

Assignment 4

The first part of this assignment gives you hands-on experience in **HTTP**. In the second part you will make a head start with the design of your **board game web application** (which you will further develop in assignments 5 and 6).

0. Preliminaries

Remember that this is a group assignment! Work efficiently as a team! If you have not programmed as a team before, read up on our introduction to [Visual Studio Code](#). Use this assignment to set up a collaborative coding environment within your team. Assignment 5 will require extensive JavaScript programming that both team members need to contribute to.

Overview of deliverables and upload procedure

If you get lost within the assignment, use this overview of deliverables to get back on track!

Task	Deliverables
1.1	HTTP requests
1.2	Answer Q1.2
1.3	Answer Q1.3
1.4	Answer Q1.4
2.1	Answer Q2.1
3.1	Answer Q3.1
3.2	Answer Q3.2
4.1	Chosen game type
4.2	Four annotated game screens (include the game URL)
4.3	Description of six game features
5.1	Splash screen design (wireframe)
5.2	Game screen design (wireframe)
5.3	→→→ upload 5.1/5.2 to 💡 Brightspace forum
6	Two html files

All deliverable text/imagery (apart from 6. which are two html files) must be included in a single PDF file. The first page of this PDF must contain the names and student numbers of the two team members as well as the team name.

The PDF and code have to be uploaded by one of the team members to 💡 Brightspace under **CSE Web assessment** (find the category your group belongs too) before the assessment session with the TAs and before

the ultimate assessment deadline. This means that the outcomes of Assignment 4, 5 and 6 are **all** uploaded to the same directory! Make sure to name your files with an **A4** prefix!

To pass this assignment, you must have completed all tasks and be able to answer the questions of the TAs.

1. HTTP request messages: GET/HEAD

👉 Hints:

- To store **telnet**'s output to file (in addition to printing it on the console), you can use the command **tee**, e.g. **telnet www.ard.de 80|tee out** will save all output to a file called **out**.
- To exit a telnet session, first press **CTRL+J**. This brings you to a **telnet>** prompt and you can type **close** to end the session.
- Telnetting **www.domain.com** is usually not the same as telnetting **domain.com**. Make sure your **host** header information matches exactly the domain you telnetted into.
- **Carriage return** in the code snippets below indicates when an empty line is expected. Press **<Enter>** to add it.
- Be aware of the **backspace key** when **telnetting**: while on a normal command line a backspace deletes the last character typed, within the **telnet** environment this key may be forwarded to the server instead. Be aware!
- This exercise requires you to use **telnet**. If you use a Linux derivative (e.g. Ubuntu, older versions of Mac OS), open a terminal and you are good to go; for new Mac OS versions you may need to **install telnet** yourself.
- If you are a Windows user, use the Windows Subsystem for Linux. As an alternative (if you really do not want to use Linux), use **PuTTY** with the following settings:
 - Use the **Raw** connection type (not **Telnet**).
 - For **Close window on exit**, use **Never**.
 - It may be useful to write your commands inside an editor first, and paste them by clicking the right mouse button inside the PuTTY session (which you start using the **Open** button).

⚠ ⚠ ⚠ Exercise (1) comes in two flavours: if you completed this part before January 1, 2020 you will have telnetted to **weer.nl. As of January 1, the website no longer works as required for this exercise and thus we offer a second variant that asks you to telnet into **surfweer.nl**.** The remaining exercises are unaffected from this change!

Exercise weer.nl (valid until January 1, 2020): Use **telnet** to request the contents of the Dutch rainfall radar section of the **www.weer.nl** website: www.weer.nl/regenradar/nederland. Start your **conversation** with the web server by typing the following into the terminal, and then perform HTTP requests to fetch the contents:

```
telnet www.weer.nl 80
```

Exercise surfweer.nl: Use **telnet** to request a list of webcams pointed at Dutch beaches from the **surfweer.nl** website: <http://surfweer.nl/surf/webcams/>. Start your *conversation* with the web server by typing the following into the terminal, and then perform HTTP requests to fetch the contents:

```
telnet www.surfweer.nl 80
```

1.1)

Write down the HTTP requests you made, the returned responses (e.g. a page has moved or is faulty) until you receive the desired contents. Always use **HEAD** first to retrieve meta-data about the resource. *Note that

1.2)

Does the content correspond to what you see when accessing the page with your browser? To check, save the response to a file, use **.html** as file ending and open it with your browser.

1.3)

What is the purpose of the **X-Cache** tag in the header information? If you have not seen it (what entity serves a resource can vary), look up the purpose of the **transfer-encoding** tag instead.

1.4)

What does the retrieved resource's **Cache-Control** directive mean?

2. HTTP request messages: PUT

While **GET** and **HEAD** are request methods accepted by virtually all web servers, **PUT**, **POST** and **DELETE** are less often available, due to the implications these methods have on the server.

To test your skills in uploading data, we will make use of <http://httpbin.org/>, a popular site designed to test HTTP messages.

Below is an example of how to upload data to the server with **PUT** (before you type it out, please read the explanations below):

```
telnet httpbin.org 80

PUT /put HTTP/1.1
host:httpbin.org
Content-type:text/plain
Content-length:12
<carriage return>
Hello World!
<carriage return>
```

Reminder: *Carriage return* in the code snippets indicates when an empty line is expected. Press `<Enter>` to add it.

With this code, we have modified the resource accessible at `/put` to now hold the string `Hello World!`. **The httpbin server sends back in the response the data just uploaded** - the response is of content-type JSON; we are interested in the `data` field, which should contain `Hello World!` if everything worked correctly. Try it for yourself!

PUT though is not only able to modify an existing resource, it can also **create** a resource on the server. The status code in the response tells us whether a resource was modified (`200 OK`) or created (`201 Create`). More information can be found on [MDN](#). What happens if you try to replace `/put` in this exercise with another resource (e.g. `/myfile`)? Does the httpbin.org server allow the creation of a new resource?

2.1)

The `Content-length` is exactly the number of characters (12 - we count the whitespace as well!) of `Hello World!`. What happens if the `Content-length` field is smaller or larger than the exact number of characters in the content?

3. Basic authentication

Let us now try to request a page, that is set up with HTTP basic authentication.

3.1)

First, open `http://httpbin.org/basic-auth/user/passwd` in your browser. You should see a dialogue, requesting username and password. Use `user` as username and `passwd` as password (*it is just a coincidence that the actual username and password is the same as the URL path*). Reload the web page - do you have to fill in the login details again? Why or why not?

3.2)

Now let's see how this works with actual HTTP messages. Go back to telnet and start off with a **HEAD** method to inspect the web page; document all following steps (requests and responses):

```
telnet httpbin.org 80

HEAD /basic-auth/user/passwd HTTP/1.1
host:httpbin.org
<carriage return>
```

Which status code do you receive now?

Next, use the **Authorization: Basic [base-64 encoded username/password string]** header field (you can type it out right after the `host:httpbin.org` line above) to provide username and password to the server. To encode the username and password, you can use any of the freely available base-64 en/decoders, e.g. <https://codebeautify.org/base64-encode>. Remember that username and password need to be combined as **username:password** before they are encoded in base-64.

Now close the TCP connection and start a new one, using again:

```
telnet httpbin.org 80
```

Request the same page without the **Authorization** header field - what happens? Is the behavior the same as reloading the page in the browser? Explain why / why not.

4. Web programming project: board game app

In this, and the upcoming two assignments, you will complete a web programming project. This year, this is the implementation of a classic **board game**.

You can choose from six games:

1. [Ludo](#): 2-4 players. **It is sufficient if you implement a working game for two players.**
2. [Draughts](#): 2 players
3. Chess - there are many variants available besides the default, [pick one](#): 2-4 players depending on the variant
4. [Mastermind](#): 2 players
5. [Connect 4](#): 2 players
6. [Reversi](#): 2 players

At the end of the three web assignments, your board game application will have the following functionalities:

- The game is for 2+ players and in 2D.
- The game works 2+ modern browsers (e.g. Firefox and Chrome).
- It works well on a laptop/desktop device, i.e. we are considering screen resolutions of ~1366x768 or higher. In this project, we are **not** concerned about apps for mobile devices.
- Upon entering your web application's URL, a **splash screen** is shown that allows a user to see some statistics of the game (how many games are currently ongoing, how many users have started a game, etc. - **pick three statistics you want to report**), a brief description of how-to-play on your platform and a *Play* button (or something to that effect).
- Upon pressing *Play*, the user enters the **game screen** and waits for a sufficient number of other gamers to start playing. It is clear for the player that s/he is waiting for more players to enter the game.
- Once there are sufficiently many players, the game automatically starts and the players play against each other. Multiple games can take place at the same time.
- The splash and game screens need to look good (we do realize that this is subjective and we handle this requirement very leniently!); all required game elements need to be visible (e.g. if a game requires a dice, a dice element needs to be visible).
- Once a player makes a move, the validity of the move is checked and invalid moves are rejected. Once a player wins the game, this information is announced to all players participating in the game.
- Players see basic information about the ongoing game, e.g. the time passed since starting the game or number of lost/won pieces.
- Players are able to play the game in fullscreen mode.
- Players play the game with the mouse.

- Once a player drops out of a game, the game is aborted; this is announced to all players currently active in the game.
- The game has at least one sound effect (e.g. a *ping* every time a move is made).

The list above should tell you that you have considerable (artistic) freedom. In assignments 5 & 6, you are given a set of requirements (e.g. here are the three types of CSS rules you need to employ in your code).

Beyond what is specified in subsequent assignments **no external libraries or frameworks are allowed**, apart from **jQuery** and (if needed) a library to help you validate the game moves. We allow **jQuery** as it is used in the web course book; you can use it too, but are not required to. You can use a JavaScript library to determine whether a player makes a valid move as for some games (such as chess) this is considerably more difficult than for others!

We also allow **small** simplifications to the chosen games, e.g. for Ludo an implementation for 2 players is sufficient, for chess you can ignore the **pawn promotion rule**. Be upfront about the simplifications made. If you are in doubt whether you are simplifying the game too much, ask us by emailing cse1500-ewi@tudelft.nl.

Optionally: when you have incorporated the requirements listed above without any additional libraries/framework besides **jQuery** (and a move validation library) and you want to keep improving your application by adding additional functionalities, you can indeed incorporate existing libraries/frameworks. Make sure to document clearly where in your code you employ them. The obvious next step to improve your app is the inclusion of a semi-intelligent computer opponent: while for the game of Ludo it would not be too difficult to come up with a number of rules to create a decent computer opponent, for chess this would not be possible in the time you have; here, a chess engine such as **Stockfish** helps.

If your team has a different idea and wants to implement another board game that has the functionalities listed above, please get permission from the instructors before you start doing any work by emailing (cse1500-ewi@tudelft.nl) your team ID and a short description of the game you have in mind.

4.1)

First of all, settle on the game you will implement in your team.

4.2)

Find **three** examples of your chosen board game (in 2D) that can be played online in a modern browser (laptop or desktop, not a mobile device). Consider the web application's design (focus on the game screen) based on the **web design principles** covered in class. Record the game URLs. Which **three design aspects** stand out positively or negatively? Make a screenshot of each example and annotate the positive/negative aspects.

4.3)

Which **game features** in the game examples of 4.2) stand out positively and which stand out negatively? (e.g. particular animations, sounds, information conveyed about the game to the players ...). Why? Discuss **three** positive and **three** negative features in total.

5. Design your own board game app

Having looked at at least three existing implementations of your chosen board game (Exercise 4.2), you are now in a position to design your own game interface. Similar to the wireframe example in the course book (check Chapter 2 if you have not done so yet) and the [demo code wireframes](#), start designing your own application. Create one **splash screen** and one **game screen**. As pointed out already, your web application should be designed for the standard Desktop interface. Use the software of your choice to create those wireframes. If you do not have any software installed on your machine that can be used for this purpose ... online platforms specifically for wireframe design are just a web search away, e.g. the simple [wireframe.cc](#) or the more elaborate [NinjaMock](#) and [Gliffy](#).

5.1)

Create a design for the splash screen (also known as *entry page*): think of a name for your application, a short description & a logo. Feel free to use media (images, sound) with a Creative Commons license. [You can start your resource search here](#). The [noun project](#) can be a useful resource for game pieces.

5.2)

Create a design for the game screen, keeping the requirements listed above in mind as well as your findings in Exercise 4.3). You have a lot of artistic freedom in designing the board and game information.

5.3)

Once you have completed the design of your app, head over to CSE1500's 💡 Brightspace, go to *Discussions* and then the forum **BOARD GAME APP DESIGNS**. Create a thread with your team's name as subject/title (e.g. [CSE234](#)) and post your team's proposed splash screen and game screen. Feel free to also add a paragraph describing your choices.

6. Your own board game app: HTML

Similar to the course book, take your design as a starting point and create the respective **two HTML documents**. These documents should **only** contain HTML, no CSS or JavaScript. To get an idea on the expected amount of content (it is not a lot!), check [game.html](#) and [splash.html](#) of the demo board game. Ignore the few lines of code loading JavaScript and CSS files, these will be covered in Assignments 5 and 6 respectively.