Intro to HTML

Programming with Web Technologies

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Auckland ICT Graduate School

HTML

- **H**yper**T**ext **M**arkup **L**anguage, used to describe the content and structure of documents on the web (web pages)
- The first version of HTML
  - text-only documents
  - define parts of a document content semantically (e.g. a title, paragraphs, headings, lists)
  - define links as anchors
- Now it is more complex – more than just text in web pages
  - more complex document structures
  - enhanced user interaction with pages

HTML

- Different versions as standards evolved
  - Current version is HTML5
  - https://en.wikipedia.org/wiki/HTML

- Standards are developed by W3C (World Wide Web Consortium)
  - http://w3.org
  - International body that defines HTML
    - Members include Google, Apple, Dropbox
  - And other standards (e.g., CSS, DOM, HTTP, XML, SVG etc.)

HTML

- An HTML document consists of two types of information
  - The content of the document
  - Markup that describes the content
- Markup is in the form of element tags
  - Content is (generally) surrounded by a start tag and an end tag
  - A tag is a letter or word, surrounded by angle brackets
    - `<b>`, `<body>`, `<table>`
  - A closing tag additionally has a forward-slash after the first angle bracket
    - `</b>`, `</body>`, `</table>`
  - Excepts for a few exceptions which will be discussed as needed, opening tags should have a corresponding closing tag

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

The <!DOCTYPE> declaration is the very first thing in the HTML document. It is an instruction to the web browser about what version of HTML the page was written in

This tag does not require a matching closing tag

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

The content of the HTML document needs to be enclosed by the `html` tag. All other tags will be nested inside this

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

A container for all informative tags. Tags inside the `head` of the document do not appear directly on the page, but can influence how elements are displayed

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

`meta` tags contain information about the HTML document. This can include a description of the page, the author and keywords. This information can be used by search engines to help index your page and by the browser to determine how to display the content

This tag does not require a matching closing tag

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

Start and End point of the text that is the title of the document. This text will appear as the tab and window name when the page is loaded in a browser

HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

A container for the tags that make up the document. Tags inside the body of the document appear directly on the page

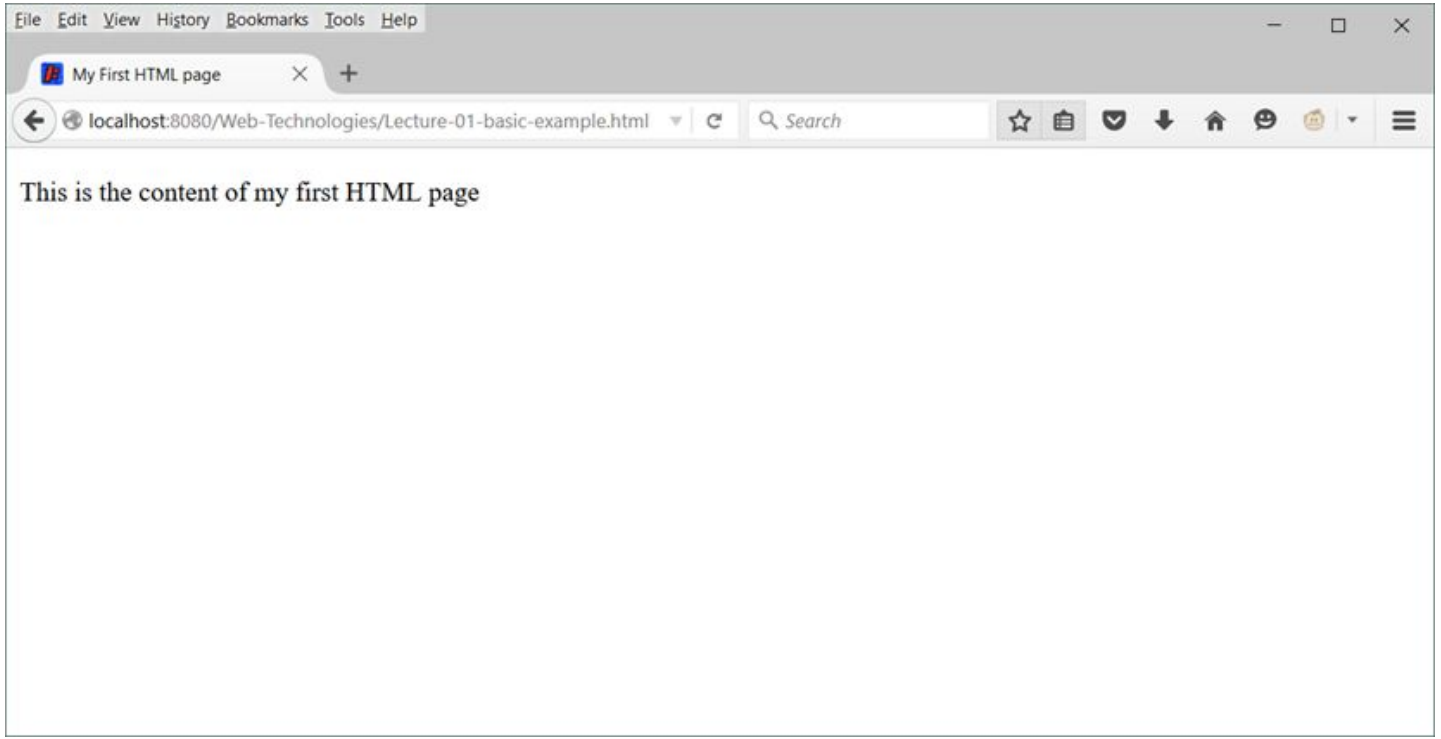
HTML Example

A simple HTML document

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>
            My First HTML page
        </title>
    </head>
    <body>
        <p>This is the content of my first HTML page</p>
    </body>
</html>
```

← A paragraph of text

HTML Example - Rendered



HTML Example - Notes

- Whitespace has no meaning most of the time, so the previous code is equivalent to this
  - ```
    <!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>My First HTML page</title></head><body> This is the content of my first HTML page </body></html>
    ```

- Code like the above is impossible to debug and maintain, so you should use whitespace liberally to indent and provide visual structure to your code
  - Your IDE and text editor will likely try to do this for you
    ```

HTML Notes

Tags should be correctly nested – Tags should be closed in the opposite order to which they were opened

```
<p><em> Some content </em></p>
```

```
<p><em> Some content </p></em>
```

Bad HTML

- When the web started to become popular with the general public, people with no experience or training began to create HTML web pages
  - A good illustration of what the web used to be
- Modern browsers are resilient to many HTML errors
  - Even with an error something would be displayed
  - It can however, be difficult to determine what something will look like if it is invalid HTML
    - Each browser will respond differently
- We do know that browsers deal correctly and predictably with valid HTML
  - We should make sure that the HTML that we write is valid
  - Validation tools are available to ensure that our HTML is valid
    - Built-in IDE tools or online validators

HTML - Spot the Errors

```
<head>
</head>
<title>
    My First HTML page
</title>
    This is the content of my first HTML page
</body>
</html>
```

HTML Element Types

There are two types of elements which can be used for content in HTML

- Block level elements
  - Always starts on a new line
  - Expands to fill the available width (stretches out to the left and right as far as it can)

- Inline elements
  - Does not start on a new line
  - Only takes up as much width as necessary

HTML Elements - Block Level

```
<p>Paragraphs are block level elements.</p><p>Even though these are written on the same line, they will render on different lines in the browser</p>
```

Paragraphs are block level elements.

Even though these are written on the same line, they will render on different lines in the browser

HTML Elements - Inline

```
<p><em>Emphasis</em> and <strong>Strong</strong> tags are
inline elements. They are written on the same line in HTML,
and also appear in the same line in the browser.</p>
```

Emphasis and **Strong** tags are inline elements. They are written on the same line in HTML, and also appear in the same line in the browser.

HTML Elements - Inline and Block Nesting

Inline elements

- Can be nested inside block level and other inline elements

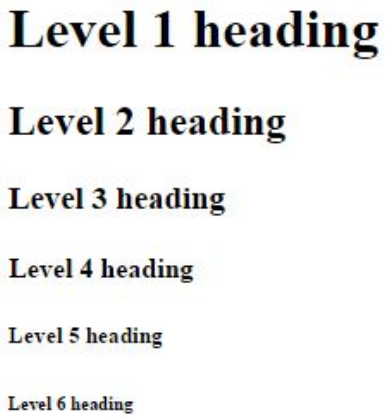
Block level elements

- Can be nested inside other block level elements
- **Cannot** be nested inside inline elements

Headings

- HTML supports six levels of heading, which can be used to organize documents into sections
- <h1> is the most important, and <h6> is the least important
- Search engines and other tools using heading importance to index web pages, so use them appropriately

```
<h1>Level 1 heading</h1>
<h2>Level 2 heading</h2>
<h3>Level 3 heading</h3>
<h4>Level 4 heading</h4>
<h5>Level 5 heading</h5>
<h6>Level 6 heading</h6>
```



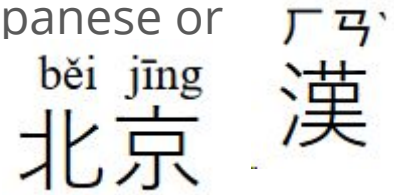
Text in HTML

Text written in the <body> of an HTML document will be rendered into the browser directly, however it is good practice to surround text with appropriate tags to convey meaning/intent

- <p> - Indicates the text enclosed is a paragraph. Default styling will force a line break between paragraphs
- <pre> - Preformatted text. Text will be displayed in the browser exactly as it is typed in the HTML, with whitespace preserved
- <code> - Source code, by default displayed using a monospaced font

Text in HTML

- <blockquote> - A blockquote originating from a different source, by default displayed using a monospaced font
- <q> - A short quote, displayed inline
- <cite> - An inline citation of the title of an article, movie, book, etc
- <address> - An address, displayed in italics by default
- <ruby> - A ruby annotation, a pronunciation guide for Japanese or Chinese text to aid pronunciation of unfamiliar words



And many more, ranging from general to extremely special cases

Lists

HTML5 supports three varieties of list

- Unordered lists
  - Bullet points. Used where order doesn’t matter
- Ordered lists
  - Numeric list. Used where order matters
- Description lists
  - Term, Description pairs. Used when you want to define or describe terms

Unordered List

HTML Markup

```
<ul>
  <li>Entry 1</li>
  <li>Entry 2</li>
  <li>Entry A</li>
  <li>Entry B</li>
</ul>
```

Rendered Result

- Entry 1
- Entry 2
- Entry A
- Entry B

Ordered List

HTML Markup

```
<ol>
  <li>Entry 1</li>
  <li>Entry 2</li>
  <li>Entry A</li>
  <li>Entry B</li>
</ol>
```

Rendered Result

1. Entry 1
2. Entry 2
3. Entry A
4. Entry B

Description List

HTML Markup

```
<dl>
  <dt>Coffee</dt>
  <dd>- Hot black drink</dd>
  <dt>Milk</dt>
  <dd>- Cold white drink</dd>
</dl>
```

Rendered Result

Coffee

- Hot black drink

Milk

- Cold white drink

Nesting Lists

HTML Markup

```
<ol>
  <li>Entry 1</li>
  <li>Entry 2</li>
  <li>Nested
    <ul>
      <li>Entry A</li>
      <li>Entry B</li>
    </ul>
  </li>
  <li>Entry 4</li>
</ol>
```

Rendered Result

1. Entry 1
2. Entry 2
3. Nested
    - Entry A
    - Entry B
4. Entry 4

More useful tags

- <hr> - Indicates a "Thematic break" in the document content, often shown as a horizontal line across the page. Does not need a closing tag
-
 - Forces a newline in some text, which is often used in representing addresses. Does not need a closing tag
- <!-- --> - A comment. Anything written between the <!-- and --> markers will not be displayed on the page

HTML Entities

- Some characters have special meaning in HTML; such as angle brackets and quotes; or can't be typed with a typical keyboard
- If we need to display one of these, we can use HTML entities

Result	Description	Entity Name
	Non-breaking space	` `
>	Greater than	`>`
<	Less than	`<`
&	Ampersand	`&`

Result	Description	Entity Name
"	Double quote	`&quot;`
'	Single quote	`&apos;`
©	Copyright Symbol	`&copy;`
™	Trademark Symbol	`&trade;`

More entities here

Browser Inspections

- Most modern web browsers allow you to see the HTML (and other files) of a webpage in a couple of ways
- View Source
  - This option can typically be found in the right-click menu of a browser, and will show you the complete source of a page in a new window.
- Inspection
  - Again, this option can often be found in the right-click menu. This differs from view source by being dynamic – You can change the source and see your changes reflected.

Intro to GitLab

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Auckland ICT Graduate School

Git & GitLab

- In this course, we will use git and GitLab to access & manage lab and other material
  - Git is a **version control** system - very useful for maintaining an “edit history” of various documents, so you can easily go back and correct any mistakes
  - Git is also great when many developers *collaborate* on the same project - helps prevent conflicts when different people’s code is merged together
- We will learn more about git in the next lecture. Today, we’ll just get started by using it to obtain the lab material, and to save your progress to the GitLab online repository.

Backing Up Your Data

- As with any work, you want to make sure you backup your code. You may have used cloud services like Google Drive or Dropbox in the past for other work
  - While these are great tools, they are not the best choice for code
- Cloud services don't typically keep incremental backups, and if they do they do not last forever
  - This means you can't 'go back' several versions, especially versions that are several months or years old
- Cloud services can't 'diff' all files, showing changes from one version to another
  - This is something that is useful to be able to do with code

Version Control Systems

- Programmers use a special type of software to manage changes to their code: Version/Source Control Systems
- These systems track changes to files. Additionally, they are aware that they are working with code which allows for smart behaviour
  - Keeps a 'history' of your code
  - Allows you to go back in time to an earlier version if something goes wrong
  - Lets you see exactly what changed between versions
- These systems can also be used to backup and share your code
  - They can connect to remote repositories
  - Your changes can be sent to configured 'remotes'

Git

- We will be using 'Git' in this course
  - Most widely used Distributed Version Control System (DVCS) in use today
  - Many commands in common with other DVCS
  - Git knowledge is desired in industry
- Git is primarily a command-line (CMD) based program, but there are a number of graphical front-ends (GUI) and IDE integrations available
  - To start you will mostly be using your IDE to control git
  - Later in the course we will revisit git and look at command-line options
  - You are welcome to use other graphical interfaces if you choose to
    - SourceTree
    - Git Tower
    - Git Kraken

So what do we get from git?

- Built-in commit messages
  - Everytime we make a change to our code, we record a message that tells us what we did
  - This message stays with the code, along with who made the commit and when
- Small disk space footprint
  - Git stores 'diffs' or 'deltas' that store just the changes in the file, not the whole thing with each commit.
  - This saves a lot of space over time, especially compared to zipping copies of the project
- Built in functions in git allow you to explore and manipulate the history
  - Allows you to work out where bugs were introduced in larger team projects
  - Find exactly when you made a change and undo it, even if it occured years prior

Backup and Sharing With Git

- git repositories can point to other 'remote' git repositories
  - These can be on your local computer, another computer on your network or even on a server somewhere in the world
  - You can perform operations on these remote repositories, such as sending changes to them, or making local copies of the repositories
  - Can act as a backup, collaboration or distribution channel

We will be using remote repositories to provide lab exercises from today onwards

Before you get started

Before you can get started with git, you need to tell it about you - specifically your name and email address. The information provided will be attached to your commit messages, and git will not function without them. This configuration is best done from the command-line

```
git config --global user.name "Cameron Grout"
git config --global user.email cameron.grout@waikato.ac.nz
```

```
git config --list
```

- View all set options in order of system ➧ global ➧ local

Creating a repository

There are two primary ways that you can get started with a git repository - creating a new empty repository, or forking and cloning an existing one

Creating a new repository is simple in the command-line or in a GUI. From the command-line, issuing the command `git init` will create an empty repository in the current directory. To do the same from a GUI you would be looking for an 'Initialize Repository' or 'Create new repository' option - it will vary between interfaces

New repositories created this way will be empty of commits, but any files that were already in the directory will be unaffected

Obtaining an existing repository

Retrieving an existing repository is also fairly easy. In this course, you will be provided with a link to a source control hosting website that has existing lab repositories to work on. Before you can get started on coding though, you first need to **fork** each repository using the controls on the hosting webpage. Forking makes a copy of the repository on the hosting site that you control

The next step is to **clone** your forked repository, which can be done through a GUI or CMD. From a GUI you look for a 'Checkout from Version Control', or 'Clone repo' option, and provide a URL that points to your repository. From a CMD, the command `git clone https://.../project.git` will copy the repository from the provided URL to your local machine

Understanding your Repository

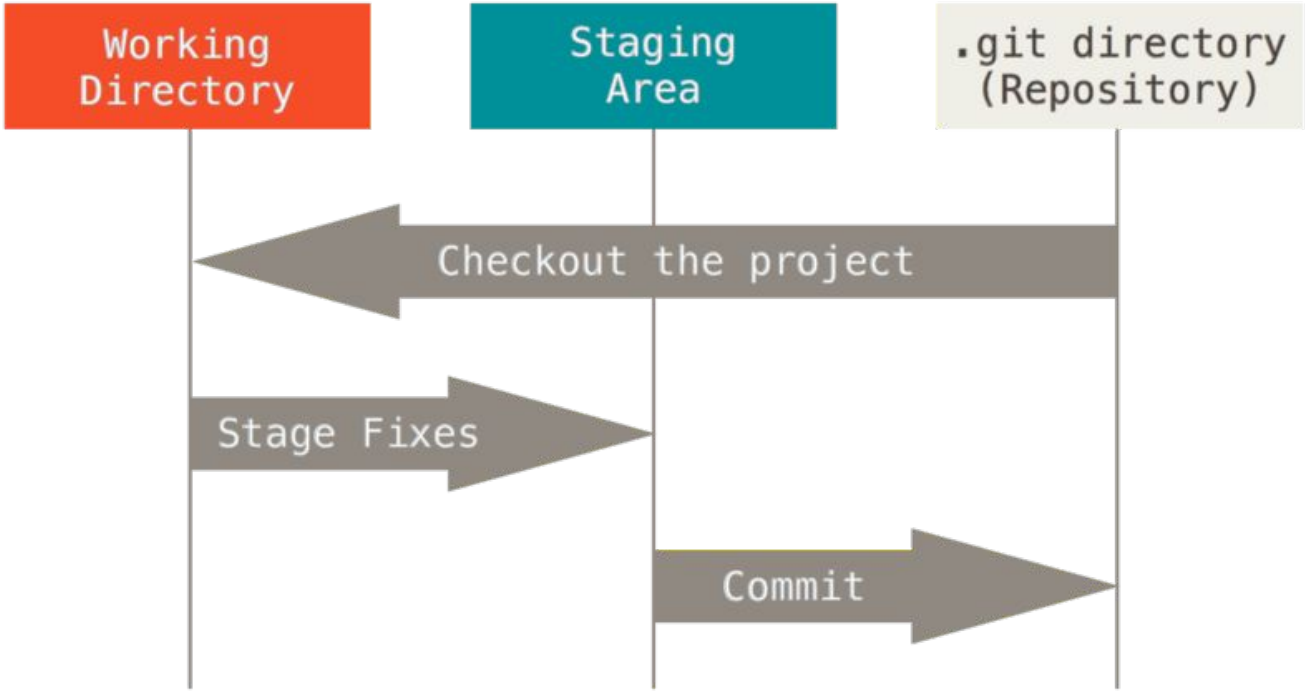
Now you have a repository and are nearly ready to get started, but first you need to know a little about how repositories work. A git repository (repo for short) has 3 'areas' where your files can be during their lifecycle

The Working Directory is what you see in your folders and editors. This represents the current state of your files

The Staging Area is where files are held for pending operation

The Repository is where the history of your files are kept

Understanding your Repository



Understanding your Repository

As you develop, you will want to periodically move your files through these 3 areas in order to create snapshot commits, or to go back to previous versions of your code. The former, creating commits, is what we care about most today

In order to create a commit, we have to move files through 2 transitions - taking changed files from the Working Area and moving them into the Staging Area, and committing the staged changes into the Repository

Staging files

When a new file is added to the repository, or an existing tracked file is modified, this change needs to be **add**ed to the Staging Area

From the command-line, this is done using the `git add filename.ext` command to stage a single file, `git add .` to stage all new or changed files, or another variation on the command to add files meeting certain criteria

From a GUI, files can be staged by selecting the 'Add' or 'Stage file' options, dependent on your GUI of choice. These interfaces often provide convenient ways to stage just part of a file rather than the whole thing - something that is much more awkward from the command-line

Staging files

Occasionally you will stage something accidently, and you will want to remove it from the stage. From the command-line the simple if a little obscure command `git reset HEAD stagedfile.ext` will remove the named file from the stage without undoing any changes you made to the file

From a GUI again doing this is quite simple. Look for 'Reset' or 'Unstage File' options, or when dealing with later committing steps, uncheck the files you don't want included from the summary list of changes

Committing changes

Now that you have staged your files, you can commit them to the repository along with a commit message describing your changes. As with the previous steps, this can be done from your GUI of choice or from the command-line

When using a GUI, look for a 'Commit' options, and follow the resulting dialogs. In general, this will ask you to confirm that you want to commit the staged files, and give you the option to unstage any you added accidently. You will also be prompted to provide a message that describes the changes made

From the command-line, running `git commit` will display an editor window for you to provide a commit message, after which staged files will be committed to the repository

Pushing changes

With your changes committed you are able to go on with developing new features. Before you do though, you will want to make sure there is a backup of your data. With git, the simplest way to do this is through pushing your local changes to your forked repository

From the command-line, this is done using the `git push -u origin master` command, and from a GUI by selecting the 'Push' option, typically found alongside the add and commit buttons

After pushing your changes, the remote fork will now contain a copy of what if found on your local machine

The Basic Git Workflow

With each of the steps addressed, we can outline a basic git workflow

1. Initialize a repository with `git init`, or obtain an existing one with `git clone`
2. Create or modify files
3. Track and stage new files or changes using `git add`
4. Commit your changes with `git commit`
5. Push your changes back to your hosting remote with `git push`
6. Loop back to #2

The Basic Git Workflow

- Be reasonably granular with your commits
  - Try to commit after each logical step in your development process
  - Try to find a happy medium between single line and entire file changes
- Try to have commits be about a single task or topic
  - You should be able to explain the main thrust/intention of the commit in a single line (80 characters) of a commit
  - If explaining your commit this concisely is too difficult, then you probably have too much in the commit
  - Provide as much information as you like for a commit, there is no character limit and it can be really helpful to read them back in the future

Recommended Reading

W3schools HTML Tutorial

MDN tag reference

HTML, CSS and JavaScript (Meloni, 2015) Chapter 2

Chacon, S & Straub, B. 2014. Pro Git. Chapters 2-4

tryGit - An interactive git tutorial