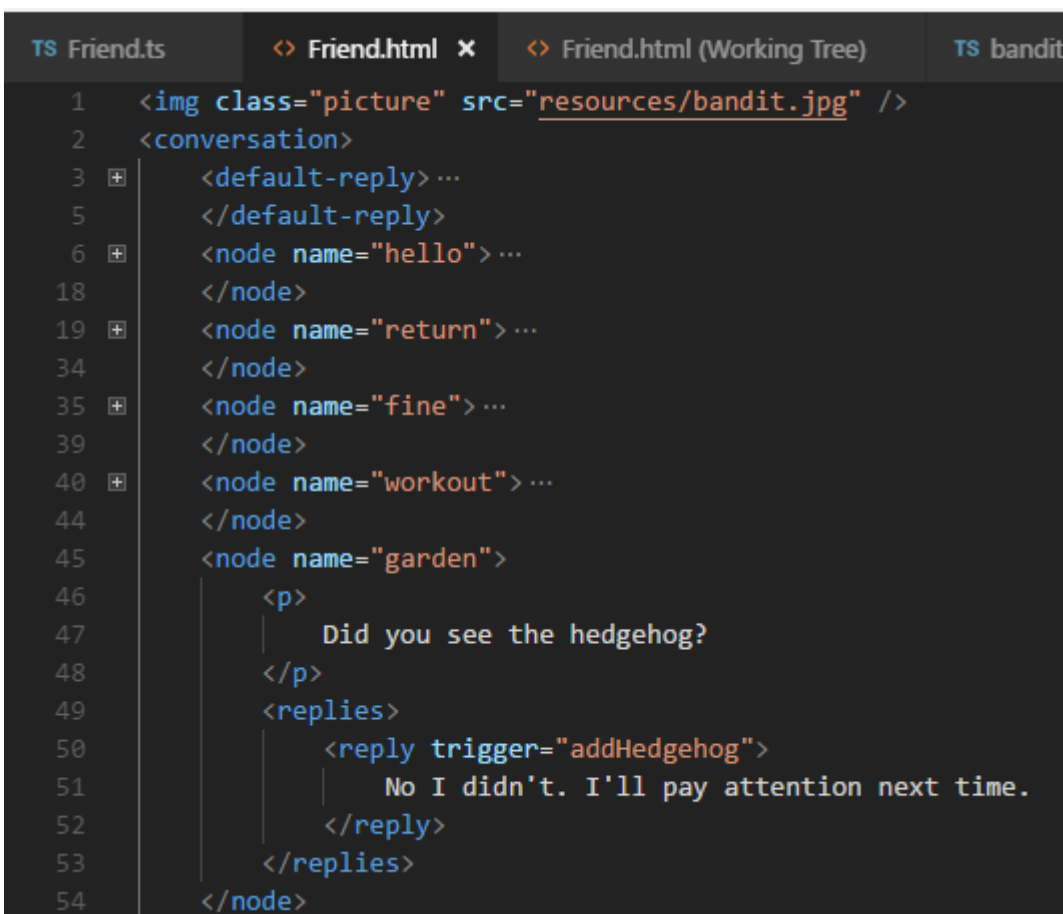
```
    },
    conversation: {
        showUnavailableReplies: true
    },
```

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:

```
TS Friend.ts          <> Friend.html ✕        <> Friend.html (Working Tree)        TS bandit
    1       <img class="picture" src="resources/bandit.jpg" />
    2       <conversation>
    3  ⊞        <default-reply>...
    5        </default-reply>
    6  ⊞        <node name="hello">...
    18       </node>
    19 ⊞        <node name="return">...
    34       </node>
    35 ⊞        <node name="fine">...
    39       </node>
    40 ⊞        <node name="workout">...
    44       </node>
    45        <node name="garden">
    46            <p>
    47                Did you see the hedgehog?
    48            </p>
    49            <replies>
    50                <reply trigger="addHedgehog">
    51                    No I didn't. I'll pay attention next time.
    52                </reply>
    53            </replies>
    54        </node>
```

Second, we specify what the addHedgehog function does in the .ts file, use the **ssConversationAction** snippet):

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

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

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 add the new event to the Garden, writing a simple message to the location log (enter and leave events are removed after they complete, hence there is no need to clean up the squirrel event that we defined before).

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

.18 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.

Create a new location for the bedroom with a .ts and .html file. Open the **bedroom.ts** file and add the following code:

```
{} tsconfig.json        TS bedroom.ts  ×     TS start.ts          TS bandit.ts

src ▷ Games ▷ MyRolePlayingGame ▷ locations ▷ TS bedroom.ts ▷ {} MyRolePla
   1    namespace MyRolePlayingGame.Locations {
   2        export function Bedroom() {
   3            return Location({
   4                name: 'Bedroom',
   5                destinations: [
   6                    {
   7                        name: 'Back to the living room',
   8                        target: Locations.Start
   9                    }
  10                ]
  11            });
  12        }
  13    }
```

After destinations, add the code for trading, using the **ssTrade** snippet. Note that the priceModifier

is optional and not part of the snippet, you can easily add the property yourself and use the **ssFunction** snippet to add a modifier function:

```
        target: Locations.Start
    }
],
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;
        }
    }
}
}
```

Link the bedroom to your home by adding a destination to the **Start.ts** file.

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:

On the ground

Actions

Your personal closet

Destinations

Back to the living room
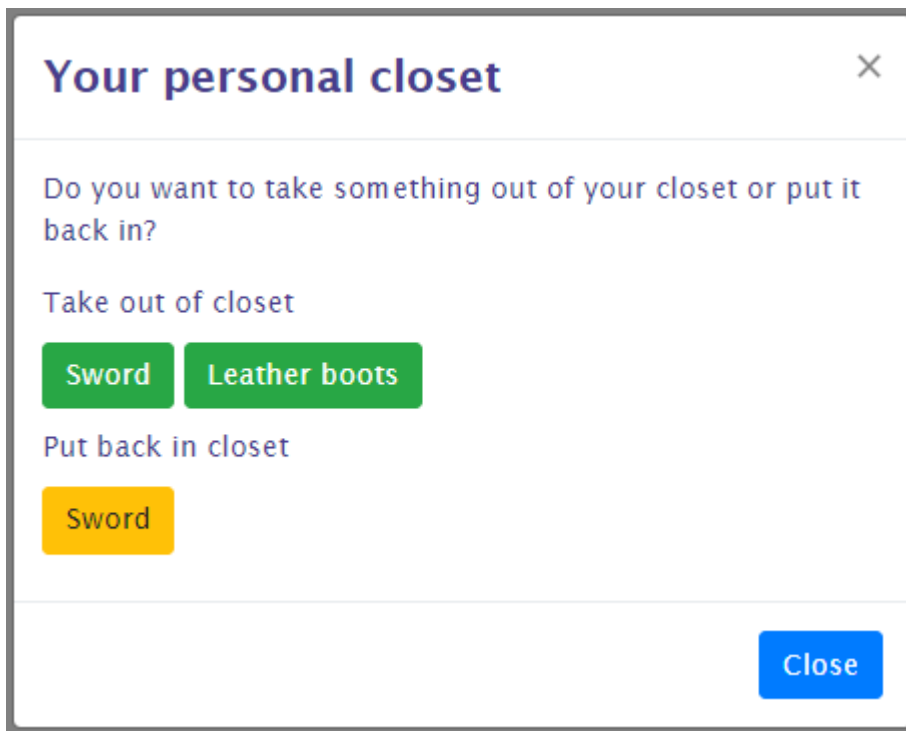
Ok, let's explore the options that are not obvious. First, you have both a buy and a sell side to the trade. This is defined from the perspective of the user, so the sell part is about the player 'selling' stuff to the trade object (in this case just putting his gear in the closet). Apart from the maxItems 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: {
            ownItemsOnly: true,
            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: {
```

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

Now you should be able to trade with Joe:
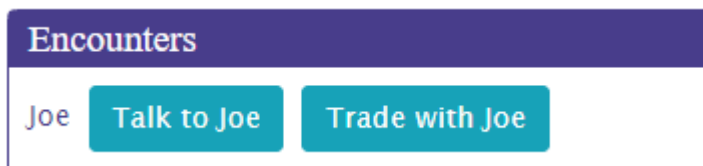
Some last things of note on trading are:

- 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.

.19 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, using the **ssQuest** snippet:

```typescript
namespace MyRolePlayingGame.Quests {
    export function Journal() {
        return Quest({
            name: "Find Joe's journal",
            status: (game, quest, done) => {
                return 'You 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) => {
                game.character.items.remove(Items.Journal);
                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 (NOTE: removing items in this way, using the item function reference instead of an actual object, will remove the first item of the specified type only. E.g. if the player would have two journals, only one would be removed. In cases like these, first get a specific item and them pass the actual item object to the remove function).

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

```typescript
namespace 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:

```
<conversation>
⊞    <default-reply>⋯
     </default-reply>
⊞    <node name="hello">⋯
     </node>
     <node name="return">
         <p>
             Back so soon?
         </p>
         <replies>
⊞            <reply node="garden" requires="location=garden">⋯
             </reply>
⊞            <reply node="key" requires="item=basementkey">⋯
             </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 name="key">⋯
     </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" set-start="return">
                 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 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:

```
    }
},
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:

```
name: 'Basement',
destinations: [
    {
        name: '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:

My Role Playing Game

Rutger

Strength 3
Agility 1
Intelligence 1
Health 8 / 10
Money 0

Equipment

Head

Right   Body   Left
hand   hand

Feet

Backpack

Joe's journal [Drop]

Quests

Current

- Find Joe's journal
  You have found Joe's journal!

Messages

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. It is very important to me. 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.

[Close]

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.

.20 Fleshing out items and enemies

You can also add images to enemies and items. To add an image to the bandit, open the **Bandit.ts** file and add the picture property like this:

```
namespace MyRolePlayingGame.Enemies {
    export function Bandit() {
        return Enemy({
            name: 'Bandit',
            picture: 'bandit.jpg',
            hitpoints: 10,
```

The **bandit.jpg** file needs to be present in the resources folder as well. Adding images to items works in the same way. Go out on the road. The bad guy should now have a face!

In order to give your items and enemies a bit more substance, you can also use an HTML file instead of a plain text description and a picture. With the HTML file, you can make them as flashy as you want. Once you're using an HTML file, it makes little sense to have to specify a picture for your enemy separately as it is so easily added there. If you add an image tag with the '*picture*' class, StoryScript will take that image and show it for the enemy (it works the same for items, note that here you do include the resources folder in the src):

```html
<img class="picture" src="resources/bandit.jpg" />
<description>
    A bandit
</description>
```

Part 5. Releasing your game and beyond

.21 Releasing your game

When you are happy with the game you've created and would like others to play it, it's time to release. There are (at least) three ways in which you can make it available:

1. Distribute your game to your friends so they can play it on their local computer. This is easy to do but cumbersome. Just run the 'publish-game-local' task and then zip the 'pub' folder in your StoryScript directory. It contains all the files the game needs. Anyone can play the game by unzipping the archive on their machine and then opening the index.html file with a browser.

2. Host your game yourself. You'll need a (virtual) webserver for this. Cloud services like Microsoft Azure make it easy to set one up. You can find plenty of info on how to do this elsewhere. When you have a server, you can just copy the contents of the 'pub' folder to a web-accessible location on your server. Then, you share the link to the game with your friends.

3. Host the game on the StoryScript website. The website is still extremely basic, but I have plans to create a nice portal from which to access a variety of games created with StoryScript. It requires no knowledge on your side, but some time on mine (or whomever else helps me with the website). There are a couple of things to do:

   a. You'll have to add some information on your game to the gameinfo.json file in your game folder. At the very least, you need to add the name and description of the game and the name of the author.

   b. You run the 'publish-game' task, then zip the contents of the 'pub' folder in your StoryScript directory and share it with us (e.g. via DropBox or OneDrive).

   c. **(Not yet, this is a future plan)** We'll review the game and then add it to the website. The information you added in a. will then be shown in the game list. Your friends can find it themselves using the list or you share a direct link with them.

.22 Additional concepts

To learn all there is about StoryScript, you'll need to study the API, experiment and look at the example games available to pick up ideas on how things can be done. There are a couple of things that are worth mentioning before letting you figure out the rest for yourself, though. These will probably make your journey a bit easier.

The inactive flag

The first is the inactive flag. This flag is available for items, enemies, persons and destinations. Instead of creating a new destination inline like we did in the tutorials so far, you can also define one in the regular way but set this flag to false. Then, the destination will not be shown in the game until you set it to true from your code. In the same manner, you can populate your locations with persons, enemies and items when designing the game, but make some of them inactive and only activate them when certain conditions have been met. When you do this, you can for example implement a day/night cycle in your game, with some enemies that are only active at night and some only during the day.

Showing messages to players

You've seen in the tutorials how to show messages to the player. There are three logs that you can write to, which show information in different places and which are also treated differently when e.g. refreshing the browser or loading the game:

- LocationLog: using the **game.logToLocation** function, you can write messages to the location description shown to the player. The idea is that you use this log to show that things in the location have changed permanently. These messages will stay when the browser refreshes.

- ActionLog: with **game.logToActionLog**, you write to the action log that should be used to show the player all the actions he's done. That can be going from one location to the next, picking up an item, starting a fight, etc. This log is cleared at browser refreshes.

- CombatLog: use **game.logToCombatLog** to show messages to the player during combat. The log will be cleared when combat is done.

- ConversationLog: a bit of a special log, this one is not available on the game object. StoryScript uses this log to keep track of what's been said during a conversation. This log is kept when the browser refreshes. You can get to it, though, by accessing the conversation object on a person in a location, if you really want to.

Rules functions

In your rules.ts, you can specify quite a few hooks to customize what happens when certain events happen during the game. For example, you can run some additional code when an enemy is defeated, when the game is started or when a location is entered. Check out the IRules definition to see what hooks are available.

Helper functions

On the game object that is available in every function you can use, there is a helpers object that has some useful functions. Right now, you can use these to get a random number by rolling some dice, calculate the total bonus for a character attribute taking into account his equipment, and getting random enemies and items.