FULL STACK DEVELOPMENT

[R22A0516] LECTURE NOTES

B.TECH III YEAR – I SEM(R22)
(2024-25)

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

MALLA REDDY COLLEGE OF ENGINEERING &TECHNOLOGY

DEPARTMENT OF CSE

INDEX

(R20A0516) FULL STACK DEVELOPMENT

COURSE OBJECTIVES:

1. To become knowledgeable about the most recent web development technologies.
2. Idea for creating two tier and three tier architectural web applications.
3. Design and Analyze real time web applications. and Constructing suitable client and server side applications.
4. To learn core concept of both front end and back end programming.
5. Students will become familiar to implement fast, efficient, interactive and scalable web applications using run time environment provided by the full stack components

UNIT - I
Web Development Basics: Web development Basics - HTML & Web servers Shell - UNIX CLI Version control - Git & Github HTML, CSS

UNIT - II
Frontend Development: Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS AJAX for data exchange with server jQuery Framework jQuery events, UI components etc. JSON data format.

UNIT - III
Angular: importance of Angular, Understanding Angular, creating a Basic Angular Application, Angular Components, Expressions, Data Binding, Built-in Directives, Custom Directives, Implementing Angular Services in Web Applications.

React:
Need of React, Simple React Structure, The Virtual DOM, React Components, Introducing React Components, Creating Components in React, Data and Data Flow in React, Rendering and Life Cycle Methods in React, Working with forms in React, integrating third party libraries, Routing in React.

UNIT - IV
Node js: Getting Started with Node.js, Using Events, Listeners, Timers, and Callbacks in Node.js, Handling Data I/O in Node.js, Accessing the File System from Node.js, Implementing Socket Services in Node.js.

UNIT - V
MongoDB:
Understanding NoSQL and MongoDB, Getting Started with MongoDB, Getting Started with MongoDB and Node.js, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js, Using Mongoose for Structured Schema and Validation, Advanced MongoDB Concepts.

TEXT BOOKS:
1. Web Design with HTML, CSS, JavaScript and JQuery Set Book by Jon Duckett Professional JavaScript for Web Developers Book by Nicholas C. Zakas. (Unit-I,II).
2. ProGit, 2nd Edition, Apress publication by Scott Chacon and Straub. (Unit I).
3. Brad Dayley, Brendan Dayley, Caleb Dayley., Node.js, MongoDB and Angular Web Development, 2nd Edition, Addison-Wesley, 2019. (Unit-III, Unit-IV, Unit-V).
4. Mark Tielens Thomas, React in Action, 1st Edition, Manning Publications. (Unit-III).

REFERENCE BOOKS:
1. Full-Stack JavaScript Development by Eric Bush.
2. Mastering Full Stack React Web Development Paperback – April 28, 2017 by Tomasz Dyl , Kamil Przeorski , Maciej Czarnecki

COURSE OUTCOMES:
1. Understand Full stack components for developing web application.
2. Apply packages of NodeJS to work with Data, Files, Http Requests and Responses.
3. Use MongoDB data base for storing and processing huge data and connects with NodeJS application.
4. Design faster and effective single page applications using Angular.
5. Create interactive user interfaces with react components

HTML Document Structure

A typical HTML document will have the following structure:

```
Document declaration tag
<html>
   <head>
       Document header related tags
   </head>

   <body>
       Document body related tags
   </body>
</html>
```

We will study all the header and body tags in subsequent chapters, but for now let's see what is document declaration tag.

The <!DOCTYPE> Declaration

The <!DOCTYPE> declaration tag is used by the web browser to understand the version of the HTML used in the document. Current version of HTML is 5 and it makes use of the following declaration:

```
<!DOCTYPE html>
```

There are many other declaration types which can be used in HTML document depending on what version of HTML is being used. We will see more details on this while discussing <!DOCTYPE...> tag along with other HTML tags.

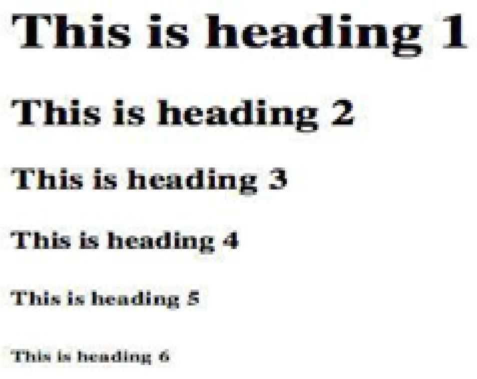
Heading Tags

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements **<h1>, <h2>, <h3>, <h4>, <h5>, and <h6>**. While displaying any heading, browser adds one line before and one line after that heading.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Heading Example</title>
</head>
<body>
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
<h4>This is heading 4</h4>
<h5>This is heading 5</h5>
<h6>This is heading 6</h6>
</body>
</html>
```

This will produce the following result:


This is heading 1

This is heading 2

This is heading 3

This is heading 4

This is heading 5

This is heading 6

Paragraph Tag

The **<p>** tag offers a way to structure your text into different paragraphs. Each paragraph of text should go in between an opening <p> and a closing </p> tag as shown below in the example:

3 | P a g e

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Paragraph Example</title>
</head>
<body>
<p>Here is a first paragraph of text.</p>
<p>Here is a second paragraph of text.</p>
<p>Here is a third paragraph of text.</p>
</body>
</html>
```

This will produce the following result:

```
Here is a first paragraph of text.
Here is a second paragraph of text.
Here is a third paragraph of text.
```

Line Break Tag

Whenever you use the **<br />** element, anything following it starts from the next line. This tag is an example of an **empty** element, where you do not need opening and closing tags, as there is nothing to go in between them.

The <br /> tag has a space between the characters **br** and the forward slash. If you omit this space, older browsers will have trouble rendering the line break, while if you miss the forward slash character and just use <br> it is not valid in XHTML.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Line Break  Example</title>
</head>
<body>
```

```
<p>Hello<br />

You delivered your assignment on time.<br />

Thanks<br />

Mahnaz</p>

</body>

</html>
```

This will produce the following result:

```
Hello
You delivered your assignment on time.
Thanks
Mahnaz
```

Centering Content

You can use **<center>** tag to put any content in the center of the page or any table cell.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Centring Content Example</title>

</head>

<body>

<p>This text is not in the center.</p>

<center>

<p>This text is in the center.</p>

</center>

</body>

</html>
```

This will produce the following result:

This text is not in the center.

This text is in the center.

Horizontal Lines

Horizontal lines are used to visually break-up sections of a document. The **<hr>** tag creates a line from the current position in the document to the right margin and breaks the line accordingly.

For example, you may want to give a line between two paragraphs as in the given example below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Horizontal Line Example</title>
</head>
<body>
<p>This is paragraph one and should be on top</p>
<hr />
<p>This is paragraph two and should be at bottom</p>
</body>
</html>
```

This will produce the following result:

This is paragraph one and should be on top

This is paragraph two and should be at bottom

Again **<hr />** tag is an example of the **empty** element, where you do not need opening and closing tags, as there is nothing to go in between them.

The **<hr />** element has a space between the characters **hr** and the forward slash. If you omit this space, older browsers will have trouble rendering the horizontal line, while if you miss the forward slash character and just use **<hr>** it is not valid in XHTML

Preserve Formatting

Sometimes, you want your text to follow the exact format of how it is written in the HTML document. In these cases, you can use the preformatted tag **<pre>**.

Any text between the opening **<pre>** tag and the closing **</pre>** tag will preserve the formatting of the source document.

6 | Page

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Preserve Formatting Example</title>
</head>
<body>
<pre>
function testFunction( strText ){
   alert (strText)
}
</pre>
</body>
</html>
```

This will produce the following result:

```
function testFunction( strText ){
  alert (strText)
}
```

Try using the same code without keeping it inside **<pre>...</pre>** tags

Nonbreaking Spaces

Suppose you want to use the phrase "12 Angry Men." Here, you would not want a browser to split the "12, Angry" and "Men" across two lines:

```
An example of this technique appears in the movie "12 Angry Men."
```

In cases, where you do not want the client browser to break text, you should use a nonbreaking space entity **&nbsp;** instead of a normal space. For example, when coding the "12 Angry Men" in a paragraph, you should use something similar to the following code:

Example

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Nonbreaking Spaces Example</title>
</head>
<body>
<p>An example of this technique appears in the movie "12&nbsp;Angry&nbsp;Men."</p>
</body>
</html>
```

3. HTML – ELEMENTS

An **HTML element** is defined by a starting tag. If the element contains other content, it ends with a closing tag, where the element name is preceded by a forward slash as shown below with few tags:

Start Tag	Content	End Tag
<p>	This is paragraph content.	</p>
<h1>	This is heading content.	</h1>
<div>	This is division content.	</div>
<br />		

So here **<p>....</p>** is an HTML element, **<h1>...</h1>** is another HTML element. There are some HTML elements which don't need to be closed, such as **<img.../>**, **<hr />** and **<br />** elements. These are known as **void elements**.

HTML documents consists of a tree of these elements and they specify how HTML documents should be built, and what kind of content should be placed in what part of an HTML document.

HTML Tag vs. Element

An HTML element is defined by a *starting tag*. If the element contains other content, it ends with a *closing tag*.

For example, **<p>** is starting tag of a paragraph and **</p>** is closing tag of the same paragraph but **<p>This is paragraph</p>** is a paragraph element.

Nested HTML Elements

It is very much allowed to keep one HTML element inside another HTML element:

Example

```
<!DOCTYPE html>

<html>
<head>

<title>Nested Elements Example</title>

</head>

<body>

<h1>This is <i>italic</i> heading</h1>

<p>This is <u>underlined</u> paragraph</p>

</body>

</html>
```

This will display the following result:

This is *italic* heading

This is underlined paragraph

4. HTML – ATTRIBUTES

We have seen few HTML tags and their usage like heading tags **<h1>, <h2>,** paragraph tag **<p>** and other tags. We used them so far in their simplest form, but most of the HTML tags can also have attributes, which are extra bits of information.

An attribute is used to define the characteristics of an HTML element and is placed inside the element's opening tag. All attributes are made up of two parts: a **name** and a **value**:

- The **name** is the property you want to set. For example, the paragraph **<p>** element in the example carries an attribute whose name is **align**, which you can use to indicate the alignment of paragraph on the page.
-
- The **value** is what you want the value of the property to be set and always put within quotations. The below example shows three possible values of align attribute: **left, center** and **right**.

Attribute names and attribute values are case-insensitive. However, the World Wide Web Consortium (W3C) recommends lowercase attributes/attribute values in their HTML 4 recommendation.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Align Attribute  Example</title>

</head>

<body>

<p align="left">This is left aligned</p>

<p align="center">This is center aligned</p>

<p align="right">This is right aligned</p>

</body>

</html>
```

This will display the following result:

```
This is left aligned
                              This is center aligned
                                                            This is right aligned
```

Core Attributes

The four core attributes that can be used on the majority of HTML elements (although not all) are:

- Id
- Title
- Class
- Style

The Id Attribute

The **id** attribute of an HTML tag can be used to uniquely identify any element within an HTML page. There are two primary reasons that you might want to use an id attribute on anelement:

- If an element carries an id attribute as a unique identifier, it is possible to identify just that element and its content.

- If you have two elements of the same name within a Web page (or style sheet), you can use the id attribute to distinguish between elements that have the same name.

We will discuss style sheet in separate tutorial. For now, let's use the id attribute to distinguish between two paragraph elements as shown below.

Example

```
<p id="html">This para explains what is HTML</p>

<p id="css">This para explains what is Cascading Style Sheet</p>
```

The title Attribute

The **title** attribute gives a suggested title for the element. They syntax for the **title** attribute is similar as explained for **id** attribute:

The behavior of this attribute will depend upon the element that carries it, although it is often displayed as a tooltip when cursor comes over the element or while the element is loading.

Example

```
<!DOCTYPE html>

<html>

<head>



<title>The title Attribute Example</title>
```

```
</head>

<body>

<h3 title="Hello HTML!">Titled Heading Tag Example</h3>

</body>

</html>
```

This will produce the following result:

Titled Heading Tag Example

Now try to bring your cursor over "Titled Heading Tag Example" and you will see that whatever title you used in your code is coming out as a tooltip of the cursor.

The class Attribute

The **class** attribute is used to associate an element with a style sheet, and specifies the class of element. You will learn more about the use of the class attribute when you will learn Cascading Style Sheet (CSS). So for now you can avoid it.

The value of the attribute may also be a space-separated list of class names. For example:

```
class="className1 className2 className3"
```

The style Attribute

The style attribute allows you to specify Cascading Style Sheet (CSS) rules within the element.

```
<!DOCTYPE html>

<html>

<head>

<title>The style Attribute</title>

</head>

<body>

<p style="font-family:arial; color:#FF0000;">Some text...</p>

</body>

</html>
```

This will produce the following result:

```
Some text...
```

At this point of time, we are not learning CSS, so just let's proceed without bothering much about CSS. Here, you need to understand what are HTML attributes and how they can be used while formatting content.

Internationalization Attributes

There are three internationalization attributes, which are available for most (although notall) XHTML elements.

- dir
- lang
- xml:lang

The dir Attribute

The **dir** attribute allows you to indicate to the browser about the direction in which the text should flow. The dir attribute can take one of two values, as you can see in the table that follows:

Value	Meaning
ltr	Left to right (the default value)
rtl	Right to left (for languages such as Hebrew or Arabic that are read right to left)

Example

```
<!DOCTYPE html>
<html dir="rtl">
<head>
<title>Display Directions</title>
</head>
<body>
This is how IE 5 renders right-to-left directed text.
</body>
</html>
```

This will produce the following result:

```
                              This is how IE 5 renders right-to-left directed text.
```

When *dir* attribute is used within the <html> tag, it determines how text will be presented within the entire document. When used within another tag, it controls the text's direction for just the content of that tag.

The lang Attribute

The **lang** attribute allows you to indicate the main language used in a document, but this attribute was kept in HTML only for backwards compatibility with earlier versions of HTML. This attribute has been replaced by the **xml:lang** attribute in new XHTML documents.

The values of the *lang* attribute are ISO-639 standard two-character language codes. Check HTML Language Codes: ISO 639 for a complete list of language codes.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>English Language Page</title>
</head>
<body>
This page is using English Language
</body>
</html>
```

The xml:lang Attribute

The *xml:lang* attribute is the XHTML replacement for the *lang* attribute. The value of the*xml:lang* attribute should be an ISO-639 country code as mentioned in previous section.

Generic Attributes

Here's a table of some other attributes that are readily usable with many of the HTML tags.

Attribute	Options	Function
align	right, left, center	Horizontally aligns tags
valign	top, middle, bottom	Vertically aligns tags within an HTML element.
bgcolor	numeric, hexidecimal, RGB values	Places a background color behind an element
background	URL	Places a background image behind an element
id	User Defined	Names an element for use with Cascading Style Sheets.
class	User Defined	Classifies an element for use with Cascading Style Sheets.
width	Numeric Value	Specifies the width of tables, images, or table cells.
height	Numeric Value	Specifies the height of tables, images, or table cells.
title	User Defined	"Pop-up" title of the elements.

We will see related examples as we will proceed to study other HTML tags. For a complete list of HTML Tags and related attributes please check reference to HTML Tags List.

If you use a word processor, you must be familiar with the ability to make text bold, italicized, or underlined; these are just three of the ten options available to indicate how text can appear in HTML and XHTML.

Bold Text

Anything that appears within **<b>...</b>** element, is displayed in bold as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Bold Text Example</title>

</head>

<body>

<p>The following word uses a <b>bold</b> typeface.</p>

</body>

</html>
```

This will produce the following result:

```
The following word uses a bold typeface.
```

Italic Text

Anything that appears within **<i>...</i>** element is displayed in italicized as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Italic Text Example</title>

</head>
<body>

<p>The following word uses a <i>italicized</i> typeface.</p>

</body>

</html>
```

This will produce the following result:

```
The following word uses an italicized typeface.
```

Underlined Text

Anything that appears within **<u>...</u>** element, is displayed with underline as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Underlined Text Example</title>

</head>

<body>

<p>The following word uses a <u>underlined</u> typeface.</p>

</body>

</html>
```

This will produce the following result:

The following word uses an <u>underlined</u> typeface.

Strike Text

Anything that appears within **<strike>...</strike>** element is displayed with strikethrough, which is a thin line through the text as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>
<title>Strike Text Example</title>

</head>

<body>

<p>The following word uses a <strike>strikethrough</strike> typeface.</p>

</body>

</html>
```

This will produce the following result:

The following word uses a ~~strikethrough~~ typeface.

Monospaced Font

The content of a **<tt>...</tt>** element is written in monospaced font. Most of the fonts are known as variable-width fonts because different letters are of different widths (for example, the letter 'm' is wider than the letter 'i'). In a monospaced font, however, each letter has the same width.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Monospaced Font Example</title>

</head>

<body>

<p>The following word uses a <tt>monospaced</tt> typeface.</p>

</body>

</html>
```

This will produce the following result:

```
The following word uses a monospaced typeface.
```

Superscript Text

The content of a **<sup>...</sup>** element is written in superscript; the font size used is the same size as the characters surrounding it but is displayed half a character's height above the other characters.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Superscript Text Example</title>

</head>

<body>

<p>The following word uses a <sup>superscript</sup> typeface.</p>

</body>

</html>
```

This will produce the following result:

```
The following word uses a superscript typeface.
```

Subscript Text

The content of a **<sub>...</sub>** element is written in subscript; the font size used is the same as the characters surrounding it, but is displayed half a character's height beneath the other characters.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Subscript Text Example</title>

</head>

<body>

<p>The following word uses a <sub>subscript</sub> typeface.</p>

</body>

</html>
```

This will produce the following result:

The following word uses a $_{\text{subscript}}$ typeface.

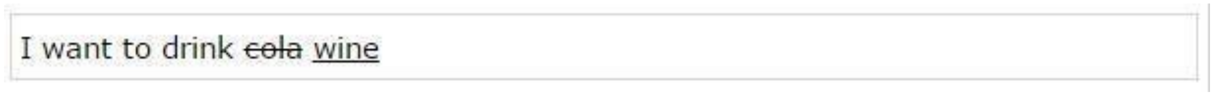
Inserted Text

Anything that appears within **<ins>...</ins>** element is displayed as inserted text.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Inserted Text Example</title>

</head>

<body>

<p>I want to drink <del>cola</del> <ins>wine</ins></p>

</body>

</html>
```

This will produce the following result:


I want to drink ~~cola~~ wine

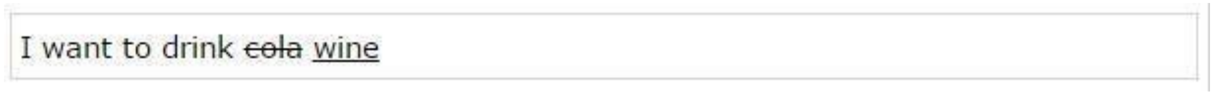
Deleted Text

Anything that appears within **<del>...</del>** element, is displayed as deleted text.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Deleted Text Example</title>

</head>

<body>

<p>I want to drink <del>cola</del> <ins>wine</ins></p>

</body>

</html>
```

This will produce the following result:


I want to drink ~~cola~~ wine

Larger Text

The content of the **<big>...</big>** element is displayed one font size larger than the rest of the text surrounding it as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Larger Text Example</title>

</head>

<body>

<p>The following word uses a <big>big</big> typeface.</p>

</body>
```

```
</html>
```

This will produce the following result:

```
The following word uses a big typeface.
```

Smaller Text

The content of the **<small>...</small>** element is displayed one font size smaller than the rest of the text surrounding it as shown below:

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Smaller Text Example</title>

</head>

<body>

<p>The following word uses a <small>small</small> typeface.</p>

</body>

</html>
```

This will produce the following result:

```
The following word uses a small typeface.
```

Grouping Content

The **<div>** and **<span>** elements allow you to group together several elements to create sections or subsections of a page.

For example, you might want to put all of the footnotes on a page within a <div> element to indicate that all of the elements within that <div> element relate to the footnotes. You might then attach a style to this <div> element so that they appear using a special set of style rules.

Example

```
<!DOCTYPE html>

<html>
```

```
<head>
<title>Div Tag Example</title>
</head>
<body>
<div id="menu" align="middle" >
<a href="/index.htm">HOME</a> |
<a href="/about/contact_us.htm">CONTACT</a> |
<a href="/about/index.htm">ABOUT</a> </div>



<div id="content" align="left" bgcolor="white">
<h5>Content Articles</h5>
<p>Actual content goes here. ...</p>
</div>
</body>
</html>
```

This will produce the following result:

HOME | CONTACT | ABOUT

CONTENT ARTICLES

Actual content goes here.....

The <span> element, on the other hand, can be used to group inline elements only. So, if you have a part of a sentence or paragraph which you want to group together, you could use the <span> element as follows

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Span Tag Example</title>
</head>
<body>
<p>This is the example of <span style="color:green">span tag</span> and the <span
style="color:red">div tag</span> alongwith CSS</p>
```

```
</body>
</html>
```

This will produce the following result:

This is the example of span tag and the div tag along with CSS

These tags are commonly used with CSS to allow you to attach a style to a section of a page.

6. HTML – PHRASE TAGS

The phrase tags have been desicolgned for specific purposes, though they are displayed in a similar way as other basic tags like **<b>, <i>, <pre>,** and **<tt>,** you have seen in previous chapter. This chapter will take you through all the important phrase tags, so let's start seeing them one by one.

Emphasized Text

Anything that appears within **<em>...</em>** element is displayed as emphasized text.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Emphasized Text Example</title>
</head>
<body>
<p>The following word uses a <em>emphasized</em> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses an *emphasized* typeface.

Marked Text

Anything that appears with-in **<mark>...</mark>** element, is displayed as marked with yellow ink.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Marked Text Example</title>
</head>
<body>
<p>The following word has been <mark>marked</mark> with yellow</p>
</body>
</html>
```

This will produce the following result:
The following word has been marked with yellow

Strong Text

Anything that appears within **<strong>...</strong>** element is displayed as important text.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Strong Text Example</title>

</head>

<body>

<p>The following word uses a <strong>strong</strong> typeface.</p>

</body>

</html>
```

This will produce the following result:

The following word uses a **strong** typeface.

Text Abbreviation

You can abbreviate a text by putting it inside opening <abbr> and closing </abbr> tags. If present, the title attribute must contain this full description and nothing else.

Example

```
<!DOCTYPE html>

<html>

<head>
<title>Text Abbreviation</title>

</head>
```

```
<body>
<p>My best friend's name is  <abbr title="Abhishek">Abhy</abbr>.</p>
</body>
</html>
```

This will produce the following result:

My best friend's name is Abhy.

Acronym Element

The **<acronym>** element allows you to indicate that the text between <acronym> and </acronym> tags is an acronym.

At present, the major browsers do not change the appearance of the content of the <acronym> element.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Acronym Example</title>
</head>
<body>
<p>This chapter covers marking up text in <acronym>XHTML</acronym>.</p>
</body>
</html>
```

This will produce the following result:

This chapter covers marking up text in XHTML.

Text Direction

The **<bdo>...</bdo>** element stands for Bi-Directional Override and it is used to override the current text direction.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Text Direction Example</title>

</head>

<body>

<p>This text will go left to right.</p>

<p><bdo dir="rtl">This text will go right to left.</bdo></p>

</body>

</html>
```

This will produce the following result:

This text will go left to right.

This text will go right to left.

Special Terms

The **<dfn>...</dfn>** element (or HTML Definition Element) allows you to specify that you are introducing a special term. It's usage is similar to italic words in the midst of a paragraph.

Typically, you would use the <dfn> element the first time you introduce a key term. Most recent browsers render the content of a <dfn> element in an italic font.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Special Terms Example</title>

</head>

<body>

<p>The following word is a <dfn>special</dfn> term.</p>

</body>
```

```
</html>
```

This will produce the following result:

The following word is a *special* term.

Quoting Text

When you want to quote a passage from another source, you should put it in between**<blockquote>...</blockquote>** tags.

Text inside a <blockquote> element is usually indented from the left and right edges of the surrounding text, and sometimes uses an italicized font.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Blockquote Example</title>

</head>

<body>

<p>The following description of XHTML is taken from the W3C Web site:</p>


<blockquote>XHTML 1.0 is the W3C's first Recommendation for XHTML, following on
from earlier work on HTML 4.01, HTML 4.0, HTML 3.2 and HTML 2.0.</blockquote>
</body>

</html>
```

This will produce the following result:

The following description of XHTML is taken from the W3C Web site:

```
XHTML 1.0 is the W3C's first Recommendation for XHTML, following on from earlier
work on HTML 4.01, HTML 4.0, HTML 3.2 and HTML 2.0.
```

Short Quotations

The **<q>...</q>** element is used when you want to add a double quote within a sentence.

Example

```
<!DOCTYPE html>

<html>

<head>
```

```
<title>Double Quote Example</title>

</head>

<body>

<p>Amit is in Spain, <q>I think I am wrong</q>.</p>

</body>

</html>
```

This will produce the following result:

Amit is in Spain, I think I am wrong.

Text Citations

If you are quoting a text, you can indicate the source placing it between an opening **<cite>**tag and closing **</cite>** tag

As you would expect in a print publication, the content of the <cite> element is rendered in italicized text by default.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Citations Example</title>

</head>

<body>

<p>This HTML tutorial is derived from <cite>W3 Standard for HTML</cite>.</p>

</body>

</html>
```

This will produce the following result:

This HTML tutorial is derived from *W3 Standard for HTML*.

Computer Code

Any programming code to appear on a Web page should be placed inside **<code>...</code>**tags. Usually the content of the <code> element is presented in a monospaced font, just like the code in most programming books.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Computer Code Example</title>

</head>

<body>

<p>Regular text. <code>This is code.</code> Regular text.</p>

</body>

</html>
```

This will produce the following result:

Regular text. `This is code.` Regular text.

Keyboard Text

When you are talking about computers, if you want to tell a reader to enter some text, you can use the **<kbd>...</kbd>** element to indicate what should be typed in, as in this example.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Keyboard Text Example</title>

</head>

<body>

<p>Regular text. <kbd>This is inside kbd element</kbd> Regular text.</p>

</body>

</html>
```

This will produce the following result:

Regular text. `This is inside kbd element` Regular text.

Programming Variables

This element is usually used in conjunction with the **<pre>** and **<code>** elements to indicate that the content of that element is a variable.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Variable Text Example</title>

</head>

<body>

<p><code>document.write("<var>user-name</var>")</code></p>

</body>

</html>
```

This will produce the following result:

```
document.write("user-name")
```

Program Output

The **<samp>...</samp>** element indicates sample output from a program, and script etc. Again, it is mainly used when documenting programming or coding concepts.

Example

```
<!DOCTYPE html>

<html>

<head>

<title>Program Output Example</title>

</head>

<body>

<p>Result produced by the program is <samp>Hello World!</samp></p>

</body>

</html>
```

This will produce the following result:

Result produced by the program is `Hello World!`

Address Text

The **<address>...</address>** element is used to contain any address.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Address Example</title>
</head>
<body>
<address>388A, Road No 22, Jubilee Hills -  Hyderabad</address>
</body>
</html>
```

This will produce the following result:

388A, Road No 22, Jubilee Hills - Hyderabad

7. HTML – META TAGS

HTML lets you specify metadata - additional important information about a document in a variety of ways. The META elements can be used to include name/value pairs describing properties of the HTML document, such as author, expiry date, a list of keywords, document author etc.

The **<meta>** tag is used to provide such additional information. This tag is an empty element and so does not have a closing tag but it carries information within its attributes.

You can include one or more meta tags in your document based on what information you want to keep in your document but in general, meta tags do not impact physical appearanceof the document so from appearance point of view, it does not matter if you include them ornot.

Adding Meta Tags to Your Documents

~~You can add metadata to your web pages by placing <meta> tags inside the header of the~~ document which is represented by **<head>** and **</head>** tags. A meta tag can have following attributes in addition to core attributes:

Attribute	Description

Name	Name for the property. Can be anything. Examples include, keywords, description, author, revised, generator etc.
content	Specifies the property's value.
scheme	Specifies a scheme to interpret the property's value (as declared in the content attribute).
http-equiv	Used for http response message headers. For example, http-equiv can be used to refresh the page or to set a cookie. Values include content-type, expires, refresh and set-cookie.

Specifying Keywords

You can use <meta> tag to specify important keywords related to the document and later these keywords are used by the search engines while indexing your webpage for searching purpose.

Example

Following is an example, where we are adding HTML, Meta Tags, Metadata as important keywords about the document.

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

This will produce the following result:

```
Hello HTML5!
```

Document Description

You can use <meta> tag to give a short description about the document. This again can be used by various search engines while indexing your webpage for searching purpose.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

Document Revision Date

You can use <meta> tag to give information about when last time the document was updated. This information can be used by various web browsers while refreshing your webpage.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
<meta name="revised" content="Tutorialspoint, 3/7/2014" />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

Document Refreshing

A <meta> tag can be used to specify a duration after which your web page will keep refreshing automatically.

Example

If you want your page keep refreshing after every 5 seconds then use the following syntax.

```
<!DOCTYPE html>

<html>
<head>

<title>Meta Tags Example</title>

<meta name="keywords" content="HTML, Meta Tags, Metadata" />

<meta name="description" content="Learning about Meta Tags." />

<meta name="revised" content="Tutorialspoint, 3/7/2014" />

<meta http-equiv="refresh" content="5" />

</head>

<body>

<p>Hello HTML5!</p>

</body>

</html>
```

Page Redirection

You can use <meta> tag to redirect your page to any other webpage. You can also specify a duration if you want to redirect the page after a certain number of seconds.

Example

Following is an example of redirecting current page to another page after 5 seconds. If you want to redirect page immediately then do not specify *content* attribute.

```
<!DOCTYPE html>

<html>

<head>

<title>Meta Tags Example</title>

<meta name="keywords" content="HTML, Meta Tags, Metadata" />

<meta name="description" content="Learning about Meta Tags." />

<meta name="revised" content="Tutorialspoint, 3/7/2014" />

<meta http-equiv="refresh" content="5; url=http://www.tutorialspoint.com" />

</head>

<body>
```

```
<p>Hello HTML5!</p>
</body>
</html>
```

WEB SERVER

A web server is a computer that stores web server software and a website's component files (for example, HTML documents, images, CSS stylesheets, and JavaScript files). A web server connects to the Internet and supports physical data interchange with other devices connected to the web.

A web server includes several parts that control how web users access hosted files. At a minimum, this is an HTTP server. An HTTP server is software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). An HTTP server can be accessed through the domain names of the websites it stores, and it delivers the content of these hosted websites to the end user's device.

At the most basic level, whenever a browser needs a file that is hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct (hardware) web server, the (software) HTTP server accepts the request, finds the requested document, and sends it back to the browser, also through HTTP. (If the server doesn't find the requested document, it returnsa 404 response instead.)

Basic representation of a client/server connection through HTTP
To publish a website, you need either a static or a dynamic web server.

A static web server, or stack, consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files as-is to your browser.

A dynamic web server consists of a static web server plus extra software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending content to your browser via the HTTP server.

For example, to produce the final webpages you see in the browser, the application server might fill an HTML template with content from a database. Sites like MDN or Wikipedia have thousands of webpages. Typically, these kinds of sites are composed of only a few HTML templates and a giant database, rather than thousands of static HTML documents. This setup makes it easier to maintain and deliver the content.

Git & Github

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Working with Git

- Initialize Git on a folder, making it a **Repository**
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered **modified**
- You select the modified files you want to **Stage**
- The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

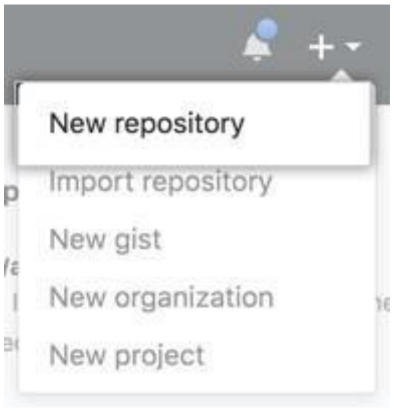
Github

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.
This tutorial teaches you GitHub essentials like repositories, branches, commits, and pull requests. You'll create your own Hello World repository

and learn GitHub's pull request workflow, a popular way to create and review code.
In this quickstart guide, you will:
• Create and use a repository
• Start and manage a new branch
• Make changes to a file and push them to GitHub as commits
• Open and merge a pull request

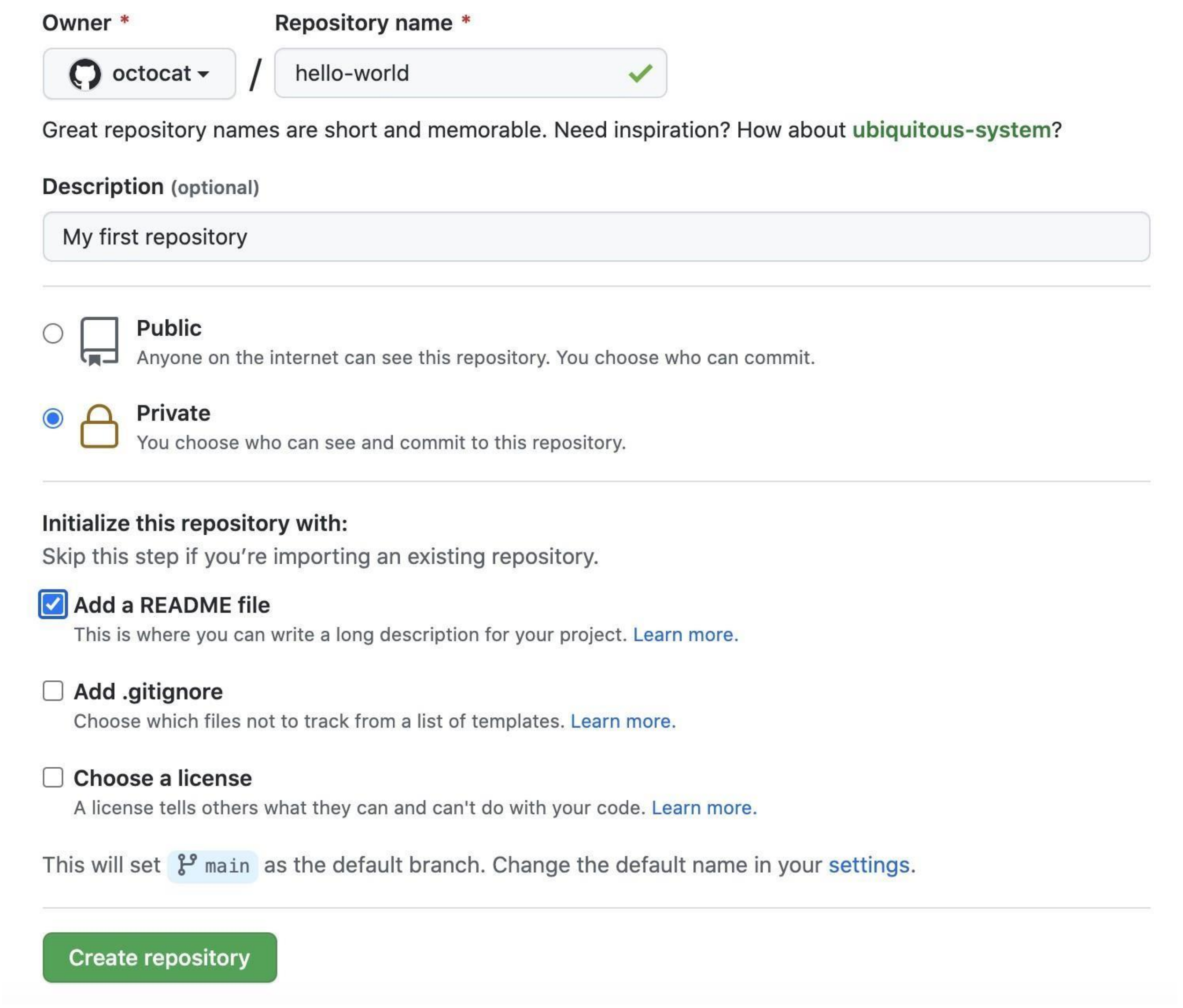
To complete this tutorial, you need a GitHub account and Internet access. You don't need to know how to code, use the command line, or install Git (the version control software that GitHub is built on). If you have a question about any of the expressions used in this guide, head on over to the glossary to find out more about our terminology.
Creating a repository
A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets -- anything your project needs. Often, repositories include a README file, a file with information about your project. README files are written in the plain text Markdown language. You can use this cheat sheet to get started with Markdown syntax. GitHub lets you add a README file at the same time you create your new repository. GitHub also offers other common options such as a license file, but you do not have to select any of them now.
Your hello-world repository can be a place where you store ideas, resources, or even share and discuss things with others.

1. In the upper-right corner of any page, use the drop-down menu, and select



2. New repository.
3. In the Repository name box, enter hello-world.
4. In the Description box, write a short description.

5. Select Add a README file.
6. Select whether your repository will be Public or Private.
7. Click Create repository.


Owner *
Repository name *
octocat / hello-world
Great repository names are short and memorable. Need inspiration? How about ubiquitous-system?
Description (optional)
My first repository

Public
Anyone on the internet can see this repository. You choose who can commit.

[✓ Private]
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

[✓] Add a README file
This is where you can write a long description for your project. Learn more.

[ ] Add .gitignore
Choose which files not to track from a list of templates. Learn more.

[ ] Choose a license
A license tells others what they can and can't do with your code. Learn more.

This will set `main` as the default branch. Change the default name in your settings.

Creating a branch

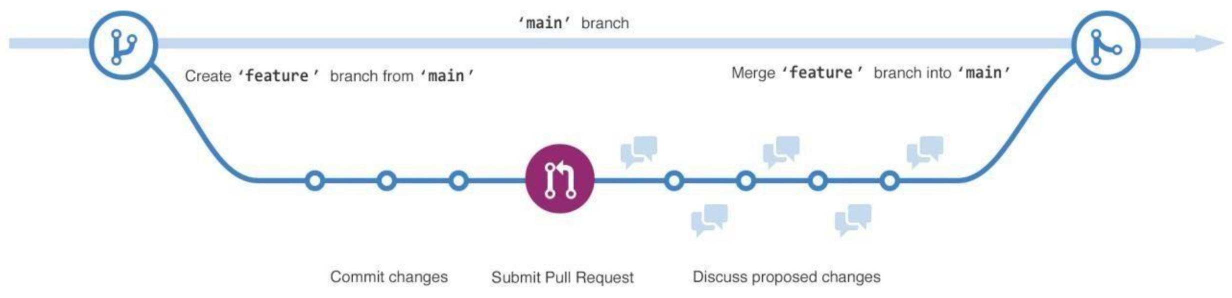
Branching lets you have different versions of a repository at one time.
By default, your repository has one branch named main that is considered to be the definitive branch. You can create additional branches off of main in your repository. You can use branches to have different versions of a projectat one time. This is helpful when you want to add new features to a project without changing the main source of code. The work done on different branches will not show up on the main branch until you merge it, which we

will cover later in this guide. You can use branches to experiment and make edits before committing them to main.

When you create a branch off the main branch, you're making a copy, or snapshot, of main as it was at that point in time. If someone else made changes to the main branch while you were working on your branch, you could pull in those updates.

This diagram shows:

- The main branch
- A new branch called feature
- The journey that feature takes before it's merged into main



Have you ever saved different versions of a file? Something like:

- story.txt
- story-edit.txt
- story-edit-reviewed.txt

Branches accomplish similar goals in GitHub repositories.

Here at GitHub, our developers, writers, and designers use branches for keeping bug fixes and feature work separate from our main (production) branch. When a change is ready, they merge their branch into main.

What is CSS?

While HTML is a **markup language** used to format/structure a web page, CSS is a **design language** that you use to make your web page look nice and presentable.
CSS stands for **Cascading Style Sheets**, and you use it to improve the appearance of a web page. By adding thoughtful CSS styles, you make your page more attractive and pleasant for the end user to view and use.
Imagine if human beings were just made to have skeletons and bare bones – how would that look? Not nice if you ask me. So CSS is like our skin, hair, and general physical appearance.

You can also use CSS to layout elements by positioning them in specified areas of your page.

To access these elements, you have to "select" them. You can select a single or multiple web elements and specify how you want them to look or be positioned.

The rules that govern this process are called CSS **selectors**.
With CSS you can set the colour and background of your elements, as well as the typeface, margins, spacing, padding and so much more.

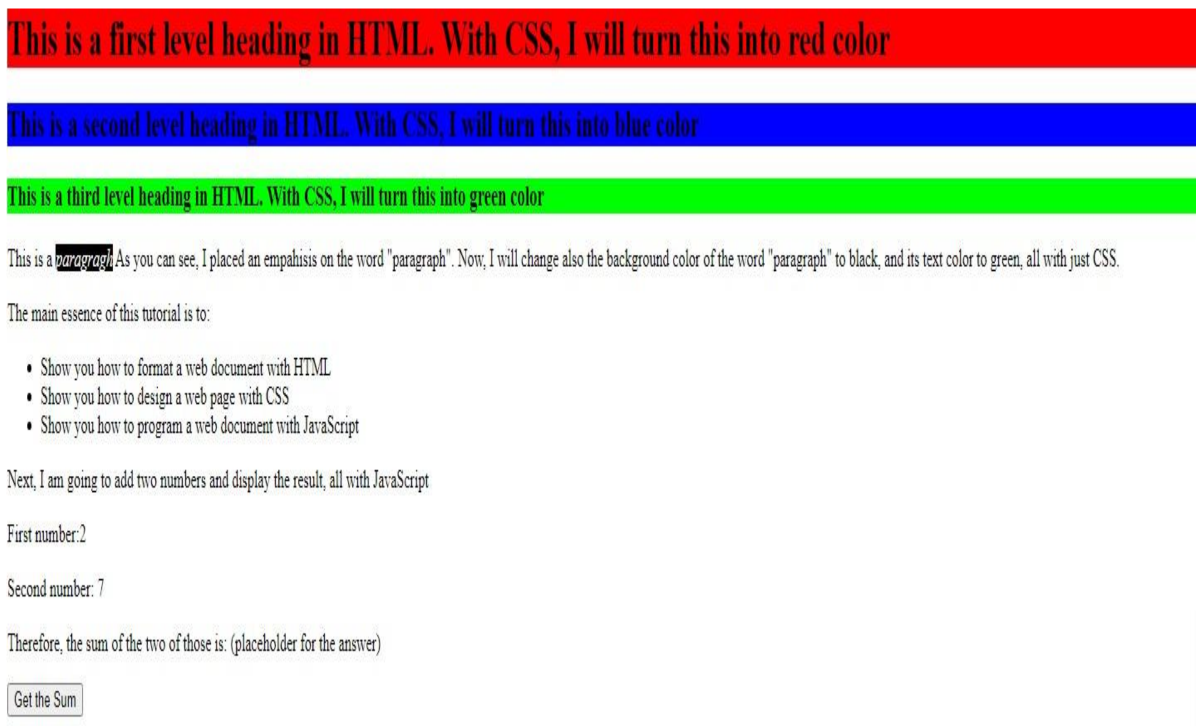
If you remember our example HTML page, we had elements which were pretty self-explanatory. For example, I stated that I would change the color of the level one heading `h1` to red.
To illustrate how CSS works, I will be sharing the code which sets the background-color of the three levels of headers to red, blue, and green respectively:

```
h1 {
   background-color: #ff0000;
}
```

```
h2 {
   background-color: #0000FF;
}

h3 {
   background-color: #00FF00;
}

em  {
   background-color: #000000;
   color: #ffffff;
}
```

localhost:3000/styles.css

The above style, when applied, will change the appearance of our web page to this:



This is a first level heading in HTML. With CSS, I will turn this into red color

This is a second level heading in HTML. With CSS, I will turn this into blue color

This is a third level heading in HTML. With CSS, I will turn this into green color

This is a *paragraph*. As you can see, I placed an empahisis on the word "paragraph". Now, I will change also the background color of the word "paragraph" to black, and its text color to green, all with just CSS.

The main essence of this tutorial is to:

- Show you how to format a web document with HTML
- Show you how to design a web page with CSS
- Show you how to program a web document with JavaScript

Next, I am going to add two numbers and display the result, all with JavaScript

First number:2

Second number: 7

Therefore, the sum of the two of those is: (placeholder for the answer)

Cool, right?

We access each of the elements we want to work on by "selecting" them. The `h1` selects all level 1 headings in the page, the `h2` selects the level 2 elements, and so on. You can select any single HTML element you want and specify how you want it to look or be positioned.
Want to learn more about CSS? You can check out the second part of freeCodeCamp's Responsive Web Design certification to get started.

What is JavaScript?

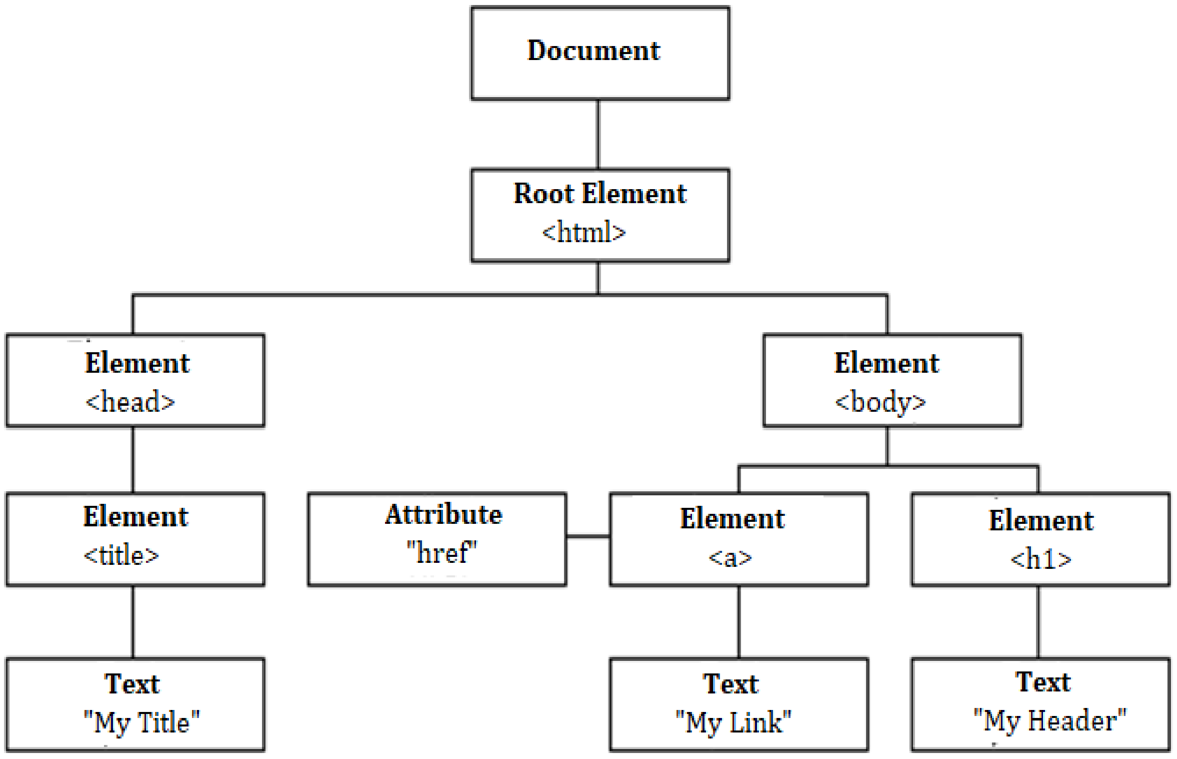
Now, if HTML is the **markup language** and CSS is the **design language**, then JavaScript is the **programming language.**
If you don't know what programming is, think of certain actions you take in your daily life:

When you sense danger, you run. When you are hungry, you eat. When you are tired, you sleep. When you are cold, you look for warmth. When crossing a busy road, you calculate the distance of vehicles away from you.

Your brain has been programmed to react in a certain way or do certain things whenever something happens. In this same way, you can program your web page or individual elements to react a certain way and to do something when something else (an event) happens.

You can program actions, conditions, calculations, network requests, concurrent tasks and many other kinds of instructions.

You can access any elements through the Document Object Model API (DOM) and make them change however you want them to.
The DOM is a tree-like representation of the web page that gets loaded into the browser.



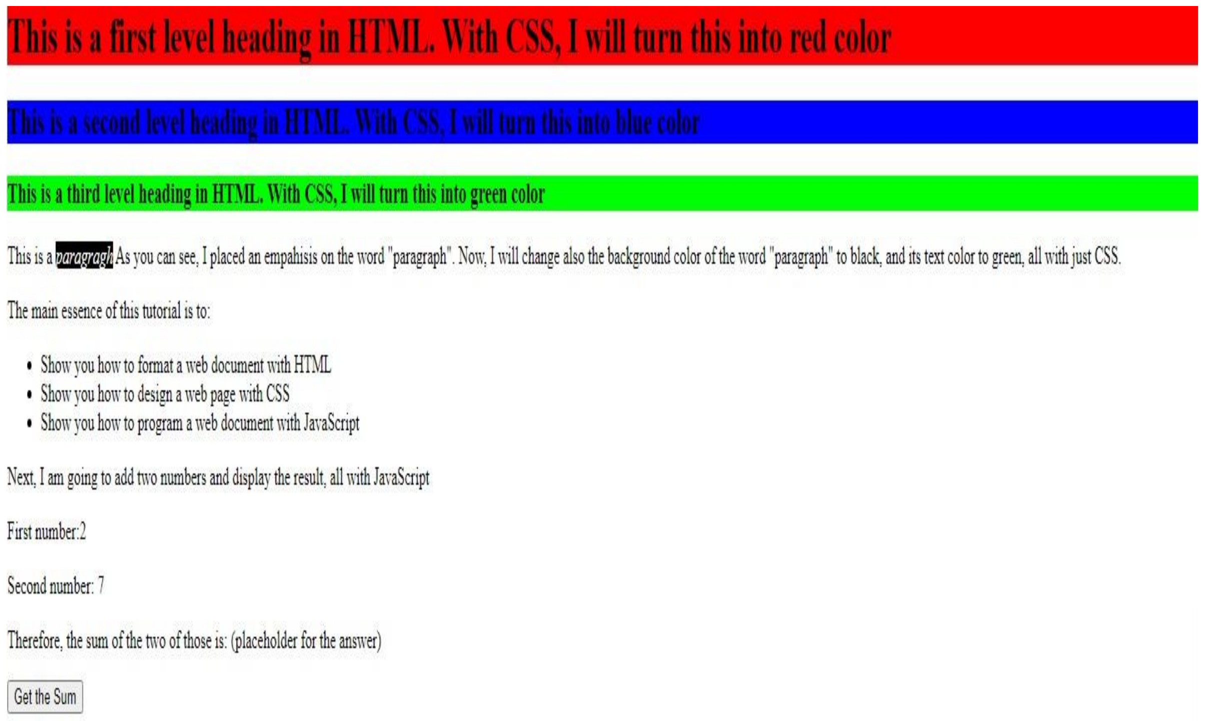
Eac

h element on the web page is represented on the DOM

Thanks to the DOM, we can use methods like `getElementById()` to access elements from our web page.

JavaScript allows you to make your webpage **"think and act"**, which is what programming is all about.

If you remember from our example HTML page, I mentioned that I was going to sum up the two numbers displayed on the page and then display the result in the place of the placeholder text. The calculation runs once the button gets clicked.



This is a first level heading in HTML. With CSS, I will turn this into red color

This is a second level heading in HTML. With CSS, I will turn this into blue color

This is a third level heading in HTML. With CSS, I will turn this into green color

This is a paragraph As you can see, I placed an empahisis on the word "paragraph". Now, I will change also the background color of the word "paragraph" to black, and its text color to green, all with just CSS.

The main essence of this tutorial is to:

- Show you how to format a web document with HTML
- Show you how to design a web page with CSS
- Show you how to program a web document with JavaScript

Next, I am going to add two numbers and display the result, all with JavaScript

First number:2

Second number: 7

Therefore, the sum of the two of those is: (placeholder for the answer)

Clicking the "Get the sum" button will display the sum of 2 and 7

This code illustrates how you can do calculations with JavaScript:

```
function displaySum()  {
   let firstNum = Number(document.getElementById('firstNum').innerHTML)
   let secondNum = Number(document.getElementById('secondNum').innerHTML)


   let total = firstNum + secondNum;
   document.getElementById("answer").innerHTML = ` ${firstNum} + ${secondNum}, equals to ${total}`
;
}

document.getElementById('sumButton').addEventListener("click", displaySum);
```

Remember what I told you about HTML attributes and their uses? This code displays just that.

The `displaySum` is a function which gets both items from the web page, converts them to numbers (with the Number method), sums them up, and passes them in as inner values to another element.

The reason we were able to access these elements in our JavaScript was because we had set unique attributes on them, to help us identify them.

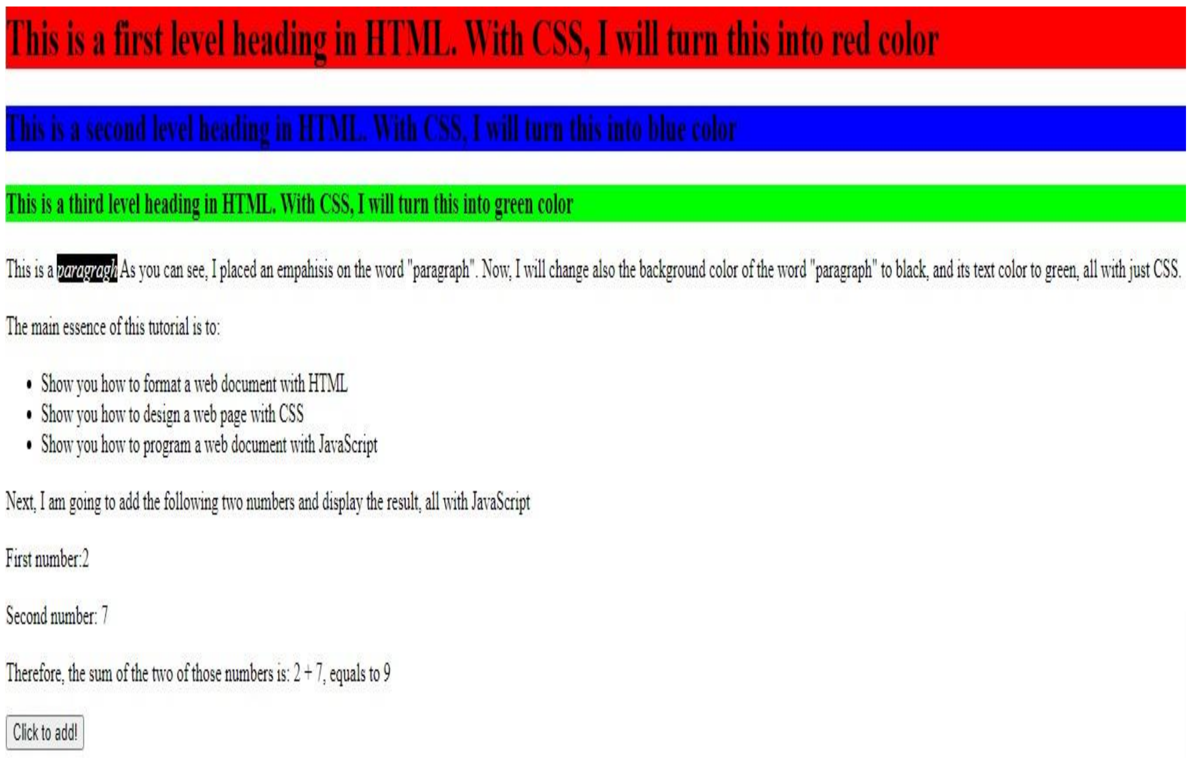
So thanks to this:

```
// id attribute has been set in all three

<span id= "firstNum">2</span> <br>
    ...<span id= "secondNum">7</span>
    ...... <span id= "answer">(placeholder for the answer)</span>
```

We were able to do this:

```
//getElementById will get all HTML elements by a specific "id" attribute
...
let firstNum = Number(document.getElementById('firstNum').innerHTML)
  let secondNum = Number(document.getElementById('secondNum').innerHTML)

  let total = firstNum + secondNum;
  document.getElementById("answer").innerHTML = ` ${firstNum} + ${secondNum}, equals to ${total}`
;
```

Finally, upon clicking the button, you will see the sum of the two numbers on the newly updated page:



This is a first level heading in HTML. With CSS, I will turn this into red color

This is a second level heading in HTML. With CSS, I will turn this into blue color

This is a third level heading in HTML. With CSS, I will turn this into green color

This is a *paragragh* As you can see, I placed an empahisis on the word "paragraph". Now, I will change also the background color of the word "paragraph" to black, and its text color to green, all with just CSS.

The main essence of this tutorial is to:

- Show you how to format a web document with HTML
- Show you how to design a web page with CSS
- Show you how to program a web document with JavaScript

Next, I am going to add the following two numbers and display the result, all with JavaScript

First number:2

Second number: 7

Therefore, the sum of the two of those numbers is: 2 + 7, equals to 9

2
plus 7 is equals to 9

If you want to get started with JavaScript, you can check out freeCodeCamp's JavaScript Algorithms and Data Structures certification. And you can use this great Intro to JS course to supplement your learning.

How to Put HTML, CSS, and JavaScript Together

Together, we use these three languages to format, design, and program web pages.

And when you link together some web pages with hyperlinks, along with all their assets like images, videos, and so on that are on the server computer, it gets rendered into a **website**.
This rendering typically happens on the front end, where the users can see what's being displayed and interact with it.

On the other hand, data, especially sensitive information like passwords, are stored and supplied from the back end part of the website. This is the part of a website which exists only on the server computer, and isn't displayed on the front-end browser. There, the user cannot see or readily access that information.

Wrapping Up

As a web developer, the three main languages we use to build websites are HTML, CSS, and JavaScript.

JavaScript is the programming language, we use HTML to structure the site, and we use CSS to design and layout the web page.

These days, CSS has become more than just a design language, though. You can actually implement animations and smooth transitions with just CSS.

In fact, you can do some basic programming with CSS too. An example of this is when you use media queries, where you define different style rules for different kinds of screens (resolutions).

JavaScript has also grown beyond being used just in the browser as well. We now use it on the server thanks to **Node.js**.
But the basic fact remains: HTML, CSS, and JavaScript are the main languages of the Web.

So that's it. The languages of the Web explained in basic terms. I really hope you got something useful from this article.

Web Servers Shell:
A web shell is a shell-like interface that enables a web server to be remotely accessed, often for the purposes of cyberattacks.[1] A web shell is unique in that a web browser is used to interact with it.[2][3]

A web shell could be programmed in any programming language that is supported on a server. Web shells are most commonly written in the PHP programming language due to the widespread usage of PHP for web applications. However, Active Server Pages, ASP.NET, Python, Perl, Ruby, and Unix shell scripts are also used, although these languages are less commonly used.[1][2][3]

Using network monitoring tools, an attacker can find vulnerabilities that can potentially allow delivery of a web shell. These vulnerabilities are often present in applications that are run on a web server.[2]

An attacker can use a web shell to issue shell commands, perform privilege escalation on the web server, and the ability to upload, delete, download, and execute files to and from the web server.[2

UNIX CLI Version control

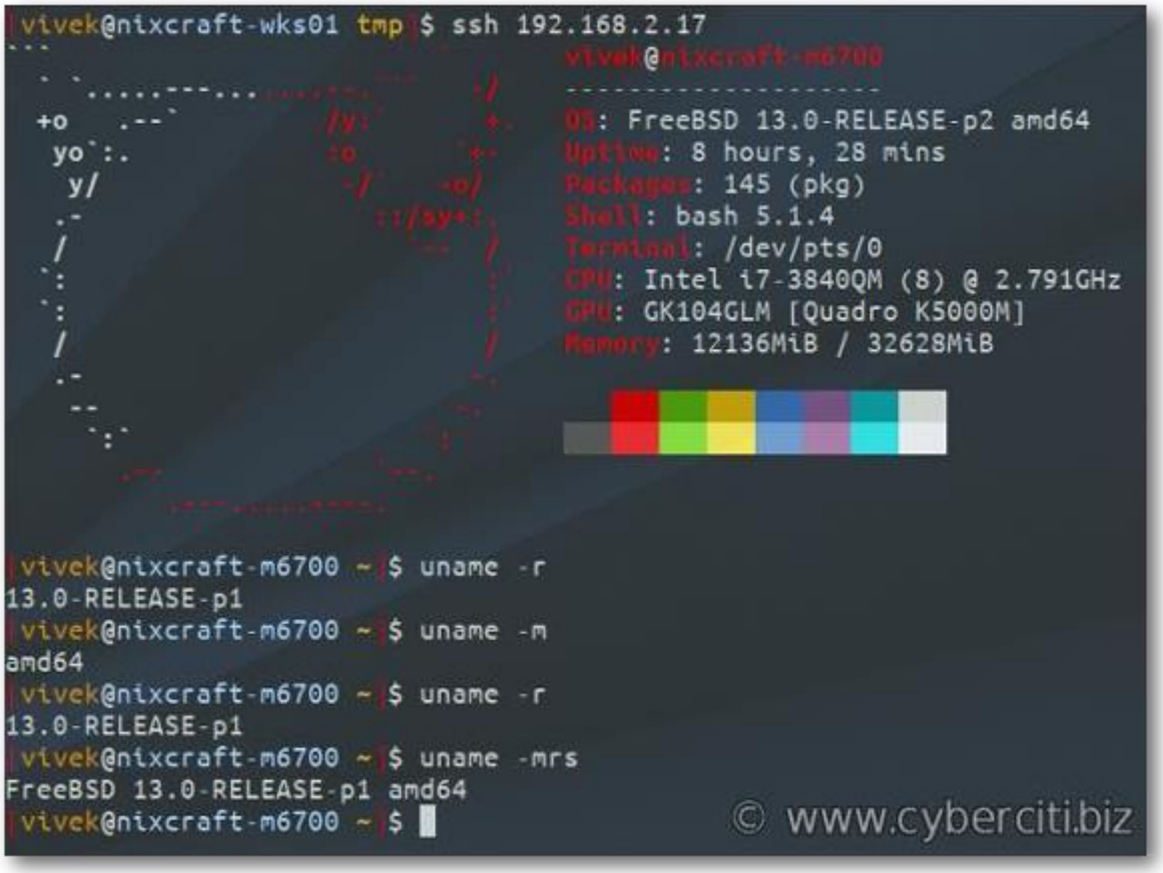
We have various commands that help us to find out the Unix variant, type, and machine. The most common Unix command is uname, and we will talk about it first, followed by variant-specific information.

ADVERTISEMENT

Checking Unix version

1. Open the terminal application and then type the following uname command:
   ```
   uname
   uname -a
   ```
2. Display the current release level (OS Version) of the Unix operating system.
   ```
   uname -r
   ```
3. You will see Unix OS version on screen. To see architecture of Unix, run:
   ```
   uname -m
   ```

Here is outputs from my FreeBSD Unix server:



```
vivek@nixcraft-wks01 tmp $ ssh 192.168.2.17
                                        vivek@nixcraft-m6700
                                        --------------------
                                        OS: FreeBSD 13.0-RELEASE-p2 amd64
                                        Uptime: 8 hours, 28 mins
                                        Packages: 145 (pkg)
                                        Shell: bash 5.1.4
                                        Terminal: /dev/pts/0
                                        CPU: Intel i7-3840QM (8) @ 2.791GHz
                                        GPU: GK104GLM [Quadro K5000M]
                                        Memory: 12136MiB / 32628MiB

vivek@nixcraft-m6700 ~ $ uname -r
13.0-RELEASE-p1
vivek@nixcraft-m6700 ~ $ uname -m
amd64
vivek@nixcraft-m6700 ~ $ uname -r
13.0-RELEASE-p1
vivek@nixcraft-m6700 ~ $ uname -mrs
FreeBSD 13.0-RELEASE-p1 amd64
vivek@nixcraft-m6700 ~ $
```

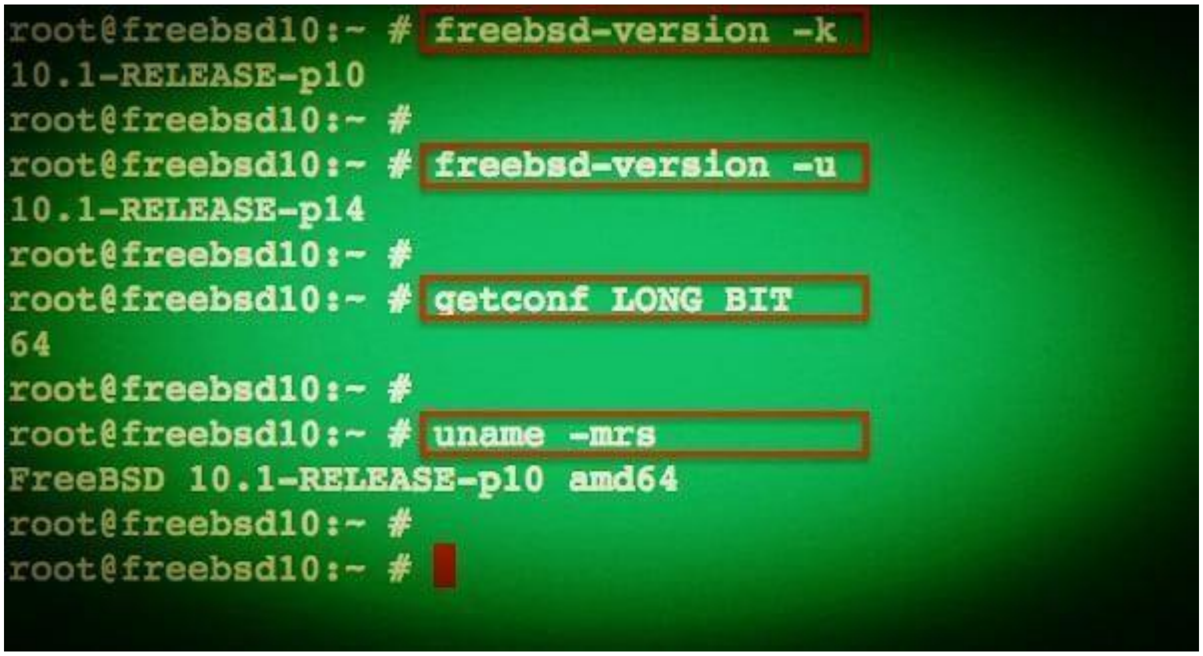
Examples

Although uname available on all Unix variants, there are other ways to display OS versions and names. Let us look at operating system-specific information.

How to check FreeBSD unix version

Type the following command to determine the version and patch level:

```
freebsd-version
freebsd-version -k
freebsd-version -r
freebsd-version -u
```



```
root@freebsd10:~ # freebsd-version -k
10.1-RELEASE-p10
root@freebsd10:~ #
root@freebsd10:~ # freebsd-version -u
10.1-RELEASE-p14
root@freebsd10:~ #
root@freebsd10:~ # getconf LONG BIT
64
root@freebsd10:~ #
root@freebsd10:~ # uname -mrs
FreeBSD 10.1-RELEASE-p10 amd64
root@freebsd10:~ #
root@freebsd10:~ #
```

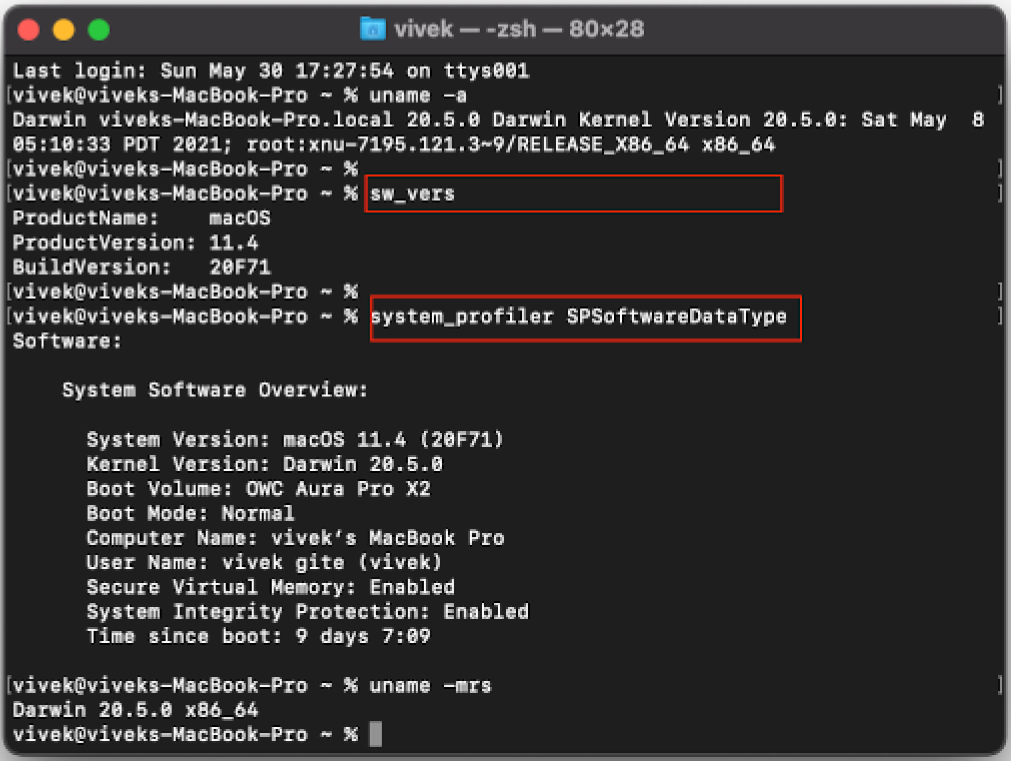
Show FreeBSD Unix Version

A note about macOS

Open the macOS Terminal app and then type any one of the following command to print macOS or Mac OS X version:

```
sw_vers
# OR #
```

```
system_profiler SPSoftwareDataType
```



```
Last login: Sun May 30 17:27:54 on ttys001
vivek@viveks-MacBook-Pro ~ % uname -a
Darwin viveks-MacBook-Pro.local 20.5.0 Darwin Kernel Version 20.5.0: Sat May  8
05:10:33 PDT 2021; root:xnu-7195.121.3~9/RELEASE_X86_64 x86_64
vivek@viveks-MacBook-Pro ~ %
vivek@viveks-MacBook-Pro ~ % sw_vers
ProductName:    macOS
ProductVersion: 11.4
BuildVersion:   20F71
vivek@viveks-MacBook-Pro ~ %
vivek@viveks-MacBook-Pro ~ % system_profiler SPSoftwareDataType
Software:

    System Software Overview:

      System Version: macOS 11.4 (20F71)
      Kernel Version: Darwin 20.5.0
      Boot Volume: OWC Aura Pro X2
      Boot Mode: Normal
      Computer Name: vivek's MacBook Pro
      User Name: vivek gite (vivek)
      Secure Virtual Memory: Enabled
      System Integrity Protection: Enabled
      Time since boot: 9 days 7:09

vivek@viveks-MacBook-Pro ~ % uname -mrs
Darwin 20.5.0 x86_64
vivek@viveks-MacBook-Pro ~ %
```

HP-UX Unix

Use the swlist command as follows for determining your HP-UX Unix system version:

```
swlist
swlist | grep -i oe
swlist HPUX*OE*
swlist HPUX*OE*
```

You will see something as follows:

```
HPUX11i-OE-Ent B.11.23.0606 HP-UX Enterprise Operating Environment
Component
```

OR

```
HPUX11i-TCOE B.11.23.0409 HP-UX Technical Computing OE Component
```

To see machine model from HP, type:

```
model
machinfo
getconf MACHINE_MODEL
```

Oracle or Sun Solaris OS

Verifying Operating system version on Oracle or Sun Solaris Unix is easy:

```
uname
uname -a
uname -r
# use the cat command #
cat /etc/release
```

You will get info such as `Oracle Solaris 11 (5.11)` OR `Oracle Solaris 11.1 SPARC`.

IBM AIX Unix

To view the base level of the Unix system OS from IBM, type:

```
uname
uname -a
uname -r
oslevel
prtconf
```

See oslvel AIX command man-page for more info.

Summing up

The uname and other Unix command commands can help you determine information about your Unix server or desktop, including its hardware type, machine model, operating system version. The uname and other options various. Hence, see the following man pages:

```
man uname
```

Git & Github

Introduction to Git

For installation purposes on ubuntu, you can refer to this article: How to Install, Configure and Use GIT on Ubuntu?

Git is a distributed version control system. So, What is a Version Control System?

A version Control system is a system that maintains different versions of your project when we work in a team or as an individual. (system managing changes to files) As the project progresses, new features get added to it. So, a version control system maintains all the different versions of your project for you and you can roll back to any version you want without causing any trouble to you for maintaining different versions by giving names to it like MyProject, MyProjectWithFeature1, etc.

Distributed Version control system means every collaborator(any developer working on a team project)has a local repository of the project in his/her local machine unlike central where team members should have an internet connection to every time update their work to the main central repository.

So, by distributed we mean: the project is distributed. A repository is an area that keeps all your project files, images, etc. In terms of Github: different versions of projects correspond to commits.

For more details on introduction to Github, you can refer: Introduction to Github

Git Repository Structure

It consists of 4 parts:

Working directory: This is your local directory where you make the project (write code) and make changes to it.
Staging Area (or index): this is an area where you first need to put your project before committing. This is used for code review by other team members.
Local Repository: this is your local repository where you commit changes to the project before pushing them to the central repository on Github. This is what is provided by the distributed version control system. This corresponds to the .git folder in our directory.
Central Repository: This is the main project on the central server, a copy of which is with every team member as a local repository.
All the repository structure is internal to Git and is transparent to the developer.

Some commands which relate to repository structure:

```
// transfers your project from working directory
// to staging area.
git add .

// transfers your project from staging area to
// Local Repository.
git commit -m "your message here"


// transfers project from local to central repository.
// (requires internet)
git push
```

Github
Github basically is a for-profit company owned by Microsoft, which hosts Git repositories online. It helps users share their git repository online, with other users, or access it remotely. You can also host a public repository for free on Github.

User share their repository online for various reasons including but not limited to project deployment, project sharing, open source contribution, helping out the community and many such.

Accessing Github central repository via HTTPS or SSH
Here, transfer project means transfer changes as git is very lightweight and works on changes in a project. It internally does the transfer by using Lossless Compression Techniques and transferring compressed files. Https is the default way to access Github central repository.

By git remote add origin http_url: remote means the remote central repository. Origin corresponds to your central repository which you need to define (hereby giving HTTPS URL) in order to push changes to Github.
Via SSH: connect to Linux or other servers remotely.
If you access Github by ssh you don’t need to type your username and password every time you push changes to GitHub.

Terminal commands:

ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
This does the ssh key generation using RSA cryptographic algorithm.

eval "$(ssh-agent -s)" -> enable information about local login session.

ssh-add ~/.ssh/id_rsa -> add to ssh key.
cat ~/.ssh/id_rsa (use .pub file if not able to connect)
add this ssh key to github.

Now, go to github settings -> new ssh key -> create key

ssh -T git@github.com -> activate ssh key (test connection)

Refresh your github Page.
Working with git – Important Git commands
Git user configuration (First Step)

```
git --version (to check git version)
git config --global user.name "your name here"
git config --global user.email "your email here"
```

These are the information attached to commits.

Initialize directory

```
git init
```

initializes your directory to work with git and makes a local repository. .git folder is made (OR)

```
git clone http_url
```

This is done if we have an existing git repository and we want to copy its content to a new place.

Connecting to the remote repository

```
git remote add origin http_url/ssh_url
```

connect to the central repo to push/pull. pull means adopting the changes on the remote repository to your local repository. push merges the changes from your local repository to the remote repository.

```
git pull origin master
```

One should always first pull contents from the central repo before pushing so that you are updated with other team members’ work. It helps prevent merge conflicts. Here, master means the master branch (in Git).

Stash Area in git

git stash
Whichever files are present in the staging area, it will move that files to stash before committing it.

git stash pop
Whenever we want files for commit from stash we should use this command.

git stash clear
By doing this, all files from stash area is been deleted.

Steps to add a file to a remote Repository:

First, your file is in your working directory, Move it to the staging area by typing:

git add -A (for all files and folders)
#To add all files only in the current directory
git add .
git status: here, untracked files mean files that you haven't added to the staging area. Changes are not staged for commit means you have staged the file earlier than you have made changes in that files in your working directory and the changes need to be staged once more. Changes ready to be committed: these are files that have been committed and are ready to be pushed to the central repository.

git commit -a -m "message for commit"
-a: commit all files and for files that have been
 staged earlier need not to be git add once more
-a option does that automatically.
git push origin master -> pushes your files to

github master branch
git push origin anyOtherBranch -> pushes any
other branch to github.
git log ; to see all your commits
git checkout commitObject(first 8 bits) file.txt->
revert back to this previous commit for file file.txt
Previous commits m=ight be seen through the git log command.

HEAD -> pointer to our latest commit.
Ignoring files while committing

In many cases, the project creates a lot of logs and other irrelevant files which are to be ignored. So to ignore those files, we have to put their names in“.gitignore” file.

touch .gitignore
echo "filename.ext" >>.gitignore
#to ignore all files with .log extension
echo "*.log" > .gitignore
Now the filenames written in the .gitignore file would be ignored while pushing a new commit. To get the changes between commits, commit, and working tree.

git diff
'git diff' command compares the staging area with the working directory and tells us the changes made. It compares the earlier information as well as the current modified information.

Branching in Git

create branch ->
git branch myBranch
or

git checkout -b myBranch -> make and switch to the
branch myBranch
Do the work in your branch. Then,

git checkout master ; to switch back to master branch
Now, merge contents with your myBranch By:

git merge myBranch (writing in master branch)
This merger makes a new commit.

Another way

git rebase myBranch
This merges the branch with the master in a serial fashion. Now,

git push origin master
Contributing to Open Source
Open Source might be considered as a way where user across the globe may share their opinions, customizations or work together to solve an issue or to complete the desired project together. Many companies host there repositories online on Github to allow access to developers to make changes to their product. Some companies(not necessarily all) rewards their contributors in different ways.

You can contribute to any open source project on Github by forking it, making desired changes to the forked repository, and then opening a pull request. The project owner will review your project and will ask to improve it or will merge it.

UNIT - II

Frontend Development: Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS AJAX for data exchange with server jQuery Framework jQuery events, UI components etc. JSON data format.

Javascript basics OOPS:

As JavaScript is widely used in Web Development, in this article we will explore some of the **Object Oriented** mechanisms supported by **JavaScript** to get the most out of it. Some of the common interview questions in JavaScript on OOPS include:

- How is Object-Oriented Programming implemented in JavaScript?
- How does it differ from other languages?
- Can you implement Inheritance in JavaScript?

and so on…

There are certain features or mechanisms which make a Language Object-Oriented like:

OOPs Concept in JavaScript

Object	Classes	Encapsulation
Abstraction	Inheritance	Polymorphism

Let's dive into the details of each one of them and see how they are implemented in JavaScript.

Object: An Object is a **unique** entity that contains **properties** and **methods**. For example "a car" is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving. The characteristics of an Object are called Properties in Object-Oriented Programming and the actions are called methods. An Object is an **instance** of a class. Objects are everywhere in JavaScript, almost every element is an Object whether it is a function, array, or string.

Note: A Method in javascript is a property of an object whose value is a function.

The object can be created in two ways in JavaScript:

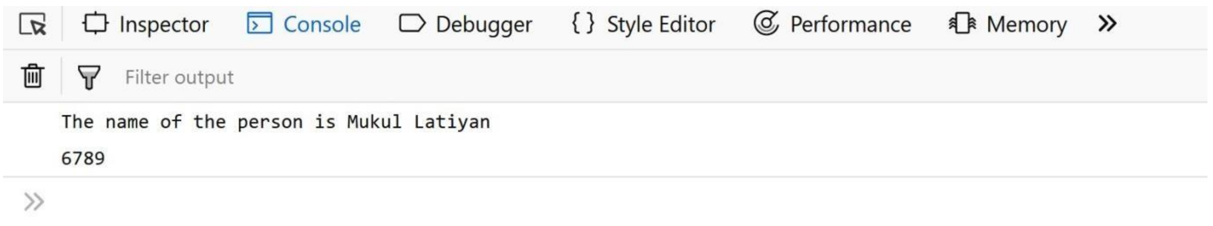
- **Object Literal**
- **Object Constructor**

Example: Using an Object Literal.

- Javascript

```
// Defining object
let person = {
    first_name:'Mukul',
    last_name: 'Latiyan',

    //method
    getFunction : function(){
        return (`The name of the person is
          ${person.first_name} ${person.last_name}`)
    },
    //object within object
    phone_number : {
        mobile:'12345',
        landline:'6789'
    }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

Output:



```
The name of the person is Mukul Latiyan
6789
```

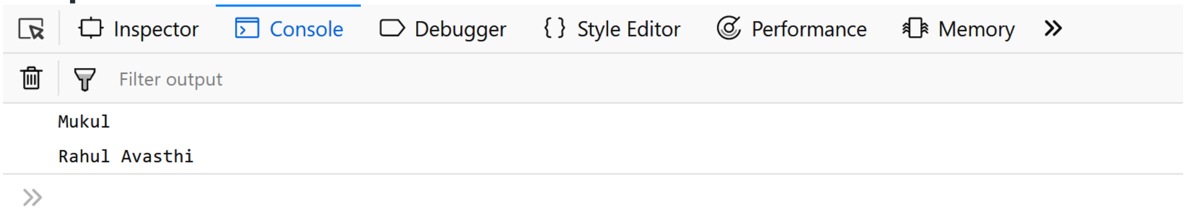
E

xample: Using an Object Constructor.

- Javascript

```
// Using a constructor
function person(first_name,last_name){
   this.first_name = first_name;
   this.last_name = last_name;
}
// Creating new instances of person object
let person1 = new person('Mukul','Latiyan');
let person2 = new person('Rahul','Avasthi');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

Output:



```
Mukul
Rahul Avasthi
```

N

ote: The JavaScript Object.create() Method creates a new object, using an existing object as the prototype of the newly created object.

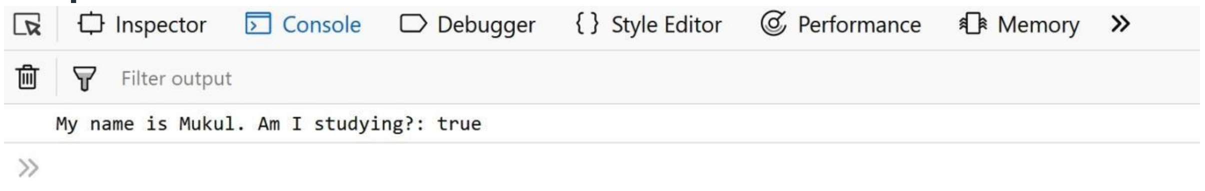
Example:

- Javascript

```
// Object.create() example a
// simple object with some properties
const coder = {
    isStudying : false,
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I
          studying?: ${this.isStudying}.`)
    }
}
// Object.create() method
const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"
me.name = 'Mukul';

// Inherited properties can be overwritten
me.isStudying = true;
```

```
    me.printIntroduction();
```

Output:



```
My name is Mukul. Am I studying?: true
```

Classes: Classes are **blueprints** of an Object. A class can have many Objects because the class is a **template** while Objects are **instances** of the class or the concrete implementation.
Before we move further into implementation, we should know unlike other Object Oriented languages there are **no classes in JavaScript** we have only Object. To be more precise, JavaScript is a prototype-based Object Oriented Language, which means it doesn't have classes, rather it defines behaviors using a constructor function and then reuses it using the prototype.
Note: Even the classes provided by ECMA2015 are objects.
JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.

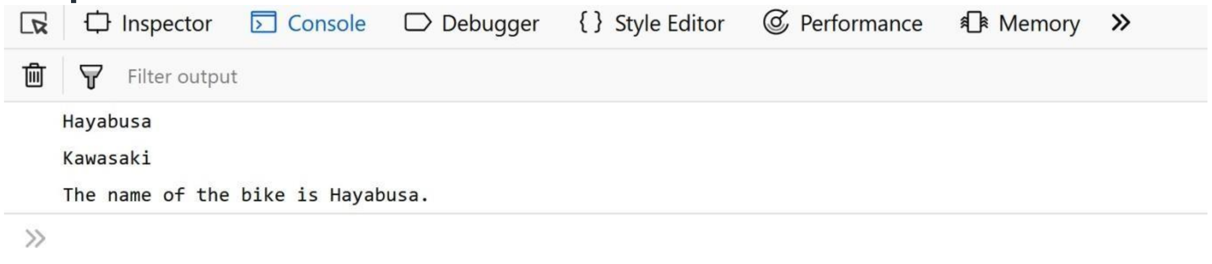
-Mozilla Developer Network

Example: Let's use ES6 classes then we will look at the traditional way of defining an Object and simulate them as classes.

- Javascript

```
// Defining class using es6
class Vehicle {
  constructor(name, maker, engine) {
    this.name = name;
    this.maker = maker;
    this.engine = engine;
  }
  getDetails(){
      return (`The name of the bike is ${this.name}.`)
  }
}
// Making object with the help of the constructor
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);   //   Hayabusa
console.log(bike2.maker);  //   Kawasaki
console.log(bike1.getDetails());
```

Output:



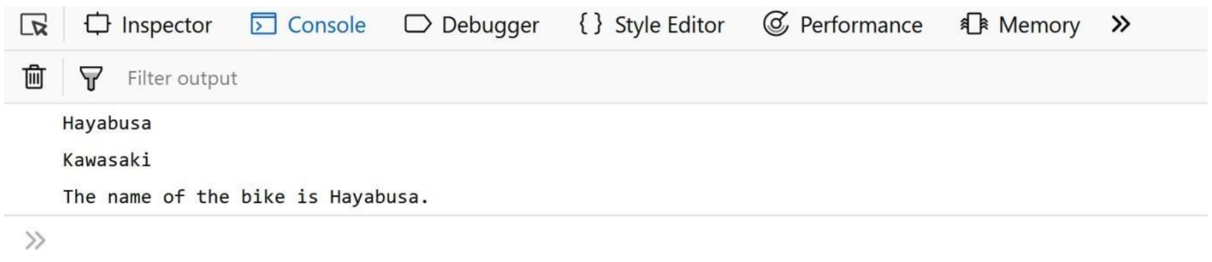
```
Hayabusa
Kawasaki
The name of the bike is Hayabusa.
```

Example: Traditional Way of defining an Object and simulating them as classes.

- Javascript

```
// Defining class in a Traditional Way.
function Vehicle(name,maker,engine){
    this.name = name,
    this.maker = maker,
    this.engine = engine
};

Vehicle.prototype.getDetails = function(){
    console.log('The name of the bike is '+ this.name);
}

let bike1 = new Vehicle('Hayabusa','Suzuki','1340cc');
let bike2 = new Vehicle('Ninja','Kawasaki','998cc');

console.log(bike1.name);
console.log(bike2.maker);
console.log(bike1.getDetails());
```

Output:



```
Hayabusa
Kawasaki
The name of the bike is Hayabusa.
```

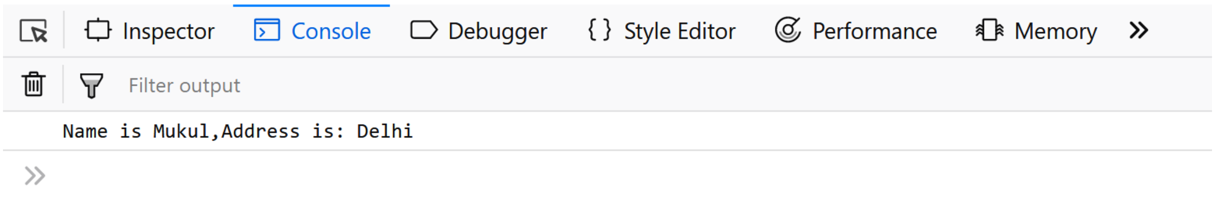
As seen in the above example it is much simpler to define and reuse objects in ES6. Hence, we would be using ES6 in all of our examples.

Abstraction: Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
Encapsulation: The process of **wrapping properties and functions** within a **single unit** is known as encapsulation.
Example: Let's understand encapsulation with an example.

- Javascript

```
// Encapsulation example
class person{
    constructor(name,id){
        this.name = name;
        this.id = id;
    }
    add_Address(add){
        this.add = add;
    }
    getDetails(){
        console.log(`Name is ${this.name},
        Address is: ${this.add}`);
    }
}

let person1 = new person('Mukul',21);
person1.add_Address('Delhi');
person1.getDetails();
```

Output: In this example, we simply create a person Object using the constructor, Initialize its properties and use its functions. We are not bothered by the implementation details. We are working with an Object's interface without considering the implementation details.



```
Name is Mukul,Address is: Delhi
```

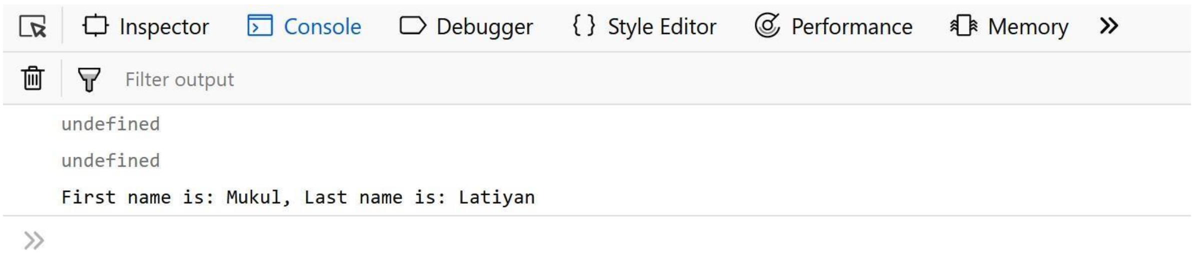
Sometimes encapsulation refers to the **hiding of data** or **data Abstraction** which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript, there are certain ways by which we can restrict the scope of variables within the Class/Object.

Example:

- Javascript

```
// Abstraction example
function person(fname,lname){
    let firstname = fname;
    let lastname = lname;

    let getDetails_noaccess = function(){
        return (`First name is: ${firstname} Last
            name is: ${lastname}`);
    }

    this.getDetails_access = function(){
        return (`First name is: ${firstname}, Last
            name is: ${lastname}`);
    }
}
let person1 = new person('Mukul','Latiyan');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```

Output: In this example, we try to access some property(person1.firstname) and functions(person1.getDetails_noaccess) but it returns undefined while there is a method that we can access from the person object(person1.getDetails_access()). By changing the way we define a function we can restrict its scope.



```
undefined
undefined
First name is: Mukul, Last name is: Latiyan
```

Inheritance: It is a concept in which some properties and methods of an Object are being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Objects inherit Objects i.e. certain features (property and methods) of one object can be reused by other Objects.

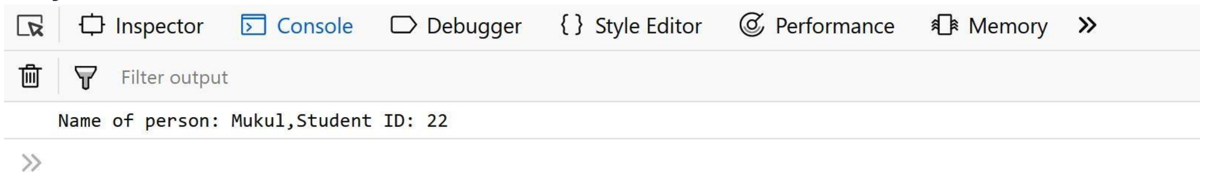
Example: Let's understand inheritance and polymorphism with an example.

- Javascript

```
// Inheritance example
class person{
    constructor(name){
        this.name = name;
    }
    // method to return the string
    toString(){
        return (`Name of person: ${this.name}`);
    }
}
class student extends person{
    constructor(name,id){
        // super keyword for calling the above
        // class constructor
        super(name);
        this.id = id;
    }
    toString(){
        return (`${super.toString()},
        Student ID: ${this.id}`);
    }
}
let student1 = new student('Mukul',22);
console.log(student1.toString());
```

Output: In this example, we define a Person Object with certain properties and methods and then we inherit the Person Object in the

Student Object and use all the properties and methods of the person Object as well as define certain properties and methods for the Student Object.



```
Name of person: Mukul,Student ID: 22
```

N

ote: The Person and Student objects both have the same method (i.e toString()), this is called **Method Overriding**. Method Overriding allows a method in a child class to have the same name(polymorphism) and method signature as that of a parent class.
In the above code, the super keyword is used to refer to the immediate parent class's instance variable.

Polymorphism: Polymorphism is one of the core concepts of object-oriented programming languages. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. Polymorphism can be achieved by method overriding and method overloading

Functions in JS AJAX

What is Ajax?

Ajax stands for Asynchronous Javascript And Xml. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the browser's built-in XMLHttpRequest (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

Note: Ajax is not a new technology, in fact, Ajax is not even really a technology at all. Ajax is just a term to describe the process of exchanging data from a web server asynchronously through JavaScript, without refreshing the page.

Tip: Don't get confused by the term X (i.e. XML) in AJAX. It is only there for historical reasons. Other data exchange format such as JSON, HTML, or plain text can be used instead of XML.

Understanding How Ajax Works

To perform Ajax communication JavaScript uses a special object built into the browser—an XMLHttpRequest (XHR) object—to make HTTP requests to the server and receive data in response.

All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the XMLHttpRequest object.

The following illustrations demonstrate how Ajax communication works:

Ajax Illustration

Since Ajax requests are usually asynchronous, execution of the script continues as soon as the Ajax request is sent, i.e. the browser will not halt the script execution until the server response comes back.

In the following section we'll discuss each step involved in this process one by one:

Sending Request and Retrieving the Response

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an XMLHttpRequest object, as shown below:

```
var request = new XMLHttpRequest();
```

Now, the next step in sending the request to the server is to instantiating the newly-created request object using the open() method of the XMLHttpRequest object.

The open() method typically accepts two parameters– the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

```
request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");
```

Tip: The file can be of any kind, like .txt or .xml, or server-side scripting files, like .php or .asp, which can perform some actions on the server (e.g. inserting or reading data from database) before sending the response back to the client.

And finally send the request to the server using the send() method of the XMLHttpRequest object.

request.send(); -Or- request.send(body);

Note: The send() method accepts an optional body parameter which allow us to specify the request's body. This is primarily used for HTTP POST requests, since the HTTP GET request doesn't have a request body, just request headers.

The GET method is generally used to send small amount of data to the server. Whereas, the POST method is used to send large amount of data, such as form data.

In GET method, the data is sent as URL parameters. But, in POST method, the data is sent to the server as a part of the HTTP request body. Data sent through POST method will not visible in the URL.

See the chapter on HTTP GET vs. POST for a detailed comparison of these two methods.

In the following section we'll take a closer look at how Ajax requests actually works.

Performing an Ajax GET Request

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change in database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript.

ExampleTry this code »

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax GET Demo</title>

<script>

function displayFullName() {

    // Creating the XMLHttpRequest object

    var request = new XMLHttpRequest();


    // Instantiating the request object

    request.open("GET", "greet.php?fname=John&lname=Clark");


    // Defining event listener for readystatechange event

    request.onreadystatechange = function() {

        // Check if the request is compete and was successful

        if(this.readyState === 4 && this.status === 200) {
            // Inserting the response from server into an HTML element

            document.getElementById("result").innerHTML =
this.responseText;

        }

    };



    // Sending the request to the server

    request.send();

}

</script>

</head>

<body>

    <div id="result">

        <p>Content of the result DIV box will be replaced by the server
response</p>

    </div>

    <button type="button" onclick="displayFullName()">Display Full
Name</button>

</body>

</html>
```

When the request is asynchronous, the send() method returns immediately after sending the request. Therefore you must check where the response currently stands in its lifecycle before processing it using the readyState property of the XMLHttpRequest object.

The readyState is an integer that specifies the status of an HTTP request. Also, the function assigned to the onreadystatechange event handler called every time the readyState property changes. The possible values of the readyState property are summarized below.

Value	State	Description
0	UNSENT	An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized).
1	OPENED	The open() method has been called (i.e. server connection established).
2	HEADERS_RECEIVED	The send() method has been called (i.e. server has received the request).
3	LOADING	The server is processing the request.
4	DONE	The request has been processed and the response is ready.

Note: Theoretically, the readystatechange event should be triggered every time the readyState property changes. But, most browsers do not fire this event when readyState changes to 0 or 1. However, all browsers fire this event when readyState changes to 4 .

The status property returns the numerical HTTP status code of the XMLHttpRequest's response. Some of the common HTTP status codes are listed below:

200 – OK. The server successfully processed the request.

404 – Not Found. The server can't find the requested page.

503 – Service Unavailable. The server is temporarily unavailable.

Please check out the HTTP status codes reference for a complete list of response codes.

Here's the code from our "greet.php" file that simply creates the full name of a person by joining their first name and last name and outputs a greeting message.

ExampleDownload

```
<?php

if(isset($_GET["fname"]) && isset($_GET["lname"])) {

   $fname = htmlspecialchars($_GET["fname"]);

   $lname = htmlspecialchars($_GET["lname"]);


   // Creating full name by joining first and last name

   $fullname = $fname . " " . $lname;


   // Displaying a welcome message

   echo "Hello, $fullname! Welcome to our website.";

} else {

   echo "Hi there! Welcome to our website.";

}

?>
```

Performing an Ajax POST Request

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

ExampleTry this code »

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax POST Demo</title>

<script>

function postComment() {

    // Creating the XMLHttpRequest object

    var request = new XMLHttpRequest();


    // Instantiating the request object

    request.open("POST", "confirmation.php");


    // Defining event listener for readystatechange event

    request.onreadystatechange = function() {
        // Check if the request is compete and was successful
        if(this.readyState === 4 && this.status === 200) {
            // Inserting the response from server into an HTML element
            document.getElementById("result").innerHTML =
this.responseText;
        }
    };

    // Retrieving the form data
    var myForm = document.getElementById("myForm");
    var formData = new FormData(myForm);

    // Sending the request to the server
    request.send(formData);
}
</script>
</head>
<body>
    <form id="myForm">
        <label>Name:</label>
        <div><input type="text" name="name"></div>
        <br>
        <label>Comment:</label>

        <div><textarea name="comment"></textarea></div>

        <p><button type="button" onclick="postComment()">Post
Comment</button></p>

    </form>

    <div id="result">

        <p>Content of the result DIV box will be replaced by the server
response</p>

    </div>

</body>

</html>
```

If you are not using the FormData object to send form data, for example, if you're sending the form data to the server in the query string format, i.e. request.send(key1=value1&key2=value2) then you need to explicitly set the request header using setRequestHeader() method, like this:

```
request.setRequestHeader("Content-type", "application/x-www-form-
urlencoded");
```

The setRequestHeader() method, must be called after calling open(), but before calling send().

Some common request headers are: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml, text/plain, text/html, and so on.

Note: The FormData object provides an easy way to construct a set of key/value pairs representing form fields and their values which can be sent using XMLHttpRequest.send() method. The transmitted data is in the same format that the form's submit() method would use to send the data if the form's encoding type were set to multipart/form-data.

Here's the code of our "confirmation.php" file that simply outputs the values submitted by the user.

ExampleDownload

```
<?php

if($_SERVER["REQUEST_METHOD"] == "POST") {

    $name = htmlspecialchars(trim($_POST["name"]));

    $comment = htmlspecialchars(trim($_POST["comment"]));



    // Check if form fields values are empty

    if(!empty($name) && !empty($comment)) {

        echo "<p>Hi, <b>$name</b>. Your comment has been received
successfully.<p>";

        echo "<p>Here's the comment that you've entered:
<b>$comment</b></p>";

    } else {

        echo "<p>Please fill all the fields in the form!</p>";

    }

} else {
    echo "<p>Something went wrong. Please try again.</p>";

}

?>
```

For security reasons, browsers do not allow you to make cross-domain Ajax requests. This means you can only make Ajax requests to URLs from the same domain as the original page, for example, if your application is running on the domain "mysite.com", you cannot make Ajax request to "othersite.com" or any other domain. This is commonly known as same origin policy.

JSON data format.

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a lightweight data-interchange format
- JSON is plain text written in JavaScript object notation
- JSON is used to send data between computers
- JSON is language independent *****

*
The JSON syntax is derived from JavaScript object notation, but the JSON format is text only.

Code for reading and generating JSON exists in many programming languages.

The JSON format was originally specified by Douglas Crockford.

Why Use JSON?

The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.

Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JavaScript has a built in function for converting JSON strings into JavaScript objects:

`JSON.parse()`

JavaScript also has a built in function for converting an object into a JSON string:

`JSON.stringify()`

You can receive pure text from a server and use it as a JavaScript object.

You can send a JavaScript object to a server in pure text format.

You can work with data as JavaScript objects, with no complicated parsing and translations.

Storing Data

When storing data, the data has to be a certain format, and regardless of where you choose to store it, *text* is always one of the legal formats.

JSON makes it possible to store JavaScript objects as text.

JSON Example

This example is a JSON string:

```
'{"name":"John", "age":30, "car":null}'
```

It defines an object with 3 properties:

- name
- age
- car

Each property has a value.

If you parse the JSON string with a JavaScript program, you can access the data as an object:

```
let personName = obj.name;
let personAge = obj.age;
```

UNIT - III

REACT JS: Introduction to React React Router and Single Page Applications React Forms, Flow Architecture and Introduction to Redux More Redux and Client-Server Communication

Introduction to ReactJS

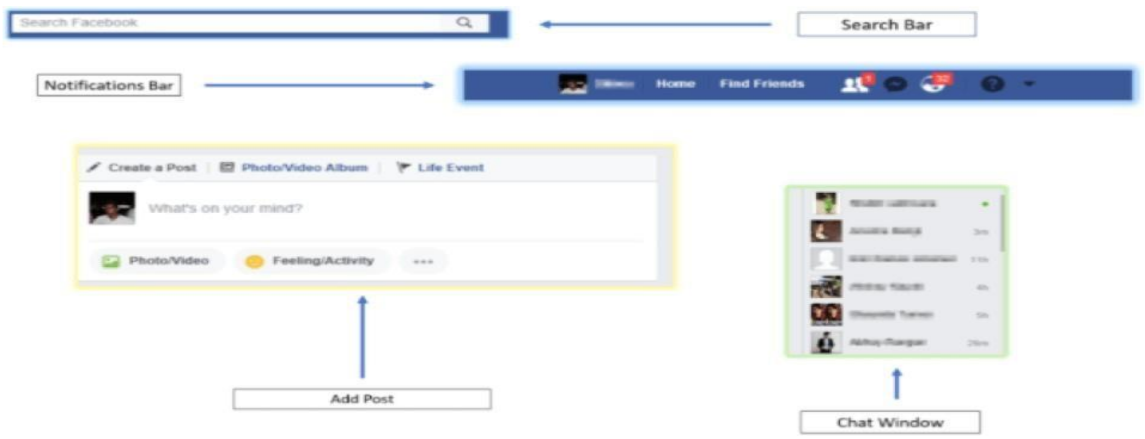
React is a popular JavaScript library used for web development. React.js or ReactJS or React are different ways to represent ReactJS. Today's many large-scale companies (Netflix, Instagram, to name a few) also use React JS. There are many advantages of using this framework over other frameworks, and It's ranking under the top 10 programming languages for the last few years under various language ranking indices.

What is ReactJS?

React.js is a front-end JavaScript framework developed by Facebook. To build composable user interfaces predictably and efficiently using declarative code, we use React. It's an open-source and component-based framework responsible for creating the application's view layer.

- ReactJs follows the Model View Controller (MVC) architecture, and the view layer is accountable for handling mobile and web apps.
- React is famous for building single-page applications and mobile apps.

Let's take an example: Look at the Facebook page, which is entirely built on React, to understand how react does works.



As the figure shows, ReactJS divides the UI into multiple components, making the code easier to debug. This way, each function is assigned to a specific component, and it produces some HTML which is rendered as output by the DOM.

ReactJS History:

Jordan Walke created React, who worked as a software engineer in Facebook has first released an early React prototype called "FaxJS."

In 2011, React was first deployed on Facebook's News Feed, and later in 2012 on Instagram.

As the figure shows, ReactJS divides the UI into multiple components, making the code easier to debug. This way, each function is assigned to a specific component, and it produces some HTML which is rendered as output by the DOM.

ReactJS History:

Jordan Walke created React, who worked as a software engineer in Facebook has first released an early React prototype called "FaxJS."

In 2011, React was first deployed on Facebook's News Feed, and later in 2012 on Instagram.

For interesting tech tutorials and career insights, subscribe to
MindMajix YouTube Channel
SUBSCRIBE

The current version of React.JS is V17.0.1.

Why do people choose to program with React?

There are various reasons why you should choose ReactJS as a primary tool for website UI development. Here, we highlight the most notable ones and explain why these specifics are so important:

- Fast - Feel quick and responsive through the Apps made in React can handle complex updates.
- Modular - Allow you to write many smaller, reusable files instead of writing large, dense files of code. The modularity of React is an attractive solution for JavaScript's visibility issues.
- Scalable - React performs best in the case of large programs that display a lot of data changes.
- Flexible - React approaches differently by breaking them into components while building user interfaces. This is incredibly important in large applications.
- Popular - ReactJS gives better performance than other JavaScript languages due to t's implementation of a virtual DOM.
- Easy to learn - Since it requires minimal understanding of HTML and JavaScript, the learning curve is low.
- Server-side rendering and SEO friendly - ReactJS websites are famous for their server-side rendering feature. It makes apps faster and much better for search engine ranking in comparison to products with client-side rendering. React even produces more opportunities for website SEO and can occupy higher positions on the search result's page.
- Reusable UI components - React improves development and debugging processes.
- Community - The number of tools and extensions available for ReactJS developers is tremendous. Along with impressive out-of-box functionalities, more opportunities emerge once you discover how giant the React galaxy is. React has a vibrant community and is supported by Facebook. Hence, it's a reliable tool for website development.

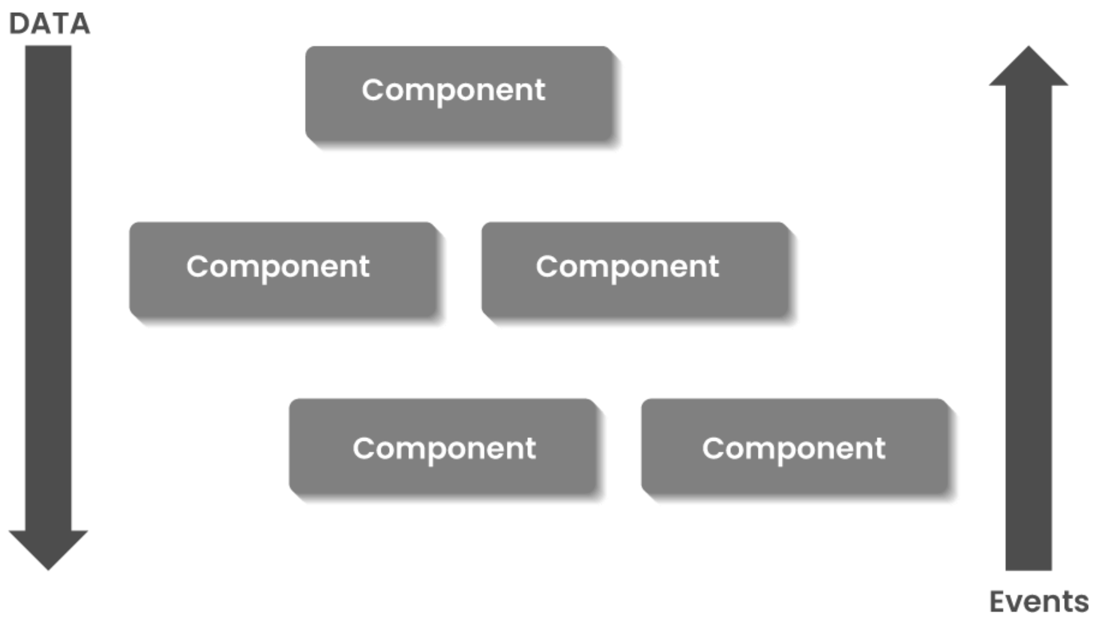
ReactJS Features:

1. JSX - JavaScript Syntax Extension

JSX is a preferable choice for many web developers. It isn't necessary to use JSX in React development, but there is a massive difference between writing react.js documents in JSX and JavaScript. JSX is a syntax extension to JavaScript. By using that, we can write HTML structures in the same file that contains JavaScript code.

2. Unidirectional Data Flow and Flux

React.js is designed so that it will only support data that is flowing downstream, in one direction. If the data has to flow in another direction, you will need additional features.

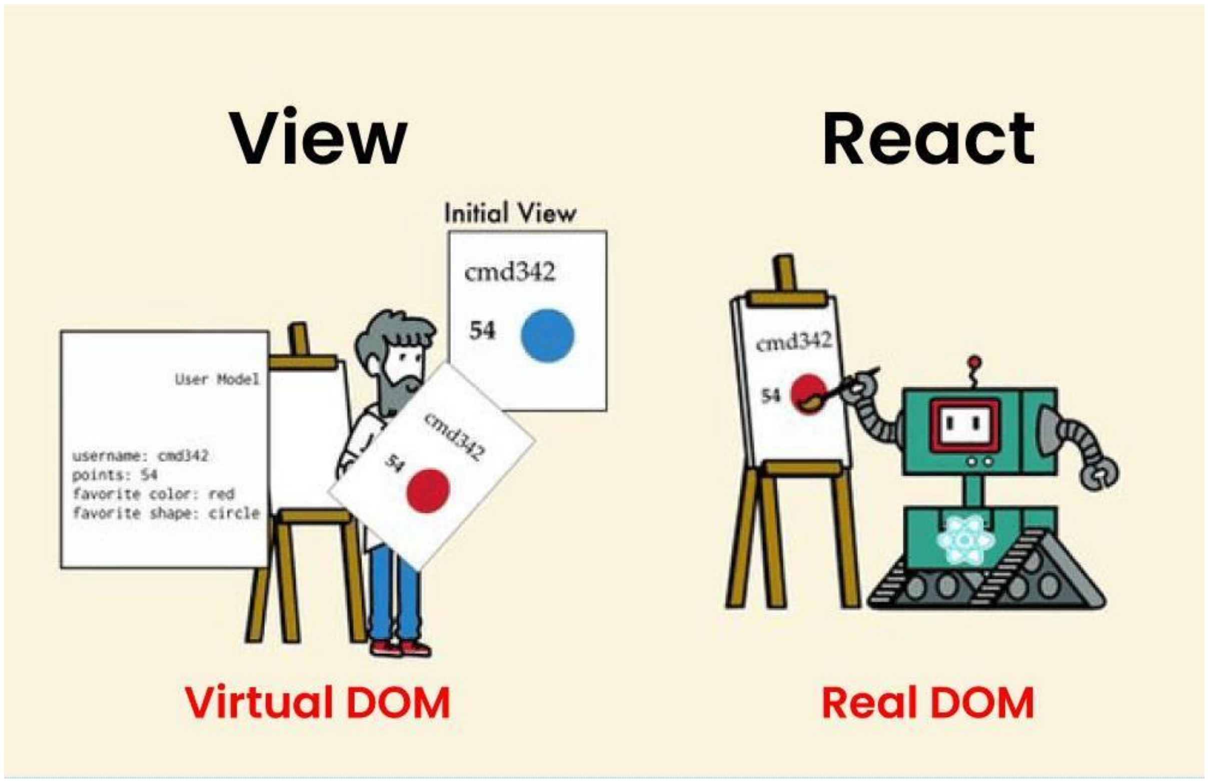


React contains a set of immutable values passed to the component renderer as properties in HTML tags. The components cannot modify any properties directly but support a call back function to do modifications.

3. Virtual Document Object Model (VDOM)

React contains a lightweight representation of real DOM in the memory called Virtual DOM. Manipulating real DOM is much slower compared to VDOM as nothing gets drawn on the screen. When any object's state changes, VDOM modifies only that object in real DOM instead of updating whole objects.

That makes things move fast, particularly compared with other front-end technologies that have to update each object even if only a single object changes in the web application.



4. Extensions

React supports various extensions for application architecture. It supports server-side rendering, Flux, and Redux extensively in web app development. React Native is a popular framework developed from React for creating cross-compatible mobile apps.

5. Debugging

Testing React apps is easy due to large community support. Even Facebook provides a small browser extension that makes React debugging easier and faster.

Next, let's understand some essential concepts of ReactJS.

Building Components of React - Components, State, Props, and Keys.

1. ReactJS Components

Components are the heart and soul of React. Components (like JavaScript functions) let you split the UI into independent, reusable pieces and think about each piece in isolation.

Components are building blocks of any React application. Every component has its structures, APIs, and methods.

In React, there are two types of components, namely stateless functional and stateful class.

- Functional Components - These components have no state of their own and contain only a render method. They are simply Javascript functions that may or may not receive data as parameters.

Stateless functional components may derive data from other components as properties (props).

An example of representing functional component is shown below:

```
function WelcomeMessage(props) {
  return <h1>Welcome to the , {props.name}</h1>;
}
```

- Class Components - These components are more complex than functional components. They can manage their state and to return JSX on the screen have a separate render method. You can pass data from one class to other class components.

An example of representing class component is shown below:

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>This is the main component.</div>
    );
  }
```

2. React State

A state is a place from where the data comes. The state in a component can change over time, and whenever it changes, the component re-renders.

A change in a state can happen as a response to system-generated events or user action, and these changes define the component’s behavior and how it will render.

```
class Greetings extends React.Component {

  state = {

    name: "World"

  };

  updateName() {

    this.setState({ name: "Mindmajix" });

  }

  render() {

      return(

          <div>

              {this.state.name}

          </div>

      )

  }

}
```

The state object is initialized in the constructor, and it can store multiple properties.

For changing the state object value, use this.setState() function.

To perform a merge between the new and the previous state, use the setState() function.

3. React Props

Props stand for properties, and they are read-only components.

Both Props and State are plain JavaScript objects and hold data that influence the output of render. And they are different in one way: State is managed within the component (like variable declaration within a function), whereas props get passed to the component (like function parameters).

Props are immutable, and this is why the component of a container should describe the state that can be changed and updated, while the child components should only send data from the state using properties.

4. React Keys

In React, Keys are useful when working with dynamically created components. Setting the key value will keep your component uniquely identified after the change.

They help React in identifying the items which have changed, are removed, or added.

In summary, State, Props, keys, and components are the few fundamental React concepts that you need to be familiar with before working on it.

React Router

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL intothe browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Installation

React contains three different packages for routing. These are:

1. **react-router:** It provides the core routing components and functions for the React Router applications.
2. **react-router-native:** It is used for mobile applications.
3. **react-router-dom:** It is used for web applications design.

It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application. The below command is used to install react router dom.

1. $ npm install react-router-dom --save

Components in React Router

There are two types of router components:

- **<BrowserRouter>:** It is used for handling the dynamic URL.
- **<HashRouter>:** It is used for handling the static request.

Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

```
import React from 'react'

class About extends React.Component {

  render() {

    return <h1>About</h1>

  }

}

export default About
```

```
import React from 'react'

class About extends React.Component {

  render() {

    return <h1>About</h1>

  }

}

export default About
```

Contact.js

```
import React from 'react'

class Contact extends React.Component {

  render() {

    return <h1>Contact</h1>

  }

}

export default Contact
```

App.js

```
import React from 'react'

class App extends React.Component {

  render() {

    return (

      <div>

        <h1>Home</h1>

      </div>

    )

  }

}

export default App
```

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react- router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

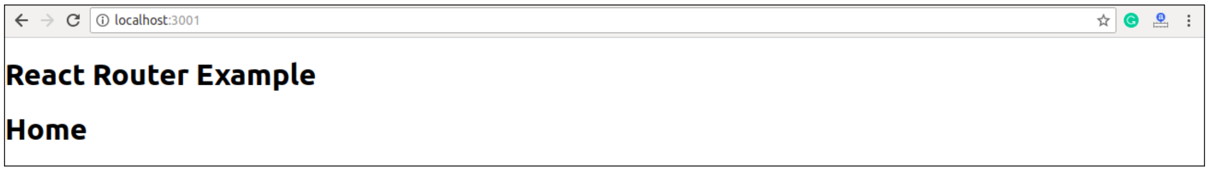
What is Route?

It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Route, Link, BrowserRouter as Router } from 'react-router-dom'

import './index.css';
import App from './App';

import About from './about'

import Contact from './contact'



const routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route path="/" component={App} />

      <Route path="/about" component={About} />

      <Route path="/contact" component={Contact} />

    </div>

  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));
```

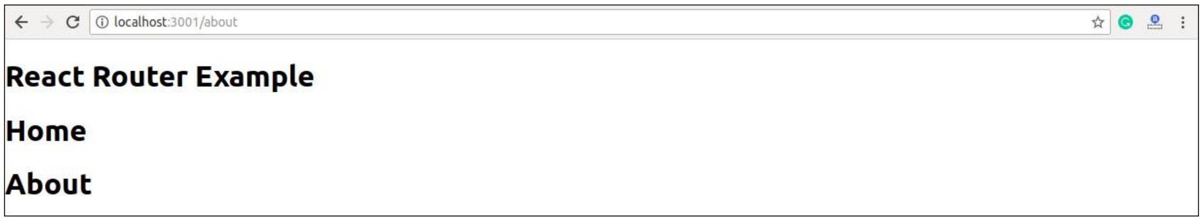
Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



React Router Example

Home

Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



React Router Example

Home

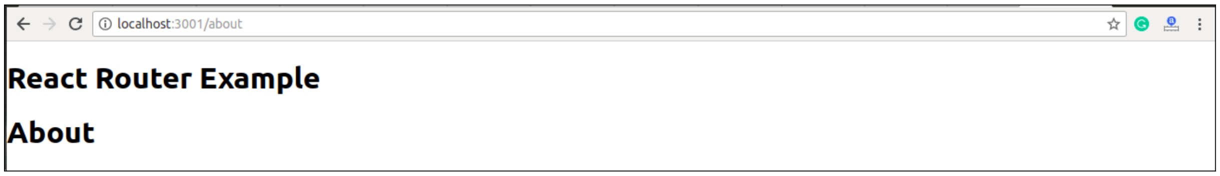
About

Step-4: In the above screen, you can see that **Home** component is still rendered. It is because the home path is '/' and about path is '**/about**', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Route, Link, BrowserRouter as Router } from 'react-router-dom'

import './index.css';

import App from './App';

import About from './about'

import Contact from './contact'



const routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

Output



React Router Example

About

Adding Navigation using Link component

Sometimes, we want to need **multiple** links on a single page. When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page. To do this, we need to import **<Link>** component in the **index.js** file.

What is < Link> component?

This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

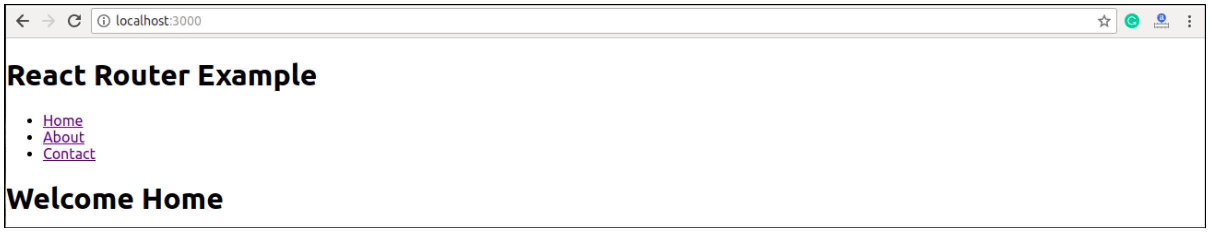
Example

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
```

```
const routing = (
 <Router>
   <div>
     <h1>React Router Example</h1>
     <ul>
       <li>
         <Link to="/">Home</Link>
       </li>
       <li>
         <Link to="/about">About</Link>
       </li>
       <li>
         <Link to="/contact">Contact</Link>
       </li>
     </ul>
     <Route exact path="/" component={App} />
     <Route path="/about" component={About} />
     <Route path="/contact" component={Contact} />
   </div>
 </Router>
)

ReactDOM.render(routing, document.getElementById('root'));
```

Output

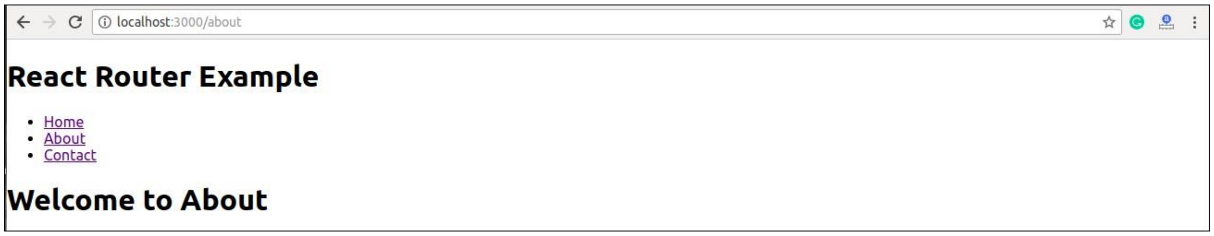

localhost:3000

React Router Example

- Home
- About
- Contact

Welcome Home

After adding Link, you can see that the routes are rendered on the screen. Now, if you click on the **About**, you will see URL is changing and About component is rendered.


localhost:3000/about

React Router Example

- Home
- About
- Contact

Welcome to About

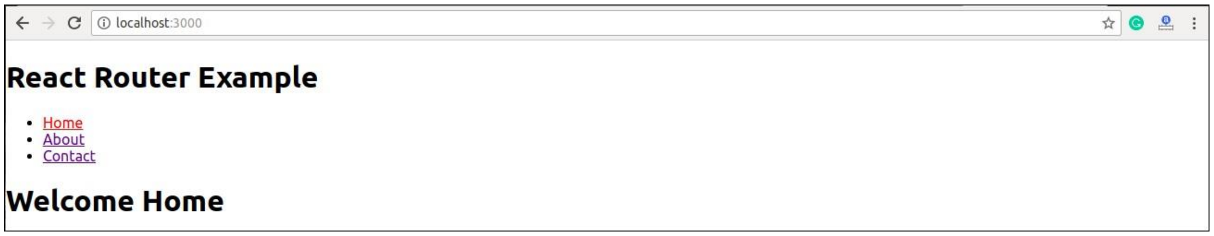
Now, we need to add some **styles** to the Link. So that when we click on any particular link, it can be easily **identified** which Link is **active**. To do this react router provides a new trick **NavLink** instead of **Link**. Now, in the index.js file, replace Link from Navlink and add properties **activeStyle**. The activeStyle properties mean when we click on the Link, it should have a specific style so that we can differentiate which one is currently active.

```
import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter as Router, Route, Link, NavLink } from 'react-router-dom'

import './index.css';

import App from './App';

import About from './about'

import Contact from './contact'
```

```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
             {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
             {color:'green'}
          }>About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" exact activeStyle={
             {color:'magenta'}
          }>Contact</NavLink>
        </li>
      </ul>
      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

      <Route path="/contact" component={Contact} />

    </div>

  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));
```

Output

When we execute the above program, we will get the following screen in which we can see that **Home** link is of color **Red** and is the only currently **active** link.

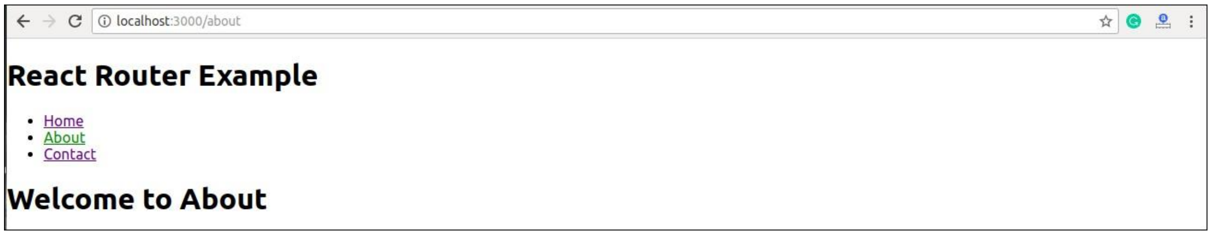

localhost:3000

React Router Example

- Home
- About
- Contact

Welcome Home

Now, when we click on **About** link, its color shown **green** that is the currently **active** link.


localhost:3000/about

React Router Example

- Home
- About
- Contact

Welcome to About

<Link> vs <NavLink>

The Link component allows navigating the different routes on the websites, whereas NavLink component is used to add styles to the active routes.

Benefits Of React Router

The benefits of React Router is given below:

- In this, it is not necessary to set the browser history manually.
- Link uses to navigate the internal links in the application. It is similar to the anchor tag.
- It uses Switch feature for rendering.
- The Router needs only a Single Child element.
- In this, every component is specified in .

React Forms

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

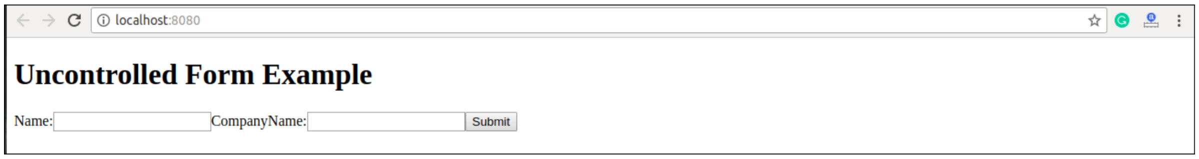
Example

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
      super(props);
      this.updateSubmit = this.updateSubmit.bind(this);
  this.input = React.createRef();
  }
  updateSubmit(event) {
      alert('You have entered the UserName and CompanyName successfully.');
      event.preventDefault();
}
  render() {
    return (
      <form onSubmit={this.updateSubmit}>
        <h1>Uncontrolled Form Example</h1>
        <label>Name:
            <input type="text" ref={this.input} />
        </label>
        <label>
            CompanyName:
            <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
}
27. }
28. export default App;
```

Output

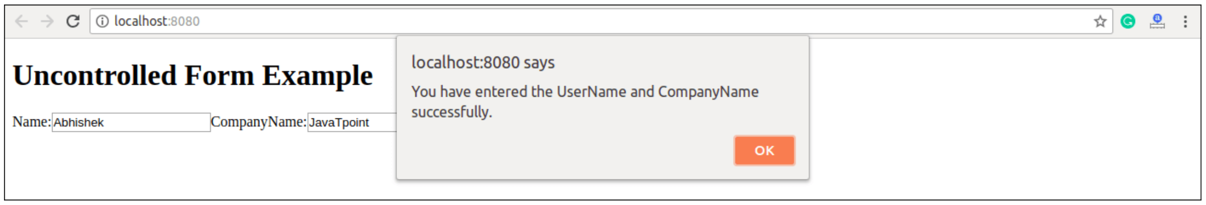
When you execute the above code, you will see the following screen.


localhost:8080

Uncontrolled Form Example

Name: CompanyName: Submit

After filling the data in the field, you get the message that can be seen in the below screen.


localhost:8080

Uncontrolled Form Example

Name: Abhishek CompanyName: JavaTpoint

localhost:8080 says

You have entered the UserName and CompanyName successfully.

OK

Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an onChange event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

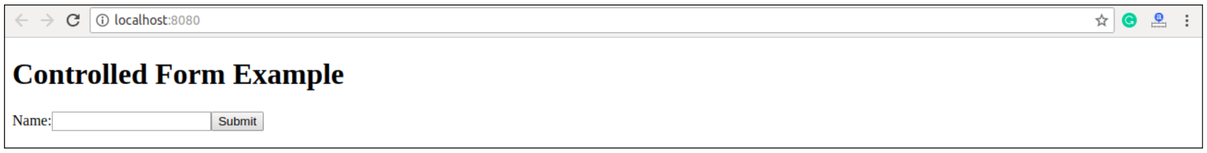
Example

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
      super(props);
      this.state = {value: ''};
      this.handleChange = this.handleChange.bind(this);
  this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
      this.setState({value: event.target.value});
}
  handleSubmit(event) {
      alert('You have submitted the input successfully: ' + this.state.value);
      event.preventDefault();
}
  render() {
      return (
          <form onSubmit={this.handleSubmit}>
            <h1>Controlled Form Example</h1>
            <label>
                Name:
<input type="text" value={this.state.value}
onChange={this.handleChange} />
            </label>
            <input type="submit" value="Submit" />
         </form>
      );
}
}
```

29. export **default** App;

Output

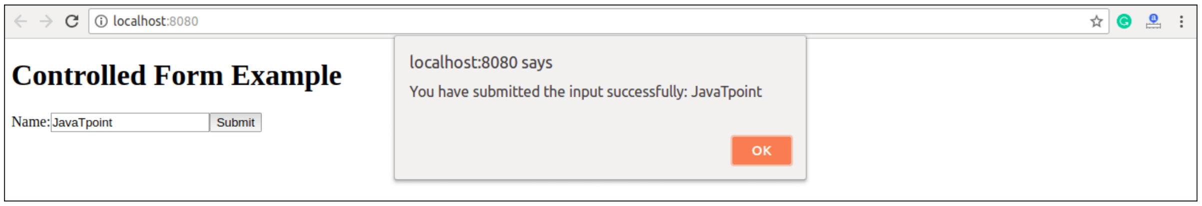
When you execute the above code, you will see the following screen.



localhost:8080

Controlled Form Example

After filling the data in the field, you get the message that can be seen in the below screen.



localhost:8080

Controlled Form Example

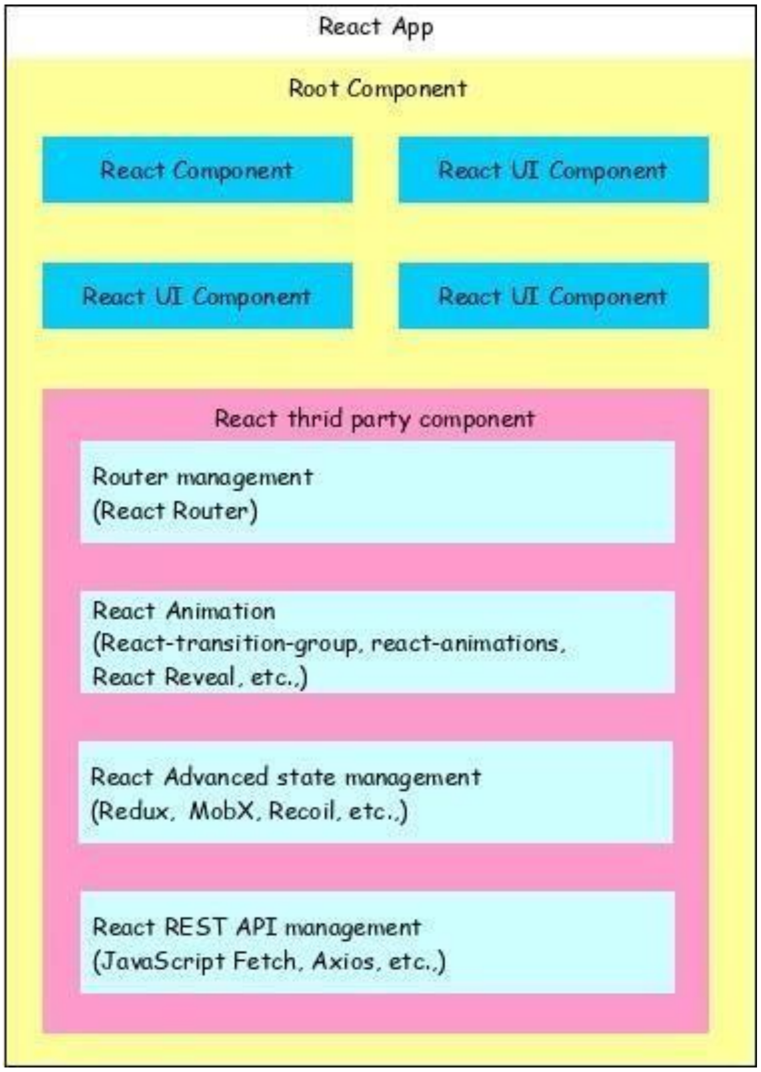
Name: JavaTpoint

localhost:8080 says

You have submitted the input successfully: JavaTpoint

Architecture of the React Application

React library is just UI library and it does not enforce any particular pattern to write a complex application. Developers are free to choose the design pattern of their choice. React community advocates certain design pattern. One of the patterns is Flux pattern. React library also provides lot of concepts like Higher Order component, Context, Render props, Refs etc., to write better code. React Hooks is evolving concept to do state management in big projects. Let us try to understand the high level architecture of a React application.


React App
Root Component
React Component
React UI Component
React UI Component
React UI Component
React thrid party component
Router management
(React Router)
React Animation
(React-transition-group, react-animations, React Reveal, etc.,)
React Advanced state management
(Redux, MobX, Recoil, etc.,)
React REST API management
(JavaScript Fetch, Axios, etc.,)

- React app starts with a single root component.
- Root component is build using one or more component.
- Each component can be nested with other component to any level.
- Composition is one of the core concepts of React library. So, each component is build by composing smaller components instead of inheriting one component from another component.
- Most of the components are user interface components.
- React app can include third party component for specific purpose such as routing, animation, state management, etc.

Unit-3

Angular

1. Getting Started with Angular

a.Defination:-

Angular is an open-source web application framework maintained by Google and a community of developers. It is designed to build dynamic and interactive single-page applications (SPAs) efficiently. With Angular, developers can create robust, scalable, and maintainable web applications.

(Or)

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables usersto create large applications in a maintainable manner.

b. History

Angular, initially released in 2010 by Google, has undergone significant transformations over the years. The first version, AngularJS, introduced concepts like two-way data binding and directives. However, as web development evolved, AngularJS faced limitations in terms of performance and flexibility.

In 2016, Angular 2 was released, which was a complete rewrite of AngularJS, focusing on modularity and performance. Since then, Angular has continued to evolve, with regular updates and improvements to meet the demands of modern web development.

c. Why Angular?

JavaScript is the most commonly used client-side scripting language. It is written into HTML documents to enable interactions with web pages in many unique ways. As a relatively easy-to- learn language with pervasive support, it is well-suited to develop modern applications. But is JavaScript ideal for developing single-page applications that require modularity, testability, and developer productivity? Perhaps not.
These days, we have a variety of frameworks and libraries designed to provide alternative solutions.With respect to front-end web development, Angular addresses many, if not all, of the issues developers face when using JavaScript on its own.

d. Here are some of the features of Angular

1. Custom Components

Angular enables users to build their components that can pack functionality along with renderinglogic into reusable pieces.

2. Data Binding

Angular enables users to effortlessly move data from JavaScript code to the view, and react touser events without having to write any code manually.

3. Dependency Injection

Angular enables users to write modular services and inject them wherever they are needed. Thisimproves the testability and reusability of the same services.

4. Testing

Tests are first-class tools, and Angular has been built from the ground up with testability in mind.You will have the ability to test every part of your application—which is highly recommended.

5. Comprehensive

Angular is a full-fledged JavaScript framework and provides out-of-the-box solutions for servercommunication, routing within your application, and more.

6. Browser Compatibility

Angular works cross-platform and compatible with multiple browsers. An Angular application can typically run on all browsers (Eg: Chrome, Firefox) and operating systems, such as Windows,macOS, and Linux.

7. **Two-Way Data Binding:** Angular provides seamless synchronization between the model andthe view, allowing for easy management of user inputs.
8. **Directives:** Angular offers a rich set of built-in directives for manipulating the DOM, such as *ngIf*, *ngFor*, and *ngSwitch*.
9. **Routing:** Angular’s powerful routing module enables to build SPAs with multiple views andnavigation between them.

10. **HTTP Client:** Angular includes an HTTP client module for making server requests,simplifying data fetching and manipulation.

e. Advantages of Angular

- **Productivity:** Angular’s extensive tooling and ecosystem streamline development tasks,enabling faster project completion.
- **Maintainability:** Angular’s modular architecture and clear separation of concerns promotecode organization and maintainability.
- **Scalability:** Angular is well-suited for building large-scale applications, thanks to itscomponent-based architecture and robust performance.
- **Community Support:** Being backed by Google and a vast community of developers, Angularenjoys strong community support and continuous improvement.

f. Disadvantages of Angular

- **Learning Curve:** Angular has a steep learning curve, especially for beginners, due to itscomplex concepts and extensive documentation.
- **Performance Overhead:** Angular’s powerful features come with a performance cost, andpoorly optimized applications may suffer from performance issues.
- **Size:** Angular applications tend to have larger file sizes compared to other frameworks, whichmay impact load times, especially on mobile devices.
- **Migration:** Upgrading between major Angular versions can be challenging and time-consuming, requiring significant changes to existing codebases.

g. Angular Prerequisites

There are three main prerequisites.

NodeJS

Angular uses Node.js for a large part of its build environment. As a result, to get started with Angular, you willneed to have Node.js installed on your system. You can head to the NodeJS official website to download the software. Install the latest version and confirm them on you command prompt by running the following commands:

Node --

versionnpm --

v

Angular CLI

The Angular team has created a command-line interface (CLI) tool to make it easier to bootstrap and developyour Angular applications. As it significantly helps to make the process of development easier, we highly recommend using it for your initial angular projects at the least.

To install the CLI, in the command prompt, type the following

commandsInstallation:

npm install -g @angular/cli

Confirmation -

ng--version

Text Editor

You need a text editor to write and run your code. The most popularly used integrated development environment (IDE) is Visual Studio Code (VS Code). It is a powerful source code editor that is available onWindows, macOS, and Linux platforms.

Now, Let's create our first Angular HelloWorld Application.

2. Angular App From Scratch Creating an Angular Application

Step 1: Install Angular CLI: Angular CLI (Command Line Interface) is a powerful tool for scaffolding and managing Angular applications. Install it globally using npm:

```
npm install -g @angular/cli
```

Step 2: Create a New Angular Project: Use Angular CLI to create a new Angular project. Navigate to the desired directory and run:

```
ng new my-angular-app //creating standalone application
```

(or)

Ng new my-angular-app -standalone false //creating non-standalone application project structure is different adding two more files app.module.ts and app-routing.module.ts

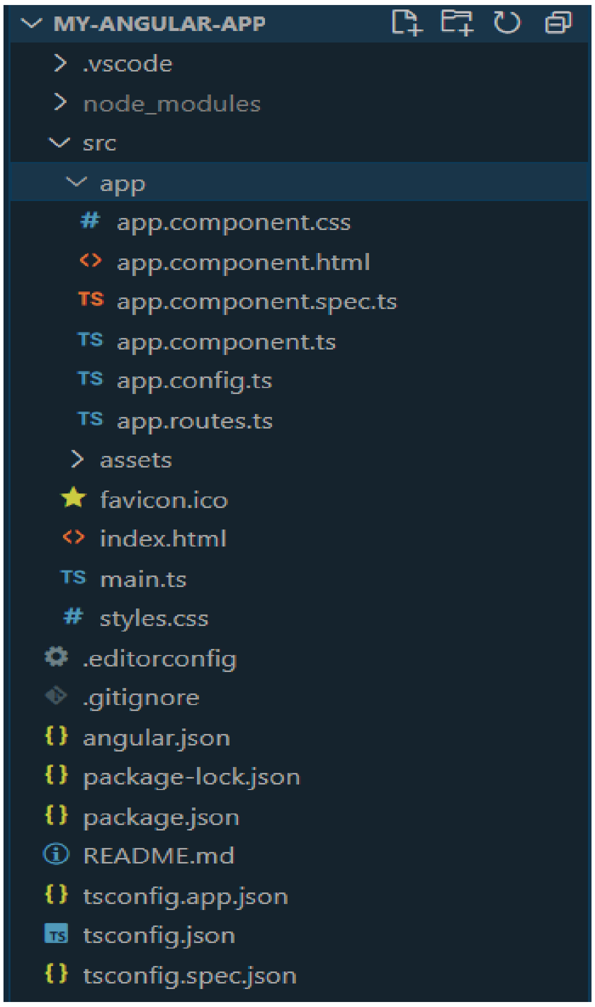
Step 3: Navigate to the Project Directory: Move into the newly created project directory:

```
cd my-angular-app
```

Step 4: Serve the Application: Launch the development server to see your app in action:

```
ng serve
```

Folder Structure:



```
MY-ANGULAR-APP
  > .vscode
  > node_modules
  v src
    v app
        app.component.css
        app.component.html
        app.component.spec.ts
        app.component.ts
        app.config.ts
        app.routes.ts
    > assets
      favicon.ico
      index.html
      main.ts
      styles.css
    .editorconfig
    .gitignore
    angular.json
    package-lock.json
    package.json
    README.md
    tsconfig.app.json
    tsconfig.json
    tsconfig.spec.json
```

What is angular

Dependencies:

```
"dependencies": {
   "@angular/animations":
   "^17.3.0", "@angular/common":
   "^17.3.0", "@angular/compiler":
   "^17.3.0", "@angular/core":
   "^17.3.0", "@angular/forms":
   "^17.3.0",
   "@angular/platform-browser": "^17.3.0",
   "@angular/platform-browser-dynamic":
   "^17.3.0", "@angular/router": "^17.3.0",
   "rxjs": "~7.8.0",
   "tslib": "^2.3.0",
```

Example:

```
<!-- app.component.html -->

<h1>Hello Angular</h1>
//app.component.ts

import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
   selector: 'app-root',
   standalone: true,
   imports: [RouterOutlet],
   templateUrl:
   './app.component.html', styleUrl:
   './app.component.css'
})
export class AppComponent {
   title = 'my-angular-app';
```

Root HTML - index.html(default code)

```
<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>HelloWorld</title>

  <base href="/">
```

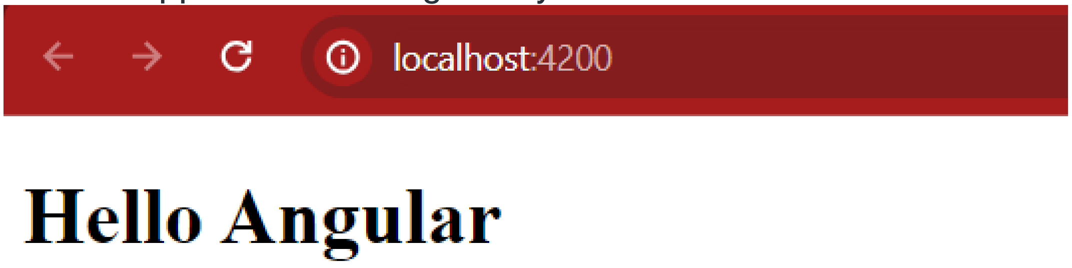
```
<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">
```

```
</head>

<body>

  <app-root></app-root>

</body>

</html>
```

The only main thing in this file is the **<app-root>** element. This is the marker for loading the application. All the application code, styles, and inline templates are dynamically injected into the index.html file at run time bythe **ng serve** command.

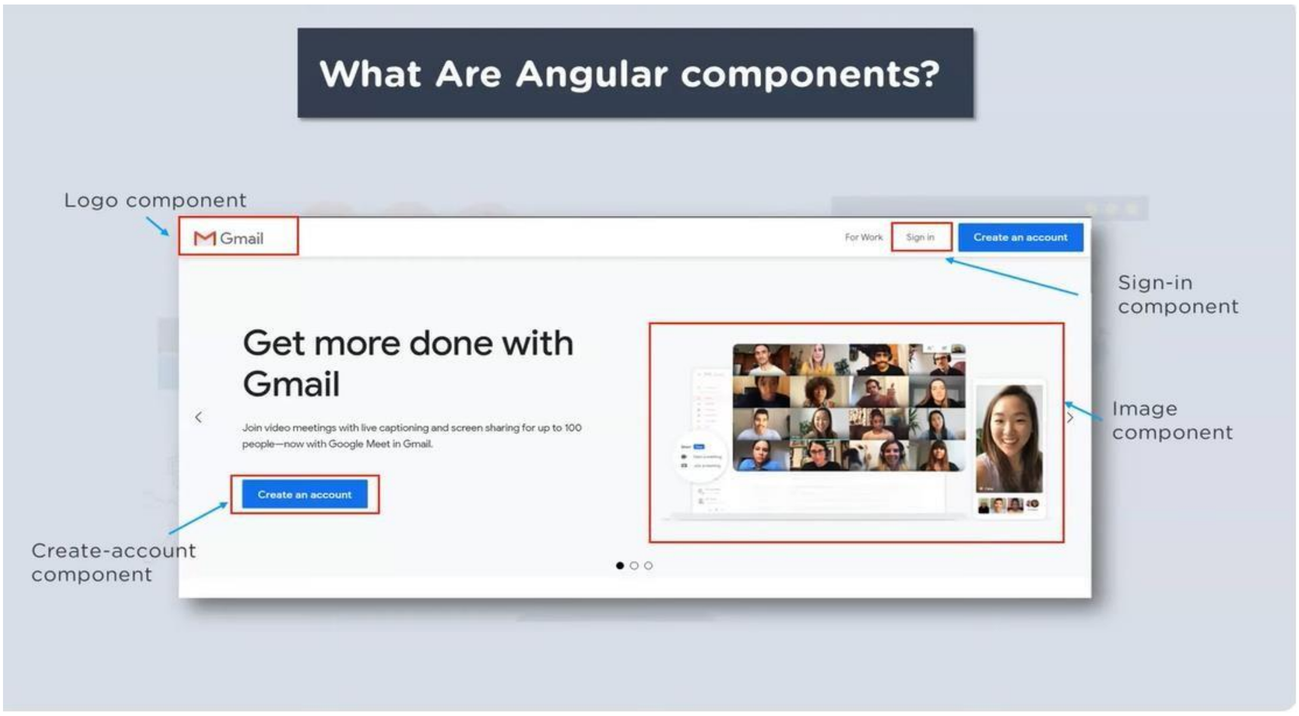
Output:

Upon running **`ng serve`**, the Angular CLI will compile the application and launch a development server. Open a web browser and navigate to **`http://localhost:4200`** to view theapplication running locally.



Hello Angular

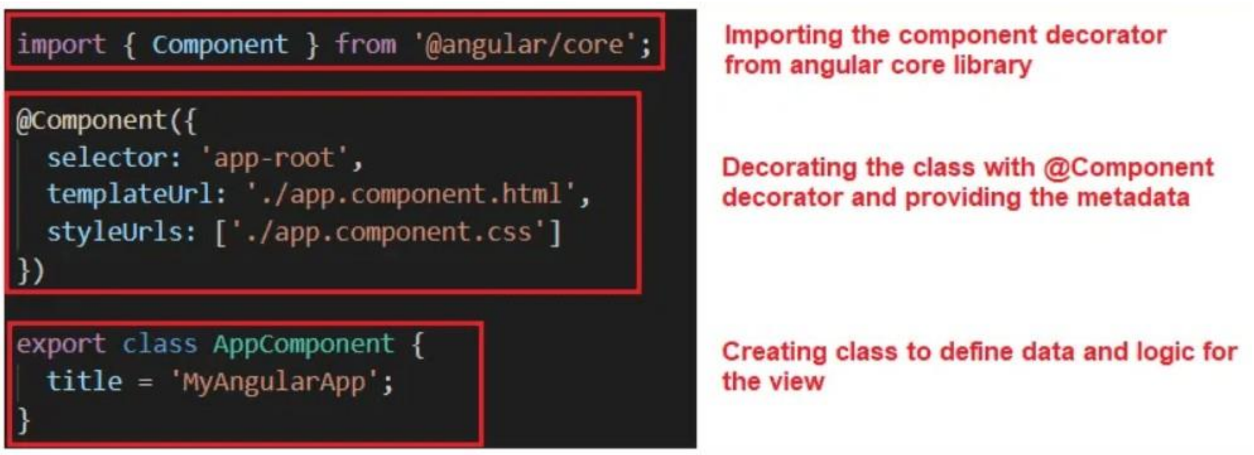
2. Component:-


What Are Angular components?
Logo component
Sign-in component
Image component
Create-account component
Gmail
For Work
Sign in
Create an account
Get more done with Gmail
Join video meetings with live captioning and screen sharing for up to 100 people—now with Google Meet in Gmail.

The above image showing Gmail is a Single Page Application each part consider like a component like logo component, sign-in component etc.,

Defination:-

The component is the basic building block of Angular. It has a selector, template, style, and otherproperties, and it specifies the metadata required to process the component.



```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'MyAngularApp';
}
```

Importing the component decorator from angular core library

Decorating the class with @Component decorator and providing the metadata

Creating class to define data and logic for the view

parts of an Angular Component

An Angular component has several parts, such as:

Selector

It is the CSS selector that identifies this component in a template. This corresponds to the HTML tag that is included in the parent component. You can create your own HTML tag. However, the same has to be includedin the parent component.

Template

It is an inline-defined template for the view. The template can be used to define some markup. The markupcould typically include some headings or paragraphs that are displayed on the UI.

TemplateUrl

It is the URL for the external file containing the template for the view.

Styles

These are inline-defined styles to be applied to the component's view

styleUrls

List of URLs to stylesheets to be applied to the component's view.

Before Creating Angular Component create Angular Project using this commandng new Projectname or na new projectname -standalone false

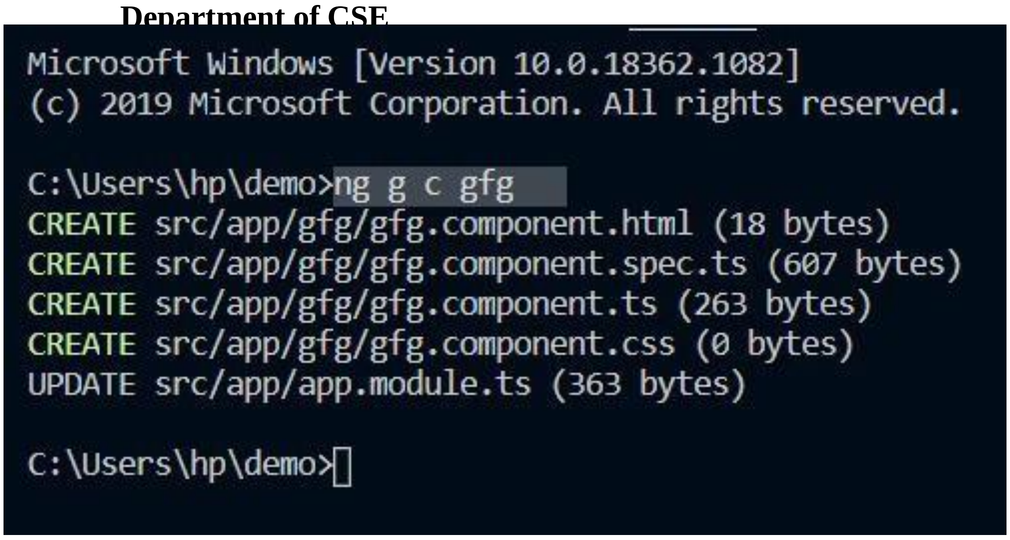
Creating a Component in Angular 8:

To create a component in any angular application, follow the below steps:

- Get to the angular app via your terminal.(ng new project_name)
- Create a component using the following command:

```
ng g c
<component_name>OR
ng generate component <component_name>
```

- Following files will be created after generating the component:
  Note:-write below picture four files in exam important to explaining component



```
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\hp\demo>ng g c gfg
CREATE src/app/gfg/gfg.component.html (18 bytes)
CREATE src/app/gfg/gfg.component.spec.ts (607 bytes)
CREATE src/app/gfg/gfg.component.ts (263 bytes)
CREATE src/app/gfg/gfg.component.css (0 bytes)
UPDATE src/app/app.module.ts (363 bytes)

C:\Users\hp\demo>
```

Using a component in Angular 8:

- Go to the component.html file and write the necessary HTML code.

gfg.component.html:

```
<h1>GeeksforGeeks</h1>
```

- Go to the component.css file and write the necessary CSS code.

gfg.component.css:

```
h1{

  color: green;

  font-size:

  30px;

}
```

- Write the corresponding code in component.ts file.

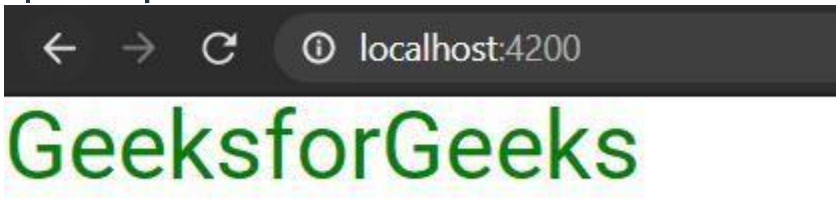
gfg.component.ts:

```
import { Component, OnInit } from



@Component({

 selector: 'app-
```

```
  templateUrl:

  './gfg.component.html',styleUrls:

  ['./gfg.component.css']

})

export class

    GfgComponent{a

    ="GeeksforGeeks";
```

- Run the Angular app using **ng serve -**

openOutput:



GeeksforGeeks

4. Properties, Events & Binding with ngModel

Data Binding

Data binding is the core concept of Angular 8 and used to define the communication between a component and the DOM. It is a technique to link your data to your view layer. In simple words, you can say that data binding is a communication between your typescript code of your component and your template which user sees. It makeseasy to define interactive applications without worrying about pushing and pulling data.

Data binding can be either one-way data binding or two-way data binding.

One-way databinding

One way databinding is a simple one way communication where HTML template is changed when we make changes in TypeScript code.

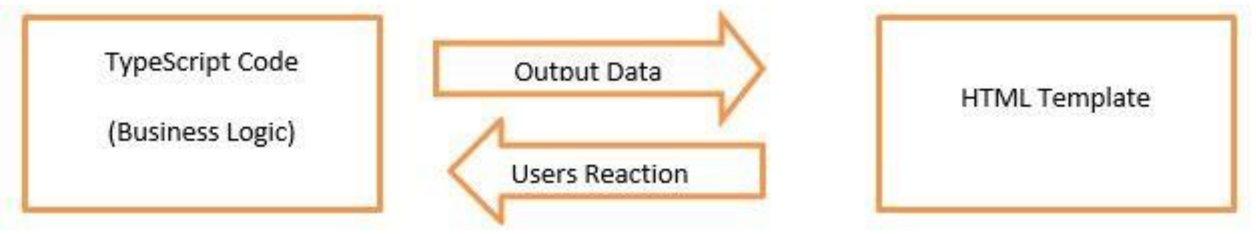
Or

In one-way databinding, the value of the Model is used in the View (HTML page) but you can't update Model from the View. Angular Interpolation / String Interpolation, Property Binding, and Event Binding are the exampleof one-way databinding.

Two-way databinding

In two-way databinding, automatic synchronization of data happens between the Model and the View. Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

This happens immediately and automatically, ensures that the HTML template and the TypeScript code are updated at all times.



Angular provides four types of data binding and they are different on the way of data flowing.

- String Interpolation
- Property Binding

- Event Binding
- Class binding
- Style binding
- Two-way binding

One way Data Binding:-

1. String Interpolation
2. Property Binding
3. Event Binding
4. Class binding
5. Style binding

1.String Interpolation

String Interpolation is a **one-way databinding** technique which is used to output the data from a TypeScript code to HTML template (view). It uses the template expression in **double curly braces** to display the data from the component to the view. String interpolation adds the value of a property from the component.

For example:

{{ data }}

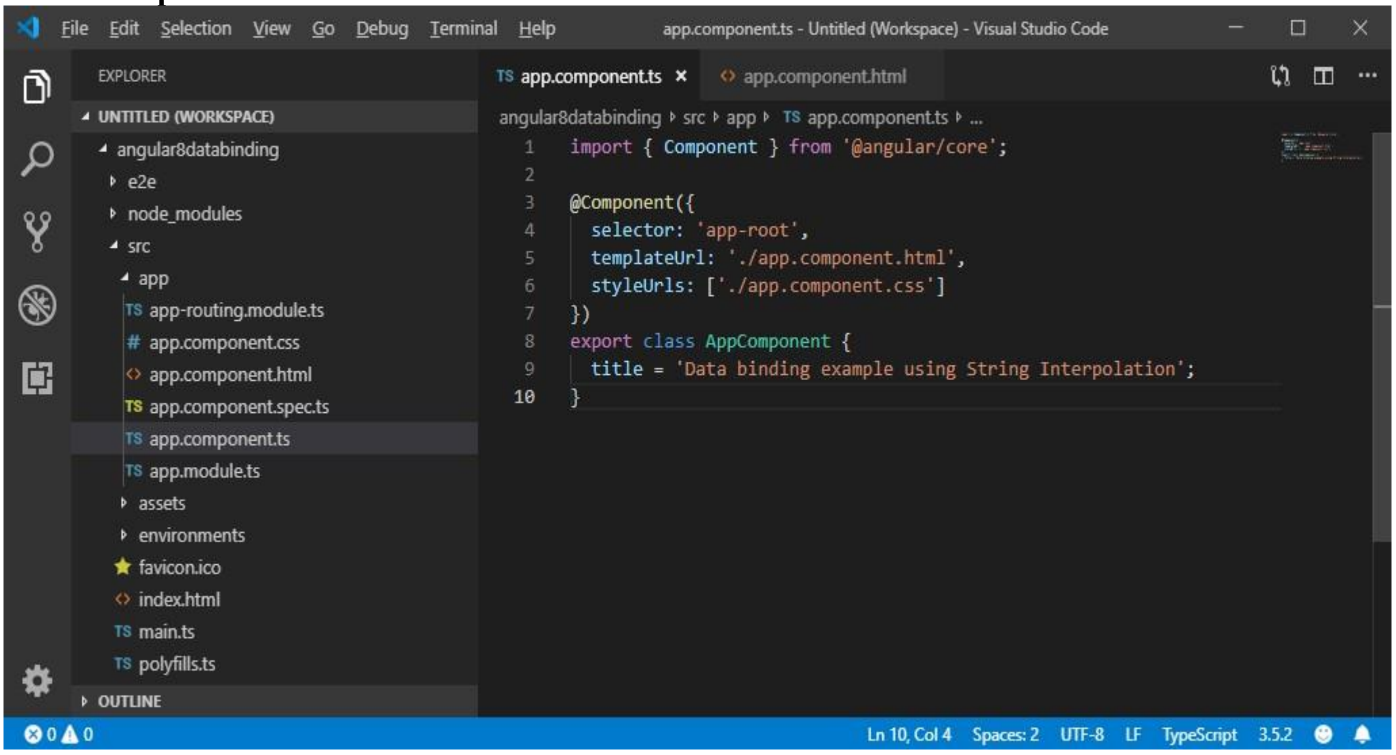
We have already created an Angular project using

Angular CLI.Here, we are using the same project for

this example.

Open **app.component.ts** file and use the following code within the file:

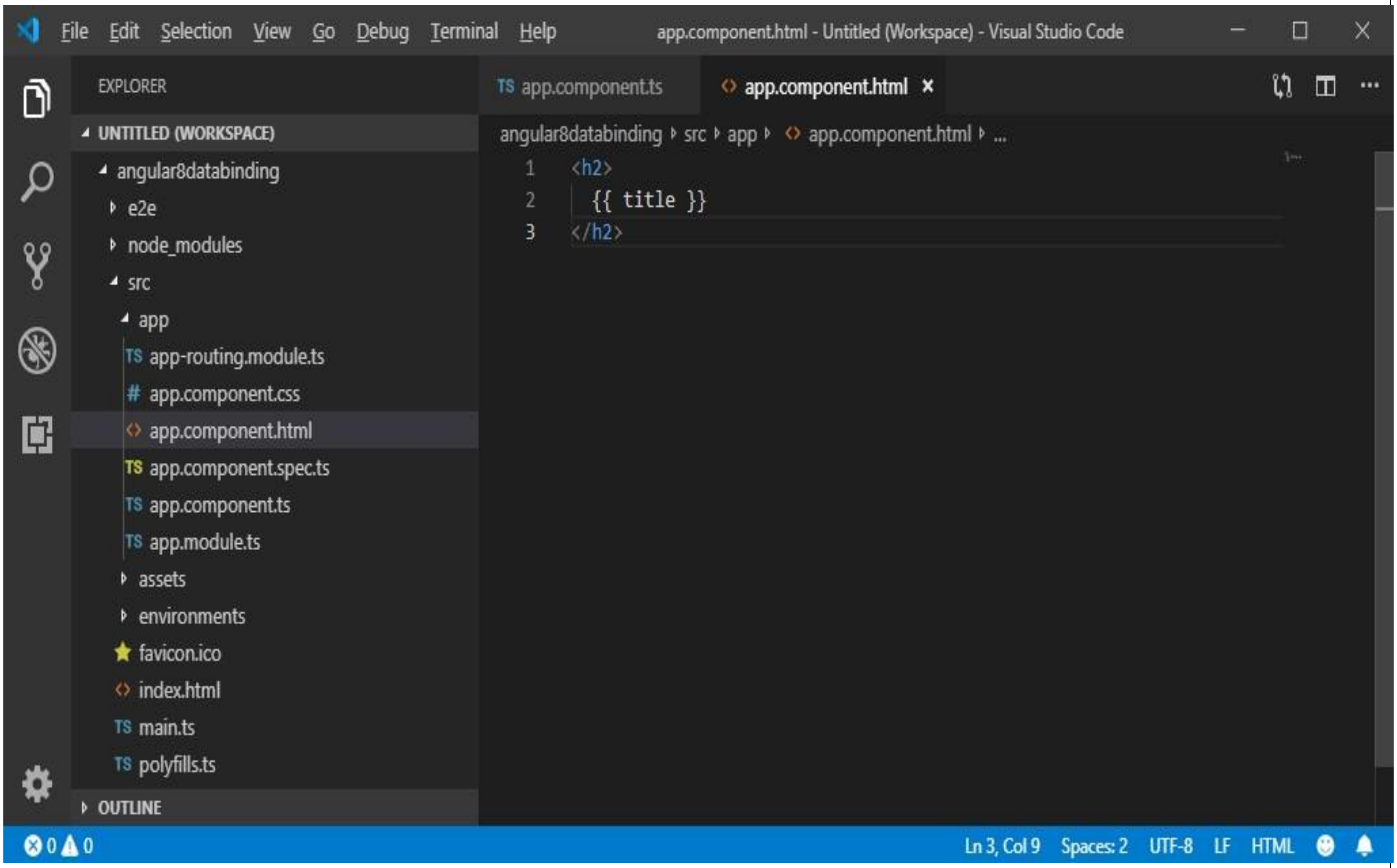
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls:
['./app.component.css']
})
export class AppComponent {
  title = 'Data binding example using String
Interpolation';
}
```



```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Data binding example using String Interpolation';
}
```

Now, open **app.component.html** and use the following code to see string interpolation.

```
1. <h2>
2.   {{ title }}
3. </h2>
```

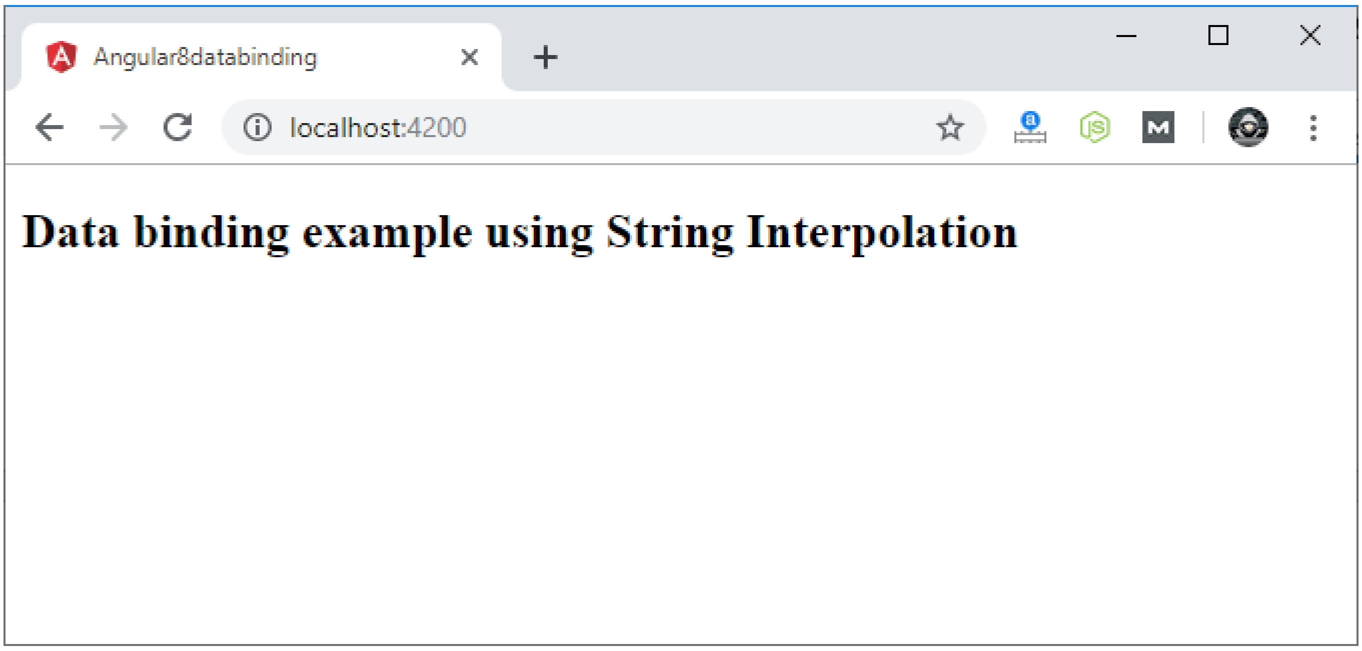


```
<h2>
  {{ title }}
</h2>
```

Now, open Node.js command prompt and run the **ng serve** command to see the result.

Output:

ADVERTISEMENT



Data binding example using String Interpolation

String Interpolation can be used to resolve some other expressions too. Let's see an example.

Example:

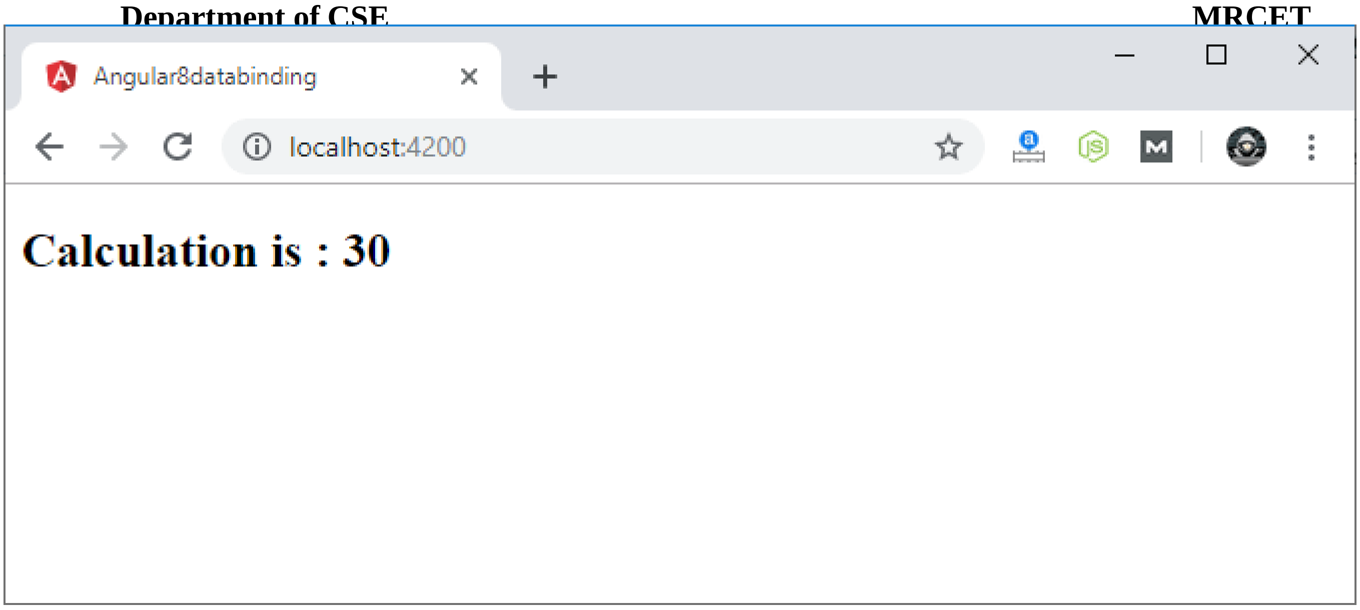
Update the **app.component.ts** file with the following code:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls:
['./app.component.css']6. })
export class AppComponent {
  title = 'Data binding example using String Interpolation';
    numberA: number = 10;
    numberB: number = 20;
 }
```

app.component.html:

```
<h2>Calculation is : {{ numberA + numberB }}</h2>
```

Output:



Calculation is : 30

2. Property Binding in Angular 8

Property Binding is also a **one-way data binding** technique. In property binding, we bind a property of a DOMelement to a field which is a defined property in our component TypeScript code. Actually Angular internally converts string interpolation into property binding.

For example:

```
<img [src]="imgUrl" />
```

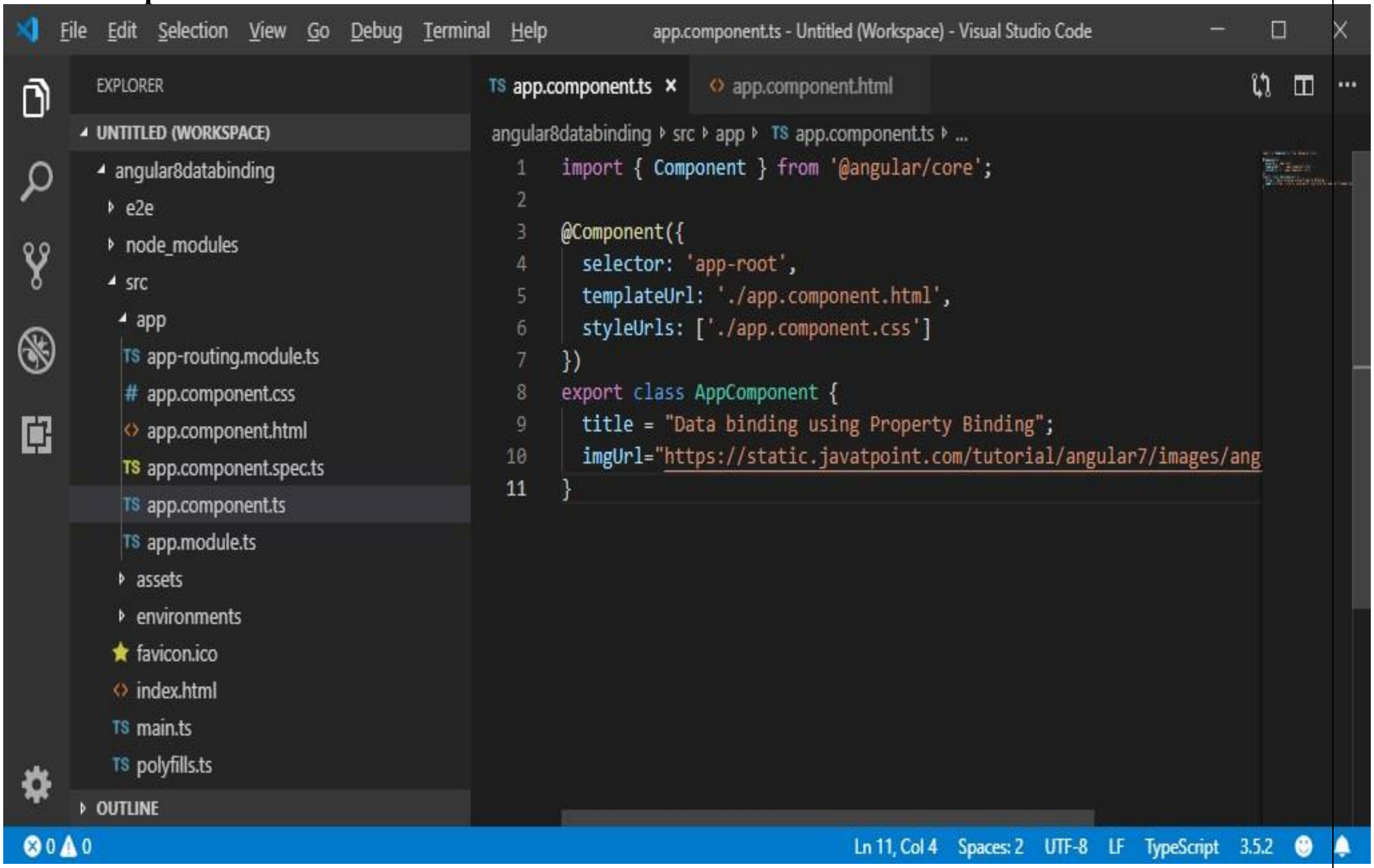
Property binding is preferred over string interpolation because it has shorter and cleaner code String interpolationshould be used when you want to simply display some dynamic data from a component on the view between headings like h1, h2, p etc.

*Note: **String Interpolation and Property binding both are one-way binding. Means, if field value in the componentchanges, Angular will automatically update the DOM. But any changes in the DOM will not be reflected back in the component.***

Property Binding Example

Open **app.componnt.ts** file and add the following code:

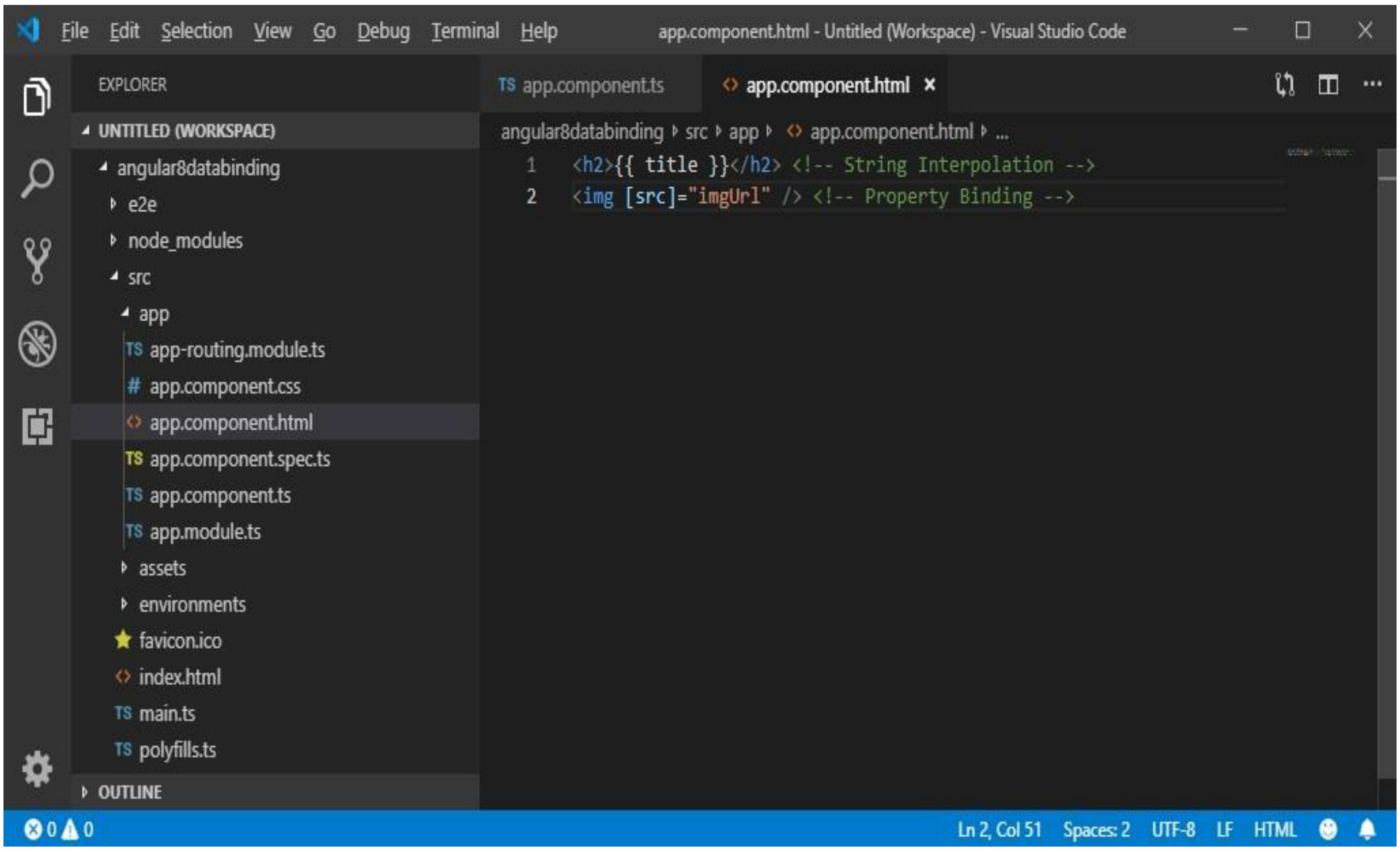
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls:
['./app.component.css']
})
export class AppComponent {
  title = "Data binding using Property Binding";
  imgUrl="https://static.javatpoint.com/tutorial/angular7/images/angular-7-logo.png";
10. }
```



```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = "Data binding using Property Binding";
  imgUrl="https://static.javatpoint.com/tutorial/angular7/images/ang
}
```

Now, open **app.component.html** and use the following code for property binding:

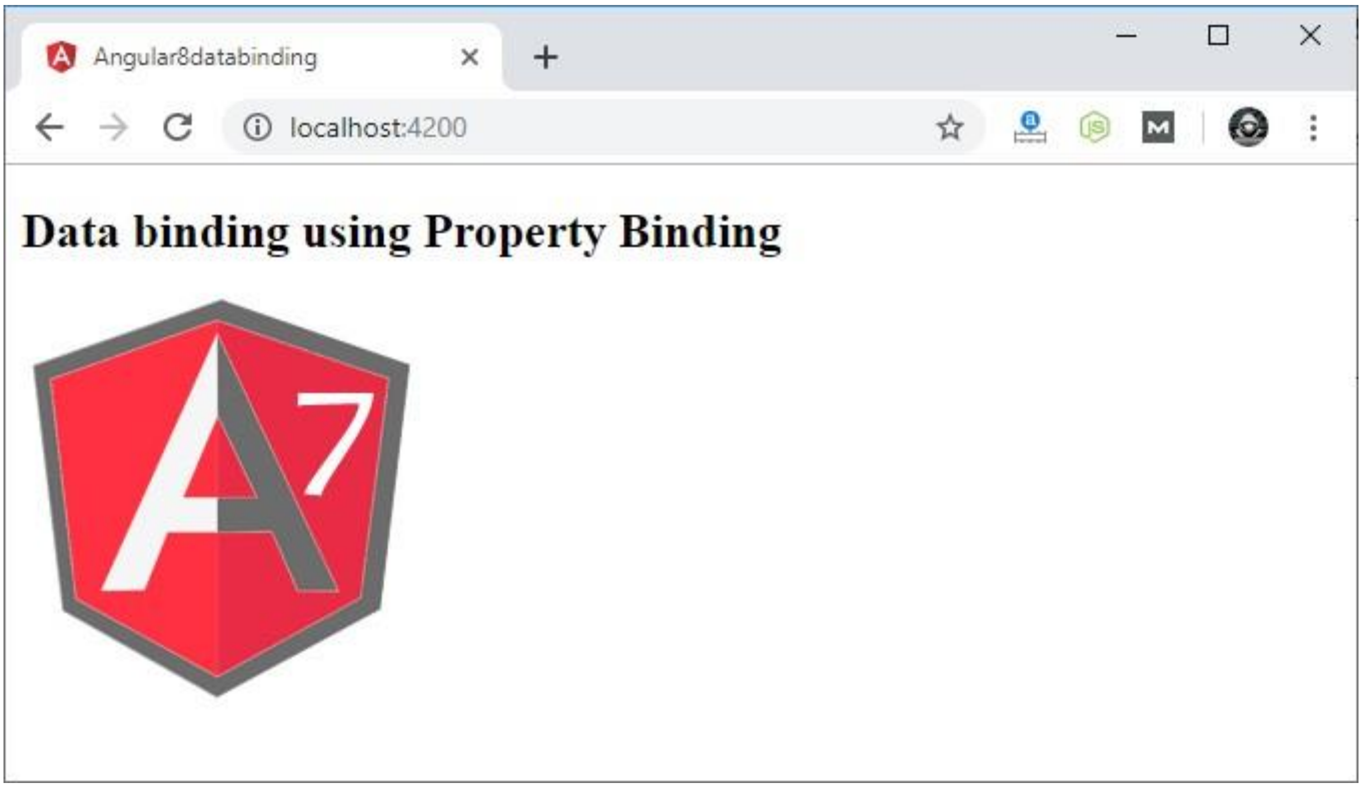
1. `<h2>{{ title }}</h2> <!-- String Interpolation -->`
2. `<img [src]="imgUrl" /> <!-- Property Binding -->`


File Edit Selection View Go Debug Terminal Help
app.component.html - Untitled (Workspace) - Visual Studio Code
EXPLORER
UNTITLED (WORKSPACE)
angular8databinding
e2e
node_modules
src
app
app-routing.module.ts
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
assets
environments
favicon.ico
index.html
main.ts
polyfills.ts
OUTLINE
app.component.ts
app.component.html
angular8databinding › src › app › app.component.html › ...
Ln 2, Col 51 Spaces: 2 UTF-8 LF HTML

```
<h2>{{ title }}</h2> <!-- String Interpolation -->
<img [src]="imgUrl" /> <!-- Property Binding -->
```

Run the ng serve command and open local host to see the result.

Output:



Data binding using Property Binding

Event Binding in Angular 8

In Angular 8, event binding is used to handle the events raised from the DOM like button click, mousemove etc. When the DOM event happens (eg. click, change, keyup), it calls the specified method in the component. In the following example, the cookBacon() method from the component is called when thebutton is clicked:

For example:

```
<button (click)="cookBacon()"></button>
```

Event Binding Example

Let's take a button in the HTML template and handle the click event of this button. To implement eventbinding, we will bind click event of a button with a method of the component.

Now, open the **app.component.ts** file and use the following code:

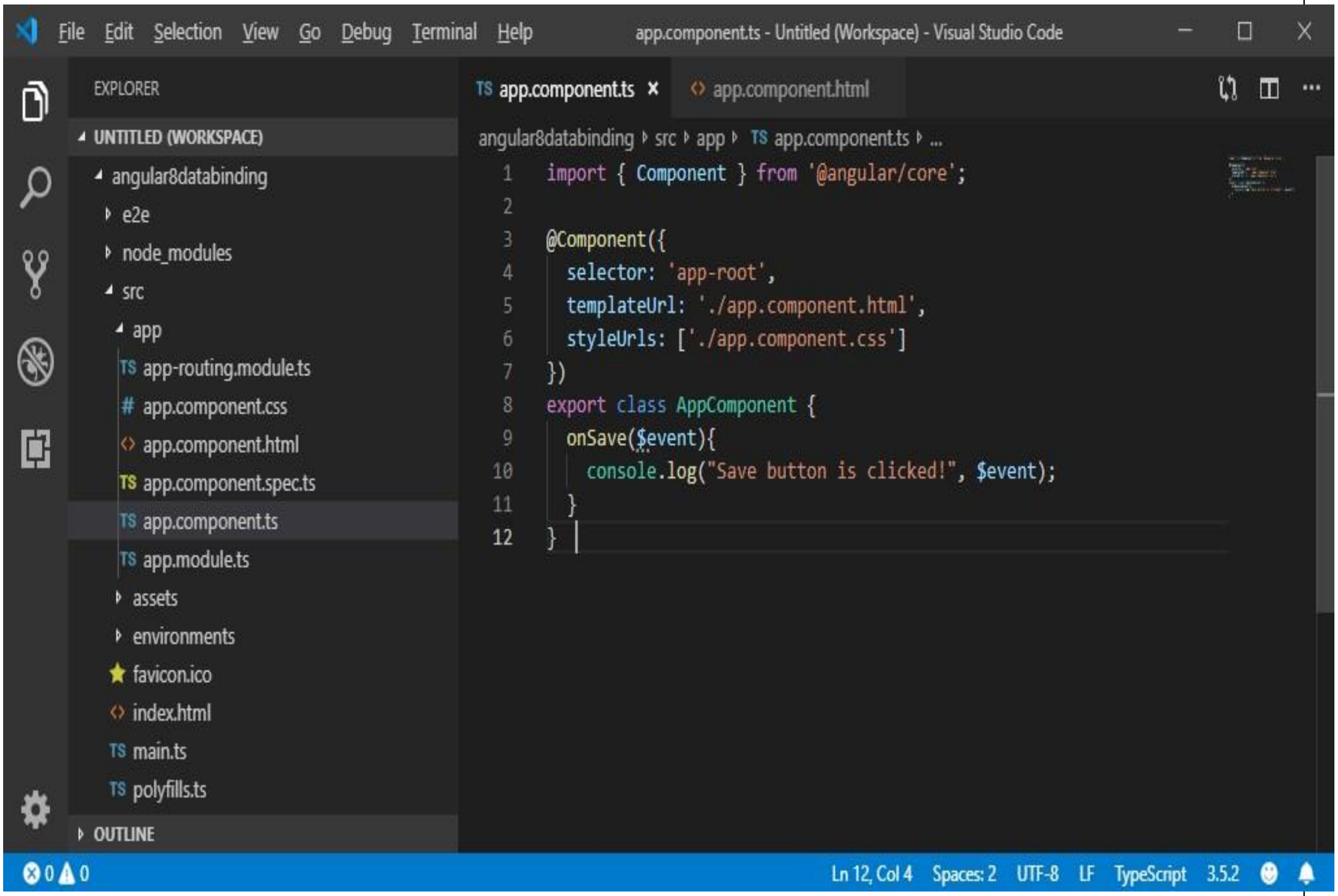
Backward Skip 10sPlay VideoForward Skip 10s

ADVERTISEMENT

```
import { Component } from '@angular/core';
@Component({
```

```
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
```

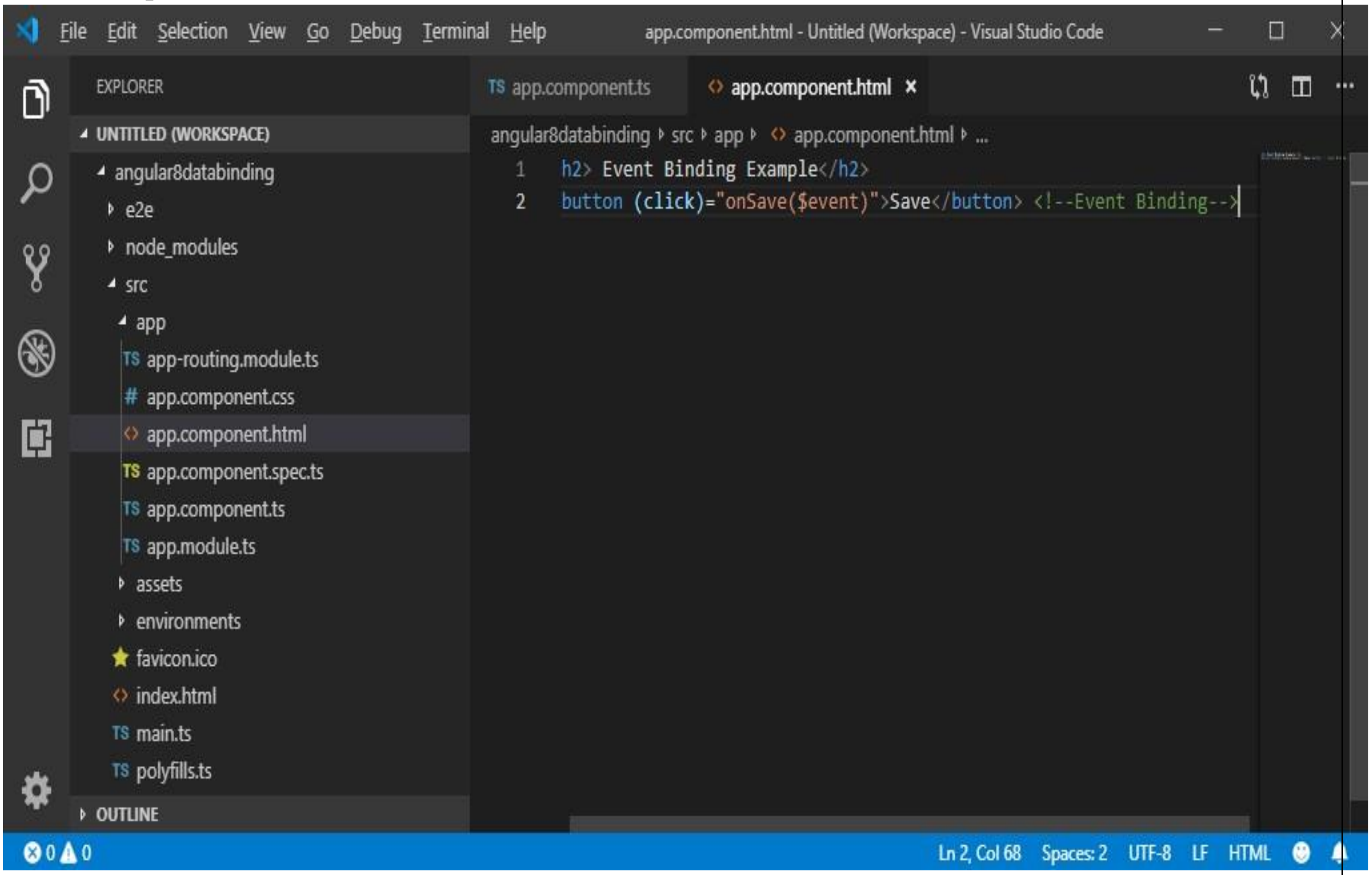
```
})
export class AppComponent {
  onSave($event){
    console.log("Save button is clicked!", $event);
  }
}
```



```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  onSave($event){
    console.log("Save button is clicked!", $event);
  }
}
```

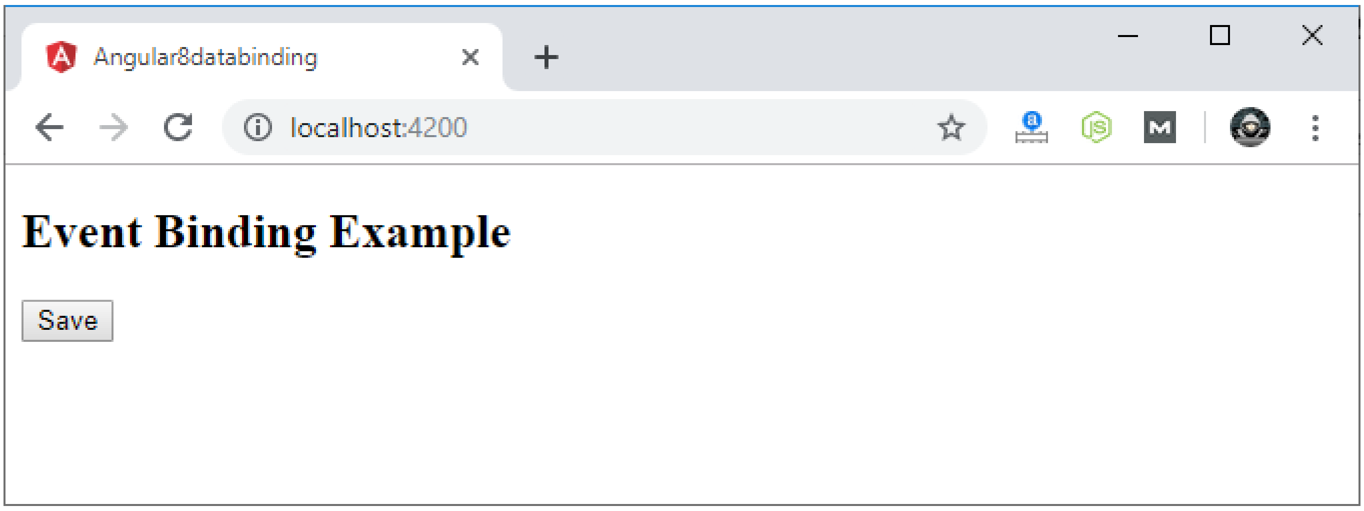
app.component.html:

```
<h2> Event Binding Example</h2>
<button (click)="onSave($event)">Save</button> <!--Event Binding-->
```



```
h2> Event Binding Example</h2>
button (click)="onSave($event)">Save</button> <!--Event Binding-->
```

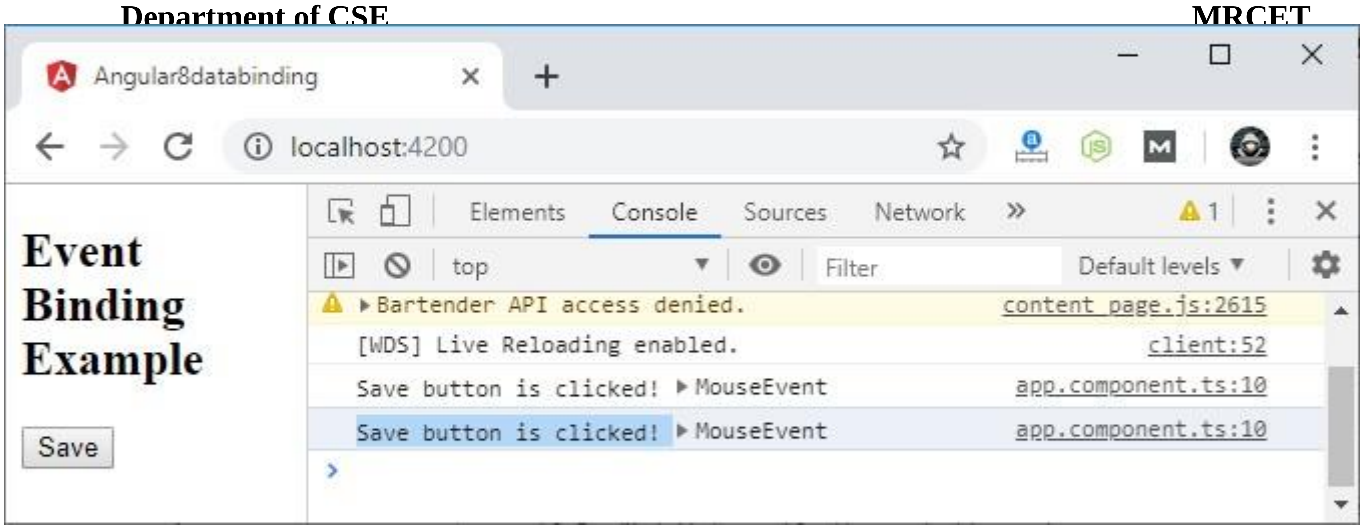
Output:



Event Binding Example

Save

Click on the "Save" button and open console to see result.



```
▶ Bartender API access denied.                 content_page.js:2615
[WDS] Live Reloading enabled.                  client:52
Save button is clicked! ▶ MouseEvent           app.component.ts:10
Save button is clicked! ▶ MouseEvent           app.component.ts:10
```

Now, you can see that the "Save" button is clicked.

4. Class Binding

Last Updated : 23 Sep, 2020

Class binding in Angular makes it very easy to set the class property of a view element. We can set or remove the CSS class names from an element's class attribute with the help of class binding.We bind a class of a DOM element to a field that is a defined property in our Typescript Code. Its syntax is like that of property binding.

Syntax:

```
<element [class] = "typescript_property">
```

Approach:

- Define a property element in the app.component.ts file.
- In the app.component.html file, set the class of the HTML element by assigning the propertyvalue to the app.component.ts file's element.

Example 1: Setting the class element using class binding.

app.component.html

- HTML

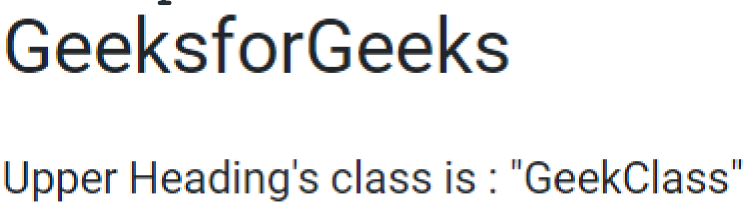
```
<h1 [class] = "geeky">

  GeeksforGeeks

</h1>

Upper Heading's class is : "{{ g[0].className }}"
```

app.component.ts

- Javascript

```
import { Component, OnInit } from '@angular/core';



@Component({

    selector: 'app-root',

    templateUrl: './app.component.html'

})

export class AppComponent {

    geeky = "GeekClass";

    g = document.getElementsByClassName(this.geeky);

}
```

Output:



```
GeeksforGeeks

Upper Heading's class is : "GeekClass"
```

5.Style Binding

Last Updated : 14 Sep, 2020

8.

It is very easy to give the CSS styles to HTML elements using style binding in Angular 8. Style binding is used to set a style of a view element. We can set the inline styles of an HTML elementusing the style binding in angular. You can also add styles conditionally to an element, hence creating a dynamically styled element.

Syntax:

```
<element [style.style-property] = "'style-value'">
```

Example 1: **app.component.html**

:

- HTML

```
<h1 [style.color] = "'green'"

    [style.text-align] = "'center'" >

  GeeksforGeeks

</h1>
```

Output:

GeeksforGeeks

b. Two way Data Binding using ngmodel

We have seen that in one-way data binding any change in the template (view) were not be reflected inthe component TypeScript code. To resolve this problem, Angular provides two-way data binding. Thetwo-way binding has a feature to update data from component to view and vice-versa.

In two-way databinding, automatic synchronization of data happens between the Model and the View.Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

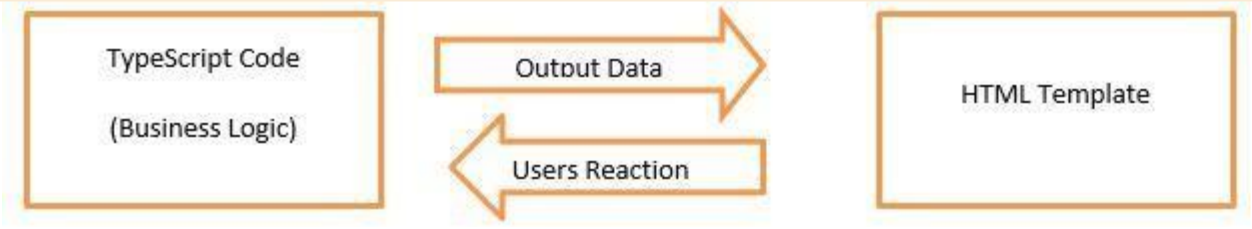
This happens immediately and automatically, ensures that the HTML template and the TypeScript codeare updated at all times.

In two way data binding, **property binding and event binding** are combined together.

Syntax:

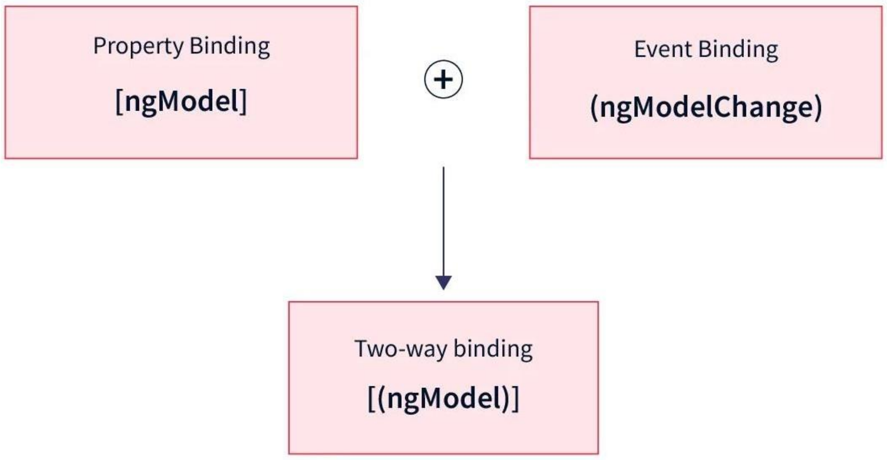
1. [(ngModel)] = "[property of your component]"

*Note: **For two way data binding, we have to enable the ngModel directive. It depends upon FormsModule in angular/forms package, so we have to add FormsModule in imports[] array in the AppModule.***



Let's take an example to understand it better.

Note:-when you are using ngmodel import FormsModule



[property binding] + (event binding) = [(property)]

--

[ngModel] + (ngModelChange) = [(ngModel)]

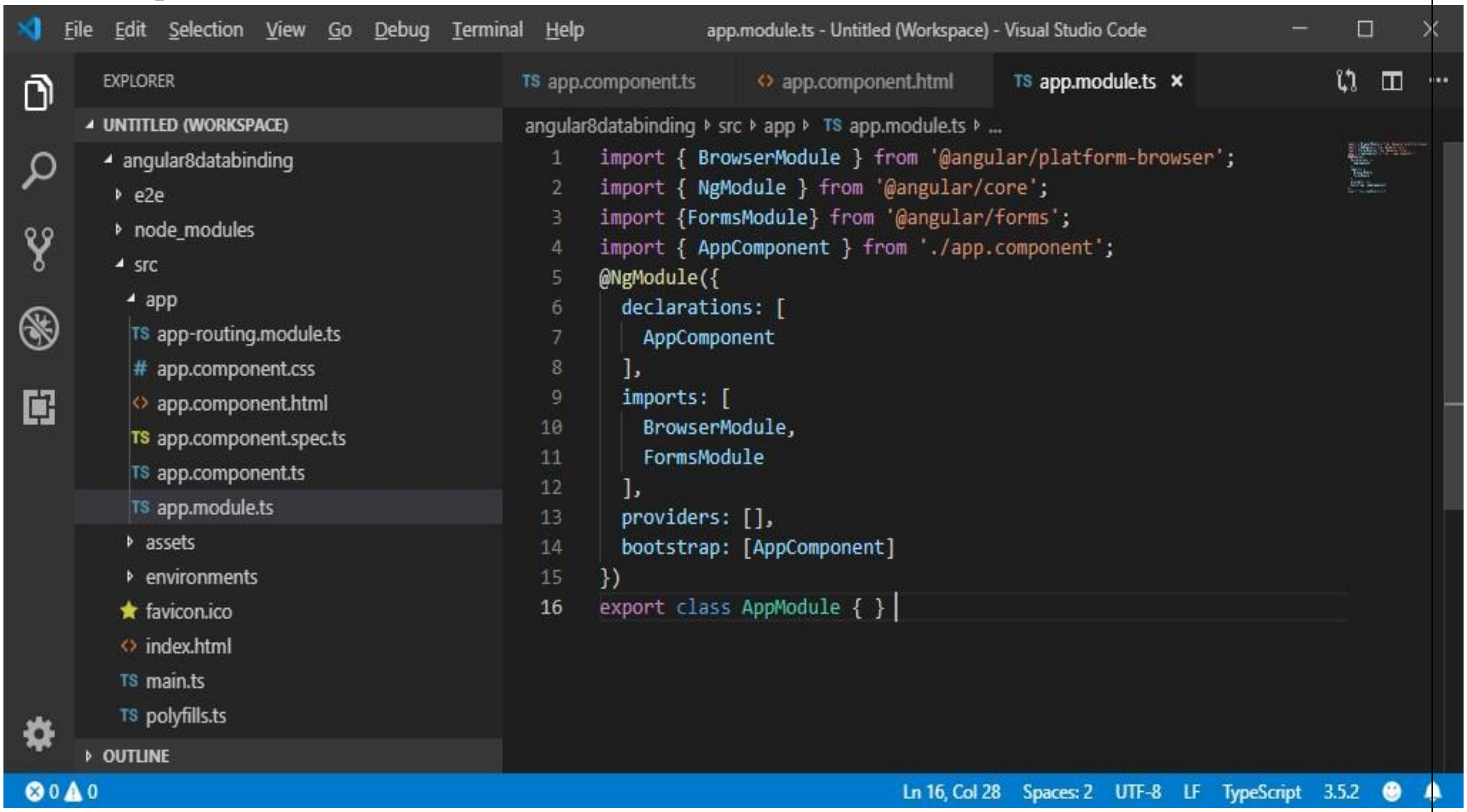
--

[text] + (textChange) = [(text)]

Open your project's **app.module.ts** file and use the following code:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
  AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
],
  providers: [],
  bootstrap: [AppComponent]
})
```

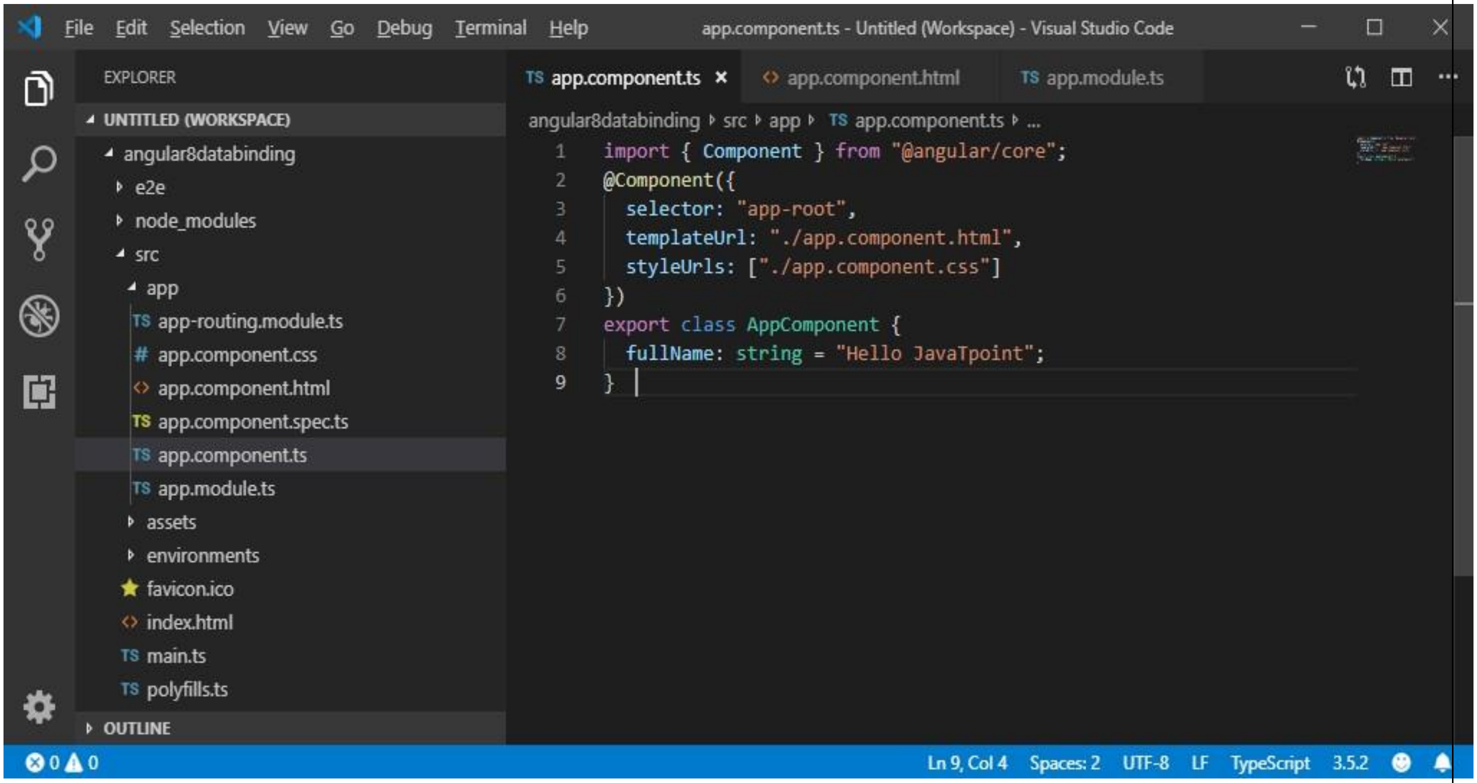
```
export class AppModule { }
```



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts file:

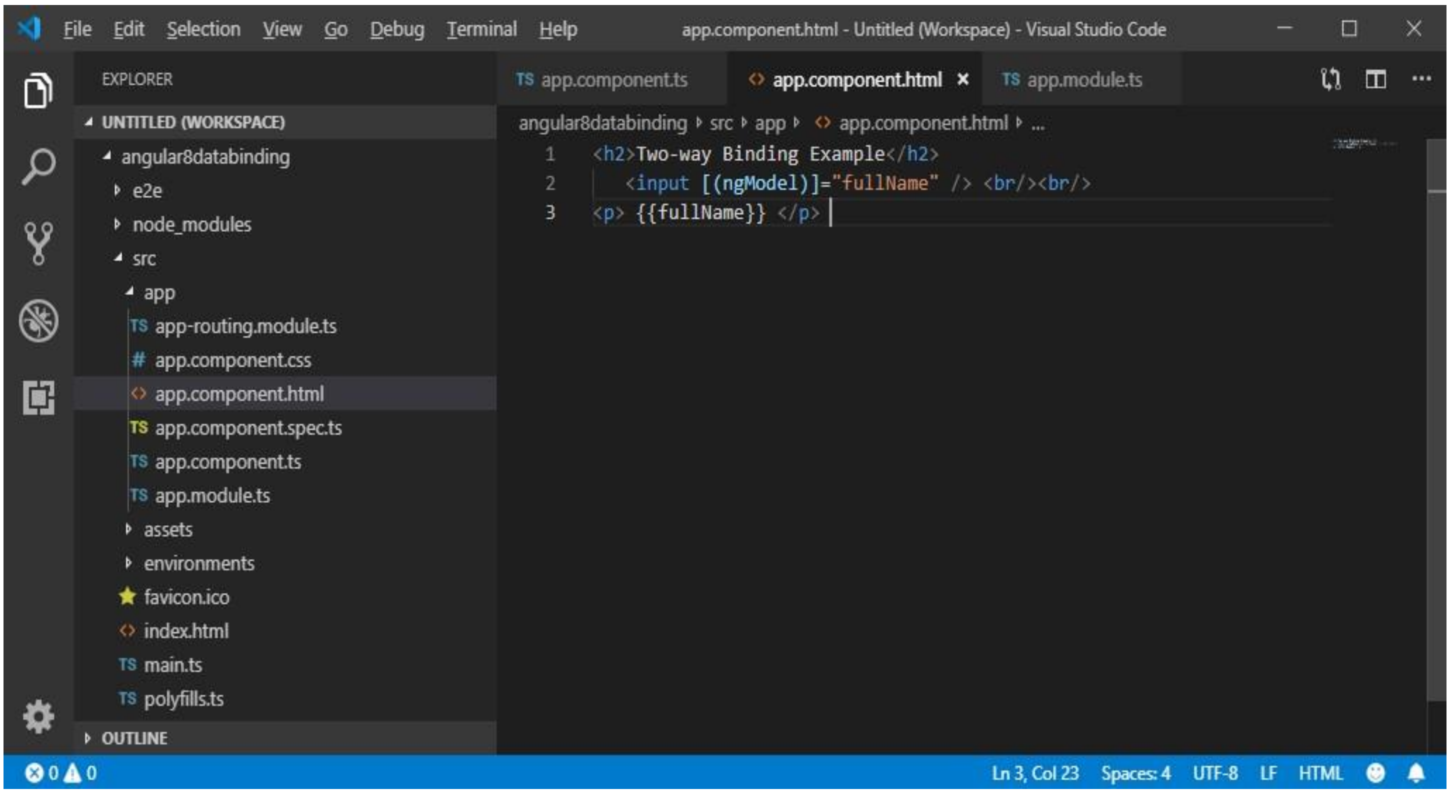
```
import { Component } from "@angular/core";
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  fullName: string = "Hello JavaTpoint";
}
```



```
import { Component } from "@angular/core";
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  fullName: string = "Hello JavaTpoint";
}
```

app.component.html file:

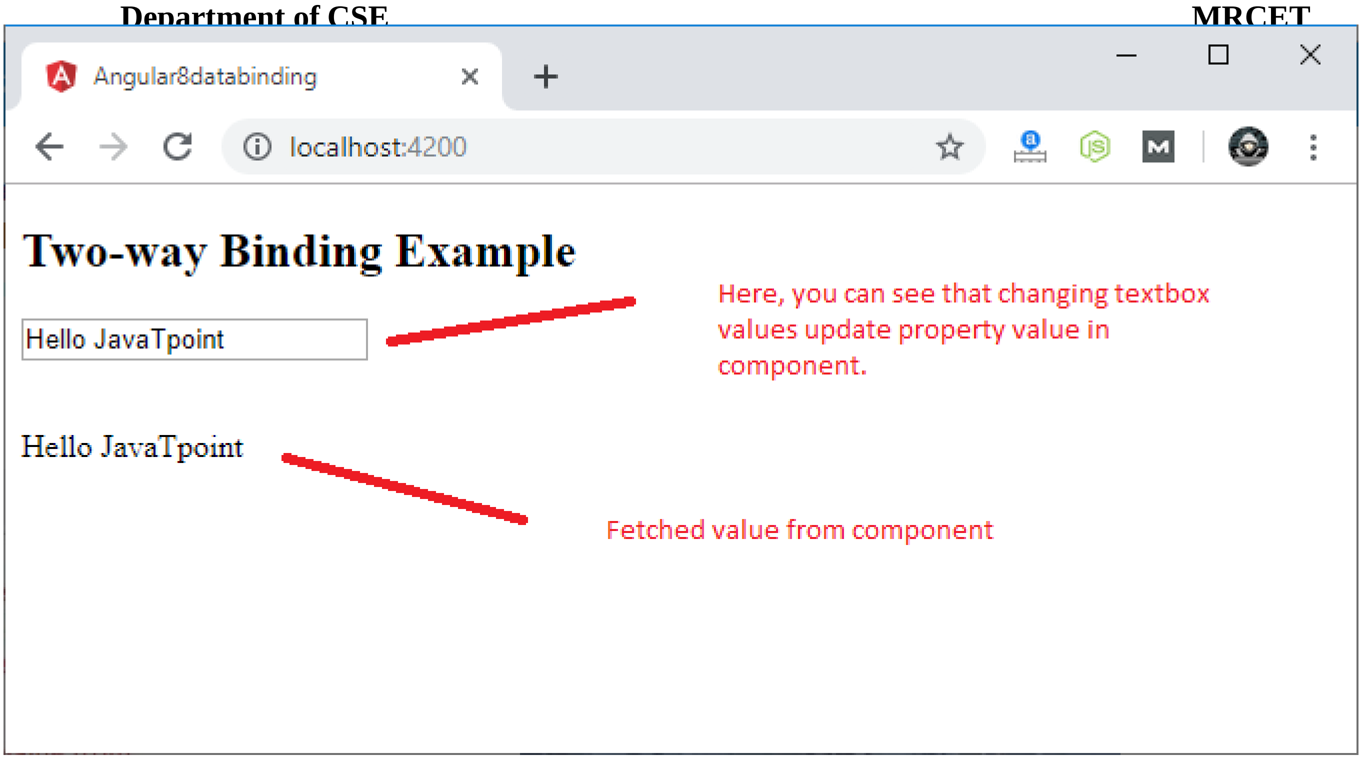
```
1. <h2>Two-way Binding Example</h2>
2.   <input [(ngModel)]="fullName" /> <br/><br/>
3. <p> {{fullName}} </p>
```



```
<h2>Two-way Binding Example</h2>
  <input [(ngModel)]="fullName" /> <br/><br/>
<p> {{fullName}} </p>
```

Now, start your server and open local host browser to see the result.

Output:



Two-way Binding Example

Hello JavaTpoint

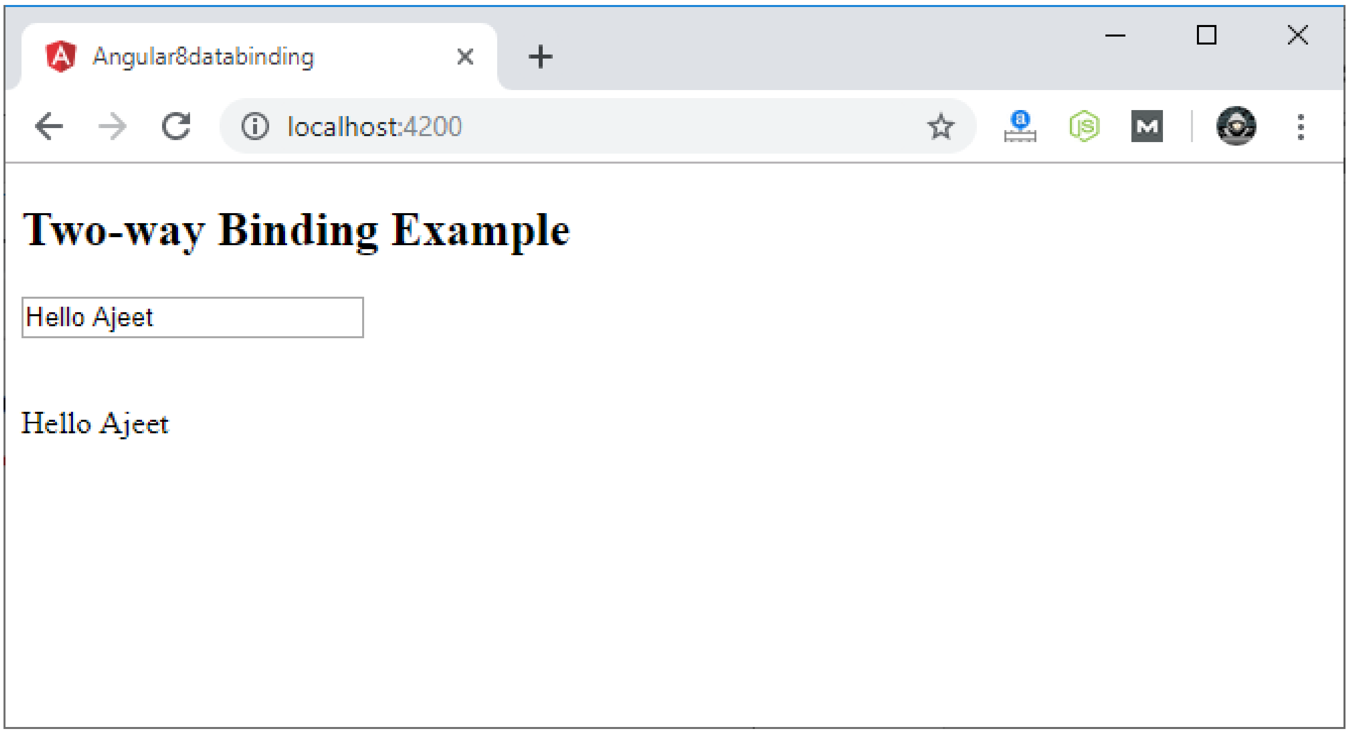
Hello JavaTpoint

Here, you can see that changing textbox values update property value in component.

Fetched value from component

You can check it by changing textbox value and it will be updated in component as well.

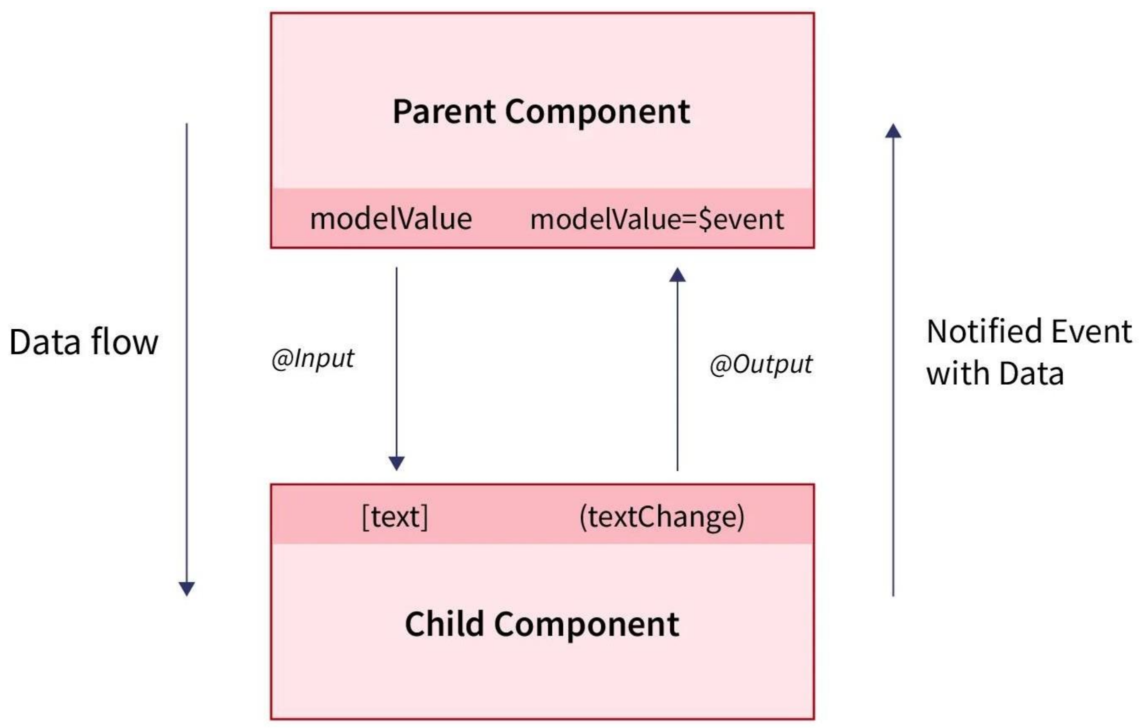
For example:



Two-way Binding Example

Hello Ajeet

Hello Ajeet



(OR)

Without using ngmodel

[property binding] + (event binding) = [(property)]

app.component.html

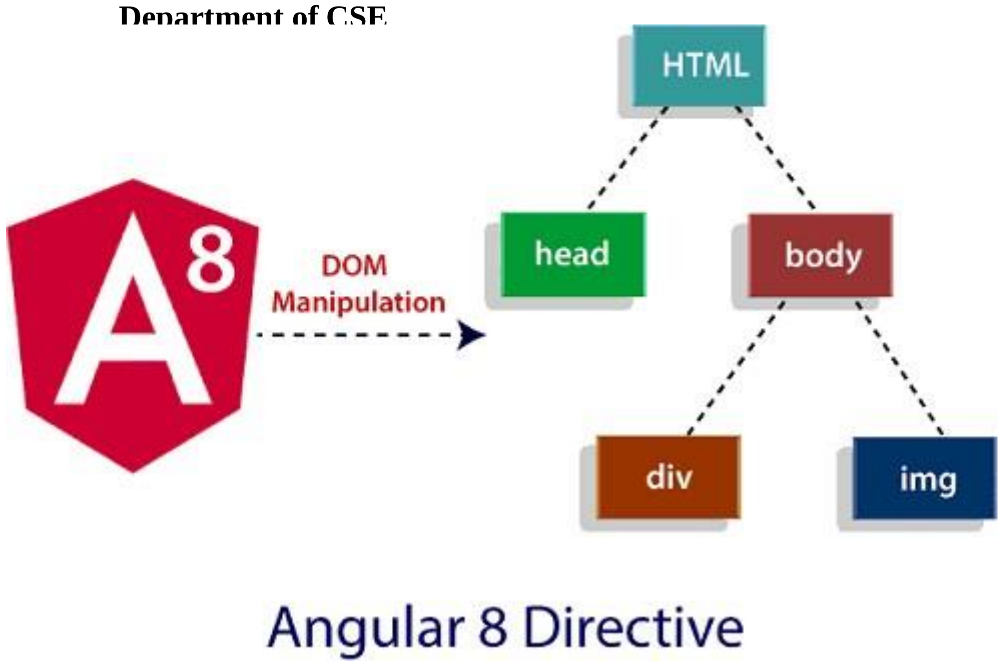
```
<label>User Name</label>
<input type="text" [value]="text" (input)="updateValue
($event)">
<h1>{{text}}</h1>
```

app.component.ts

```
fullName: string = "Hello JavaTpoint";
```

Angular Directives

The Angular 8 directives are used to manipulate the DOM. By using Angular directives, you can changethe appearance, behavior or a layout of a DOM element. It also helps you to extend HTML.



Angular 8 directives can be classified in 3 categories based on how they behave:

- Component Directives
- Structural Directives
- Attribute Directives

Component Directives: Component directives are used in main class. They contain the detail of howthe component should be processed, instantiated and used at runtime.

Structural Directives: Structural directives start with a * sign. These directives are used to manipulateand change the structure of the DOM elements. For example, *ngIf directive, *ngSwitch directive, and
*ngFor directive.

- ***ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
- ***ngSwitch Directive:** The *ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C#.
- ***ngFor Directive:** The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

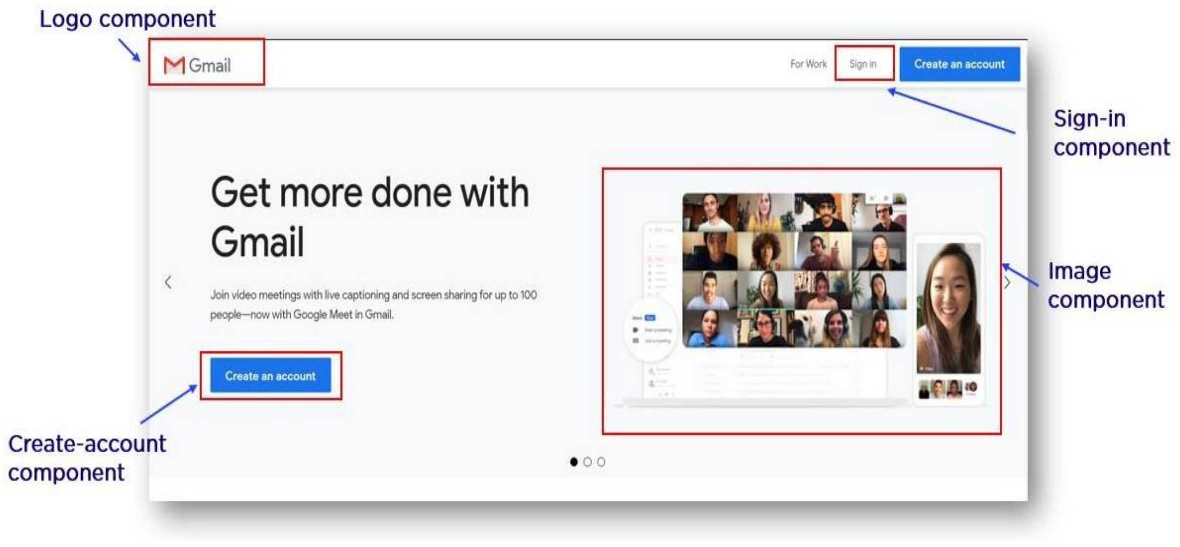
Attribute Directives: Attribute directives are used to change the look and behavior of the DOMelements. For example: ngClass directive, and ngStyle directive etc.

- **ngClass Directive:** The ngClass directive is used to add or remove CSS classes to an HTML element.
- **ngStyle Directive:** The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element.

5. Fetch Data from a Service

What is the Need for Angular Services?

We're sure you are aware of the concept of components in Angular. The user interface of the application isdeveloped by embedding several components into the main component.

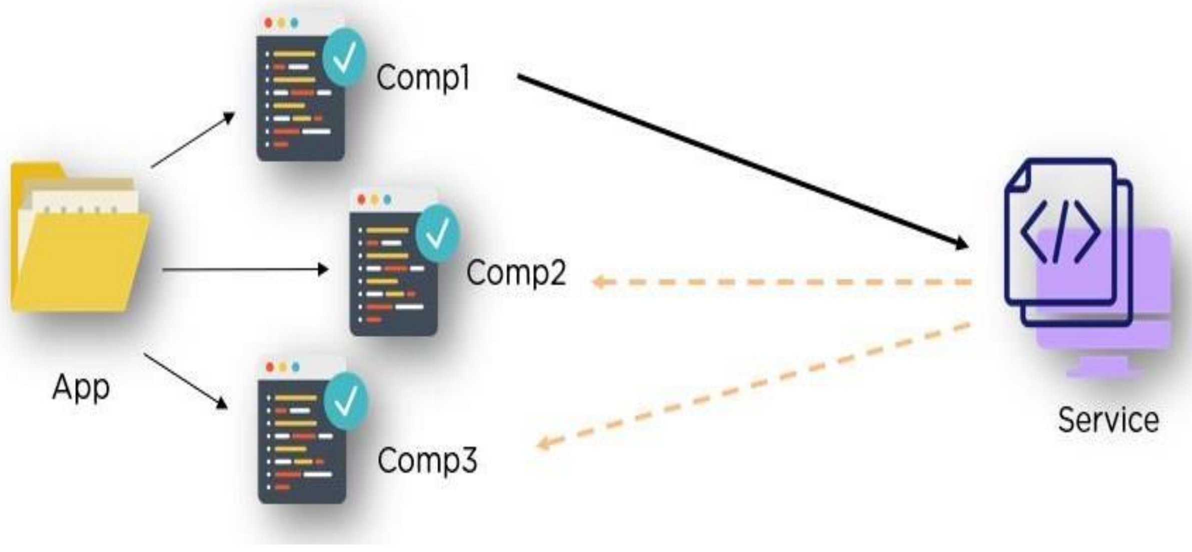


However, these components are generally used only for rendering purposes. They are only used to define whatappears on the user interface. Ideally, other tasks, like data and image fetching, network connections, databasemanagement, are not performed. Then how are these tasks achieved? And what if more than one component performs similar tasks? Well, Services take care of this. They perform all the operational tasks for the components.

- Services avoid rewriting of code. A service can be written once and injected into all the componentsthat use that service
- A service could be a function, variable, or feature that an application needs

What Are Angular Services?

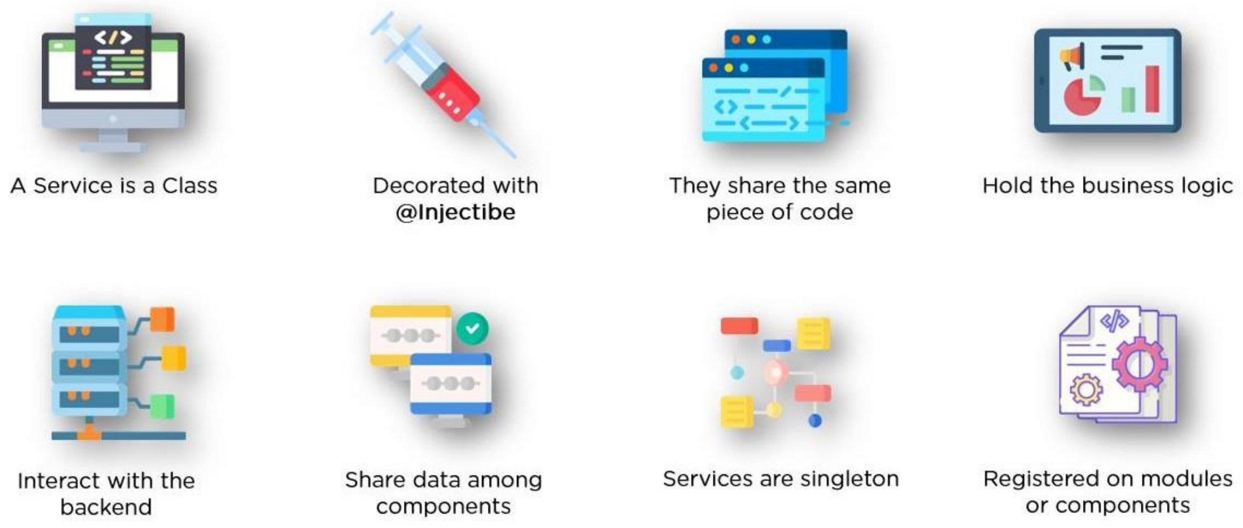
Angular services are objects that get instantiated just once during the lifetime of an application. They containmethods that maintain data throughout the life of an application, i.e., data is available all the time.



The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application. They are usually implemented through dependency injection.

Features of Angular Services

- Services in Angular are simply typescript classes with the @injectible decorator. This decorator tells angularthat the class is a service and can be injected into components that need that service. They can also inject other services as dependencies.
- As mentioned earlier, these services are used to share a single piece of code across multiple components.These services are used to hold business logic.
- Services are used to interact with the backend. For example, if you wish to make AJAX calls, you can havethe methods to those calls in the service and use it as a dependency in files.


A Service is a Class
Decorated with @Injectibe
They share the same piece of code
Hold the business logic
Interact with the backend
Share data among components
Services are singleton
Registered on modules or components

- In angular, the components are singletons, meaning that only one instance of a service that gets created, andthe same instance is used by every building block in the application.

- A service can be registered as a part of the module, or as a part of the component. To register it as a part ofthe component, you'll have to specify it in the providers' array of the module.

Fetch data from service Example:-

Use this command **ng g s service_name**

Creating Angular Project Use below Commands

- npm install -g @angular/cli //creating cli
- ng version
- ng new prog9 --standalone false //creating angular project SPA(Single page application) //app component is a default component.
- cd prog9
- ng g c header //creating header component
- ng g c home //creating home component
- ng g c profile //creating profile component
- ng serve //running angular project
- Use this command for creating service **ng g s service_name**

Creating body of about, contact, home and header. Header is a navbar this page creating routerLinks about andcontact. Home is a default link when header loaded it is displayed. App.module.ts file creating url paths for each page.

Here test.service.ts file is creating for displaying fruits names you can access service data any component herefetching data service to about.

1) about.component.html:-

```
<h1>This is About Component</h1>
<h3>Which Fruit You Like?</h3>

<br>
<div *ngFor="let m of names">
  {{m}}
</div>
```

About.component.ts:-

```
import { Component } from
'@angular/core';import { TestService }
from '../test.service'; @Component({
  selector: 'app-about',
  templateUrl:
  './about.component.html',styleUrl:
```

```
  './about.component.css'
})
export class AboutComponent {

  constructor(private

  ts:TestService){

  }
  names=this.ts.names;

}
```

2) **contact.component.html:-**

```
<h1>This is Contact Component</h1>
```

3) **header.component.css:-**

```
ul li{
   list-style: none;
}
ul li a{
   text-decoration: none;
}
ul{
   display: flex;
   justify-content: flex-
   start;gap: 20px;
   background-color:
   aqua;height: 50px;
}
a{
   line-
   height:50px;
   color:black;
   margin:0 20px;

   font-weight:

   bold;font-

   size:30px;

}
```

4) header.component.html:-

```
<ul>

   <li>
```

```
    <a routerLink="/about" router="active">about</a>
  </li>
  <li>
    <a routerLink="/contact" routerActiveLink="active">contact</a>
  </li>
</ul>
```

5) home.component.html:-

```
<h1>This is Home Component</h1>
```

6) notfound.component.html:-

```
<p>notfound works!</p>
```

7) app.component.html:-

```
<app-header></app-header>
<router-outlet></router-outlet>
```

8) app.module.ts:-

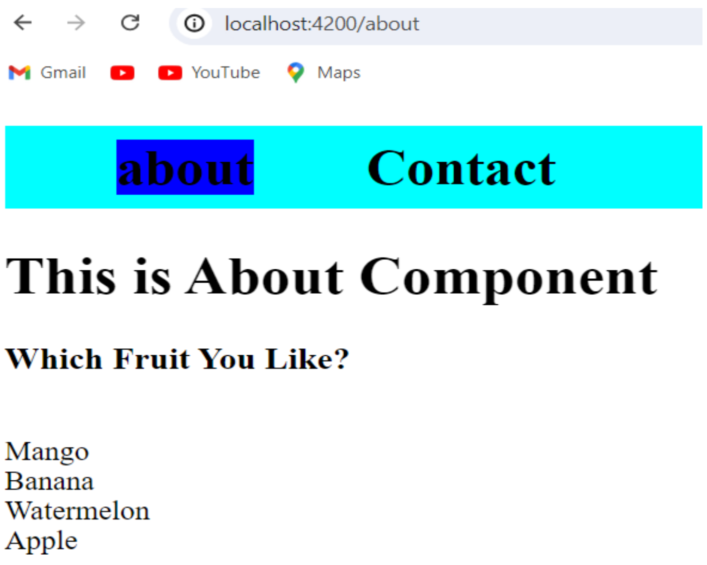
```
import { AppComponent } from './app.component';
import { HeaderComponent } from
'./header/header.component'; import {
AboutComponent } from './about/about.component';
import { ContactComponent } from
'./contact/contact.component';import {
HomeComponent } from './home/home.component';
import { NotfoundComponent } from
'./notfound/notfound.component';import {
RouterModule,Routes } from '@angular/router';
```

```
const routes:Routes=[
 {
  path:'',component:HomeComponent
 },
 {
  path:'about',component:AboutComponent
 },
 {
  path:'contact',component:ContactComponent
 },
 {
  path:'**',component:NotfoundComponent
 }
]

imports: [

  RouterModule.forRoot(routes)
 ],
```

Test.service.ts:-

```
import { Injectable } from '@angular/core';

@Injectable({
 providedIn:
 'root'
})
export class TestService {

 constructor() { }
 names=['Mango','Banana','Watermelon','Apple'];
}
```

Output:



This is About Component

Which Fruit You Like?

```
Mango
Banana
Watermelon
Apple
```

6. Submit data to service:-

npm install bootstrap --save

When Bootstrap is installed open angular.json file and add bootstrap.min.css file reference under"styles":

```
"styles": [
    "src/styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Now we need to create components and service. Use the following commands to create the same.

ng g c header
ng g c reg

Note
g stands for generate | c stands for Component | s stands for

ServiceOpen app.modules.ts file and add these lines:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { RegComponent } from './reg/reg.component';
import { HeaderComponent } from './header/header.component';
import { FormsModule,ReactiveFormsModule } from '@angular/forms';
import { RouterModule,Routes } from '@angular/router';

const routes: Routes = [

  { path: "reg", component: RegComponent }
];

  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule.forRoot(routes)
  ]
```

In app.component.html replace the existing code with the below code:

```
<app-header></app-header>
    <router-outlet></router-outlet>
```

Let's start with components now.

Open header.component.html file and replace with the code below.

```
<ul>
    <li>
        <a routerLink="/reg">create</a>
    </li>
</ul>
```

Open header.component.css file and replace with the code below.

```
ul{
    background-color:aqua;
}
```

Now let's create a function in Service.

Open data.service.ts file and replace with the code below.

```
public SaveEmployee(empdata) {
    console.log("Full Name : " + empdata.regFullName);
    console.log("Email Id : " + empdata.regEmail);
}
```

Open reg.component.html file and replace with the code below.

```
<div class="container" style="margin-top: 150px;">
    <form [formGroup]="frmRegister" (ngSubmit)="SaveEmployee(frmRegister.value)">
        <div class="panel panel-primary">
            <div class="panel-heading">
                <h3 class="panel-title">Employee Registration</h3>
            </div>
            <div class="panel-body">
                <div class="form-group">
                    <label for="fullName">Full Name</label>
                    <input id="fullName" formControlName="regFullName" type="text" class="form-control" required />
                </div>
                <div class="form-group">
                    <label for="email">Email</label>
                    <input id="email" formControlName="regEmail" type="email" class="form-control" required />
                </div>
            </div>
            <div class="panel-footer">
                <button type="submit" class="btn btn-primary">Save</button>
            </div>
        </div>
        </form>
</div>
```

Open reg.component.ts file and replace with the code below.

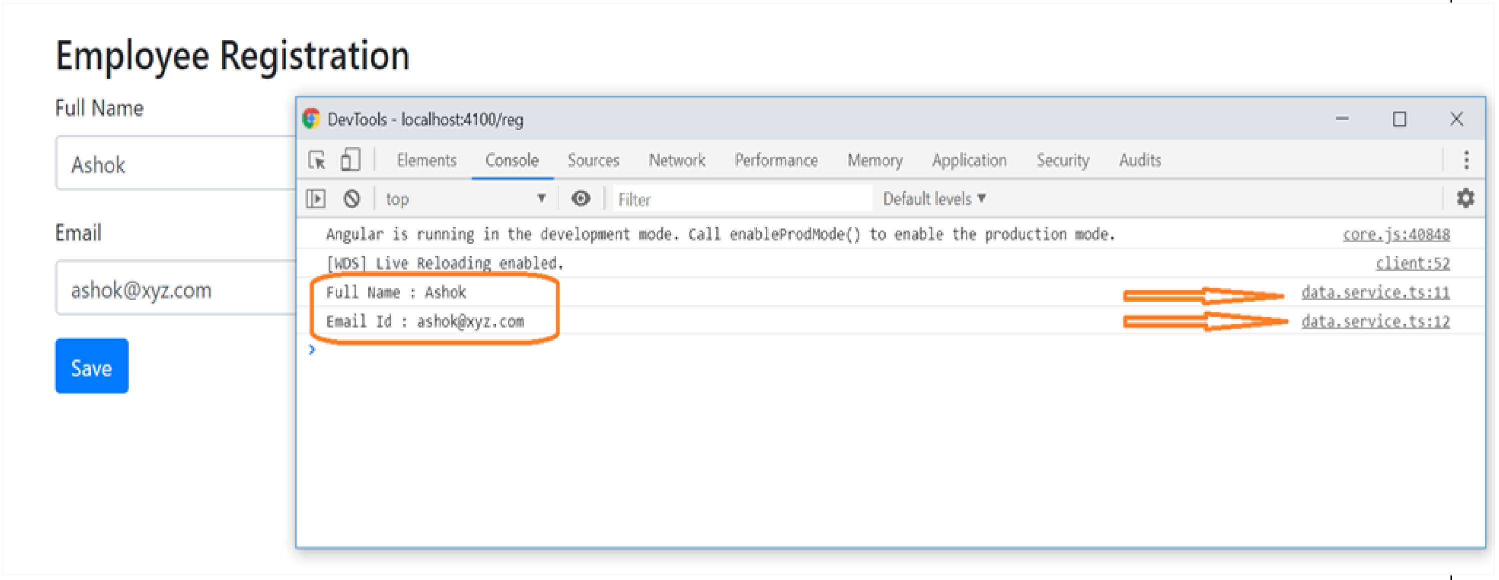
```
import {
      Component,
      OnInit
} from '@angular/core';
```

```
import {
    FormGroup,
    FormBuilder
} from '@angular/forms';
import {
    DataService
} from '../data.service';
@Component({
    selector: 'app-reg',
    templateUrl: './reg.component.html',
    styleUrls: ['./reg.component.css']
})
export class RegComponent implements OnInit {
    frmRegister: FormGroup;
    constructor(private _fb: FormBuilder, private dataservice: DataService)
{}
    ngOnInit(): void {
        this.frmRegister = this._fb.group({
            regFullName: "",
        regEmail: ""
        });
    }
    SaveEmployee(value) {
        this.dataservice.SaveEmployee(value);
    }
}
```

Now build your application by ng

build.Run application by **ng serve**.

Output:-



```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.   core.js:40848
[WDS] Live Reloading enabled.                                                                        client:52
Full Name : Ashok                                                                          data.service.ts:11
Email Id : ashok@xyz.com                                                                   data.service.ts:12
```

7. Http Module:-

Defination:-

$http is an AngularJS service for reading data from remote servers. Implements an HTTP client API for Angular apps that relies on the XMLHttpRequest interface exposed by browsers. Includes testability features, typed request and response objects, request and response interception, observable APIs, and streamlined error handling.

These components are self-sufficient and can be used on their own without being tied to a specific **NgModule**. But, sometimes, when you're working with these standalone components, you might need to fetch data from servers or interact with APIs using HTTP requests.

We need to import the http module to make use of the http service. Let us consider an example to understandhow to make use of the http service.

Example1:-Fetching data from API and displayed console

To start using the http service, we need to import the module in **app.module.ts** as shown below −

```
import { BrowserModule } from '@angular/platform-
browser';import { NgModule } from '@angular/core';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';import { HttpClientModule } from
'@angular/common/http';
import { AppComponent } from
'./app.component';@NgModule({
  declarations: [
    AppCompon
    ent
  ],
  imports: [
    BrowserModu
    le,
    BrowserAnimationsMod
    ule,HttpClientModule
  ],
```

If you see the highlighted code, we have imported the HttpClientModule from @angular/common/http and thesame is also added in the imports array.

Let us now use the http client in the **app.component.ts**.

```
import { Component } from '@angular/core';
import { HttpClient } from
'@angular/common/http';@Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
  styleUrls:
  ['./app.component.css']
})
export class AppComponent {
  constructor(private http:
    subscribe((data) ⇒ console.log(data))
  }
}
```

Let us understand the code highlighted above. We need to import http to make use of the service, which is doneas follows −

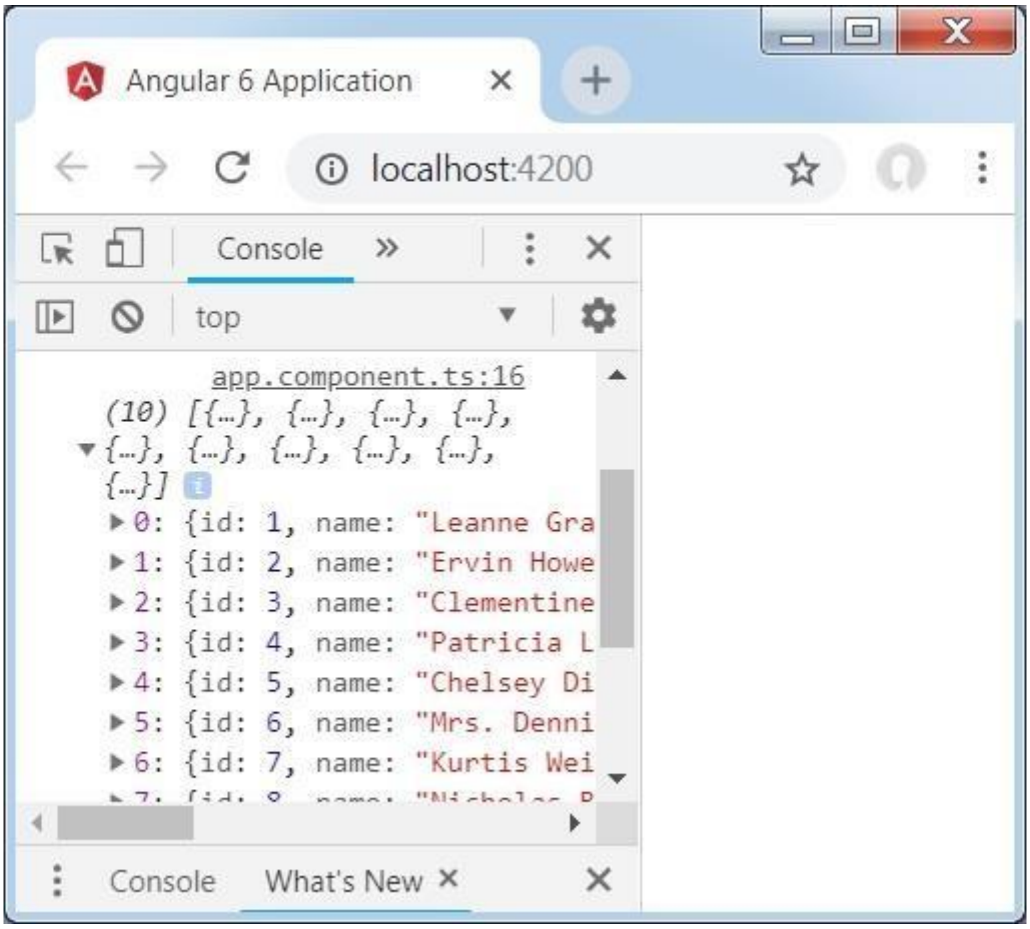
import { HttpClient } from '@angular/common/http';

In the class **AppComponent**, a constructor is created and the private variable http of type Http. To fetch thedata, we need to use the **get API** available with http as follows

this.http.get();

It takes the url to be fetched as the parameter as shown in the code.

We will use the test url − https://jsonplaceholder.typicode.com/users to fetch the json data. The subscribe willlog the output in the console as shown in the browser −



```
app.component.ts:16
(10) [{…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…},
{…}]
▶0: {id: 1, name: "Leanne Gra
▶1: {id: 2, name: "Ervin Howe
▶2: {id: 3, name: "Clementine
▶3: {id: 4, name: "Patricia L
▶4: {id: 5, name: "Chelsey Di
▶5: {id: 6, name: "Mrs. Denni
▶6: {id: 7, name: "Kurtis Wei
```

If you see, the json objects are displayed in the console. The objects can be displayed in the browser too.

Example2:-

For the objects to be displayed in the browser, update the codesin **app.component.html** and **app.component.ts** as follows −

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
```

```
@Component({
  selector: 'app-
  root',
  templateUrl:
  './app.component.html',
  styleUrls:
  ['./app.component.css']
})
export class AppComponent {
  constructor(private http:
  HttpClient) { }httpdata;
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/
```

In **app.component.ts**, using the subscribe method we will call the display data method and pass the datafetched as the parameter to it.

In the display data method, we will store the data in a variable httpdata. The data is displayed in the browserusing **for** over this httpdata variable, which is done in the **app.component.html** file.

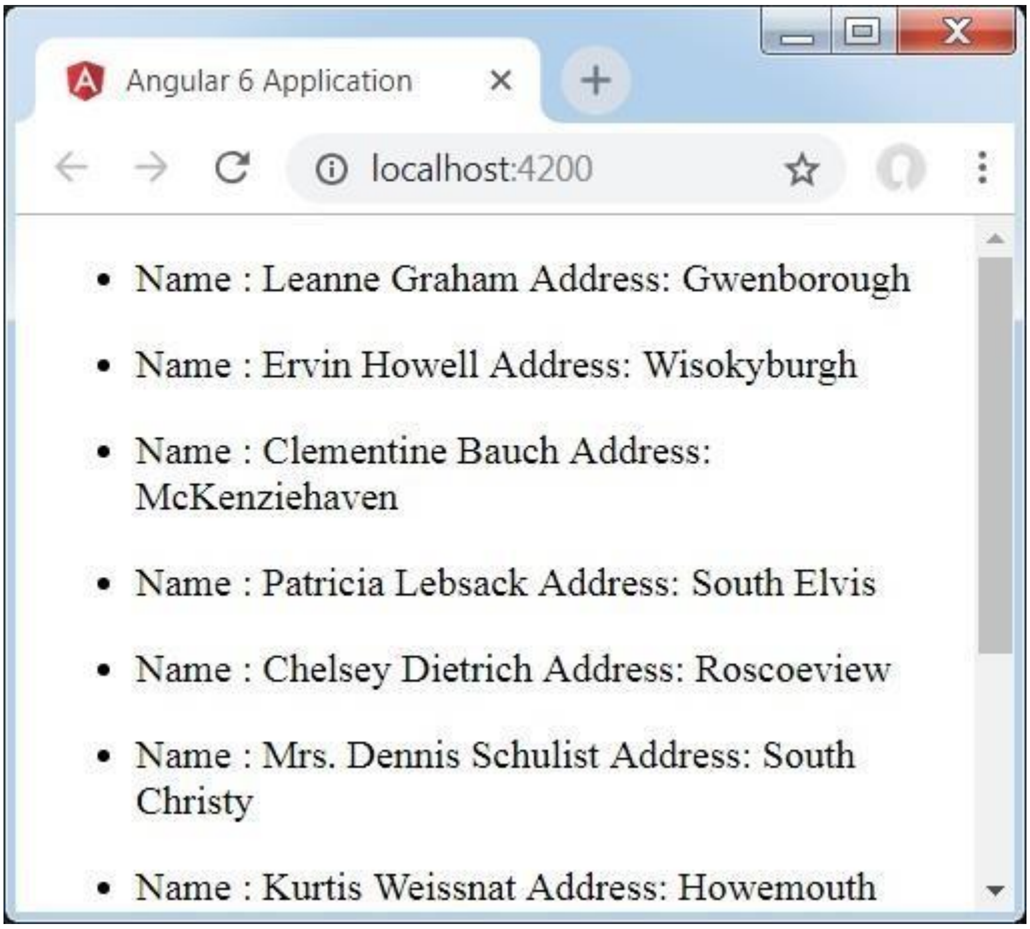
```
<ul *ngFor = "let data of httpdata">
  <li>Name : {{data.name}} Address: {{data.address.city}}</li>
</ul>
```

The json object is as follows −

```
{
  "id": 1,
  "name": "Leanne
  Graham","username":
  "Bret",
  "email": "Sincere@april.biz",

  "address": {
    "street": "Kulas
    Light", "suite": "Apt.
    556", "city":
    "Gwenborough",
    "zipcode": "92998-
    3874","geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },

  "phone": "1-770-736-8031 x56442",
  "website":
  "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server
    neural-net","bs": "harness real-time e-
    markets"
  }
}
```

The object has properties such as id, name, username, email, and address that internally has street, city, etc. andother details related to phone, website, and company. Using the **for** loop, we will display the name and the city details in the browser as shown in the **app.component.html** file.

This is how the display is shown in the browser −



- Name : Leanne Graham Address: Gwenborough
- Name : Ervin Howell Address: Wisokyburgh
- Name : Clementine Bauch Address: McKenziehaven
- Name : Patricia Lebsack Address: South Elvis
- Name : Chelsey Dietrich Address: Roscoeview
- Name : Mrs. Dennis Schulist Address: South Christy
- Name : Kurtis Weissnat Address: Howemouth

Let us now add the search parameter, which will filter based on specific data.

Example 3:-

We need to fetch the data based on the search param passed.

Following are the changes done in **app.component.html** and **app.component.ts**

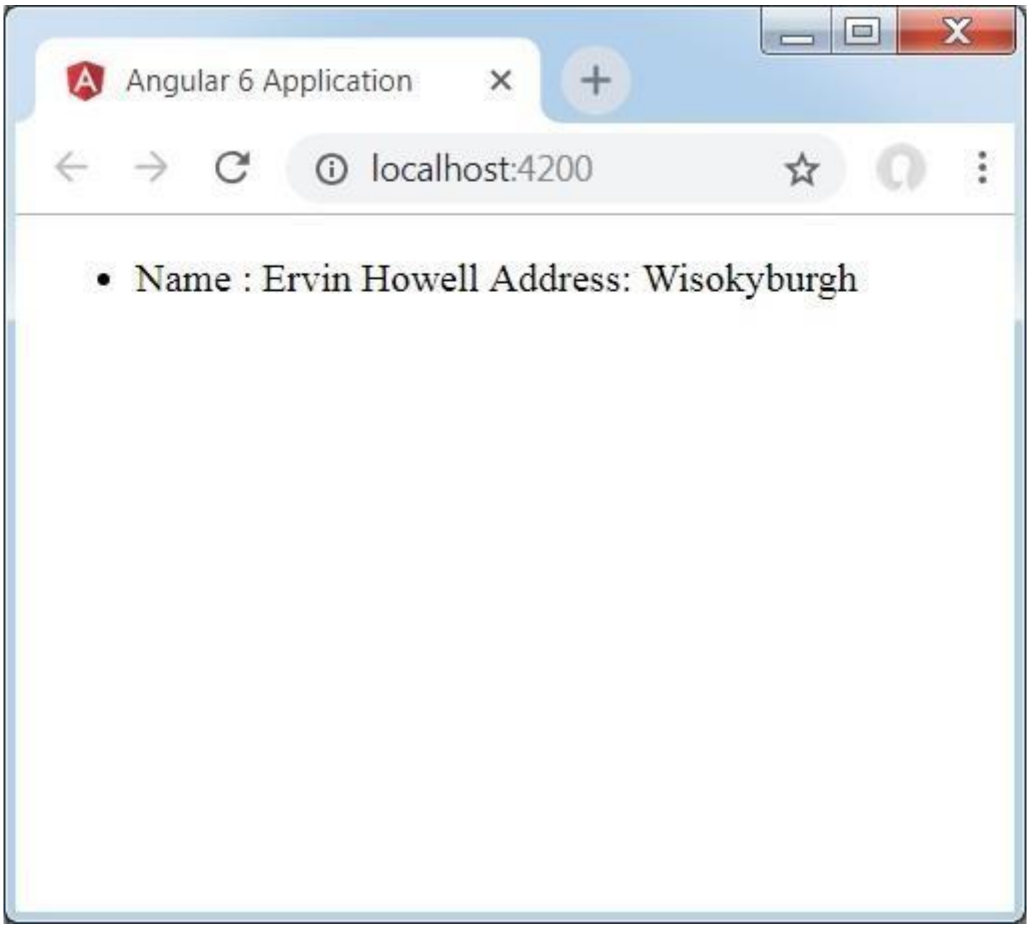
files −app.component.ts

```
import { Component } from '@angular/core';
import { HttpClient } from
'@angular/common/http';@Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
  styleUrls:
```

```
export class AppComponent {
  constructor(private http:
  HttpClient) { }httpdata;
  name;
  searchparam
  = 2;ngOnInit()
  {
    this.http.get("http://jsonplaceholder.typicode.com/users?id="+this.searchparam)
    .subscribe((data) => this.displaydata(data));
  }
  displaydata(data) {this.httpdata = data;}
```

For the **get api**, we will add the search param id = this.searchparam. The searchparam is equal to 2. We needthe details of **id = 2** from the json file.

This is how the browser is displayed −



- Name : Ervin Howell Address: Wisokyburgh

We have consoled the data in the browser, which is received from the http. The same is displayed in thebrowser console. The name from the json with **id = 2** is displayed in the browser.

Example4:-

Creating Angular Project Use below Commands

- npm install -g @angular/cli //creating cli
- ng version
- ng new prog9 //creating angular project SPA(Single page application) //app component is a defaultcomponent.

- cd prog9
- ng g c header //creating header component
- ng g c home //creating home component
- ng g c profile //creating profile component
- ng serve //running angular project

Step1: header.component.css

```
ul li{
   list-style: none;
}
ul li a{
   text-decoration: none;
}
ul{
   background-color:
   aqua;height: 50px;
}
a{
   line-height:50px
   ;    font-weight:
   bold;       font-
   size:20px;
}
```

Step 2:- Create navbar in **header.component.html**

```
<ul>
   <li>
      <a routerLink="/profile">Profile</a>
   </li>
</ul>
```

Step 3:- Create navbar in **home.component.html**

```
<h1>Welcome to Home Page</h1>
```

Step 4:-Configure route links in **app.module.ts**

```
import { ProfileComponent } from

'./profile/profile.component';import { HeaderComponent }

from './header/header.component';import {

HomeComponent } from './home/home.component'; import

{ RouterModule,Routes } from '@angular/router';

import { HttpClientModule } from

'@angular/common/http';const routes:Routes=[
 {
   path:'',component:HomeComponent
 },
 {
   path:'profile',component:ProfileComponent
 },
]
```

```
imports:
   [RouterModule.forRoot(routes),
   HttpClientModule]
```

Step 5:-Use header selector in the **app.component.html** along with <router-outlet>

```
<app-header></app-header>
<router-outlet></router-outlet>
```

Step6:Profile.component.css

```
img {
   border-radius: 50%;
}
```

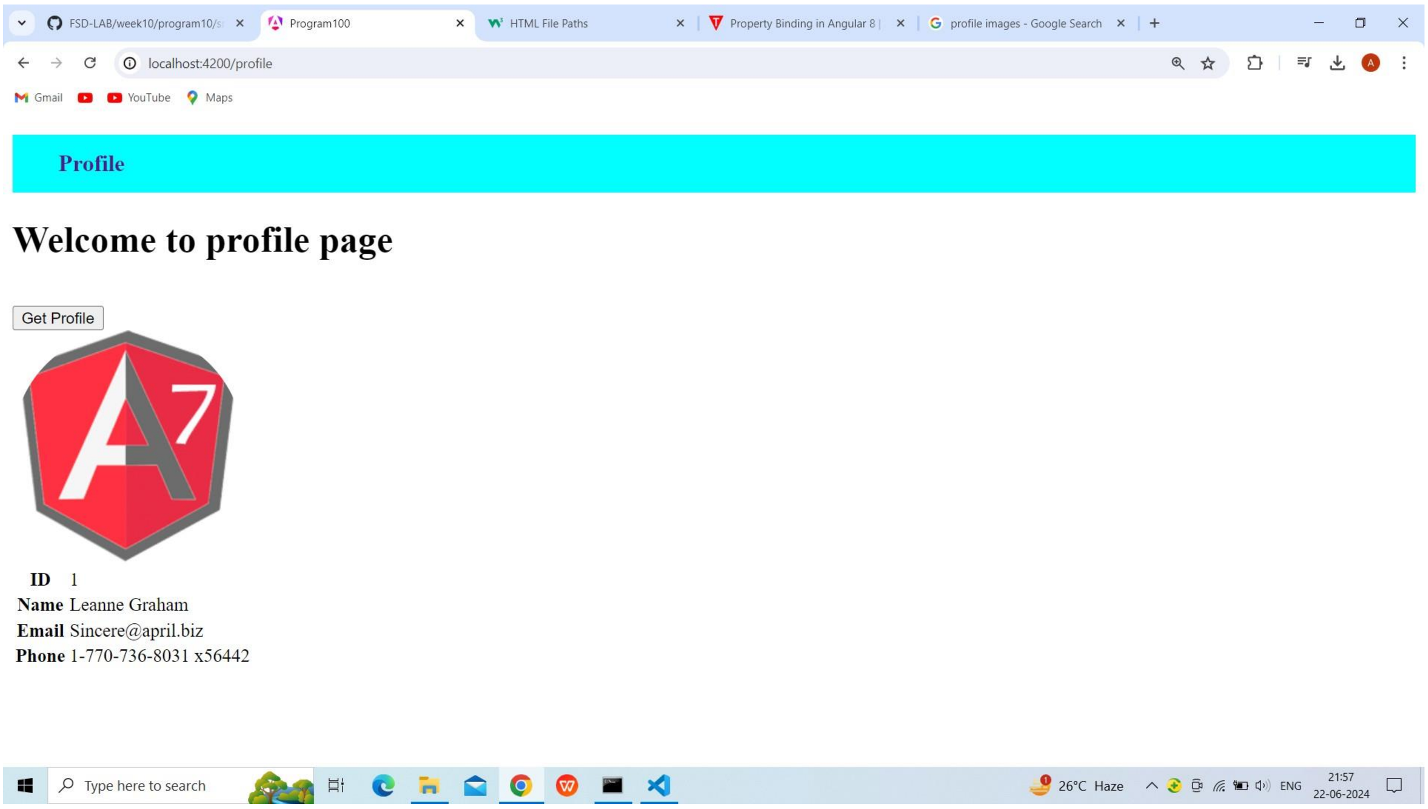
Step7:Profile.component.html

```
<h1>Welcome to profile page</h1>
 <br>
 <button (click)="getData()">Get Profile</button>
 <br>
 <div *ngIf="data">
    <img [src]="ImagePath"  alt="Profile">
 <table>

        <tr><th>ID</th>
        <td>{{data.id}}</td></tr>

        <tr><th>Name</th>
        <td>{{data.name}}</td></tr>

        <tr><th>Email</th>
        <td>{{data.email}}</td></tr>

        <tr><th>Phone</th>
        <td>{{data.phone}}</td></tr>


 </table>
 </div>
```

Step8:Profile.component.ts

```
import { Component } from '@angular/core';
import { HttpClient} from '@angular/common/http';

@Component({
 selector: 'app-
 profile',
 templateUrl:
  './profile.component.html',styleUrl:
  './profile.component.css'
})
export class ProfileCompnt { ImagePath:any;

constructor(private http:HttpClient){
  this.ImagePath = 'https://static.javatpoint.com/tutorial/angular7/images/angular-7-logo.png';
}
data:any;
getData()
{
  this.http.get('https://jsonplaceholder.typicode.com/users/1')
  .subscribe((data)=
>{this.data=data;
  })
}
}
```

Output:



Profile

Welcome to profile page

Get Profile

ID 1
Name Leanne Graham
Email Sincere@april.biz
Phone 1-770-736-8031 x56442

8.Observables:-

Defination:-

Observables provide support for data sharing between publishers and subscribers in an angular application. Itis referred to as a better technique for event handling, asynchronous programming, and handling multiple values as compared to techniques like promises.
A special feature of Observables is that it can only be accessed by a consumer who subscribes to it i.e A function for publishing values is defined, but it is not executed by the subscribed consumer (it can be anycomponent) only via which the customer can receive notifications till the function runs or till they subscribed.
An observable can deliver multiple values of any type. The API for receiving values is the same in any condition and the setup and the logic are both handled by the observable. Rest thing is only about subscribingand unsubscribing the information required.
Observers: To handle receiving observable messages, we need an observable interface which consists ofcallback methods with respect to the messages by observables.

Usage

The basic usage of Observable in Angular is to create an instance to define a **subscriber function**. Whenever a consumer wants to execute the function the **subscribe()** method is called. This function defines how to obtain messages and values to be published.

To make use of the observable, all you need to do is to begin by creating notifications using subscribe() method, and this is done by passing observer as discussed previously. The notifications are generally Javascript objects that handle all the received notifications. Also, the unsubscribe() method comes along with subscribing () methodso that you can stop receiving notifications at any point in time.

Types of Notifications and Description

1. **next**: It is called after the execution starts for zero times or more than that. It is a mandatory notificationfor catching each value delivered.
2. **error:** It is a handler for each error message. An error stops execution of the observable instance.
3. **complete:** It is a handles in which the completion of observable execution is notified.

Before using Observables do import Observables from rxjs library by writing the following code.

```
import {Observables} from 'rxjs'
```

Error Handling:
Observables produce asynchronous values and thus try/catch do not catch any errors because it may lead to stop the code irrespective of other tasks running at that instance of time. Instead, we handle errors by specifying an error callback on the observer. When an error is produced, it causes the observable to clean upsubscriptions and stop producing values for that subscription. An observable can either produce values (calling the next callback), or it can complete, calling either the complete or error callback.

The syntax for error callback

```
observable.subscribe({
   next(val) { console.log('Next: ' +
   val)},error(err) {
   console.log('Error: ' + err)}
```

Example:-

app.component.html:-

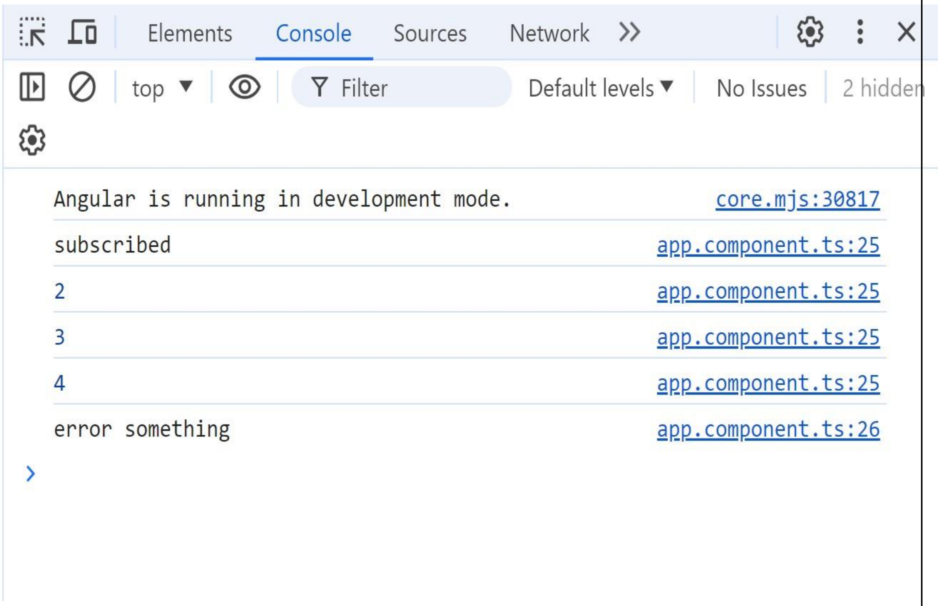
```
<button (click)="test()">get</button>
<button (click)="lose()">lose</button>
```

app.component.ts:-

```
import { Component } from
'@angular/core';import { Observable }
from 'rxjs'; @Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {


  myobs=new Observable(
    (listener)=>{
    listener.next("subscribed");
    listener.next(2);
      setTimeout(()=>listener.next(3),1000);
      setTimeout(()=>listener.next(4),1000);
      setTimeout(()=>listener.error("error
      something"),1000);
      setTimeout(()=>listener.next(6),1000);
      //setTimeout(()=>listener.complete(),1000);
    }
  )
  aaa:an
  y;
  test(){
    this.aaa=this.myobs.subscribe
      ( (data)=>{console.log(data)},
      err=>{console.log(err)},
      ()=>{console.log("completed")
      }
    )
  }
```

```
 lose(){
   this.aaa.unsubscri
   be();
 }

}
```

Output:-



```
Angular is running in development mode.          core.mjs:30817
subscribed                                  app.component.ts:25
2                                           app.component.ts:25
3                                           app.component.ts:25
4                                           app.component.ts:25
error something                             app.component.ts:26
```

More Information:-

Observables in Angular?

We use Observable to perform asynchronous operations and handle asynchronous data. Another way of handlingasynchronous is using promises. We can handle asynchronous operations using either Promises or Observables.

What are asynchronous operations and asynchronous data?

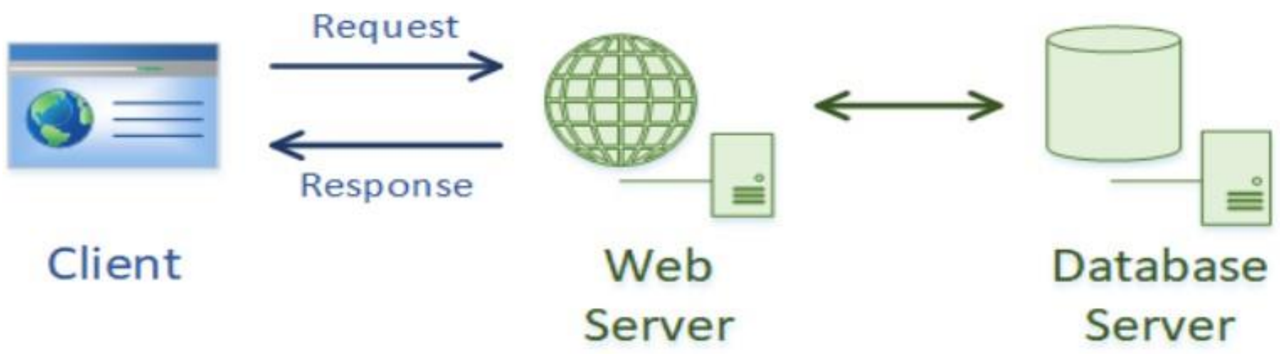
We already know that JavaScript is a single-threaded language. That means the code is executed line by line andonce the execution of one code is complete then only the next code of the program will be executed. When we make a request to the HTTP server that will take more time.So the next statement after the HTTP request has to wait for the execution. It will only get executed when the HTTP request completes. We can say that the synchronized code is blocked in nature.

This is the way the asynchronous programs came into the picture. Asynchronous code executing in thebackground without blocking the execution of the code in the main thread. Asynchronous code is non-blocking.That means we can make HTTP requests asynchronously.

Using an asynchronous program we can perform long network requests without blocking the main thread. Thereare 2 ways in which we can do that.

- Using Observables
- Using Promises

What is the difference between Promises and Observables?



Let’s say we are requesting a list of users from the server. From the browser, we are sending a request to the server and the server will get the data from the database. Let’s say the data which we are requesting is huge. In that case, the server will get some time to get the data from the database.

Once the data is ready the data will send from the server to the client-side. Here server gathered all the data and when the data is ready that will send back to the client-side. This is how gets the Promise work. It promises to provide data over a period of time. Promise provides us the data once the complete data is ready. The data can bethe actual data that we requested or it can also be an error. If there is no internet connection. In that case, also promises to return some data. That data will be the error message or an error object.

Observables are not waiting for the complete data to be available. An Observable streams the data. When the datais available partially it will send to the client.

Promises

1. Helps you run functions asynchronously, and use their return values (or exceptions), but only once whenexecuted.

2. Not lazy.

3. Not cancellable (there are Promise libraries out there that support cancellation,

but ES6Promise doesn’t so far). The two possible decisions are Reject and

Resolve.

4. Cannot be retried (Promises should have access to the original function that

returned thepromise to have a retry capability, which is a bad practice)

5. Provided by JavaScript language.

Observables

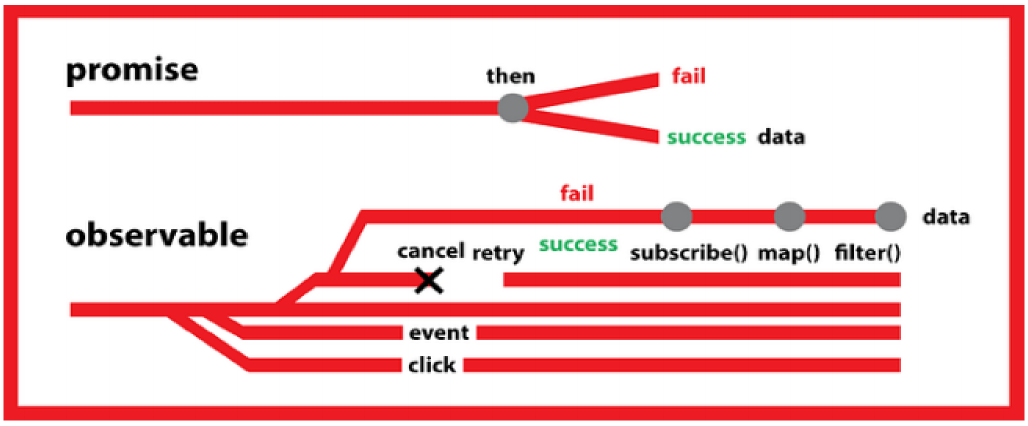
1. Helps you run functions asynchronously, and use their return values in a continuous sequence (multiple times)when executed.

2. By default, it is lazy as it emits values when time progresses.

3. Has a lot of operators which simplifies the coding effort.

4. One operator retry can be used to retry whenever needed, also if we need to retry the observable based on someconditions retryWhen can be used.

5. Not a native feature of Angular or JavaScript. Provide by another JavaScript library which is called Rxjs.



An Observable is a function that converts the ordinary stream of data into an Observable stream of data. You canthink of Observable as a wrapper around the Ordinary stream of data.

9. Routing

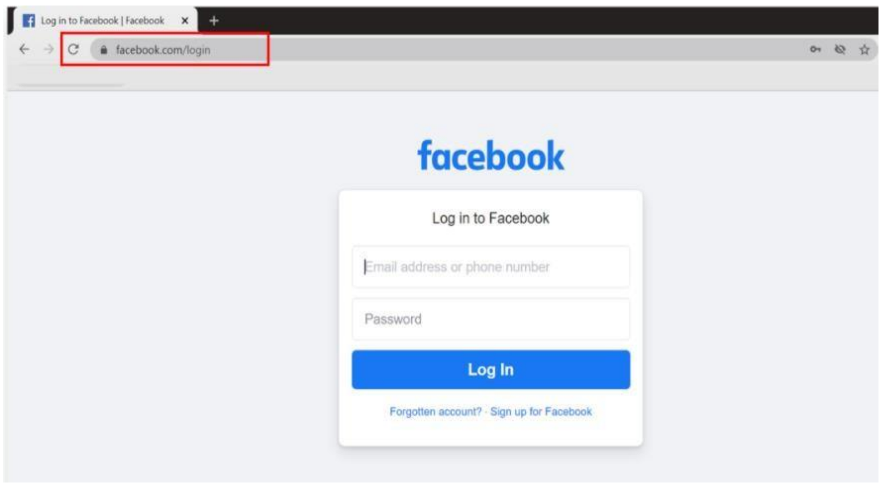
The ngRoute module helps your application to become a Single Page Application.

What is Routing in AngularJS?

If you want to navigate to different pages in your application, but you also want the application to be a SPA(Single Page Application), with no page reloading, you can use the ngRoute module.
The ngRoute module *routes* your application to different pages without reloading the entire application.

In Angular, routing plays a vital role since it is essentially used to create Single Page Applications. These applications are loaded just once, and new content is added dynamically. Applications like Google, Facebook, Twitter, and Gmail are a few examples of SPA. The best advantage of SPA is that they provide an excellent user experience and you don't have to wait for pages to load, and by extension, this makes the SPA fast and gives a desktop-like feel.

It generally specifies navigation with a forward slash followed by the path defining the new content.


Log in to Facebook | Facebook
facebook.com/login
facebook
Log in to Facebook
Email address or phone number
Password
Log In
Forgotten account? · Sign up for Facebook

Example:-

Note:-Install Node Js Software and Visual Studio.

Creating Angular Project Use below Commands

- npm install –g @angular/cli //creating cli
- ng version
- ng new prog9 --standalone false //creating angular project SPA(Single page application)
  //app component is a default component.
- cd prog9

- ng g c header //creating header component
- ng g c home //creating home component
- ng g c profile //creating profile component
- ng serve //running angular project

Creating body of about, contact, home and header. Header is a navbar this page creating routerLinks aboutand contact. Home is a default link when header loaded it is displayed. App.module.ts file creating url pathsfor each page.

1) **about.component.html:-**

```
<h1>This is About Component</h1>
```

2) **contact.component.html:-**

```
<h1>This is Contact Component</h1>
```

3) **header.component.css:-**

```
ul li{
   list-style: none;
}
ul li a{
   text-decoration: none;
}
ul{
   display: flex;
   justify-content: flex-
   start;gap: 20px;
   background-color:
   aqua;height: 50px;
}
a{
   line-
   height:50px;
   color:black;
   margin:0 20px;
   font-weight:
   bold;font-
   size:30px;
}
```

4) **header.component.html:-**

```
<ul>
   <li>
      <a routerLink="/about" router="active">about</a>
   </li>
   <li>
      <a routerLink="/contact" routerActiveLink="active">contact</a>
   </li>
</ul>
```

5) home.component.html:-

```
<h1>This is Home Component</h1>
```

6) **notfound.component.html:-**

```
<p>notfound works!</p>
```

7) **app.component.html:-**

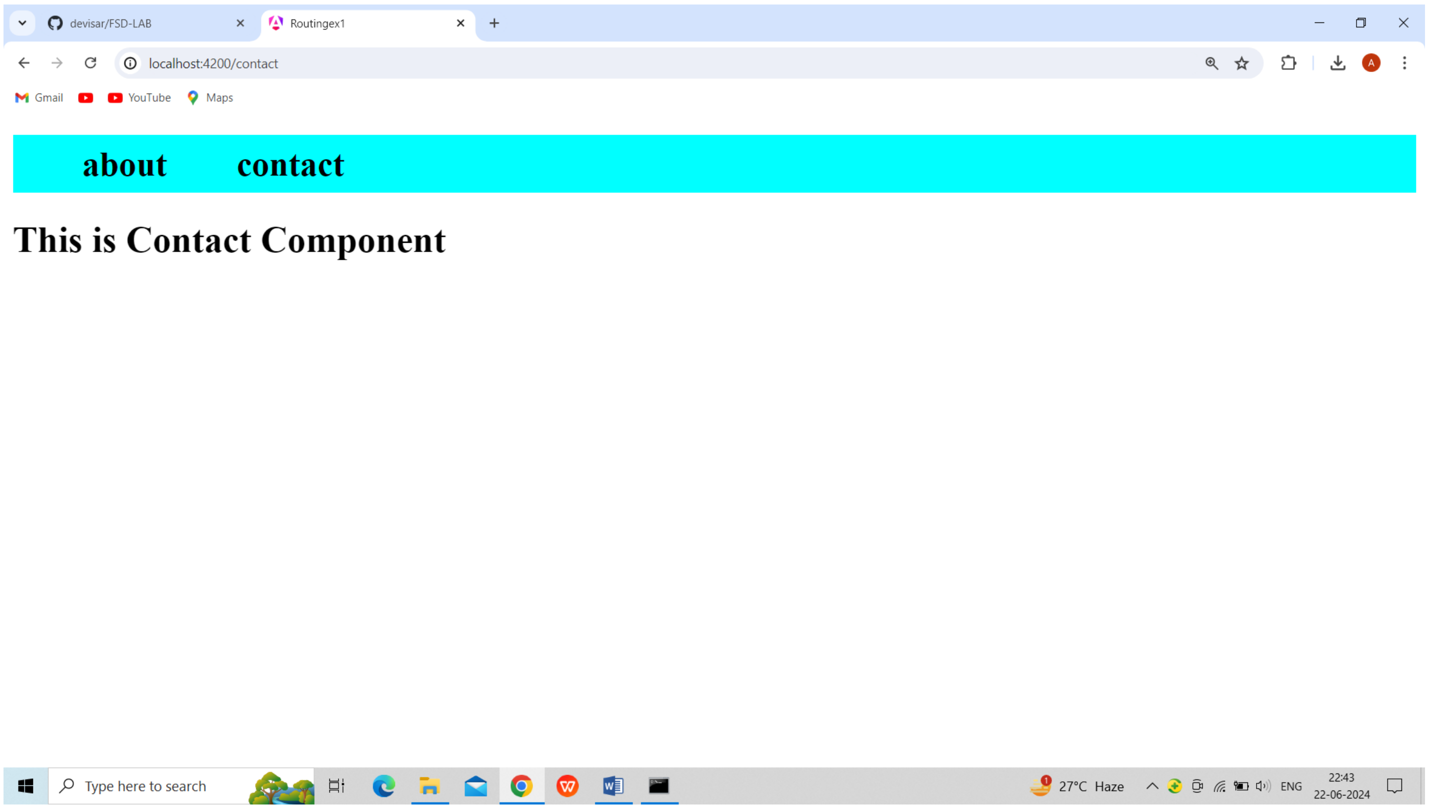
```
<app-header></app-header>
<router-outlet></router-outlet>
```

8) **app.module.ts:-**

```
import { AppComponent } from './app.component';
import { HeaderComponent } from
```

```
'./header/header.component'; import {
AboutComponent } from './about/about.component';
import { ContactComponent } from
'./contact/contact.component';import {
HomeComponent } from './home/home.component';
import { NotfoundComponent } from
'./notfound/notfound.component';import {
RouterModule,Routes } from '@angular/router';

const routes:Routes=[
 {
   path:'',component:HomeComponent
 },
 {
   path:'about',component:AboutComponent
 },
 {
   path:'contact',component:ContactComponent
 },
 {
   path:'**',component:NotfoundComponent
 }
]

imports: [

   RouterModule.forRoot(routes)
 ],
```

Output:



about contact

This is Contact Component

UNIT-4

Your First Node API

Hello Node.js

Node.js was first released in 2009 by Ryan Dahl as a reaction to how slow web servers were at the time. Most web servers would block for any I/O task[4], such as reading from the file system or accessing the network, and this would dramatically lower their throughput. For example, if a server was receiving a file upload, it would not be able to handle any other request until the upload was finished.

At that time, Dahl mostly worked with Ruby, and the dominant model for web applications was to have a pool of ruby processes that a web server (e.g. Ngninx) would proxy to. If one Ruby process was blocked with an upload, Nginx served the request to another.

Node.js changed this model by making all I/O tasks non-blocking and asynchronous. This allowed web servers written in Node.js to serve **thousands of requests concurrently** – subsequent requests didn't have to wait for previous ones to complete.

The first demo of Node.js was generated so much interest because it was the first time that a developer could create their own web server easily and have it work so well.

Over time Node.js became good at system tasks other than web serving and started to shine as a flexible yet lower level server-side language. It could do anything typically done with Python, Ruby, Perl, and PHP, and it was faster, used less memory, and in most cases had better APIs for the system calls.

For example, with Node.js we can create HTTP and TCP servers with only a few lines of code. We'll dive in and build one together soon, but just to show what we mean, here's a functioning Node.js web server in only 80 characters:

01-first-node-api/00-hello-world.js

```
require('http')
  .createServer((req, res) => res.end('hello world!'))
  .listen(8080)
```

A Rich Module Ecosystem

Node.js began to shine with the introduction of `npm`, the package manager bundled with Node.js. A core philosophy of Node.js is to have only **a small collection of built-in modules** that come preinstalled with the language.

[4]https://en.wikipedia.org/wiki/Input/output

Examples of these modules are `fs`, `http`, `tcp`, `dns`, `events`, `child_process`, and `crypto`. There's a full list in the Node.js API documentation[5].

This may seem to be a bad thing. Many people would be puzzled as why Node.js would choose not to have a large collection of standard modules preinstalled and available to the user. The reason is a bit counterintuitive, but has ultimately been very successful.

Node.js wanted to encourage a rich ecosystem of third-party modules. Any module that becomes a built-in, core module will automatically prevent competition for its features. In addition, the core module can only be updated on each release of Node.js.

This has a two-fold suppression effect on module authorship. First, for each module that becomes a core module in the standard library, many third-party modules that perform a similar feature will never be created. Second, any core modules will have development slowed by the Node.js release schedule.

This strategy has been a great success. `npm` modules have grown at an incredible pace, overtaking all other package managers. In fact, **one of the best things about Node.js is having access to a gigantic number of modules**.

When To Use Node.js

Node.js is a great choice for any task or project where one would typically use a dynamic language like Python, PHP, Perl, or Ruby. Node.js particularly shines when used for:

- HTTP APIs,
- distributed systems,
- command-line tools, and
- cross-platform desktop applications.

Node.js was created to be a great web server and it does not disappoint. In the next section, we'll see how easy it is to build an HTTP API with the built-in core `http` module.

Web servers and HTTP APIs built with Node.js generally have much higher performance than other dynamic languages like Python, PHP, Perl, and Ruby. This is partly because of its non-blocking nature, and partly because the Node.js V8 JavaScript interpreter is so well optimized.

There are many popular web and API frameworks built with Node.js such as `express`[6], `hapi`[7], and `restify`[8].

Distributed systems are also very easy to build with Node.js. The core `tcp` module makes it very easy to communicate over the network, and useful abstractions like streams allow us to build systems using composable modules like `dnode`[9].

[5]https://nodejs.org/api/index.html
[6]https://expressjs.com
[7]https://hapijs.com
[8]https://restify.com
[9]https://www.npmjs.com/package/dnode

Command-line tools can make a developer's life much easier. Before Node.js, there wasn't a good way to create CLIs with JavaScript. If you're most comfortable with JavaScript, Node.js will be the best way to build these programs. In addition, there are tons of Node.js modules like `yargs`[10], `chalk`[11], and `blessed`[12] that make writing CLIs a breeze.

`Electron`[13], allows us to build cross-platform desktop applications using JavaScript, HTML, and CSS. It combines a browser GUI with Node.js. Using Node.js we're able to access the filesystem, network, and other operating system resources. There's a good chance you use a number of Electron apps regularly.

When Node.js May Not Be The Best Choice

Node.js is a dynamic, interpreted language. It is very fast compared to other dynamic languages thanks to the V8 JIT compiler. However, if you are looking for a language that can squeeze the most performance out of your computing resources, Node.js is not the best.

CPU-bound workloads can typically benefit from using a lower-level language like C, C++, Go, Java, or Rust. As an extreme example, when generating fibonacci numbers[14] Rust and C are about three times faster than Node.js. If you have a specialized task that is particularly sensitive to performance, and does not need to be actively developed and maintained, consider using a lower-level level language.

Certain specialized software communities like machine learning, scientific computing, and data science have traditionally used languages other than JavaScript. Over time they have created many packages, code examples, tutorials, and books using languages like Python, R, and Java that either do not exist in JavaScript, are not at the same level of maturity, or do not have the same level of optimization and performance. Node.js might become more popular for these tasks in the future as more flagship projects like `TensorFlow.js`[15] are developed. However, at this current time, fewer people in these disciplines have much Node.js experience.

Front-end Vs. Back-end JavaScript

If you're more familiar with using JavaScript in the browser than you are with using it in Node.js, there a few differences worth paying attention to.

The biggest difference between Node.js and running JavaScript in the browser is the lack of globals and common browser APIs. For example, `window`[16] and `document`[17] are unavailable in Node.js. Of

[10]https://yargs.js.org/
[11]https://github.com/chalk/chalk#readme
[12]https://github.com/chjj/blessed
[13]https://github.com/electron/electron#readme
[14]https://github.com/RisingStack/node-with-rust
[15]https://js.tensorflow.org/
[16]https://developer.mozilla.org/en-US/docs/Web/API/Window
[17]https://developer.mozilla.org/en-US/docs/Web/API/Document

course, this should not be not surprising; Node.js does not need to maintain a DOM or other browser- related technologies to function. For a list of global objects that browsers and Node.js share, see MDN's list of Standard Built-in Objects[18].

Both Node.js and browser-based JavaScript can perform many of the same functions such as access the network or filesystem. However, the way these functions are accomplished will be different. For example, in the browser one will use the globally available `fetch()` API to create an HTTP request. In Node.js, this type of action would be done by first using `const http = require('http')` to load the built-in core `http` module, and afterwards using `http.get('http://www.fullstack.io/', function (res) { ... })`.

Diving In: Your First Node.js API

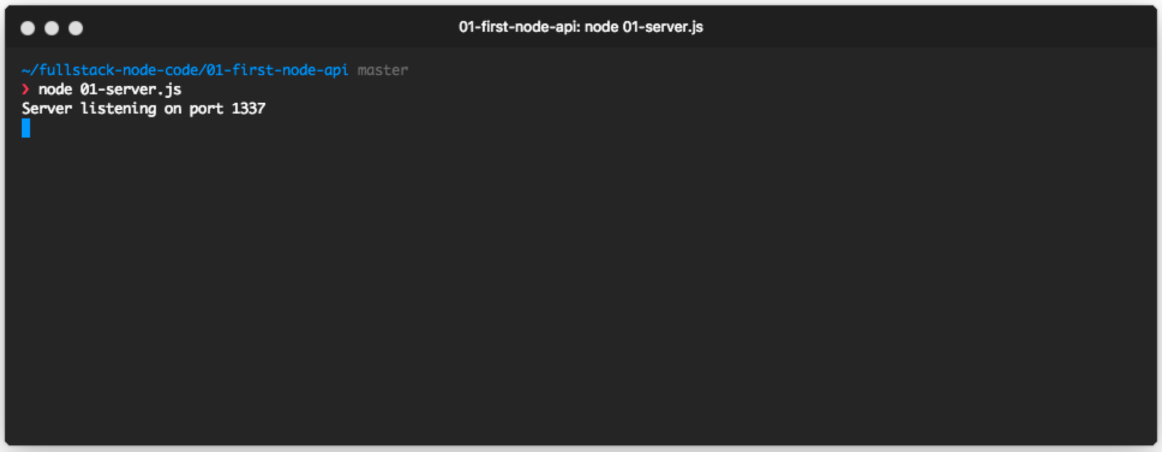
We're going to start off by creating our own web server. At first, it will be very simple; you'll be able to open your browser and see some text. What makes this impressive is just how little code is required to make this happen.

01-first-node-api/01-server.js

```
const http = require('http')

const port = process.env.PORT || 1337

const server = http.createServer(function (req, res) {
  res.end('hi')
})

server.listen(port)
console.log(`Server listening on port ${port}`)
```

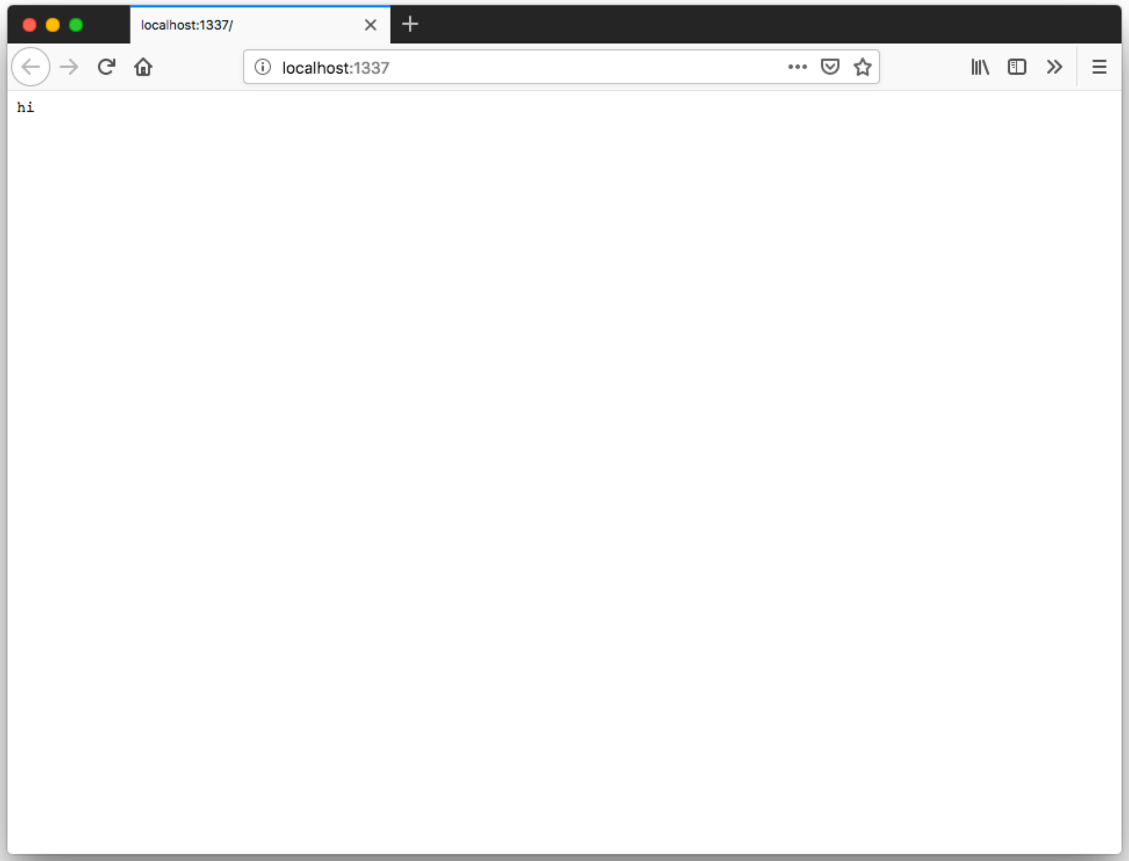
Run this file with `node 01-server.js`, and you should see `Server listening on port 1337` printed to your terminal:

[18]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects



```
~/fullstack-node-code/01-first-node-api master
› node 01-server.js
Server listening on port 1337
```

After you see that, open your browser and go to `http://localhost:1337` to see your message:



```
hi
```

Hello!

Let's look at this file line by line. First up:

01-first-node-api/01-server.js

```
const http = require('http')
```

This loads the core `http`[19] module and stores it in our `http` variable. `require()`[20] is a globally accessible function in Node.js and is always available. `http` is a core module, which means that it is always available to be loaded via `require()`. Later on we'll cover third-party modules that need to be installed before we can load them using `require()`.

[19]https://nodejs.org/api/http.html
[20]https://nodejs.org/api/modules.html#modules_require_id

01-first-node-api/01-server.js

```
const port = process.env.PORT || 1337
```

Here we choose which port our web server should listen to for requests. We store the port number in our `port` variable.

Also, we encounter a Node.js global object, `process`[21]. `process` is a global object[22] with information about the currently running process, in our case it's the process that is spawned when we run `node 01-server.js`. `process.env` is an object that contains all environment variables. If we were to run the server with `PORT=3000 node 01-server.js` instead, `process.env.PORT` would be set to `3000`. Having environment variable control over port usage is a useful convention for deployment, and we'll be starting that habit early.

01-first-node-api/01-server.js

```
const server = http.createServer(function (req, res) {
  res.end('hi')
})
```

Now we get to the real meat of the file. We use `http.createServer()`[23] to create a HTTP server object and assign it to the `server` variable. `http.createServer()` accepts a single argument: a request listener function.

Our request listener function will be called every time there's an HTTP request to our server (e.g., each time you hit `http://localhost:1337` in your browser). Every time it is called, this function will receive two arguments: a request object[24] (`req`) and a response object[25] (`res`).

For now we're going to ignore `req`, the request object. Later on we'll use it to get information about the request like url and headers.

The second argument to our request listener function is the response object, `res`. We use this object to send data back to the browser. We can both send the string `'hi'` and end the connection to the browser with a single method call: `res.end('hi')`.

At this point our server object has been created. If a browser request comes in, our request listener function will run, and we'll send data back to the browser. The only thing left to do, is to allow our server object to listen for requests on a particular port:

[21]https://nodejs.org/api/process.html#process_process
[22]https://nodejs.org/api/globals.html#globals_require
[23]https://nodejs.org/api/http.html#http_http_createserver_options_requestlistener
[24]https://nodejs.org/api/http.html#http_class_http_incomingmessage

[25]https://nodejs.org/api/http.html#http_class_http_serverresponse

01-first-node-api/01-server.js

```
server.listen(port)
```

Finally, for convenience, we print a message telling us that our server is running and which port it's listening on:

01-first-node-api/01-server.js

```
console.log(`Server listening on port ${port}`)
```

And that's all the code you need to create a high-performance web server with Node.js.

Of course this is the absolute minimum, and it's unlikely that this server would be useful in the real world. From here we'll begin to add functionality to turn this server into a usable JSON API with routing.

Serving JSON

When building web apps and distributed systems, it's common to use JSON APIs to serve data. With one small tweak, we can change our server to do this.

In our previous example, we responded with plain text:

01-first-node-api/01-server.js

```
const server = http.createServer(function (req, res) {
  res.end('hi')
})
```

In this example we're going to respond with JSON instead. To do this we're going to replace our request listener function with a new one:

01-first-node-api/02-server.js

```
const server = http.createServer(function (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
})
```

When building a production server, it's best to be explicit with responses so that clients (browsers and other consumers of our API) don't handle our data in unexpected ways (e.g. rendering an image as text). By sending plain text without a `Content-Type` header, we didn't tell the client what kind of data it should expect.

In this example, we're going to let the client know our response is JSON-formatted data by setting the `Content-Type` response header. In certain browsers this will allow the JSON data to be displayed with pretty printing and syntax highlighting. To set the `Content-Type` we use the `res.setHeader()`[26] method:

01-first-node-api/02-server.js

```
  res.setHeader('Content-Type', 'application/json')
```

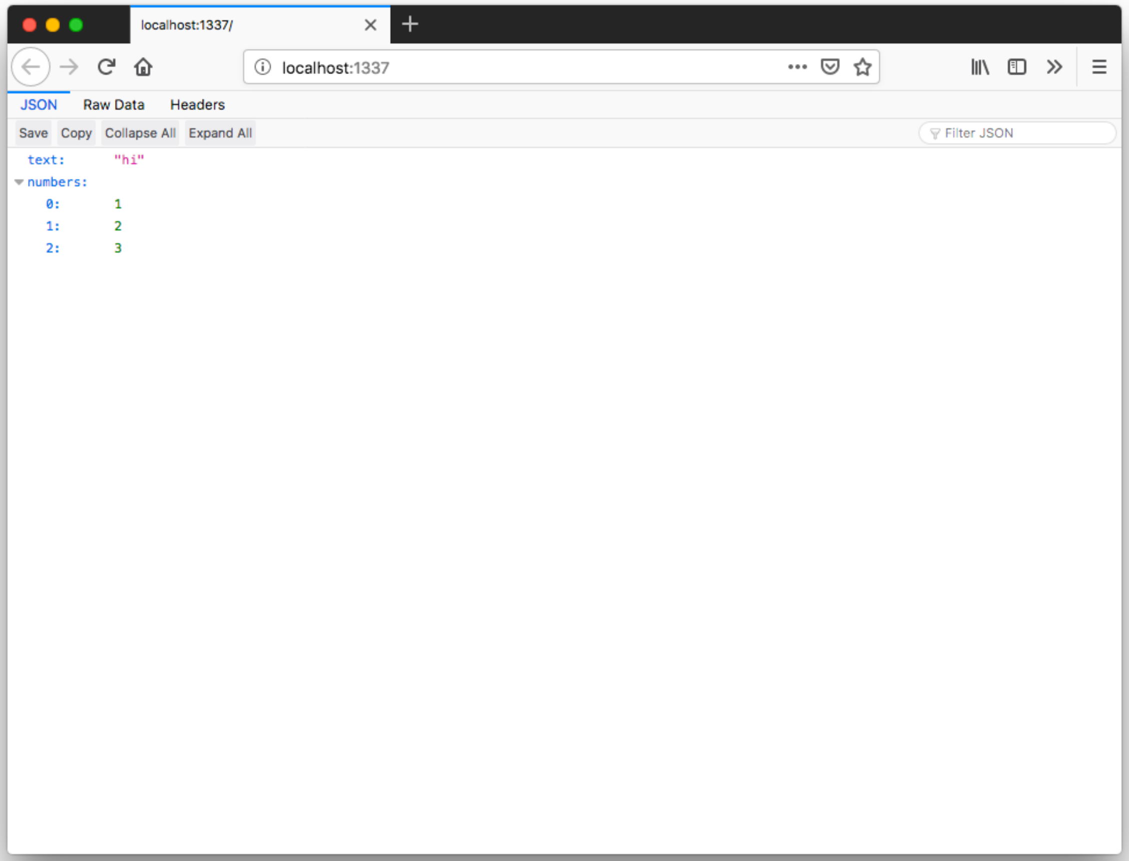
Next, we use the same method as last time to send data and close the connection. The only difference is that instead of sending plain text, we're sending a JSON-stringified object:

01-first-node-api/02-server.js

```
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
```

Run `node 02-server.js` and navigate to `http://localhost:1337` in your browser to see our new JSON response:

[26]https://nodejs.org/api/http.html#http_request_setheader_name_value



```
text:      "hi"
numbers:
  0:       1
  1:       2
  2:       3
```

What our JSON response looks like

Not all browsers will pretty-print JSON. In this screenshot I'm using Firefox, but there are several extensions available for Chrome like JSON Formatter[27] that will achieve the same result.

Of course, our API needs some work before it's useful. One particular issue is that no matter the URL path we use, our API will always return the same data. You can see this behavior by navigating to each of these URLs:

- http://localhost:1337/a
- http://localhost:1337/fullstack
- http://localhost:1337/some/long/random/url.

If we want to add functionality to our API, we should be able to handle different requests to different url paths or endpoints. For starters, we should be able to serve both our original plain text response and our new JSON response.

[27]https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa/related

Basic Routing

Not all client requests are the same, and to create a useful API, we should be able to respond differently depending on the requested url path.

We previously ignored the request object argument, `req`, and now we're going to use that to see what url path the client is requesting. Depending on the path, we can do one of three things:

- respond with plain text,
- respond with JSON, or
- respond with a 404 "Not Found" error.

We're going to change our request listener function to perform different actions depending on the value of `req.url`[28]. The `url` property of the `req` object will always contain the full path of the client request. For example, when we navigate to `http://localhost:1337` in the browser, the path is `/`, and when we navigate to `http://localhost:1337/fullstack`, the path is `/fullstack`.

We're going to change our code so that when we open `http://localhost:1337` we see our initial plain-text "hi" mesage, when we open `http://localhost:1337/json` we'll see our JSON object, and if we navigate to any other path, we'll receive a 404 "Not Found" error.

We can do this very simply by checking the value of `req.url` in our request listener function and running a different function depending on its value.

First we need to create our different functions – one for each behavior. We'll start with the functions for responding with plain-text and JSON. These new functions will use the same arguments as our request listener function and behave exactly the same as they did before:

01-first-node-api/03-server.js

```
function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
}
```

The third function will have new behavior. For this one, we'll respond with a 404 "Not Found" error. To do this, we use the `res.writeHead()`[29] method. This method will let us set both a response status code and header. We use this to respond with a 404 status and to set the `Content-Type` to `text/plain`.

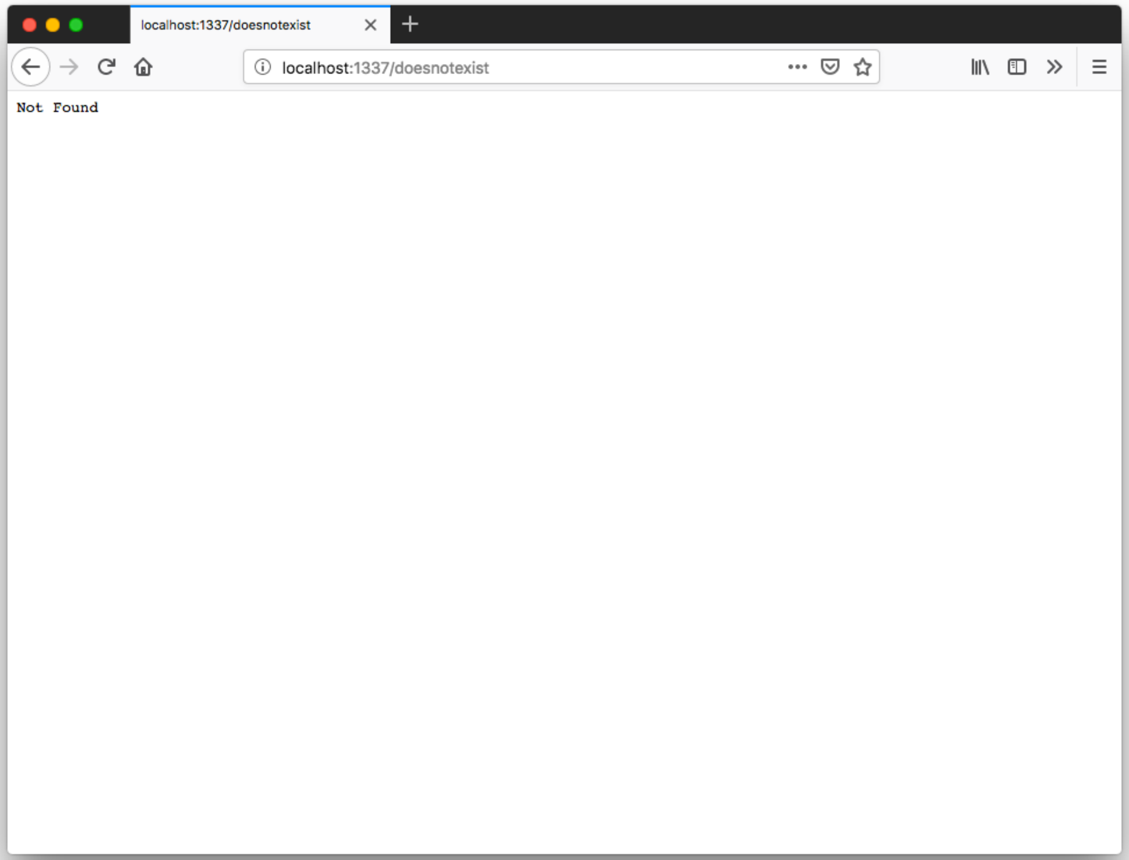
[28]https://nodejs.org/api/http.html#http_message_url
[29]https://nodejs.org/api/http.html#http_response_writehead_statuscode_statusmessage_headers

The 404 status code tells the client that the communication to the server was successful, but the server is unable to find the requested data.

After that, we simply end the response with the message `"Not Found"`:

01-first-node-api/03-server.js

```
function respondNotFound (req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' })
  res.end('Not Found')
}
```



```
Not Found
```

What our server returns for paths that don't exist

With our functions created, we can now create a request listener function that calls each one depending on the path in `req.url`:

01-first-node-api/03-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)

  respondNotFound(req, res)
})
```

Now that we have basic routing set up, we can add more functionality to our server at different endpoints.

Dynamic Responses

Currently, our endpoints respond with the same data every time. For our API to be dynamic, it needs to change its responses according to input from the client.

For apps and services in the real-world, the API will be responsible for pulling data out of a database or other resource according to specific queries sent by the client and filtered by authorization rules.

For example, the client may want the most recent comments by user `dguttman`. The API server would first look to see if that client has authorization to view the comments of `"dguttman"`, and if so, it will construct a query to the database for this data set.

To add this style of functionality to our API, we're going to add an endpoint that accepts arguments via query parameters. We'll then use the information provided by the client to create the response. Our new endpoint will be `/echo` and the client will provide input via the `input` query parameter. For example, to provide "fullstack" as input, the client will use `/echo?input=fullstack` as the url path.

Our new endpoint will respond with a JSON object with the following properties:

- normal: the input string without a transformation
- shouty: all caps
- characterCount: the number of characters in the input string
- backwards: the input string ordered in reverse

To begin, we'll first have our request listener function check to see if the `request.url` begins with `/echo`, the endpoint that we're interested in. If it is, we'll call our soon-to-be-created function `respondEcho()`:

01-first-node-api/04-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)

  respondNotFound(req, res)
})
```

Next, we create the `respondEcho()` function that will accept the request and response objects. Here's what the completed function looks like:

01-first-node-api/04-server.js

```
function respondEcho (req, res) {
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )

  res.setHeader('Content-Type', 'application/json')
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
}
```

The important thing to notice is that the first line of our function uses the `querystring.parse()`[30] method. To be able to use this, we first need to use `require()` to load the `querystring`[31] core module. Like `http`, this module is installed with Node.js and is always available. At the top of our file we'll add this line:

[30]https://nodejs.org/api/querystring.html#querystring_querystring_parse_str_sep_eq_options
[31]https://nodejs.org/api/querystring.html

01-first-node-api/04-server.js

```
const querystring = require('querystring')
```

Looking at our `respondEcho()` function again, here's how we use `querystring.parse()`:

01-first-node-api/04-server.js

```
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )
```

We expect the client to access this endpoint with a url like `/echo?input=someinput`. `querystring.parse()` accepts a raw querystring argument. It expects the format to be something like `query1=value1&query2=value2`.
The important thing to note is that `querystring.parse()` does not want the leading `?`. Using some quick string transformations we can isolate the `input=someinput` part of the url, and pass that in as our argument.

`querystring.parse()` will return a simple JavaScript object with query param key and value pairs. For example, `{ input: 'someinput' }`. Currently, we're only interested in the `input` key, so that's the only value that we'll store. If the client doesn't provide an `input` parameter, we'll set a default value of `''`.

Next up, we set the appropriate `Content-Type` header for JSON like we have before:

01-first-node-api/04-server.js

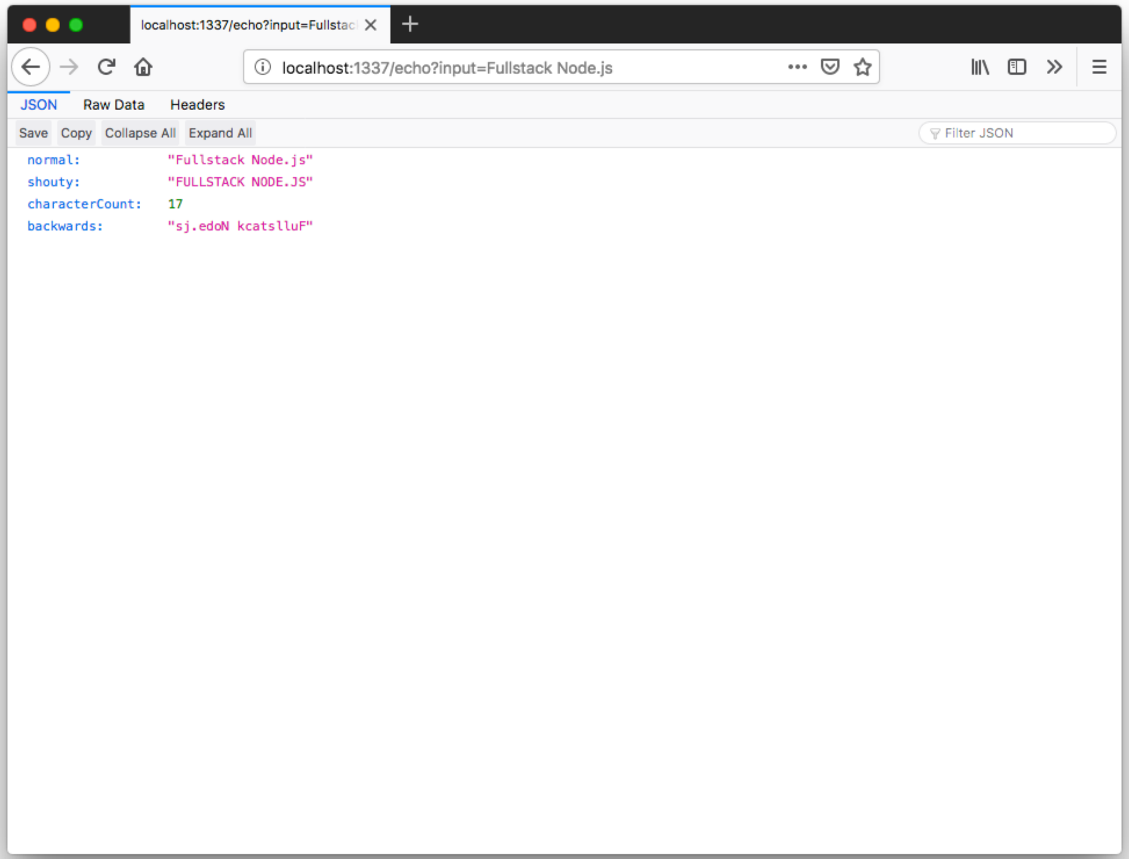
```
  res.setHeader('Content-Type', 'application/json')
```

Finally, we use `res.end()` to send data to the client and close the connection:

01-first-node-api/04-server.js

```
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
```

And there we have it, our first dynamic route! While simple, this gives us a good foundation for being able to use client-provided input for use in our API endpoints.


localhost:1337/echo?input=Fullstac
localhost:1337/echo?input=Fullstack Node.js
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
normal:         "Fullstack Node.js"
shouty:         "FULLSTACK NODE.JS"
characterCount: 17
backwards:      "sj.edoN kcatslluF"
```

Our server can now respond dynamically to different inputs

File Serving

One of the most common uses of a web server is to serve html and other static files. Let's take a look at how we can serve up a directory of files with Node.js.

What we want to do is to create a local directory and serve all files in that directory to the browser. If we create a local directory called `public` and we place two files in there, `index.html` and `ember.jpg`,
we should be able to visit `http://localhost:1337/static/index.html` and `http://localhost:1337/static/ember.`

to receive them. If we were to place more files in that directory, they would behave the same way.

To get started, let’s create our new directory `public` and copy our two example files, `index.html` and `ember.jpg` into it.

The first thing we'll need to do is to create a new function for static file serving and call it when a request comes in with an appropriate `req.url` property. To do this we'll add a fourth conditional to our request listener that checks for paths that begin with `/static` and calls `respondStatic()`, the function we'll create next:

01-first-node-api/05-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)
  if (req.url.match(/^\/static/)) return respondStatic(req, res)

  respondNotFound(req, res)
})
```

And here's the `respondStatic()` function we need:

01-first-node-api/05-server.js

```
function respondStatic (req, res) {
  const filename = `${__dirname}/public${req.url.split('/static')[1]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}
```

The first line is fairly straightforward. We perform a simple conversion so that we can translate the incoming `req.url` path to an equivalent file in our local directory. For example, if the `req.url` is
`/static/index.html`, this conversion will translate it to `public/index.html`.

Next, once we have our `filename` from the first line, we want to open that file and send it to the browser. However, before we can do that, we need to use a module that will allow us to interact with the filesystem. Just like `http` and `querystring`, `fs` is a core module that we can load with `require()`. We make sure that this line is at the top of our file:

01-first-node-api/05-server.js

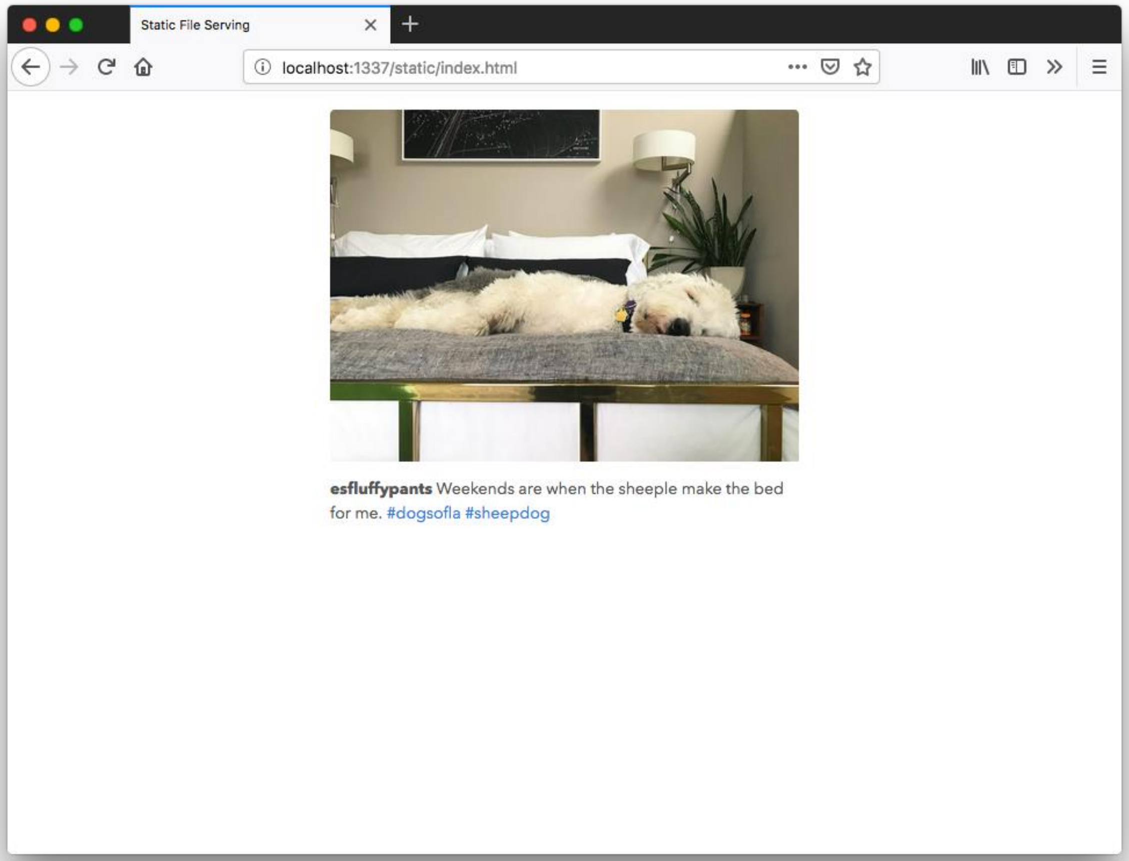
```
const fs = require('fs')
```

We use the `fs.createReadStream()` method to create a `Stream` object representing our chosen file. We then use that stream object's `pipe()` method to connect it to the response object. Behind the scenes, this efficiently loads data from the filesystem and sends it to the client via the response object.

We also use the stream object's `on()` method to listen for an error. When any error occurs when reading a file (e.g. we try to read a file that doesn't exist), we send the client our "Not Found" response.

If this doesn't make much sense yet, don't worry. We'll cover streams in more depth in later chapters. The important thing to know is that they're a very useful tool within Node.js, and they allow us to quickly connect data sources (like files) to data destinations (client connections).

Now run `node 05-server.js` and visit `http://localhost:1337/static/index.html` in your browser. You should see the `index.html` page from `/public` load in your browser. Additionally, you should notice that the image on this page *also* comes from the `/public` directory.


Static File Serving
localhost:1337/static/index.html
esfluffypants Weekends are when the sheeple make the bed for me. #dogsofla #sheepdog

Now serving static files. Notice how the path `/static` is mapped to the local directory `/public`

Here's what the full `05-server.js` file looks like:

01-first-node-api/05-server.js

```
const fs = require('fs')
const http = require('http')
const querystring = require('querystring')

const port = process.env.PORT || 1337

const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)
  if (req.url.match(/^\/static/)) return respondStatic(req, res)

  respondNotFound(req, res)
})

server.listen(port)
console.log(`Server listening on port ${port}`)

function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
}

function respondEcho (req, res) {
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )

  res.setHeader('Content-Type', 'application/json')
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
}

function respondStatic (req, res) {
  const filename = `${__dirname}/public${req.url.split('/static')[1]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}

function respondNotFound (req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' })
  res.end('Not Found')
}
```

Up until this point we've only used core modules that come built into Node.js. Hopefully we've shown how easy it is to create APIs and static file servers with the core modules of Node.js.

However, one of the best things about Node.js is its incredible ecosystem of third-party modules available on npm (over 842,000 as I write this). Next we're going to show how our server would look using `express`, the most popular web framework for Node.js.

Express

`express` is a "fast, unopinionated, minimalist web framework for Node.js" used in production environments the world over. `express` is so popular that many people have never tried to use Node.js *without* `express`.

`express` is a drop-in replacement for the core `http` module. The biggest difference between using `express` and core `http` is routing, because unlike core `http`, `express` comes with a built-in router.

In our previous examples with core `http`, we handled our own routing by using conditionals on the value of `req.url`. Depending on the value of `req.url` we execute a different function to return the appropriate response.

In our `/echo` and `/static` endpoints we go a bit further. We pull additional information out of the path that we use in our functions.

In `respondEcho()` we use the `querystring` module get key-value pairs from the search query in the url. This allows us to get the input string that we use as the basis for our response.

In `respondStatic()` anything after `/static/` we interpret as a file name that we pass to `fs.createReadStream()`. By switching to `express` we can take advantage of its router which simplifies both of these use-cases.

Let's take a look at how the beginning of our server file changes:

01-first-node-api/06-server.js

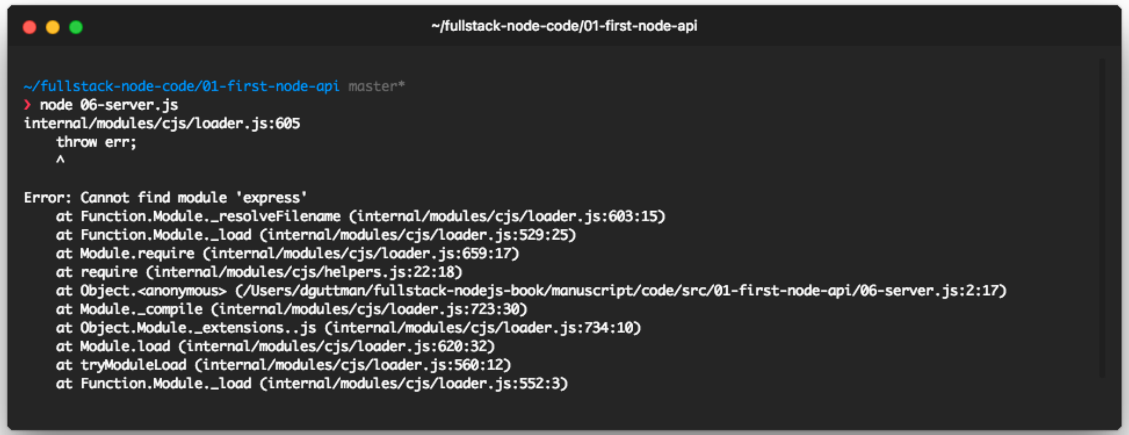
```
const fs = require('fs')
const express = require('express')

const port = process.env.PORT || 1337

const app = express()

app.get('/', respondText)
app.get('/json', respondJson)
app.get('/echo', respondEcho)
app.get('/static/*', respondStatic)

app.listen(port, () => console.log(`Server listening on port ${port}`))
```

First, you'll notice that we add a `require()` for `express` on line 2:

01-first-node-api/06-server.js

```
const express = require('express')
```

Unlike our previous examples, this is a third-party module, and it does not come with our installation of Node.js. If we were to run this file before installing `express`, we would see a `Cannot find module 'express'` error like this:



```
~/fullstack-node-code/01-first-node-api

~/fullstack-node-code/01-first-node-api master*
❯ node 06-server.js
internal/modules/cjs/loader.js:605
    throw err;
    ^

Error: Cannot find module 'express'
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:603:15)
    at Function.Module._load (internal/modules/cjs/loader.js:529:25)
    at Module.require (internal/modules/cjs/loader.js:659:17)
    at require (internal/modules/cjs/helpers.js:22:18)
    at Object.<anonymous> (/Users/dguttman/fullstack-nodejs-book/manuscript/code/src/01-first-node-api/06-server.js:2:17)
    at Module._compile (internal/modules/cjs/loader.js:723:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:734:10)
    at Module.load (internal/modules/cjs/loader.js:620:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:560:12)
    at Function.Module._load (internal/modules/cjs/loader.js:552:3)
```

What it looks like when we try to use a third-party module before it's installed

To avoid this, we need to install `express` using `npm`. When we installed Node.js, `npm` was installed automatically as well. To install `express`, all we need to do is to run `npm install express` from our application directory. `npm` will go fetch `express` from its repository and place the `express` module in a folder called `node_modules` in your current directory. If `node_modules` does not exist, it will be created. When we run a JavaScript file with Node.js, Node.js will look for modules in the `node_- modules` folder.

Node.js will also look in other places for modules such as parent directories and the global `npm`
installation directory, but we don't need to worry about those for right now.

The next thing to notice is that we no longer use `http.createServer()` and pass it a request listener function. Instead we create a server instance with `express()`. By `express` convention we call this `app`.

Once we've created our server instance, `app`, we take advantage of `express` routing. By us- ing `app.get()` we can associate a path with an endpoint function. For example, `app.get('/', respondText)` makes it so that when we receive an HTTP GET request to '/', the `respondText()` function will run.

This is similar to our previous example where we run `respondText()` when `req.url === '/'`. One difference in functionality is that `express` will only run `respondText()` if the method of the request is `GET`. In our previous examples, we were not specific about which types of methods we would respond to, and therefore we would have also responded the same way to `POST`, `PUT`, `DELETE`, `OPTIONS`, or `PATCH` requests.

Under the covers, `express` is using core `http`. This means that if you understand core `http`, you'll have an easy time with `express`. For example, we don't need to change our `respondText()` function at all:

01-first-node-api/06-server.js

```
function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}
```

However, the nice thing about `express` is that it gives us a lot of helpers and niceties that can make our code more succinct. For example, because responding with JSON is so common, `express` adds a `json()` function to the response object. By calling `res.json()`, `express` will automatically send the correct `Content-Type` header and stringify our response body for us. Here's our `respondJson()` function updated for `express`:

01-first-node-api/06-server.js

```
function respondJson (req, res) {
  res.json({ text: 'hi', numbers: [1, 2, 3] })
}
```

Another difference with `express` routing is that we don't have to worry about search query parameters when defining our routes. In our previous example we used a regular expression to check that `req.url` string started with `/echo` instead of checking for equality (like we did with `/` and `/json`). We did this because we wanted to allow for the user-created query parameters. With `express`, `app.get('/echo', respondEcho)` will call `respondEcho()` for any value of search query parameters - even if they are missing.

Additionally, `express` will automatically parse search query parameters (e.g. `?input=hi`) and make them available for us as an object (e.g. `{ input: 'hi' }`). These parameters can be accessed via `req.query`. Here's an updated version of `respondEcho()` that uses both `req.query` and `res.json()`:

01-first-node-api/06-server.js

```
function respondEcho (req, res) {
  const { input = '' } = req.query

  res.json({
    normal: input,
    shouty: input.toUpperCase(),
    characterCount: input.length,
    backwards: input
      .split('')
      .reverse()
      .join('')
  })
}
```

The last thing to notice about `express` routing is that we can add regex wildcards like `*` to our routes:

01-first-node-api/06-server.js

```
app.get('/static/*', respondStatic)
```

This means that our server will call `respondStatic()` for any path that begins with `/static/`. What makes this particularly helpful is that `express` will make the wildcard match available on the request object. Later we'll be able to use `req.params` to get the filenames for file serving. For our `respondStatic()` function, we can take advantage of wildcard routing. In our `/static/*` route, the star will match anything that comes after `/static/`. That match will be available for us at `req.params[0]`.

Behind the scenes, `express` uses path-to-regexp[32] to convert route strings into regular expressions. To see how different route strings are transformed into regular expressions and how parts of the path are stored in `req.params` there's the excellent Express Route Tester[33] tool.

Here's how we can take advantage of `req.params` in our `respondStatic()` function:

01-first-node-api/06-server.js

```
function respondStatic (req, res) {
  const filename = `${__dirname}/public/${req.params[0]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}
```

With just a few changes we've converted our core `http` server to an `express` server. In the process we gained more powerful routing and cleaned up a few of our functions.

Next up, we'll continue to build on our `express` server and add some real-time functionality.

Real-Time Chat

When Node.js was young, some of the most impressive demos involved real-time communication. Because Node.js is so efficient at handling I/O, it became easy to create real-time applications like chat and games.

To show off the real-time capabilities of Node.js we're going to build a chat application. Our application will allow us to open multiple browsers where each has a chat window. Any browser can send a message, and it will appear in all the other browser windows.

Our chat messages will be sent from the browser to our server, and our server will in turn send each received message to all connected clients.

[32]https://www.npmjs.com/package/path-to-regexp
[33]http://forbeslindesay.github.io/express-route-tester/

For example, if Browser A is connected, Browser A will send a message to our server, which will in turn send the message back to Browser A. Browser A will receive the message and render it. Each time Browser A sends a message, it will go to the server and come back.

Of course, chat applications are only useful when there are multiple users, so when both Browser A and Browser B are connected, when *either* send a message to our server, our server will send that message to *both* Browser A *and* Browser B.

In this way, all connected users will be able to see all messages.

Traditionally, for a browser to receive data from the server, the browser has to make a request. This is how all of our previous examples work. For example, the browser has to make a request to `/json` to receive the JSON payload data. The server can not send that data to the browser before the request happens.

To create a chat application using this model, the browser would have to constantly make requests to the server to ask for new messages. While this would work, there is a much better way.

We're going to create a chat application where the server can push messages to the browser without the browser needing to make multiple requests. To do this, we'll use a technology called SSE (Server- Sent Events). SSE is available in most browsers through the EventSource API, and is a very simple way for the server to "push" messages to the browser.

What about websockets? Much like websockets, SSE is a good way to avoid having the browser poll for updates. SSE is much simpler than websockets and provide us the same functionality. You can read more about SSE and the EventSource API in the MDN web docs[34].

Building the App

To set this up we'll need to create a simple client app. We'll create a `chat.html` page with a little bit of HTML and JavaScript. We'll need a few things:

- JavaScript function to receive new messages from our server
- JavaScript function to send messages to the server
- HTML element to display messages
- HTML form element to enter messages

Here's what that looks like:

[34]https://developer.mozilla.org/en-US/docs/Web/API/EventSource

01-first-node-api/public/chat.html

```
<!DOCTYPE html>
<html lang="en">
  <title>Chat App</title>
  <link rel="stylesheet" href="tachyons.min.css">
  <link rel="stylesheet" href="chat.css">
  <body>
    <div id="messages">
      <h4>Chat Messages</h4>
    </div>

    <form id="form">
      <input
        id="input"
        type="text"
        placeholder="Your message...">
    </form>

    <script src='chat.js'></script>
  </body>
</html>
```

This is some basic HTML. We load some stylesheets from our public directory, create some elements to display and create messages, and finally we load some simple client-side JavaScript to handle communication with our server.

Here's what `chat.js`, our client-side JavaScript, looks like:

01-first-node-api/public/chat.js

```
new window.EventSource('/sse').onmessage = function (event) {
  window.messages.innerHTML += `<p>${event.data}</p>`
}

window.form.addEventListener('submit', function (evt) {
  evt.preventDefault()

  window.fetch(`/chat?message=${window.input.value}`)
  window.input.value = ''
})
```

We're doing two basic things here. First, when our soon-to-be-created `/sse` route sends new messages, we add them to the `div` element on the page with the `id "messages"`.

Second, we listen for when the user enters a message into the text box. We do this by adding an event listener function to our `form` element. Within this listener function we take the value of the input box `window.input.value` and send a request to our server with the message. We do this by sending a GET request to the `/chat` path with the message encoding in the query parameters. After we send the message, we clear the text box so the user can enter a new message.

While the client-side markup and code is very simple, there are two main takeaways. First, we need to create a `/chat` route that will receive chat messages from a client, and second, we need to create a `/sse` route that will send those messages to all connected clients.

Our `/chat` endpoint will be similar to our `/echo` endpoint in that we'll be taking data (a message) from the url's query parameters. Instead of looking at the `?input=` query parameter like we did in
`/echo`, we'll be looking at `?message=`.

However, there will be two important differences. First, unlike `/echo` we don't need write any data when we end our response. Our chat clients will be receiving message data from `/sse`. In this route, we can simply end the response. Because our server will send a `200` "OK" HTTP status code by default, this will act as a sufficient signal to our client that the message was received and correctly handled.

The second difference is that we will need to take the message data and put it somewhere our other route will be able to access it. To do this we're going to instantiate an object outside of our route function's scope so that when we create another route function, it will be able to access this "shared" object.

This shared object will be an instance of `EventEmitter` that we will call `chatEmitter`. We don't need to get into the details yet, but the important thing to know right now is this object will act as an information relay. The `EventEmitter` class is available through the core `events` module, and `EventEmitter` objects have an `emit(eventName[, ...args])` method that is useful for broadcasting data. When a message comes in, we will use `chatEmitter.emit()` to broadcast the message. Later, when we create the `/sse` route we can listen for these broadcasts.

For this example, it's not important to know exactly how event emitters work, but of you're curious about the details, check out the next chapter on async or the official API documentation for `EventEmitter`[35].

First, add a new route just like

before: **01-first-node-api/07-server.js**

```
app.get('/chat', respondChat)
```

And then we create the corresponding function:

[35]https://nodejs.org/api/events.html#events_class_eventemitter

01-first-node-api/07-server.js

```
function respondChat (req, res) {
  const { message } = req.query

  chatEmitter.emit('message', message)
  res.end()
}
```

There are two things to notice here. First, access the message sent from the browser in similar way to how we access `input` in the `/echo` route, but this time use the `message` property instead. Second, we're calling a function on an object that we haven't created yet: `chatEmitter.emit('message', message)`. If we were to visit `http://localhost:1337/chat?message=hi` in our browser right now, we would get an error like this:

```
ReferenceError: chatEmitter is not defined
```

To fix this, we'll need to add two lines to the top of our file. First, we require the `EventEmitter` class via the `events` module, and then we create an instance:

01-first-node-api/07-server.js

```
const EventEmitter = require('events')

const chatEmitter = new EventEmitter()
```

Now our app can receive messages, but we don't do anything with them yet. If you'd like to verify that this route is working, you can add a line that logs messages to the console. After the `chatEmitter` object is declared, add a line that listens messages like this:

```
const chatEmitter = new EventEmitter()
chatEmitter.on('message', console.log)
```

Then visit `http://localhost:1337/chat?message=hello!` in your browser and verify that the message "hello!" is logged to your terminal.

With that working, we can now add our `/sse` route that will send messages to our chat clients once they connect with the `new window.EventSource('/sse')` described above:

01-first-node-api/07-server.js

```
app.get('/sse', respondSSE)
```

And then we can add our `respondSSE()` function:

01-first-node-api/07-server.js

```
function respondSSE (req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive'
  })

  const onMessage = msg => res.write(`data: ${msg}\n\n`)
  chatEmitter.on('message', onMessage)

  res.on('close', function () {
    chatEmitter.off('message', onMessage)
  })
}
```

Let’s break this down into its three parts:

1. We establish the connection by sending a 200 OK status code, appropriate HTTP headers according to the SSE specification[36]:

01-first-node-api/07-server.js

```
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive'
  })
```

2. We listen for message events from our chatEmitter object, and when we receive them, we write them to the response body using res.write(). We use res.write() instead of res.end() because we want to keep the connection open, and we use the data format from the SSE specification[37]:

01-first-node-api/07-server.js

```
  const onMessage = msg => res.write(`data: ${msg}\n\n`)
  chatEmitter.on('message', onMessage)
```

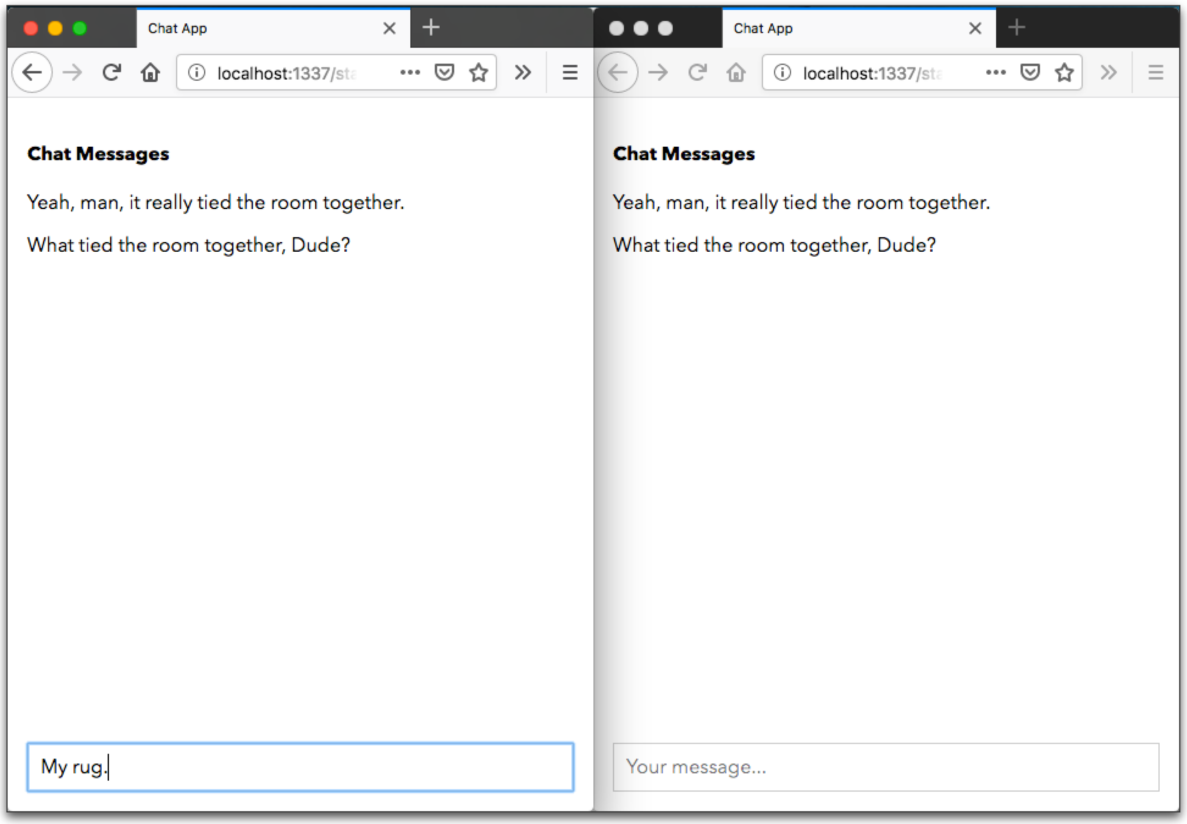
3. We listen for when the connection to the client has been closed, and when it happens we disconnect our onMessage() function from our chatEmitter object. This prevents us from writing messages to a closed connection:

[36]https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events
[37]https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

01-first-node-api/07-server.js

```
  res.on('close', function () {
    chatEmitter.off('message', onMessage)
  })
```

After adding this route and function, we now have a functioning real-time chat app. If you open `http://localhost:1337/static/chat.html` in multiple browsers you'll be able to send messages back and forth:



Chat Messages

Yeah, man, it really tied the room together.

What tied the room together, Dude?

My rug.

Chat Messages

Yeah, man, it really tied the room together.

What tied the room together, Dude?

Your message...

If you open two browsers you can talk to yourself

Here's our fully completed server file:

01- first-node-api/07-server.js

```
const fs = require('fs')
const express = require('express')
const EventEmitter = require('events')

const chatEmitter = new EventEmitter()
const port = process.env.PORT || 1337

const app = express()

app.get('/', respondText)
app.get('/json', respondJson)
app.get('/echo', respondEcho)
app.get('/static/*', respondStatic)
app.get('/chat', respondChat)
app.get('/sse', respondSSE)

app.listen(port, () => console.log(`Server listening on port ${port}`))

function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.json({ text: 'hi', numbers: [1, 2, 3] })
}

function respondEcho (req, res) {
  const { input = '' } = req.query

  res.json({
    normal: input,
    shouty: input.toUpperCase(),
    characterCount: input.length,
    backwards: input
      .split('')
      .reverse()
      .join('')
  })
}

```

```
function respondStatic (req, res) {
  const filename = `${__dirname}/public/${req.params[0]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}

function respondChat (req, res) {
  const { message } = req.query

  chatEmitter.emit('message', message)
  res.end()
}

function respondSSE (req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive'
  })

  const onMessage = msg => res.write(`data: ${msg}\n\n`)
  chatEmitter.on('message', onMessage)

  res.on('close', function () {
    chatEmitter.off('message', onMessage)
  })
}

function respondNotFound (req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' })
  res.end('Not Found')
}
```

Wrap Up

In this chapter we've shown how Node.js combines the power and performance of a low-level language with the flexibility and maintainability of a high-level language.

We've created our first API, and we learned how to send different data formats like plain-text and JSON, respond dynamically to client input, serve static files, and handle real-time communication. Not bad!

Challenges

1. Add a “chat log” feature: use fs.appendFile()[38] to write chat messages sent through the chat app to the filesystem.
2. Add a “previous messages” feature: using the chat log, use fs.readFile()[39] to send previous messages to clients when they first connect.

[38]https://nodejs.org/api/fs.html#fs_fs_appendfile_path_data_options_callback
[39]https://nodejs.org/api/fs.html#fs_fs_readfile_path_options_callback

Async

Node.js was created in reaction to slow web servers in Ruby and other dynamic languages at that time.

These servers were slow because they were only capable of handling a single request at a time. Any work that involved I/O (e.g. network or file system access) was "blocking". The program would not be able to perform any work while waiting on these blocking resources.

Node.js is able to handle many requests concurrently because it is non-blocking by default. Node.js can continue to perform work while waiting on slow resources.

The simplest, and most common form of asynchronous execution within Node.js is the callback. A callback is a way to specify that "after X happens, do Y". Typically, "X" will be some form of slow I/O (e.g. reading a file), and "Y" will be work that incorporates the result (e.g. processing data from that file).

If you're familiar with JavaScript in the browser, you'll find a similar pattern all over the place. For example:

```
window.addEventListener('resize', () => console.log('window has been resized!'))
```

To translate this back into words: "After the window is resized, print 'window has been resized!'" Here's another example: {lang=js,line-numbers=off} setTimeout(() ⇒ console.log('hello from the past'), 5000) "After 5 seconds, print 'hello from the past'" This can be confusing for most people the first time they encounter it. This is understandable because it requires thinking about multiple points in time at once.

In other languages, we expect work to be performed in the order it is written in the file. However in JavaScript, we can make the following lines print in the reverse order from how they are written:

```
const tenYears = 10 * 365 * 24 * 60 * 60 * 1000
setTimeout(() => console.log('hello from the past'), tenYears)
console.log('hello from the present')
```

If were were to run the above code, you would immediately see "hello from the present," and 10 years later, you would see "hello from the past."

Importantly, because `setTimeout()` is non-blocking, we don't need to wait 10 years to print "hello from the present" - it happens immediately after.

Let's take a closer look at this non-blocking behavior. We'll set up both an interval and a timeout. Our interval will print the running time of our script in seconds, and our timeout will print “hello from the past” after 5.5 seconds (and then exit so that we don’t count forever):

02- async/01-set-timeout.js

```
let count = 0
setInterval(() => console.log(`${++count} mississippi`), 1000)

setTimeout(() => {
  console.log('hello from the past!')
  process.exit()
}, 5500)
```

`process`[40] is a globally available object in Node.js. We don't need to use `require()` to access it. In addition to providing us the `process.exit()`[41] method, it's also useful for getting command-line arguments with `process.argv`[42] and environment variables with `process.env`[43]. We'll cover these and more in later chapters.

If we run this with `node 01-set-timeout.js` we should expect to see something like this:

```
node 01-set-timeout.js
1 mississippi
2 mississippi
3 mississippi
4 mississippi
5 mississippi
hello from the past!
```

Our script dutifully counts each second, until our timeout function executes after 5.5 seconds, printing "hello from the past!" and exiting the script.

Let's compare this to what would happen if instead of using a non-blocking `setTimeout()`, we use a blocking `setTimeoutSync()` function:

[40] https://nodejs.org/api/process.html
[41] https://nodejs.org/api/process.html#process_process_exit_code
[42] https://nodejs.org/api/process.html#process_process_argv
[43] https://nodejs.org/api/process.html#process_process_env

02-async/02-set-timeout-sync.js

```
let count = 0
setInterval(() => console.log(`${++count} mississippi`), 1000)

setTimeoutSync(5500)
console.log('hello from the past!')
process.exit()

function setTimeoutSync (ms) {
  const t0 = Date.now()
  while (Date.now() - t0 < ms) {}
}
```

We've created our own `setTimeoutSync()` function that will block execution for the specified number of milliseconds. This will behave more similarly to other blocking languages. However, if we run it, we'll see a problem:

```
node 02-set-timeout-sync.js
hello from the past!
```

What happened to our counting?

In our previous example, Node.js was able to perform two sets of instructions concurrently. While we were waiting on the "hello from the past!" message, we were seeing the seconds get counted. However, in this example, Node.js is blocked and is never able to count the seconds.

Node.js is non-blocking by default, but as we can see, it's still possible to block. Node.js is single- threaded, so long running loops like the one in `setTimeoutSync()` will prevent other work from being performed (e.g. our interval function to count the seconds). In fact, if we were to use `setTimeoutSync()` in our API server in chapter 1, our server would not be able to respond to any browser requests while that function is active!

In this example, our long-running loop is intentional, but in the future we we'll be careful not to unintentionally create blocking behavior like this. Node.js is powerful because of its ability to handle many requests concurrently, but it's unable to do that when blocked.

Of course, this works the same with JavaScript in the browser. The reason why async functions like `setTimeout()` exist is so that we don't block the execution loop and freeze the UI. Our `setTimeoutSync()` function would be equally problematic in a browser environment.

What we're really talking about here is having the ability to perform tasks on different timelines. We'll want to perform some tasks sequentially and others concurrently. Some tasks should be performed immediately, and others should be performed only after some criteria has been met in the future.

JavaScript and Node.js may seem strange because they try not to block by running everything sequentially in a single timeline. However, we'll see that this is gives us a lot of power to efficiently program tasks involving multiple timelines.

In the next sections we'll cover some different ways that Node.js allows us to do this using asynchronous execution. Callback functions like the one seen in `setTimeout()` are the most common and straightforward, but we also have other techniques. These include promises, async/await, event emitters, and streams.

Callbacks

Node.js-style callbacks are very similar to how we would perform asynchronous execution in the browser, and are just an sleight variation on our `setTimeout()` example above.

Interacting with the filesystem is extremely slow relative to interacting with system memory or the CPU. This slowness makes it conceptually similar to `setTimeout()`.

While loading a small file may only take two milliseconds to complete, that's still a really long time – enough to do over 10,000 math operations. Node.js provides us asynchronous methods to perform these tasks so that our applications can continue to perform operations while waiting on I/O and other slow tasks.

Here's what it looks like to read a file using the core `fs`[44] module:

The core `fs` module has methods that allow us to interact with the filesystem. Most often we'll use this to read and write to files, get file information such as size and modified time, and see directory listings. In fact, we've already used it in the first chapter to send static files to the browser.

02-async/03-read-file-callback.js

```
const fs = require('fs')

const filename = '03-read-file-callback.js'

fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

From this example we can see that `fs.readFile()` expects two arguments, `filename` and `callback`:

[44]https://nodejs.org/api/fs.html

```
fs.readFile(filename, callback)
```

`setTimeout()` also expects two arguments, `callback` and `delay`:

```
setTimeout(callback, delay)
```

This difference in ordering highlights an important Node.js convention. In Node.js official APIs (and most third-party libraries) the callback is always the last argument.

The second thing to notice is the order of the arguments in the callback itself:

02-async/03-read-file-callback.js

```
fs.readFile(filename, (err, fileData) => {
```

Here we provide an anonymous function as the callback to `fs.readFile()`, and our anonymous function accepts two arguments: `err` and `fileData`. This shows off another important Node.js convention: the error (or `null` if no error occurred) is the first argument when executing a provided callback.

This convention signifies the importance of error handling in Node.js. The error is the first argument because we are expected to check and handle the error first before moving on.

In this example, that means first checking to see if the error exists and if so, printing it out with `console.error()` and skipping the rest of our function by returning early. Only if `err` is falsy, do we print the filename and file size:

02-async/03-read-file-callback.js

```
fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

Run this file with `node 03-read-file-callback.js` and you should see output like:

```
node 03-read-file-callback.js
03-read-file-callback.js: 204
```

To trigger our error handling, change the filename to something that doesn't exist and you'll see something like this:

```
node 03-read-file-callback-error.js
{ [Error: ENOENT: no such file or directory, open 'does-not-exist.js']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'does-not-exist.js' }
```

If you were to comment out our line for error handling, you would see a different error: `TypeError: Cannot read property 'length' of undefined`. Unlike the error above, this would would crash our script.

TODO: `fs.readFileSync()`

We now know how basic async operations work in Node.js. If instead of reading a file, we wanted to get a directory list, it would work similarly. We would call `fs.readdir()`, and because of Node.js convention, we could guess that the first argument is the directory path and the last argument is a callback. Furthermore, we know the callback that we pass should expect `error` as the first argument, and the directory listing as the second argument.

02-async/04-read-dir-callback.js

```
fs.readdir(directoryPath, (err, fileList) => {
  if (err) return console.error(err)

  console.log(fileList)
})
```

If you're wondering why `fs.readFile()` and `fs.readdir()` are capitalized differently, it's because `readdir` is a system call[45], and `fs.readdir()` follows its C naming convention. `fs.readFile()` and most other methods are higher-level wrappers and conform to the typical JavaScript camelCase convention.

Let's now move beyond using single async methods in isolation. In a typical real-world app, we'll need to use multiple async calls together, where we use the output from one as the input for others.

Async in Series and Parallel

In the previous section, we learned how to perform asynchronous actions in series by using a callback to wait until an asynchronous action has completed. In this section, we'll not only perform asynchronous actions in series, but we will also perform a group of actions in parallel.

Now that we know how to read files and directories. Let's combine these to so that we can first get a directory list, and then read each file on that list. In short, our program will:

[45]http://man7.org/linux/man-pages/man3/readdir.3.html

1. Get a directory list.
2. For each item on that list, print the file's name and size (in the same alphabetical order as the list).
3. Once all names and sizes have been printed, print "done!"

We might be tempted to write something like this:

02-async/05-read-dir-callbacks-broken.js

```
const fs = require('fs')

fs.readdir('./', (err, files) => {
  if (err) return console.error(err)

  files.forEach(function (file) {
    fs.readFile(file, (err, fileData) => {
      if (err) return console.error(err)

      console.log(`${file}: ${fileData.length}`)
    })
  })

  console.log('done!')
})
```

If we run `node 05-read-dir-callbacks-broken.js`, we'll see some problems with this approach. Let's look at the output:

```
node 05-read-dir-callbacks-broken.js
done!
01-set-timeout.js: 161
02-set-timeout-sync.js: 242
04-read-dir-callback.js: 166
05-read-dir-callbacks-broken.js: 306
04-read-file-sync.js: 191
03-read-file-callback.js: 204
03-read-file-callback-error.js: 197
```

Two problems jump out at us. First, we can see that "done!" is printed before all of our files, and second, our files are not printed in the alphabetical order that `fs.readdir()` returns them.

Both of these problems stem from the following lines:

02-async/05-read-dir-callbacks-broken.js

```
  files.forEach(function (file) {
    fs.readFile(file, (err, fileData) => {
      if (err) return console.error(err)

      console.log(`${file}: ${fileData.length}`)
    })
  })

  console.log('done!')
```

If we were to run the following, we would have the same issue:

```
const seconds = [5, 2]
seconds.forEach(s => {
  setTimeout(() => console.log(`Waited ${s} seconds`), 1000 * s)
})
console.log('done!')
```

Just like `setTimeout()`, `fs.readFile()` does not block execution. Therefore, Node.js is not going to wait for the file data to be read before printing "done!" Additionally, even though `5` is first in the `seconds` array, "Waited 5 seconds" will be printed last. In this example it's obvious because 5 seconds is a longer amount of time than 2 seconds, but the same issue happens with `fs.readFile()`; it takes less time to read some files than others.

If we can't use `Array.forEach()` to do what we want, what can we do? The answer is to create our own async iterator. Just like how there can be synchronous variants of asynchronous functions, we can make async variants of synchronous functions. Let's take a look at an async version of `Array.map()`

02-async/06a-read-dir-callbacks.js

```
function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

If this function looks confusing, that's OK; it's abstract so that it can accept an arbitrary array (`arr`) and iterator function (`fn`), and it takes advantage of Node.js conventions to make things work. We'll go through it piece by piece to make sure everything is clear.

For comparison, let's look at a simple synchronous version of `Array.map()`:

```
function mapSync (arr, fn) {
  const results = []

  arr.forEach(function (item, i) {
    const data = fn(item)
    results[i] = data
  })

  return results
}
```

At a high level, our `mapAsync()` is very similar to `mapSync()`, but it needs a little extra functionality to make async work. The main additions are that it (1) keeps track of how many items from the array (`arr`) have been processed, (2) whether or not there's been an error, and (3) a final callback (`onFinish`) to run after all items have been successfully processed or an error occurs.

Just like `mapSync()`, the first two arguments of `mapAsync()` are `arr`, an array of items, and `fn`, a function to be executed for each item in the array. Unlike `mapSync()`, `mapAsync()` expects `fn` to be an asynchronous function. This means that when we execute `fn()` for an item in the array, it won't immediately return a result; the result will be passed to a callback.

Therefore, instead of being able to synchronously assign values to the `results` array in `mapSync()`:

```
arr.forEach(function (item, i) {
  const data = fn(item)
  results[i] = data
})
```

We need assign values to the `results` within the callback of `fn()` in `mapAsync()`:

```
arr.forEach(function (item, i) {
  fn(item, function (err, data) {
    results[i] = data
  })
})
```

This means that when using `mapAsync()`, we expect the given iterator function, `fn()`, to follow Node.js convention by accepting an `item` from `arr` as its first argument and a callback as the last argument. This ensures that any function following Node.js convention can be used. For example, `fs.readFile()` could be given as the `fn` argument to `mapAsync()` because it can be called with the form: `fs.readFile(filename, callback)`.

Because the `fn()` callback for each item will execute in a different order than the items array (just like our `setTimeout()` example above), we need a way to know when we are finished processing all items in the array. In our synchronous version, we know that we're finished when the last item is processed, but that doesn't work for `mapAsync()`.

To solve this problem, we need to keep track of how many items have been completed successfully. By keeping track, we can make sure that we only call `onFinish()` after all items have been completed successfully.

Specifically, within the `fn()` callback, after we assign value to our `results` array, we check to see if we have processed all items. If we have, we call `onFinish()` with the results, and if we haven't we do nothing.

```
function mapAsync (arr, fn, onFinish) {
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

At this point, after everything goes well, we'll call `onFinish()` with a correctly ordered `results` array. Of course, as we know, everything does not always go well, and we need to know when it doesn't.

Node.js convention is to call callbacks with an error as the first argument if one exists. In the above example, if we call `mapAsync()` and any of the items has an error processing, we'll never know. For example, if we were to use it with `fs.readFile()` on files that don't exist:

```
mapAsync(['file1.js', 'file2.js'], fs.readFile, (err, filesData) => {
  if (err) return console.error(err)
  console.log(filesData)
})
```

Our output would be `[undefined, undefined]`. This is because `err` is `null`, and without proper error handling, our `mapAsync()` function will push `undefined` values into the `results` array. We've received a faulty results array instead of an error. This is the opposite of what we want; we want to follow Node.js convention so that we receive an error instead of the results array.

If any of our items has an error, we'll call `onFinish(err)` instead of `onFinish(null, results)`. Because `onFinish()` will only be called with the results after all items have finished successfully, we can avoid that with an early return:

```
function mapAsync (arr, fn, onFinish) {
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (err) return onFinish(err)

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Now, if we run into an error, we'll immediately call `onFinish(err)`. In addition, because we don't decrease our `nRemaining` count for the item with an error, `nRemaining` never reaches `0` and `onFinish(null, results)` is never called.

Unfortunately, this opens us up to another problem. If we have multiple errors, `onFinish(err)` will be called multiple times; `onFinish()` is expected to be called only once.

Preventing this is simple. We can keep track of whether or not we've encountered an error already. Once we've already encountered an error and called `onFinish(err)`, we know that (A) if we encounter another error, we should call `onFinish(err)` again, and (B) even if we don't encounter another error, we'll never have a complete set of results. Therefore, there's nothing left to do, and we can stop:

02-async/06a-read-dir-callbacks.js

```
function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Now that we've added proper error handling, we have a useful asynchronous `map()` function that we can use any time we want to perform a task concurrently for an array of items. By taking advantage of and following Node.js convention, our `mapAsync()` is directly compatible with most core Node.js API methods and third-party modules.

Here's how we can use it for our directory listing challenge:

02-async/06b-read-dir-callbacks-direct.js

```
fs.readdir('./', function (err, files) {
  if (err) return console.error(err)

  mapAsync(files, fs.readFile, (err, results) => {
    if (err) return console.error(err)

    results.forEach((data, i) => console.log(`${files[i]}: ${data.length}`))

    console.log('done!')
  })
})
```

Creating A Function

Now that we have a general-purpose async map and we've used it to get a directory listing and read each file in the directory, let's create a general-purpose function that can perform this task for *any* specified directory.

To achieve this we'll create a new function `getFileLengths()` that accepts two arguments, a directory and a callback. This function will first call `fs.readdir()` using the provided directory, then it pass the file list and `fs.readFile` to `mapAsync()`, and once that is all finished, it will call the callback with an error (if one exists) or the results array - pairs of filenames and lengths.

Once we've created `getFileLengths()` we'll be able to use it like this:

02-async/06c-read-dir-callbacks-cli.js

```
const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory, function (err, results) {
  if (err) return console.error(err)

  results.forEach(([file, length]) => console.log(`${file}: ${length}`))

  console.log('done!')
})
```

`process.argv` is an globally available array of the command line arguments used to start Node.js. If you were to run a script with the command `node file.js`, `process.argv` would be equal to `['node', 'file.js']`. In this case we allow our file to be run like `node 06c-read-dir-callbacks-cli.js ../another-directory`. In that case `process.argv[2]` will be `../another-directory`.

To make this work we'll define our new `getFileLengths()` function:

02-async/06c-read-dir-callbacks-cli.js

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}
```

Unsurprisingly, the first thing we do is use `fs.readdir()` to get the directory listing. We also follow Node.js convention, and if there's an error, we'll return early, and call the callback `cb` with the error.

Next, we need perform a little extra work on our file list for this function to be generalizable to *any* specified directory. `fs.readdir()` will only return file names - it will not return the full directory paths. This was fine for our previous example, because we were getting the file list of `./`, our current working directory, and `fs.readFile()` wouldn't need the full path to read a file in the current working directory. However, if we want this function to work for other directories, we need to be sure to pass more than just the file name to `fs.readFile()`. We use the built-in `path` module (`require('path')`) to combine our specified directory with each file name to get an array of file paths.

After we have our file paths array, we pass it to our `mapAsync()` function along with a customized `readFile()` function. We're not using `fs.readFile()` directly, because we need to alter its output a little bit.

It's often useful to wrap an async function with your own to make small changes to expected inputs and/or outputs. Here's what our customized `readFile()` function looks like:

02-async/06c-read-dir-callbacks-cli.js

```
function readFile (file, cb) {
  fs.readFile(file, function (err, fileData) {
    if (err) {
      if (err.code === 'EISDIR') return cb(null, [file, 0])
      return cb(err)
    }
    cb(null, [file, fileData.length])
  })
}
```

In our previous example, we could be sure that there were no subdirectories. When accepting an arbitrary directory, we can’t be so sure. Therefore, we might be calling `fs.readFile()` on a directory.

If this happens, `fs.readFile()` will give us an error. Instead of halting the whole process, we'll treat directories as having a length of 0.

Additionally, we want our `readFile()` function to return an array of both the file path *and* the file length. If we were to use the stock `fs.readFile()` we would only get the data.

`readFile()` will be called once for each file path, and after they have all finished, the callback passed to `mapAsync()` will be called with an array of the results. In this case, the results will be an array of arrays. Each array within the results array will contain a file path and a length.

Looking back at where we call `mapAsync()` within our `getFileLengths()` definition, we can see that we're taking the callback `cb` passed to `getFileLengths()` and handing it directly to `mapAsync()`:

02-async/06c-read-dir-callbacks-cli.js

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}
```

This means that the results of `mapAsync()` will be the results of `getFileLengths()`. It is functionally equivalent to:

```
mapAsync(filePaths, readFile, (err, results) => cb(err, results))
```

Here's our full implementation of `getFileLengths()`:

02-async/06c-read-dir-callbacks-cli.js

```
const fs = require('fs')
const path = require('path')

const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory, function (err, results) {
  if (err) return console.error(err)

  results.forEach(([file, length]) => console.log(`${file}: ${length}`))

  console.log('done!')
```

```
    })
```

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}

function readFile (file, cb) {
  fs.readFile(file, function (err, fileData) {
    if (err) {
      if (err.code === 'EISDIR') return cb(null, [file, 0])
      return cb(err)
    }
    cb(null, [file, fileData.length])
  })
}

function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Wrapping Up

Callbacks can be quite confusing at first because how different they can be from working with other languages. However, they are a powerful convention that allows us to create async variations of common synchronous tasks. Additionally, because of their ubiquity in Node.js core modules, it's important to be comfortable with them.

That said, there are alternative forms of async that build on top of these concepts that many people find easier to work with. Next up, we'll talk about promises.

Promises

A promise is an object that represents a future action and its result. This is in contrast to callbacks which are just conventions around how we use functions.

Let's take a look to see how we can use `fs.readFile()` with promises instead of callbacks:

02-async/07-read-file-promises.js

```
const fs = require('fs').promises

const filename = '07-read-file-promises.js'

fs.readFile(filename)
  .then(data => console.log(`${filename}: ${data.length}`))
  .catch(err => console.error(err))
```

vs:

02-async/03-read-file-callback.js

```
const fs = require('fs')

const filename = '03-read-file-callback.js'

fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

So far, the biggest difference is that the promise has separate methods for success and failure. Unlike callbacks that are a single function with both error and results arguments, promises have separate methods `then()` and `catch()`. If the action is successful, `then()` is called with the result. If not, `catch()` is called with an error.

However, let's now replicate our previous challenge of reading a directory and printing all the file lengths, and we'll begin to see how promises can be helpful.

Real World Promises

Just looking at the above example, we might be tempted to solve the challenge like this:

02-async/08-read-dir-promises-broken.js

```
const fs = require('fs').promises

fs.readdir('./')
  .catch(err => console.error(err))
  .then(files => {
    files.forEach(function (file) {
      fs.readFile(file)
        .catch(err => console.error(err))
        .then(data => console.log(`${file}: ${data.length}`))
    })

    console.log('done!')
  })
```

Unfortunately, when used this way, we'll run into the same issue with promises as we did with callbacks. Within the `files.forEach()` iterator, our use of the `fs.readFile()` promise is non- blocking. This means that we're going to see `done!` printed to the terminal before any of the results, and we're going get the results out of order.

To be able to perform multiple async actions concurrently, we'll need to use `Promise.all()`. `Promise.all()` is globally available in Node.js, and it execute an array of promises at the same time. It's conceptually similar to the `mapAsync()` function we built. After all promises have been completed, it will return with an array of results.

Here's how we can use `Promise.all()` to solve our challenge:

02-async/09a-read-dir-promises.js

```
const fs = require('fs').promises

fs.readdir('./')
  .then(fileList =>
    Promise.all(
      fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
    )
  )
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))
```

After we receive the file list `fileList` from `fs.readdir()` we use `fileList.map()` to transform it into an array of promises. Once we have an array of promises, we use `Promise.all()` to execute them all in parallel.

One thing to notice about our transformation is that we are not only transforming each file name into a `fs.readFile()` promise, but we are also customizing the result:

02-async/09a-read-dir-promises.js

```
      fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
```

If we had left the transformation as:

```
fileList.map(file => fs.readFile(file))
```

When `Promise.all()` finishes, `results` will simply be an array of file data. Without also having the file names, we no longer will know which files the lengths belong to. In order to keep each length properly labeled, we need to modify what each promise returns. We do this by adding `.then()` returning `[file, data.length]`.

Creating A Function

Now that we've solved the challenge for a single directory, we can create a generalized function that can be used for any directory. Once we've generalized it, we can use it like this:

02-async/09b-read-dir-promises-fn.js

```
const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory)
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))
```

Our new `getFileLengths()` is similar to what we did in the previous section. First we read the directory list, and then we use `fileList.map()` to transform that list into an array of promises. However, just like our callback example, we need some extra logic to handle arbitrary directories. Before creating a promise to read a file, we use `path.join()` to combine the directory with the file name to create a usable file path.

02-async/09b-read-dir-promises-fn.js

```
function getFileLengths (dir) {
  return fs.readdir(dir).then(fileList => {
    const readFiles = fileList.map(file => {
      const filePath = path.join(dir, file)
      return readFile(filePath)
    })
    return Promise.all(readFiles)
  })
}
```

Just like our callback example, we use a customized `readFile()` function so that we can both ensure that our final result array is made up of file path, file length pairs, and that subdirectories are correctly handled. Here's what that looks like:

02-async/09b-read-dir-promises-fn.js

```
function readFile (filePath) {
  return fs
    .readFile(filePath)
    .then(data => [filePath, data.length])
    .catch(err => {
      if (err.code === 'EISDIR') return [filePath, 0]
      throw err
    })
}
```

The promise version of `readFile()` behaves similarly, but the implementation is a little different. As mentioned above, one of the biggest differences between callbacks and promises is error handling. In contrast to callbacks, the success and error paths of callbacks are handled with separate functions.

When `then()` is called, we can be certain that we have not run into an error. We will have access to the file's data, and we can return a `[filePath, data.length]` pair as the result.

Our callback example was able to return early with a value if we encountered a particular error code (`EISDIR`). With promises, we need to handle this differently.

With promises, errors flow into a the separate `catch()` function. We can intercept `EISDIR` errors and prevent them from breaking the chain. If the error code is `EISDIR`, we return with our modified result, `[filePath, 0]`. By using `return` within a `catch()`, we prevent the error from propagating. To downstream code, it will look like the operation successful returned this result.

If any other error is thrown, we make sure not to return with a value. Instead, we re-throw the error. This will propagate the error down the chain - successive `then()` calls will be skipped, and the next `catch()` will be run instead.

Each call to `readFile()` will return a promise that results in a file path and length pair. Therefore when we use `Promise.all()` on an array of these promises, we will ultimately end up with an array of these pairs - our desired outcome.

Here's what the full file looks like:

02-async/09b-read-dir-promises-fn.js

```
const fs = require('fs').promises
const path = require('path')

const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory)
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))

function getFileLengths (dir) {
  return fs.readdir(dir).then(fileList => {
    const readFiles = fileList.map(file => {
      const filePath = path.join(dir, file)
      return readFile(filePath)
    })
    return Promise.all(readFiles)
    })
```

```
}

function readFile (filePath) {
  return fs
    .readFile(filePath)
    .then(data => [filePath, data.length])
    .catch(err => {
      if (err.code === 'EISDIR') return [filePath, 0]
      throw err
    })
}
```

Wrapping Up

Promises give us new ways of handling sequential and parallel async tasks, and we can take advantage of chaining `.then()` and `.catch()` calls to compose a series of tasks into a single function.

Compared to callbacks, we can see that promises have two nice properties. First, we did not need to use our own `mapAsync()` function - `Promise.all()` is globally available and handles that functionality for us. Second, errors are automatically propagated along the promise chain for us. When using callbacks, we need to check for errors and use early returns to manage this ourselves.

In the next section we'll build on top of promises to show off the use of the `async` and `await` language features. These allow us to use promises as if they were synchronous.

TODO: rename custom `readFile()` to `getFileLength()`

Async & Await

What if we could have the non-blocking performance of asynchronous code, but with the simplicity and familiarity of synchronous code? In this section we'll show how we can get a bit of both.

The `async` and `await` keywords allow us to treat specific uses of promises as if they were synchronous. Here's an example of using them to read data from a file:

02-async/10-read-file-await.js

```
const fs = require('fs').promises

printLength('10-read-file-await.js')

async function printLength (file) {
  try {
    const data = await fs.readFile(file)
    console.log(`${file}: ${data.length}`)
  } catch (err) {
    console.error(err)
  }
}
```

One cool thing about this is that we can use standard synchronous language features like `try/catch`. Even though `fs.readFile()` is a promise (and therefore asynchronous), we're able to wrap it in a `try/catch` block for error handling – just like we would be able to do for synchronous code. We don't need to use `catch()` for error handling.

In fact, we don't need to use `then()` either. We can directly assign the result of the promise to a variable, and use it on the following line.

However, it's important to note, that we can only do these things within special `async` functions. Because of these, when we declare our `printLength()` function we use this syntax:

```
async function printLength (file) { ... }
```

Once we do that, we are able to use the `await` keyword within. For the most part, `await` allows us to treat promises as synchronous. As seen above, we can use `try/catch` and variable assignment. Most importantly, even though our code will run *as if* these operations are synchronous, they won't block other executing tasks.

In many cases this can be very helpful and can simplify our code, but there are still gotchas to be aware of. Let's go through our directory reading challenge one last time and take a look.

Real World Async/Await

Just like before we're going to first get a directory list, then get each file's length, print those lengths, and after that's all finished, we'll print 'done!'.

For those of us who are new to asynchronous programming in Node.js, it might have felt a bit complicated to perform these tasks using callbacks or promises. Now that we've seen `async` and `await`, it's tempting to think that we'll be able to handle this task in a much more straightforward way.

Let's look at how we might try to solve this challenge with `async` and `await`:

02-async/11-read-dir-await-broken.js

```
const fs = require('fs').promises

printLengths('./')

async function printLengths (dir) {
  const fileList = await fs.readdir(dir)

  const results = fileList.map(
    async file => await fs.readFile(file).then(data => [file, data.length])
  )

  results.forEach(result => console.log(`${result[0]}: ${result[1]}`))

  console.log('done!')
}
```

Unfortunately, this won't work. If we run `node 11-read-dir-await-broken.js` we'll see something like this:

```
 node 11-read-dir-await-broken.js
undefined: undefined
undefined: undefined
undefined: undefined
undefined: undefined
undefined: undefined
...
```

What happened to our file names and lengths? The problem is our use of `fileList.map()`. Even though we specify the iterator function as `async` so that we can use `await` each time we call `fs.readFile()`, we can't use `await` on the call to `fileList.map()` itself. This means that Node.js will not wait for each promise within the iterator to complete before moving on. Our `dataFiles` array will not be an array of file data; it will be an array of promises.

When we iterate over our `dataFiles` array, we will print the length of each item. Instead of printing the length of a file, we're printing the length of a promise – which is `undefined`.

Luckily, the fix is simple. In the last section we used `Promise.all()` to treat an array of promises as a single promise. Once we convert the array of promises to a single promise, we can use `await` as we expect. Here's what that looks like:

02-async/12a-read-dir-await.js

```
const fs = require('fs').promises

printLengths('./')

async function printLengths (dir) {
  const fileList = await fs.readdir(dir)
  const results = await Promise.all(
    fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
  )
  results.forEach(([file, length]) => console.log(`${file}: ${length}`))
  console.log('done!')
}
```

Creating Async/Await Functions

Since `printLengths()` accepts a directory argument, it may look like we've already created a generalized solution to this problem. However, our solution has two issues. First, it is currently unable to properly handle subdirectories, and second, unlike our previous generalized solutions, our `printLengths()` function will not return the files and lengths – it will only print them.

Like we've done with our promise and callback examples, let's create a generalized `getFileLengths()` function that can work on arbitrary directories and will return with an array of file and length pairs.

We need to keep `printLengths()` because we can't take advantage of `await` outside of an `async` function. However, within `printLengths()` we will call our new `getFileLengths()` function, and unlike before, we can take advantage of `async` and `await` to both simplify how our code looks and to use `try/catch` for error handling:

02-async/12b-read-dir-await-fn.js

```
const targetDirectory = process.argv[2] || './'

printLengths(targetDirectory)

async function printLengths (dir) {
  try {
    const results = await getFileLengths(dir)
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  } catch (err) {
    console.error(err)
  }
}
```

Unlike our previous promise example of `getFileLengths()`, we don't need to use `then()` or `catch()`. Instead, we can use direct assignment for `results` and `try/catch` for errors.

Let's take a look at our `async/await` version of `getFileLengths()`:

02-async/12b-read-dir-await-fn.js

```
async function getFileLengths (dir) {
  const fileList = await fs.readdir(dir)

  const readFiles = fileList.map(async file => {
    const filePath = path.join(dir, file)
    return await readFile(filePath)
  })

  return await Promise.all(readFiles)
}
```

Like before, we can do direct assignment of `fileList` without using `then()` or `catch()`. Node.js will wait for `fs.readdir()` to finish before continuing. If there's an error, Node.js will throw, and it will be caught in the `try/catch` block in `printLengths()`.

Also, just like our callback and promise versions, we're going to use a customized `readFile()` function so that we can handle subdirectories. Here's what that looks like:

02-async/12b-read-dir-await-fn.js

```
async function readFile (filePath) {
  try {
    const data = await fs.readFile(filePath)
    return [filePath, data.length]
  } catch (err) {
    if (err.code === 'EISDIR') return [filePath, 0]
    throw err
  }
}
```

We're able to return a value from within our `catch` block. This means that we can return `[filePath, 0]` if we encounter a directory. However, if we encounter any other error type, we can `throw` again. This will propagate the error onwards to any `catch` block surrounding the use of this function. This is conceptually similar to how we would selectively re-throw in the promises example.

Once `readFile()` has been called for each file in the `fileList` array, we'll have an array of promises, `readFiles` - calling an `async` function will return a promise. We then return `await Promise.all(readFiles)`, this will be an array of results from each of the `readFile()` calls.

And that's all we need. If there's an issue in any of the `readFile()` calls within the `Promise.all()`, the error will propagate up to where we call `getFileLengths(dir)` in `printLengths()` - which can be caught in the `try/catch` there.

Here's the full generalized solution to the challenge:

02-async/12b-read-dir-await-fn.js

```
const fs = require('fs').promises
const path = require('path')

const targetDirectory = process.argv[2] || './'

printLengths(targetDirectory)

async function printLengths (dir) {
  try {
    const results = await getFileLengths(dir)
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  } catch (err) {
    console.error(err)
  }
}

async function getFileLengths (dir) {
  const fileList = await fs.readdir(dir)

  const readFiles = fileList.map(async file => {
    const filePath = path.join(dir, file)
    return await readFile(filePath)
  })

  return await Promise.all(readFiles)
}

async function readFile (filePath) {
  try {
    const data = await fs.readFile(filePath)
    return [filePath, data.length]
  } catch (err) {
    if (err.code === 'EISDIR') return [filePath, 0]
    throw err
  }
}
```

Wrapping Up

We have now solved the same real-world challenge with three different techniques for handling asynchronous tasks. The biggest differences with `async/await` is being able to use a more syn- chronous coding style and `try/catch` for error handling.

It's important to remember that under the hood, `async/await` is using promises. When we declare an async function, we're really creating a promise. This can be seen most clearly with our use of `Promise.all()`.

We're now going to move beyond callbacks, promises, and `async/await`. Each one of these styles are focused on performing "one and done" asynchronous tasks. We're now going to turn our attention to ways of handling types of repeated asynchronous work that are very common in Node.js: event emitters and streams.

Event Emitters

Event emitters are not new to Node.js. In fact, we've already used an example that's common in the browser:

```
window.addEventListener('resize', () => console.log('window has been resized!'))
```

It's true that like callbacks and promises, adding event listeners allow us create logic around future timelines, but the big difference is that events are expected to repeat.

Callbacks and promises have an expectation that they will resolve once and only once. If you recall from our callback example, we needed to add extra logic to `mapAsync()` to ensure that multiple errors would not cause the callback to be called multiple times.

Event emitters, on the other hand, are designed for use-cases where we expect a variable number of actions in the future. If you recall from our chat example when we built the API, we used an event emitter to handle chat message events. Chat messages can happen repeatedly or not at all. Using an event emitter allows us to run a function each time one occurs.

We didn't dive into the details of that event emitter, but let's take a closer look now. We'll create a command line interface where a user can type messages. We'll use an event emitter to run a function each time the user enters in a full message and presses "return".

Event Emitters: Getting Started

When we built our chat app, we created our own event emitter. Many core Node.js methods provide a callback interface, but many also provide an event emitter interface.

For our first example, we're going to use the core module `readline`. `readline` allows us to "watch" a source of data and listen to "line" events. For our use-case we're going to watch `process.stdin`, a data source that will contain any text that a user types in while our Node.js app is running, and we're going to receive message events any time the user presses "return."

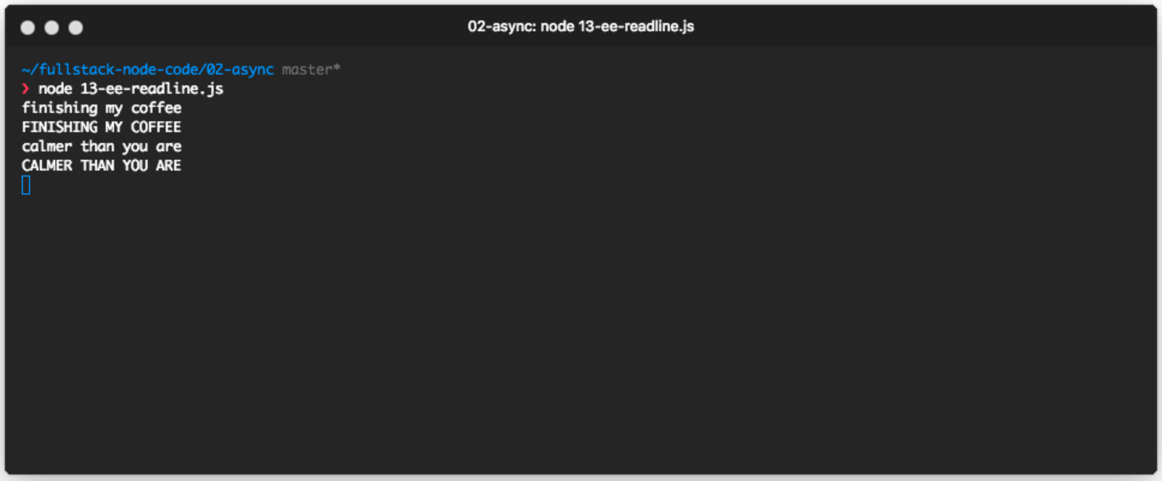
Each time we receive a message, we're going to transform that message to all uppercase and print it out to the terminal.

Here's how we can do that:

02-async/13-ee-readline.js

```
const readline = require('readline')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line => console.log(line.toUpperCase()))
```

If we run it, we can get something like the following:



```
~/fullstack-node-code/02-async master*
› node 13-ee-readline.js
finishing my coffee
FINISHING MY COFFEE
calmer than you are
CALMER THAN YOU ARE
```

The lower case text is our input, and the uppercase is printed out by our script.

Once we create an instance of `readline`, we can use its `on()` method to listen for particular types of events. For right now, we're interested in the `line` event type.

`readline` is a core Node.js module. You can see all other event types that are available in the official Node.js documentation[46]

Creating event handlers for specific event types is similar to using callbacks, except that we don't receive an error argument. This is just like how it works in the browser when we use `document.addEventListener(event => {})`. The event handler function does not expect an error argument.

In addition to basic logging, we can use event handlers to perform other work. In the next section, we'll see how we can use our event emitter to provide another interface to the chat app we built in the first chapter.

Event Emitters: Going Further

In the first chapter we built a chat app. We could open two different browsers, and anything we typed into one window would appear in the other. We're going to show that we can do this without a browser.

By making a small tweak to our `readline` example, we'll be able to open our chat app in a browser window and send messages to it using our terminal:

02-async/15-ee-readline-chat-send.js

```
const http = require('http')
const readline = require('readline')
const querystring = require('querystring')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line =>
  http.get(
    `http://localhost:1337/chat?${querystring.stringify({ message: line })}`
  )
)
```

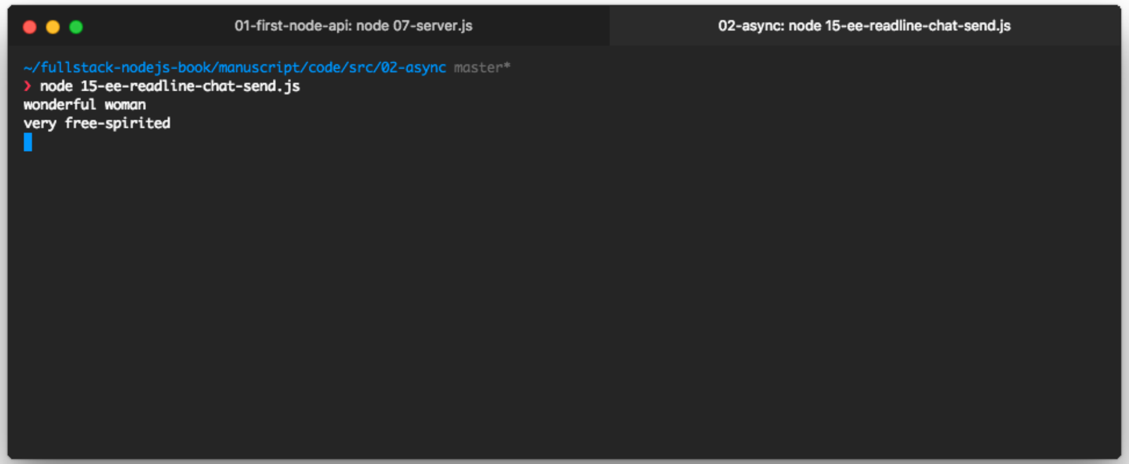
For this to work, we need to make sure that our server from chapter 1 is running and listening on port 1337. You can do this by opening a new terminal tab, navigating to the chapter 1 code directory, and running `node 07-server.js`. You should see a message saying `Server listening on port 1337`.

The code hasn't changed much. We have only replaced our `console.log()` with a call to `http.get()`. We use the same built-in `http` module that we used in chapter 1. However, this time we using `http` to create requests instead of responding to them.

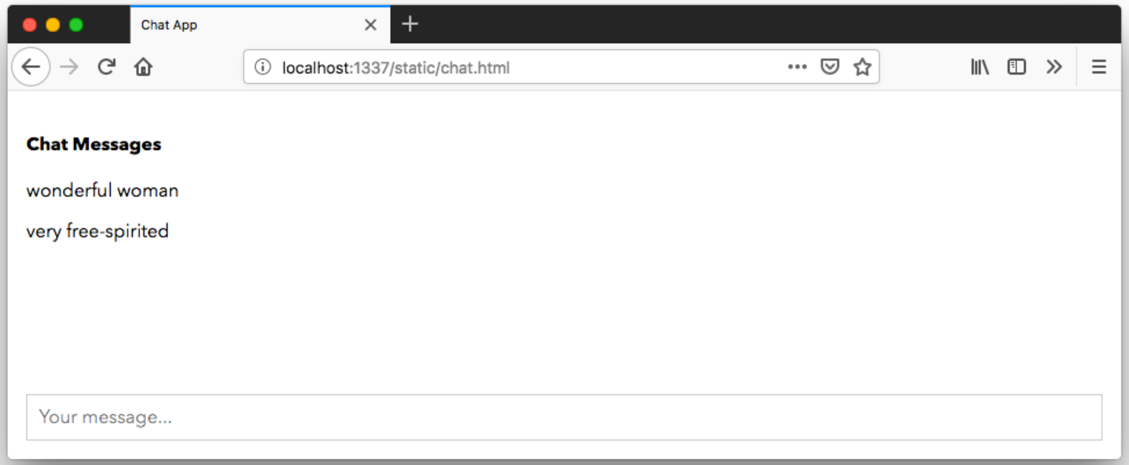
[46]https://nodejs.org/api/readline.html#readline_class_interface

To ensure that our messages are properly formatted and special characters are escaped, we also use the built-in `querystring` module.

We can now run our server from chapter 1, our browser window to the chat app, run `node 15-ee-readline-chat-send.js`, and start typing messages into the terminal. Each message should appear in the open browser window:


01-first-node-api: node 07-server.js
02-async: node 15-ee-readline-chat-send.js

```
~/fullstack-nodejs-book/manuscript/code/src/02-async master*
› node 15-ee-readline-chat-send.js
wonderful woman
very free-spirited
```



Chat Messages

wonderful woman

very free-spirited

What's cool about this is that it shows off how easily Node.js can be used to chain functionality together. By creating small pieces of functionality with well defined interfaces, it's very easy to get new functionality for free.

When we built our chat app in chapter 1, we didn't plan on wanting a CLI client. However, because the interface is straightforward, we were easily able to get that functionality.

In the next section we'll take it even further. We're going to create our own event emitter object, and not only will we be able to send messages, but we'll be able to receive them as well.

Event Emitters: Creating Custom Emitters

Now that we've added the capability to send messages, we can also add some functionality to receive them.

The only thing we need to do is to make an HTTP request to our API and log the messages as they come in. In the previous section, we make HTTP requests to send messages, but we don't do anything with the response. To handle response data from the HTTP request, we'll need to use both a callback and an event emitter:

02-async/16-ee-readline-chat-receive.js

```
http.get('http://localhost:1337/sse', res => {
  res.on('data', data => console.log(data.toString()))
})
```

`http.get()` is interesting because it accepts a callback as its second argument - and the response argument (`res`) the callback receives is an event emitter.

What's going on here is that after we make the request, we need to wait for the response. The response object doesn't come with all of the data, instead we have to subscribe to the `"data"` events. Each time we receive more data, that event will fire, passing along the newest bit.

You'll notice that we call `data.toString()` to log our chat messages. If we don't do that, we would see the raw bytes in hex. For efficiency, Node.js often defaults to a data type called `Buffer`. We won't go into detail here, but it's easy enough to convert buffers into strings using `buffer.toString()`.

Here's what the full file looks like with our addition:

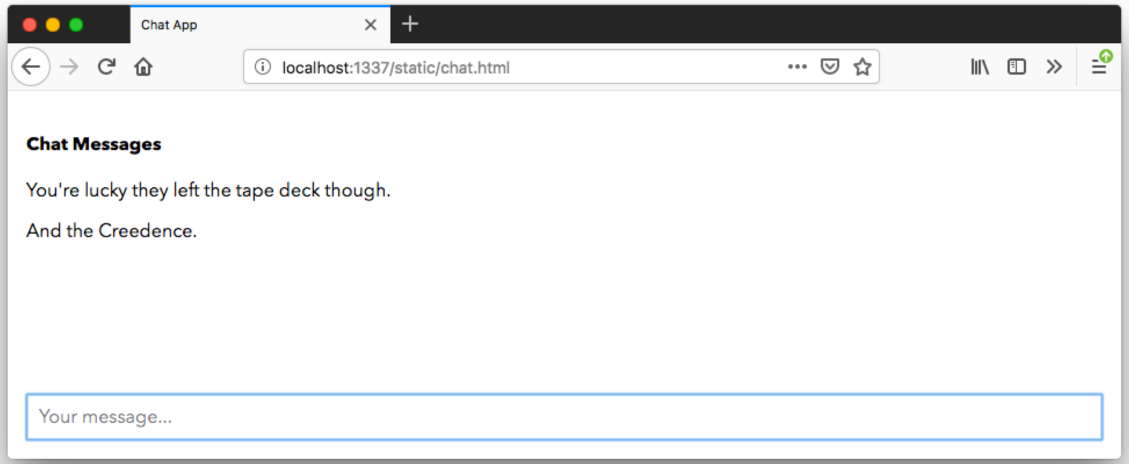
02-async/16-ee-readline-chat-receive.js

```
const http = require('http')
const readline = require('readline')
const querystring = require('querystring')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line => {
  http.get(
    `http://localhost:1337/chat?${querystring.stringify({ message: line })}`
```

)

```
})

http.get('http://localhost:1337/sse', res => {
  res.on('data', data => console.log(data.toString()))
})
```

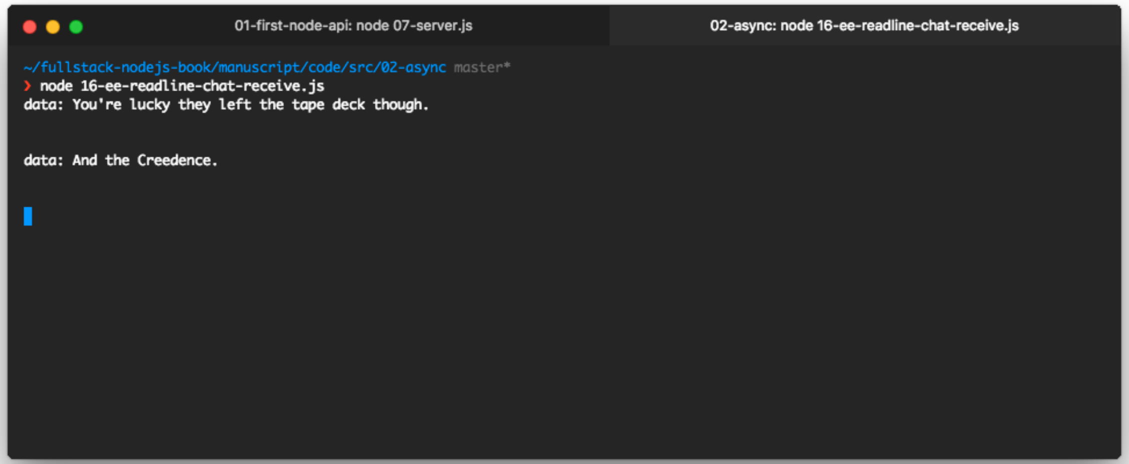
If we run it with `node 16-ee-readline-chat-receive.js`, we will be able to see any messages we type into the browser:



Chat Messages

You're lucky they left the tape deck though.

And the Creedence.



01-first-node-api: node 07-server.js
02-async: node 16-ee-readline-chat-receive.js

```
~/fullstack-nodejs-book/manuscript/code/src/02-async master*
› node 16-ee-readline-chat-receive.js
data: You're lucky they left the tape deck though.


data: And the Creedence.
```

This works, but we can do better. Each message is prefixed with `data:` and is followed by two newlines. This is expected because it's the data format of the Server-sent events specification[47].

[47]https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

We're going to create a simple interface where we can pull messages from any SSE endpoint, and we're going to use the power of event emitters to make this really easy.

Instead of just making an HTTP request and logging what we receive, we'll create our own general- purpose event emitter that will give us custom "message" events without the extra stuff.

To do this we're going to create a function that returns a new event emitter. However, before this function returns the event emitter, it will also set up the HTTP connection to receive chat messages. When those messages come in, it will use the new event emitter to emit them.

This is a useful pattern because it allows us to synchronously create an object that will act asynchronously. In some ways, this is similar to how a promise works.

Here's our new `createEventSource()` function:

02-async/17-ee-create-emitter.js

```
function createEventSource (url) {
  const source = new EventEmitter()

  http.get(url, res => {
    res.on('data', data => {
      const message = data
        .toString()
        .replace(/^data: /, '')
        .replace(/\n\n$/, '')

      source.emit('message', message)
    })
  })

  return source
}
```

And here's how we can use it:

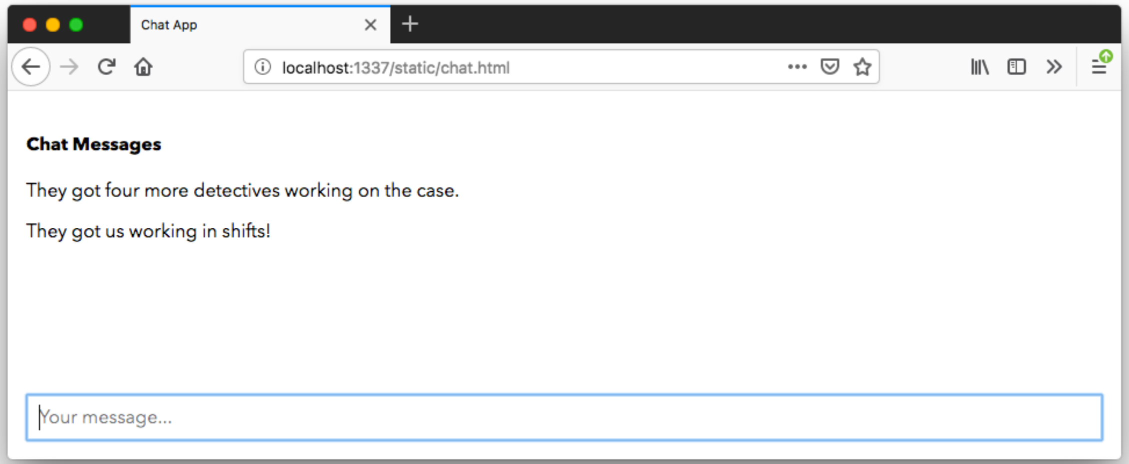
02-async/17-ee-create-emitter.js

```
const source = createEventSource('http://localhost:1337/sse')
```

02-async/17-ee-create-emitter.js

```
source.on('message', console.log)
```

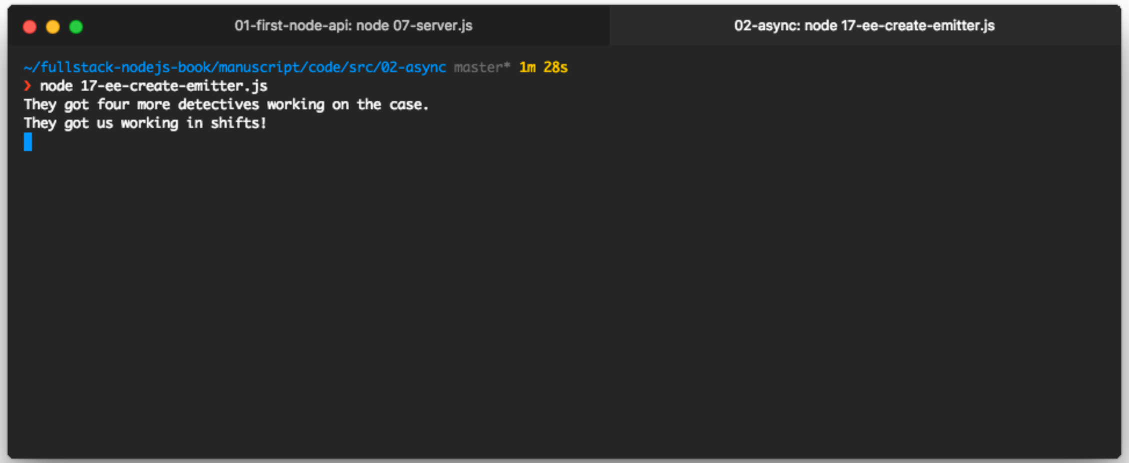
Because we're cleaning up the data before we pass it to our new event emitter, when we log messages they are trimmed down to just the parts we care about:



Chat Messages

They got four more detectives working on the case.

They got us working in shifts!


01-first-node-api: node 07-server.js
02-async: node 17-ee-create-emitter.js

```
~/fullstack-nodejs-book/manuscript/code/src/02-async master* 1m 28s
❯ node 17-ee-create-emitter.js
They got four more detectives working on the case.
They got us working in shifts!
```

This is great, but one of the biggest advantages of event emitters is that they can emit lots of different types of events. Instead of just emitting a "message" event, we can go further and emit different events depending on the content of the message.

There's no limit to the number or types of events that we can emit. For example, we can look at a chat message to see if it ends with a question mark. If it does, we can can emit a "question" event, and if not, we can emit a "statement" event. Furthermore, these can be in addition to our exiting "message" event.

This is nice because it gives choice to the consumer of our event emitter. The consumer can choose which events they would like to subscribe to.

By adding a small amount of logic to determine the event type of our additional `emit()` call:

02-async/18-ee-create-emitter-custom-events.js

```
function createEventSource (url) {
  const source = new EventEmitter()

  http.get(url, res => {
    res.on('data', data => {
      const message = data
        .toString()
        .replace(/^data: /, '')
        .replace(/\n\n$/, '')

      source.emit('message', message)

      const eventType = message.match(/\?$/) ? 'question' : 'statement'
      source.emit(eventType, message)
    })
  })

  return source
}
```

We can choose to only listen for questions, and change our language to suit:

```
source.on('question', q => console.log(`Someone asked, "${q}"`))
```

Event Emitters: Wrapping Up

The big advantage of event emitters is that they allow us to handle async behavior when the future action is either uncertain or repetitive. Unlike callbacks and promises (and therefore async/await), event emitters are not as useful for “one and done”-type behaviors.

In addition, by encapsulating filtering behavior into the event emitter itself, we can make very clean interfaces for consumers.

Next up we’re going to learn about one more common form of async behavior in Node.js which is actually a specific type of event emitter: streams.

Streams

Streams are ubiquitous within Node.js. In fact, almost all Node.js applications, no matter how simple, use streams in some manner[^From https://nodejs.org/docs/latest-v11.x/api/stream.html#stream_-

api_for_stream_consumers]. Our apps are no exception, we’ve already used them a number of times.

Streams are a specific type of event emitter. This means that they have the same methods available like `emit()` and `on()`. However, what makes streams so useful is that they follow conventions about how their events are emitted.

This is easiest to see with an example. Let's take a look at stream object that we just encountered, the `http` response object.

```
http.get(url, res => {
  res.on('data', data => {
    // here's where we use the data
  })
})
```

In the snippet above, `res` is a stream, and because streams are event emitters, we can use `on()` to listen to its events and receive updates.

What's cool about streams is that they standardize on the types of events that they emit. This makes them very predictable to use, and allows us to chain them together in interesting ways.

When we created our event emitter, we arbitrarily decided that we would emit a "message", "question", and "statement" event types. This means that any consumer of our event emitter would have to look at the code or at documentation to know to subscribe to those event types.

On the other hand, when reading from a stream, we can always expect "data", "error", and "end" events. Just by knowing that `res` is a stream, we know that we can get its data with the "data" event.

For example, here's what it looks like to download a file, using a `https` response stream's events:

02-async/20-streams-download-book-cover-batch.js

```
const fs = require('fs')
const https = require('https')

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'

https.get(fileUrl, res => {
  const chunks = []

  res.on('data', data => chunks.push(data)).on('end', () =>
    fs.writeFile('book.png', Buffer.concat(chunks), err => {
      if (err) console.error(err)
      console.log('file saved!')
    })
  )
})
```

? By default, Node.js uses `Buffer` objects to store and transmit data because it's more efficient. We'll go over `Buffer` more later, but for now just know that (1) `Buffer.concat()` can convert an array of `Buffer` objects into a single `Buffer`, and (2) `fs.writeFile()` is happy to accept a `Buffer` as its argument for what should be written to a file.

The way we do this makes sense when we think about the response stream as an event emitter. We don't know ahead of time how many times the "data" event will be emitted. If this file is small, it could happen only once. Alternatively, if the file is really large, it could happen many times. If there's an error connecting, it may not happen at all.

Our approach here is to collect all of the data "chunks" in an array, and only once we receive the "end" event to signal that no more are coming, do we proceed to write the data to a file.

This is a very common pattern for performing a batched write. As each chunk of data is received, we store it in memory, and once we have all of them, we write them to disk.

The downside of batching is that we need to be able to hold all of the data in memory. This is not a problem for smaller files, but we can run into trouble when working with large files - especially if the file is larger than our available memory. Often, it's more efficient to write data *as we receive it*.

In addition to readable streams, there are also *writable* streams. For example, here's how we can create a new file (`time.log`), and append to it over time:

```
const writeStream = fs.createWriteStream('time.log')
setInterval(() => writeStream.write(`The time is now: ${new Date()}\n`), 1000)
```

Writable streams have `write()` and `end()` methods. In fact, we've already seen these in chapter 1. The HTTP response object is a writable stream. For most of our endpoints we send data back to the browser using `res.end()`. However, when we want to keep the connection open for SSE, we used `res.write()` so that we did not close the connection.

Let's change our previous example to use a writable stream to avoid buffering the image data in memory before writing it to disk:

02-async/20-streams-download-book-cover-write-stream.js

```
const fs = require('fs')
const https = require('https')

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'
```

```
https.get(fileUrl, res => {
  const fileStream = fs.createWriteStream('book.png')
  res.on('data', data => fileStream.write(data)).on('end', () => {
    fileStream.end()
    console.log('file saved!')
  })
})
```

As we can see, we were able to eliminate the `chunks` array that we used to store all the file data in memory. Instead, we write each chunk of data to disk as it comes in.

The beauty of streams is that because all of these methods and events are standardized, we actually don't need to listen to these events or call the methods manually.

Streams have an incredibly useful `pipe()` method that takes care of all of this for us. Let's do the same thing, but instead of setting up our own handlers, we'll use `pipe()`:

02-async/21-streams-download-book-cover.js

```
const fs = require('fs')
const https = require('https')

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'

https.get(fileUrl, res => {
  res
    .pipe(fs.createWriteStream('book.png'))
    .on('finish', () => console.log('file saved!'))
})
```

Streams provide us a very efficient way of transferring data, and using `pipe()` we can do this very succinctly and easily.

Composing Streams

So far we've seen readable streams and writable streams, and we've learned that we can connect readable streams to writable streams via `pipe()`.

This is useful for moving data from one place to another, but often times we'll want to transform the data in some way. We can do this using transform streams.

A transform stream behaves as both a readable stream and a writable stream. Therefore, we can pipe a read stream to a transform stream, and then we can pipe the transform stream to the write stream.

For example, if we wanted to efficiently transform the text in a file to upper case, we could do this:

02-async/23-streams-shout.js

```
const fs = require('fs')
const { Transform } = require('stream')

fs.createReadStream('23-streams-shout.js')
  .pipe(shout())
  .pipe(fs.createWriteStream('loud-code.txt'))

function shout () {
  return new Transform({
    transform (chunk, encoding, callback) {
      callback(null, chunk.toString().toUpperCase())
    }
  })
}
```

In this case we've created a function `shout()` that creates a new transform stream. This trans- form stream is both readable and writable. This means that we can pipe our read stream that we get from `fs.createReadStream()` to it, and we can also pipe it to our write stream from `fs.createWriteStream()`.

Our transform stream is created by a simple function that expects three arguments. The first is the chunk of data, and we're already familiar with this from our use of `on('data')`. The second is encoding, which is useful if the chunk is a string. However, because we have not changed any default behaviors with or read stream, we expect this value to be "buffer" and we can ignore it. The final argument is a callback to be called with the results of transforming the chunk. The value provided to the callback will be emitted as data to anything that is reading from the transform stream. The callback can also be used to emit an error. You can read more about transform streams in the official Node.js documentation[48].

In this particular case we are performing a synchronous transformation. However, because we are given a callback, we are also able to do asynchronous transformations. This is useful if you need to look up information from disk or from a remote network service like an HTTP API.

If we run this script, and we open the resulting file, we'll see that all the text has been transformed to upper case.

Of course, in real life we don't often need to perform streaming case changes, but this shows how we can create general-purpose modular transform streams that can be used with a wide range of data.

In the next section we'll take a look at a transform stream that is useful in the real world.

[48]https://nodejs.org/api/stream.html#stream_implementing_a_transform_stream

Real World Transform Streams

A common use-case in the real world is converting one source of data to another format. When the source data is large, it's useful to use streams to perform the transformation.

Let's look at an example where we have a csv file, and we'd like to change a number of things:

1) "name" should be replaced with two separate "firstName" and "lastName" fields 2) the "dob" should be converted to an "age" integer field 3) the output should be newline delimited JSON instead of csv

Here's some data from our example `people.csv` file:

```
name,dob
Liam Jones,1988-06-26
Maximus Rau,1989-08-21
Lily Ernser,1970-01-18
Alvis O'Keefe,1961-01-19
...
```

Here's what we'd like `people.ndjson` to look like when we're finished:

```
{"firstName":"Liam","lastName":"Jones","age":30}
{"firstName":"Maximus","lastName":"Rau","age":29}
{"firstName":"Lily","lastName":"Ernser","age":49}
{"firstName":"Alvis","lastName":"O'Keefe","age":58}
{"firstName":"Amy","lastName":"Johnson","age":59}
...
```

Just like before, the first thing that we need to do is to use `fs.createReadStream()` to create a readable stream object for our `people.csv` file:

02-async/24-transform-csv.js

```
fs.createReadStream('people.csv')
```

Next, we want to pipe this stream to a transform stream that can parse csv data. We could create our own, but there's no need. We're going to use an excellent and appropriately named module `csv-parser` that is available on `npm`. However, before we can use this in our code, we need to run `npm install csv-parser` from our code directory.

Once that is installed we can use it like so:

```
const fs = require('fs')
const csv = require('csv-parser')

fs.createReadStream('people.csv')
  .pipe(csv())
  .on('data', row => console.log(JSON.stringify(row)))
```

When we run this, we'll see that when we pipe to the transform stream created with `csv()` will emit `data` events, and each logged event will be an object representing the parsed csv row:

```
{"name":"Liam Jones","dob":"1988-06-26"}
{"name":"Maximus Rau","dob":"1989-08-21"}
{"name":"Lily Ernser","dob":"1970-01-18"}
{"name":"Alvis O'Keefe","dob":"1961-01-19"}
{"name":"Amy Johnson","dob":"1960-03-04"}
...
```

By using `console.log()` on each JSON stringified row object, our output format is newline delimited JSON, so we're almost finished already. The only thing left to do is to add another transform stream into the mix to convert the objects before they are logged as JSON.

This will work the same way as our previous transform stream example, but with one difference. Streams are designed to work on `String` and `Buffer` types by default. In this case, our `csv-parser` stream is not emitting those types; it is emitting objects.

If we were to create a default transform stream with `clean()`:

02-async/24-transform-csv-error.js

```
const fs = require('fs')
const csv = require('csv-parser')
const { Transform } = require('stream')

fs.createReadStream('people.csv')
  .pipe(csv())
  .pipe(clean())
  .on('data', row => console.log(JSON.stringify(row)))

function clean () {
  return new Transform({
    transform (row, encoding, callback) {
      callback(null, row)
    }
  })
}
```

We would get the following error:

```
 node 24-transform-csv-error.js
events.js:174
          throw er; // Unhandled 'error' event
          ^

TypeError [ERR_INVALID_ARG_TYPE]: The "chunk" argument must be one of type string or\
  Buffer. Received type object
      at validChunk (_stream_writable.js:263:10)
      at Transform.Writable.write (_stream_writable.js:297:21)
```

Instead, we need to make sure that the `objectMode` option is set to `true`:

```
return new Transform({
  objectMode: true,
  transform (row, encoding, callback) { ... }
})
```

With that issue out of the way, we can create our `transform()` function. The two things we need to do are to convert the single “name” field into separate “firstName” and “lastName” fields, and to change the “dob” field to an “age” field. Both are easy with some simple string manipulation and date math:

02-async/24-transform-csv.js

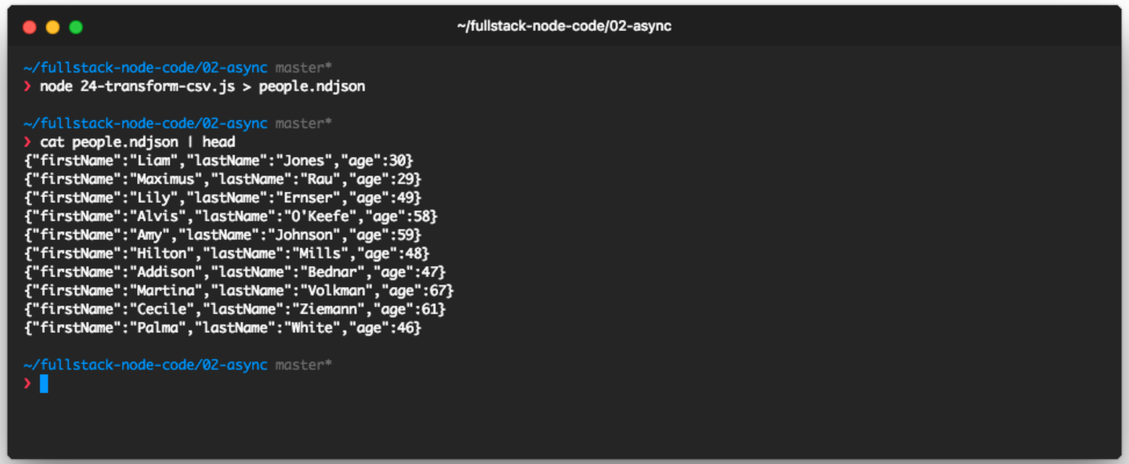
```
    transform (row, encoding, callback) {
      const [firstName, lastName] = row.name.split(' ')
      const age = Math.floor((Date.now() - new Date(row.dob)) / YEAR_MS)
      callback(null, {
        firstName,
        lastName,
        age
      })
    }
```

Now, when our transform stream emits events, they will be properly formatted and can be logged as JSON:

02-async/24-transform-csv.js

```
const fs = require('fs')
const csv = require('csv-parser')
const { Transform } = require('stream')

const YEAR_MS = 365 * 24 * 60 * 60 * 1000

fs.createReadStream('people.csv')
  .pipe(csv())
  .pipe(clean())
  .on('data', row => console.log(JSON.stringify(row)))

function clean () {
  return new Transform({
    objectMode: true,
    transform (row, encoding, callback) {
      const [firstName, lastName] = row.name.split(' ')
      const age = Math.floor((Date.now() - new Date(row.dob)) / YEAR_MS)
      callback(null, {
        firstName,
        lastName,
        age
      })
    }
  })
}
```

Now when we run this with `node 24-transform-csv.js > people.ndjson` our csv rows will be transformed and the newline delimited JSON will be written to `people.ndjson`:



```
~/fullstack-node-code/02-async

~/fullstack-node-code/02-async master*
› node 24-transform-csv.js > people.ndjson

~/fullstack-node-code/02-async master*
› cat people.ndjson | head
{"firstName":"Liam","lastName":"Jones","age":30}
{"firstName":"Maximus","lastName":"Rau","age":29}
{"firstName":"Lily","lastName":"Ernser","age":49}
{"firstName":"Alvis","lastName":"O'Keefe","age":58}
{"firstName":"Amy","lastName":"Johnson","age":59}
{"firstName":"Hilton","lastName":"Mills","age":48}
{"firstName":"Addison","lastName":"Bednar","age":47}
{"firstName":"Martina","lastName":"Volkman","age":67}
{"firstName":"Cecile","lastName":"Ziemann","age":61}
{"firstName":"Palma","lastName":"White","age":46}

~/fullstack-node-code/02-async master*
›
```

Our data in our csv file is transformed and converted to ndjson

Steams: Wrapping Up

In this section we've seen how to use streams to efficiently transform large amounts of data. In the real world, we'll typically receive some large file or have to deal with a large data source, and it can be infeasible to process it all at once. Sometimes we need to change the format (e.g. from csv to ndjson), and other times we need to clean or modify the data. In either case, transform streams are a great tool to have at our disposal.

Async Final Words

In this chapter we've explored many of the different async patterns available in Node.js. Not all of them are appropriate in all circumstances, but all of them are important to be familiar with.

As we begin to build more functionality into our applications and services, it's important to know how to be efficient and what tools we have available.

While callbacks and promises serve a similar purpose, it's important to be familiar with both so that we can use Node.js core APIs as well as third-party modules. Similarly, using `async/await` can make certain types of operations cleaner to write, but using `async` functions can impose other restrictions on our code, and it's great to have alternatives when it's not worth the trade-off.

Additionally, we've learned about event emitters and streams, two higher-level abstractions that allow us to work with multiple future actions and deal with data in a more efficient, continuous way.

All of these techniques will serve us well as we continue to build with Node.js.

MongoDB:-

1. Install MongoDB
2. **Data Modeling**
3. Query and Projection
4. **Aggregation**

PipelineIntroduction:-

1. Data
2. **Database**
3. **NoSQL**
4. **What is MongoDB**
5. Features of MongoDB
6. **How MongoDB works?**
7. **Database, Collection and Documents**

Introduction:-

1. Data:-

Data is information such as facts and numbers used to analyze something or make decisions. Computer data is information in a form that can be processed by a computer.

2. Database:-

A database is an organized collection of structured information, or data, typically storedelectronically in a computer system. A database is usually controlled by a database management system (DBMS).

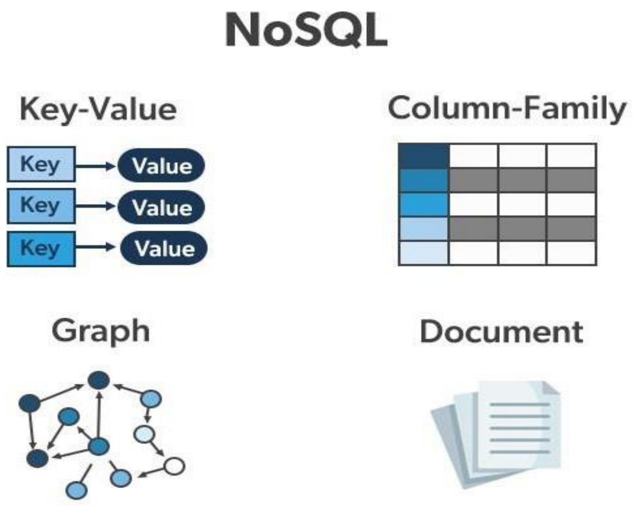
3. NoSQL:-

A database is a collection of structured data or information which is stored in a computer system and can be accessed easily. A database is usually managed by a Database Management System (DBMS).

NoSQL is a non-relational database that is used to store the data in the nontabular form. NoSQL stands for Not only SQL. The main types are documents, key-value, wide-column, and graphs

Types of NoSQL Database:

- Document-based databases
- Key-value stores
- Column-oriented databases
- Graph-based databases



1. Document-Based Database:

The document-based database is a nonrelational database. Instead of storing the data in rowsand columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications which means less translation is required to use these data in the applications. In theDocument database, the particular elements can be accessed by using the index value that is assigned for faster querying.

Collections are the group of documents that store documents that have similar contents. Not all the documents are in any collection as they require a similar schema because document databases have a flexible schema.

Key features of documents database:

- Flexible schema: Documents in the database has a flexible schema. It means the documents in the database need not be the same schema.
- Faster creation and maintenance: the creation of documents is easy and minimal maintenance is required once we create the document.

- No foreign keys: There is no dynamic relationship between two documents so documents can be independent of one another. So, there is no requirement for a foreign key in a document database.
- Open formats: To build a document we use XML, JSON, and others.

2. Key-Value Stores:

A key-value store is a nonrelational database. The simplest form of a NoSQL database is a key-value store. Every data element in the database is stored in key-value pairs. The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.

A key-value store is like a relational database with only two columns which is the key and the value.

Key features of the key-value store:

- Simplicity.
- Scalability.
- Speed.

3. Column Oriented Databases:

A column-oriented database is a non-relational database that stores the data in columns instead of rows. That means when we want to run analytics on a small number of columns, you canread those columns directly without consuming memory with the unwanted data.

Columnar databases are designed to read data more efficiently and retrieve the data with greaterspeed. A columnar database is used to store a large amount of data. Key features of columnar oriented database:

- Scalability.
- Compression.
- Very responsive.

4. Graph-Based databases:

Graph-based databases focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships.

Key features of graph database:

- In a graph-based database, it is easy to identify the relationship between the data by usingthe links.
- The Query's output is real-time results.
- The speed depends upon the number of relationships among the database elements.
- Updating data is also easy, as adding a new node or edge to a graph database is a straightforward task that does not require significant schema changes.

4. What is MongoDB?

MongoDB is a **document-oriented** NoSQL database system that provides high scalability, flexibility, and performance. Unlike standard relational databases, MongoDB stores data in a JSON document structure form. This makes it easy to operate with dynamic and unstructureddata and MongoDB is an open-source and cross-platform database System.

Why Use MongoDB?

Document Oriented Storage − Data is stored in the form of JSON documents.

- **Index on any attribute**: Indexing in MongoDB allows for faster data retrieval by creatinga searchable structure on selected attributes, optimizing query performance.
- **Replication and high availability**: MongoDB's replica sets ensure data redundancy bymaintaining multiple copies of the data, providing fault tolerance and continuous availability even in case of server failures.
- **Auto-Sharding**: Auto-sharding in MongoDB automatically distributes data across multiple servers, enabling horizontal scaling and efficient handling of large datasets.
- **Big Data and Real-time Application**: When dealing with massive datasets or applicationsrequiring real-time data updates, MongoDB's flexibility and scalability prove advantageous.
- **Rich queries**: MongoDB supports complex queries with a variety of operators, allowingyou to retrieve, filter, and manipulate data in a flexible and powerful manner.
- **Fast in-place updates**: MongoDB efficiently updates documents directly in their place,minimizing data movement and reducing write overhead.
- **Professional support by MongoDB**: MongoDB offers expert technical support and resources to help users with any issues or challenges they may encounter during theirdatabase operations.

Internet of Things (IoT) Applications: **Storing and analyzing sensor data with its diverseformats often aligns well with MongoDB's document structure.**

Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

- **Big Data**: If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and shardingyour database.
- **Unstable Schema**: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.
- **Distributed data** Since multiple copies of data are stored across different servers,recovery of data is instant and safe even if there is a hardware failure.

Language Support by MongoDB:

MongoDB currently provides official driver support for all popular programming languageslike C, C++, Rust, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.

5. Features of MongoDB –

- **Schema-less Database:** It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words,in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.

- **Document Oriented:** In MongoDB, all the data stored in the documents instead of tables like in RDBMS. In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. Andeach document contains its unique object id.
- **Indexing:** In MongoDB database, every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool ofthe data. If the data is not indexed, then database search each document with the specified query which takes lots of time and not so efficient.
- **Scalability:** MongoDB provides horizontal scalability with the help of sharding. Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shardsthat reside across many physical servers. It will also add new machines to a running database.
- **Replication:** MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different serverso that if one server fails, then the data is retrieved from another server.
- **Aggregation:** It allows to perform operations on the grouped data and get a single result or computed result. It is similar to the SQL GROUPBY clause. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregationmethods
- **High Performance:** The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.

Advantages of MongoDB :

- It is a schema-less NoSQL database. You need not to design the schema of the databasewhen you are working with MongoDB.
- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous data.
- It provides high performance, availability, scalability.
- It supports Geospatial efficiently.
- It is a document oriented database and the data is stored in BSON documents.
- It also supports multiple document ACID transition(string from MongoDB 4.0).
- It does not require any SQL injection.
- It is easily integrated with Big Data Hadoop

Disadvantages of MongoDB :

- It uses high memory for data storage.
- You are not allowed to store more than 16MB data in the documents.

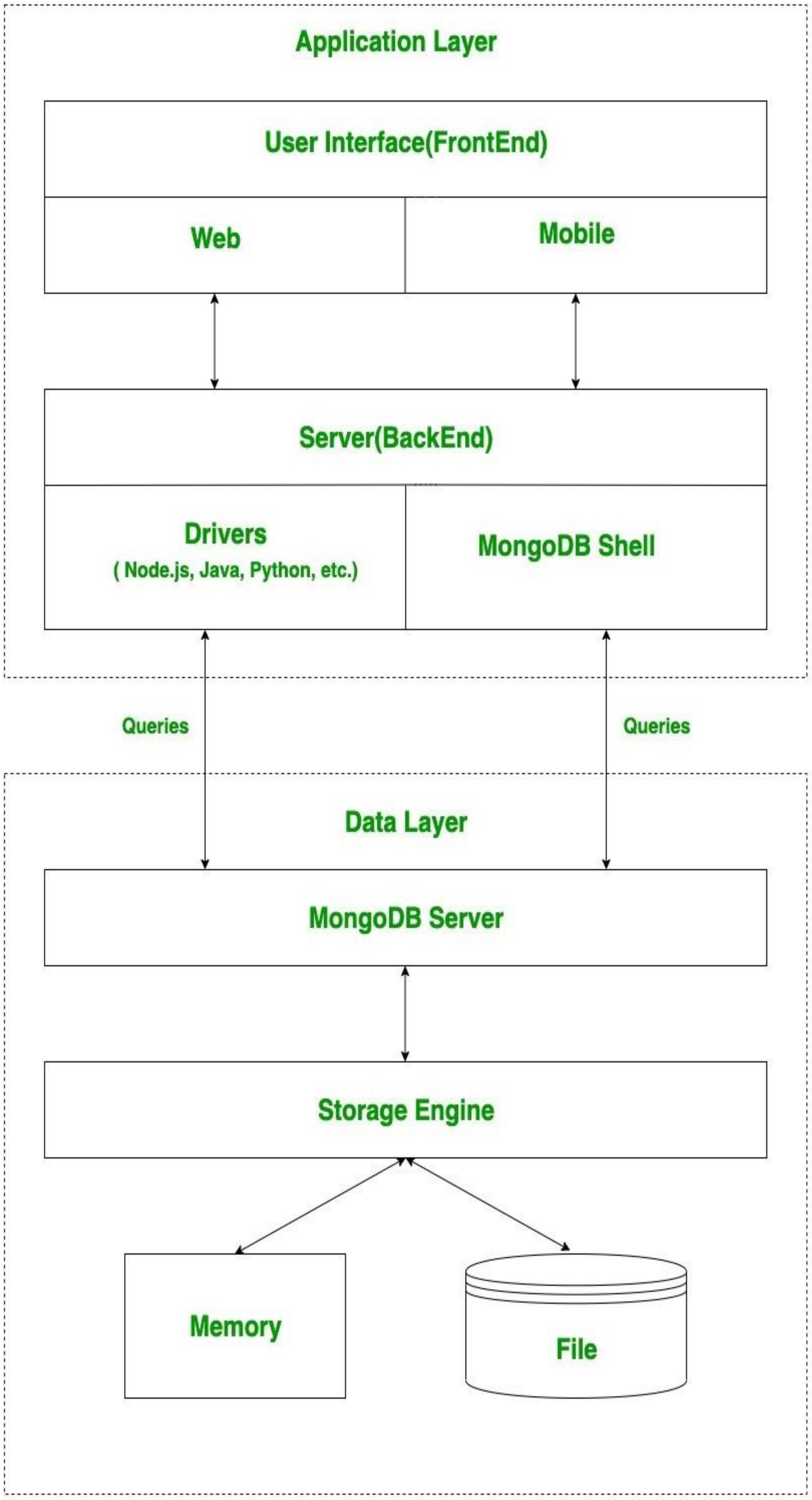
- The nesting of data in BSON is also limited you are not allowed to nest data more than 100levels.

6. How MongoDB works ?

MongoDB is an open-source document-oriented database. It is used to store a larger amount ofdata and also allows you to work with that data. MongoDB is not based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data, that's why known as NoSQL database. Here, the term 'NoSQL' means 'non- relational'. The format of storage is called BSON (similar to JSON format). Now, let's see how actually this MongoDB works? But before proceeding to its working, first, let's discuss some important parts of MongoDB –

- **Drivers:** Drivers are present on your server that are used to communicate with MongoDB.The drivers support by the MongoDB are C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid.
- **MongoDB Shell:** MongoDB Shell or mongo shell is an interactive JavaScript interface forMongoDB. It is used for queries, data updates, and it also performs administrative operations.
- **Storage Engine:** It is an important part of MongoDB which is generally used to manage how data is stored in the memory and on the disk. MongoDB can have multiple search engines. You are allowed to use your own search engine and if you don't want to use your own search engine you can use the default search engine, known as *WiredTiger Storage Engine* which is an excellent storage engine, it efficiently works with your data like reading,writing, etc.

Working of MongoDB -The following image shows how the MongoDB works:



MongoDB work in two layers –

- **Application Layer** and
- **Data layer**

Application Layer is also known as the **Final Abstraction Layer**, it has two-parts, first is a **Frontend (User Interface)** and the second is **Backend (server)**. The frontend is the place
where the user uses MongoDB with the help of a Web or Mobile. This web and mobile includeweb pages, mobile applications, android default applications, IOS applications, etc. The backend contains a server which is used to perform server-side logic and also contain drivers ormongo shell to interact with MongoDB server with the help of queries.
These queries are sent to the MongoDB server present in the **Data Layer**. Now, the MongoDB server receives the queries and passes the received queries to the storage engine. MongoDB server itself does not directly read or write the data to the files or disk or memory. After passingthe received queries to the storage engine, the storage engine is responsible to read or write the data in the files or memory basically it manages the data.

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table- like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON (similar to JSON format).
A simple MongoDB document Structure:

```
{
  title:
  'Geeksforgeeks',by:
  'Harshit Gupta',
  url: 'https://www.geeksforgeeks.org',
  type: 'NoSQL'
}
```

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today's real-world highly growing applications. **Modern applications are more networked, social and interactive than ever**. Applications are storing more and more data and are accessing it at higher rates.
Relational Database Management System(RDBMS) **is not the correct choice when it comesto handling big data by the virtue of their design since they are not horizontally scalable**.If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

Getting Started

After you install MongoDB, you can see all the installed file inside C:\ProgramFiles\MongoDB\ (default location). In the C:\Program Files\MongoDB\Server\3.2\bin directory, there are a bunch of executables and a short-description about them would be:

mongo: The Command Line Interface to interact with the db.
mongod: This is the database. Sets up the server.
mongodump: It dumps out the Binary of the Database(BSON)
mongoexport: Exports the document to Json, CSV format
mongoimport: To import some data into the DB.
mongorestore: to restore anything that you've exported.
mongostat: Statistics of databases

7. Database, Collection and Documents:-

Database

- Database is a container for collections.
- Each database gets its own set of files.
- A single MongoDB server can has multiple databases.

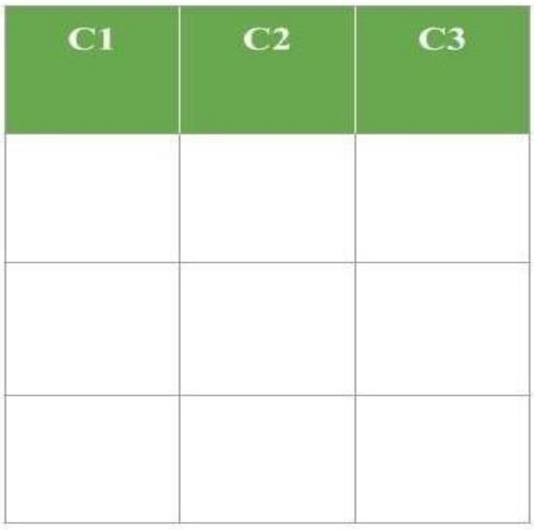
Collection

- Collection is a group of documents.
- Collection is equivalent to RDBMS table.
- A collection consist inside a single database.
- Collections do not enforce a schema.
- A Collection can have different fields within a Documents.
-

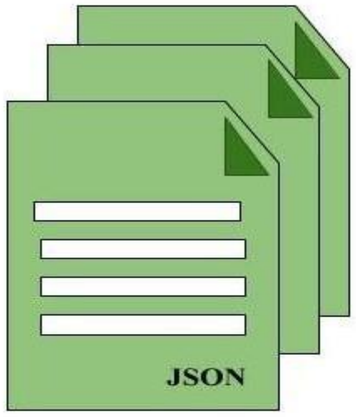
Document:-

A document database has information retrieved or stored in the form of a document or other words semi-structured database. Since they are non-relational, so they are often referred to asNoSQL data.

The document database fetches and accumulates data in forms of key-value pairs but here, thevalues are called as Documents. A document can be stated as a complex data structure. Document here can be a form of text, arrays, strings, JSON, XML, or any such format. The useof nested documents is also very common. It is very effective as most of the data created is usually in the form of JSON and is unstructured.

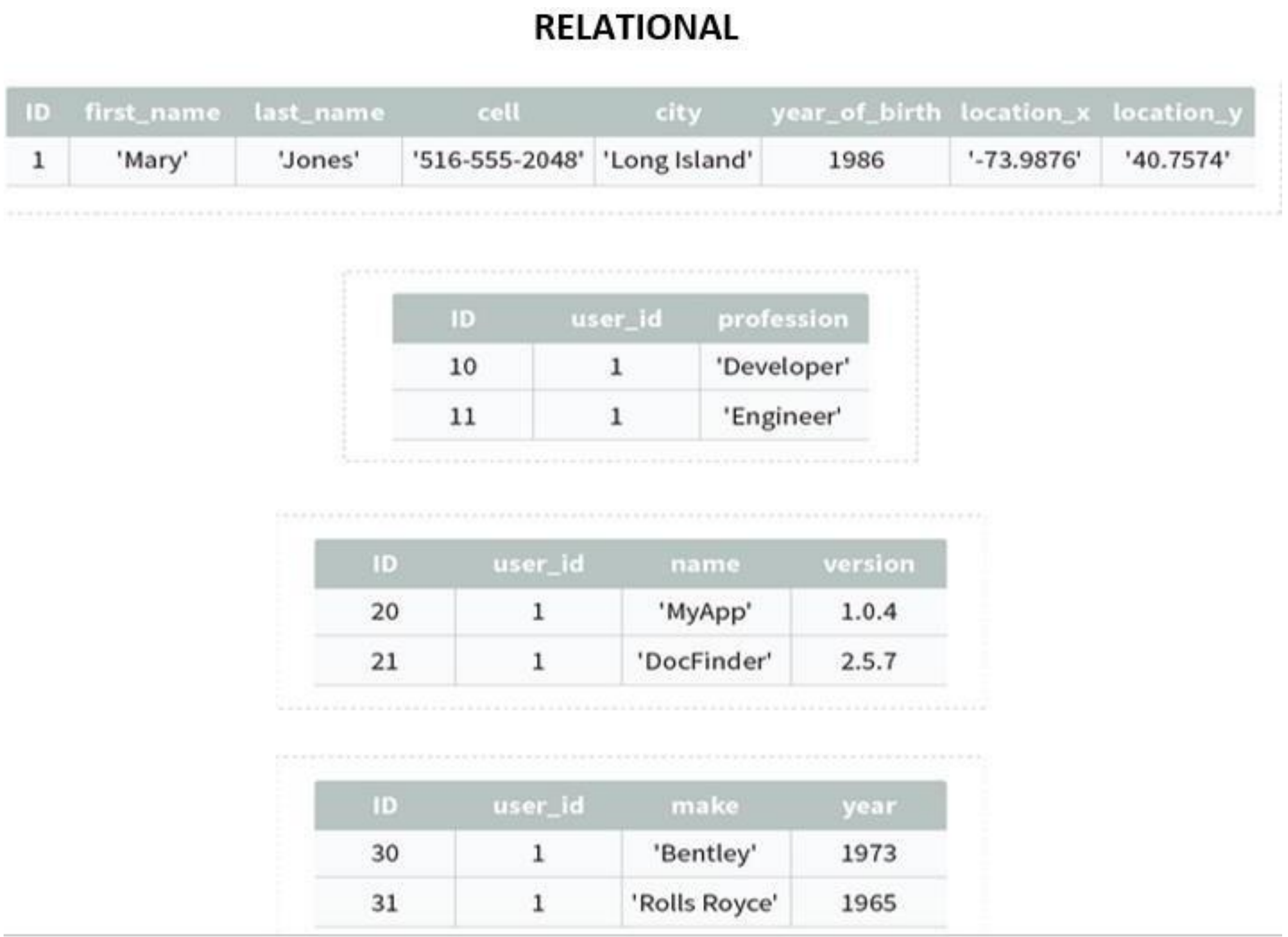


Relational Data Model



Document Store Model

Consider the below example that shows a sample database stored in both Relational and Document Database



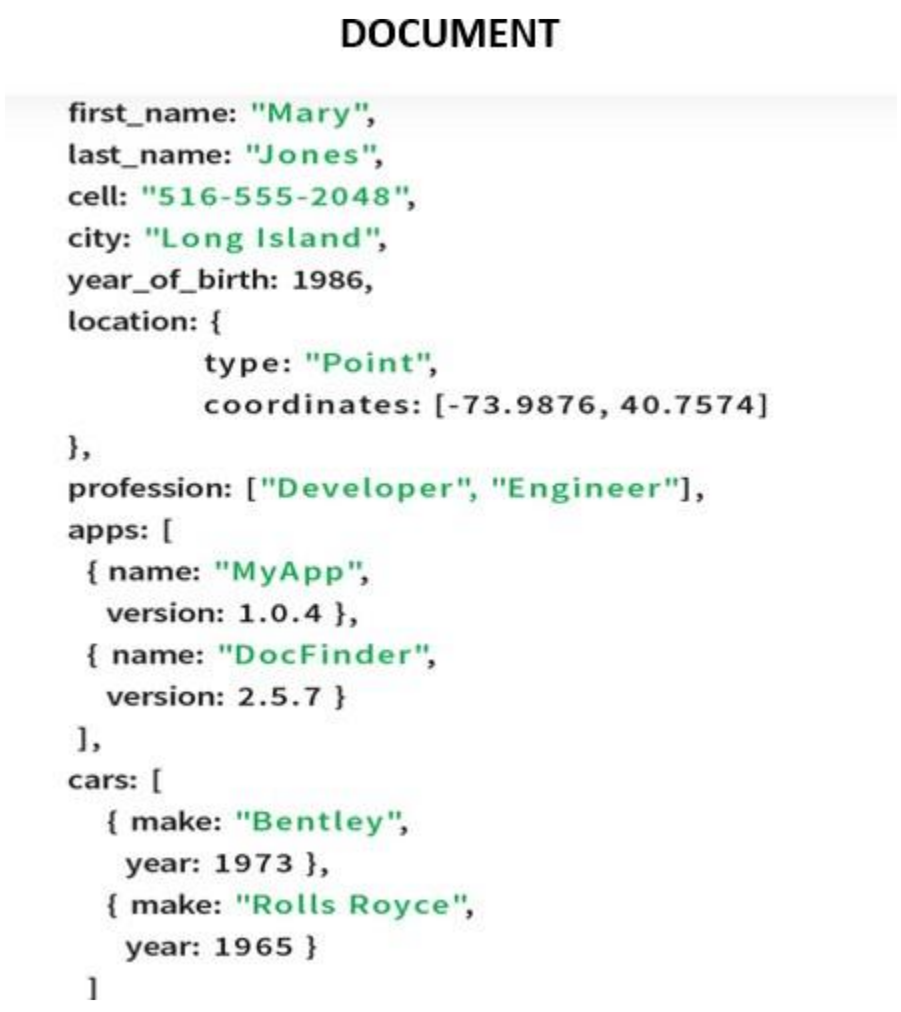
RELATIONAL

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

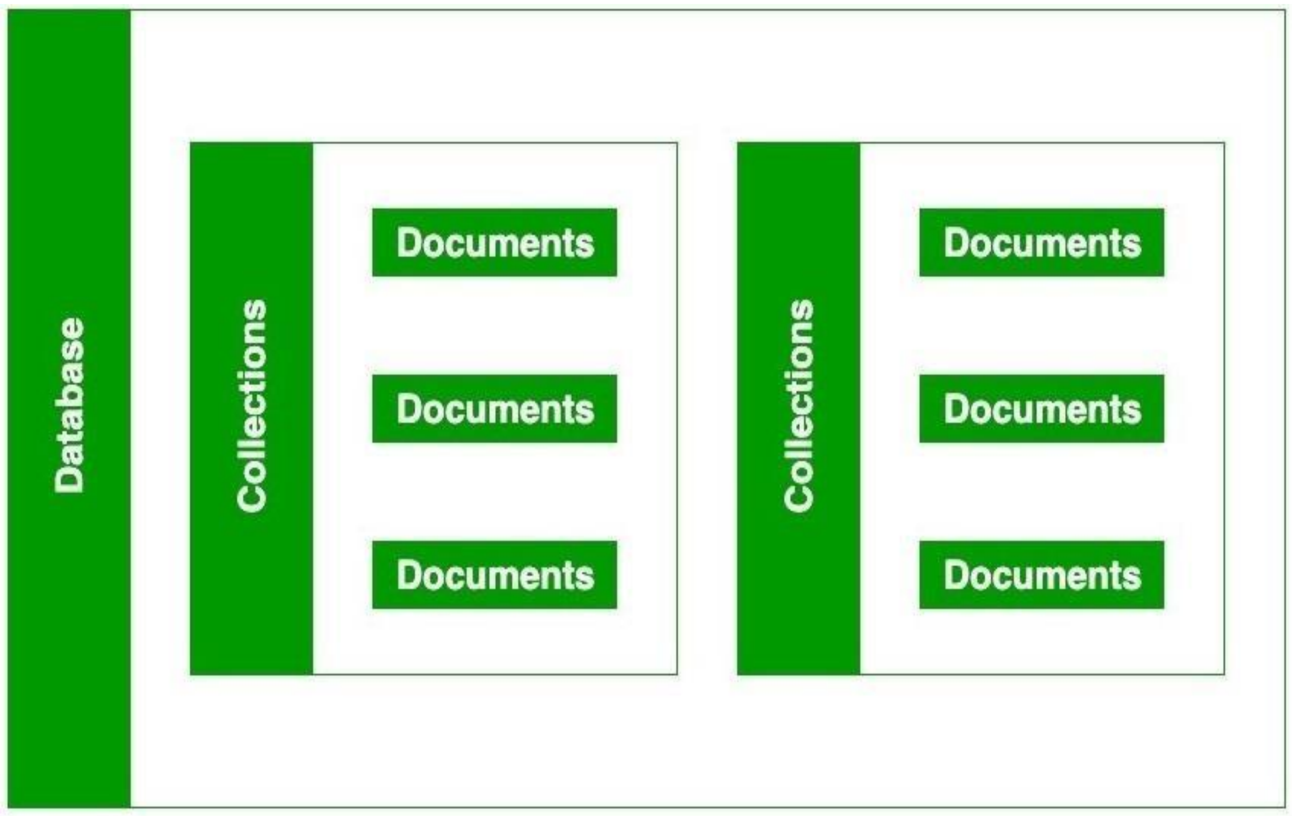


DOCUMENT

```
    first_name: "Mary",
    last_name: "Jones",
    cell: "516-555-2048",
    city: "Long Island",
    year_of_birth: 1986,
    location: {
              type: "Point",
              coordinates: [-73.9876, 40.7574]
    },
    profession: ["Developer", "Engineer"],
    apps: [
     { name: "MyApp",
       version: 1.0.4 },
     { name: "DocFinder",
       version: 2.5.7 }
     ],
    cars: [
      { make: "Bentley",
        year: 1973 },
      { make: "Rolls Royce",
        year: 1965 }
     ]
```

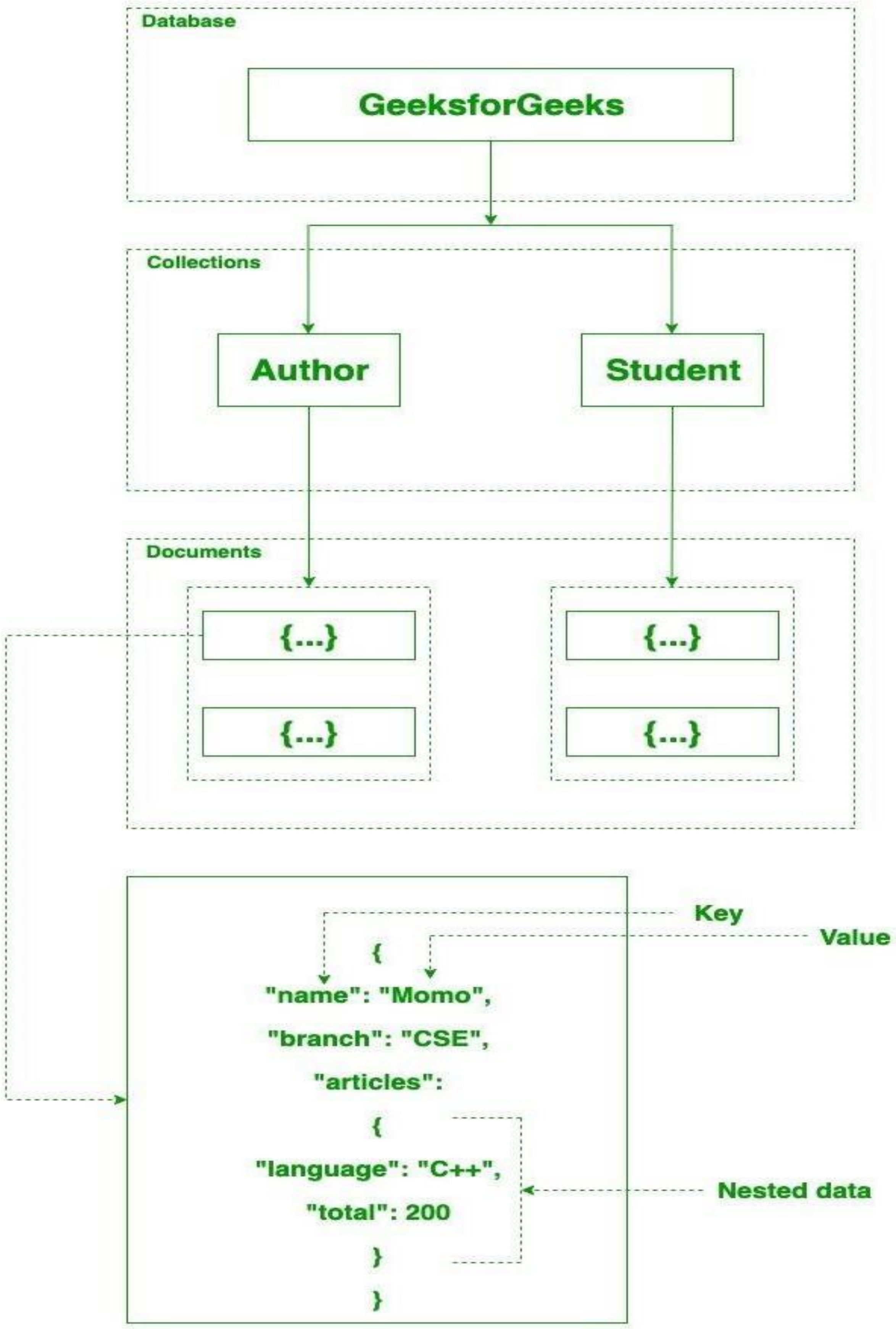
How it works ?

Now, we will see how actually thing happens behind the scene. As we know that MongoDB is adatabase server and the data is stored in these databases. Or in other words, MongoDB environment gives you a server that you can start and then create multiple databases on it using MongoDB.
Because of its NoSQL database, the data is stored in the collections and documents. Hence the database, collection, and documents are related to each other as shown below:



- The MongoDB database contains collections just like the MYSQL database contains tables.You are allowed to create multiple databases and multiple collections.
- Now inside of the collection we have documents. These documents contain the data we wantto store in the MongoDB database and a single collection can contain multiple documents and you are schema-less means it is not necessary that one document is similar to another.
- The documents are created using the fields. Fields are key-value pairs in the documents, it isjust like columns in the relation database. The value of the fields can be of any BSON data types like double, string, boolean, etc.
- The data stored in the MongoDB is in the format of BSON documents. Here, BSON stands for Binary representation of JSON documents. Or in other words, in the backend, the MongoDB server converts the JSON data into a binary form that is known as BSON and thisBSON is stored and queried more efficiently.
- In MongoDB documents, you are allowed to store nested data. This nesting of data allows you to create complex relations between data and store them in the same document which makes the working and fetching of data extremely efficient as compared to SQL. In SQL, you need to write complex joins to get the data from table 1 and table 2. The maximum sizeof the BSON document is 16MB.

NOTE: In MongoDB server, you are allowed to run multiple databases.
For example, we have a database named GeeksforGeeks. Inside this database, we have two collections and in these collections we have two documents. And in these documents we storeour data in the form of fields. As shown in the below image:


Database
GeeksforGeeks
Collections
Author
Student
Documents
{...}
{...}
{...}
{...}
Key
Value
{
"name": "Momo",
"branch": "CSE",
"articles":
{
"language": "C++",
"total": 200
}
}
Nested data

How mongoDB is different from RDBMS ?

Some major differences in between MongoDB and the RDBMS are as follows:

MongoDB	RDBMS
It is a non-relational and document-oriented database.	It is a relational database.
It is suitable for hierarchical data storage.	It is not suitable for hierarchical data storage.
It has a dynamic schema.	It has a predefined schema.
It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).	It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).
In terms of performance, it is much fasterthan RDBMS.	In terms of performance, it is slower than MongoDB.

1.Install MongoDB

- **There are 3 ways to install and use MongoDB**
    1. Community Server(free and open source. after download use local system)
    2. Visual Studio Extension
    3. MongoDB Atlas(cloud hosted DB offered by MONGO DB company)

1. Let's install MongoDB on our machines(Windows)

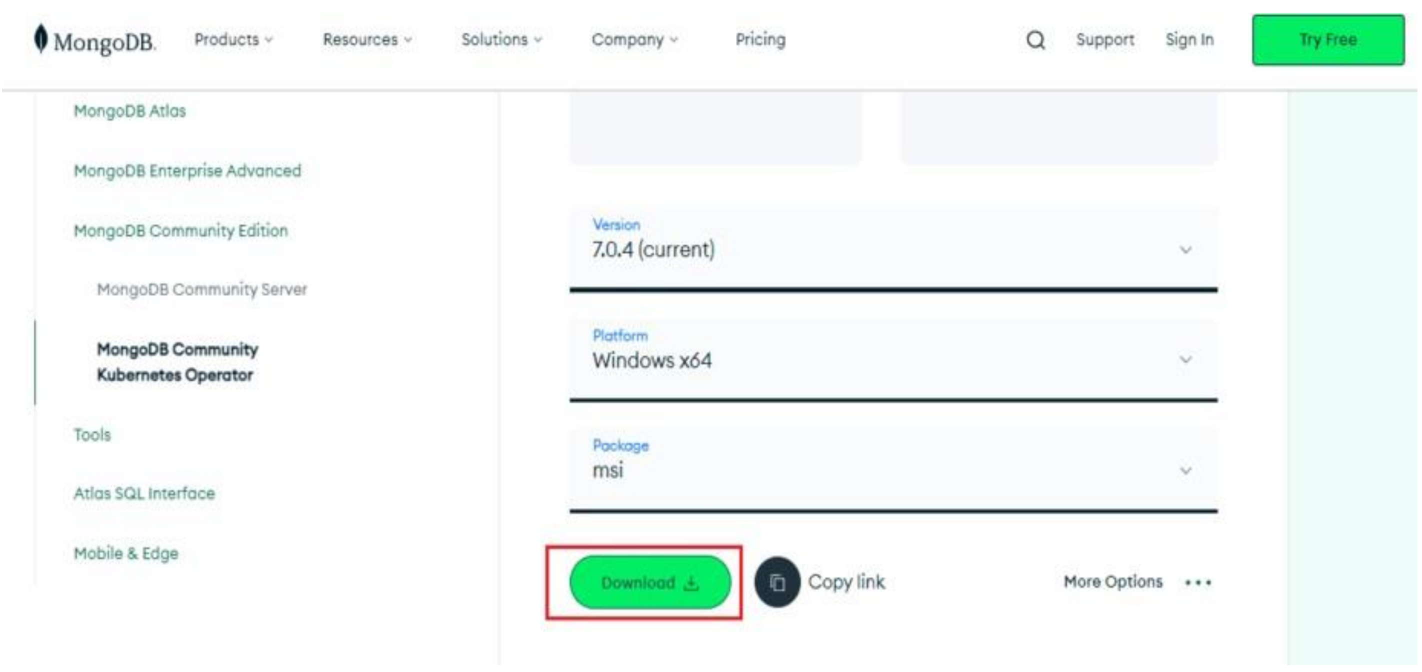
- Visit official website: http://mongodb.com
- Download the latest stable version from Community Server
- The Community server will also install the following apps
    a. Community Server
    b. Compass-GUI Tool for MongoDB

Install MongoDB on Windows using MSI Requirements to Install MongoDB on Windows

- **MongoDB 4.4** and later only **support 64-bit versions** of Windows.
- MongoDB 7.0 Community Edition supports the following **64-bit versions** of Windows onx86_64 architecture:
    - **Windows Server 2022**
    - **Windows Server 2019**
    - **Windows 11**

To install MongoDB on windows, first, download the MongoDB server and then install the MongoDB shell. The Steps below explain the installation process in detail and provide the required resources for the smooth **download and install MongoDB**.

Step 1: Go to the MongoDB Download Center to download the MongoDB Community Server.



Version
7.0.4 (current)

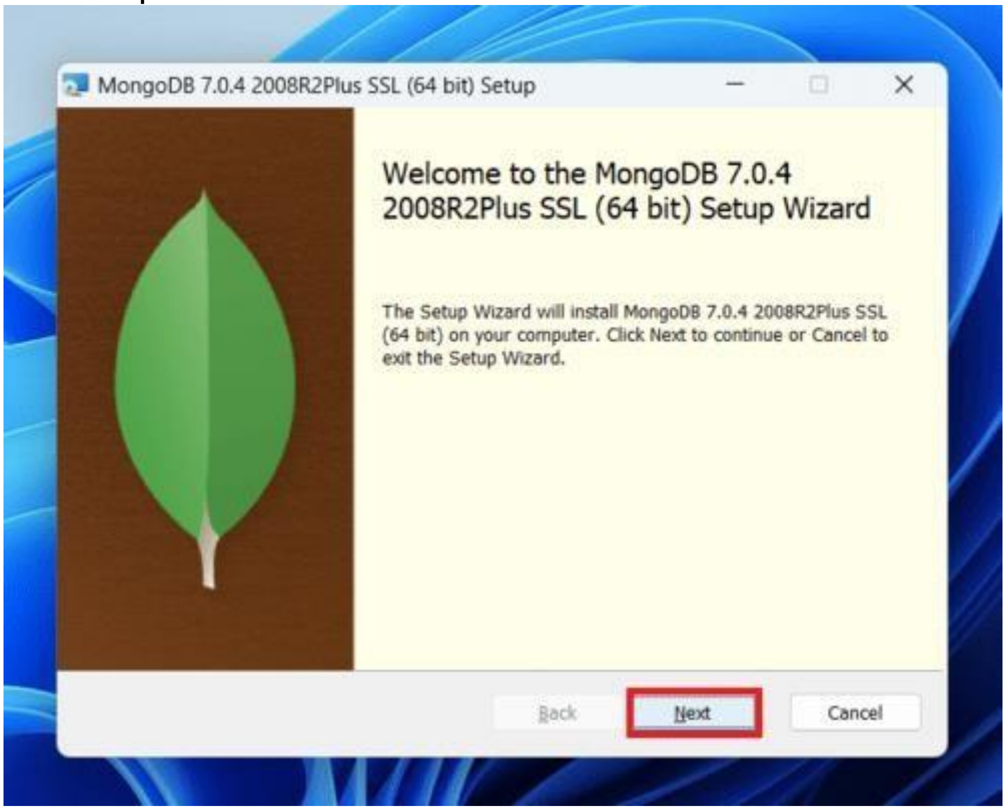
Platform
Windows x64

Package
msi

Here, You can select any version, Windows, and package according to your requirement. ForWindows, we need to choose:

- Version: 7.0.4
- OS: Windows x64
- Package: msi

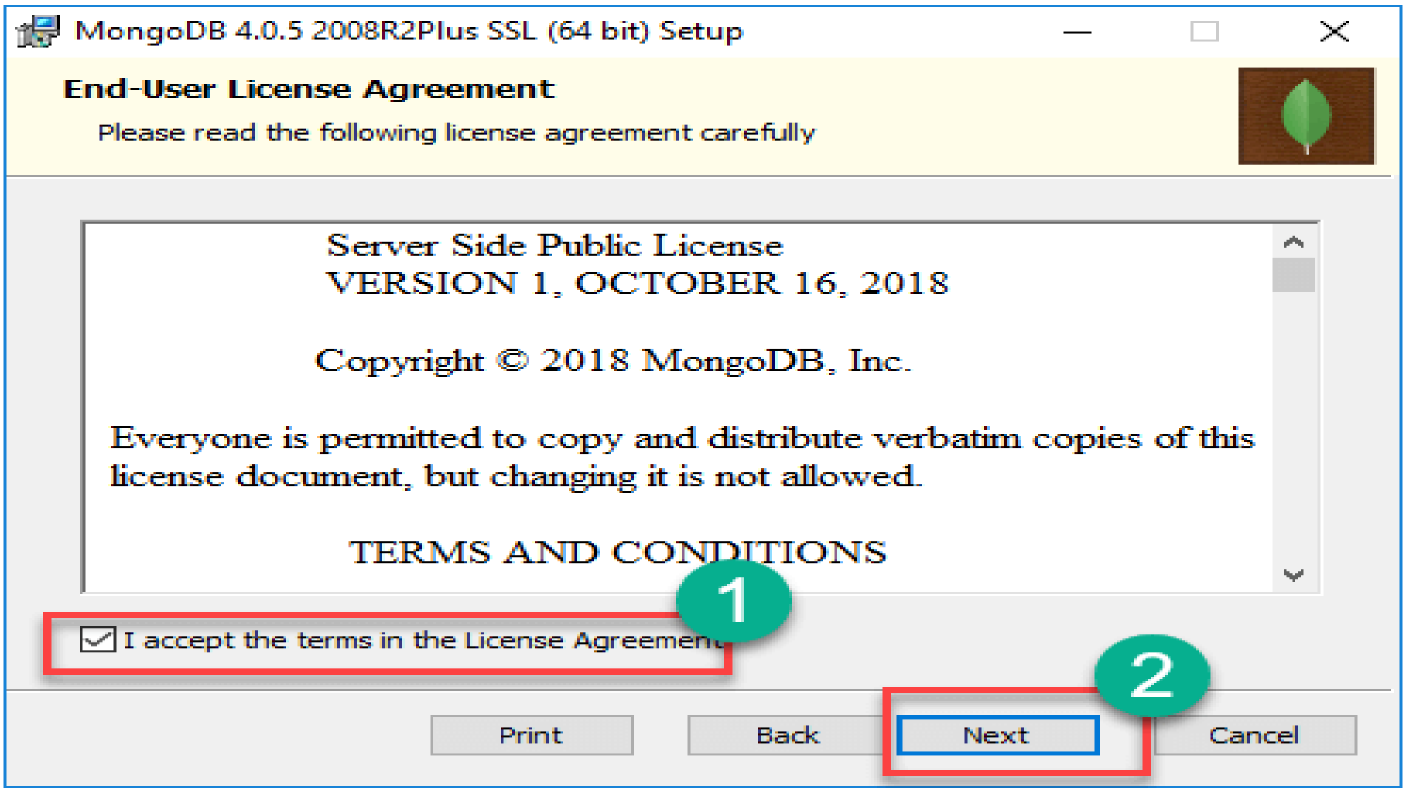
Step 2: When the download is complete open the msi file and click the *next button* in thestartup screen:


MongoDB 7.0.4 2008R2Plus SSL (64 bit) Setup

Welcome to the MongoDB 7.0.4 2008R2Plus SSL (64 bit) Setup Wizard

The Setup Wizard will install MongoDB 7.0.4 2008R2Plus SSL (64 bit) on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Step 3: Now accept the End-User License Agreement and click the next button:


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

End-User License Agreement

Please read the following license agreement carefully

Server Side Public License
VERSION 1, OCTOBER 16, 2018

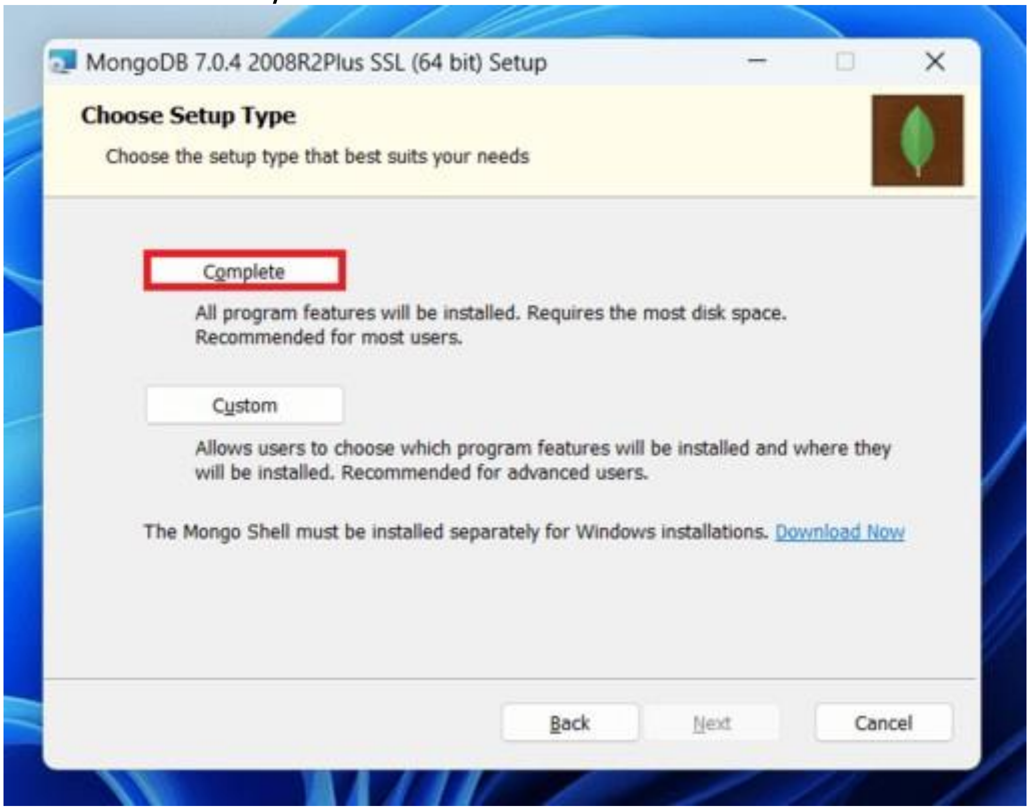
Copyright © 2018 MongoDB, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

TERMS AND CONDITIONS

✓ I accept the terms in the License Agreement

Step 4: Now select the *complete option* to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the *Custom option:*



MongoDB 7.0.4 2008R2Plus SSL (64 bit) Setup

Choose Setup Type

Choose the setup type that best suits your needs

Complete

All program features will be installed. Requires the most disk space. Recommended for most users.

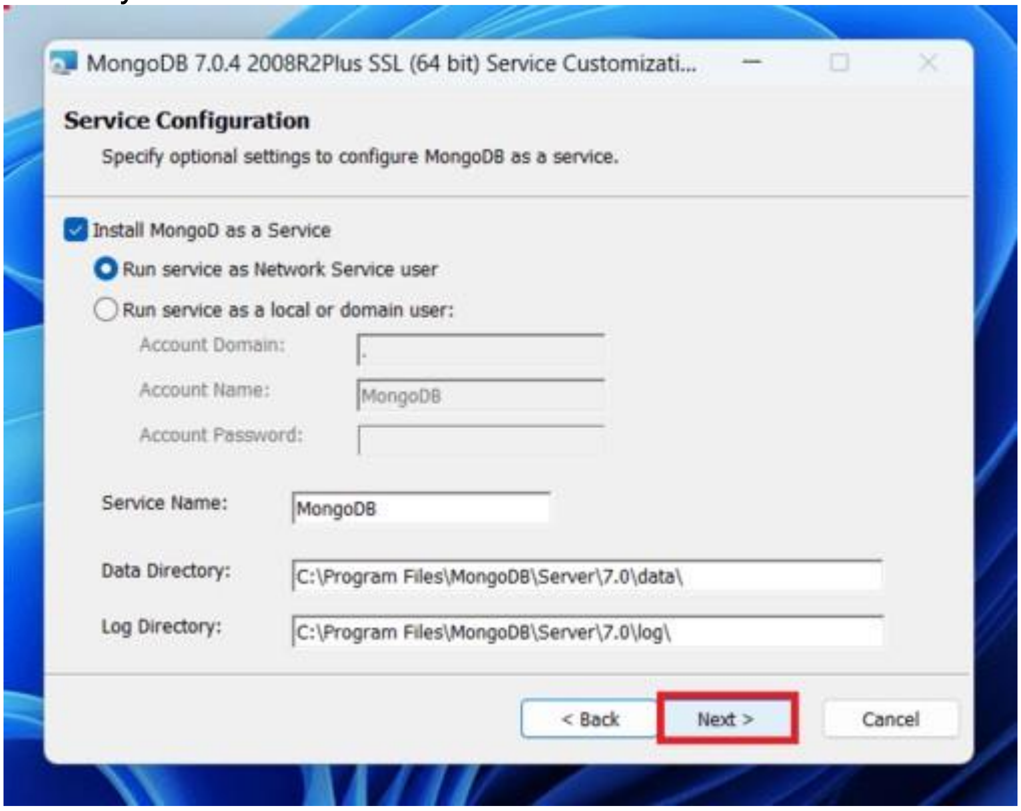
Custom

Allows users to choose which program features will be installed and where they will be installed. Recommended for advanced users.

The Mongo Shell must be installed separately for Windows installations. Download Now

Back | Next | Cancel

Step 5: Select "Run service as Network Service user" and copy the path of the data directory.Click Next:



MongoDB 7.0.4 2008R2Plus SSL (64 bit) Service Customizati...

Service Configuration

Specify optional settings to configure MongoDB as a service.

[✓] Install MongoD as a Service

[●] Run service as Network Service user

[ ] Run service as a local or domain user:

Account Domain: .

Account Name: MongoDB

Account Password:

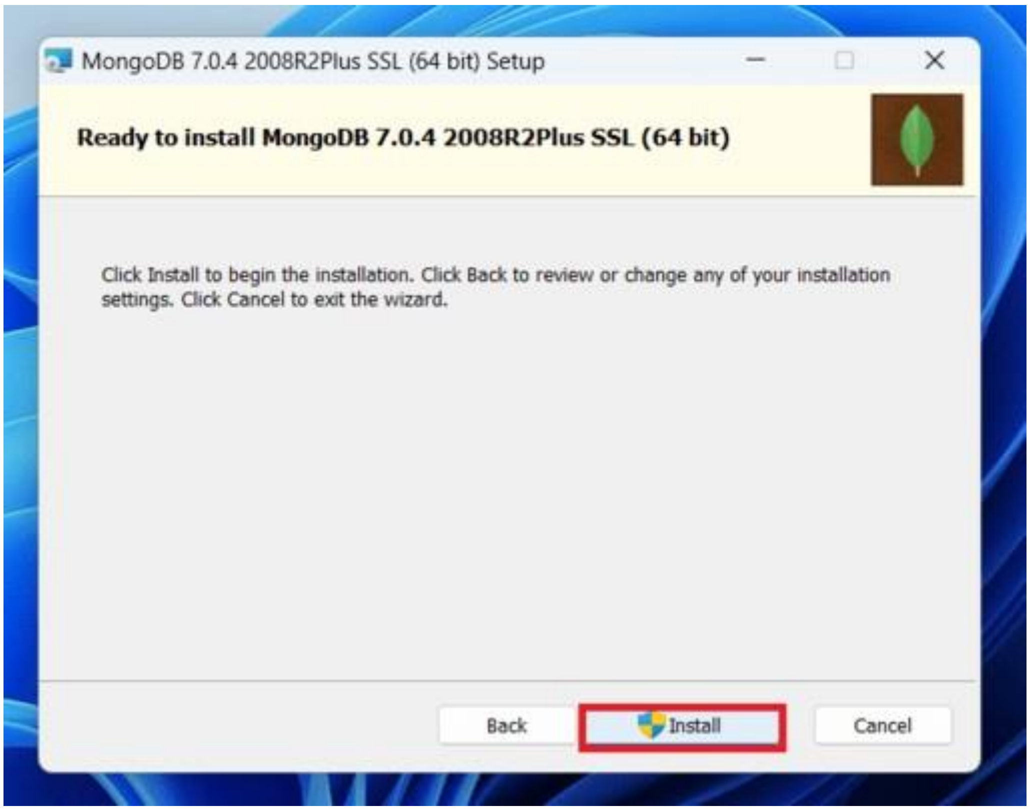
Service Name: MongoDB

Data Directory: C:\Program Files\MongoDB\Server\7.0\data\

Log Directory: C:\Program Files\MongoDB\Server\7.0\log\

< Back | Next > | Cancel

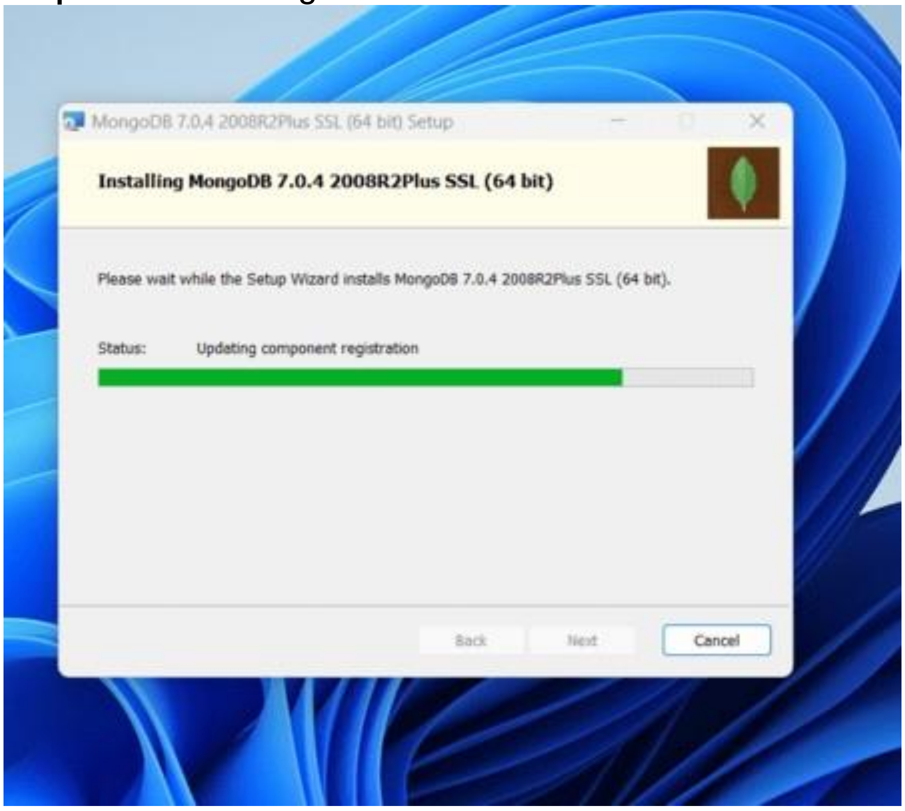
Step 6: Click the *Install button* to start the MongoDB installation process:


MongoDB 7.0.4 2008R2Plus SSL (64 bit) Setup

Ready to install MongoDB 7.0.4 2008R2Plus SSL (64 bit)

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

Step 7: After clicking on the install button installation of MongoDB begins:

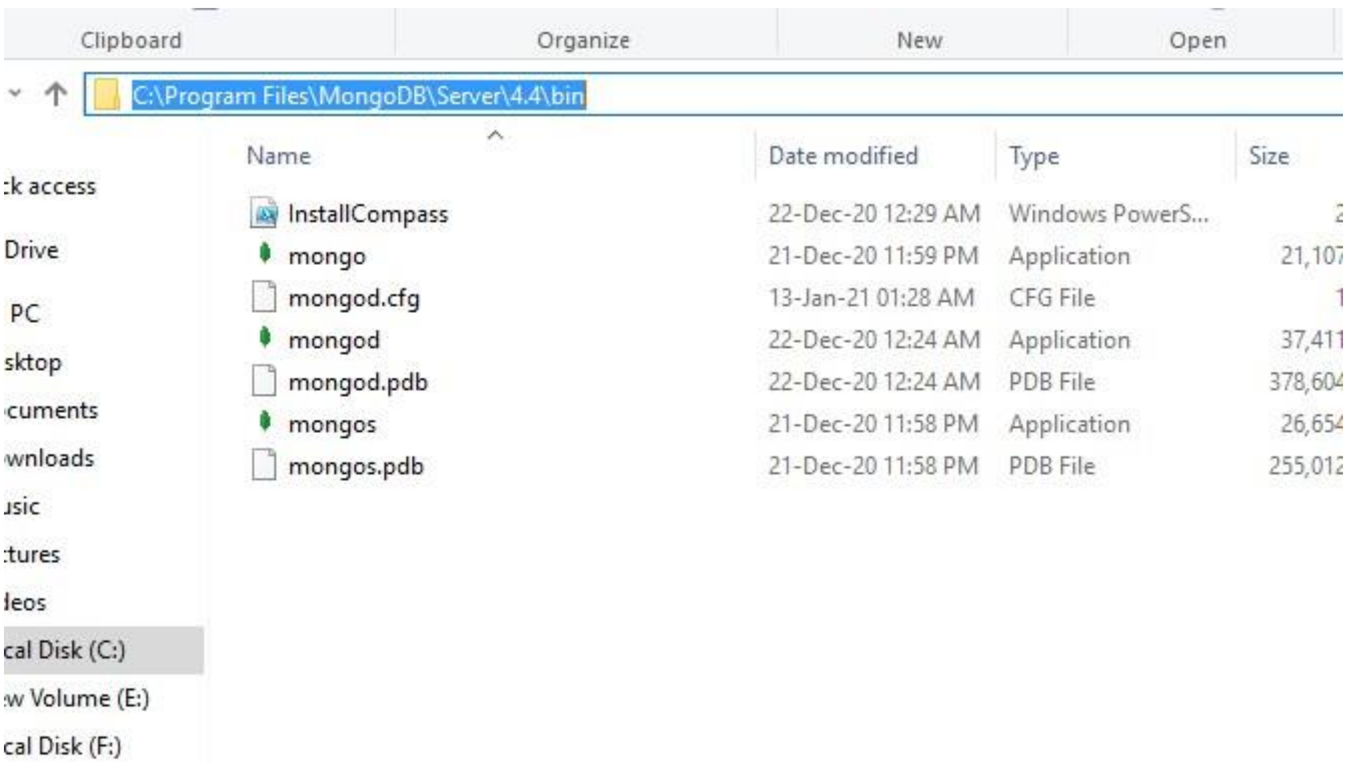

MongoDB 7.0.4 2008R2Plus SSL (64 bit) Setup

Installing MongoDB 7.0.4 2008R2Plus SSL (64 bit)

Please wait while the Setup Wizard installs MongoDB 7.0.4 2008R2Plus SSL (64 bit).

Status: Updating component registration

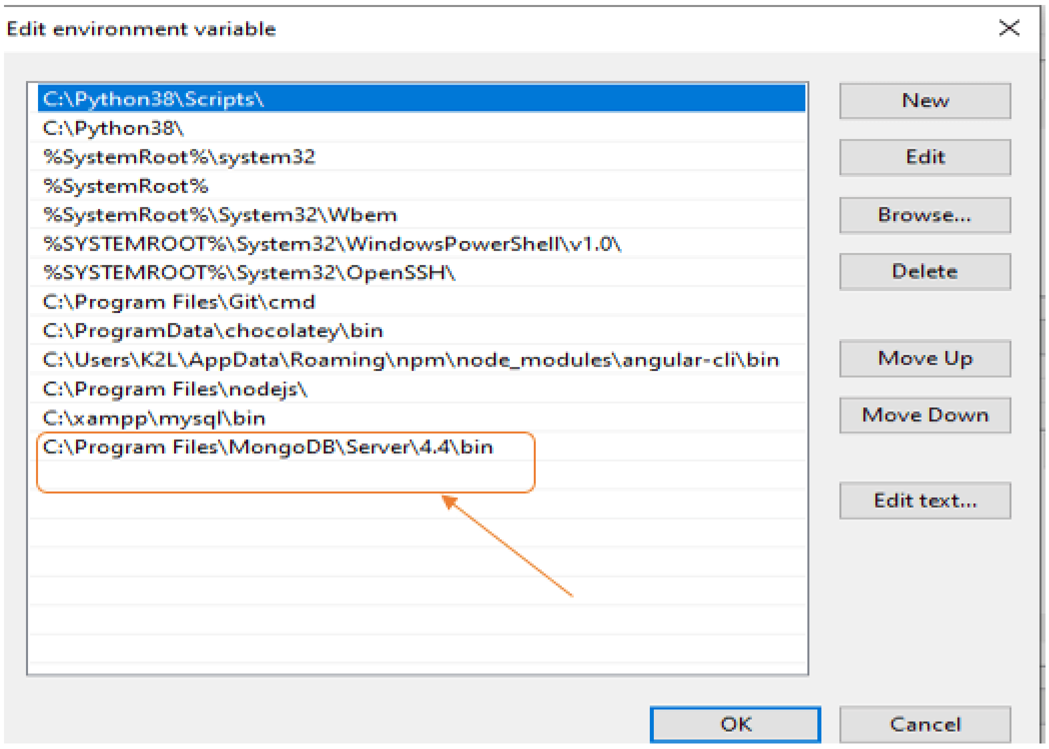
Step 8: Now click the ***Finish button*** to complete the MongoDB installation process:
Step 9: Now we go to the location where MongoDB installed in step 5 in your system and copythe bin path:


Clipboard Organize New Open

C:\Program Files\MongoDB\Server\4.4\bin

Name	Date modified	Type	Size
InstallCompass	22-Dec-20 12:29 AM	Windows PowerS...	2
mongo	21-Dec-20 11:59 PM	Application	21,107
mongod.cfg	13-Jan-21 01:28 AM	CFG File	1
mongod	22-Dec-20 12:24 AM	Application	37,411
mongod.pdb	22-Dec-20 12:24 AM	PDB File	378,604
mongos	21-Dec-20 11:58 PM	Application	26,654
mongos.pdb	21-Dec-20 11:58 PM	PDB File	255,012

Step 10: Now, to create an environment variable open system properties << Environment Variable << System variable << path << Edit Environment variable and paste the copied link toyour environment system and click Ok:


Edit environment variable

```
C:\Python38\Scripts\
C:\Python38\
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\Git\cmd
C:\ProgramData\chocolatey\bin
C:\Users\K2L\AppData\Roaming\npm\node_modules\angular-cli\bin
C:\Program Files\nodejs\
C:\xampp\mysql\bin
C:\Program Files\MongoDB\Server\4.4\bin
```

Step 11: After setting the environment variable, we will run the MongoDB server, i.e. mongod. So, open the command prompt and run the following command:

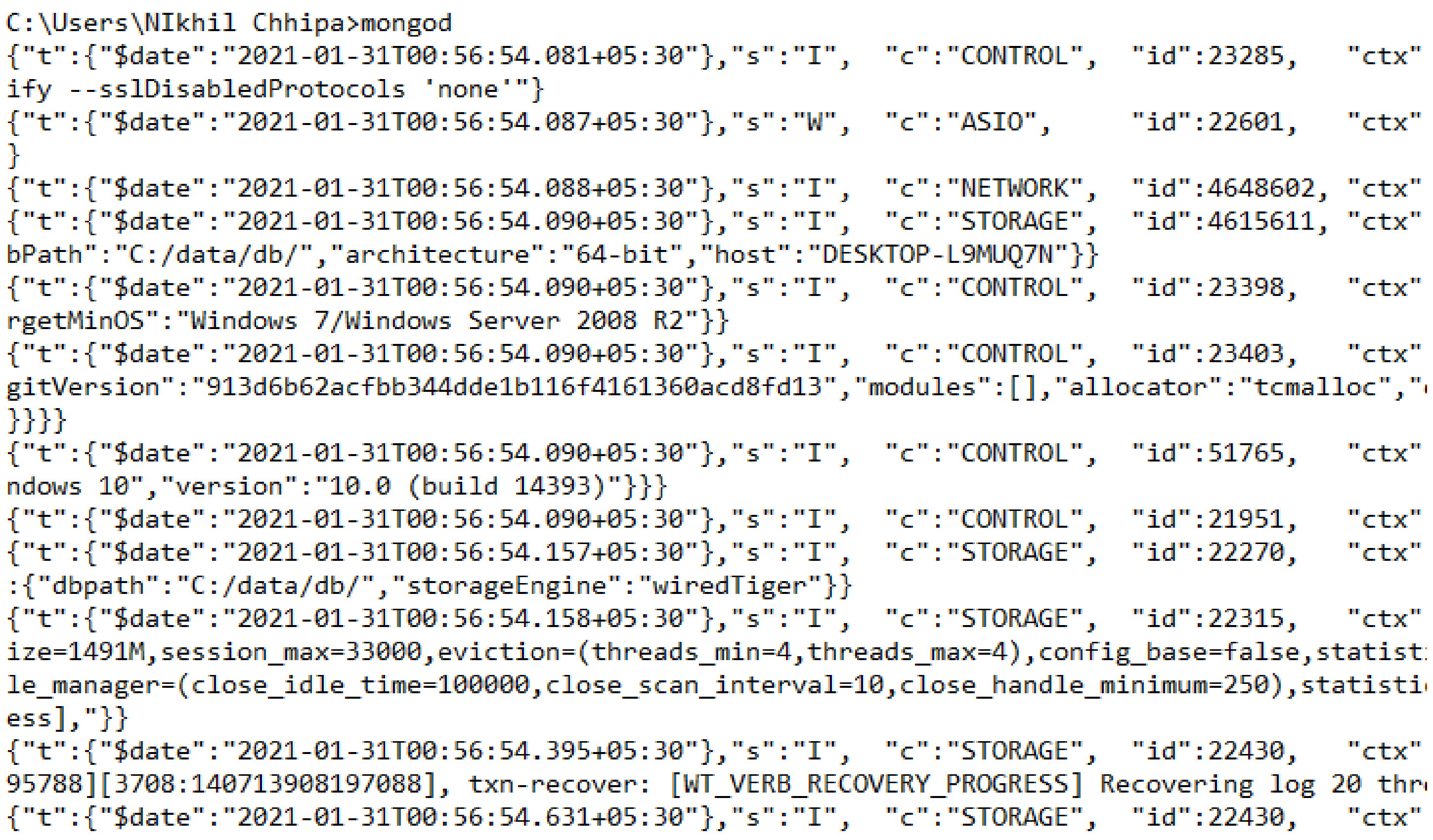
```
mongod
```

When you run this command you will get an error i.e. *C:/data/db/ not found*.

Step 12: Now, Open C drive and create a folder named “data” inside this folder create another folder named “db”. After creating these folders. Again open the command prompt and run the following command:

mongod

Now, this time the MongoDB server(i.e., mongod) will run successfully.

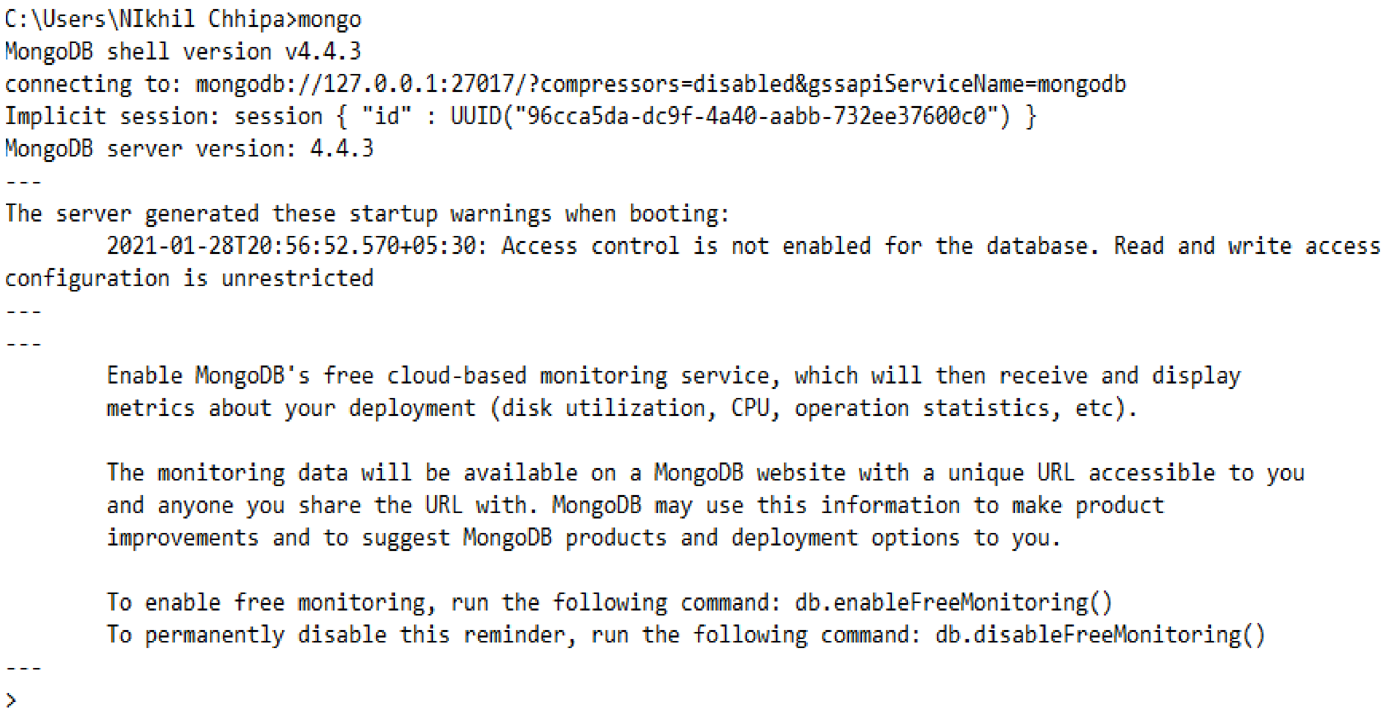


```
C:\Users\NIkhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W",  "c":"ASIO",     "id":22601,   "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23398,   "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx"
gitVersion":"913d6b62acfbb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc","
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx"
ndows 10","version":"10.0 (build 14393)"}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I",  "c":"STORAGE",  "id":22270,   "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
```

Run mongo Shell

Step 13: Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window and open a new command prompt window and write **mongo.** Now, our mongo shell will successfully connect to the mongod.

Important Point: Please do not close the mongod window if you close this window your server will stop working and it will not able to connect with the mongo shell.



```
C:\Users\NIkhil Chhipa>mongo
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("96cca5da-dc9f-4a40-aabb-732ee37600c0") }
MongoDB server version: 4.4.3
---
The server generated these startup warnings when booting:
        2021-01-28T20:56:52.570+05:30: Access control is not enabled for the database. Read and write access
configuration is unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

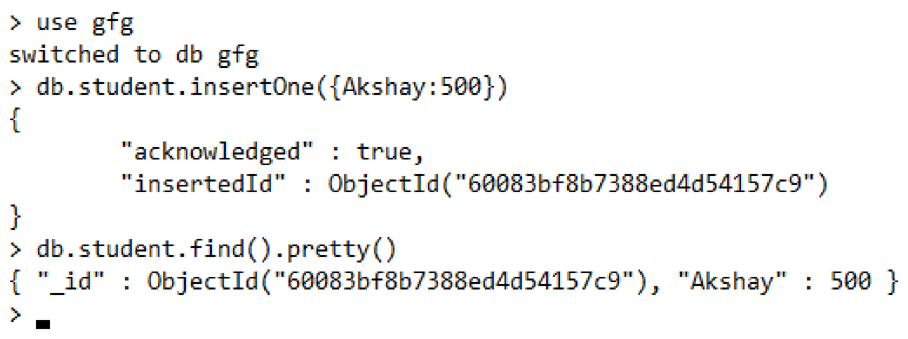
Now, you are ready to write queries in the mongo Shell.

Run MongoDB
Now you can make a new database, collections, and documents in your shell. Below is anexample of how to make a new database:
The *use Database_name* command makes a new database in the system if it does not exist, ifthe database exists it uses that database:

```
use gfg
```

Now your database is ready of name gfg.
The db.Collection_name command makes a new collection in the gfg database and the insertOne() method inserts the document in the student collection:

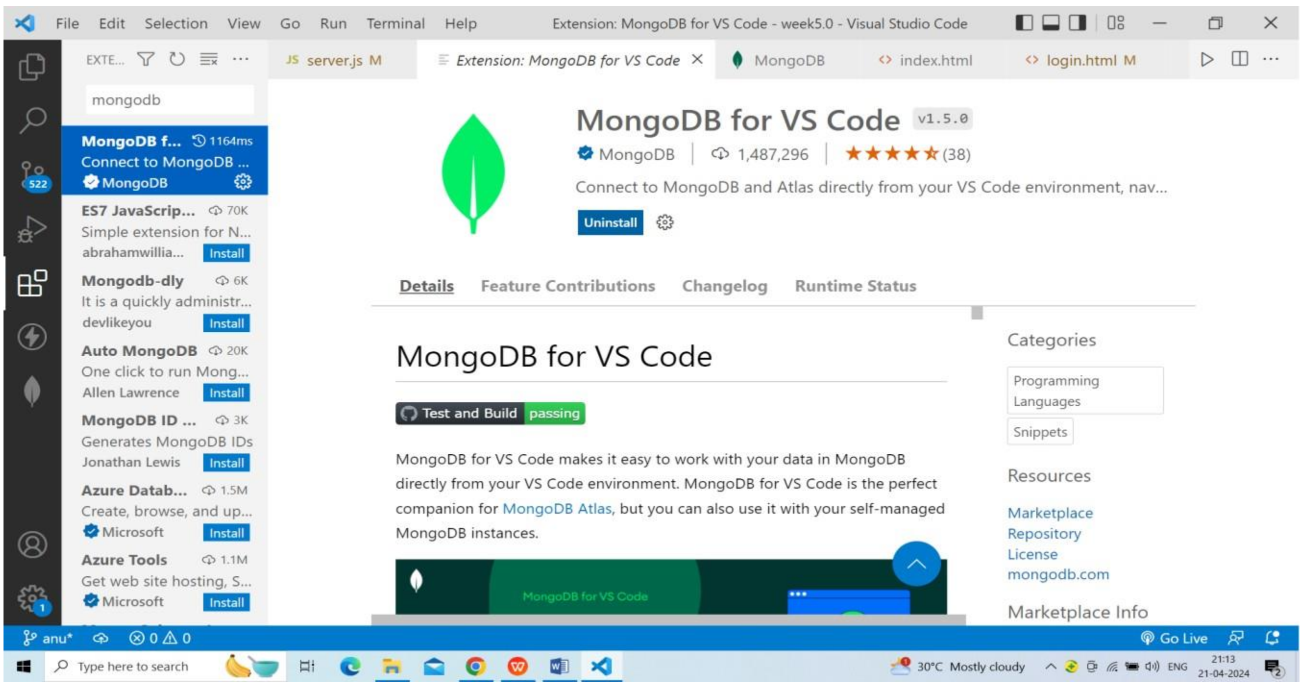
```
db.student.insertOne({Akshay:500})
```



```
> use gfg
switched to db gfg
> db.student.insertOne({Akshay:500})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60083bf8b7388ed4d54157c9")
}
> db.student.find().pretty()
{ "_id" : ObjectId("60083bf8b7388ed4d54157c9"), "Akshay" : 500 }
> ▂
```

2. **MongoDB – Visual Studio Extension**

- Search and install MongoDB Visual Studio Code - Extension



File Edit Selection View Go Run Terminal Help
Extension: MongoDB for VS Code - week5.0 - Visual Studio Code
server.js M
Extension: MongoDB for VS Code
MongoDB
index.html
login.html M
mongodb
MongoDB f... 1164ms Connect to MongoDB ... MongoDB
ES7 JavaScrip... 70K Simple extension for N... abrahamwillia... Install
Mongodb-dly 6K It is a quickly administr... devlikeyou Install
Auto MongoDB 20K One click to run Mong... Allen Lawrence Install
MongoDB ID ... 3K Generates MongoDB IDs Jonathan Lewis Install
Azure Datab... 1.5M Create, browse, and up... Microsoft Install
Azure Tools 1.1M Get web site hosting, S... Microsoft Install
MongoDB for VS Code v1.5.0
MongoDB | 1,487,296 | ★★★★☆ (38)
Uninstall
Details Feature Contributions Changelog Runtime Status
Test and Build passing
Categories
Programming Languages
Snippets
Resources
Marketplace
Repository
License
mongodb.com
Marketplace Info
Go Live
Type here to search
30°C Mostly cloudy
ENG 21:13 21-04-2024

Connect to MongoDB and Atlas directly from your VS Code environment, nav...

MongoDB for VS Code

MongoDB for VS Code makes it easy to work with your data in MongoDB directly from your VS Code environment. MongoDB for VS Code is the perfect companion for MongoDB Atlas, but you can also use it with your self-managed MongoDB instances.

Fig.1. **MongoDB – Visual Studio Extension**

1. MongoDB - Atlas

- Cloud-Hosted and Fully Managed MongoDB

- Pay as you go model
- Very cost-effective
- Fully secured and reliable

2. Data Modeling

Definition:-

Data modelling refers to the organization of data within a database and the links between related entities. Datain MongoDB has a **flexible schema model**, which means:

- Documents within a single collection are not required to have the same set of fields.
- A field's data type can differ between documents within a collection.

The primary problem in data modeling is balancing application needs, **database engine performance** features, and **data retrieval patterns**. Always consider the application uses of thedata (i.e. **queries**, **updates**, and **data processing**) as well as the fundamental design of the data itself when creating data models.

➢ Advantages Of Data Modelling

Data modelling is essential for a successful application, even though at first it might just seem likeone more step. In addition to increasing **overall efficiency** and **improving development cycles**, data modelling helps you better understand the data at hand and identify future business requirements, which can save time and money. In particular, applying suitable data models:

- Improves application performance through better database strategy, design, andimplementation.
- Allows faster application development by making object mapping easier.
- Helps with better data learning, standards, and validation.
- Allows organizations to assess long-term solutions and model data while solving not justcurrent projects but also future application requirements, including maintenance.

Different Types of Data Models

The three types of data models that are typically classified as follows:

1. Conceptual data model

Conceptual Data Models are **rough sketches** that provide the big picture, detailing where data/information from various business processes will be stored in the database system and therelationships they will be involved with. A conceptual data model typically includes the **entityclass**, **attributes**, **constraints**, and the relationship between security and data integrity requirements.
This model describes the types of data that should be in the system and how they relate to one another. This model, which is typically developed with the support of the business stakeholders, itcontains the business logic of the application, often involves **domain-**

driven design (DDD) principles, and serves as the foundation for one or more of the following models. The primary purpose of the conceptual model is to identify the **information that will be essential to an organization**.

2. Logical data model

Logical data models provide more detailed, **subjective information about data set relationships**.At this stage, we can clearly connect what data types and relations are used. Logical data models are generally missed in rapid business contexts, having their utility in **data-driven** initiatives requiring important procedure execution.
The logical data model specifies **how data will be organized**. The relationship between entities isestablished at a high level .In this model, and a list of entity properties is also provided. This data model can be viewed as a "**blueprint**" for the data that will be used.

3. Physical data model

The schema/layout for data storage routines within a database is defined by the **physical datamodel**. A physical data model is a ready-to-implement plan that can be stored in a **relational database**.
The physical data model is a representation of **how data will be stored in a particular database** management system (DBMS). In this approach, main and secondary keys in a relational database are defined, or the decision to include or connect data in a document database such as MongoDB based on entity relationships is made. This is also where you will define the **data typesfor each of your fields**, which will create the database structure.

➢ Data Model Design (or) Types

For modelling data in MongoDB, two strategies are available. These strategies are different and itis recommended to analyze our scenario for a better flow. The two methods are as follows:

1. Embedded Data Model
2. Normalized Data Model

1. Embedded Data Model

This method, also known as the **de-normalized** data model, allows you to have (embed) all ofthe **related data in a single document**.
For example, if we obtain student information in three different documents, Personal_details,Contact, and Address, we can embed all three in a single one, as shown below.

```
{
   _id: ,
   Std_ID: "987STD001"
   Personal_details:{
      First_Name:
      "Rashmika",
      Last_Name: "Sharma",
      Date_Of_Birth: "1999-08-26"
   },
   Contact: {
      e-mail:
      "rashmika_sharma.123@gmail.com",
      phone: "9987645673"
   },
   Address: {
      city: "Karnataka",
      Area:
      "BTM2ndStage",
      State: "Bengaluru"
```

2. Normalized Data Model (or) Reference Data Model:

In a normalized data model, object references are used to express the **relationships between documents and data objects**. Because this approach **reduces data duplication**, it is relatively simple to document **many-to-many relationships** without having to repeat content. Normalizeddata models are the most effective technique to model large **hierarchical data** with cross- collection relationships.

Student:

```
{
   _id: <StudentId101>,
   Std_ID:
   "10025AE336"
}
```

Personal_Details:

```
{
   _id: <StudentId102>,
   stdDocID: " StudentId101",
   First_Name: "Rashmika",
   Last_Name: "Sharma",
   Date_Of_Birth: "1999-08-
   26"
}
```

Contact:

```
{
   _id: <StudentId103>,
   stdDocID: "
   StudentId101",
   e-mail:
   "rashmika_sharma.123@gmail.com",
   phone: "9987645673"
```

Address:

```
{
   _id: <StudentId104>,
   stdDocID: "
   StudentId101",city:
   "Karnataka",
   Area:
   "BTM2ndStage",
   State: "Bengaluru"
```

Considerations while designing Schema in MongoDB

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them(but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.

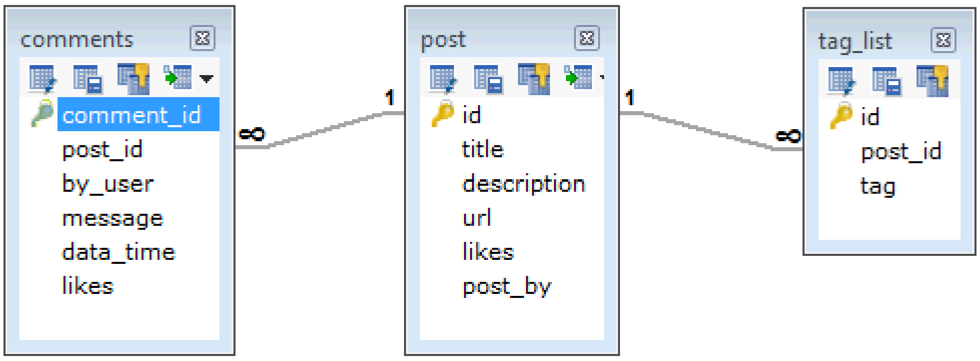
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time andlikes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure −

```
{
   _id: POST_ID
   title:  TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

➢ Connect MongoDB:-

Mongodb compass

- Connect with compass app
- Understand basics of compass app
- Get your hands-on examples with

compassHostname:-Localhost

Port:-27017

Visual Studio:-

- Connect with Visual Studio Code Extension
- Understand basics of visual Studio Code Extension
- Get your hands-on examples with Visual Studio Code Extension

Connect:- mongodb://localhost:27017

Shell:**mongos**

h(Or)

Mongod creating server and **mongo** for shell.

➢ CURD Operation:-

Creating and drop database:-

- use anu;//creating
- show dbs;//display all db
- db//current db
- db.dropDatabase();//deleting

db**Creating and drop collections:-**

Syntax:-

- db.createCollection(name,options)//creating
- db.collection.drop()
- db.collection.insertOne({key:"value"})

Ex:-

- db.createCollection(“products”);
- db.products.drop()

Inserting Documents into

Collections:-Syntax:-

- db.collection_name.insert({“name”:”aaa”})//one
- db.collection_name.insertMany([{“name”:”aaa”}, {“name”:”bbb”}])//many
- **Example:-** db.aaa.insert({“name”:”aaa”})//one
- db.bbb.insertMany([{"name":"aaa"}, {"name":"bbb"}])//many

Update:-

- db.bbb.update({"name":"bbb"},{$set:{"name":"ccc","isActive":true}});

Read:-

- db.bbb.find();
- db.bbb.findOne();
- db.bbb.find({"name":"ccc"});
- db.bbb.findOneAndReplace({"name":"ccc"},{"name":"eee"});
- db.bbb.findOneAndDelete({"name":"eee"});

Delete:-

db.orders.deleteOne({"name":"aaa"})

/*

Db.student.insertOne({name:”anusha”})

Db.student.find().pretty()

*/

3. Query and Projection

MongoDB Query

MongoDB Query Operators

Similar to SQL MongoDB have also some operators to operate on data in the collection. MongoDBquery operators check the conditions for the given data and **logically** compare the data with the help of two or more fields in the document.

Query operators help to filter data based on specific conditions. **E.g.**, $eq,$and,$exists, etc.

MongoDB provides the function names as *db.collection_name.find()* to operate query operationon database.

Syntax:

db.collection_name.find()

Example:

db.article.find()

Types of Query Operators in MongoDB

The Query operators in MongoDB can be further classified into 8 more types. The 8 types ofQuery Operators in MongoDB are:

1. Comparison Operators
2. Logical Operators
3. Array Operators
4. Evaluation Operators
5. Element Operators
6. Bitwise Operators
7. Geospatial Operators
8. Comment Operators

1. Comparison Operators

The comparison operators in MongoDB are used to perform value-based comparisons in queries.The comparison operators in the MongoDB are shown as below:

Comparison Operator	Description	Syntax
$eq	Matches values that are equal to a specified value.	{ field: { $eq: value } }
$ne	Matches all values that are not equal to a specified value.	{ field: { $ne: value } }
$lt	Matches values that are less than a specified value.	{ field: { $lt: value } }
$gt	Matches values that are greater than a specified value.	{ field: { $gt: value } }
$lte	Matches values that are less than or equal to a specified value.	{ field: { $lte: value } }
$gte	Matches values that are greater than or equal to a specified value.	{ field: { $gte: value } }
$in	Matches any of the values specified in an array.	{ field: { $in: [<value1>, <value2>, ...] } }

Documents:

db.books.insertMany([{"p_name":"book","price":50},{"p_name":"pen","price":100},{"p_name":"pencilbox","price":500},{"p_name":"ball","price":200}]);

MongoDB Comparison Operators

1. $eq

The $eq specifies the equality condition. It matches documents where the value of a field equals the specifiedvalue.

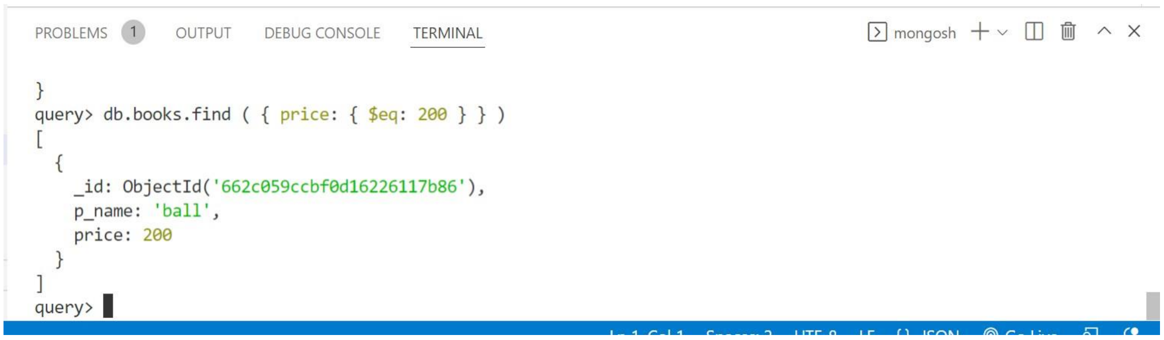
Syntax:

```
1. { <field> : { $eq: <value> } }
```

Example:

```
db.books.find ( { price: { $eq: 200 } } )
```

The above example queries the books collection to select all documents where the value of the price filed equals300.



```
}
query> db.books.find ( { price: { $eq: 200 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query> 
```

2. $gt

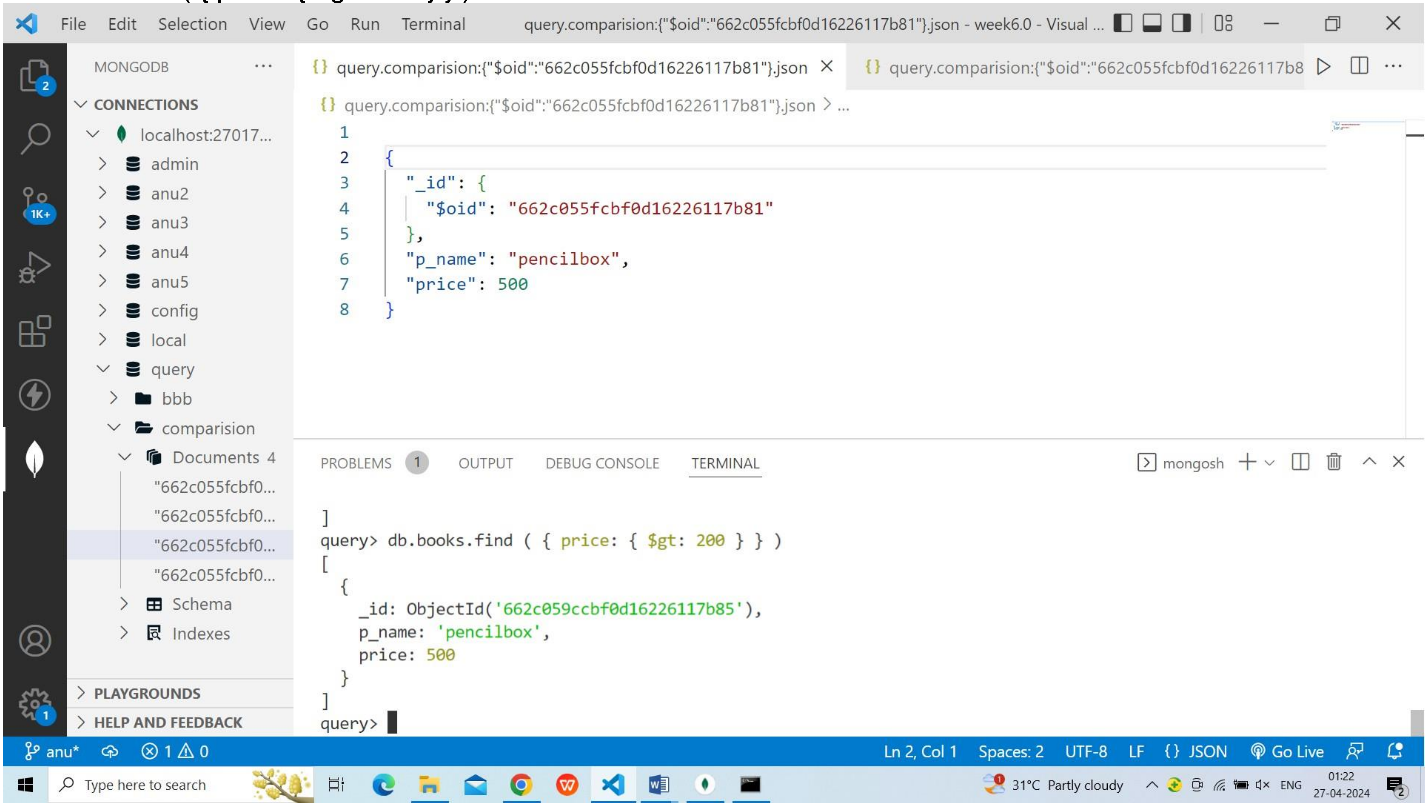
The $gt chooses a document where the value of the field is greater than the specified value.

Syntax:

1. { field: { $gt: value } }

Example:

1. db.books.find ({ price: { $gt: 200 } })



```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b81"
  },
  "p_name": "pencilbox",
  "price": 500
}
```

```
]
query> db.books.find ( { price: { $gt: 200 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  }
]
query> 
```

3.$gte

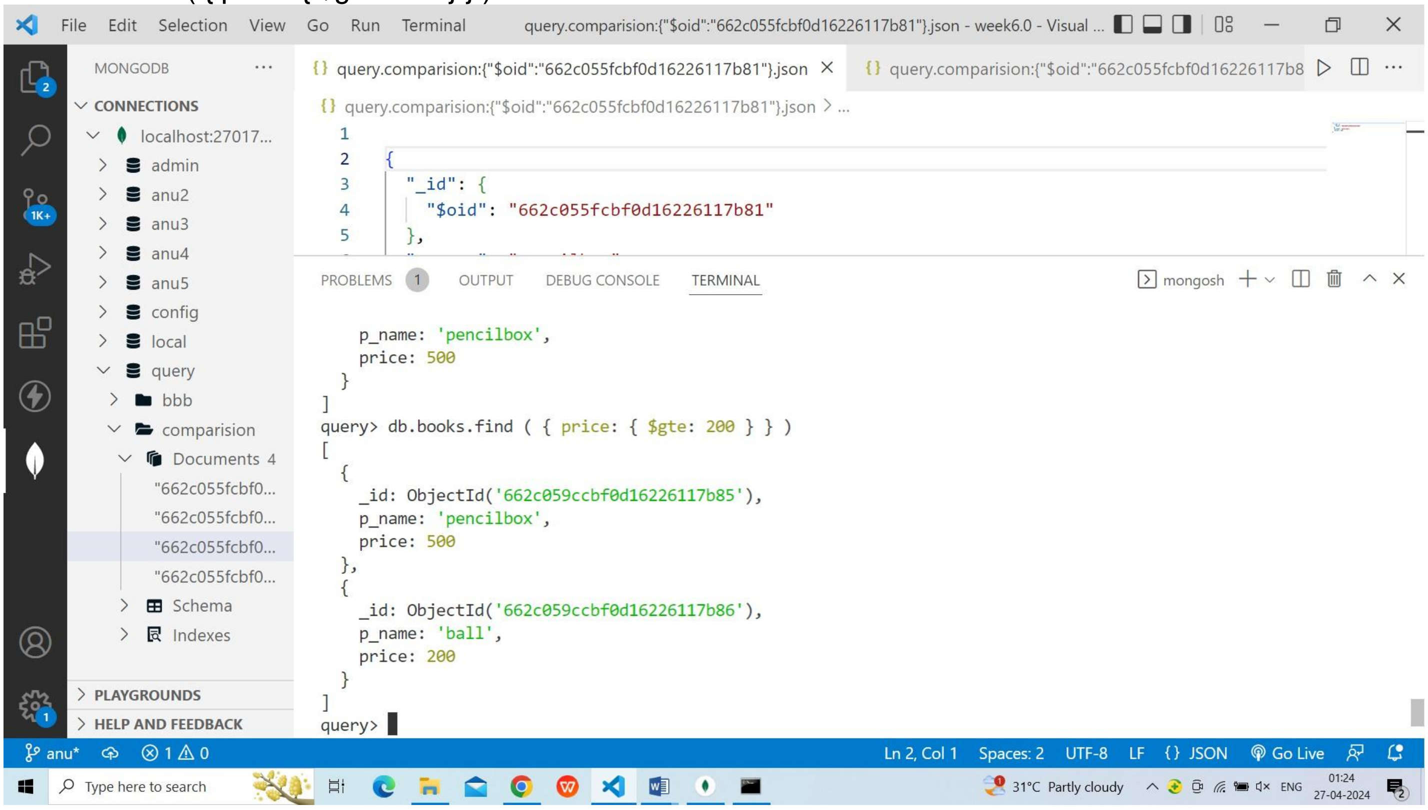
The $gte choose the documents where the field value is greater than or equal to a specified value.

Syntax:

```
1. { field: { $gte: value } }
```

Example:

1. db.books.find ({ price: { $gte: 250 } })


File Edit Selection View Go Run Terminal
query.comparision:{"$oid":"662c055fcbf0d16226117b81"}.json - week6.0 - Visual ...
MONGODB
CONNECTIONS
localhost:27017...
admin
anu2
anu3
anu4
anu5
config
local
query
bbb
comparision
Documents 4
"662c055fcbf0...
"662c055fcbf0...
"662c055fcbf0...
"662c055fcbf0...
Schema
Indexes
PLAYGROUNDS
HELP AND FEEDBACK
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
mongosh
Ln 2, Col 1 Spaces: 2 UTF-8 LF {} JSON Go Live
Type here to search
31°C Partly cloudy ENG 01:24 27-04-2024

```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b81"
  },
```

```
    p_name: 'pencilbox',
    price: 500
  }
]
query> db.books.find ( { price: { $gte: 200 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query>
```

4.$in

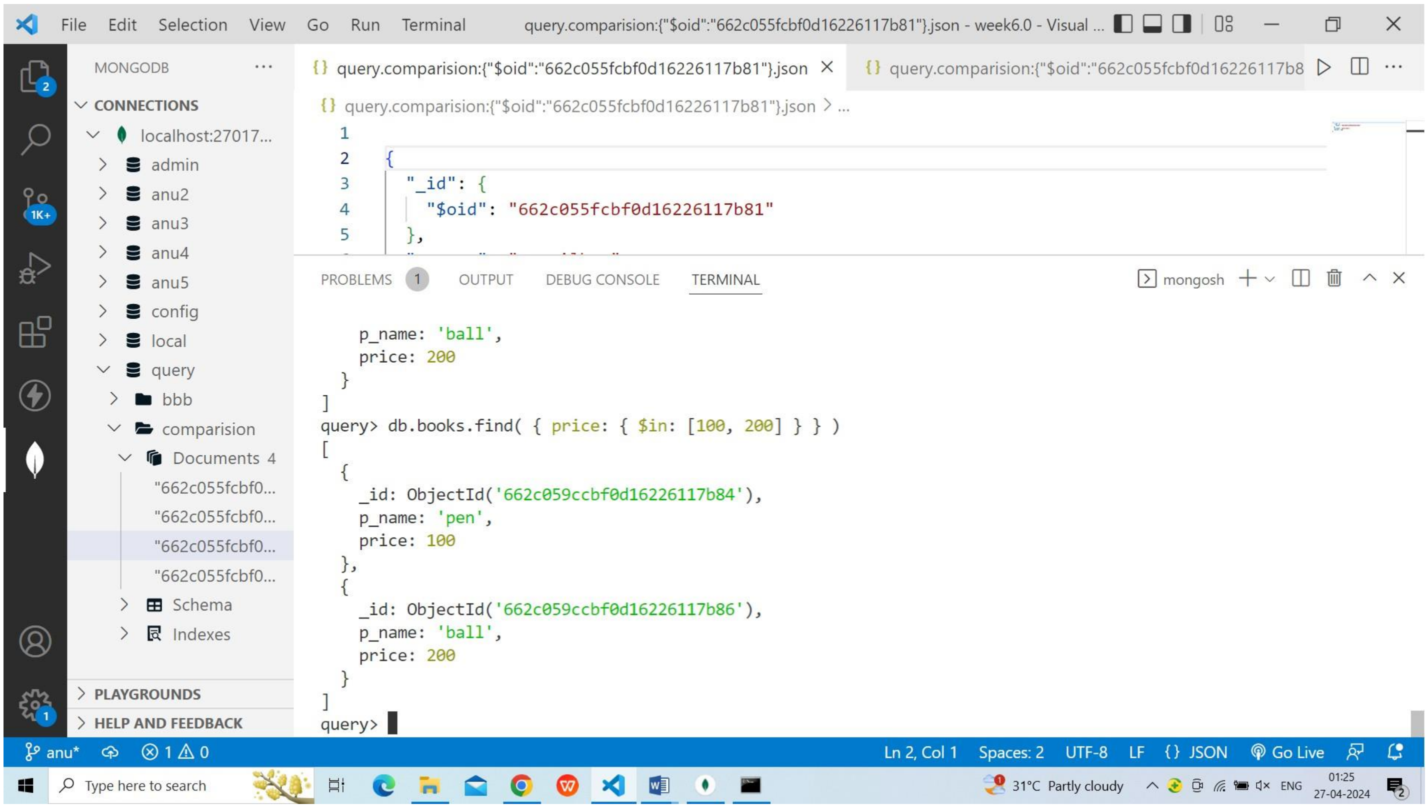
The $in operator choose the documents where the value of a field equals any value in the specified array.

Syntax:

1. { filed: { $in: [<value1>, <value2>, ...] } }

Example:

1. db.books.find({ price: { $in: [100, 200] } })


```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b81"
  },
```

```
    p_name: 'ball',
    price: 200
  }
]
query> db.books.find( { price: { $in: [100, 200] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query>
```

5.$lt

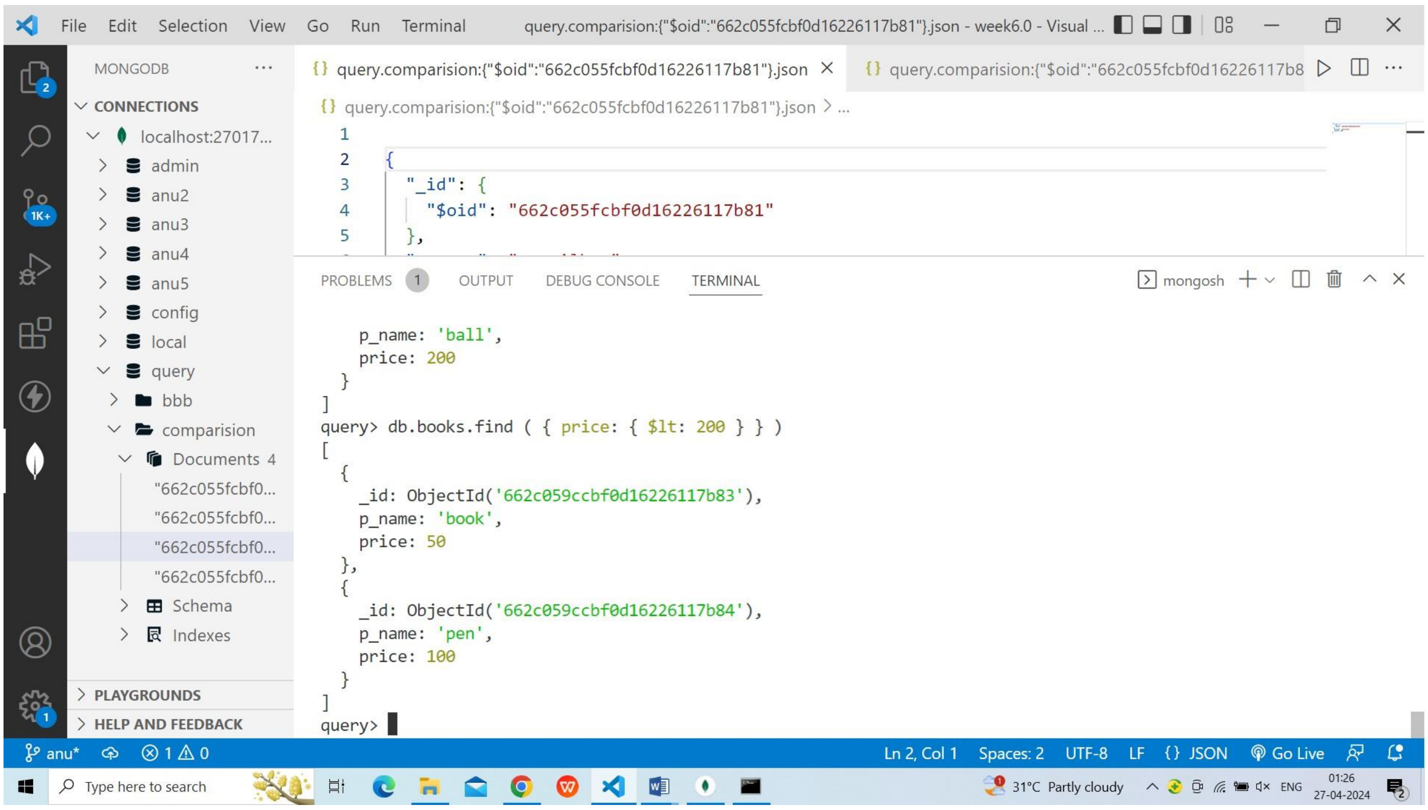
The $lt operator chooses the documents where the value of the field is less than the specified value.

Syntax:

```
1. { field: { $lt: value } }
```

Example:

```
1. db.books.find ( { price: { $lt: 20 } } )
```



```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b81"
  },
```

```
    p_name: 'ball',
    price: 200
  }
]
query> db.books.find ( { price: { $lt: 200 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  }
]
query>
```

6. $lte

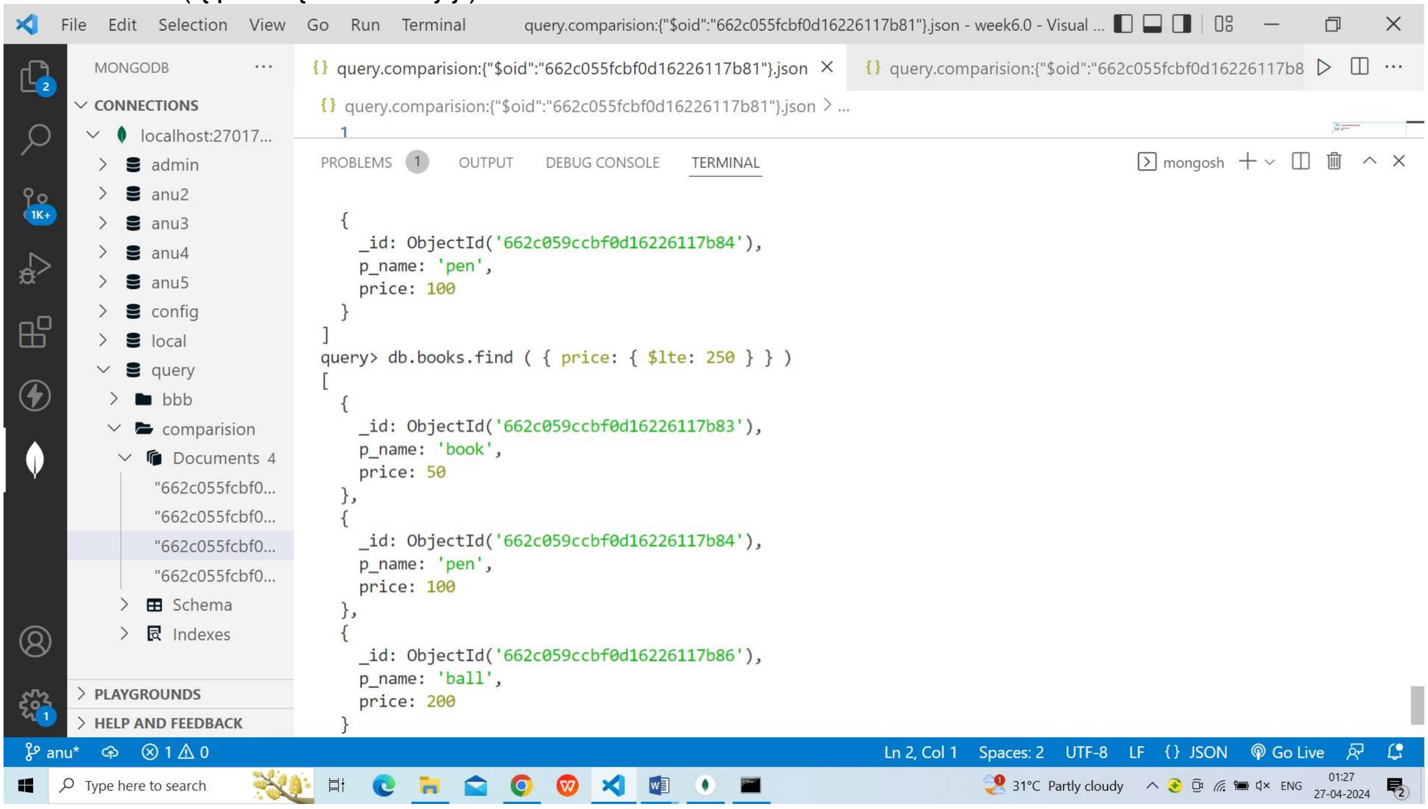
The $lte operator chooses the documents where the field value is less than or equal to a specified value.

Syntax:

1. { field: { $lte: value } }

Example:

1. db.books.find ({ price: { $lte: 250 } })



```
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  }
]
query> db.books.find ( { price: { $lte: 250 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
```

7. $ne

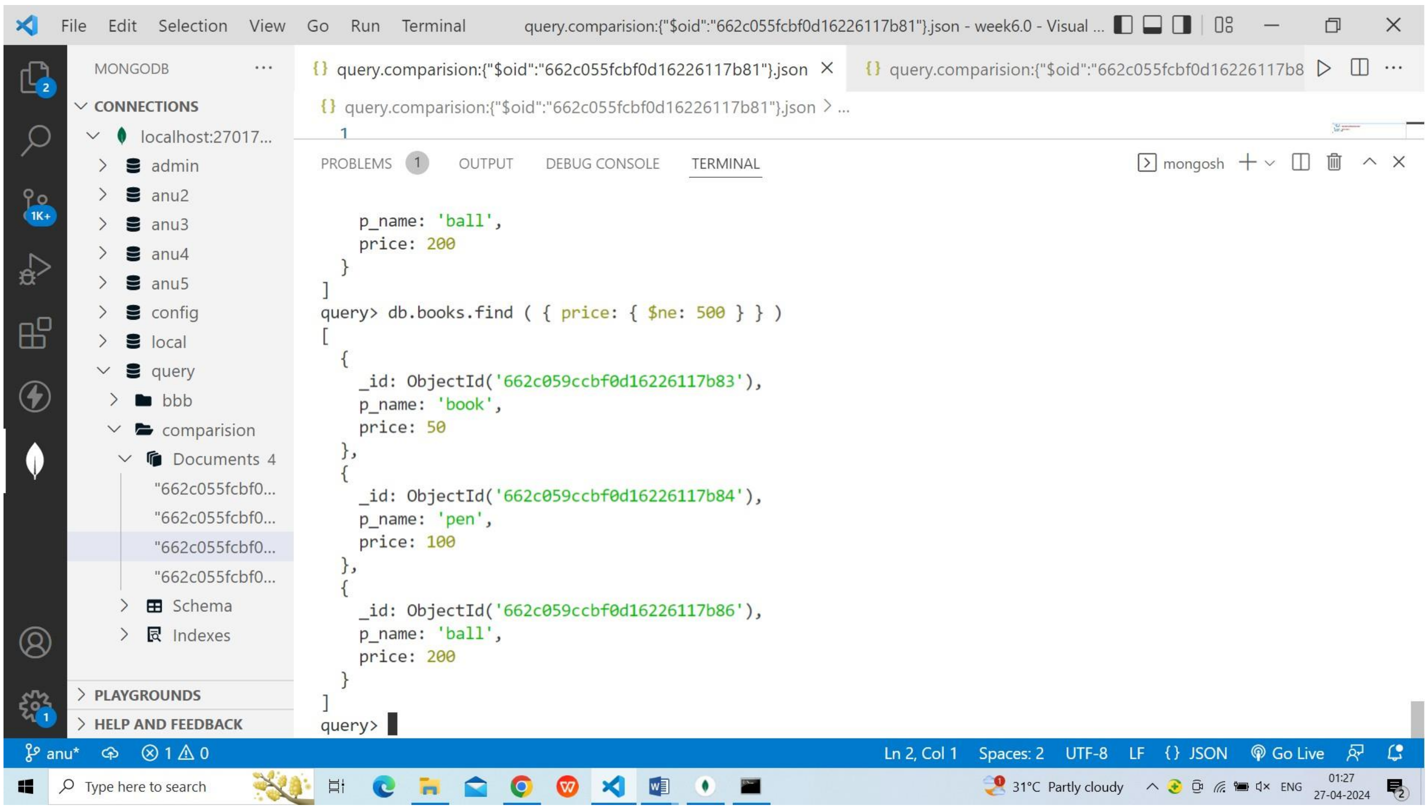
The $ne operator chooses the documents where the field value is not equal to the specified value.

Syntax:

1. { <field>: { $ne: <value> } }

Example:

1. db.books.find ({ price: { $ne: 500 } })



```
    p_name: 'ball',
    price: 200
  }
]
query> db.books.find ( { price: { $ne: 500 } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query> 
```

8. $nin

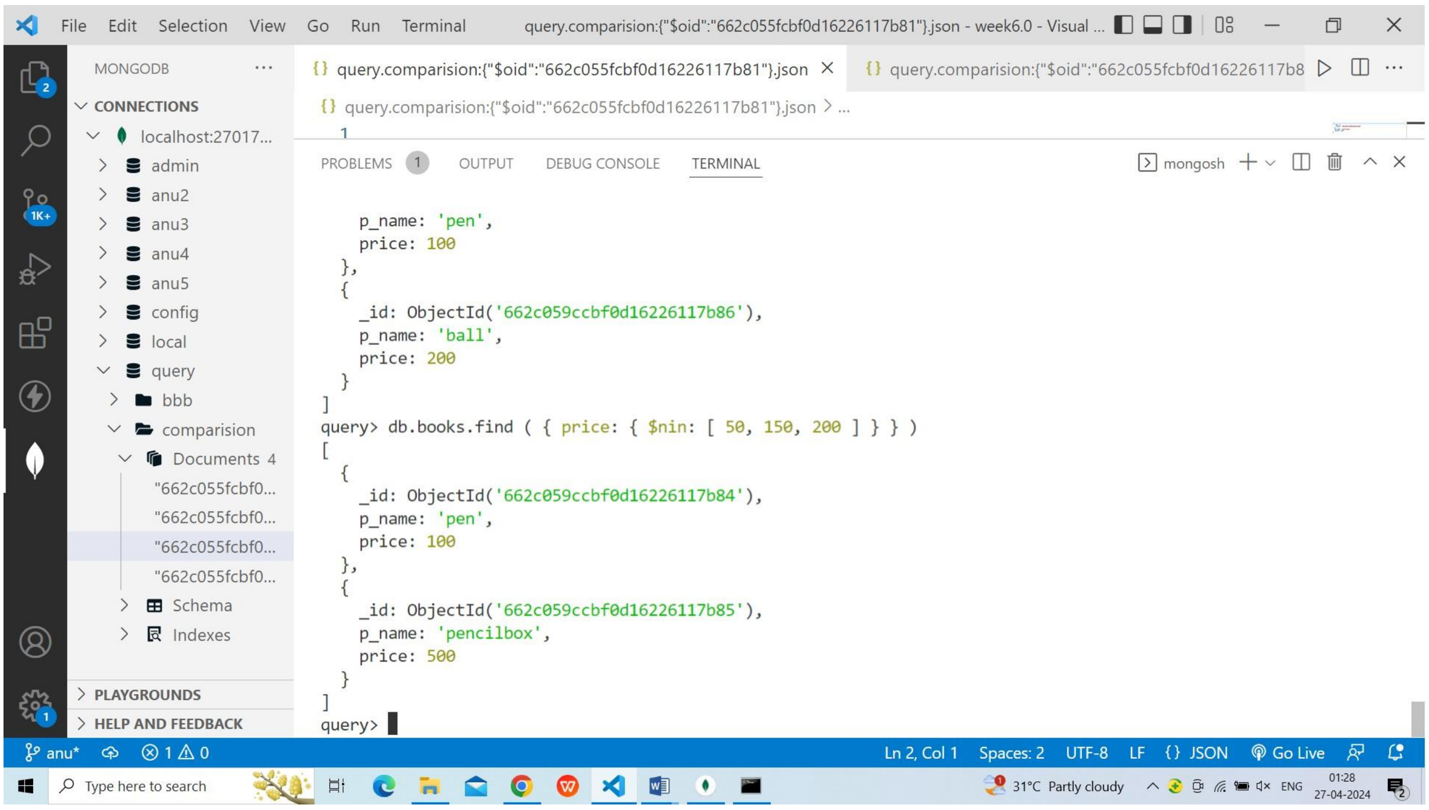
The $nin operator chooses the documents where the field value is not in the specified array or does not exist.

Syntax:

1. { field : { $nin: [<value1>, <value2>, ..] } }

Example:

1. db.books.find ({ price: { $nin: [50, 150, 200] } })



```
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query> db.books.find ( { price: { $nin: [ 50, 150, 200 ] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  }
]
query> 
```

2.MongoDB Logical Operator

Logical Operators

The logical operators in MongoDB are used to filter data based on expressions that evaluate totrue or false.
The Logical operators in MongoDB are shown in the table below:

Logical Operator	Description	Syntax
$and	Returns all the documents that satisfy all the conditions.	{ $and: [{ <expression1> }, { <expression2> } , ... , { <expressionN> }] }
$not	Inverts the effect of the query expression and returns documents that do not match the query expression.	{ field: { $not: { <operator-expression> } } }
$or	Returns the documents from the query that match either one of the conditions in the query.	{ $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }
$nor	Returns the documents that fail to match both conditions.	{ $nor: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

$and

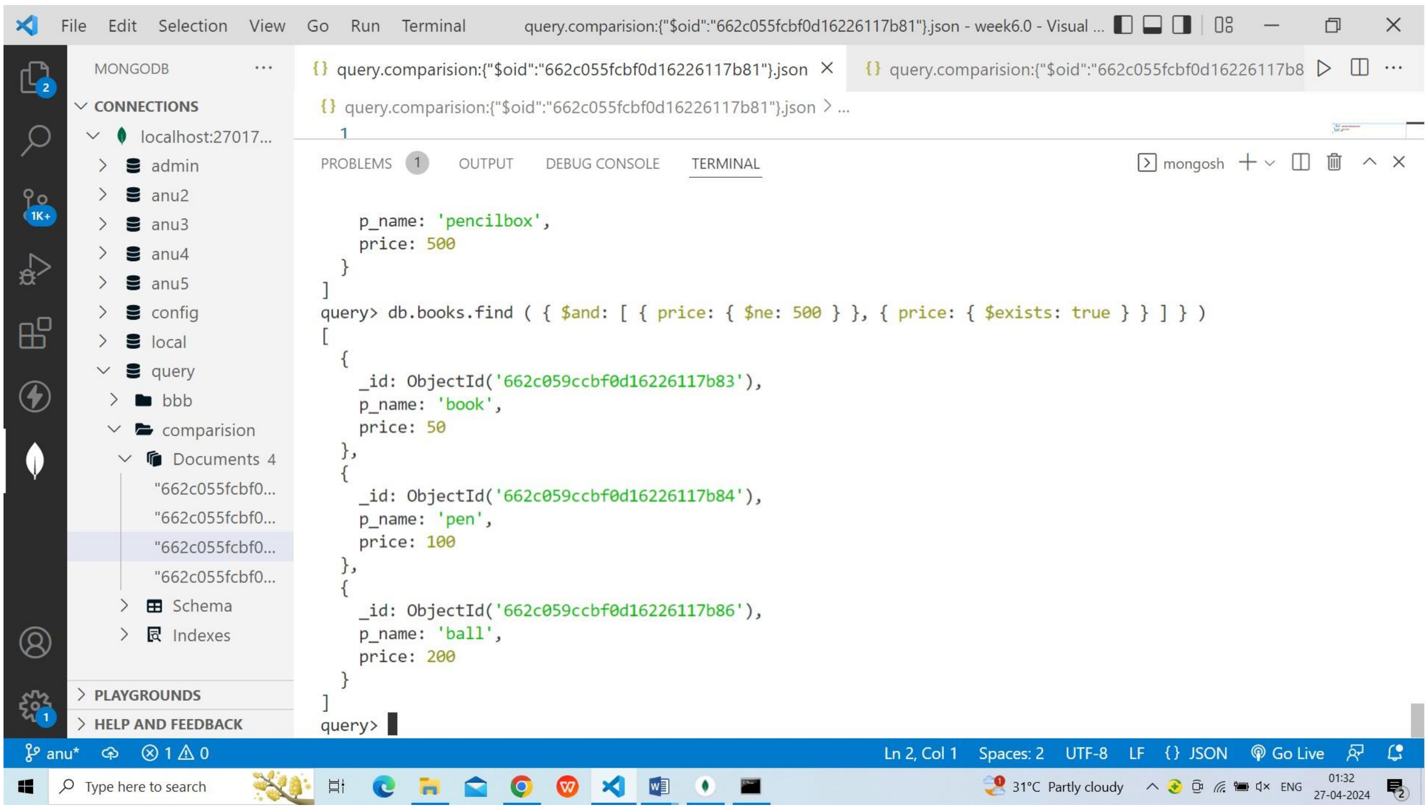
The $and operator works as a logical AND operation on an array. The array should be of one or more expressionsand chooses the documents that satisfy all the expressions in the array.

Syntax:

1. `{ $and: [ { <exp1> }, { <exp2> }, ..]}`

Example:

1. `db.books.find ( { $and: [ { price: { $ne: 500 } }, { price: { $exists: true } } ] } )`


```
    p_name: 'pencilbox',
    price: 500
  }
]
query> db.books.find ( { $and: [ { price: { $ne: 500 } }, { price: { $exists: true } } ] } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query> 
```

$not

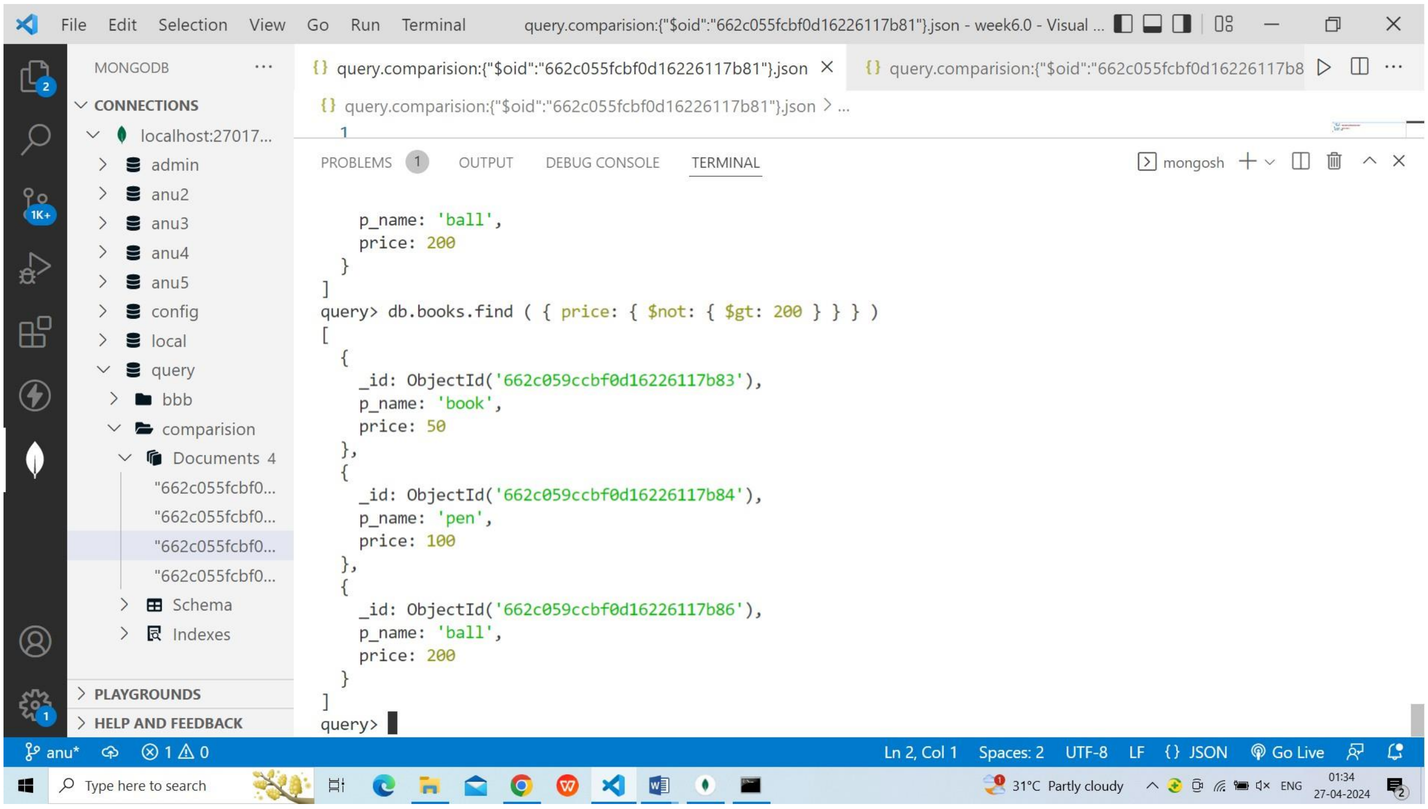
The $not operator works as a logical NOT on the specified expression and chooses the documents that are notrelated to the expression.

Syntax:

1. { field: { $not: { <operator-expression> } } }

Example:

1. db.books.find ({ price: { $not: { $gt: 200 } } })



```
    p_name: 'ball',
    price: 200
  }
]
query> db.books.find ( { price: { $not: { $gt: 200 } } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query> 
```

$nor

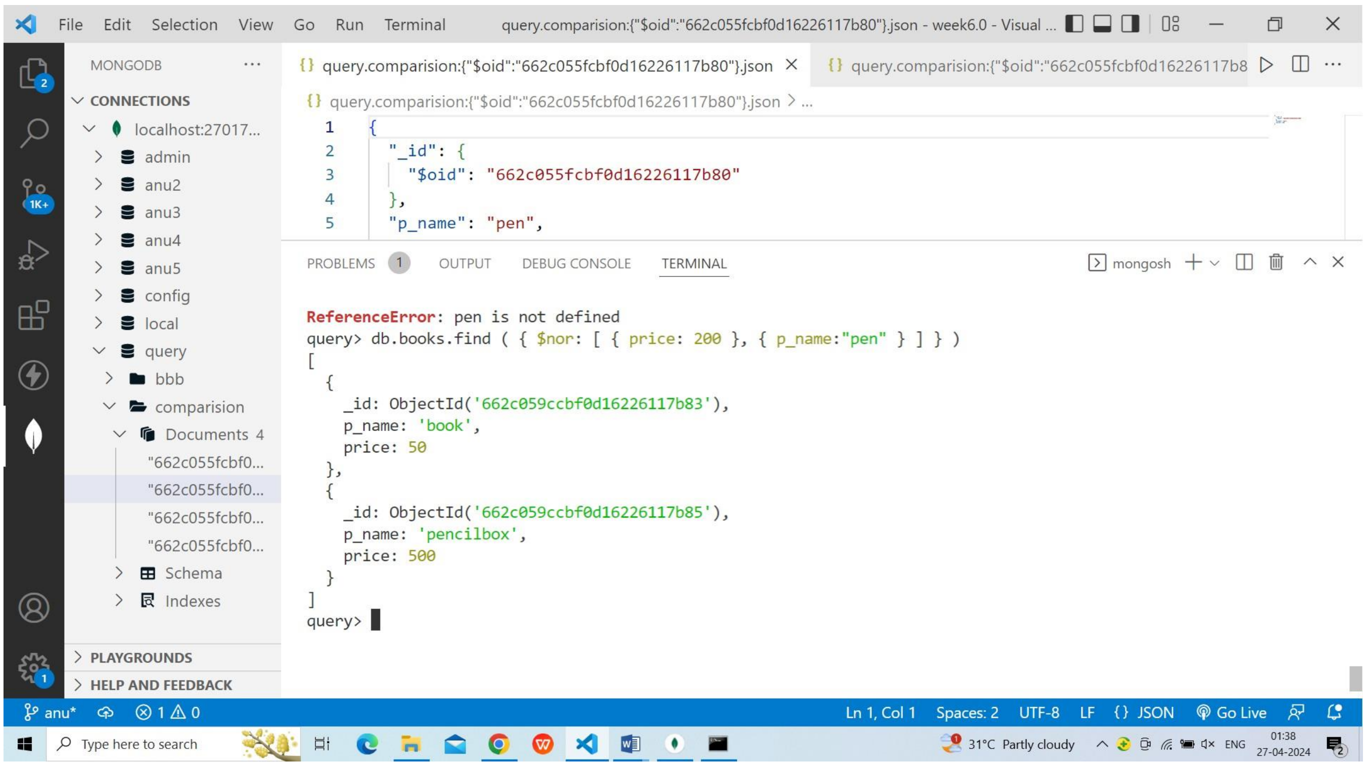
The $nor operator works as logical NOR on an array of one or more query expression and chooses the documentsthat fail all the query expression in the array.

Syntax:

1. { $nor: [{ <expression1> } , { <expresion2> } , ..] }

Example:

db.books.find ({ $nor: [{ price: 200 }, { p_name:"pen" }] })



```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b80"
  },
  "p_name": "pen",
```

```
ReferenceError: pen is not defined
query> db.books.find ( { $nor: [ { price: 200 }, { p_name:"pen" } ] } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  }
]
query> 
```

$or

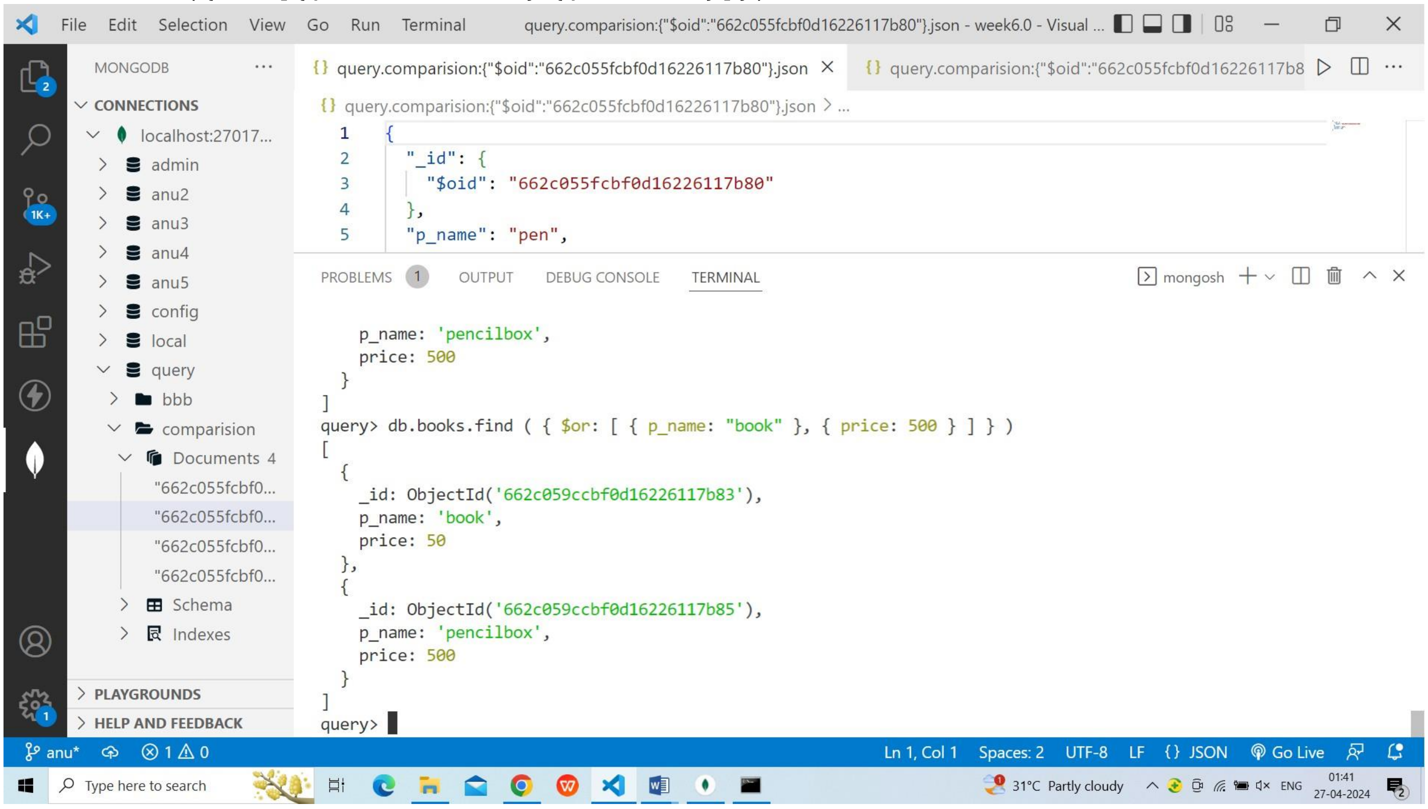
It works as a logical OR operation on an array of two or more expressions and chooses documents that meet theexpectation at least one of the expressions.

Syntax:

1. { $or: [{ <exp_1> }, { <exp_2> }, ... , { <exp_n> }] }

Example:

db.books.find ({ $or: [{ p_name: "book" }, { price: 500 }] })



```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b80"
  },
  "p_name": "pen",
```

```
    p_name: 'pencilbox',
    price: 500
  }
]
query> db.books.find ( { $or: [ { p_name: "book" }, { price: 500 } ] } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  }
]
query> 
```

3.Array Operator

Name	**Description**
$all	Matches arrays that contain all elements specified in the query.
$elemMatch	Selects documents if element in the array field matches all the specified $elemMatch conditio
$size	Selects documents if the array field is a specified size.

$all

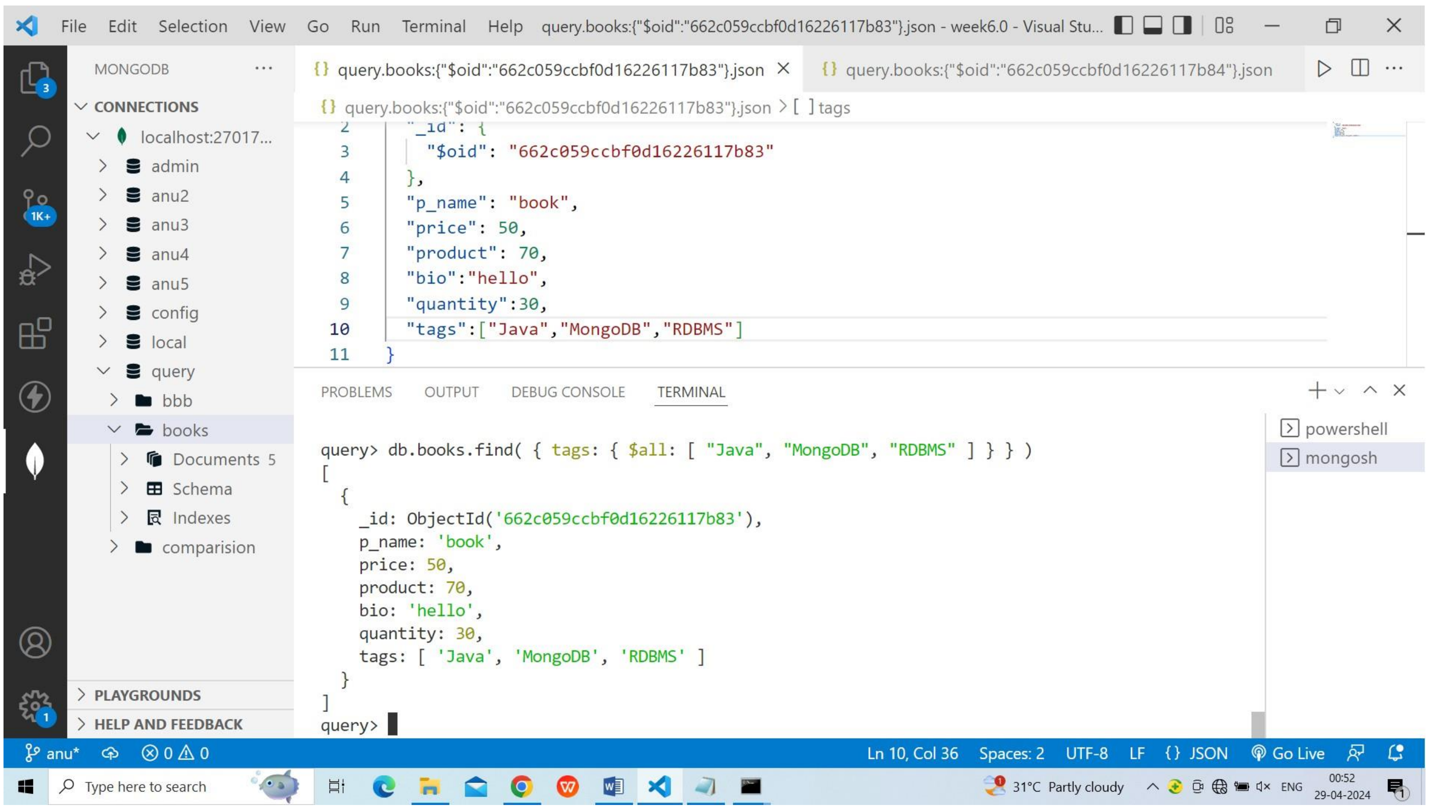
It chooses the document where the value of a field is an array that contains all the specified elements.

Syntax:

1. { <field>: { $all: [<value1> , <value2> ...] } }

Example:

1. db.books.find({ tags: { $all: ["Java", "MongoDB", "RDBMS"] } })



```
"_id": {
  "$oid": "662c059ccbf0d16226117b83"
},
"p_name": "book",
"price": 50,
"product": 70,
"bio":"hello",
"quantity":30,
"tags":["Java","MongoDB","RDBMS"]
}
```

```
query> db.books.find( { tags: { $all: [ "Java", "MongoDB", "RDBMS" ] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70,
    bio: 'hello',
    quantity: 30,
    tags: [ 'Java', 'MongoDB', 'RDBMS' ]
  }
]
query> 
```

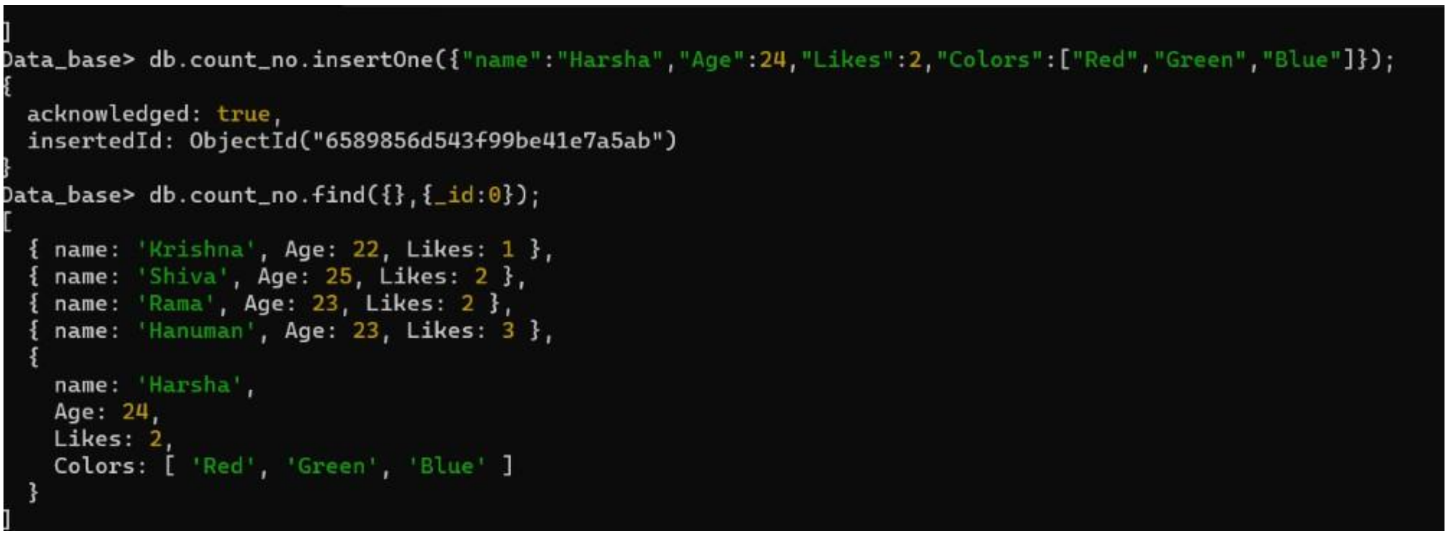
$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all thegiven query criteria.

Syntax:

1. { <field>: { $elemMatch: { <query1>, <query2>, ... } } }

Example of Using $elemMatch Operator
Let's first make some updates in our demo collection. Here we will Insert some data into thecount_no database
Query:
db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"] });
Output:



```
]
Data_base> db.count_no.insertOne({"name":"Harsha","Age":24,"Likes":2,"Colors":["Red","Green","Blue"]});
{
  acknowledged: true,
  insertedId: ObjectId("6589856d543f99be41e7a5ab")
}
Data_base> db.count_no.find({},{_id:0});
[
  { name: 'Krishna', Age: 22, Likes: 1 },
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 },
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
```

Inserting the array of elements

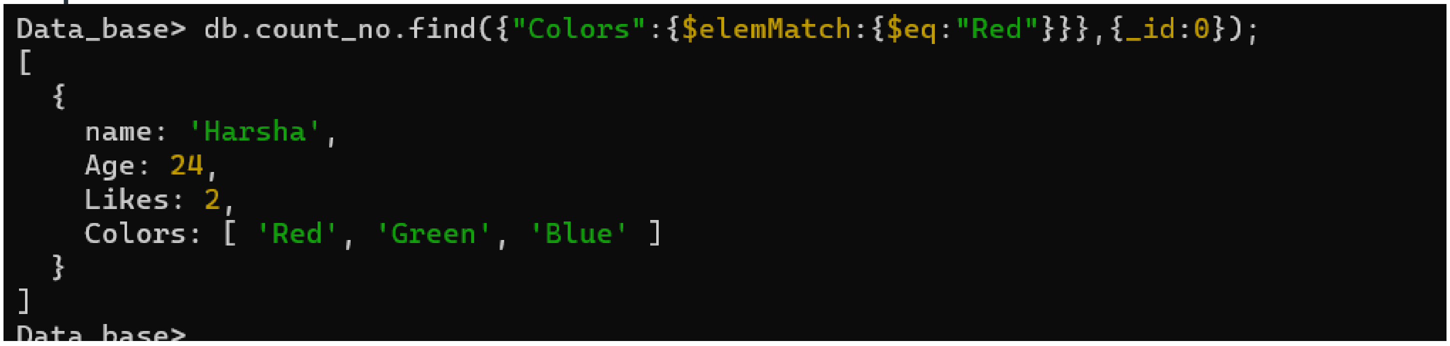
.

Now, using the $**elemMatch** operator in MongoDB let's match the Red colors from a set ofColors.

Query:

db.count_no.find({"Colors": {$elemMatch: {$eq: "Red" } } },{_id: 0 });

Output:



```
Data_base> db.count_no.find({"Colors":{$elemMatch:{$eq:"Red"}}},{_id:0});
[
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```

Using the elemMatch Operator.

Explanation: The $elemMatch operator is used with the field **Colors** which is of type array. In the above query, it returns the documents that have the field Colors and if any of the values in theColors field has "**Red**" in it.

Example:

1. db.books.find({ price: { $elemMatch: { $gte: 500, $lt: 400 } } })

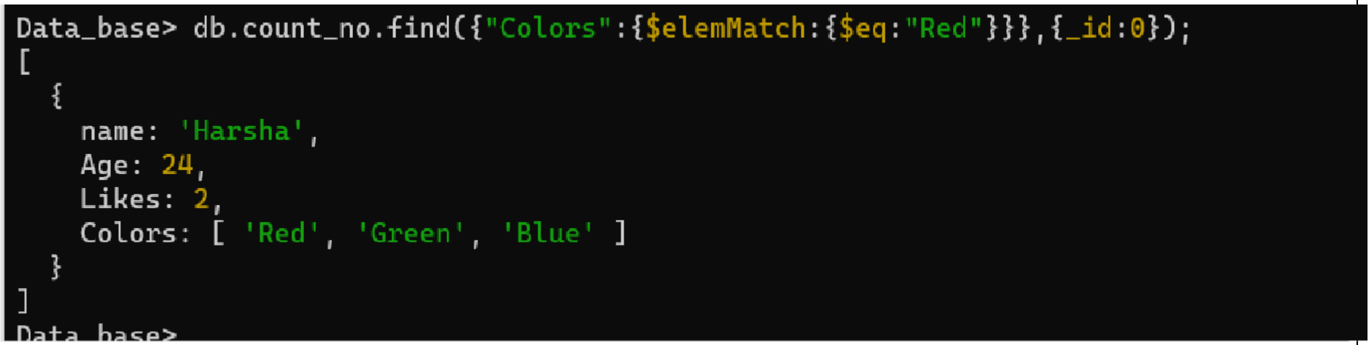
$size

It selects any array with the number of the element specified by the argument.

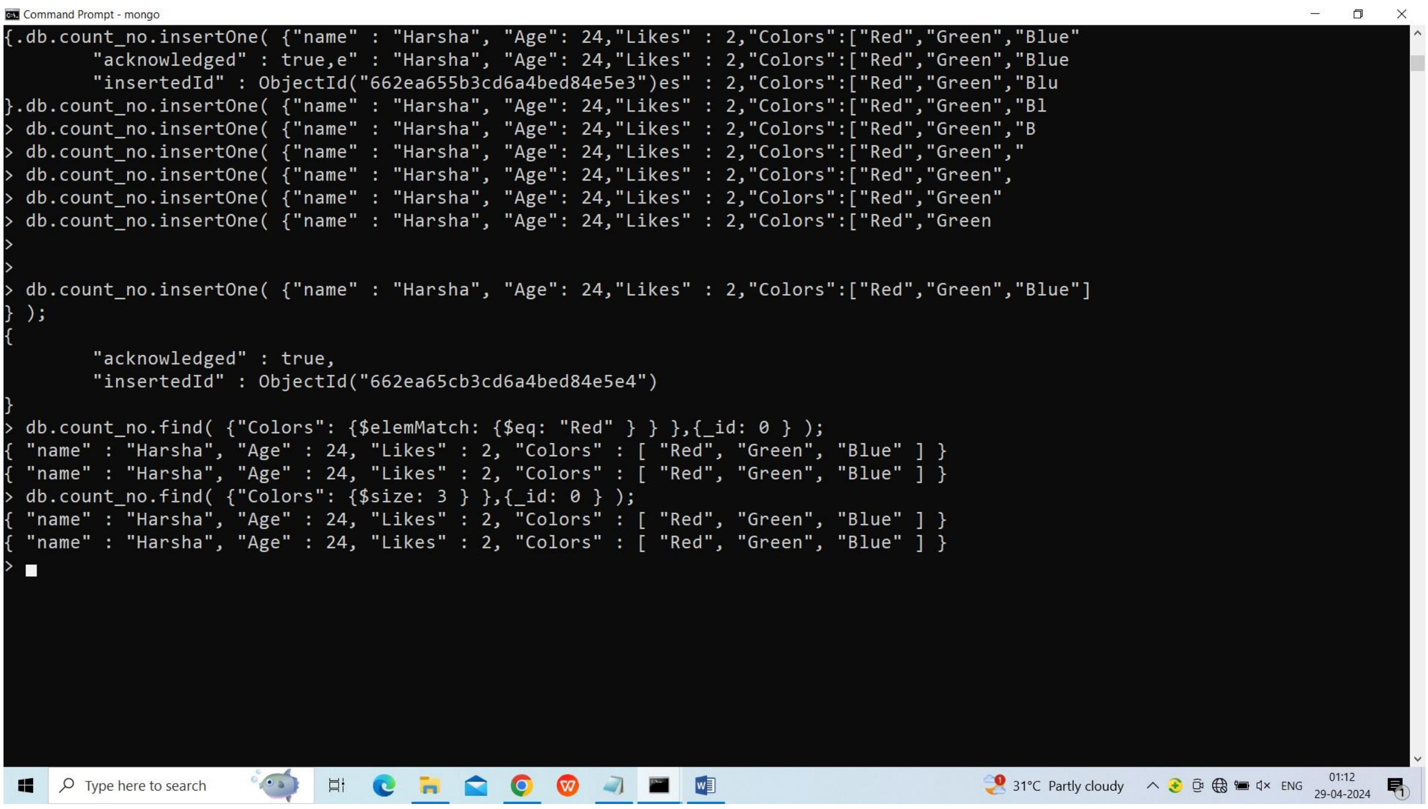
Syntax:

```
1. db.collection.find( { field: { $size: 2 } } );
```

2. db.count_no.find({"Colors": {$size: 3 } },{_id: 0 });



```
Data_base> db.count_no.find({"Colors":{$elemMatch:{$eq:"Red"}}},{_id:0});
[
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```



```
Command Prompt - mongo
{.db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"
        "acknowledged" : true,e" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue
        "insertedId" : ObjectId("662ea655b3cd6a4bed84e5e3")es" : 2,"Colors":["Red","Green","Blu
}.db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Bl
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","B
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green",
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green"
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green
>
>
> db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"]
} );
{
        "acknowledged" : true,
        "insertedId" : ObjectId("662ea65cb3cd6a4bed84e5e4")
}
> db.count_no.find( {"Colors": {$elemMatch: {$eq: "Red" } } },{_id: 0 } );
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : [ "Red", "Green", "Blue" ] }
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : [ "Red", "Green", "Blue" ] }
> db.count_no.find( {"Colors": {$size: 3 } },{_id: 0 } );
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : [ "Red", "Green", "Blue" ] }
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : [ "Red", "Green", "Blue" ] }
> ■
```

4.MongoDB Evaluation Operator

The evaluation operators in the MongoDB are used to return the documents based on the result ofthe given expression.
Some of the evaluation operators present in the MongoDB are:

Evaluation Operator	Description	Syntax
$mod operator	The $mod operator in MongoDB performs a modulo operation on the value of a field and selects documents where the modulo equalsa specified value. It only works with numerical fields.	{ field: { $mod: [divisor, remainder] } }
$expr operator	The $expr operator in MongoDB allows aggregation expressions tobe used as query conditions. It returns documents that satisfy the conditions of the query.	{ $expr: { <aggregation expression> } }
$where operator	The $where operator in MongoDB uses JavaScript expression or function to perform queries. It evaluates the function for every document in the database and returns the documents that match the condition.	{ $where: <JavaScript expression

Evaluation

Name	Description
$expr	Allows use of aggregation expressions within the query language.
$jsonSchema	Validate documents against the given JSON Schema.
$mod	Performs a modulo operation on the value of a field and selects documentswith a specified result.
$regex	Selects documents where values match a specified regular expression.
$text	Performs text search.
$where	Matches documents that satisfy a JavaScript expression.

$expr

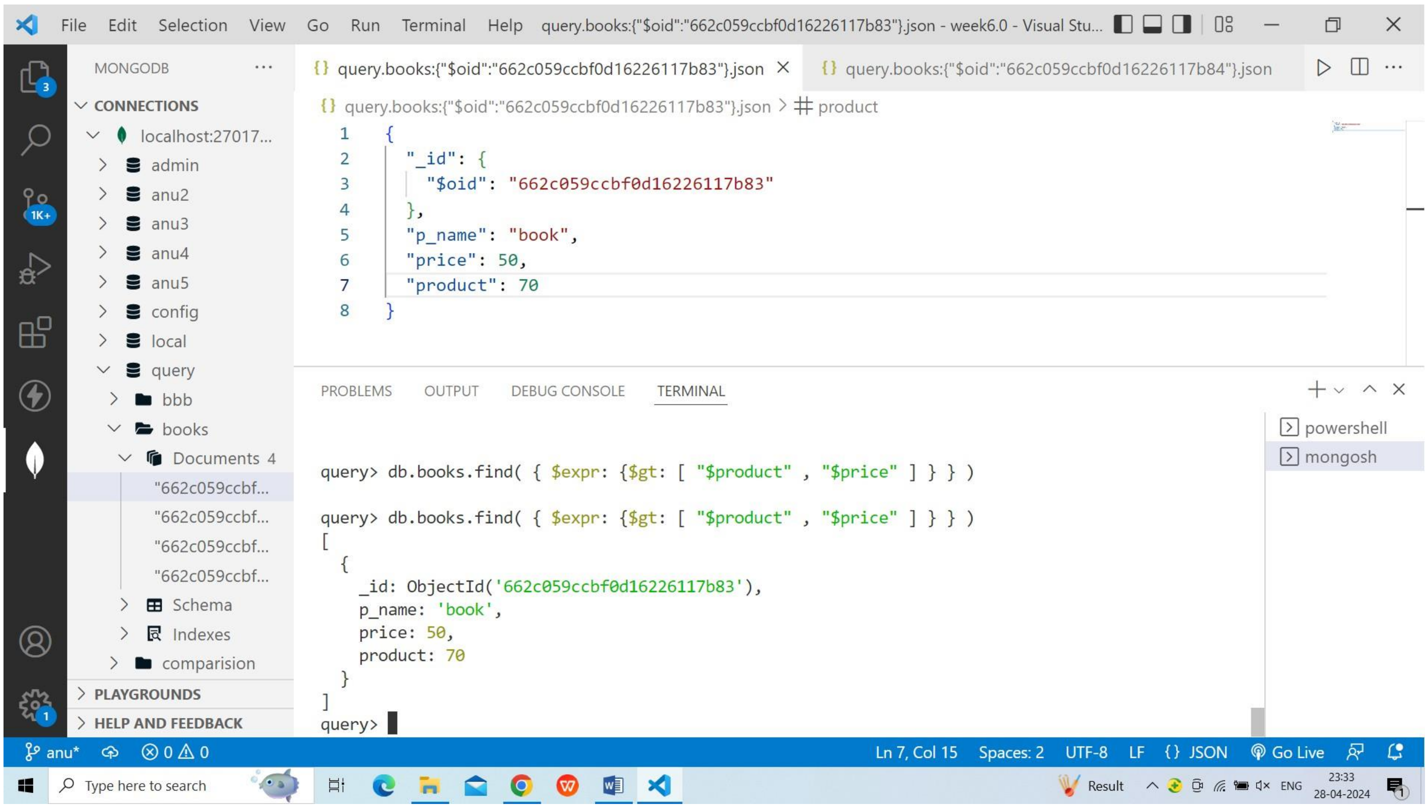
The expr operator allows the use of aggregation expressions within the query language.

Syntax:

```
1. { $expr: { <expression> } }
```

Example:

```
1. db.store.find( { $expr: {$gt: [ "$product" , "$price" ] } } )
```



```
{
  "_id": {
    "$oid": "662c059ccbf0d16226117b83"
  },
  "p_name": "book",
  "price": 50,
  "product": 70
}
```

```
query> db.books.find( { $expr: {$gt: [ "$product" , "$price" ] } } )

query> db.books.find( { $expr: {$gt: [ "$product" , "$price" ] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70
  }
]
query>
```

$jsonSchema

It matches the documents that satisfy the specified JSON

Schema.db.createCollection("students12",{

validator:{

 $jsonSchema: {

  required: ["name", "major", "gpa",

  "address"],properties: {

   name: {

    bsonType: "string",

    description: "must be a string and is required"

   },

   address: {

    bsonType: "object",

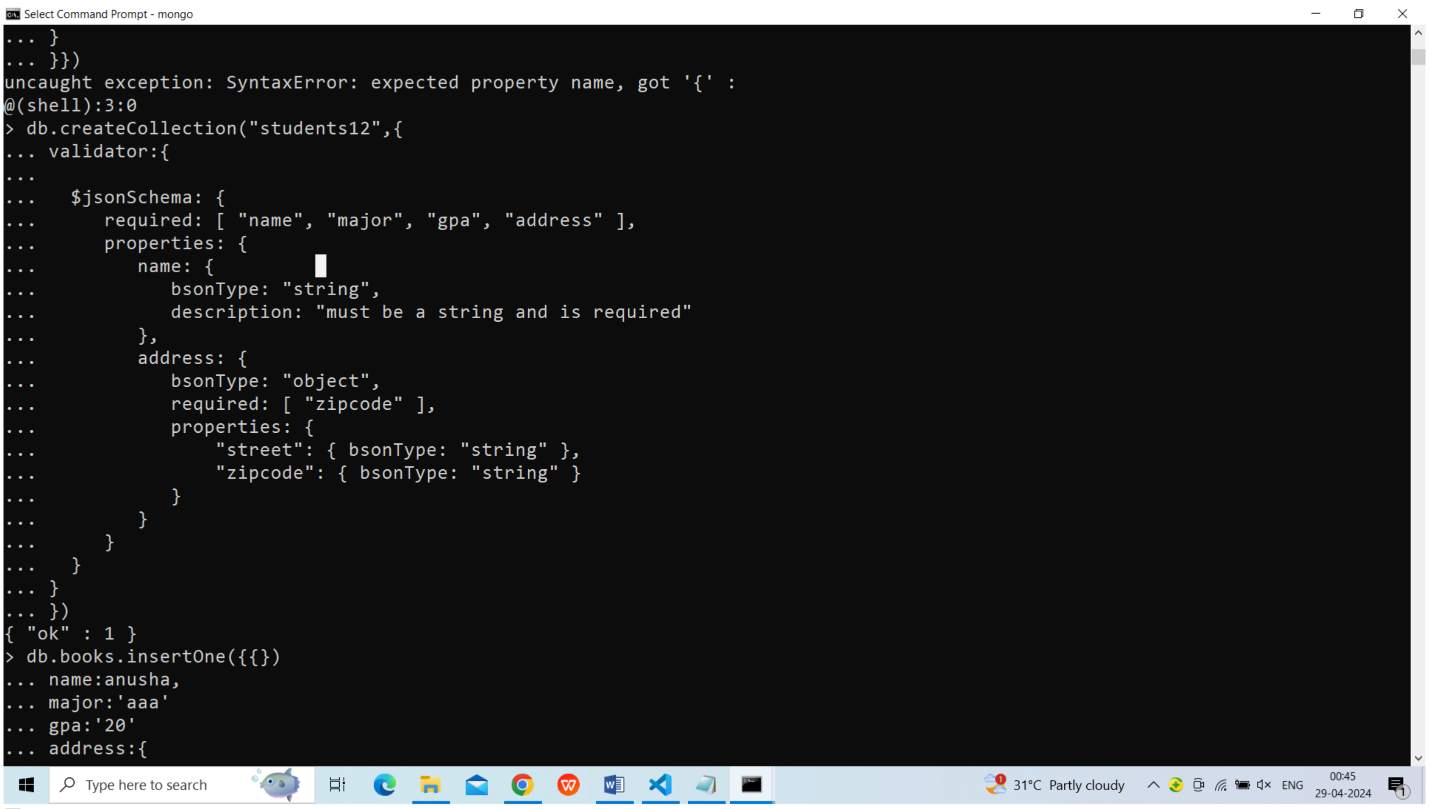
    required: ["zipcode"

```
],properties: {
```

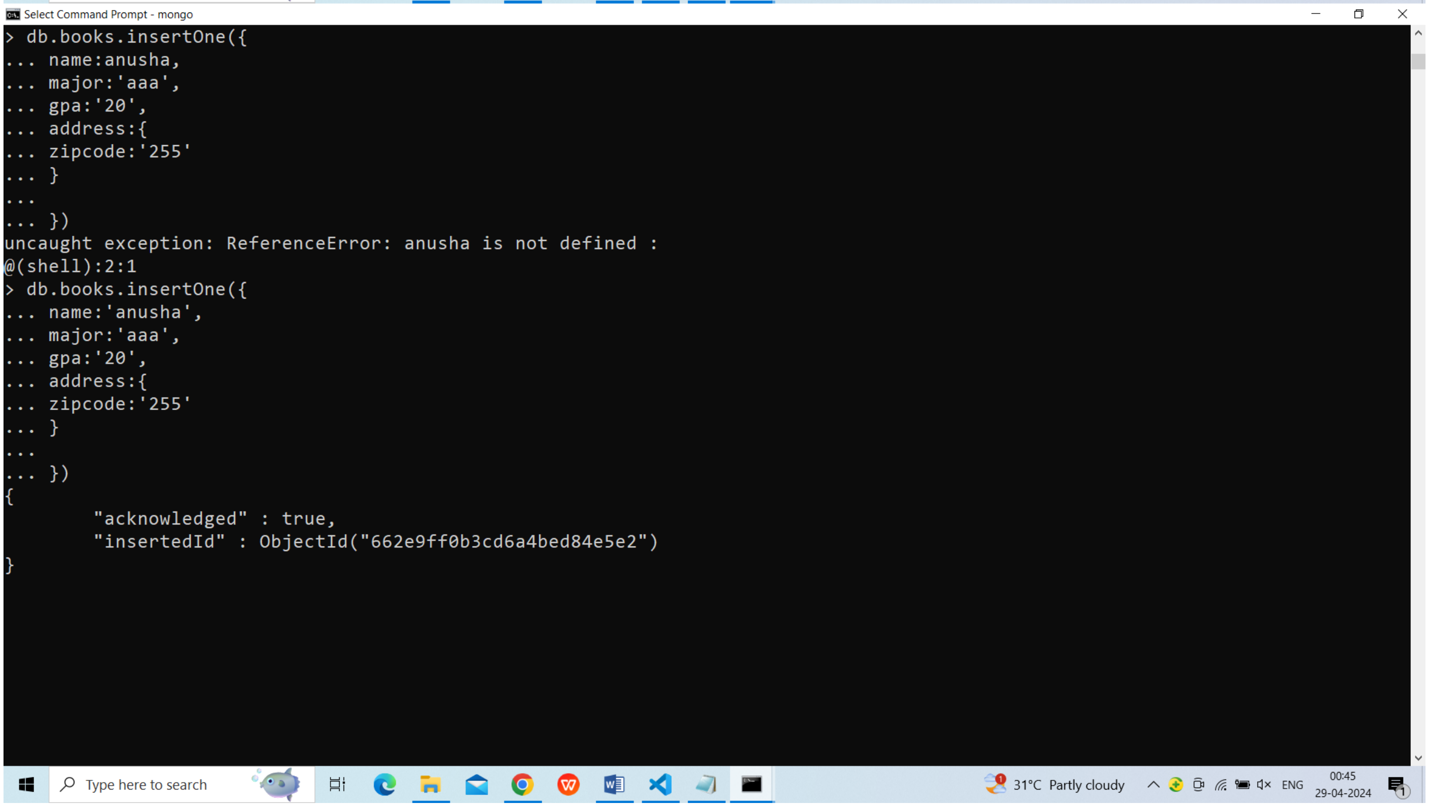
```
          "street": { bsonType: "string" },
          "zipcode": { bsonType: "string" }
        }
      }
    }
  }
}
})
//////
Student:
{
name:'anush
a',
major:'aaa',
gpa:'20',
address:{
zipcode:'25
5'
}

}
```

Syntax:

1. { $jsonSchema: <JSON schema object> }



```
... }
... }})
uncaught exception: SyntaxError: expected property name, got '{' :
@(shell):3:0
> db.createCollection("students12",{
... validator:{
...
...    $jsonSchema: {
...       required: [ "name", "major", "gpa", "address" ],
...       properties: {
...          name: {
...             bsonType: "string",
...             description: "must be a string and is required"
...          },
...          address: {
...             bsonType: "object",
...             required: [ "zipcode" ],
...             properties: {
...                 "street": { bsonType: "string" },
...                 "zipcode": { bsonType: "string" }
...             }
...          }
...       }
...    }
... }
... })
{ "ok" : 1 }
> db.books.insertOne({{})
... name:anusha,
... major:'aaa'
... gpa:'20'
... address:{
```



```
> db.books.insertOne({
... name:anusha,
... major:'aaa',
... gpa:'20',
... address:{
... zipcode:'255'
... }
...
... })
uncaught exception: ReferenceError: anusha is not defined :
@(shell):2:1
> db.books.insertOne({
... name:'anusha',
... major:'aaa',
... gpa:'20',
... address:{
... zipcode:'255'
... }
...
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("662e9ff0b3cd6a4bed84e5e2")
}
```

$mod

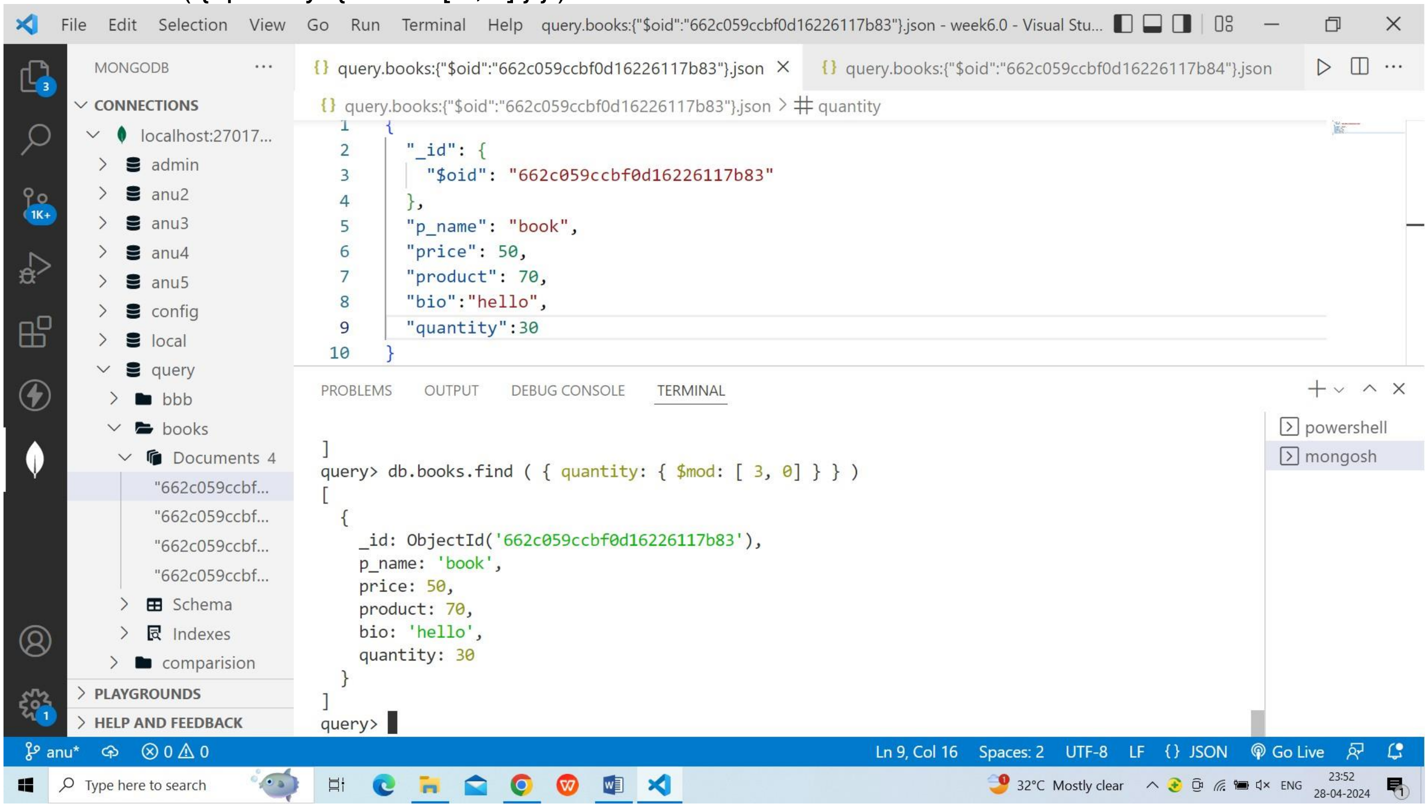
The mod operator selects the document where the value of a field is divided by a divisor has the specifiedremainder.

Syntax:

1. { field: { $mod: [divisor, remainder] } }

Example:

1. db.books.find ({ quantity: { $mod: [3, 0] } })



```
{
  "_id": {
    "$oid": "662c059ccbf0d16226117b83"
  },
  "p_name": "book",
  "price": 50,
  "product": 70,
  "bio":"hello",
  "quantity":30
}
```

```
]
query> db.books.find ( { quantity: { $mod: [ 3, 0] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70,
    bio: 'hello',
    quantity: 30
  }
]
query> 
```

$regex

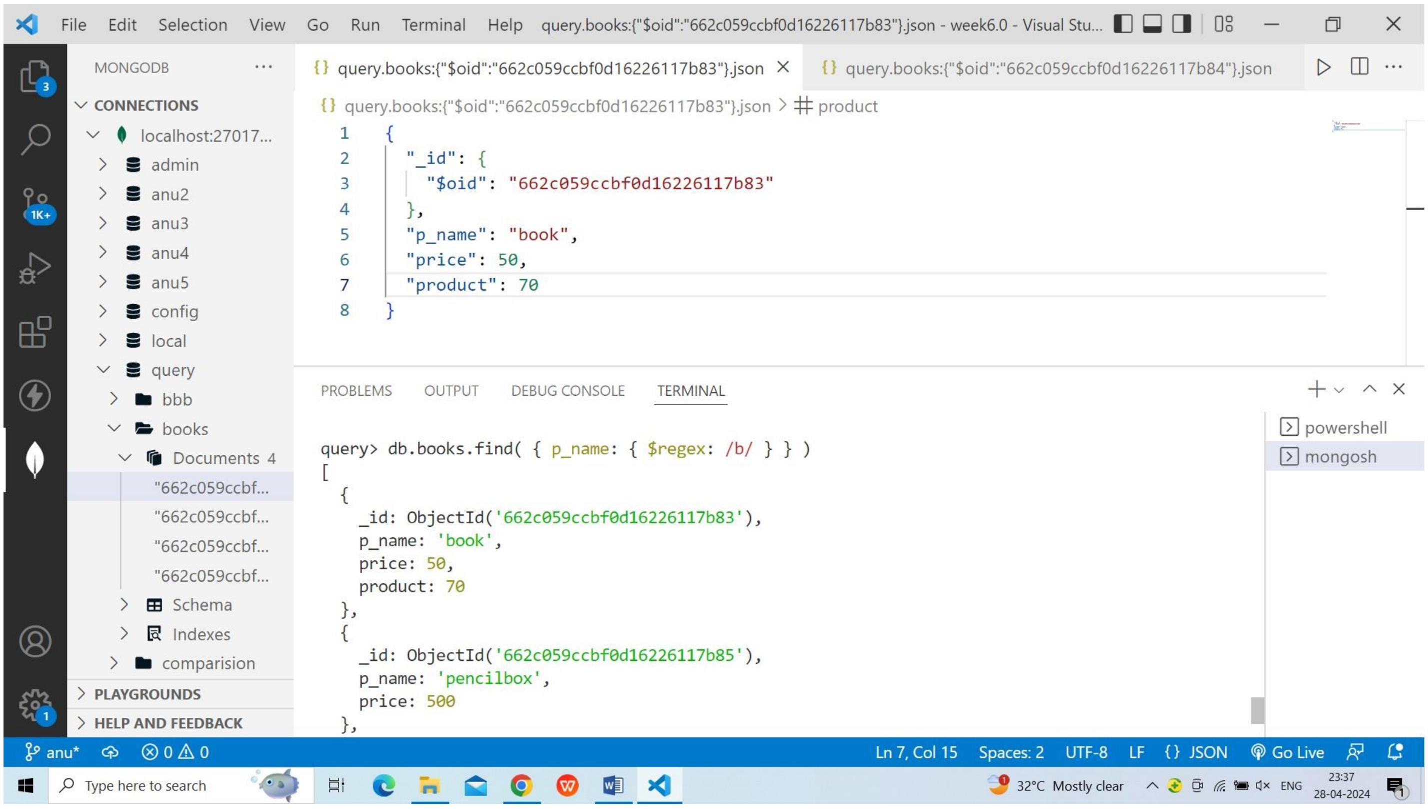
It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regularexpressions that are compatible with Perl.

Syntax:

1. { <field>: /pattern/<options> }

Example:

db.books.find({ p_name: { $regex: /b/ } })



```
{
  "_id": {
    "$oid": "662c059ccbf0d16226117b83"
  },
  "p_name": "book",
  "price": 50,
  "product": 70
}
```

```
query> db.books.find( { p_name: { $regex: /b/ } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  },
```

$text

The $text operator searches a text on the content of the field, indexed with a text index.

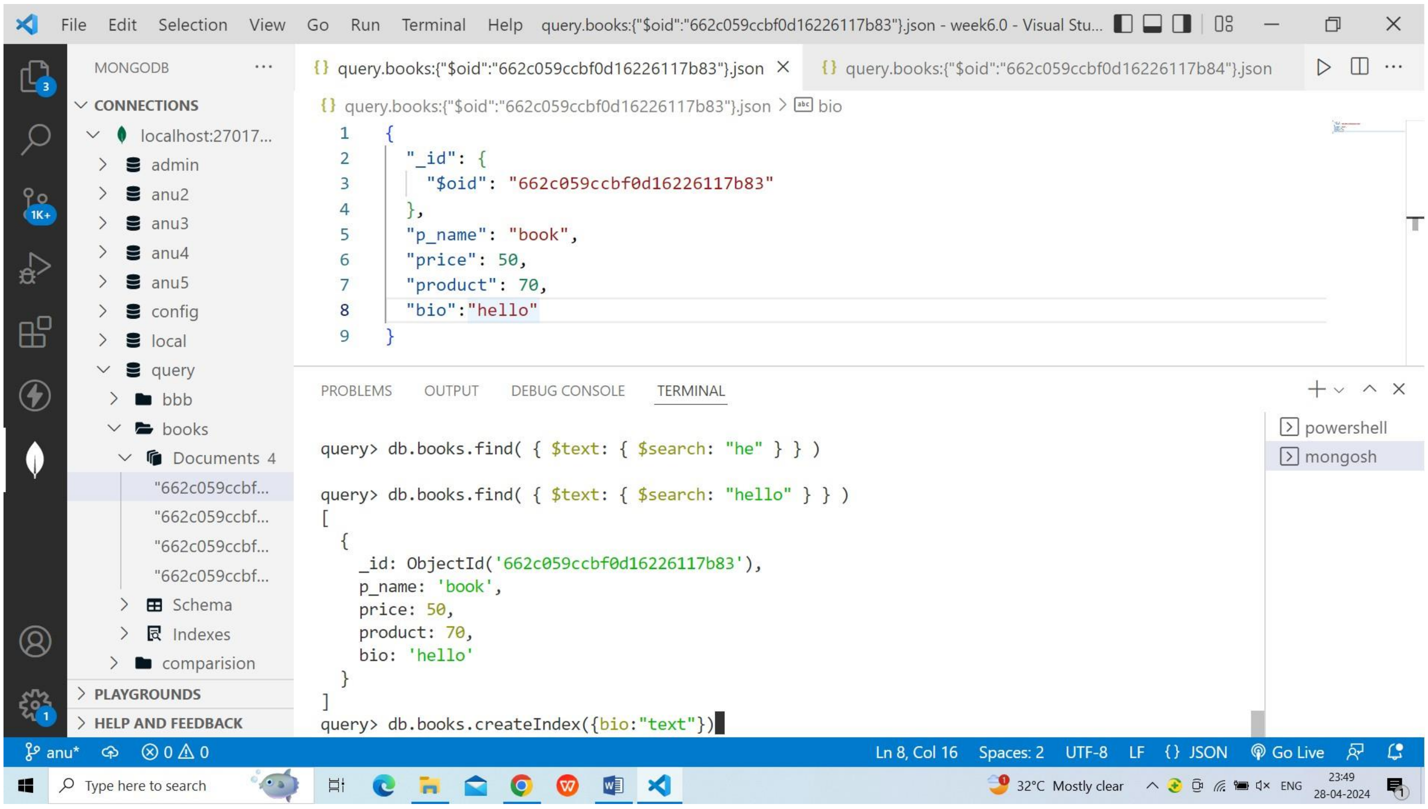
db.books.createIndex({bio:"text"})

Syntax:

1. {
2. $text:
3. {
4. $search: <string>,
5. $language: <string>,
6. $caseSensitive: <boolean>,
7. $diacriticSensitive:
<boolean>8. }
9. }

Example:

1. db.books.find({ $text: { $search: "hello" } })


File Edit Selection View Go Run Terminal Help query.books:{"$oid":"662c059ccbf0d16226117b83"}.json - week6.0 - Visual Stu...
MONGODB
CONNECTIONS
localhost:27017...
admin
anu2
anu3
anu4
anu5
config
local
query
bbb
books
Documents 4
"662c059ccbf...
Schema
Indexes
comparision
PLAYGROUNDS
HELP AND FEEDBACK
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
powershell
mongosh
Ln 8, Col 16 Spaces: 2 UTF-8 LF {} JSON Go Live

```
{
  "_id": {
    "$oid": "662c059ccbf0d16226117b83"
  },
  "p_name": "book",
  "price": 50,
  "product": 70,
  "bio":"hello"
}
```

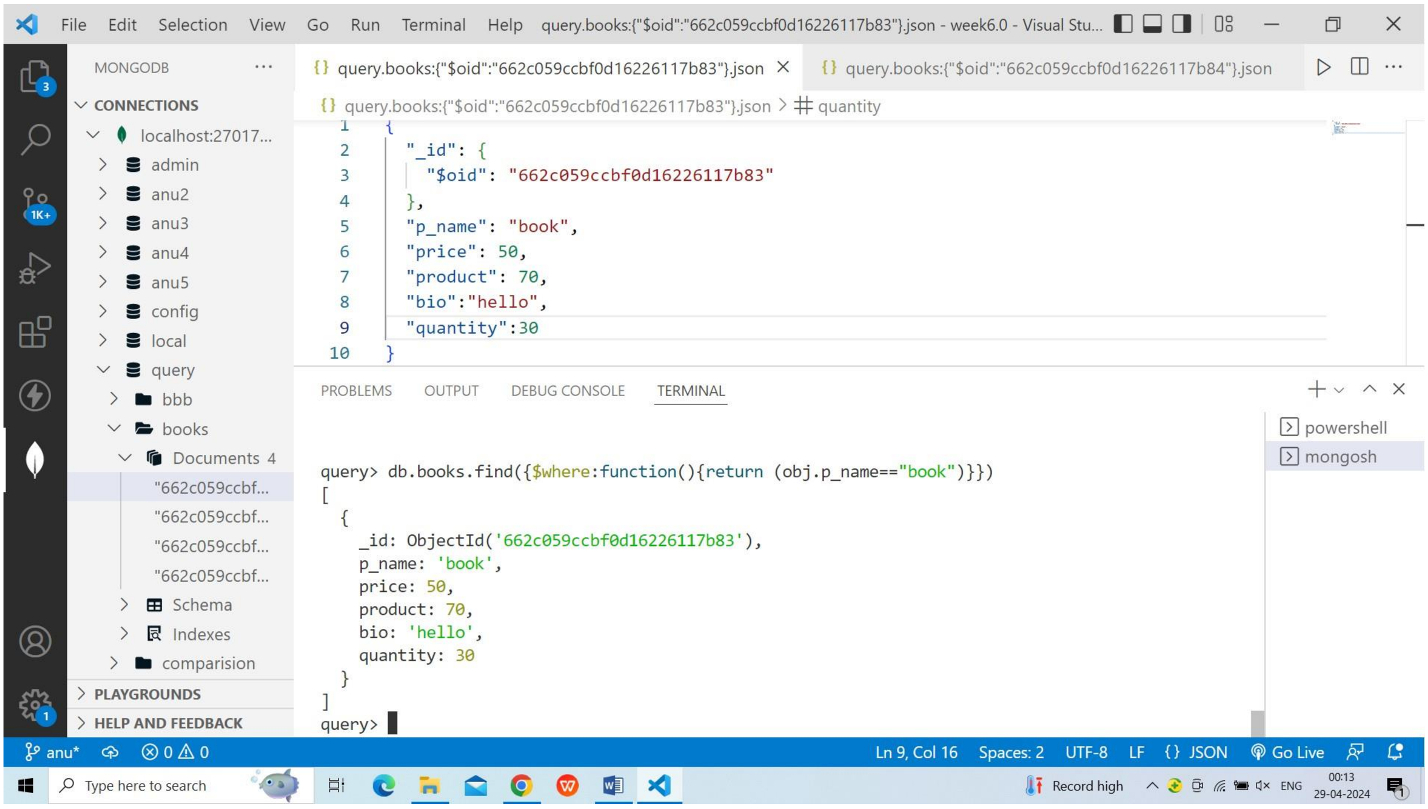
```
query> db.books.find( { $text: { $search: "he" } } )

query> db.books.find( { $text: { $search: "hello" } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70,
    bio: 'hello'
  }
]
query> db.books.createIndex({bio:"text"})
```

$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScriptfunction to the query system.

Example:

db.books.find({$where:function(){return (obj.p_name=="book")}})


File Edit Selection View Go Run Terminal Help query.books:{"$oid":"662c059ccbf0d16226117b83"}.json - week6.0 - Visual Stu...
MONGODB
CONNECTIONS
localhost:27017...
admin
anu2
anu3
anu4
anu5
config
local
query
bbb
books
Documents 4
"662c059ccbf...
Schema
Indexes
comparision
PLAYGROUNDS
HELP AND FEEDBACK
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
powershell
mongosh
Ln 9, Col 16 Spaces: 2 UTF-8 LF {} JSON Go Live

```
{
  "_id": {
    "$oid": "662c059ccbf0d16226117b83"
  },
  "p_name": "book",
  "price": 50,
  "product": 70,
  "bio":"hello",
  "quantity":30
}
```

```
query> db.books.find({$where:function(){return (obj.p_name=="book")}})
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50,
    product: 70,
    bio: 'hello',
    quantity: 30
  }
]
query>
```

5. Element Operators

The element operators in the MongoDB return the documents in the collection which returns trueif the keys match the fields and datatypes.
There are mainly two Element operators in MongoDB:

Element Operator	Description	Syntax
$exists	Checks if a specified field exists in the documents.	{ field: { $exists: <boolean> } }
$type	Verifies the data type of a specified field in the documents.	{ field: { $type: <BSON type> } }

MongoDB Element Operator

$exists

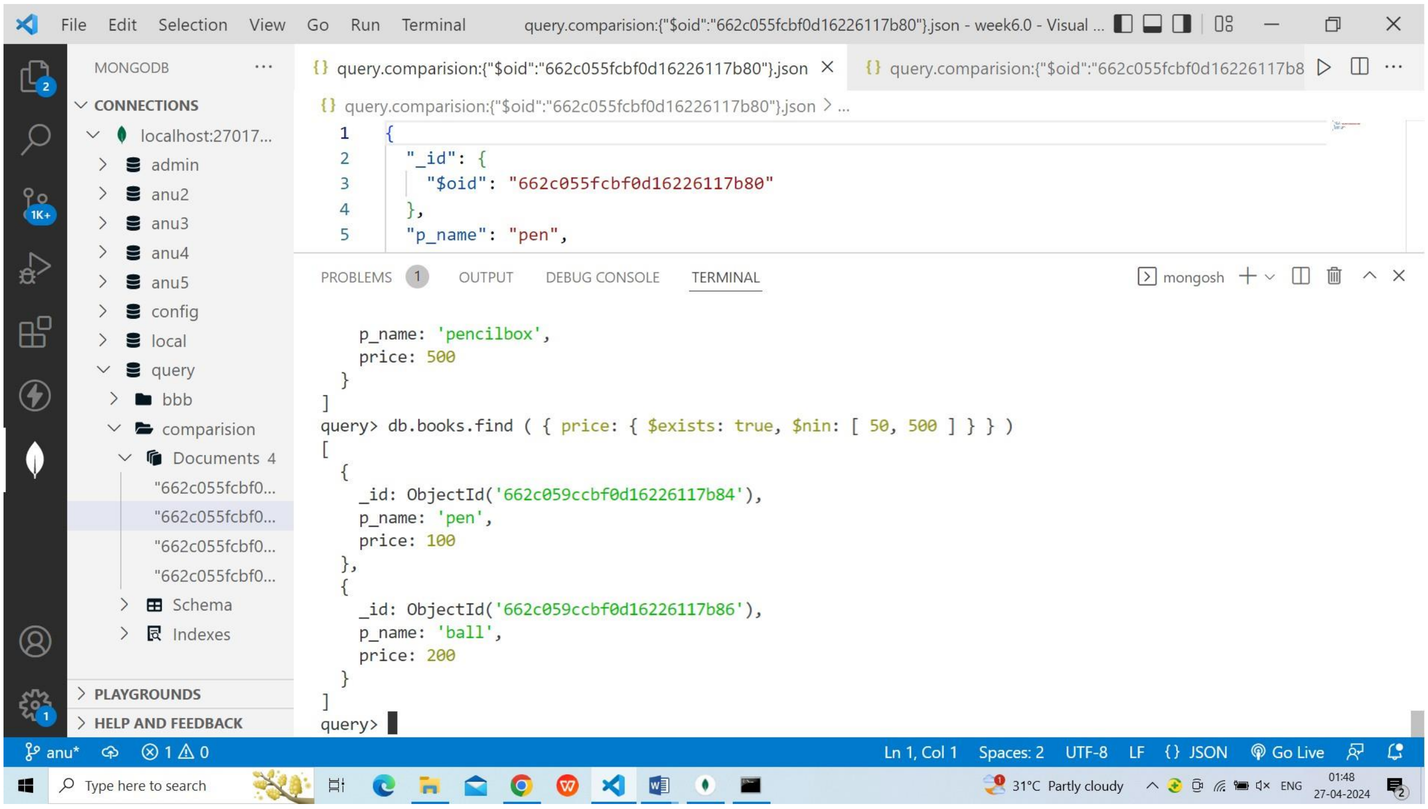
The exists operator matches the documents that contain the field when Boolean is true. It also matches thedocument where the field value is null.

Syntax:

```
1. { field: { $exists: <boolean> } }
```

Example:

```
db.books.find ( { price: { $exists: true, $nin: [ 50, 500 ] } } )
```

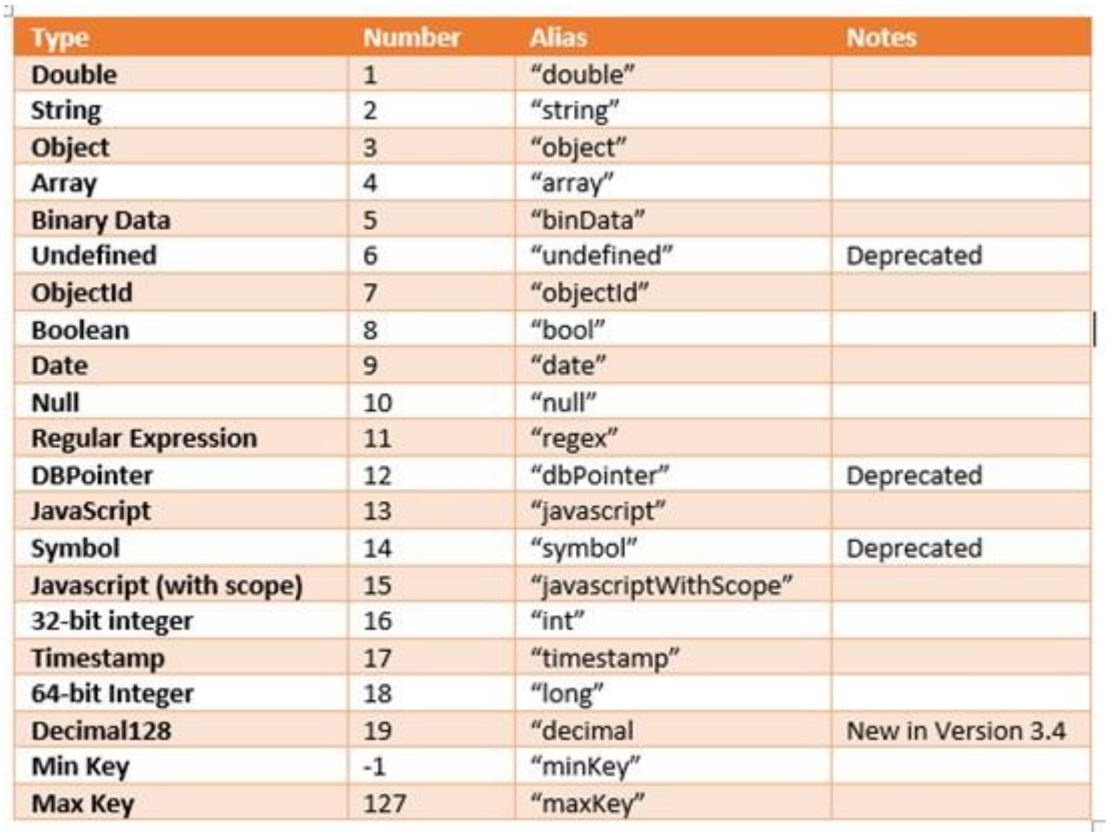


```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b80"
  },
  "p_name": "pen",
```

```
    p_name: 'pencilbox',
    price: 500
  }
]
query> db.books.find ( { price: { $exists: true, $nin: [ 50, 500 ] } } )
[
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
    price: 200
  }
]
query>
```

$type

The type operator chooses documents where the value of the field is an instance of the specified BSON type.



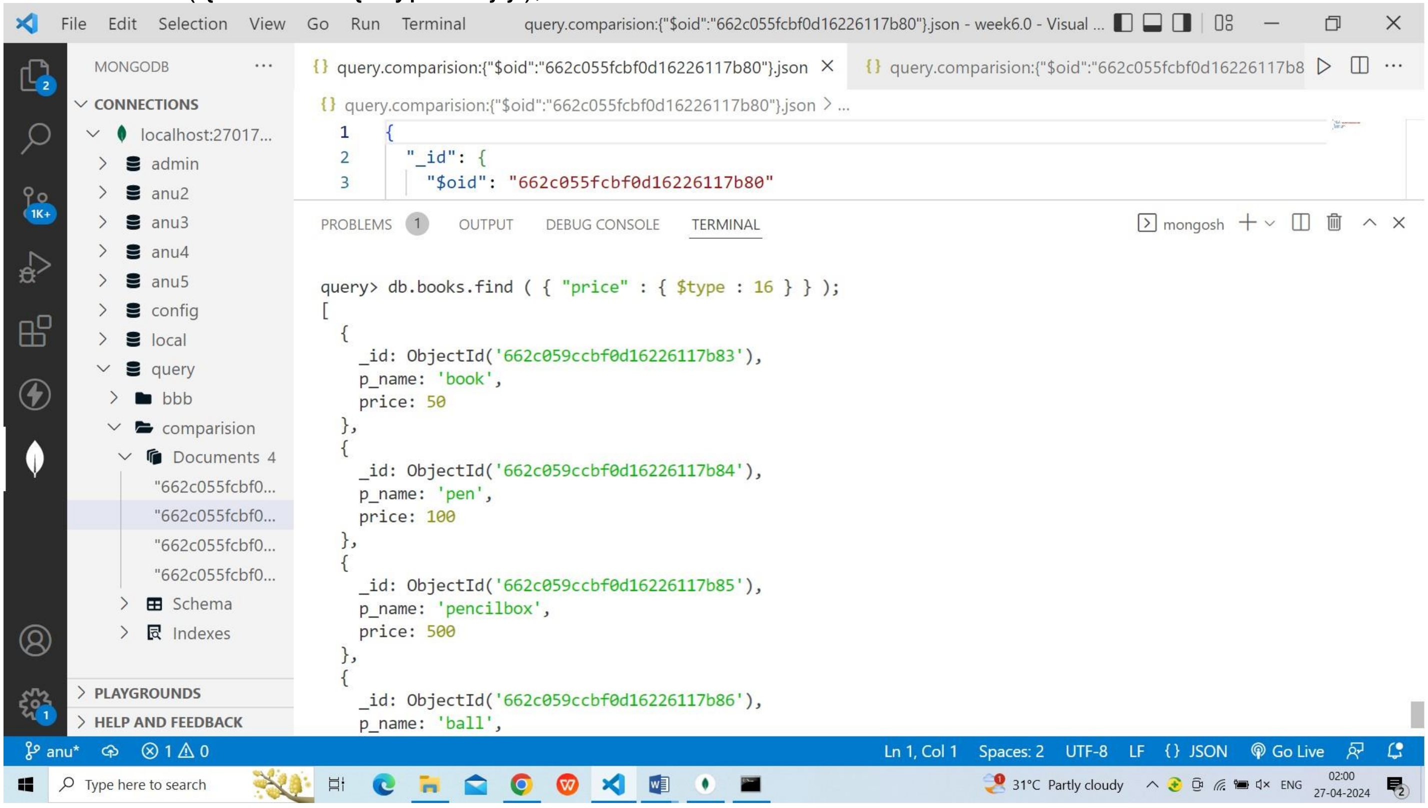
Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary Data	5	"binData"	
Undefined	6	"undefined"	Deprecated
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated
Javascript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit Integer	18	"long"	
Decimal128	19	"decimal	New in Version 3.4
Min Key	-1	"minKey"	
Max Key	127	"maxKey"	

Syntax:

```
1. { field: { $type: <BSON type> } }
```

Example:

1. db.books.find ({ "bookid" : { $type : 2 } });



```
{
  "_id": {
    "$oid": "662c055fcbf0d16226117b80"
```

```
query> db.books.find ( { "price" : { $type : 16 } } );
[
  {
    _id: ObjectId('662c059ccbf0d16226117b83'),
    p_name: 'book',
    price: 50
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b84'),
    p_name: 'pen',
    price: 100
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b85'),
    p_name: 'pencilbox',
    price: 500
  },
  {
    _id: ObjectId('662c059ccbf0d16226117b86'),
    p_name: 'ball',
```

6. Bitwise Operators

The Bitwise operators in the MongoDB return the documents in the MongoDB mainly on the fields that have **numeric values based on their bits** similar to other programming languages.

Bitwise Operator	**Description**	**Syntax**
$bitsAllClear	Returns documents where all bits in the specified field are 0.	{ field: { $bitsAllClear: <bitmask> } }
$bitsAllSet	Returns documents where all bits in the specified field are 1.	{ field: { $bitsAllSet: <bitmask> } }
$bitsAnySet	Returns documents where at least one bit in the specified field is set (1).	{ field: { $bitsAnySet: <bitmask> } }

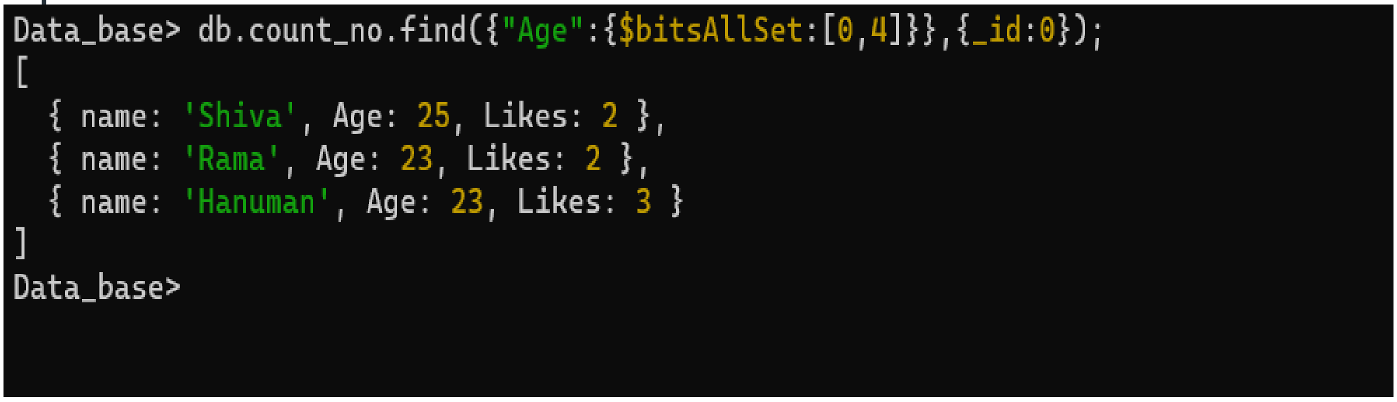
$bitsAnyClear	Returns documents where at least one bit in the specified field is clear (0).	{ field: { $bitsAnyClear: <bitmask> } }

In the below example, we also specified the positions we wanted.

Example of Using $bitsAllSet Operator
Let's find the **Person** whose **Age** has bit **1** from position **0 to 4**.
Query:

```
db.count_no.find({"Age":{$bitsAllSet: [0,4] } },{_id:0 } );
```

Output:



```
Data_base> db.count_no.find({"Age":{$bitsAllSet:[0,4]}},{_id:0});
[
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 }
]
Data_base>
```

$bitsAllSet in MongoDB

Explanation: In the above query, we have used $**bitsAllSet** and it returns documents whose **bits position from 0 to 4 are only ones.** It works only with the **numeric values**. The numeric values will be converted into the bits and the bits numbering takes place from the right.

7. Geospatial Operators

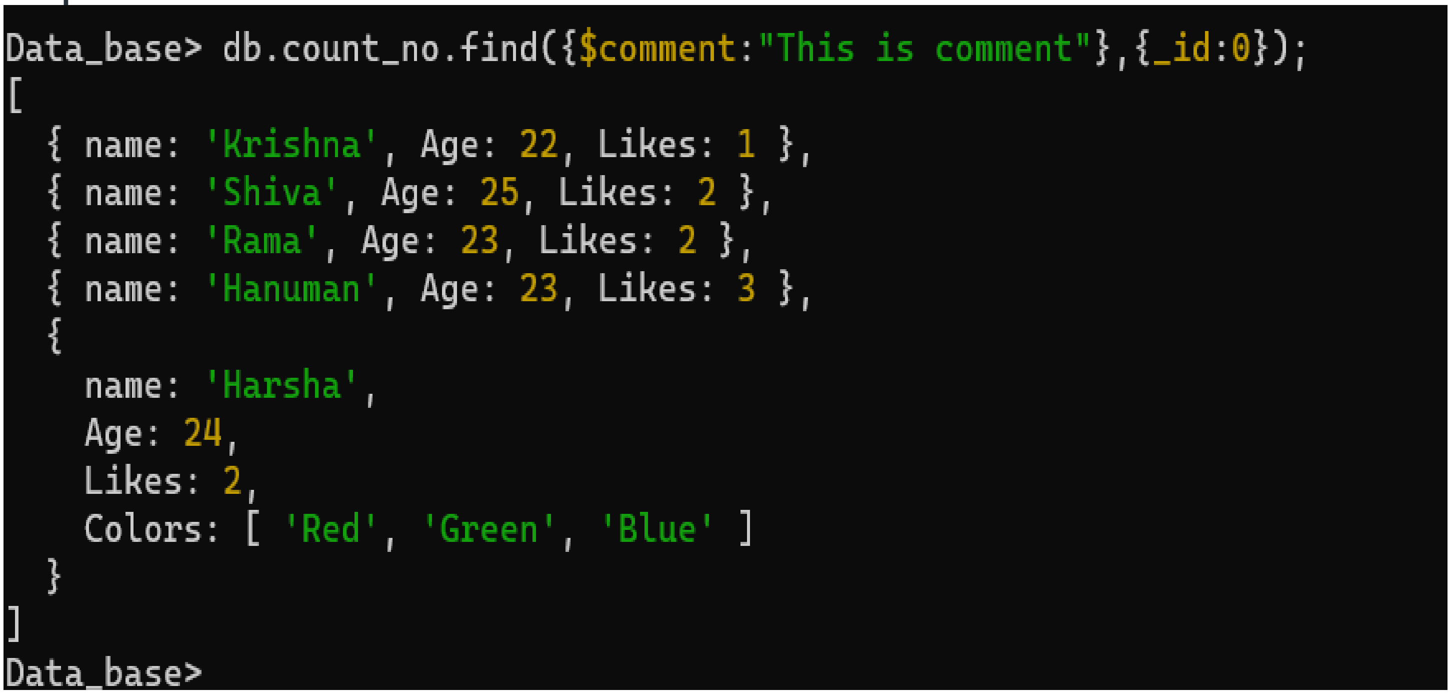
The Geospatial operators in the MongoDB are used mainly with the terms that relate to the data which mainly focuses on the directions such as **latitude** or **longitudes**.
The Geospatial operators in the MongoDB are:

Geospatial Operator	**Description**	**Syntax**
$near	Finds geospatial objects near a point. Requires a geospatial index.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> (optional) } }
$center	(For $geoWithin with planar geometry) Specifies a circle around a center point	{ $geoWithin: { $center: [<longitude>, <latitude>], radius: <distance> } }
$maxDistance	Limits results of $near and $nearSphere queries to a maximum distance from the point.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> } }

$minDistance	Limits results of $near and $nearSphere queries to a minimum distance from the point.	{ $near: { geometry: <point_geometry>, minDistance: <distance> } }

8. Comment Operators

The $comment operator in MongoDB is used to **write the comments along with the query in the MongoDB** which is used to easily understand the data.
Comment Operator Example
Let’s apply some comments in the queries using the **$comment** Operator.
Query:
```
db.collection_name.find( { $comment : comment })
```
Output:



```
Data_base> db.count_no.find({$comment:"This is comment"},{_id:0});
[
  { name: 'Krishna', Age: 22, Likes: 1 },
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 },
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```

$Comment operator in MongoDB

Explanation: In the above query we used the $comment operator to mention the comment. We have used “**This is a comment**” with $comment to specify the comment. The comment operator in the MongoDB is used to represent the comment and it increases the understandibility of thecode.

MongoDB Projection

MongoDB provides a special feature that is known as **Projection**. It allows you to select only thenecessary data rather than selecting whole data from the document. For example, a document contains 5 fields, i.e.,

```
{
name:
"Roma",age:
30, branch:
EEE,
department: "HR",
salary: 20000
```

But we only want to display the *name* and the *age* of the employee rather than displaying whole
details. Now, here we use projection to display the name and age of the employee.
One can use projection with db.collection.find() method. In this method, the second parameter is theprojection parameter, which is used to specify which fields are returned in the matching documents.

Syntax:

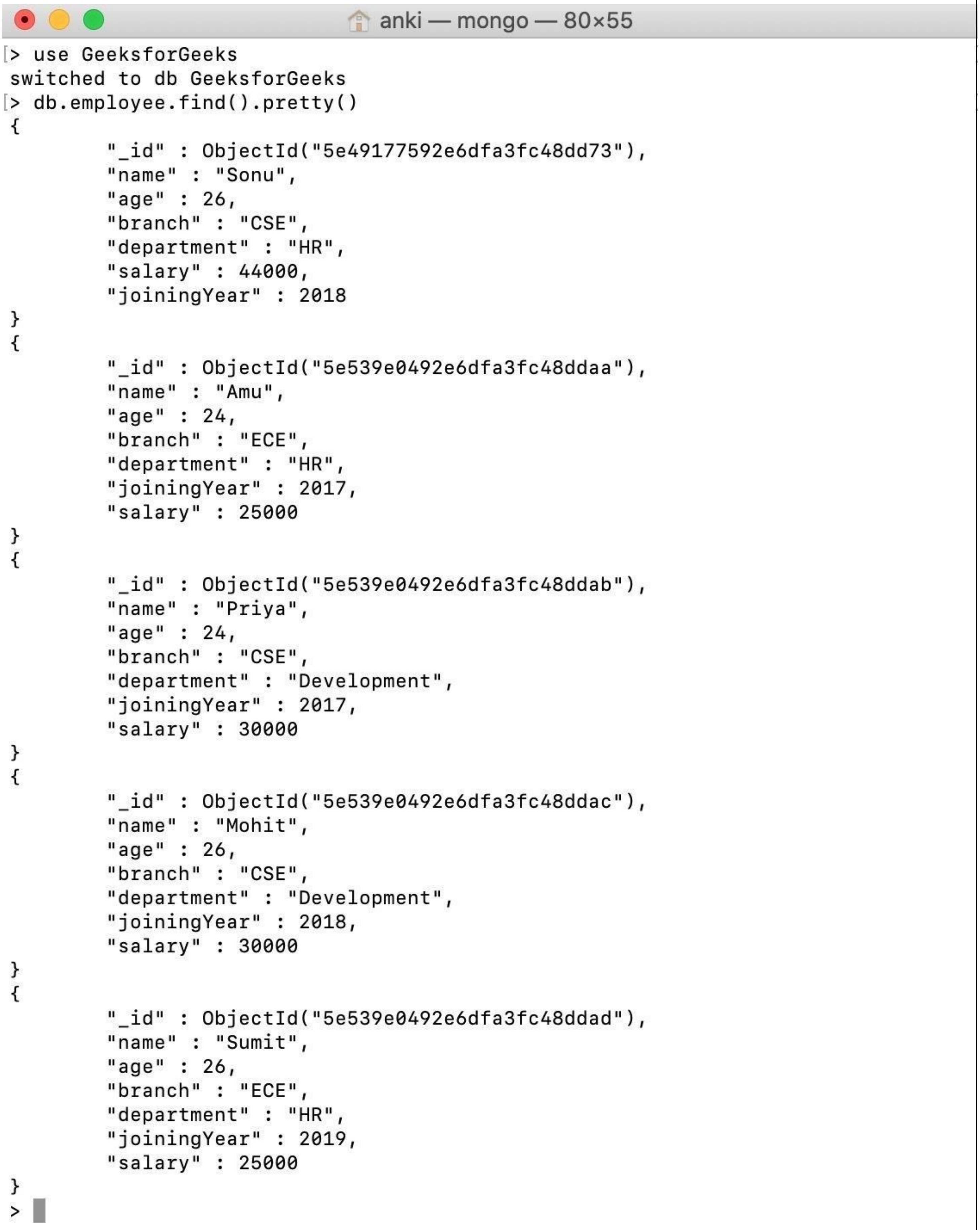
```
db.collection.find({}, {field1: value2, field2: value2, ..})
```

- If the value of the field is set to 1 or true, then it means the field will include in the returndocument.
- If the value of the field is set to 0 or false, then it means the field will not include in the returndocument.
- You are allowed to use projection operators.
- There is no need to set _id field to 1 to return _id field, the find() method always return _id unlessyou set a _id field to 0.

Examples:

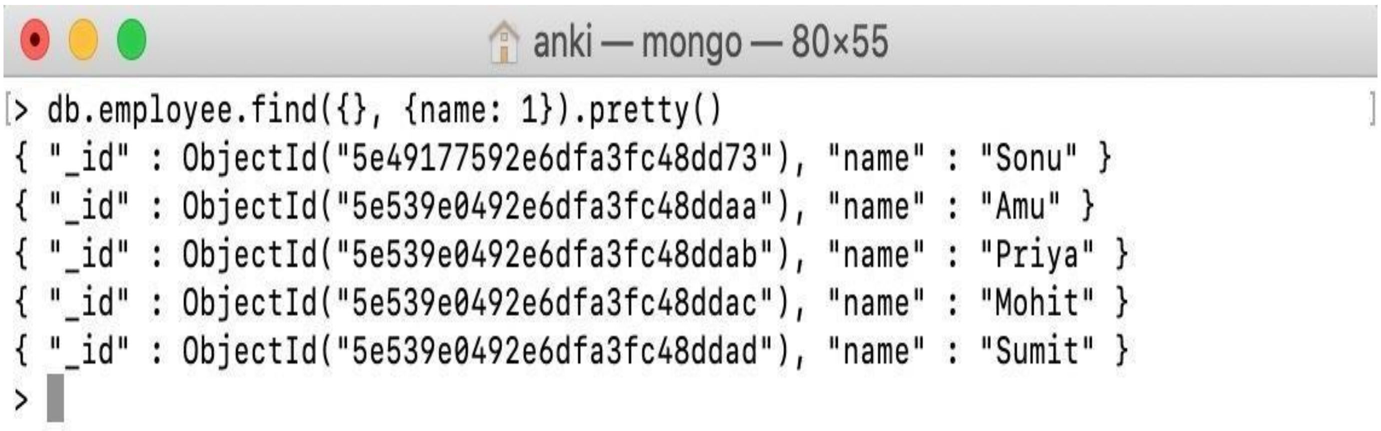
In the following examples, we are working with:

Database: GeeksforGeeks
Collection: employee
Document: five documents that contain the details of the employees in the form of field-value pairs.



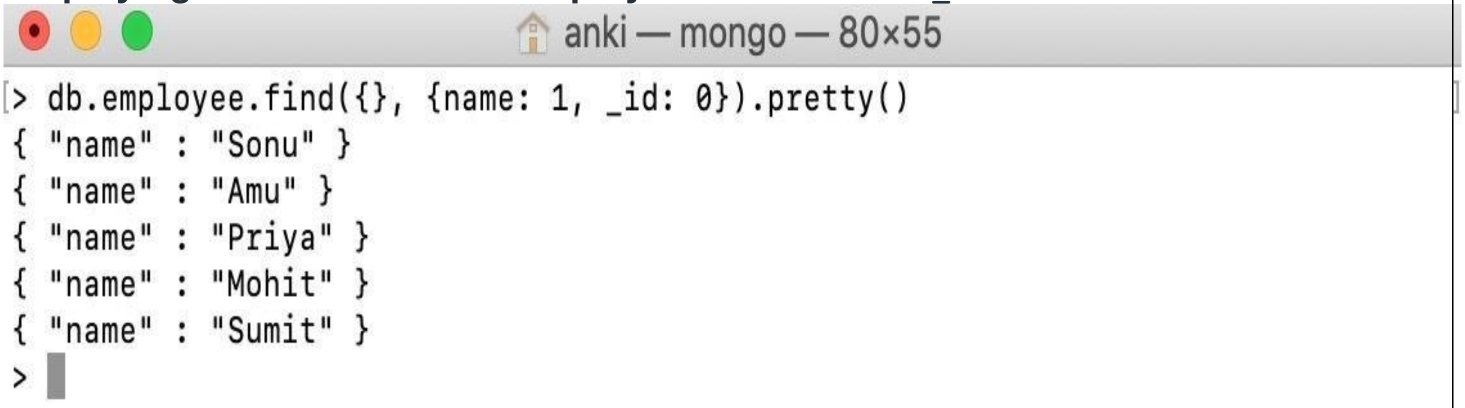
```
> use GeeksforGeeks
switched to db GeeksforGeeks
> db.employee.find().pretty()
{
        "_id" : ObjectId("5e49177592e6dfa3fc48dd73"),
        "name" : "Sonu",
        "age" : 26,
        "branch" : "CSE",
        "department" : "HR",
        "salary" : 44000,
        "joiningYear" : 2018
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddaa"),
        "name" : "Amu",
        "age" : 24,
        "branch" : "ECE",
        "department" : "HR",
        "joiningYear" : 2017,
        "salary" : 25000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddab"),
        "name" : "Priya",
        "age" : 24,
        "branch" : "CSE",
        "department" : "Development",
        "joiningYear" : 2017,
        "salary" : 30000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddac"),
        "name" : "Mohit",
        "age" : 26,
        "branch" : "CSE",
        "department" : "Development",
        "joiningYear" : 2018,
        "salary" : 30000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddad"),
        "name" : "Sumit",
        "age" : 26,
        "branch" : "ECE",
        "department" : "HR",
        "joiningYear" : 2019,
        "salary" : 25000
}
>
```

Displaying the names of the employees –



```
> db.employee.find({}, {name: 1}).pretty()
{ "_id" : ObjectId("5e49177592e6dfa3fc48dd73"), "name" : "Sonu" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddaa"), "name" : "Amu" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddab"), "name" : "Priya" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddac"), "name" : "Mohit" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddad"), "name" : "Sumit" }
>
```

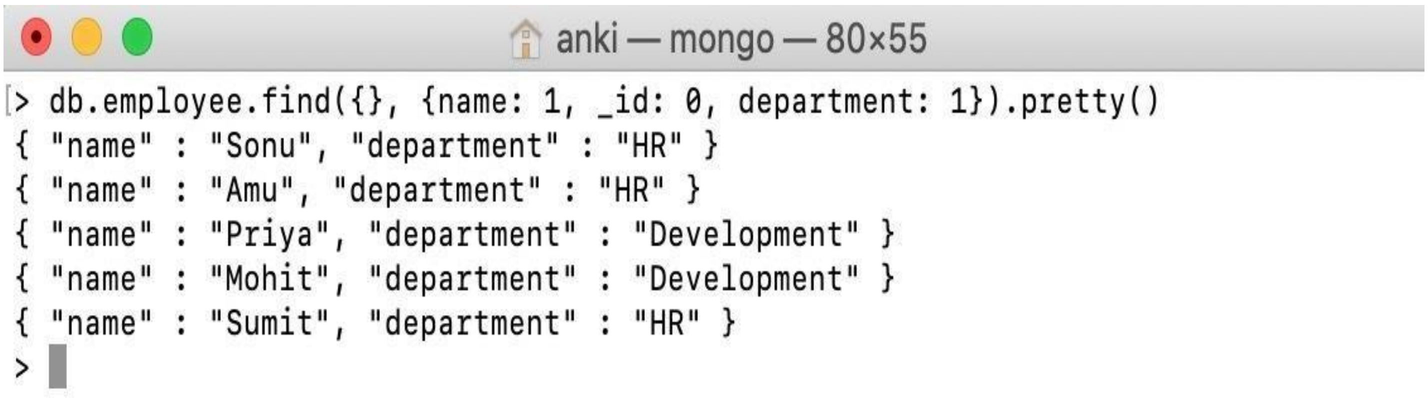
Displaying the names of the employees without the `_id` field –



```
> db.employee.find({}, {name: 1, _id: 0}).pretty()
{ "name" : "Sonu" }
{ "name" : "Amu" }
{ "name" : "Priya" }
{ "name" : "Mohit" }
{ "name" : "Sumit" }
>
```

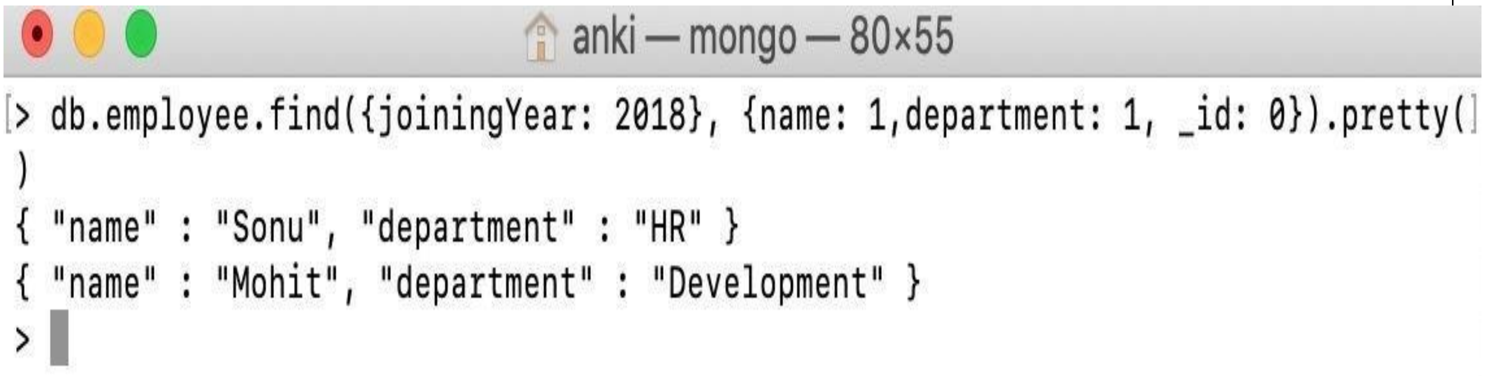
Displaying the name and the department of the employees without the `_id` field

–



```
> db.employee.find({}, {name: 1, _id: 0, department: 1}).pretty()
{ "name" : "Sonu", "department" : "HR" }
{ "name" : "Amu", "department" : "HR" }
{ "name" : "Priya", "department" : "Development" }
{ "name" : "Mohit", "department" : "Development" }
{ "name" : "Sumit", "department" : "HR" }
>
```

Displaying the names and the department of the employees whose joining year is 2018 –


anki — mongo — 80×55

```
> db.employee.find({joiningYear: 2018}, {name: 1,department: 1, _id: 0}).pretty(
)
{ "name" : "Sonu", "department" : "HR" }
{ "name" : "Mohit", "department" : "Development" }
>
```

Projection Operators

Name	Description
`$`	Projects the first element in an array that matches the query condition.
`$elemMatch`	Projects the first element in an array that matches the specified `$elemMatch` condition.
`$meta`	Projects the document's score assigned during the `$text` operation. **NOTE** `$text` provides text query capabilities for self-managed (non-Atlas) deployments. For data hosted on MongoDB Atlas, MongoDB offers an improved full-text query solution, Atlas Search.
`$slice`	Limits the number of elements projected from an array. Supports skip and limit slices.

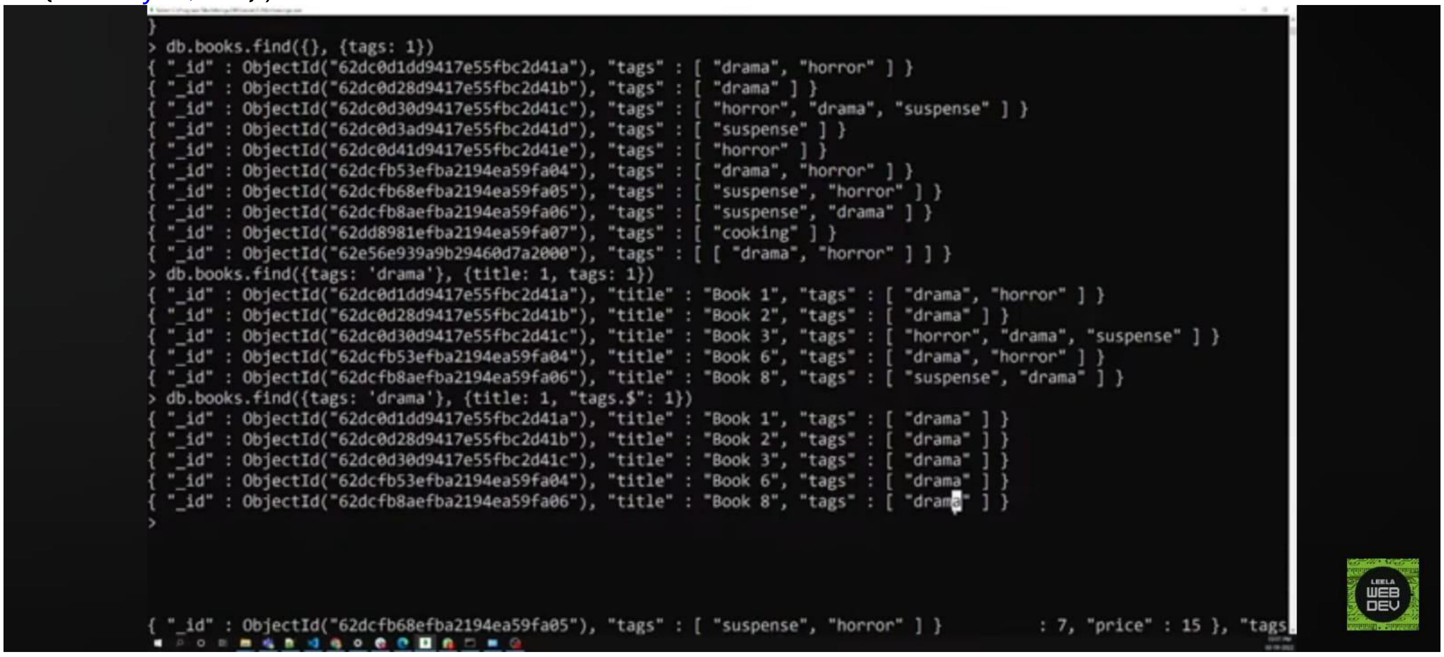
MongoDB Projection Operator

$

The $ operator limits the contents of an array from the query results to contain only the first element matching the query document.

Syntax:

```
db.books.find( { <array>: <value> ... },
      { "<array>.$": 1 } )
db.books.find( { <array.field>: <value> ...},
      { "<array>.$": 1 } )
```



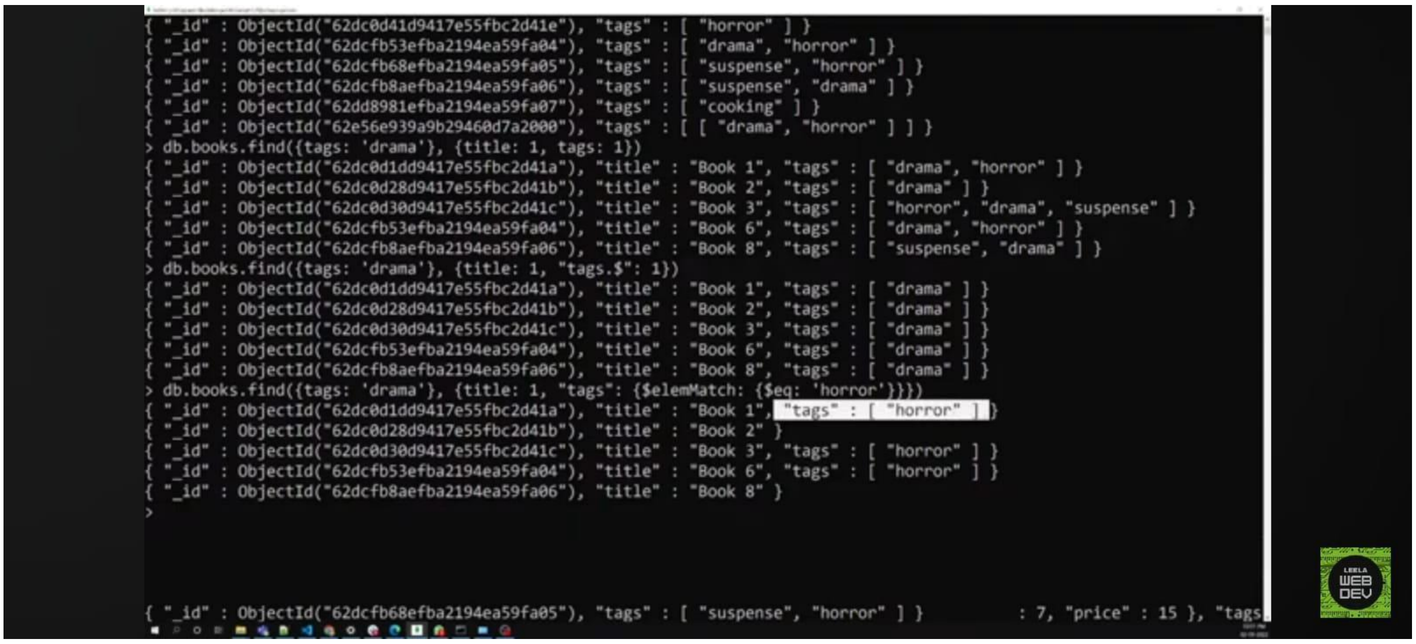
```
}
> db.books.find({}, {tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "tags" : [ "horror", "drama", "suspense" ] }
{ "_id" : ObjectId("62dc0d3ad9417e55fbc2d41d"), "tags" : [ "suspense" ] }
{ "_id" : ObjectId("62dc0d41d9417e55fbc2d41e"), "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : [ "suspense", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "tags" : [ "suspense", "drama" ] }
{ "_id" : ObjectId("62dd8981efba2194ea59fa07"), "tags" : [ "cooking" ] }
{ "_id" : ObjectId("62e56e939a9b29460d7a2000"), "tags" : [ [ "drama", "horror" ] ] }
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror", "drama", "suspense" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "suspense", "drama" ] }
> db.books.find({tags: 'drama'}, {title: 1, "tags.$": 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "drama" ] }
>


{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : [ "suspense", "horror" ] }          : 7, "price" : 15 }, "tags
```

$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element $elemMatch condition.

Syntax:

```
db.library.find( { bookcode: "63109" },
{ students: { $elemMatch: { roll: 102 } } } )
```



```
{ "_id" : ObjectId("62dc0d41d9417e55fbc2d41e"), "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : [ "suspense", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "tags" : [ "suspense", "drama" ] }
{ "_id" : ObjectId("62dd8981efba2194ea59fa07"), "tags" : [ "cooking" ] }
{ "_id" : ObjectId("62e56e939a9b29460d7a2000"), "tags" : [ [ "drama", "horror" ] ] }
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror", "drama", "suspense" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "suspense", "drama" ] }
> db.books.find({tags: 'drama'}, {title: 1, "tags.$": 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "drama" ] }
> db.books.find({tags: 'drama'}, {title: 1, "tags": {$elemMatch: {$eq: 'horror'}}})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2" }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8" }
>



{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : [ "suspense", "horror" ] }          : 7, "price" : 15 }, "tags
```

$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

Syntax:

1. { $meta: <metaDataKeyword> }

Example:

1. db.books.find(
2. <query>,
3. { score: { $meta: "textScore" } }

$slice

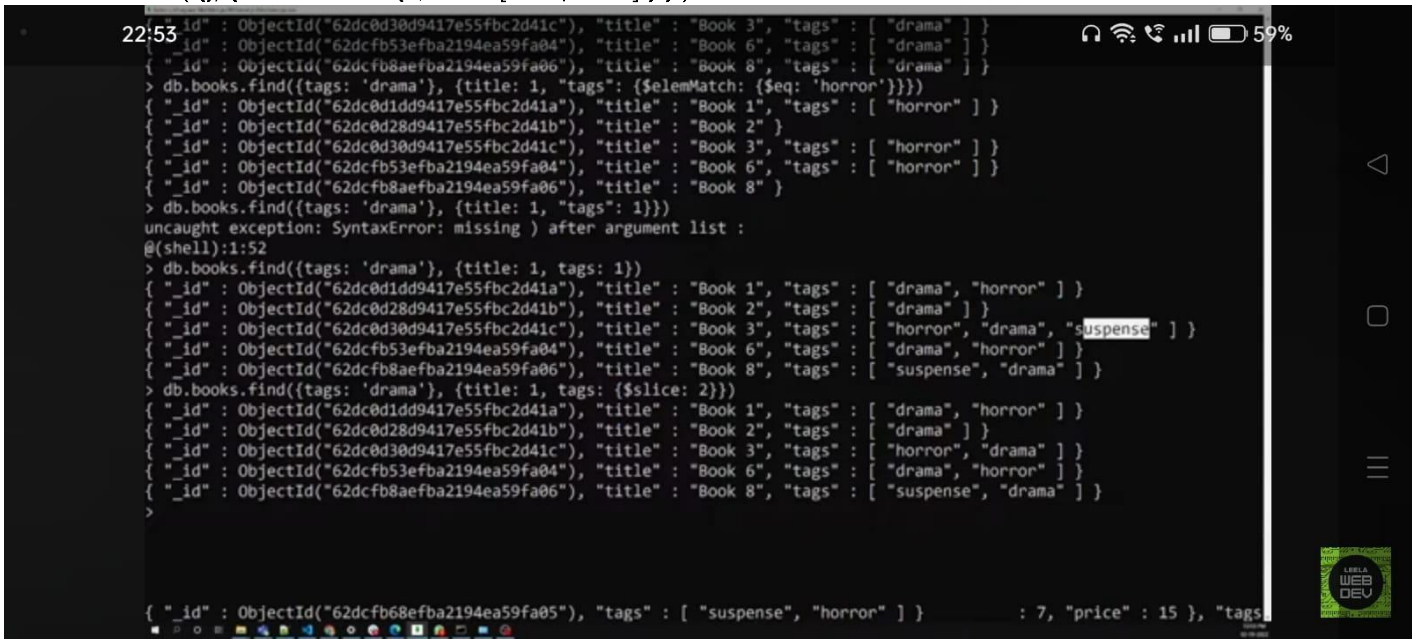
It controls the number of values in an array that a query returns.

Syntax:

1. db.books.find({ field: value }, { array: {$slice: count } });

Example:

1. db.books.find({}, { comments: { $slice: [200, 100] } })



```
  "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "drama" ] }
  "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "drama" ] }
> db.books.find({tags: 'drama'}, {title: 1, "tags": {$elemMatch: {$eq: 'horror'}}})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2" }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8" }
> db.books.find({tags: 'drama'}, {title: 1, "tags": 1}})
uncaught exception: SyntaxError: missing ) after argument list :
@(shell):1:52
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror", "drama", "suspense" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "suspense", "drama" ] }
> db.books.find({tags: 'drama'}, {title: 1, tags: {$slice: 2}})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : [ "drama" ] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : [ "horror", "drama" ] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : [ "drama", "horror" ] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : [ "suspense", "drama" ] }
>


{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : [ "suspense", "horror" ] }          : 7, "price" : 15 }, "tags
```

4. Aggregation Pipeline

a. What is Aggregation in MongoDB?

Aggregation is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort,group, reshape and modify documents that pass through the pipeline.

One of the most common use cases of Aggregation is to calculate aggregate values for groups of documents. Thisis similar to the basic aggregation available in SQL with the GROUP BY clause and COUNT, SUM and AVG functions. MongoDB Aggregation goes further though and can also perform relational-like joins, reshape documents, create new and update existing collections, and so on.

There are what are called **single purpose methods** like estimatedDocumentCount(), count(), and distinct() whichare appended to a find() query making them quick to use but limited in scope.

- Each stage of the pipeline transforms the documents as they pass through it and allowing for operationslike **filtering**, **grouping**, **sorting**, **reshaping** and performing calculations on the data.

b. MongoDB aggregate pipeline syntax

This is an example of how to build an aggregation query:

`db.`*collectionName*`.aggregate(`*pipeline, options*`)`,

- where *collectionName* - is the name of a collection,
- *pipeline* - is an array that contains the aggregation stages,
- *options* - optional parameters for the

aggregation This is an example of the

```
pipeline = [

    { $match : { … } },

    { $group : { … } },

    { $sort : { … } }

   ]
```
aggregation pipeline syntax:

c. Single-purpose aggregation

- It is used when we need simple access to document like counting the number of documents or for findingall distinct values in a document.
- It simply provides the access to the common aggregation process using

the **count()**, **distinct()** and **estimatedDocumentCount()** methods so due to which it lacks the flexibilityand capabilities of the pipeline.

Example of Single-purpose aggregation

Let’s consider a single-purpose aggregation example where we find the total number of users in each city fromthe `users` collection.

```
db.users.aggregate([
  { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```
Output:
```
[
  { _id: 'Los Angeles', totalUsers: 1 },
  { _id: 'New York', totalUsers: 1 },
  { _id: 'Chicago', totalUsers: 1 }
]
```
In this example, the aggregation pipeline first groups the documents by the `city` field and then usesthe `$sum` accumulator to count the number of documents (users) in each city.
The result will be a list of documents, each containing the city (`_id`) and the total number of users (`totalUsers`)in that city.

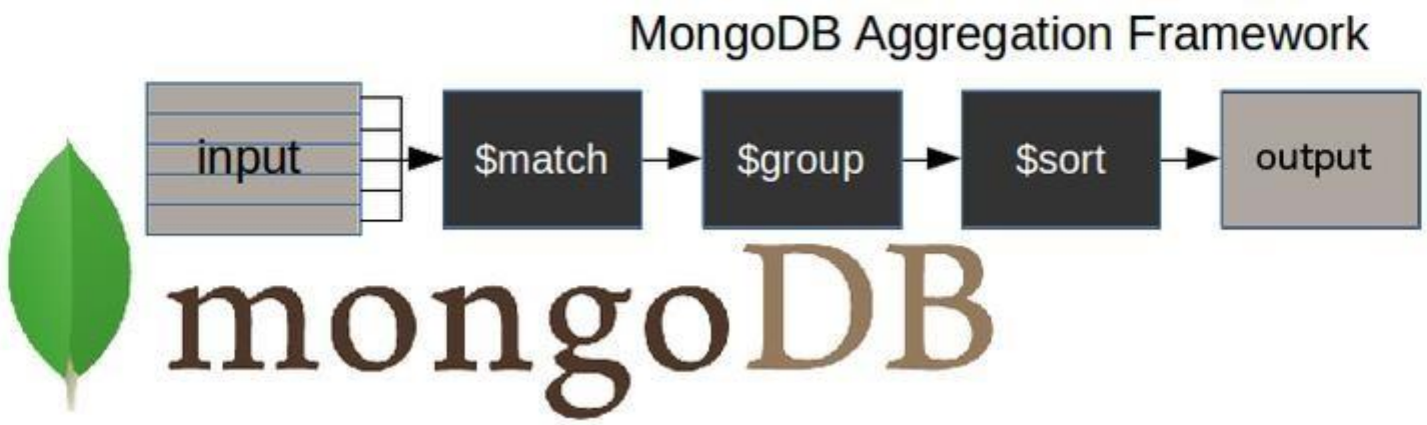
d. How to use MongoDB to Aggregate Data?

To use MongoDB for aggregating data, follow below steps:

1. **Connect to MongoDB**: Ensure you are connected to your MongoDB instance.
2. **Choose the Collection**: Select the collection you want to perform aggregation on, such as students.
3. **Define the Aggregation Pipeline**: Create an array of stages, like $group to group documents and performoperations (e.g., calculate the average grade).
4. **Run the Aggregation Pipeline**: Use the aggregate method on the collection with your defined pipeline.

Example:
```
db.students.aggregate([
  {
    $group: {
      _id: null,
      averageGrade: { $avg: "$grade" }
    }
  }
])
```
This calculates the average grade of all students in the `students` collection.

e. Mongodb Aggregation Pipeline



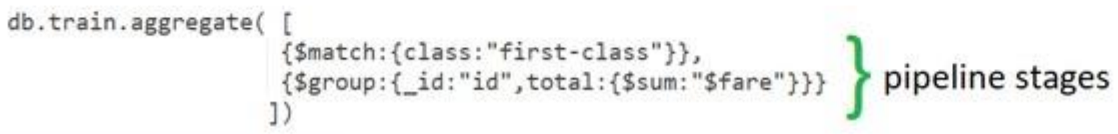
- **Mongodb Aggregation Pipeline** consist of stages and each stage transforms the document. It is a **multi-stage pipeline** and in each state and the documents are taken as input to produce the resultant set of documents.

- In the next stage (ID available) the resultant documents are taken as input to produce output, this processcontinues till the last stage.
- The **basic pipeline stages** are defined below:
  1. filters that will operate like queries.

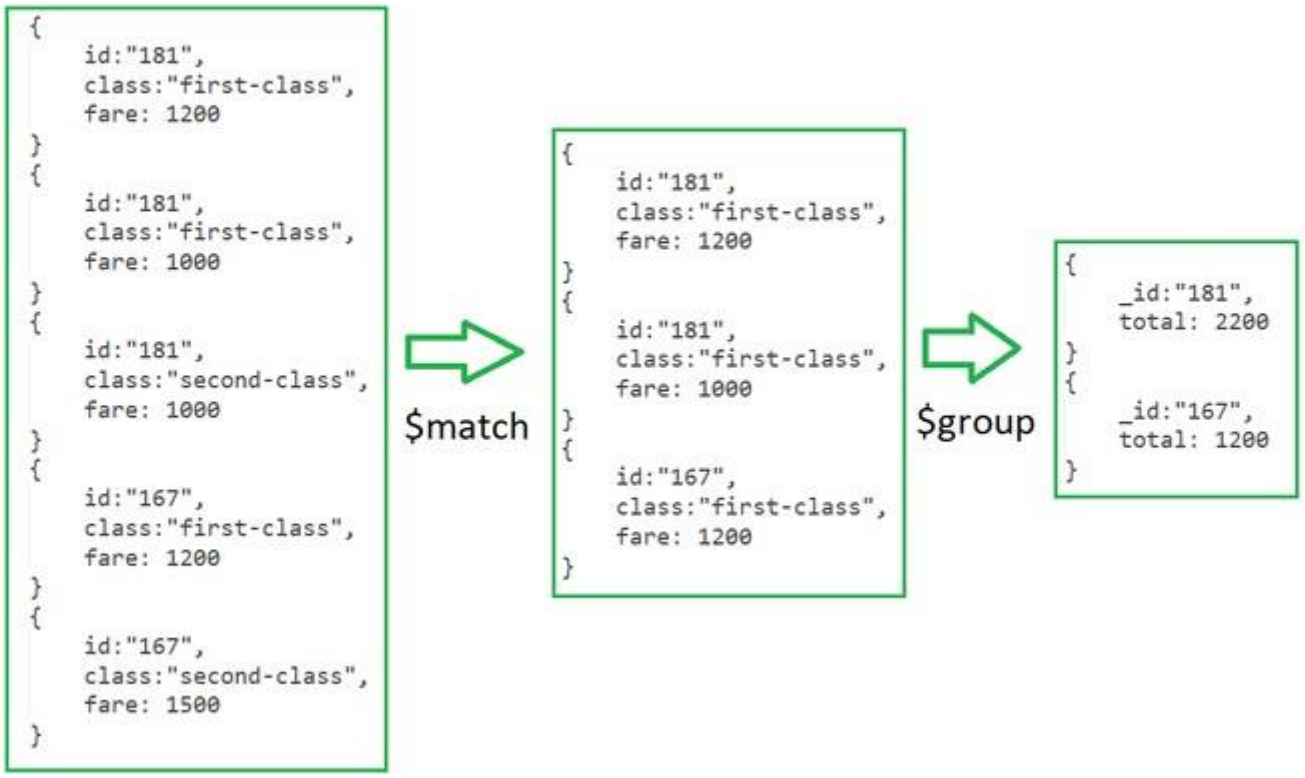
2. the document transformation that modifies the resultant document.
3. provide pipeline provides tools for **grouping and sorting documents.**

- Aggregation pipeline can also be used in **sharded collection**.

Example:



```
db.train.aggregate( [
                    {$match:{class:"first-class"}},
                    {$group:{_id:"id",total:{$sum:"$fare"}}}
                   ])
```

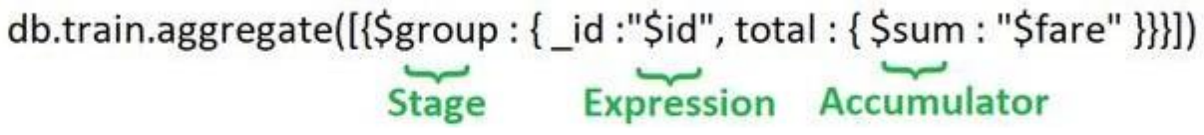


```
{
    id:"181",
    class:"first-class",
    fare: 1200
}
{
    id:"181",
    class:"first-class",
    fare: 1000
}
{
    id:"181",
    class:"second-class",
    fare: 1000
}
{
    id:"167",
    class:"first-class",
    fare: 1200
}
{
    id:"167",
    class:"second-class",
    fare: 1500
}
```

```
{
    id:"181",
    class:"first-class",
    fare: 1200
}
{
    id:"181",
    class:"first-class",
    fare: 1000
}
{
    id:"167",
    class:"first-class",
    fare: 1200
}
```

```
{
    _id:"181",
    total: 2200
}
{
    _id:"167",
    total: 1200
}
```

Explanation:
In the above example of a collection of "train fares". **$match** stage filters the documents by the value in classfield i.e. class: "first-class" in the first stage and passes the document to the second stage.
In the Second Stage, the **$group stage** groups the documents by the id field to calculate the sum of fare for each unique id.
Here, the **aggregate() function** is used to perform aggregation. It can have three operators **stages** , **expression** and **accumulator.** These operators work together to achieve final desired outcome.



```
db.train.aggregate([{$group : { _id :"$id", total : { $sum : "$fare" }}}])
```

f. Aggregation Pipeline Method

To understand Aggregation Pipeline Method Let's imagine a collection named **users** with some documents for our examples.

```
{
  "_id":
  ObjectId("60a3c7e96e06f64fb5ac0700"),
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
{
  "_id":
  ObjectId("60a3c7e96e06f64fb5ac0701"),
  "name": "Bob",
  "age": 35,
  "email":
  "bob@example.com","city":
  "Los Angeles"
}
{
  "_id":
  ObjectId("60a3c7e96e06f64fb5ac0702"),
  "name": "Charlie",
  "age": 25,
```

0. **$group:** It Groups documents by the `city` field and calculates the average age usingthe **$avg** accumulator.

```
db.users.aggregate([
  { $group: { _id: "$city", averageAge: { $avg: "$age" } } }
])
```

Output:

```
[
  { _id: 'New York', averageAge: 30 },
  { _id: 'Chicago', averageAge: 25 },
  { _id: 'Los Angeles', averageAge: 35 }
]
```

1. **$project**: Include or exclude fields from the output documents.

```
db.users.aggregate([
  { $project: { name: 1, city: 1, _id: 0 } }
])
```

Output:

```
[
  { name: 'Alice', city: 'New York' },
  { name: 'Bob', city: 'Los Angeles' },
  { name: 'Charlie', city: 'Chicago' }
]
```

2. **$match**: Filter documents to pass only those that match the specified condition(s).

```
db.users.aggregate([
  { $match: { age: { $gt: 30 } } }
])
```

Output:

```
[
  {
    _id:
    ObjectId('60a3c7e96e06f64fb5ac0701'),
    name: 'Bob',
    age: 35,
    email:
  }
]
```

3. **$sort**: It Order the documents.

```
db.users.aggregate([
  { $sort: { age: 1 } }
])
```

Output:

```
[
  {
    _id:
    ObjectId('60a3c7e96e06f64fb5ac0702'),
    name: 'Charlie',
    age: 25,
    email:
    'charlie@example.com',city:
    'Chicago'
  },
  {
    _id:
    ObjectId('60a3c7e96e06f64fb5ac0700'),
    name: 'Alice',
    age: 30,
    email:
    'alice@example.com',city:
    'New York'
  },
  {
    _id:
    ObjectId('60a3c7e96e06f64fb5ac0701'),
    name: 'Bob',
```

4. **$limit**: Limit the number of documents passed to the next stage.

```
db.users.aggregate([
  { $limit: 2 }
])
```

Output:

```
[
 {
   _id:
   ObjectId('60a3c7e96e06f64fb5ac0700'),
   name: 'Alice',
   age: 30,
   email:
   'alice@example.com',city:
   'New York'
 },
 {
   _id:
   ObjectId('60a3c7e96e06f64fb5ac0701'),
   name: 'Bob',
   age: 35,
   email:
```

g. How Fast is MongoDB Aggregation?

- The speed of MongoDB aggregation depends on various factors such as the complexity of the aggregation pipeline, the size of the data set, the hardware specifications of the MongoDB server and the efficiency of the indexes.
- In general, MongoDB’s aggregation framework is designed to efficiently process large volumes of dataand complex aggregation operations. When used correctly it can provide fast and scalable aggregation capabilities.
- So with any database operation, the performance can vary based on the specific use case and configuration. It is important to optimize our aggregation queries and use indexes where appropriate andensure that our MongoDB server is properly configured for optimal performance.

How to Insert a Document into a MongoDB Collection using Node.js?

MongoDB, a popular NoSQL database, offers flexibility and scalability for handling data. Ifyou're developing a Node.js application and need to interact with MongoDB, one of the fundamental operations you'll perform is inserting a document into a collection. This article provides a step-by-step guide on how to accomplish this using Node.js.

Prerequisites:

- NPM
- NodeJS
- MongoDB

The steps to insert documents in MongoDB collection are given below

Table of Content

Steps to Setup the Project

Step 1: Create a nodeJS application by using this command

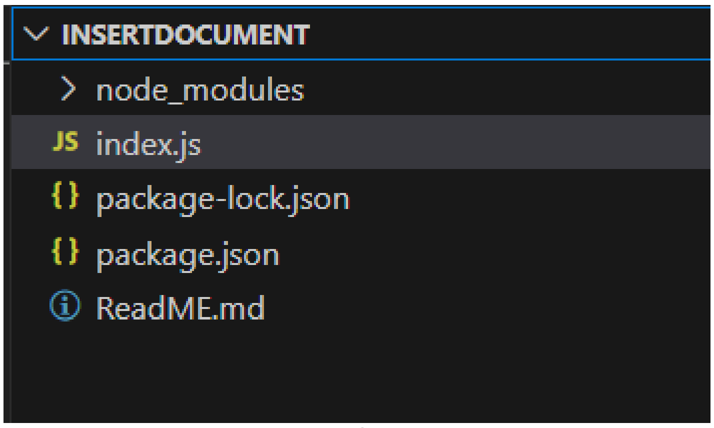
```
npm
initor
npm init -y
```

- **npm init** command asks some setup questions that are important for the project
- **npm init -y** command is used to set all the answers of the setup questions as **yes**.

Step 2: Install the necessary packages/libraries in your project using the following commands.

```
npm install mongodb
```

Project Structure:



Project Structure

The updated dependencies in package.json file will look like:

```
"dependencies": {
   "mongodb":
   "^6.6.1"
```

NodeJS and MongoDB Connection

Once the MongoDB is installed we can use MongoDB database with the Nodejs Project.Initiallywe need to specify the database name ,connection URL and the

```
const { MongoClient } = require('mongodb');
// or as an ecmascript module:
// import { MongoClient } from 'mongodb'

// Connection URL
const url =
'mongodb://localhost:27017';const
client = new MongoClient(url);

const dbName = 'project_name';  // Database

Nameasync function main() {

  await client.connect();
  console.log('Connected successfully to server');

  const db = client.db(dbName);
  const collection = db.collection('collection_name');

//Can Add the CRUD operations
   }

main() .then(console.log)
          .catch(console.error)
          .finally(() => client.close());
```

instance of MongoDBClient.

- **MongoClient** class provided method, to connect MongoDB and Nodejs.
- **client** is the instance of MongoDb and Node Js connection.
- **client.connect()** is used to connect to MongoDB database ,it **awaits** until the the connection isestablished.

Create a Collection in MongoDb using Node Js

In this operation we create a collection inside a database.Intilally we specify the database in whichcollections is to be created.

```
//Sepcify Database
const dbName =

'database_name';const db =

client.db(dbName);
```

- **client** is the instance of the connection which provides the **db**() method to create a new Database.
- **collection**() method is used to set the instance of the collection .

Insert a Single Document

To insert a document into the collection **insertOne**() method is used.

```
const insertDoc = await
   collection.insertOne({filed1: value1,
   field2: value2,
);
```

```
//Insert into collection
console.log('Inserted documents =>', insertDoc);
```

Insert Many Document

To insert a document into the collection insertMany() method is used.

```
const doc_array = [
   { document1 },
   { document2 },
   { document3 },
];

//Insert into
collectionconst
insertDoc =
   await collection.insertMany(doc_array);
```

Handling Insertion Results

In a project we have different tasks which needs to be executed in specific order.In the MongoDB and Node Js project we must ensure that connection is set.While performing insertion of documents
, we perform asynchronous insertion so that execution is not interrupted.We use try-catch block to handle errors while setting up connection, inserting document or while performing any other operation. If an error occurs during execution ,catch block handles it or provide the details about theerror ,which helps to resolve the error.

```
try {
   const dbName =
   'database_name';await
   client.connect();
   const collection = db.collection('collection_name');

    const doc_array
             = [
     { document1 },
     { document2 },
     { document3 },
   ];

//Insert into collection
const insertDoc = await collection.insertMany(doc_array);

   console.log('Inserted documents =>', insertDoc);
} catch (error) {
```

- Initally connection is established .AS the connection is established insertMany() method or insertOne() method is used to insert the document in the collection.
  - **insertDoc** stores the result of the insertion which is further logged.

Read Documents from the collection

We can read the documents inside the collection using the find() method.

```
const doc = await
collection.find({}).toArray();
```

find() method is used to along with empty {} are used to read all the documents in the collection.Which are further converted into the array using the toArray() method.

Closing the Connection

```
finally{
client.close
()
```

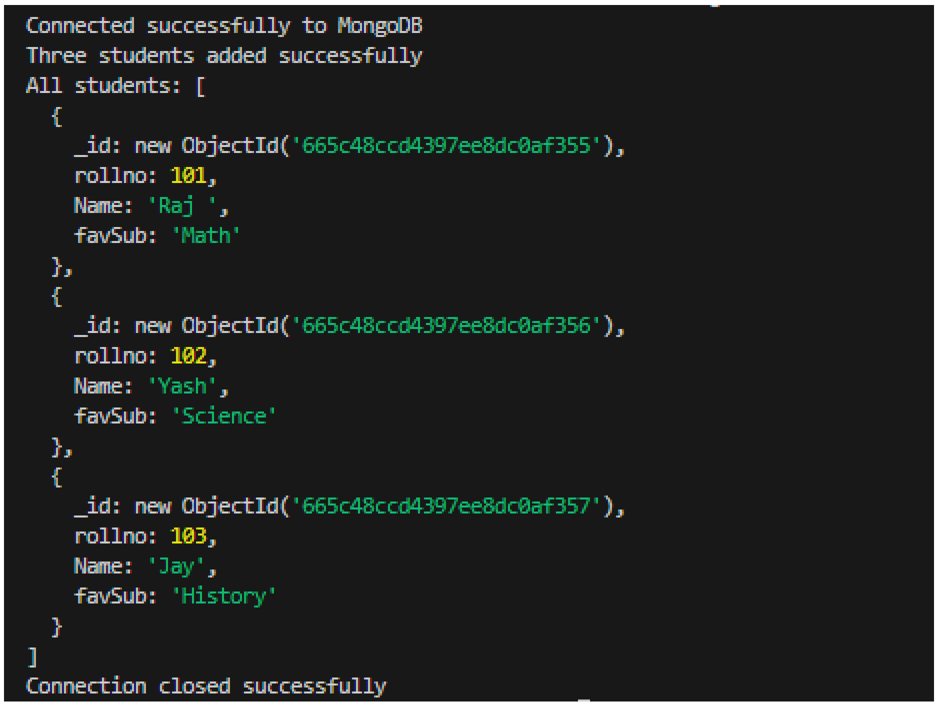
- Once the promise is resolved or rejected , code in finally block is executed. The **close()** methodis used to close the connection.
- Connection is closed irrespective of the error .It is generally used to cleanup and release theresource.

Example: Implementation to show Insertion of documents into a MongoDB collectionusing Node.js

JavaScript

```
const { MongoClient } = require("mongodb");

async function main() {
  const url = "mongodb://127.0.0.1:27017";
  const dbName = "GeeksforGeeks";
  const studentsData = [
    { rollno: 101, Name: "Raj ", favSub: "Math" },
    { rollno: 102, Name: "Yash", favSub: "Science" },
    { rollno: 103, Name: "Jay", favSub: "History" },
  ];

  let client = null;

  try {
    // Connect to MongoDB
    client = await MongoClient.connect(url);
    console.log("Connected successfully to
    MongoDB");

    const db = client.db(dbName);
    const collection = db.collection("students");

    // Add students to the database
    await collection.insertMany(studentsData);
    console.log("Three students added
    successfully");

    // Query all students from the database
    const students = await
    collection.find().toArray();console.log("All
    students:", students);
  } catch (err) {
    console.error("Error:",
    err);
  } finally {
    // Close the connection
    if (client) {
      client.close(
      );
      console.log("Connection closed successfully");
    }
  }
}

main();
```

Output:



```
Connected successfully to MongoDB
Three students added successfully
All students: [
  {
    _id: new ObjectId('665c48ccd4397ee8dc0af355'),
    rollno: 101,
    Name: 'Raj ',
    favSub: 'Math'
  },
  {
    _id: new ObjectId('665c48ccd4397ee8dc0af356'),
    rollno: 102,
    Name: 'Yash',
    favSub: 'Science'
  },
  {
    _id: new ObjectId('665c48ccd4397ee8dc0af357'),
    rollno: 103,
    Name: 'Jay',
    favSub: 'History'
  }
]
Connection closed successfully
```

Insert Document in MongoDB

Explanation :

In the above example, Initially **MongoClient** class is imported which is used to connect MongoDB and Nodejs .**client** is the instance of MongoDb and Node Js connection. which is used to name the database .As database is set ,**collection()** method sets the instance of the collection .Three documentsare inserted in the **students** collection using **insertMany**() method .Error during the execution are handled using the try catch block ,finally connection is closed using the **close() method**