Chapter 1 : Introduction to Web X.0 1-1 to 1-25

Syllabus : Evolution of Web X.0; Web Analytics 2.0 : Introduction to Web Analytics, Web Analytics 2.0, Clickstream Analysis, Strategy to choose your web analytics tool, Measuring the success of a website; Web 3.0 and Semantic Web : Characteristics of Semantic Web, Components of Semantic Web, Semantic Web Stack, N-Triples and Turtle, Ontology, RDF and SPARQL

Self-learning Topics : Semantic Web Vs AI, SPARQL Vs SQL

Chapter 2 : TypeScript

Syllabus : Overview, TypeScript Internal Architecture, TypeScript Environment Setup, TypeScript Types, variables and operators, Decision Making and loops, TypeScript Functions, TypeScript Classes and Objects, TypeScript Modules Self-learning Topics : Javascript Vs TypeScript

Chapter 3 : Introduction to AngularJS 3-1 to 3-46

Syllabus : Overview of AngularJS, Need of AngularJS in real web sites, AngularJS modules, AngularJS built-in directives, AngularJS custom directives, AngularJS expressions, Angular JS Data Binding, AngularJS filters, AngularJS controllers, AngularJS scope, AngularJS dependency injection, Angular JS Services, Form Validation, Routing using ng-Route, ng-Repeat, ng-style, ng-view, Built-in Helper Functions, Using Angular JS with Typescript

Self-learning Topics : MVC model, DOM model, Javascript functions and Error Handling

Chapter 4 : MongoDB and Building REST API using MongoDB 4-1 to 4-31

Syllabus : MongoDB: Understanding MongoDB, MongoDB Data Types, Administering User Accounts, Configuring Access Control, Adding the MongoDB Driver to Node.js, Connecting to MongoDB from Node.js, Accessing and Manipulating Databases, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js, Using Mongoose for Structured Schema and Validation.

REST API: Examining the rules of REST APIs, Evaluating API patterns, Handling typical CRUD functions (create, read, update, delete), Using Express and Mongoose to interact with MongoDB, Testing API endpoints

Self-learning Topics : MongoDB vs SQL DB

Chapter 5 : Flask 5-1 to 5-21

Syllabus : Introduction, Flask Environment Setup, App Routing, URL Building, Flask HTTP Methods, Flask Request Object, Flask cookies, File Uploading in Flask

Self-learning Topics : Flask Vs Django

Chapter 6 : Rich Internet Application 6-1 to 6-34

Syllabus : AJAX : Introduction and Working, Developing RIA using AJAX Techniques : CSS, HTML, DOM, XML HTTP Request, JavaScript, PHP, AJAX as REST Client Introduction to Open Source Frameworks and CMS for RIA: Django, Drupal, Joomla

Self-learning Topics : Applications of AJAX in Blogs, Wikis and RSS Feeds

❑❑❑

1 Introduction to Web X.0

Syllabus

Evolution of Web X.0; Web Analytics 2.0 : Introduction to Web Analytics, Web Analytics 2.0, Clickstream Analysis, Strategy to choose your web analytics tool, Measuring the success of a website; Web 3.0 and Semantic Web : Characteristics of Semantic Web, Components of Semantic Web, Semantic Web Stack, N-Triples and Turtle, Ontology, RDF and SPARQL

Self-learning Topics : Semantic Web Vs AI, SPARQL Vs SQL

1.1 What is Web X.0?

Web X.0 is the generic terminology that refers to the Xth phase in the evolution of the World Wide Web.

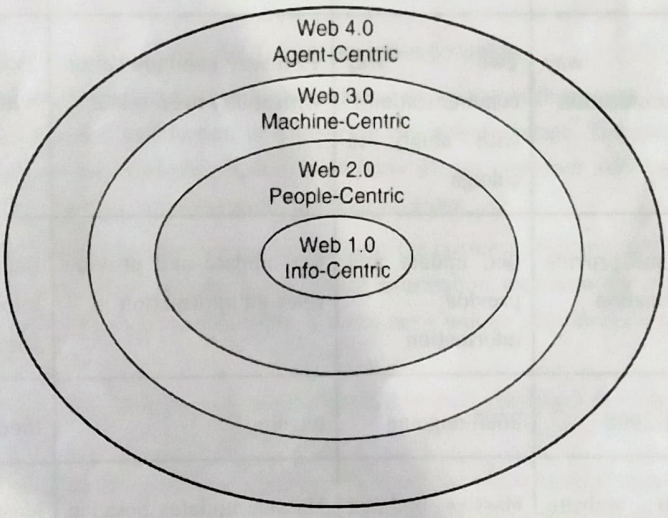


Fig.1.1.1

- In the very nascent or beginning stages of the WWW, or as we will now call it, **Web 1.0**, is all about publishing information as a one-way medium. The objective was to share information with the world as widely and as fast as possible. This stage set the foundation for the web as we know it today. It is in that phase that the HTTP and Web Browsers were born.
- **Web 2.0** was the paradigm shift from publishing to participating. Users were encouraged to not just put information out there, but communicate with others and let information "evolve". This was the birth of social media and community resource-sharing. Basic social media like MySpace and Wikipedia were born.

- **Web 3.0** was the decentralized and intelligent version of the Web, widely known as Semantic Web. People were encouraged to share and build in a way that machines could communicate and aggregate information. This was the first time that users could communicate with the outside world without authenticating themselves via a trusted third-party. With little to no verification and open access, cryptocurrencies and blockchain were born. Humans and computers now worked in tandem to create a semantic form of the World Wide Web.
- **Web 4.0** or symbiotic web is where the humans and the machines feed off of each other, learn from each other and develop the combined knowledge base to take it to new heights. This version supports the adaptive presentation of data and advanced search engine optimization that tailors each individual's internet experience to their needs. Though formally not defined, many refer to it as "active web", where the machine understands human speech well enough to provide meaningful responses. We are today formally progressing toward this iteration of the World Wide Web.
- **Web 5.0**, a theoretically sensory and emotive Web, will communicate with you not just on a logical level, but also on an emotional one. It will map your intent and your actions, continue to learn, grow and then respond effectively. This version of the Web will detect subtleties in speech patterns and your voice. Think SkyNet!

Table 1.1.1 : Compare and Contrast Web X.0

Web 1.0	Web 2.0	Web 3.0	Web 4.0	Web 5.0
Old and discarded - almost out of use	Old, but still in use where net bandwidth is low	Current iteration with maximum users	Moving from theoretical to real	Still theoretical
One way communication	Two way communication	Two way communication with ability to change	Two way communication with ability to co-relate	Two way communication with ability to understand
Provide information	Get and provide information	Get, update and provide information	Get, update and provide relevant information	Get, update, provide relevant information by inferring who the user is
1993-2001	2001-2008	2008-ongoing	ongoing	theoretical
No website updates	Minor website updates from end user	Massive updates possible from end user	Massive updates possible from both end user and data from similar users on the world wide web	Massive updates will be possible by every linked entity on the world wide web
Examples are news websites	Examples are online calculators	Examples are Facebook and LinkedIn	Examples are Siri and Alexa	Examples are Skynet from Terminator movies and Allie A.I. from The 100

1.2 Web Analytics 2.0

1.2.1 Introduction to Web Analytics

Web Analytics is the understanding of a consumer's behavior when they visit a website. This involves everything from tracking their keystrokes and IP addresses, reviewing web history, monitoring cookies and reporting all aggregated information to understand the website visitor's intent.

Efficient use of web analytics allows online businesses to attract more eyeballs, convert more customers and increase a customer's spend. The objective is to promote to the visitor exactly what they are most likely to buy based on their purchase history, site visitation history, geographic region, race, etc.

The basic steps involved in a web analytics process are as shown in the Fig. 1.2.1.

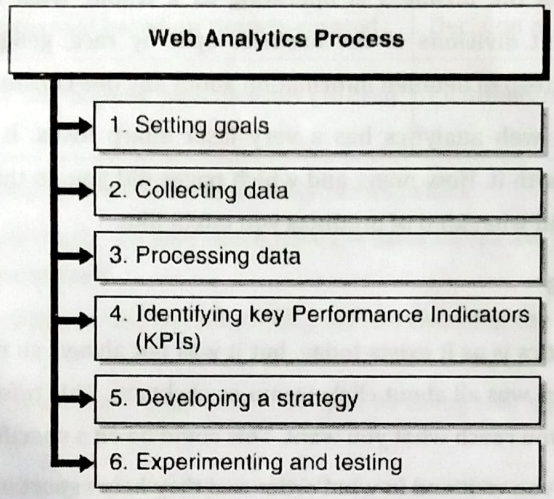


Fig. 1.2.1: Web analytics process

- **Setting Goals :** The business decides the high-level goal or what it is they want to achieve. Examples could be increasing sales & customer satisfaction or enhancing the brand's image. The goal could be quantitative like decreasing the value of per marketing dollar spent from $1 per customer acquired to $0.8 or subjective like enhancing the business' rating on the customer satisfaction index.
- **Collecting Data :** Depending on the use case for the data, the business decides and collects data according to their needs. HTTP header will get the business geographical information, web crawling and keystroke logging will give the business your path followed on a website, a customer's unique identification number will give them their purchase history and so on.
- **Processing Data :** This step involves aggregating the information, cleaning it to make it usable and turning it into actionable results.
- **Identifying Key Performance Indicators (KPIs) :** KPIs tell the business how useful the actionable insights and results are and enable continuous monitoring of these indicators to understand short and long term patterns in customer behavior.
- **Developing a Strategy :** The business decides what the next step once they have understood what the customer wants and what insights they have received from a large number of customers. This step can dictate any changes necessary to the business model to get and retain more eyeballs.
- **Experimenting and Testing** : The business in this step tests out different tactics to see which one works best. This strategy is called A/B Testing. It tests how different customers respond to different content under different conditions. Here, a customer is shown multiple kinds of pre-chosen strategies and the business tracks the customer's response to each of them to determine what worked best.

Primary Types of Web Analytics

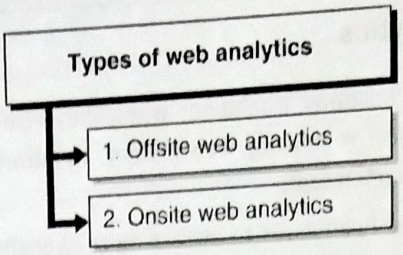


Fig. 1.2.2 : Types of Web Analytics

- **Offsite Web Analytics:** In this solution, the business aggregates all information outside of the organization's website to provide an analysis of how well the business is operating as a whole, what customer segments look like when combined, the health of different divisions of the business split by race, geography, etc. Offsite analytics usually pertains to high level insights instead of detailed information about any one consumer.
- **Onsite Web Analytics:** On-site web analytics has a very laser sharp focus. It tracks visitors to the website and understands how they interact with it. How many and which pages did you go through before you bought something, why did you leave the website and what kind of products you might like.

1.2.2 Clickstream Analysis

The above explanation of Web Analytics is as it exists today, but it was not always so rosy. After the birth of the WWW, for almost a decade, the old web analytics was all about clickstream or clickpath. This refers to the series of hyperlinks or web pages you visit in a sequence before you reach what you want. This could be on a specific website or all over the web.

Log files are used to store what sites you visit and in what order and they help report user behavior, where they are coming from (advertisement on Instagram perhaps) and where they are going after they leave you (your competitor for a cheaper price maybe). As an example, if you leave Amazon's webpage to immediately go check Flipkart for a better rate, Amazon will want to know that and vice-versa.

The basic benefits of clickstream analysis are:

Benefits of click stream include :

- Identify latest consumer trends
- Discover new mediums to do business
- Enhance conversion rates from visitor to buyer
- Analyze user behavior for what works and what doesn't

The drawbacks of clickstream analysis are as follows:

- Log files store IP Addresses and not names and can be quite unreliable at determining correct user sessions, so analytics can often be mixed up with different users.
- Individualization is expensive and if even one wrong recommendation is made to a customer, they might deviate to something entirely different from what they want on the website, causing them to leave and causing even more losses than if no individualization was used.

1.2.3 Web Analytics 2.0

Let us start directly with the major differences between Web Analytics 1.0 and Web Analytics 2.0.

Table 1.2.1 : Differences between Web Analytics 1.0 and Web Analytics 2.0

Sr. No.	Web Analytics 1.0	Web Analytics 2.0
1	Clickstream rules	Also cares about experimentation, competitor analysis and the needs of the customer
2	Only quantitative	Quantitative and qualitative
3	Scope of analysis is only your business	Scope of analysis is you and your competitors
4	Decision making is manual based on reports created	Decision making is more automated
5	Business needs of the highest paid person (HiPPo) are important	Customer needs are important

Clickstream determines the what - what data we want to actually see and store.

Multiple Outcomes Analysis is the how much - by how much have the sales increased, how much has the cost decreased and how much has the customer loyalty increased.

Experimentation and testing & the customer are the why - why are we choosing the tactic we have chosen to achieve the desired outcome with the customer.

Competitive intelligence determines what else - what else could you be doing that your competitors are doing to get more business and keep what you have.

Insights are the gold - the gold about what your customers want that you are mining for!

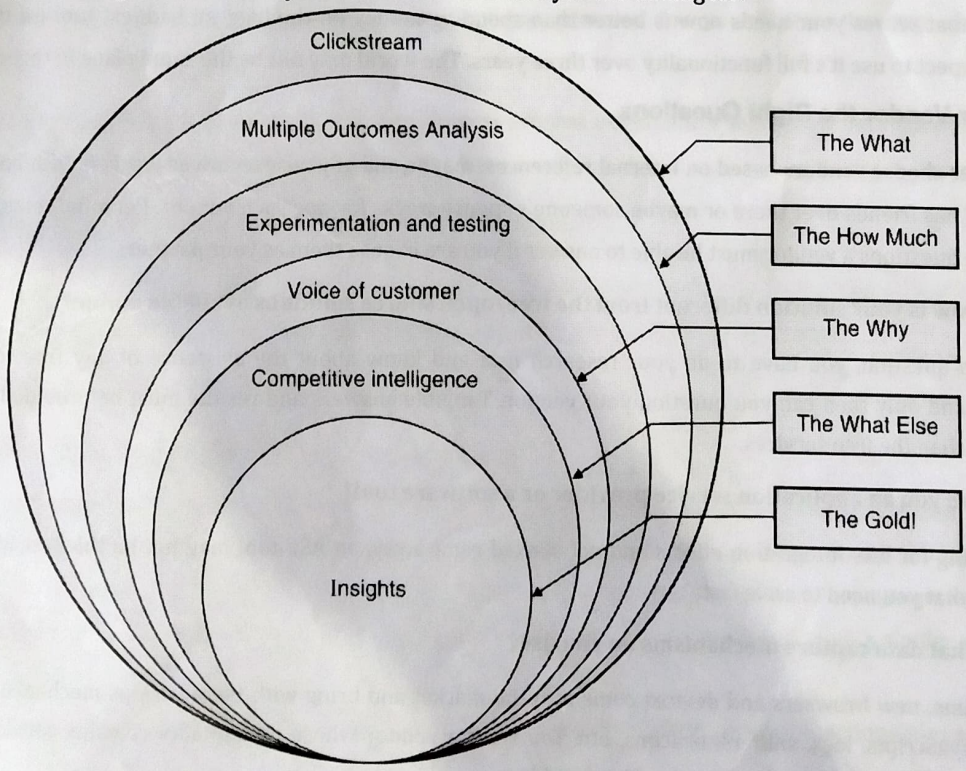


Fig.1.2.3

1.2.4 Choosing the Right Web Analytics Tool

The biggest mistake in choosing the right web analytics tool or any tool for that matter, is jumping to the first tool you come across. We hardly ever consider the qualities that determine what tool is right for us.

(A) Ask Yourself the Right Questions

The following questions are a good start for anyone looking to choose a tool :

Question 1 : Do I want reporting or an analysis?

Companies don't want to admit it, but most of them do not have the risk propensity or the fast-moving company culture to need analytics. But it is hard to admit that they only need reporting for public image and they end up with analytics tools where none is needed.

Question 2 : Do I have IT strength, Business strength, or both?

Very few companies have a solid foundation with both IT and management. The first step is accepting what's missing and if so, how do you augment that issue? Hire an external consultant? Hire inhouse? Or choose a tool that supports what you are missing in your organization with their own expertise? By accepting what may be missing, you are already a step ahead of your competitors.

Question 3 : Am I solving just for Clickstream or for Web Analytics 2.0?

The difference between these is in the mindset. Is the organization looking just to see a pattern of clicks or something more? Who makes the decision? Some man in a suit or is the customer's needs that dictate the path forward? Understanding this is core to determining the short and long term goals.

Buying a tool that serves your needs now is better than spending money on the biggest, baddest tool on the market just because you expect to use it's full functionality over three years. The world may not be the same place in three years.

(B) Ask Your Vendor the Right Questions

Most companies choose vendors based on internal references: maybe one of your executives worked with one such vendor in the past and has friends over there or maybe someone's spouse works for one such vendor. Personal relationships aside, there are some questions a vendor must be able to answer if you are choose them as your partner:

Question 1 : How is your solution different from the free/open source solutions available online?

To answer this question, you have to do your research first and know about the existence of any free tools and their functionalities and only then can you question your vendor. Tangible answers and results must be provided to show how they are better than the free services.

Question 2 : Are you an application service provider or a software tool?

If you are looking for less integration efforts and get started right away, an ASP tool may not be the best idea. It may be cheap but not what you need to scale fast.

Question 3 : What data capture mechanisms do you use?

From time to time, new browsers and devices come into the market and bring with them various mechanisms to capture data online - JavaScripts, logs, sniffers, beacons, etc. You want a vendor whose data quality remains consistent and can capture as many modes of information retrieval as possible.

Question 4 : What is the Total Cost of Ownership (TCO) for your tool?

This is a lot more than just the annual subscription fee. It includes the licensing, integration, support, any additional hardware you may need, licensing, additional hiring and training and even the risk you take by hiring the vendor. If things go bad, will your organization brush it off or fire you for your poor judgement? There is a cost associated even with the free tools, so ask all you need from a vendor to calculate this effectively.

Question 5 : What is the timeline for your support?

Your organization loses business if the vendor is offline or fails at their job. How good their backup and support services are is important to make an informed decision.

Question 6 : How can I segment the data you give me?

You may need insights not just on all customers, but on specific segments. Why are my sales down in Gujrat but not in Mumbai. Why are the millennials buying my products but not Gen Z. Why are most visitors from Kerala signing in between 8-10 PM only? Can the vendor give you detailed enough segments to better understand your consumer base?

Question 7 : How can I export data from your system into my company's system?

You may want this data for reporting, aggregation, regulations or any other reason. Knowing this will help better plan for what you buy.

Question 8 : Can I integrate other external data and tools into your service?

A tool should be nimble enough to integrate all top data sources and tools in the market instead of making it unnecessarily difficult to do so. This shows the vendor's insecurity in their own service.

Question 9 : What are the new features your company has planned for your future to stay ahead of your competition?

If a company is not agile enough to make quick changes, chances are that a competitor will beat them right from under their noses and your business may find better value somewhere else.

Question 10 : Have you lost any contracts to your competitors? Why?

You will probably get twisted answers but they can also help you understand the vendor. In the long term, the relationship with good people will be more important than the technologically best tool available in the market.

(C) Comparing Different Vendors

You have now shortlisted a few vendors. But which one to choose?

Three Bucket Strategy : Categorizing vendors for their diverse offerings is important. Let us assume that A, B and C are similar vendors and X, Y and Z are similar vendors. If you chose A initially, don't like it and then switch to vendor B, which is very similar to A, you've basically fallen into a trap. Comparing vendors and bucketing them into categories ensures that you not capture how different and similar vendors are but also saves you from getting stuck with a similar vendor in case the original choice does not work out.

If you have the budget to, choosing a tool from each bucket will give you the highest range of functionalities and ensure you make a truly diverse and differentiated choice.

1.3 Measuring the Success of your Website

1.3.1 Top KPIs

Measuring how successful a website is depends on asking yourself how important the website is and what purpose it serves or what need it satisfies. We have Key Performance Indicators (KPIs) to measure what that looks like. Let us look at a few possible KPIs that companies use to track success for a website:

1. **Task Completion Rate**

   How many visitors who visited the website actually completed what they came to the website for? Some companies quantify this using on-exit surveys, some track the pages customers go through and guess their purpose. Spending money is not the only intended outcome. This metric tells us how successfully customers can get to what is important to them and help you set your priorities.

2. **Share of Search**

   How many customers visit your website as compared to your competitors? How fast is your share of that pie growing when compared to others? Is it fast enough? This metric tells us how well the marketing dollars are spent to attract eyeballs and determines the health of your website compared to your competitors in the same space.

3. **Visitor Loyalty and Recency**

   Visitor loyalty tells us how many times a certain customer visits your website. Are most of your customers just fleeting? They come and go like the wind? Or do they stick with you? Recency dictates the interval between their visits. They may visit your site multiple times every day or once a month. There is a fine line here. You don't want only one visitor visiting the site all the time. You want your metric to show a large customer base and a large chunk of them revisit at frequent intervals.

4. **RSS/Feed Subscribers**

   Really Simple Syndication or RSS how many people are staying tuned to your site's blogs or releases as subscribers. This is important because the subscriber traffic is outside your website and is not tracked by web analytics tools. This is the behavior shown by your committed website audience who look for your content rather than you having to push it out to them.

5. **Percentage of Valuable Exits**

   This is the percentage of people who left the website after doing something that benefits you. They bought something, viewed your blogs, donated money, etc. This metric is often a key monetary indicator. This metric can help segment the most valuable customer base to determine what age group they are, what part of the web they come from, how hey found you, where they live and help you determine how to increase your customer base and who all to most focus on.

6. **Abandonment**

   How many visitors abandoned a cart after adding items to it and how many customers abandoned a purchase after reaching checkout? This metric often tells you about a problematic part of the customer experience. Is the shipping cost too high? Are there other associated charges involved? Are you not covering their postal codes for shipping? Are you not accepting their mode of payment?

1.3.2 Macro and Micro Website Conversions

A macro conversion is the key customer conversion on your website. For instance, a successful sale on an e-commerce website or a completed customer sigh-up form. A micro conversion talks about the small interactions a customer has like the blog sign-up or a potential customer watching a newly featured video. Micro conversions often foretell and create desire and precede the macro conversion.

Table 1.3.1 shows differences between Macro and Micro Conversion.

Table 1.3.1 : Differences between Macro and Micro Conversion

Sr. No.	Macro Conversion	Micro Conversion
1	Revenue-based	Site Navigation-based
2	Goal is the primary objective of the website	Goal is to guide customers to macro conversions
3	**Examples :** Creating an account, buying an item, clicking a sponsored link	**Examples :** watching a demo, signing up for the newsletter, adding to the cart

1.4 Web 3.0 and Semantic Web

1.4.1 Components of Semantic Web

The Semantic Web or Web 3.0 is based on several component standards and tools as explained below:

- **XML :** XML gives semantic web a basic format for the use of structured documents and has no particular semantics. It has many parts:
  - XML Namespaces distinguish between multiple vocabularies.
  - XML Schemas restrain the XML document structure.
  - XML Parsers code and decode XML.
  - XML Query goes through the database for requested information.
- **XML Schema :** XML Schema is a language that gives the semantic web its data typing and allows the XML-created document structure to be restricted enough to allow predictable computation. No typing could mean lack of typing and unpredictable document reading capabilities.
- **Resource Description Framework (RDF) :** RDF is an easy three-part data model (subject, predicate, object) that is used to refer to objects or resources within the semantic web and how these objects or resources relate to each other. An RDF model is usually shown with XML.
- **Uniform (or Universal) Resource Identifiers (URI) :** URIs are short strings created using ASCII characters that identify all that is available on the Web. This could be documents, videos, audio, images, downloadable files, unknown file types, viruses, services, electronic mailboxes and everything else. URIs (also called uniform resource locators, or URLs) give us access to address resources in a simplified and easy-to-understand way and with just one click, they can help download information, log into servers and issue different kinds of commands over the web.
- URI contains the URL (Uniform Resource Locator) and the URN (Uniform Resource Name). If the web is your memory palace, URIs are the tools and manipulations we use to find our way around that palace.

- **Resource Description Framework Schema (RDF-S)** : RDF-S describes the properties, the classes within and the hierarchies within. It stores extensive knowledge of the data model. It allows information to be identified, collated and interconnected by different systems. This is similar to a SuperClass in Object Oriented Programming where it describes the relationships between the objects it contains.
- **Web Ontology Language (OWL)** : OWL is basically the data dictionary. It stores the meaning of all terminology and the relationships between them. The terms and their relationships are called an ontology. It is a semantic web language that was designed to store rich and complex language. It is a computational logic-based old-school language that is exploited by other computer programs all over the web.

1.4.2 Semantic Web Stack

The Semantic Web Stack describes the architecture of the Semantic Web (Web 3.0).

The Semantic Web is a combined movement spearheaded by an international standards body widely known as the World Wide Web Consortium (W3C).

W3C promotes a common data format all over the World Wide Web for ease of use and access. By pushing for the inclusion of semantic content in web pages all over the world and targeting for massive adoption, the Semantic Web can convert the current web as we know it, which is dominated by unstructured and semi-structured documents into a well designed web of structured data.

The Semantic Web stack was the first step that was built on W3C's Resource Description Framework (RDF).

Fig. 1.4.1 shows the Semantic Web Stack, the hierarchy of languages as they exist today (and is constantly changing), where each layer in the stack makes use of the functionalities of the layers under it.

The stack shows how various technologies all over the world are standardized for Semantic Web and are organized to make the Semantic Web as we want it to be possible. It also portrays how Web 3.0 is basically an extension of and not a replacement of the classical hypertext web from the 90's.

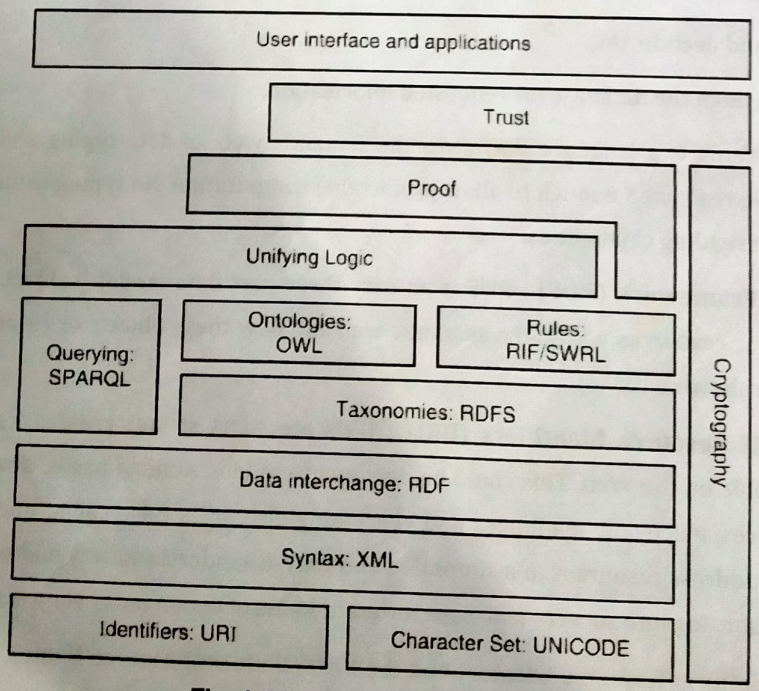


Fig. 1.4.1 : Semantic Web Stack

Hypertext Web Technologies

The bottom layers of the semantic web stack contain old technologies of the times past that are well renowned from hypertext web and provide a base upon which the semantic web was built.

Internationalized Resource Identifier (IRI), a general way of writing URI, gives us a way to uniquely identify resources and information all over the semantic web. Semantic Web requires unique identification to allow access and manipulation of resources within the newer top meant for Semantic Web.

UNICODE is meant to represent and allow manipulation of text in multiple languages to facilitate wider information access. Semantic Web can also help act as a translator between documents in multiple human languages.

XML is the web's markup language that allows creation of documents made up of semi-structured information. Semantic web gives meaning or "semantics" to use a better word to semi-structured information.

XML Namespaces are a way to use markups from multiple sources. Semantic Web is all about stitching data together, and so this ensures that more than one source can be used in a document.

Standardized Semantic Web Technologies

Middle layers of the semantic web stack contain technologies that have been standardized by the World Wide Web Consortium to enable the creation of semantic web applications by end users.

- **Resource Description Framework (RDF)** is meant to create statements in the form of so-called triples. It allows information about resources to be stored in the form of a graph - and this is why the semantic web is often referred to as the Giant Global Graph.
- **RDF Schema (RDF-S)** provides the very basic vocabulary for the RDF as explained in the semantic web components. Using RDF-S, it is possible to create a large hierarchy of classes and properties that describe them.
- **Web Ontology Language (OWL)** extends the RDFS by adding additional constructs that can describe the semantics of RDF statements. It allows creating and manipulating additional constraints like cardinality (just like databases), placing any number of restrictions of values, or characterizing properties like transitivity. It is based on description logic and can thus add the power to reason with the semantic web.
- **SPARQL** is an RDF querying language and it is used to query an RDF-based data store. Querying languages are required to retrieve any level of intelligence for semantic web applications.
- **RIF** is a rule interchange format. It is important, for example, to allow describing relations that cannot be directly described using description logic used in OWL.

Unrealized Semantic Web Technologies

Top layers of the semantic web stack contain technologies that are yet to be standardized by the W3C or are ideas and possible technologies that should be implemented in order to realize the dream of the Semantic Web as we want it.

- **Cryptography** is essential to ensuring and verifying that the semantic web statements are coming from a trusted source and haven't been spoofed or manipulated while transferring data in any way. This can be achieved by using appropriate digital signatures (mostly public-private key pairs) of RDF statements.
- **Trust** is meant to indicate that derived statements are coming from a trusted source and that derived statements are following all necessary rules while being converted from new information.
- **User Interface** is the final user-facing layer that enables us to use semantic web applications. It is the only layer we interact with.

1.4.3 Difference between Semantic Web and Artificial Intelligence

Table 1.4.1 : Difference between Semantic Web and Artificial Intelligence

Sr. No.	Semantic Web	Artificial Intelligence
1	Semantic Web is as much about the data (graph models with their entities and relationships) as it is about the logic (rules and inferences)	Artificial Intelligence is a field of study and it focuses on the logic used to achieve an outcome and is not concerned with the data.
2	End goal of Semantic Web is to exhibit intelligent behavior (a very modest goal).	End goal of artificial intelligence is to exhibit human-level intelligence (a very lofty goal).
3	Semantic Web twists and updates the knowledge (data) to increase the size of the known world to achieve the end goal.	Artificial Intelligence twists the model and the computational methodology to achieve the end goal.
4	Semantic Web is the infrastructure for Artificial Intelligence which is used to convert documents into machine readable documents just like Natural Language Processing converts language into machine readable language. Its output is the input for A.I.	Artificial Intelligence is one field of study that makes use of the output from Semantic Web, i.e. machine readable documents.
5	Semantic Web's inferences use straightforward rules and thus lack the complexity of traditional Artificial Intelligence.	Artificial Intelligence makes use of a complicated web of modelling paradigms and feedback mechanisms that is lacking in Semantic Web.

1.5 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a highly flexible technology which can address a wide variety of problems. The RDF is a language created to support the workings of the Semantic Web, in the same way that HTML has helped jumpstart the original Web.

The RDF is a framework for supporting resource descriptions within the Semantic Web, or metadata (data descriptions about data), for the Web. RDF gives us common structures that are used for interoperable XML data exchanges throughout the Web.

RDF is just a standard model for data exchange on the Web. It has many features that allow for data merging even if the underlying schemas and architectures are different, and it even supports the evolution of such schemas.

RDF expands the interlinked structure of the Web to use Uniform Resource Identifiers to identify the relationship between things and the two ends of this linked relationship (this is usually referred to as a "triple"). Using this model, RDF facilitates both structured and semi-structured data to be mingled and shared all over the Web through different applications.

This interlinked data structure forms a directed and labeled graph where the edges of the graph represent the name of the link between two endpoints. These are the graph nodes. This graph is the easiest possible model to remember for RDF and is an easy to understand visual explanation.

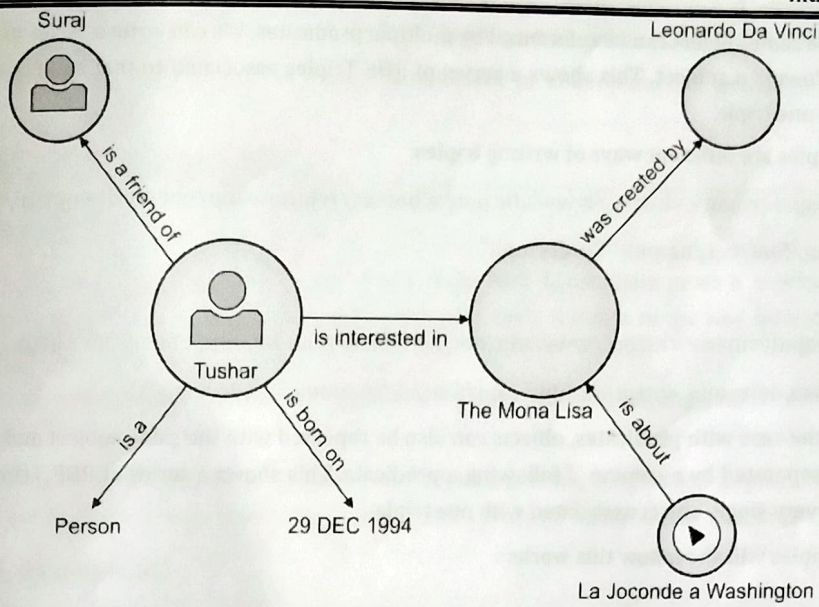


Fig. 1.5.1

1.6 N - Triples and Turtle

1.6.1 Turtle

Turtle is the Terse RDF Triple Language - a textual representation of an RDF graph. The below Turtle document will detail the relationship between Venom and Spiderman from the Marvel Multiverse:

```
@base <http://example.org/> .
@prefix rdf: <http://www.abc.org/2021/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.abc.org/2021/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.abc.net/schemas/relationship/> .
<#venom>
  rel:enemyOf <#spiderman> ;
  a foaf:Human ;   # in the Marvel multiverse
  foaf:name "Venom" .
<#spiderman>
  rel:enemyOf <#venom> ;
  a foaf:Human ;
  foaf:name "Spiderman", "author"@in .
```

P.S. "foaf" is the machine-readable ontology, which is short for "friend of a friend" and is used to describe persons.

An RDF-graph consists of triples containing a subject, a predicate and an object.

Simple Triples: The simplest triple statement is just a sequence written as (subject, predicate, object), separated by a whitespace and ending with a'.' after each triple.

```
<http://abc.org/#spiderman>  <http://www.abc.net/schemas/relationship/enemyOf><http://abc.org/#venom> .
```

Predicate Lists: The same subject can be referenced by multiple predicates. We can write a series of predicates and objects, separated by ';', following a subject. This shows a series of RDF Triples associated to that subject and each predicate and object associated to one triple.

The following examples are different ways of writing triples:

<http://abc.org/#spiderman> <http://www.abc.net/schemas/relationship/enemyOf> <http://abc.org/#venom> ;

<http://xmlns.com/foaf/0.1/name> "Spiderman" .

Is the same as:

<http://abc.org/#spiderman> <http://www.abc.net/schemas/relationship/enemyOf> <http://abc.org/#venom> .

<http://abc.org/#spiderman> <http://xmlns.com/foaf/0.1/name> "Spiderman" .

Object Lists: As is the case with predicates, objects can also be repeated with the same subject and predicate. We can write a series of objects separated by a comma ',' following a predicate. This shows a series of RDF Triples with the subject and the predicate and every single object associated with one triple.

The following examples will show how this works :

<http://abc.org/#spiderman> <http://xmlns.com/foaf/0.1/name> "Spiderman", "author"@in .

Is the same as:

<http://abc.org/#spiderman> <http://xmlns.com/foaf/0.1/name> "Spiderman" .

<http://abc.org/#spiderman> <http://xmlns.com/foaf/0.1/name> "author"@in .

1.6.2 N-Triple

N-Triple is a format used for storing and transmitting data. It uses plain text serialization to denote RDF graphs (relationships).

The simplest N-triple is a sequence of 3 terms: a subject, a predicate and an object.

Each is separated by a whitespace and is terminated by a period: '.' after the triple.

<http://example.org/#tushar> <http://www.example.net/relationship/friendOf> <http://example.org/#suraj> .

This format breaks down an RDF graph into its component triples, with one displayed as one line. N-Triples that have been created from the same RDF graph will always produce the same result and thus look the same, making it a very effective method to verify and validate the processing of an RDF/XML document.

The basic structure of an N triple is:

subject predicate object .

A space or a tab separates the three component elements of an N triple from each other. A space or a tab will also precede the elements.

In addition, each N-triple must be ended with a period / full-stop (.) followed by a new line.

An N-Triples file can also make use of comments to explain intentions in the following format:

comment

Each line contained in an N-Triple file can have either a triple or a comment, but cannot have both.

The subject of an N-triple can contain either a uriref or a blank node identifier The latter is a value generated for blank nodes within N-Triples syntax, as there can be no blank subjects within legal N-Triples-formatted output. The blank node identifier format is:

_:name

where name is a string. The predicate is always a uriref, and the object can be a uriref, a blank node, or a literal.

A basic and rudimentary understanding of the N-Triples is sufficient to understand the next topic, which is of utmost importance: SPARQL.

1.7 Web Ontology Language (OWL)

1.7.1 OWL Definition

Web Ontology Language (OWL) is the backbone of the World Wide Web Consortium or as it is widely known, the W3C standard. It is a knowledge denotation language used in the Semantic Web. It refers to the way information is modelled to portray entities and their relationships.

OWL is extremely intuitive and expressive, truly very flexible and a very efficient language, used the whole world over in many domains from healthcare to social services.

OWL supports reasoning services, meaning that knowledge can be processed for insights and implicit information. A simple example of this is that if every king is a male and Tushar is a king, then it can be inferred that Tushar is a male.

1.7.2 Basic OWL Constructs

The primary element of the OWL is an International Resource Identifier. Each entity has an IRI associated with it.

Classes, their properties, associated individuals and data types are the building blocks of the Web Ontology Language.

Classes are conceptual entities. In the common tongue, think of them as common nouns. Author, publisher, paper, worker, etc. will all be Classes in OWL.

Instances of these Classes will be the Individuals. Tushar as an individual is the author of this book. Think of Tushar belonging to the CLass Author.

Properties are relationships and are of two types:

1. **OWLDatatypeProperty :** this denotes the relationship between an individual and a datatype. Eg: hasName, hasTitle, etc. "Tushar" with the title "Author" would be an example.
2. **OWLObjectProperty :** this denotes the relationship between individuals and individuals. hasAuthor would be an excellent example. An instance of the Book Class with the name "WebX.0" has an author who is an instance of the Class "Author" with the name "Tushar".

Let us see some basic constructs in OWL:

```
< owl : Class rdf : about = "Author"/ >                          (1)
< owl : Class rdf : about = "Book"/ >                            (2)
< owl : OWLObjectProperty rdf : about = "hasAuthor"/ >           (3)
< owl : OWLDatatypeProperty rdf : about = "hasName"/ >           (4)
< Author rdf : about = "Tushar"/ >                               (5)
< rdf : Description rdf : about = "Tushar" >
< rdf : type rdf : resource = "Author"/ >                        (6)
< /rdf : Description" >
```

Explanation

The first and second statements say that the "Author" and the "Book" are concepts that have been represented by OWL.

Statements 3 and 4 tell us that the "hasAuthor" is an OWLObjectProperty while "hasName" is an OWLDatatypeProperty.

Statements 5 and 6 are just alternate ways of saying that the individual "Tushar" is an instance of the Class "Author".

1.7.3 OWL Class Relations and Constructors

Taxonomy or classification is important for modelling. It helps model relationships between entities like Classes and subClasses and understand the properties that exist in the data.

Consider the following constructs:

- ***Every Author is a Contributor***. This tells us that "Author" is a subClass of "Contributor".
- ***Every Author is a Writer and vice-versa***. This tells us that "Author" and "Writer" are equivalent Classes.
- ***An Author cannot be a Subscriber and vice versa***. This tells us that "Author" and "Subscriber" are disjoint Classes.

OWL has some predefined Classes that help determine hierarchy.

- **< owl : Thing >** is always the topmost Class, meaning that all other Classes are subClasses of this Class. The complementary Class to this Class is < owl : Nothing > and it is the bottommost class, meaning that it is the subClass of all other Classes.

A few complex Class constructors have been explained below:

- **<owl : intersectionOf> (conjunction)** - This denotes a Class that is an intersection of two other classes, meaning members of this new Class are members of both Classes that this new Class is an intersection of.
- **<owl : unionOf> (disjunction)** - This denotes a Class that is a union of two other classes, meaning members of this new Class are members of either or both of the Classes that this new Class is a Union of.
- **<owl : complementOf> (negation)** - This denotes a Class in terms of a second Class such that any members of the second CLass are not members of this Class.

Even more complex class expressions can be created using property restrictions that impose restrictions on the properties of a Class. OWL has the following restrictions available for use:

<owl : someValuesFrom>

<owl : allValuesFrom>

<owl : minQualifiedCardinality>

<owl : maxQualifiedCardinality>

<owl : qualifiedCardinality>

The example below shows us how to use <owl : someValuesFrom> restriction.

It is the OWL way to write "every book must have at least one author". Other restrictions can be utilized with a similar syntax.

```
< owl : Class rdf : about = "Book" >
        < rdfs : subClassOf >
                < owl : Restriction >
                        < owl : onProperty rdf : resource = "hasAuthor" / > (7)
                        < owl : someValuesFrom rdf : resource = "Author" / >
                < /owl : Restriction >
        < rdfs : subClassOf >
< /owl : Class >
```

1.7.4 OWL Property Relations and their Characteristics

Properties are an essential component of the Web Ontology Language. It makes use of different terms to denote relationships between properties. Properties can also be related to each other via:

- <rdfs : subPropertyOf>,
- <rdfs : equivalentProperty> and
- <rdf : inverseOf>.

If a certain property_1 is a sub property of property_2, then if property_1 holds true, property_2 will also always hold true. If property_1 and property_@ are equivalent to each other, then both are sub properties of each other. If property_1 is the inverse of property_2, then if property_1 holds true for (x,y), property_2 will hold true for (y,x).

1.7.5 OWL Reasoning (Important)

The Web Ontology Language supports inherent reasoning as explained already. The OWL has a piece of software called the reasoner that makes these matches internally and supports multiple reasoning services as will now be explained:

1. **Ontology Satisfiability** : The reasoner determines if the input ontology is consistent and is free of any contradictions. For example, "Tushar" is an instance of Class "Author" and "Tushar" is also a negation of Class "Author". Both cannot exist simultaneously.
2. **Instance Checks :** Check if an instance of a Class is correctly created.
3. **Class Satisfiability** : If a Class exists in data, it must have an instance. A Class with no instance is like a variable never used - it's worthless.
4. **Subsumption** : Ensure that a subClass declared for a Class has been declared correctly.
5. **Classification** : Generates all subClass relationships for any given Ontology.

1.7.6 OWL Profiles

The Web Ontology Language provides flexibility in choosing profiles based on expressivity and scalability needs of the programmers. The most expressive ones with best decidability when it comes to reasoning power need the highest amount of computation time.

1. **OWL EL:** The computational complexity of this OWL profile is polynomial time. It is used if there are a large number of Classes and properties but the required constructs are not too complex.
2. **OWL QL:** It has the worst computational complexity of all OWL profiles and was designed for conjunctive query answering on basic relational databases.
3. **OWL RL:** It uses rule-based systems and the computational complexity for this profile is also polynomial time. Specifically designed for systems created using rule-based inference engines.
4. **OWL DL:** This OWL profile has second highest expressivity and can provide very high decidability and completeness in its inferences. It has one restriction: it will terminate after a specified amount of time if a computational answer cannot be reached.
5. **OWL FULL:** The most expressive OWL profile that provides the best decidability, soundness and completeness in its inferences with no restrictions on the number of constructs. There is no guarantee that any reasoning computation written using this profile will ever terminate and reach an answer because it can go on searching forever.

1.8 SPARQL

RDF or the Resource Description Framework is a labeled and directed format to store graph data and represent it on the World Wide Web (WWW). RDF is made use of to represent personal information about individuals, social network graphs that denote connections between individuals and provide a way to integrate multiple sources of information. RDF defines the syntax and the necessary semantics of the SPARQL, which is a querying language for the Resource Description Framework. This language was designed keeping in mind the use cases and requirements of the RDF.

Before we begin in earnest, let's take a simple example where data for a graph will be given with Turtle, so each triple can be explicitly shown.

Let us look at a simple query over a simple piece of data:

1.8.1 Data

<http://learningtest.org/titles/book1> <http://abc.org/dc/titles/1.1/title> "SPARQL Learning" .

Query

```
SELECT ?title
WHERE
{
 <http://learningtest.org/titles/book1> <http://abc.org/dc/titles/1.1/title> ?title .
}
```

Result

title
SPARQL Learning

Explanation

- The SPARQL query above will find the title of the book from the data graph shown in the form of a triplet. The SELECT clause of the query identifies the variables in the result and the WHERE clause provides the graph pattern to match against. The graph here is a simple triple pattern with one variable in the object position (?title).
- This can be confusing. Let's imagine a graph for a band called "Funk" with an album called "Funky" and a song in that album called "Full Funk" released on December 29, 1994. This band has 4 members: Tushar, Hardik, Ram and Prerna. Each of them is also a solo player.
- Here's the graphical representation of such a band:

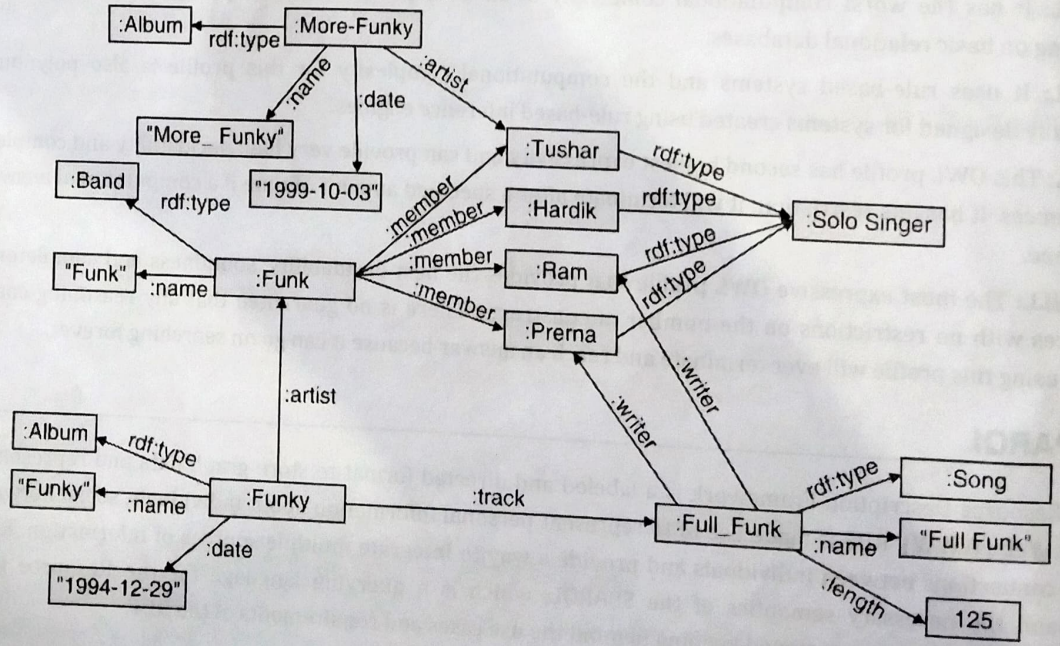


Fig. 1.8.1

1.8.2 SELECT

The primary SPARQL query is a simple SELECT query which is similar to an SQL query. It has two parts:

1. The selected variables; and
2. A WHERE clause for specifying what to search.

```
SELECT <your variables>
WHERE {
  <graph pattern to search>
}
```

The result of this query will be a table with one column for each selected variable and a row for each pattern that matched.

1.8.3 TRIPLES

We can make use of triple patterns to find any matching triples in the graph we have shown above and the variables in the query become wildcards that can match with any node.

```
SELECT ?album
WHERE {
  ?album rdf:type :Album .
}
```

This is a simple SPARQL SELECT query with just one triple pattern: ?album rdf:type :Album.

This is a task to search for any nodes where the rdf:type is "Album".

Another way to write this is:

```
SELECT * { ?album a :Album }
```

Result

album
:Funky
:More_Funky

1.8.4 Graph Patterns

We can use multiple triple patterns together as a graph pattern. Let's add a triple pattern to the query above and get the artist with each album.

```
SELECT *
{
  ?album a :Album .
  ?album :artist ?artist .
}
```

Result

album	artist
:Funky	:Funk
:More_Funky	:Tushar

Let us add another triple pattern to get album and artist information for only artists of the SoloSinger type.

```
SELECT *
{
  ?album a :Album .
  ?album :artist ?artist .
  ?artist a :SoloSinger .
}
```

Here, only one member is found:

album	artist
:More_Funky	:Tushar

1.8.5 Ordering Results

If the query above was run on a much larger dataset with hundreds of songs and albums, you will get hundreds of rows in no specified order. Plus, SPARQL does not have an inbuilt ordering system, so everytime you run it, the table might look different.

To get results sorted by ordering them using a condition, use the keyword "ODRER BY" just like SQL:

```
SELECT *
{
  ?album a :Album .
      :artist ?artist .
      :date ?date .
}
ORDER BY ?date
```

Result

Album	artist	date
:Funky	:Funk	"1994-12-29"
:More_Funky	:Tushar	"1999-10-03"

1.8.6 Slicing

If the database is too large and the result is taking too much time to compute, you can also just slice the result to get only a few rows. This is done with the "LIMIT" keyword.

In the query below, out of the two rows in the last example, only one will be shown and just to show more functionality, I have also reversed the "ORDER BY" keyword to show dates in descending order (just like SQL).

```
SELECT *
{
  ?album a :Album .
      :artist ?artist .
      :date ?date
}
ORDER BY desc(?date)
LIMIT 1
```

Result

Album	artist	date
:More_Funky	:Tushar	"1999-10-03"

1.8.7 Filters

Results can be filtered based on any needed criteria using the comparison, logical and mathematical operators just like SQL.

1. Comparison operators are as follows: (=, !=, <, <=, >, >=)
2. Logical operators are as follows: (&&, ||, !)
3. Mathematical operators are as follows: (+, -, /, *)

Let us filter on the query to find albums released after 1995:

```
SELECT *
{
  ?album a :Album .
      :artist ?artist .
      :date ?date
  FILTER (?date >= "1995-01-01")
}
ORDER BY ?date
```

Result

Album	artist	date
:More_Funky	:Tushar	"1999-10-03"

We can also use a function to convert dates to just the year component and then filter on that as shown below:

```
SELECT *
{
  ?album a :Album .
```

```
    :artist ?artist .
    :date ?date
  FILTER (year(?date) >= 1995)
}
ORDER BY ?date
ORDER BY ?date
```

Result

Album	artist	date
:More_Funky	:Tushar	"1999-10-03"

1.8.8 Binding

The output of a function can be bound to a variable using the "BIND" keyword. This ensures reusability. Let's rewrite the same query from above by first attaching the result of the year function to a variable and then using that for a filter.

```
SELECT *
{
  ?album a :Album .
      :artist ?artist .
      :date ?date
  BIND (year(?date) AS ?year)
  FILTER (?year <= 1995)
}
ORDER BY ?date
```

Result

album	artist	date
:Funky	:Funk	"1994-12-29"

1.8.9 Deduplication

Let us assume that the dataset has a large number of albums and songs. If we select only the year from all songs and print that, if any songs/albums were released in the same year, we'll see a large number of duplicates. In order to handle this issue, we use the keyword "DISTINCT" just like SQL. We will only see two rows because our limited dataset has only two items.

```
SELECT DISTINCT ?year
{
 ?album a :Album ,
     :artist ?artist .
     :date ?date
 BIND (year(?date) AS ?year)
}
ORDER BY ?date
```

Result

date
1994
1999

1.8.10 Aggregation

Aggregation refers to applying functions to a list of values rather than just one. If we want to find the earliest date of release of the band's songs/albums and latest date for the band' songs/albums, we will use the MIN/MAX functions on the aggregated dates of all songs/albums available in the dataset.

Many built-in aggregate functions are available in SPARQL: COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT, and SAMPLE.

Let us look at an example of MIN and MAX

```
SELECT (MIN(?date) as ?minDate) (MAX(?date) as ?maxDate)
{
  ?album a :Album .
      :date ?date
}
```

Result

minDate	maxDate
"1994-12-29"	"1999-10-03"

What if we want to count the number of albums?

```
SELECT (count(?album) as ?count)
{
  ?album a :Album
}
```

Result

count
2

1.8.11 Grouping

We can also group results from multiple rows and variables to condense them into tables that provide aggregated or combined intelligence. This is useful for reporting.

```
SELECT ?year (count(distinct ?album) AS ?count)
{
  ?album a :Album ;
      :date ?date ;
  BIND (year(?date) AS ?year)
}
```

GROUP BY ?year

ORDER BY ?count

Result

year	count
1994	1
1999	1
...	...
...	...

1.8.12 Describe

DESCRIBE returns the RDF graph. It is useful when you don't know anything about the graph.

DESCRIBE : Funk

The resulting triples has :Funk as the subject.

```
:Funk a :Band ;
  :member :Tushar , :Hardik , :Ram , :Prerna ;
  :name "Funk" ;
  :description "..." .
```

1.8.13 Update

We can add to the RDF graph using SPARQL. Let us another member to the "Funk" band:

```
INSERT {
  ?member a :Suraj
}
WHERE {
  ?band a :Funk ;
      :member ?member
}
```

Result : Suraj is now a member of the band "Funk".

1.8.14 Ask

These queries return a boolean value if the graphical expression used has a match and returns false if no match is found. This query is used to find boolean answers and is faster than querying for actual data, so in cases when the database is large and you only need an answer but don't care about the data, this is a great feature.

The following example query is asking if there is any band in the database that has been credited as the writer of a song:

```
ASK {
  ?band a :Band .
  ?song :writer ?band .
}
```

Result: true.

1.9 Differences between SQL and SPARQL

Table 1.9.1 : Differences between SQL and SPARQL

SPARQL	SQL
Used to access a web of linked data	Used to access relational databases
SELECT <{variable list}> WHERE <{graph pattern}>	SELECT {attribute list} FROM {table} WHERE {test expression(s)}
It can federate queries across various repositories. NOTE: Query federation means to send a query to an external table and get the result as an output table which can be used in your query.	No query federation.
SPARQL was designed to deal with data where relationships are often stated in the data.	SQL was designed to use data without relationships.
SPARQL supports ASK queries where the answer is a boolean True or False.	No ASK support in SQL.

Review Questions

Q. 1 Explain the progression of the Web from 1.0 to 5.0. **(5 Marks)**

Q. 2 What is Clickstream Analytics? State its benefits and drawbacks. **(5 Marks)**

Q. 3 What questions do you ask a vendor before you choose one for your company? **(10 Marks)**

Q. 4 What are the top KPIs that help determine the success of your website? **(5 Marks)**

Q. 5 Give the full form of and define the following: a) RDF and b) URI **(5 Marks)**

Q. 6 Explain in simple language: the Semantic Web Stack. **(10 Marks)**

Q. 7 Differentiate between the Semantic Web and Artificial Intelligence. **(5 Marks)**

Q. 8 Give an example of a Simple Triple in two ways: a) one simple sentence in English, and b) the sentence's Subject Predicate Object form. **(5 Marks)**

Q. 9 Explain OWL, its basic constructs and its types of properties. **(10 Marks)**

Q. 10 What is OWL? Explain the three ways in which OWL properties can be related to each other. **(5 Marks)**

Q. 11 What are OWL profiles? Explain the 5 OWL profiles. **(10 Marks)**

Q. 12 Explain the following in SPARQL : a) Slicing, and b) Filtering **(5 Marks)**

Q. 13 How are SQL and SPARQL different? **(5 Marks)**

❑❑❑

2 TypeScript

Syllabus

Overview, TypeScript Internal Architecture, TypeScript Environment Setup, TypeScript Types, variables and operators, Decision Making and loops, TypeScript Functions, TypeScript Classes and Objects, TypeScript Modules **Self-learning Topics :** Javascript Vs TypeScript

2.1 TypeScript Overview

TypeScript is an open-source, object-oriented programming language which was created and is still maintained by Microsoft.

TypeScript is an extension of JavaScript. It becomes an extension by adding additional data types, classes, and other object-oriented features like type-checking. By doing so, TypeScript becomes a superset of JavaScript which compiles just like plain old JavaScript.

JavaScript is a dynamic programming language but it has no type system. It provides the most basic and primitive data types like string, number and object, but cannot check any values assigned to a variable. JavaScript variables must be declared using the "var" keyword, and can be made to point to a value, but it doesn't have any functionality to support classes and other object-oriented programming features. So, in the absence of a type system, it is quite difficult to use JavaScript to create complex applications with multiple people developing and maintaining the same code.

Having a type system increases the quality of the code, increases readability and legibility & makes it easy to maintain and refactor the entire codebase. But most importantly, any errors can be caught at the compile time itself rather than being pushed ahead to runtime. This is where TypeScript has an advantage: it catches any errors at compile-time itself, so that you can fix them before you run your code. TypeScript also supports object-oriented programming features like multiple data types, classes, enums and a lot more and by doing so, it allows JavaScript to be used at scale.

TypeScript compiles into basic JavaScript. This means that the TypeScript compiler is implemented in TypeScript but can also be used with any browser or JavaScript engines like Node.js. All of the popular JavaScript frameworks like Angular.js and WinJS have been written using TypeScript.

2.1.1 Primary TypeScript Features

- **Cross-Platform :** TypeScript can run on any platform that JavaScript can run on. The TypeScript compiler can also be installed on any OS (Windows, Mac, Linux, Ubuntu).
- **Object-Oriented :** TypeScript gives you access to powerful OOP features like Classes, Interfaces and Modules. We can write pure OOP code for both client and server side.

- **Static Type-Checking** : TypeScript makes use of static typing by using type annotations at compile time itself. So, errors can be found beforehand. Plus, if a variable has been declared without a data type, the type will be inferred based on the value assigned to it (just like Python).
- **Optional Static Typing** : TypeScript static typing is an optional feature and not mandatory, so you can even use JavaScript's dynamic typing if you wish to.
- **DOM Manipulation** : Just akin to JavaScript, TypeScript can manipulate Document Object Model to manipulate web document structures.

2.1.2 Advantages of TypeScript

1. Open source with continuous development and improvement.
2. Can run on any browser or JS Engine.
3. Very similar to JS so limited additional learning is needed.
4. Code is similar to backend languages like Java and Scala, so easier to work with for both sides.
5. Data types can be inferenced.
6. Can be invoked from existing JS Code, so less integration issues.
7. Supports all JS libraries, so all existing benefits remain and only new ones are added.

2.2 TypeScript Internal Architecture

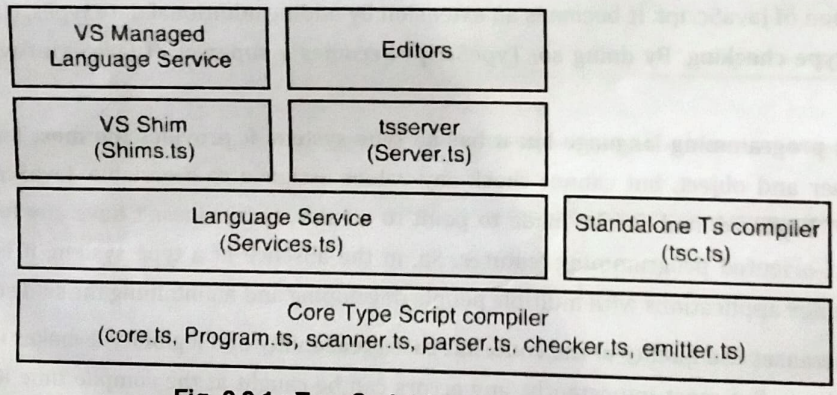


Fig. 2.2.1 : TypeScript Internal Architecture

- **Parser :** Starting with the data sources and following the rules of the language grammar, it creates an Abstract Syntax Tree (AST).
- **Binder :** It links declarations to the structure using symbols which are different declarations of the same interface or module or a function and a module. This lets the type system reason with named declarations and understand them.
- **Type Resolver/Checker** : It resolves the types of constructs, checks for any semantic operations and creates diagnostics as needed.
- **Emitter :** It outputs from any number of inputs (.ts and .d.ts). Files could be multiple types like JavaScript (.js), definitions (.d.ts), or javascript maps (.js.map)
- **Pre-processor** : It refers to all files used in a "program". The context is created by looking at all files sent to the compiler and then adding any files that the original files may reference directly or indirectly. The result of this exercise is an ordered list of source files that create what we call a program.

- **Standalone Compiler (tsc) :** It chiefly handles reading and writing files for all the supported engines (e.g. Node.js). It does the actual parsing of code, type checking within the code and the transformation of TypeScript into JavaScript.
- **VS Managed Language Service :** It exposes a layer around the core compiler to support a set of edit-operations like statement completions, signatures, formatting and outlining, colors, etc. The language service gives us a slightly new perspective around working with code and source files as compared to other compilation interfaces. It basically contains the TypeScript language elements.
- **Language Service (services.ts) :** It gives birth to information that helps the tools provide better features like IntelliSense (what parameters you type, what values to use, keep track of data types, etc) or automated refactoring.
- **Standalone Server (tsserver) :** The standalone server wraps the compiler and the services layer within it and exposes them to the end user through a JSON protocol.

2.3 TypeScript Necessary Environment Setup

TypeScript is an Open Source Object Oriented Programming Language. It can run on absolutely any browser and on any Operating System. All we need are some tools to write and then subsequently test a Typescript program.

Text Editor : A text editor will aid in the writing of a source code file. Examples of such text editors include Notepad, Notepad++ on Windows, vim or vi on Linux, Atom, Pages on Mac, etc. Text editors can vary with the OS being used.

The source code files for TypeScript are saved with the extension ".ts"

The TypeScript Compiler : The TS compiler is also just a .ts file which has been compiled down to JavaScript (.js) file. The TypeScript Compiler or TSC creates a JavaScript version of the .ts source code file that has been passed. This is called transpilation, This TS compiler will however, not process any raw JS file given to it. The TS compiler handles exclusively ".ts" or ".d.ts" files.

Node.js : Node.js is an open source runtime environment that is used for server-side JavaScript programs. It is needed for JavaScript runs without any browser assistance. It can be downloaded as Node.js source code or as a pre-built .exe installer for your OS.

2.4 TypeScript Data Types

1. **Number** : Numbers are the most basic floating point values. They can be assigned any numeric value like decimals, hexadecimals, binaries, and octals.

Examples

- *let decimalValue: number = 25;*
- *let hexaDecimalValue: number = 0xb10a;*
- *let binaryValue: number = 0b100111;*
- *let octalValue: number = 1o301;*

2. **String :** To be used for any kind of textual data. They are used and are shown using the keyword "string". Akin to JavaScript, TypeScript also makes use of double quotations (") and single quotations (') on both sides of the string.

Examples

- *let firstName : string = "Tushar"; // using double quotations*
- *let lastName : string = Agarwal; // using single quotations*

3. **Boolean :** For true/false flags and indicators, we use the "boolean" keyword.

Examples

- *let isPrimaryAccountHolder: boolean = false;*
- *let isDead: boolean = true;*

4. **Enum :** These are a kind of numeric values with easier names for friendlier usage. This was added to JavaScript. The variables of this data type are declared with the keyword "enum".

Examples

- *enum FruitTypes { mango, apple, banana}*
- *let fruit: FruitTypes = FruitTypes.mango;*

5. **Void :** This data type is used if a function does not return any value. In TypeScript, you can declare a variable with the type void, but if you do so, you can only assign "undefined" or "null" to it.

6. **Null :** This data type can be used to declare a variable of type null using the keyword "null" and can assign only a null value to it. A "null" can be assigned to a numeric or boolean value.

Examples

- *let booleanVariable: flag = null;*
- *let numericVariable: num= null;*

7. **Undefined :** The "undefined" keyword can be used to assign an undefined value. It is a subtype of all other types like the "null" keyword and thus can be assigned to a numeric or boolean value.

Examples

- *let booleanVariable: flag = undefined;*
- *let numericVariable: num = undefined;*

8. **Array :** We can use arrays in TypeScript (TS) by defining them in two ways; both of which have been shown below.

Examples

- *let gradePoints: number[] = [70, 75, 90];*
- *let gradePoints: Array<number> = [70, 75, 90];*

9. **Tupe :** Tuples are the kind of data type that let you create an array where the data type of the elements is known to us. In order to access an element inside a tuple, you would need an index and using this index, the data of the correct data type will be returned. Asking for incorrect data type will result in an error.

Examples

```
// correct
let person: [string, number] = ["Tushar", 1994];
// error
let person: [string, number] = [1994, "Tushar"];
```

10. **Any :** If we are not sure of a variable's data type because you will be accepting data from a function or a library or from the user,, we use the keyword "any" to declare the variable. If you have an array with mixed data types, this variable can come in handy.

Examples

```
let dynamicVariableValue: x= "Tushar";          // I've set it with a string first
dynamicValue = 1994;                            // Now I assign a number to it
dynamicValue = true;                            // I can even give it a boolean now
let dynamicArrayList: array_x[] = [ "Tushar", 1994, true ];
```

11. **Never :** This data type is used if you never want to return a value from a variable. For instance, if a function were to always trigger an exception and never return a value, we would use this data type. Here, we simply set the return data type for the function to "never".

Example

```
function triggerError(message: string): never {
 throw new Error("Error 404: Brain Not Found");
};
```

2.5 TypeScript Variables

What is a Variable?

A variable is, in terms of computer programming, just a reserved space in memory with a name attached to it that can store a value once assigned to that memory. It is simply a memory container. It can be both full and empty based on whether the programmer has actually assigned a value to that block or declared the variable and left it unassigned.

TypeScript variables follow the same rules for variables as JavaScript.

1. Variable definition names can be made of both alphabets and numerals.
2. Variable definition names cannot use spaces or any special characters with the exception of an underscore and the dollar sign.
3. Variable definition names cannot start with a numeral.
4. A variable in TypeScript has to be declared before being used.
5. The "var" keyword is used to declare a variable in TypeScript.

The syntax to declare a variable in TypeScript includes the use of a colon (:) right after the variable name, followed by the data type for that variable. We always use the keyword "var" to declare TypeScript variables.

TypeScript has 4 ways to define a variable :

Declare the variable data type and its value together.	var myName:string = "Tushar"
Declare the variable data type without a value and the variable will automatically be set to undefined.	var myName:string;
Declare the variable's value but not the data type. The variable's data type will be automatically inferred and set.	var myName= "Tushar"
Declare neither the data type of the variable nor its value. The data type of such a variable will now become "any" and it will be set to undefined.	var myName;

TypeScript Variable Typing

TypeScript follows ***Strong Typing for Variables*** - meaning that the data type on both sides of an " = " must be the same, or a compilation error will be thrown as shown below :

```
var x:number = "Tushar"    // compilation error
```

TypeScript Variable Assertion

TypeScript also allows ***Variable Type Assertion*** - the ability to change the data type of a variable. This is done by putting the new data type between angular brackets < > and placing it ahead of the variable.

Example

- *let numericalID: any = 999;*
- *let studentID = <number> numericalID;*

TypeScript Variable Inferred Typing

TypeScript also allows ***Inferred Typing for Variables*** - the ability to detect the best suited data type for a variable based on its first occurrence in the program. This feature is optional in TypeScript. It means that you can declare a variable without mentioning its data type. The compiler automatically determines the data type of the variable by finding the first usage of this variable within the code and then using it for the rest of the program.

Example Code

```
var x = 5;  // compiler read it as a number
console.log("The number is: "+x);
x= "Tushar"; //compilation error
console.log(x);
```

The compiler sees the first occurrence of the variable and reads 5 as a number and assigns it to the variable. If we now try to assign a string value to it, the compiler will throw an error.

TypeScript Variables Scope

The scope of any variable in TypeScript is determined by where that variable has been declared. Their scope thus determines where they will be available and from where they can be accessed.

Different kinds of TS Variable Scopes

- **Global Scope :** Declared outside programming constructs and can be accessed from anywhere in your source program.
- **Class Scope** : These are the variables defined within a class but outside all of that class' methods. Any object of this class can make use of the class variables.
- **Local Scope :** These variables are defined within the methods, loops or functions (class constructs) within a class and can only be accessed by the class construct that created it.

Example Code for Variable Scope

```
var global_var = 1;          // global variable
class VaiablerScoping {

 class_var = 2;              // class variable
```

```
dummyFunction():void {
   var local_var = 3;       // local variable
  }
}
```

2.6 TypeScript Operators

What is an Operator?

An operator explains the calculations / processing to be performed on data. The data over which these calculations are to be performed are known as the operands.

In 1 + 2 = 3.

1 and 2 are the operands, while + and = are the operators. 3 is the result on the operation.

The operators supported by TypeScript are as shown in Fig. 2.6.1.

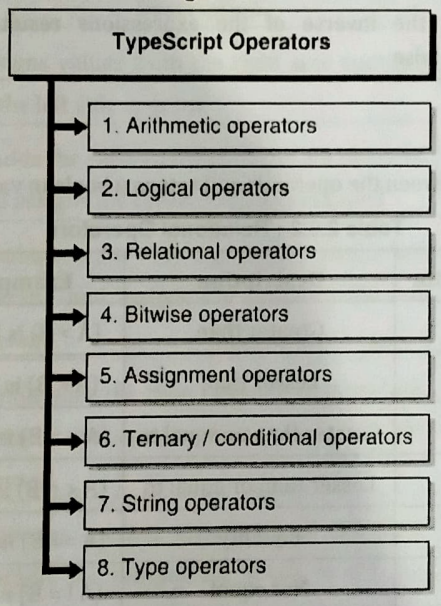


Fig. 2.6.1 : TypeScript Operators

2.6.1 Arithmetic Operators

Table 2.6.1 : Arithmetic Operators

Operator	Description	Example
+ (Addition)	Returns the sum of the operands	a + b is 15
– (Subtraction)	Returns the difference of the values	a – b is 5
* (Multiplication)	Returns the product of the values	a * b is 50
/ (Division)	Performs division operation and returns the quotient	a / b is 2
% (Modulus)	Performs division operation and returns the remainder	a % b is 0

Operator	Description	Example
+ + (Increment)	Increments the value of the variable by one	a ++ b is 11
– – (Decrement)	Decrements the value of the variable by one	a –– b is 9

2.6.2 Logical Operators

They are used to combine two or more operations and return boolean values as their result, i.e. True or False.

Table 2.6.2 : Logical Operators

Operator	Description	Example
&& (AND)	The operator returns true only if all the expressions specified return true	(A > 10 && B > 10) is False
|| (OR)	The operator returns true if at least one of the expression specified returns true	(A > 10 || B > 10) is True
! (NOT)	The operator returns the inverse of the expressions result. For E.g. : ! (> 5) returns false	! (A > 10) is True

2.6.3 Relational Operators

These operators test the relationship between the operands and return a boolean value, i.e. True or False.

Table 2.6.3 : Relational Operators

Operator	Description	Example
>	Greater than	(A > B) is False
<	Lesser than	(A < B) is True
> =	Greater than or equal to	(A > = B) is False
< =	Lesser than or equal to	(A < = B) is True
= =	Equality	(A == B) is False
! =	Not equal	(A ! = B) is True

2.6.4 Bitwise Operators

Table 2.6.4 : Bitwise Operators

Operator	Description	Example
& (Bitwise AND)	It performs a Boolean AND operation on each bit of its integer arguments	(A & B) is 2
| (Bitwise OR)	It performs a Boolean OR operation on each bit of its integer arguments	(A | B) is 3
^ (Bitwise XOR)	It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true but not both	(A ^ B) is 1
~ (Bitwise Not)	It is a unary operator and operates by reversing all the bits in the operand	(~ B) is – 4

Operator	Description	Example
< < (Left shift)	It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros, shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4 and so on.	(A << 1) is 4
>> (Right shift)	Binary right shift operator. The left operands value is moved right by the number of bits specified by the right operand.	(A >> 1) is 1
>>> (Right shift with zero)	This operator is just like the >> operator except that the bits shifted in on the left are always zero.	(A >>>) is 1

2.6.5 Assignment Operators

Table 2.6.5 : Assignment Operators

Operator	Description	Example
= (Simple Assignment)	Assigns values from the right side operand to the left side operand	C = A + B will assign the value of A + B into C
+ = (Add and Assignment)	It adds the right operand to the left operand and assigns the result to the left operand	C + = A is equivalent to C = C + A
– = (Subtract and Assignment)	it subtracts the right operand form the left operand and assigns the result to the left operand	C – = A is equivalent to C = C – A
* = (Multiply and Assignment)	It multiplies the right operand with the left operand and assigns the result to the left operand	C * = A is equivalent to C = C * A
/ = (Divide and Assignment)	It divides the left operand with the right operand and assigns the result to the left operand	

2.6.6 Ternary / Conditional Operators

The ternary or conditional operator is a test of a condition performed as follows:

Conditional Test ? Expression 1 : Expression 2

- Conditional Test – the expression to be tested
- Expression 1– the value returned by the operator if the conditional test is true
- Expression 2– the value returned by the operator if the conditional test is false

2.6.7 String Operators

The "+" sign is used as a string operator when concatenation of two strings is needed.

```
var msg:randomString = "Tushar"+"Agarwal"
```

2.6.8 Type Operators

This is a unary operator denoted using the keyword "***typeof***". It returns the data type of the operand fed to it.

```
var x= 1994
console.log(typeof x);  // Output: number
```

2.7 TypeScript Decision Making

Decision making is the ability in programming to create branches in the flow of the program based on conditions created by the programmer, the coming true of which triggers an expected outcome. Essentially, if a condition is True, a certain block of code is to be run and if the condition is False, another block of code is run.

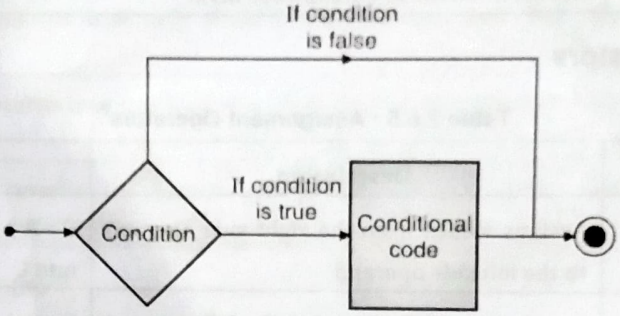


Fig. 2.7.1

Let us look at the kinds of decision statements supported by TypeScript

Table 2.7.1

Sr. No.	Statement and Description
1.	if statement An 'if' statement has a Boolean expression followed by one/more statement
2.	is ... else statement An 'if' followed by an optional else which excite when Boolean expression is False
3.	else ... if and nested if statements We can use just one 'is' statements inside an offer 'is' or 'else if' statement
4.	Switch statement
5.	Allows a variable to be tested against a list of values and execute code based on the value

2.8 TypeScript Loops

Sometimes, a code block may need to be executed several times. The basic loop structure is applicable to TypeScript where for as long as a condition remains set by a programmer, a block of code will execute on demand over and over again.

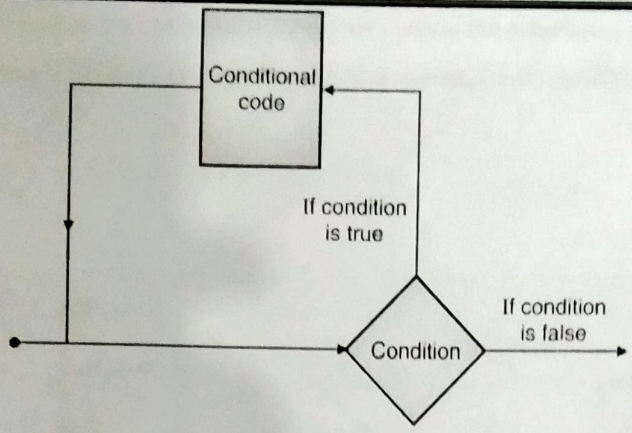


Fig. 2.8.1

Table 2.8.1

Sr. No.	Loops and description
1.	For loop The for loop is an implementation of a definite loop
2.	While loop The while loop executes the instructions each time the condition specified evaluates to true
3.	do ... while The do... while loop is similar to the while loop except that the do...while loop doesn't evaluate the condition for the first time the loop executes

For Loop Example

```
for (let x = 0; x < 3; x++) {
  console.log ( x );
}
```

Output

```
0
1
2
```

While Loop Example:

```
let x: number = 2;
while (x < 5) {
  console.log ( x )
  x++;
}
```

Output:

2

3

4

Do While Loop Example

```
let x: number = 2;
do {
   console.log( x )
   x++;
} while ( x < 5)
```

Output

2

3

4

The "break" keyword can be used in TypeScript as well to break out of a construct within which it is invoked. It essentially ends the "loop".

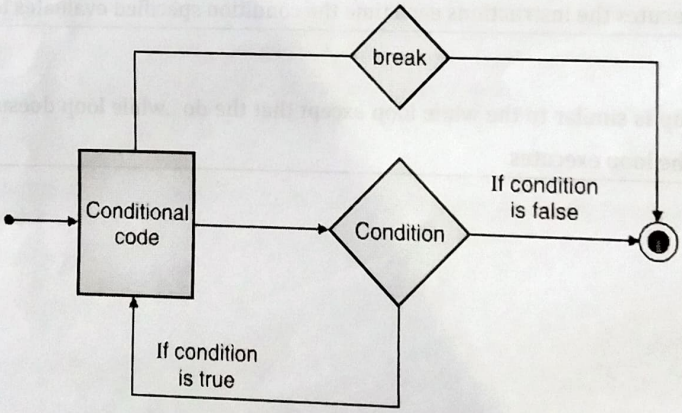


Fig. 2.8.2

```
var x = 1;
while (x <= 20) {
  if (x % 10 == 0) {
    console.log("The loop was broken at : " + x);
    break;
  }
  x++;
} // loop outputs 10 and then breaks
```

The "continue" keyword moves back to the condition and does not execute the subsequent statements after the keyword in the conditional block for the current iteration. It simply goes back to re-evaluate the condition.

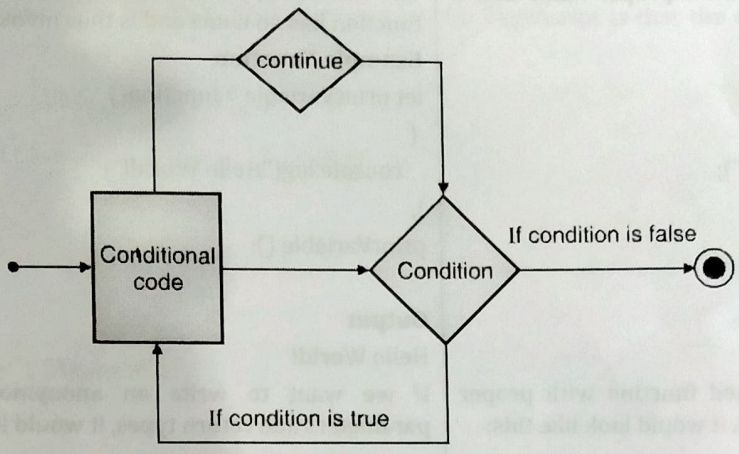


Fig. 2.8.3

```
var num:number = 0
var count:number = 0;
for(x=0;x<=10;x++) {
  if (x % 2==0) {
    continue
  }
  console.log (" Current loop number: "+x)    //outputs 1,3,5,7,9
}
```

2.9 TypeScript Functions

Functions are the primary code blocks of any programming language. Only with functions can we implement object-oriented concepts like classes, objects, polymorphism, abstraction, etc.

They make the program reusable because the same function can be called any number of times. They make the code maintainable by keeping it short and allowing different functions to handle different functionalities.

Functions can be created with or without parameters. Parameters or as many call them, arguments to a function are the operands passed to a function that need to be worked upon within the function. TypeScript requires the same number of parameters to be passed during invocation as were defined for that function - no more and no less. We will see examples of this ahead.

2.9.1. Function Types

In TypeScript, functions are of two types - named functions and anonymous functions

Table 2.9.1

Named Function	Anonymous Function

The function is declared with a proper name and is invoked by that name.
Example Function
function printSomething()
{
console.log("Hello World!");
}
display();
 	
Output
Hello World!
If we want to write a named function with proper parameters and return types, it would look like this:
 	
Example Function
function AddNumbers(x: number, y: number) : number
{
console.log(x+y);
}
AddNumbers(1,2);
 	
Output
3
	The function is just an expression and is stored in a variable. The function has no name and is thus invoked using the variable name.
	Example Function
	let printVariable = function()
	{
	console.log("Hello World!");
	};
	printVariable ();
	
	Output
	Hello World!
	If we want to write an anonymous function with proper parameters and return types, it would look like this:
	
	Example Function
	let SumVaraible = function(x: number, y: number) : number
	{
	console.log(x+y);
	}
	SumVaraible (1,2);
	
	Output
	3

2.9.2 Arrow Functions

Arrow functions are an alternate way of writing anonymous functions. Some languages even call them lambda functions.

Using a fat arrow as depicted in the example below (=>), we no longer need to use the "function" keyword. Parameters are passed as usual within the parentheses and the function is enclosed with curly braces.

Example Arrow Function

```
let addNumbersVariable= (x: number, y: number): number => {
  console.log(x+y);
}
addNumbersVariable(1, 2);
```

Output

3

An arrow function like a normal anonymous function, can also be written without any parameters:

```
let printConsoleVariable = () => console.log ("Hello World!");
printConsoleVariable ();
```

Output

Hello World!

2.9.3 Function Overloading

TypeScript allows for the concept of function overloading by allowing you to reuse a function name with different parameter types and return types. The only condition enforced by TypeScript is that the number of parameters for the overloaded functions should be the same.

```
function addTwoThings (a:string, b:string): string;
function addTwoThings (a:number, b:number): number; {
  console.log ( a + b );
}
```

Input

```
addTwoThings ("Tushar ", "Agarwal");
```

Output

Tushar Agarwal

Input

```
addTwoThings (1, 2);
```

Output

3

2.9.4 Rest Parameters

When you don't know how many parameters your function may receive, you use this functionality. TypeScript can bundle multiple arguments to a function using a variable and bind them together to be used.

As an example, you want users to enter their name. A person could have 1 string in their name (Beyonce) or they could have even six (Parampeel Parambatur Chinnaswami Muthuswami Venugopal Iyer). In such a case, rest parameters can be used:

```
function buildPersonName(firstName: string, ...RemainingName: string[]) {
 console.log ( firstName + " " + RemainingName.join(" ") );
}
let variableName = buildPersonName("Parampeel", "Parambatur ", "Chinnaswami ", "Muthuswami ", "Venugopal ",
"Iyer");
```

Output

Parampeel Parambatur Chinnaswami Muthuswami Venugopal Iyer

```
let variableName = buildPersonName("Beyonce");
```

Output

Beyonce

2.10 TypeScript Classes and Objects

2.10.1 Classes Recapped

Classes are the blueprint in Object Oriented Programming used to create objects.

The "class" keyword is used in TypeScript to declare a class. After this keyword, you add the name of the class. A class definition in TypeScript must include these:

- **Fields :** These are the variables declared in a class. This is the data used by objects.
- **Constructors :** They provide memory for the objects of a class.
- **Functions :** The actual work / processing for which the class was created. Also called methods.

All these components of the class combined are the data members for this class.

```
class Person
{
  //field
  name:string;
  //constructor
  constructor(name:string)
  {
    this.name = name
  }
  //function
  printName():void
  {
    console.log("Name is :  "+this.name)
  }
}
```

This is an example Class called Person. It has a field called "name". Please note that the "var" keyword is not necessary here and hence has not been used.

The constructor here is the special function that initializes the variables in this class. In TypeScript, we define a constructor for a class using the "constructor" keyword. Just like any other function, a constructor can also have parameters.

The "this" keyword refers to the current instance of the class. You will see that the parameter name on the right hand side is the same as the class' field on the left hand side. And so, we prefix the class' field with the keyword "this".

printName() is a simple function defined here. You will note that the keyword "function" is not needed here and thus, has not been used.

2.10.2 Instance of a Class (Object)

We create a new instance of a class in TypeScript by using the keyword "new" followed by the name of the class you want to create an instance of. This is then assigned to a variable object.

```
var object_name = new class_name([ arguments if any ])
```

The RHS here invokes the new class constructor along with any parameters if defined.

```
var object1 = new Person("Tushar")
```

2.10.3 Accessing Attributes and Functions via Objects

The functions and attributes defined within a class can be accessed after an object for that class has been created using the dot notation (" . ").

Let us access the Class created above.

```
//creating an object of the class
var object1 = new Person ("Tushar")
//accessing the class' field
console.log ("Print my name :  "+object1.name)
//accessing the class' function
obj.printName ()
```

2.10.4 Inheritance in Classes

TypeScripts allows a class to inherit another class' methods and functions via the use of the keyword "extends". The class that uses an existing class' methods is called the child class while the class being used is the parent class.

Child classes will inherit all the properties and the methods except any private ones.

```
class Shape
{
  Perimeter:number
  constructor(p:number)
 {
    this.Perimeter = p
  }
}

class Square extends Shape
{
    printPerimeter ():void
  {
     console.log("Perimeter of the square: "+this.Perimeter)
   }
}
var obj = new Square (115);
obj.printPerimeter ()
```

Output

```
Perimeter of the square: 115
```

Inheritance is of three distinct kinds

- **Single Inheritance** : Classes can extend from at the most one parent class. TypeScript allows single inheritance.
- **Multiple Inheritance** : Classes can inherit their properties from multiple classes together. TypeScript does not allow this.
- **Multilevel Inheritance** : Classes can inherit in a chain. Class C inherits from B and Class B inherits from A, so Class C can now access properties and functions from Class A. TypeScript allows this form of Inheritance as well.

2.10.5 Method Overriding

This is the functionality with which a child class can override the functionality of the parent class' methods. It is achieved using the "super" keyword. This keyword refers to the parent class version of the variable. An example of this functionality can be seen below:

```
class A
{
  protected printSomething()
  {
    console.log("Printing from Class A")
  }
  // Expose the protected method as a public function
  public callPrintSomething()
  {
    this.printSomething();
  }
}
class B extends A
{
  // Overriding the method from Class A
  protected printSomething()
  {
    super.printSomething();
    console.log("Printing from Class B")
  }
}
var x = new A();
x.callPrintSomething();
```

Output

```
Printing from Class A
```

```
var y = new B ()
y.callPrintSomething ();
```

Output

```
Printing from Class A
Printing from Class B
```

2.10.6 Encapsulation

Classes in TypeScript or most other object oriented programming languages for that matter, can control visibility of their resources/data to decide what part of the code can access what kind of data.

Programmers call the keywords used for data hiding (encapsulation) access modifiers.

Table 2.10.1

Sr. No.	Access specifier and description
1.	**Public** A public data member has universal accessibility. Data member in a class are public by default
2.	**Private** Private data members are accessible only within the class that defines these members. If an external class member tries to access a private member, the complier throws an error.
3.	**Protected** A protected data member is accessible by the members within the same class as that of the former and also by the members of the child classes.

Example

```
class DataHidingOrEncapsulation{
  x:string = "Tushar"
  private y:string = "Agarwal"
}
var obj = new DataHidingOrEncapsulation()
console.log(obj.x)
```

Output

```
Tushar
console.log(obj.y)
Compilation Error
```

2.10.7 Interfaces

An interface defines a structure for any classes you define in your program. It is a contract. If a class or method in the program uses the interface, it can access all its methods but only if it adheres to the structure defined by the interface.

The keyword "interface" is used by TypeScript to define an interface.

The following is a contract defined by the interface. Any class or method that wants to use its properties and methods must implement this interface and abide by its structure.

```
interface KeyValuePair
{
  keyComponent: number;
  valueComponent: string;
}
```

The following are examples that make use of this interface:

```
let x: KeyValuePair = { keyComponent:1, valueComponent:"Tushar" };
// Successful
let y: KeyValuePair = { keyComponent:1, value:"Tushar" };
// Compilation Error: 'value' doesn't exist in KeyValuePair
let z: KeyValuePair = { keyComponent:1, valueComponent:1 };
// Compilation Error: Incorrect data type
```

In the above example, an interface "KeyValuePair " has been defined and it has two properties:

1) keyComponent and
2) valueComponent.

Variables x and y have been defined as KeyValuePair type, so they must follow the structure defined by the Interface. Only an object with a number type (keyComponent) and a string type (valueComponent) can be assigned. Any change to the name or data type will result in a compilation error.

Interface Extension: Interfaces can even extend to use another interface's properties and methods.

```
interface Human
{
  name: string;
  gender: string;
}
interface Student extends Human
{
  studentCode: number;
}
let studentObj:Student  = {
  studentCode: 44156,
  name: "Tushar",
  gender: "Male"
}
```

If the Student Interface above is used to create an object, the Human Interface's properties and methods must be included. Not doing so will lead to a compilation error.

Implementing an Interface: Just like any other object oriented programming language, TypeScript also allows Classes to "implement" an interface by using the keyword "implements".

```
interface Student
 {
   studentCode: number;
   studentName: string;
   studentAge: (studentCode: number) => number;
}
class NewStudent implements Student
{
   studentCode: number;
   studentName: string;

    constructor(code: number, name: string)
    {
      this.studentCode = code;
      this.studentName = name;
    }
    getAge (studentCode:number):number {
    return 18;
  }
}
let x = new Student(123, "Tushar");
x.getAge(123);
```

Output

```
18
```

Here, the Student Interface is implemented by the Class NewStudent using the "implements" keyword. The implementing class here is defining all the properties of the Interface in adherence with the contract using the same name and the same data type. Now doing so would result in a compilation error.

2.11 TypeScript Modules

TypeScript by default has global scope. This means that if your code is split into different files and you update a variable by mistake in one file, it will update that variable in another file (if it exists). This is dangerous.

SourceCodeFile1.ts

```
var randomVariable: string = "Hello World!";
```

SourceCodeFile2.ts

```
console.log (randomVariable);
randomVariable= "Hello Tushar!"; // This is acceptable in TypeScript
console.log(randomVariable);
```

Output

```
Hello World!
Hello Tushar!
```

A variable declared in file 1 has been updated by file 2. It is accessible and also open to modifications. Any random programmer can override variables without intent, opening up the program to unintended errors and bugs. In order to take care of such an issue, TypeScript has the concept of modules and namespaces.

Modules create a local scope within the file. Now, all methods, variables, functions, etc. in a module are not by default accessible outside the file. They can be used with the keywords "export" and "import". One module shares what it has by exporting itself and the other explicitly uses the exported module by importing it.

The syntax is as follows :

Import { resource name } from "file path upto file name without file extension"

Example

SourceCodeFile1.ts

```
export var randomVariable: string = "Hello World!";
```

SourceCodeFile2.ts

```
console.log (randomVariable);
// Compilation Error
```

SourceCodeFile3.ts

```
Import { randomVariable } from "...../SourceCodeFile1"
console.log (randomVariable);
randomVariable= "Hello Tushar!"; // This is now acceptable
console.log(randomVariable);
```

Output

```
Hello World!
Hello Tushar!
```

Let us look at a little more complicated example to understand the concept. The following is a file that wants to export its contents :

StudentFile.ts

```
export let age : studentAge = 18;
export class Student
{
  studentCode: number;
  studentName: string;
  constructor(sname: string, scode: number)
  {
    this.studentName = sname;
    this.studentCode = scode;
  }
  printStudent()
  {
    console.log ("Student Code: " + this.studentCode +  ", StudentName: " + this.studentName );
  }
}
let collegeName:string = "Carnegie Mellon";
```

Now, we have a file that wants to import the methods from the Student Class in StudentFile.ts

AccessRequestingFile.ts

```
import { Student} from "./StudentFile";
let studentObj = new Student ("Tushar Agarwal", 123);
studentObj .printStudent();
console.log("Student College: " + collegeName)
```

Output

Student Code: 123, Student Name: Tushar Agarwal

Student College: Carnegie Mellon

2.12 Differences between JavaScript and TypeScript

Table 2.12.1

JavaScript	TypeScript
Lightweight with good functions. Missing many OOP features	Powerful type system and has all JS features
Scripting Language	Object Oriented Language
Can be converted to TS by changing file extension from .js to .ts	Cannot be converted to JS by changing file extension from .ts to .js

JavaScript	TypeScript
No Static Typing	Enforces Static Typing (value has to be the same as the variable data type when defined so that type correctness can be checked at compile time itself)
Does not support modules	Supports modules
No support for Interfaces	Can implement interfaces
No support for compile time. JS is an interpreted language (only runtime).	Can point out errors in compile time
Faster execution	Takes time to compile because code is converted into JS upon compilation
Easy to learn	Steep learning curve. Having a scripting foundation helps.

Review Questions

Q. 1 What are the 5 primary features of TypeScript ? How are JavaScript and TypeScript different ? Explain with 5 differences. **(10 Marks)**

Q. 2 Explain TypeScript's Internal Architecture. **(10 Marks)**

Q. 3 List 10 TypeScript Data Types. How are JavaScript and TypeScript different? Explain with 5 differences. **(10 Marks)**

Q. 4 List 2 TypeScript Data Types with 2 examples each. **(5 Marks)**

Q. 5 What is a variable? List 4 ways a variable can be defined in TS. **(5 Marks)**

Q. 6 Explain Global Scope, Local Scope and Class Scope in TS with a pseudo code. **(5 Marks)**

Q. 7 Explain 5 TS Operators. **(10 Marks)**

Q. 8 List 3 kinds of loops allowed in TS. Draw a flow diagram and write the TS code to loop from 1 to 10 in TS for any two of them. **(10 Marks)**

Q. 9 Explain an arrow function in TS. **(5 Marks)**

Q. 10 Write the TS pseudo code for Inheritance in TS Classes. List the three kinds of Inheritance in TS. **(10 Marks)**

Q. 11 What is Encapsulation. Explain TypeScript's 3 access specifiers. **(5 Marks)**

Q. 12 Write a brief on Modules in TS with an example. **(10 Marks)**

Q. 13 Explain while loop and do while loop with one example of each type of loop. **(10 Marks)**

□□□

3 Introduction to AngularJS

Syllabus

Overview of AngularJS, Need of AngularJS in real web sites, AngularJS modules, AngularJS built-in directives, AngularJS custom directives, AngularJS expressions, Angular JS Data Binding, AngularJS filters, AngularJS controllers, AngularJS scope, AngularJS dependency injection, Angular JS Services, Form Validation, Routing using ng-Route, ng-Repeat, ng-style, ng-view, Built-in Helper Functions, Using Angular JS with Typescript

Self-learning Topics : MVC model, DOM model, Javascript functions and Error Handling

3.1 Prerequisites

3.1.1 Model-View-Controller

The Model-View-Controller or the MVC is an architectural framework that modularizes an application into three primary components segregated by the purpose they serve: 1) the model, 2) the view, and 3) the controller.

Each component is built to handle a tailored aspect of development for an application. It is the most widely recognized web development framework used to create projects that can scale well.

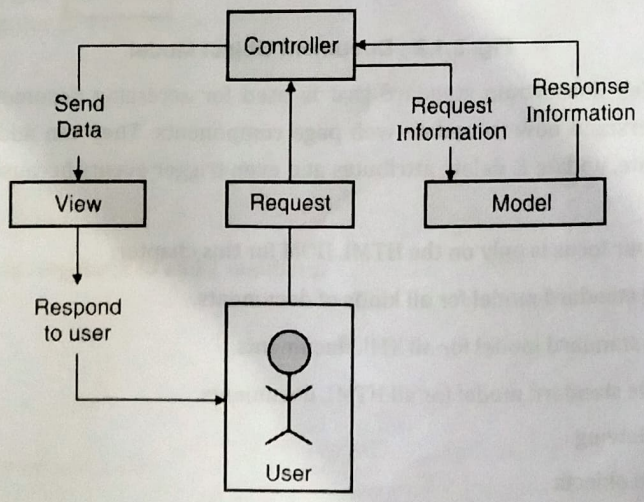


Fig. 3.1.1 : MVC architecture

1. Model

This component has all the logic and processes that deal with data. It could be as simple as getting data from View via the Controller and back or processing it first. It fetches data from databases, manipulates it, runs queries, spits out the results and even updates the databases if needed.

2. View

Handles all the UI (User Interface) logic for your application. The text boxes, forms, validations, final results, any tabular displays, the background, styling, etc. all come under the purview of this component. This is the only component that the user interacts with and can see.

3. Controller

This component is the interface between the Model and the View. It processes the business logic and contacts the Model based on the needs of the application. It handles the customer input, routes it as needed and controls the overall flow of the application.

3.1.2 Document Object Model

The Document Object Model or DOM is a tree of the objects created when the browser loads an HTML file.

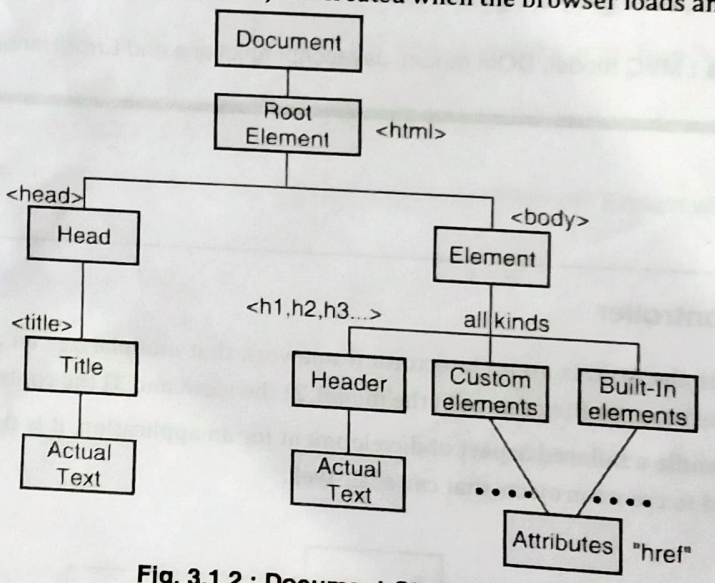


Fig. 3.1.2 : Document Object Model

The DOM is a World Wide Web Consortium standard that is used for accessing documents over the web and helping languages like JavaScript understand how to update web page components. They can add and remove HTML elements, update the styling via CSS, create, update & delete attributes and even trigger events because the DOM makes it possible to do so.

The DOM is of three kinds but our focus is only on the HTML DOM for this chapter:

1. **Core DOM** - this is the standard model for all kinds of documents.
2. **XML DOM** - this is the standard model for all XML documents.
3. **HTML DOM** - this is the standard model for all HTML documents.

The HTML DOM defines the following :

- All HTML Elements as objects
- All properties of HTML elements

- All methods to access the HTML elements
- All events for the HTML elements

3.2 AngularJS - An Overview

AngularJS is nothing more than a JavaScript framework which can be added to an HTML page using a script tag <script> </script>..

Knowing the basics of HTML and JavaScript is a prerequisite to understanding AngularJS (just like TypeScript).

AngularJS can extend the attributes allowed by HTML using what are called **Directives** and can bind data to HTML using what are called **Expressions**. We will discuss them in more detail over this chapter.

You can download the required AngularJS software from the library catalogue per your system specifications from angularjs.org and use any regular IDE that you would use to write JavaScript code.

Let us look at the steps to write a basic AngularJS application.

Below is a simple HTML page:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
<body>
</body>
</html>
```

Now, we include angular.js in the head section of the HTML page. The path will be where you saved your file.

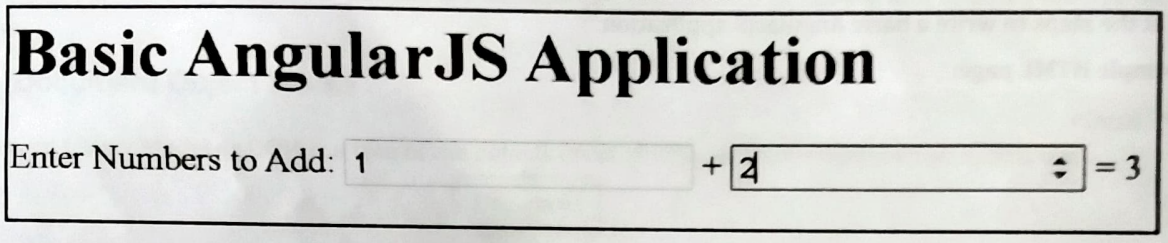
```
<!DOCTYPE html>
<html>
<head>
  <title>Basic AngularJS Application</title>
  <script src= "~/Scripts/angular.js"></script>
</head>
<body>
 </body>
</html>
```

Let's write a simple application in AngularJS to add 2 numbers:

```
<!DOCTYPE html>
<html>
<head>
  <title>Basic AngularJS Application</title>
  <script src="~/angularjsprogramfile.js"></script>
```

```
</head>
<body ng-app >
  <h1>Basic AngularJS Application</h1>
  Enter Numbers to Add:
  <input type="number" ng-model="x" /> + <input type="number" ng-model="y" />
  = <span>{{x + y}}</span>
</body>
</html>
```

This is what you see on your browser when this basic code block is executed:



The strange attributes (ng-app, ng-model) and curly braces {{ }} you see in the code are called Directives and Expressions. We will discuss them in detail all over the chapter. Here's a brief:

Directives are the attributes that tell AngularJS to attach the needed and defined behavior to the Document Object Model (DOM) element. Most Directives in AngularJS begin with ng. NG stands for Angular. We have used two of them in the code above: 1) ng-app and 2) ng-model.

The "ng-app" is the origin and it tells the Angular framework to initialize itself and get ready. The "ng-model" binds HTML to AngularJS and assigns the textbox values to the assigned variables x and y.

The curly braces {{ }} are the Expressions in AngularJS and as the name clearly states: they compute an expression to produce a result.

3.3 Need for AngularJS

What are the benefits to using AngularJS in the real world? How does it help improve our websites over basic HTML and CSS? Let's have a look:

1. Easy to Learn

For programmers with basic knowledge about HTML, CSS and JavaScript, AngularJS is very easy to learn. It opens further opportunities to be creative with next to no learning curve. It reduces the time to build new applications and makes them more interactive and appealing.

2. Two-Way Binding

The view is what the user sees on their browser and the model is what dictates the functionality behind the view. With AngularJS, any changes to the model are seamlessly pushed to the view with no added change and any change to the view percolates to the model in real time. This simplifies the application's view layer and makes changes to the UI less intrusive.

3. Supports Single Page Applications

Angular allows changes to the UI to be made without having to reload the entire page, making a website feel more like a native application. This is called a Single Page Application. The pages load quicker, the changes seem more fluid, the user experience is improved significantly, less data is needed for loading and the code is easier to maintain and debug.

4. Declarative UI

AngularJS provides many directives that are ready to use like ng-app, ng-model, ng-repeat etc (all of which will be studied in this chapter). This makes the UI easy to understand and allows designers and programmers to work together. In the declarative paradigm, the UI can be designed in any way necessary based on the data in it. How would a textbox look when empty, or filled with invalid or valid data? How would an error be shown? Any such changes can be seamlessly made in declarative UI without having to throw a popup or reloading the page.

5. Supported by Google

AngularJS gained immense credibility because it is used even by Google for its own pages. This means significant updates and new functionalities and ready code are often available for the most commonly required use cases. The massive community of developers is a huge bonus.

6. Optimal Web Application Management

Archaic web designers and programmers break down code into three parts: 1) Model, 2) View, and 3) Control. Then the code was merged manually. AngularJS does all of the merging automatically without separate code merges being needed. The model maintains the data, the view is the UI and the Control helps dictate the relationship between the model and the view.

7. Real-Time Testing

AngularJS allows end to end unit testing. It has features to oversee the components of the application and has testing features to test dependencies. No third party resources are needed for testing with AngularJS.

3.4 AngularJS Built-In Directives

Directives in AngularJS are an extension to HTML tags. They can be built-in or even custom created by programmers. We'll discuss the latter kind further into the chapter. Directives start with the ng "prefix". Let us start by discussing the most common built-in directives.

Note from Author : This chapter covers the entire syllabus, but to ensure better understanding, it does not follow the same flow as given in the official syllabus. The material has been moved around to ensure that a basic prerequisite understanding is developed in the reader as they move through this chapter.

Ng-app: This is the directive that begins an AngularJS application.

It defines the root and automatically initializes the application when a webpages using AngularJS is loaded. It begins loading all AngularJS modules and components.

```
<div ng-app = "">
  ...
</div>
```

Ng-init: This is the directive that initializes the AngularJS application data.

This directive initializes the AngularJS application data. It assigns values to the variables. The code below initializes the data within the array and assign the values to the countries array.

```
<div ng-app = "" ng-init = "countries = [{countrycode:'IN',countryname:'India'},
                                {countrycode:'SL',countryname:'Sri Lanka'},
                                {countrycode:'PK',countryname:Pakistan}]">
  ...
</div>
```

Ng-model: This is the directive that defines the model as a variable. The variable is then used in AngularJS.

It defines the model or variable to be used in the application.

```
<div ng-app = "">
  ...
  <p>Enter your ID: <input type = "text" ng-model = "id"></p>
</div>
```

Ng-repeat: This is the directive that repeats HTML elements in a collection once.

The "country in countries" below iterates over each element in the countries array initialized above.

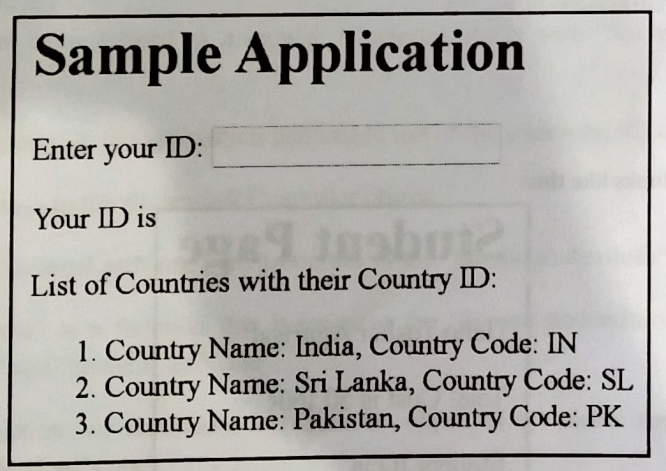
```
<div ng-app = "" ng-init = "countries = [{countrycode:'IN',countryname:'India'},
                                {countrycode:'SL',countryname:'Sri Lanka'},
                                {countrycode:'PK',countryname:Pakistan}]">
  ...
  <p>List of Countries with their Country Code:</p>
  <ol>
    <li ng-repeat = "country in countries">
      {{ 'Country Name: ' + country.countryname + ', Country Code: ' + country.countrycode }}
    </li>
  </ol>
</div>
```

Let us look at a piece of code that uses all these directives together:

```
<html>
  <head>
    <title>AngularJS Directives</title>
  </head>

  <body>
    <h1>Sample Application</h1>

    <div ng-app = "" ng-init = "countries = [{countrycode:'IN',countryname:'India'},
```

```
                    {countrycode:'SL',countryname:'Sri Lanka'},
                    {countrycode:'PK',countryname:'Pakistan'}]">
    <p>Enter your ID: <input type = "text" ng-model = "id"></p>
    <p>Your ID is <span ng-bind = "id"></span></p>
    <p>List of Countries with their Country ID:</p>
      <ol>
      <li ng-repeat = "country in countries">
       {{ 'Country Name: ' + country.countryname +
         ', Country Code: ' + country.countrycode }}
      </li>
     </ol>
   </div>

   <script src = "~/angulartestfile.js">
   </script>

  </body>
</html>
```

Output



As you type your ID in the textbox, the numbers will begin to appear after "Your ID is".

3.5 AngularJS Expressions

Expressions in AngularJS can bind data to HTML. They are written using double curly braces {{ }} with the actual expression inside them. These expressions output the final result of the expression within the braces.

Let us look at an example that computes an expression using numbers:

```
<p>Total Cost : {{unitCost * quantityOfItems}} INR</p>
```

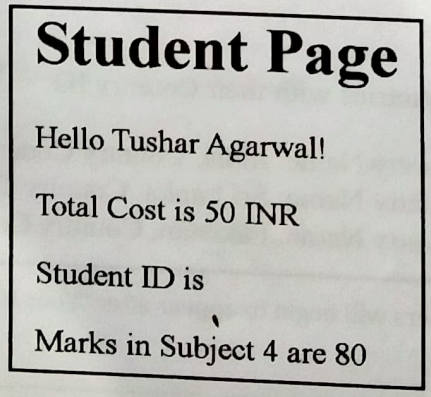
Expressions can use all data types. Let us look at another expression that computes something with strings:

```
<p>Greetings {{student.firstName + " " + student.lastName}}! Welcome to College!</p>
```

Let us look at an all-encompassing example that uses expressions for numbers, strings, arrays and even objects:

```
<html>
  <head>
    <title>Learning AngularJS Expressions</title>
  </head>
  <body>
    <h1>Student Page</h1>
    <div ng-app = "" ng-init = "quantityOfItems = 5;
                                costOfItem = 10;
                                studentInformation =
                                    {firstName:'Tushar',lastName:'Agarwal',studentID:123};
                                marks = [50,60,70,80,90]">
      <p>Hello {{studentInformation.firstName + " " + studentInformation.lastName}}!</p>
      <p>Total Cost is {{costOfItem * quantityOfItems}} INR</p>
      <p>Student ID is {{student.studentID}}</p>
      <p>Marks in Subject 4 are {{marks[3]}}</p>
    </div>
    <script src = "~/angularExpressions.js">
    </script>
  </body>
</html>
```

The output for this code block looks like this:



3.6 AngularJS Controllers

Controllers in AngularJS control the flow of data in a .js application.

It is defined using an "ng-controller" built-in directive. It is an object with properties and attributes and functions. It requires $scope as a parameter which tells it the application or module that it needs to take care of.

Let us look at an example and understand this better:

```
<script>
  function studentInfoController($scope) {
    $scope.studentInfo = {
      studentFirstName: "Tushar",
      studentLastName: "Agarwal",

      studentFullName: function() {
        var studentInfoObject;
        studentInfoObject= $scope.studentInfo;
        return studentInfoObject.studentFirstName + " " + studentInfoObject.studentLastName;
      }
    };
  }
</script>
```

Points to note from the code above

The studentInfoController has been defined as a regular JavaScript object with "$scope" being passed to it as a parameter..

The "$scope" parameter here points to the application that makes use of the studentInfoController object.

The "$scope.studentInfo" belongs to the studentInfoController object.

The "studentFirstName" and "studentLastName" are properties of the "$scope.studentInfo" object.

The property "studentFullName" is a function that belongs to the "$scope.studentInfo" object and it returns the combined "studentFirstName" and "studentLastName".

In the "studentFullName" function, we instantiate a studentInfo object as a variable and then return the combined "studentFirstName" and "studentLastName".

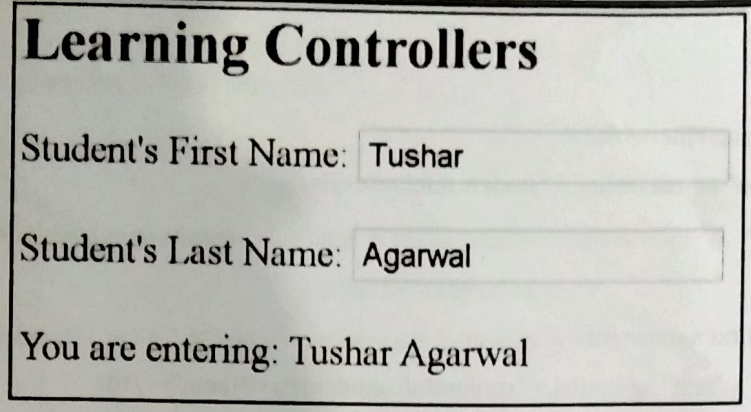
The studentInfoController's studentInfo property can be used with expressions now to get the combined names:

```
Enter first name: <input type = "text" ng-model = "student.firstName"><br>
Enter last name: <input type = "text" ng-model = "student.lastName"><br>
<br>
You are entering: {{student.fullName()}}
```

Let us now look at a full-fledged example with the output

```
<html>
  <head>
  <title>Learning AngularJS Controllers</title>
  <script src = "~/angularControllers.js"></script>
 </head>
 <body>
  <h2>Learning Controllers</h2>
  <div ng-app = "myApp" ng-controller = "studentInfoController">
   Student's First Name: <input type = "text" ng-model = "studentInfo.studentFirstName"> <br><br>
   Student's Last Name: <input type = "text" ng-model = "studentInfo.studentLastName"> <br><br>
   You are entering: {{studentInfo.studentFullName()}}
  </div>
  <script>
   var myApp = angular.module( "myApp", [] );
   myApp.controller('studentInfoController', function($scope) {
    $scope.studentInfo = {
     studentFirstName: "Tushar",
     studentLastName: "Agarwal",
     studentFullName: function() {
      var studentInfoObject;
      studentInfoObject = $scope.studentInfo;
      return studentInfoObject.studentFirstName + " " +
           studentInfoObject.studentLastName;
     }
    };
   });
  </script>
 </body>
</html>
```

The output for this program is as shown below. The output changes dynamically if you change the input in the textboxes.



3.7 AngularJS Filters

Filters are made use of in AngularJS to modify and twist data as needed. They are usually clubbed with the AngularJS expressions or directives using a pipe (|) symbol.

Uppercase Filter

As the name suggests, the uppercase filter prints text in uppercase. Let us take just a few lines from the code explained in the last section to show the minor change.

```
Enter your first name:<input type = "text" ng-model = "studentInfo.studentFirstName">
Enter your last name: <input type = "text" ng-model = "studentInfo.studentLastName">
Your name in Uppercase: {{studentInfo.studentFullName() | uppercase}}
```

Lowercase Filter

As the name suggests, the lowercase filter prints text in lowercase. Let us take just a few lines from the code explained in the last section to show the minor change.

```
Enter your first name:<input type = "text" ng-model = "studentInfo.studentFirstName">
Enter your last name: <input type = "text" ng-model = "studentInfo.studentLastName">
Your name in Lowercase: {{studentInfo.studentFullName() | lowercase}}
```

Currency Filter

As the name suggests, the currency filter prints numbers in the format of currency. Let us create a few lines of code similar to the code explained in the last section to show the changes necessary.

```
Enter the student's fees: <input type = "text" ng-model = "studentInfo.studentFees">
studentFees: {{studentInfo.studentFees | currency}}
```

You can even use filters over text to only show what you type in a textbox or even order items by value in ascending order. Let us look at an all-encompassing example that uses all these filters with its output:

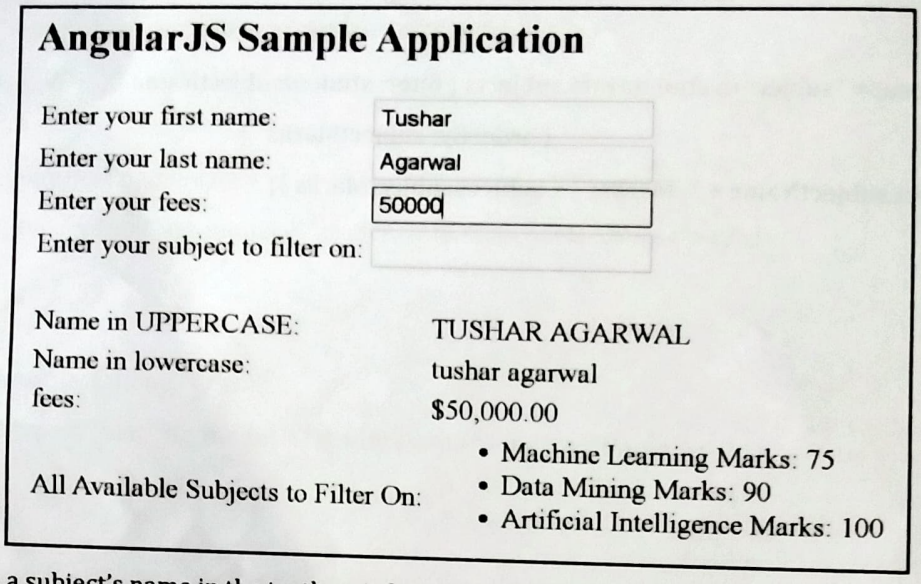
```
<html>
  <head>
    <title>Angular JS Filters</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
    </script>
```

```
</head>
<body>
 <h2>AngularJS Learning Filters</h2>
 <div ng-app = "myApp" ng-controller = "studentInfoController">
  <table border = "0">
   <tr>
    <td>Enter your first name:</td>
    <td><input type = "text" ng-model = "studentInfo.studentFirstName"></td>
   </tr>
   <tr>
    <td>Enter your last name: </td>
    <td><input type = "text" ng-model = "studentInfo.studentLastName"></td>
   </tr>
   <tr>
    <td>Enter your fees: </td>
    <td><input type = "text" ng-model = "studentInfo.fees"></td>
   </tr>
   <tr>
    <td>Enter your subject to filter on: </td>
    <td><input type = "text" ng-model = "studentsubjectName"></td>
   </tr>
  </table>
  <br/>
  <table border = "0">
   <tr>
    <td>Name in UPPERCASE: </td><td>{{studentInfo.studentFullName() | uppercase}}</td>
   </tr>
   <tr>
    <td>Name in lowercase: </td><td>{{studentInfo.studentFullName() | lowercase}}</td>
   </tr>
   <tr>
    <td>fees: </td><td>{{studentInfo.fees | currency}}
    </td>
   </tr>
```

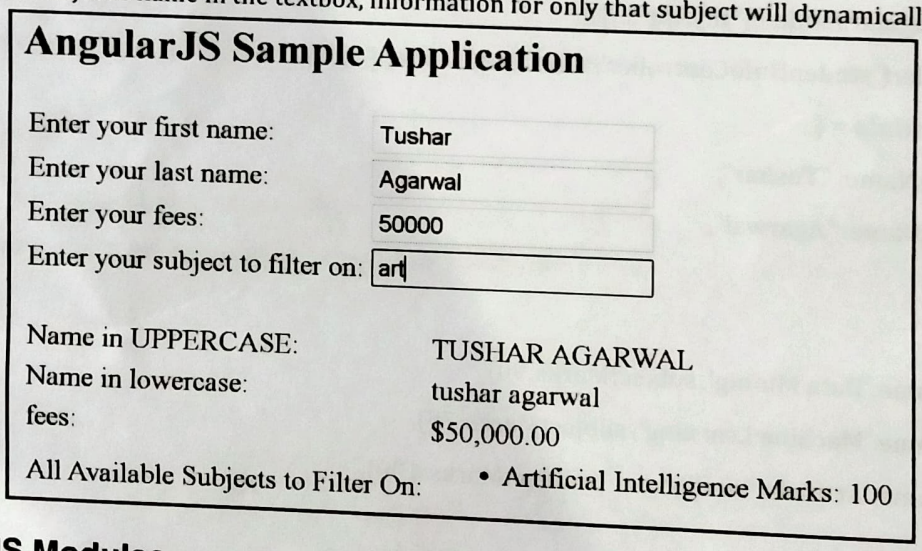
```
    <tr>
     <td>All Available Subjects to Filter On:</td>
     <td>
      <ul>
       <li ng-repeat = "subject in studentInfo.subjects | filter: studentsubjectName
                                                    | orderBy:'subjectMarks'">
        {{ subject.subjectName + '   Marks:  ' + subject.subjectMarks }}
       </li>
      </ul>
     </td>
    </tr>
   </table>
</div>
<script>
  var myApp = angular.module("myApp", []);
  myApp.controller('studentInfoController', function($scope) {
    $scope.studentInfo = {
     studentFirstName: "Tushar",
     studentLastName: "Agarwal",
     fees:500,
     subjects:[
      {subjectName:'Data Mining', subjectMarks:90},
      {subjectName:'Machine Learning', subjectMarks:75},
      {subjectName:'Artificial Intelligence', subjectMarks:100}
     ],
     studentFullName: function() {
      var studentInfoObject;
      studentInfoObject = $scope.studentInfo;
      return studentInfoObject.studentFirstName + " " +
            studentInfoObject.studentLastName;
     }
    };
  });
</script>
```

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

Output 1 : If you type your fees, it dynamically shows up in the format of currency ($)



Output 2 : If you type a subject's name in the textbox, information for only that subject will dynamically show up below:



3.8 AngularJS Modules

Modules in AngularJS are just containers for code within the application. Modules in AngularJS are created with the AngularJS function "angular.module".

Why do we want to use AngularJS Modules (advantages)?

1. Modules package the source program into smaller chunks of reusable modules.
2. The programmer can load modules in any order.
3. Modules make testing and debugging easier.
4. Modules help organize the application so a programmer can work on one module without having to know the inner workings of the entire application.

We will begin by understanding how to add Controllers to a Module. Let us have a detailed look at how it works:

```
<!DOCTYPE html>
<html>
<script src="~/angularModules.js"></script>
<body>
<div ng-app="anyName" ng-controller="controllerName">
{{ studentFirstName + " " + studentLastName }}
</div>
<script>
var app = angular.module("anyName", []);
app.controller("controllerName", function($scope) {
  $scope.studentFirstName = "Tushar";
  $scope.studentLastName = "Agarwal";
});
</script>
</body>
</html>
```

Output

Tushar Agarwal

In the above code, we added a controller to a module. We can add a built-in directive to modules as well.

```
<!DOCTYPE html>
<html>
<script src="~.LearningDirectives.js"></script>
<body>
<div ng-app="randomApp" learning-directives></div>
<script>
var app = angular.module("randomApp", []);
app.directive("learningDirectives", function() {
  return {
    template : "Hello, Tushar Agarwal!"
  };
});
</script>
</body>
</html>
```

Output

Hello, Tushar Agarwal!

When you have to work on larger files, creating one file to add the modules and another file to add the controllers helps keep the complexity down and increases the modularity of the application.

Let us see how that works:

Step 1 : Create a file with the module. Let's call this file "**moduleFile.js**"

```
var application = angular.module("learningModulesAngularJSApp", []);
```

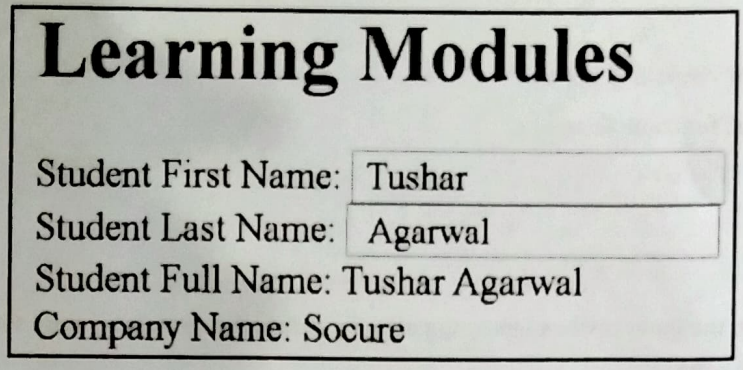
Step 2 : Create a file with the controllers. Let's call this file "**controllersFile.js**"

```
application .controller("companyNameController", function($scope) {
    $scope.studentFirstName = "Tushar",
    $scope.studentLastName = "Agarwal",
    $scope.companyName = "Socure"
} );
```

Step 3 : Include the two in your primary AngularJS application.

```
<!DOCTYPE html>
<html ng-app="learningModulesAngularJSApp">
 <head>
<title> Learning Modules </title>
<script src="~LearningModules.js"></script>
</head>
 <body>
 <div ng-controller="companyNameController">
   Student First Name: <input type="text" ng-model="studentFirstName"><br>
   Student Last Name: <input type="text" ng-model="studentLastName"><br>
   <br>
   Student Full Name: {{studentFirstName + " " + studentLastName}}
   <br>
   Company Name : {{ companyName }}
 </div>
 <script src="moduleFile.js"></script>
<script src="controllersFile.js"></script>
 </body>
</html>
```

Output



3.9 AngularJS Data Binding

We have been using data binding above without you knowing it. Data binding in AngularJS basically binds the Model and the View.

Model is the collection of all the data available for use as shown below:

```
var application1 = angular.module(randomApp, [ ]);
app.controller(controlName, function($scope) {
 $scope.studentFirstName = "Tushar";
 $scope.studentLastName = "Agarwal";
});
```

View is the HTML tags that create the UI you view. There are various ways to access the **Model** from the **View**:

Double braces {{ }} can be used in the View to access the Model:

```
<p>First name: {{firstname}}</p>
```

The keyword "ng-bind" can be used to achieve the same purpose. It binds the HTML element to the Model property:

```
<p ng-bind="studentFirstName"></p>
```

We can even use the "ng-model" keyword to create two-way data binding. What is it? Two-way binding ensures that any changes to the Model or the View are reflected in the other simultaneously and automatically, making sure they remain updated and in sync at all times. This creates the illusion of updates in real time.

```
<input ng-model="studentFirstName">
```

Let us see a full example of two-way data binding using controllers:

```
<!DOCTYPE html>
<html>
<script src="~LearningTwoWayBinding.js"></script>
<body>
<div ng-app="randomApp" ng-controller="controller">
 Name: <input ng-model="textboxName">
 <h1>{{textboxName}}</h1>
```

```
</div>
<script>
var app = angular.module('randomApp', [ ]);
app.controller('controller', function($scope) {
  $scope.textboxName = "Tushar";
});
</script>
<p>Changing the name in the input textbox will automatically update the model and thus, the UI will change without having to reload the page.</p>
</body>
</html>
```

Output

Name: Tushar

Tushar

Changing the name in the input textbox will automatically update the model and thus, the UI will change without having to reload the page.

3.10 AngularJS Custom Directives

AngularJS supports not only the built-in Directives already discussed, but also allows programmers to create their own custom Directives. They are created using the ".directive" function.

When Directives are named, we use the camelcase (thisIsCamelCase), but when they are invoked, we separate the name in lowercase with dashes (this-is-camel-case).

Example

```
<!DOCTYPE html>
<html>
<script src="~LearningCustomDirectives.js"></script>
<body ng-app="randomApp">
<this-is-a-custom-directive></this-is-a-custom-directive>
<script>
var app = angular.module("randomApp", [ ]);
app.directive("thisIsACustomDirective", function() {
  return {
    template : "<h2>Hello, Tushar Agarwal!</h2>"
```

```
  };
});
</script>
</body>
</html>
```

Output

Hello, Tushar Agarwal!

Custom Directives can be invoked using any of the following four ways. Let us rewrite the same functionality shown above to see how all four will produce the same result:

1. Element Name

```
<!DOCTYPE html>
<html>
<script src="~CustomDirectives.js"></script>
<body ng-app="randomApp">
<this-is-a-custom-directive>
</this-is-a-custom-directive>
<script>
var app = angular.module("randomApp", [ ]);
app.directive("thisIsACustomDirective", function() {
  return {
    template : "<h2>Hello, Tushar Agarwal!</h2>"
  };
});
</script>
</body>
</html>
```

2. Attribute

```
<!DOCTYPE html>
<html>
<script src="~CustomDirectives.js"></script>
<body ng-app="randomApp">
<div this-is-a-custom-directive>
```

```
</div>
<script>
var app = angular.module("randomApp", [ ]);
app.directive("thisIsACustomDirective", function() {
  return {
    template : "<h2>Helo, Tushar Agarwal!</h2>"
  };
});
</script>

</body>
</html>
```

3. Class

```
<!DOCTYPE html>
<html>
<script src="~CustomDirectives.js"></script>
<body ng-app="randomApp">
<div class="this-is-a-custom-directive">
</div>
<script>
var app = angular.module("randomApp", [ ]);
app.directive("thisIsACustomDirective", function() {
  return {
    restrict : "C",
    template : "<h2>Hello, Tushar Agarwal!</h2>"
  };
});
</script>
<! "C" must be added to the restrict property to allow invocation of the directive from a class name>
</body>
</html>
```

4. Comment

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

```
<script src="~CustomDirectives.js"></script>
<body ng-app="randomApp">
<!-- directive: this-is-a-custom-directive -->
<script>
var app = angular.module("randomApp", []);
app.directive("thisIsACustomDirective", function() {
  return {
    restrict : "M",
    template : "<h2>Hello, Tushar Agarwal!</h2>"
  };
});
</script>
<!"M" must be added to the restrict property to allow invocation of the directive from a comment>
</body>
</html>
```

Output for All 4 Ways

Hello, Tushar Agarwal!

3.11 AngularJS Scope

Scope in AngularJS contains the application's methods and the data used by those methods. It is a built-in object and properties for scope can be created inside a controller function and values can be assigned to it.

It is denoted as "$scope) and is the intermediary between the Controller and the View. It is responsible for data transfer between the Controller and the View.

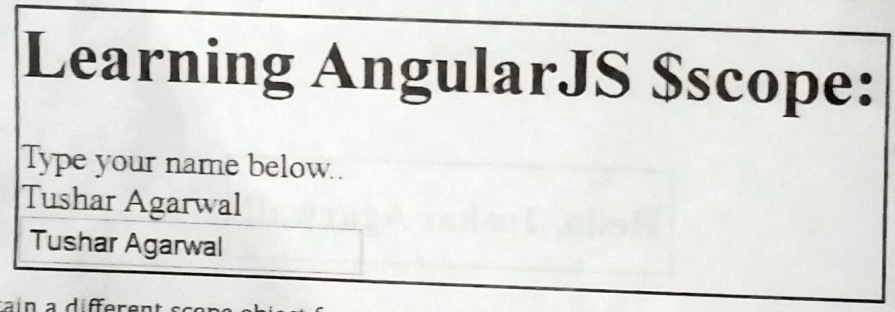
Properties can be attached to the scope object inside the controller function as mentioned already and the view can then display the scope's data using ng model or bind directives that have already been explained above.

Let us see how this works:

```
<!DOCTYPE html>
<html>
<head>
  <script src="~/LearningScope.js"></script>
</head>
<body ng-app="randomApp">
<h1>AngularJS $scope Demo: </h1>
  <div ng-controller="theControllerFunction">
```

```
    {{message}}<br />
    <span ng-bind="printStatement"></span> <br />
    <input type="text" ng-model="printStatement" />
  </div>
  <script>
    var ngApp = angular.module('randomApp', []);
    ngApp.controller('theControllerFunction', function ($scope) {
      $scope.message = "Type your name below..";
    });
  </script>
</body>
</html>
```

Output



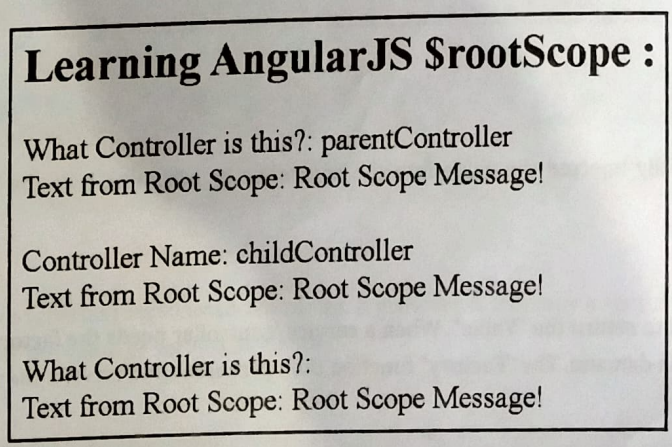
AngularJS must maintain a different scope object for every controller you have in your application. Just like classes in Java, the data and methods attached to the scope in one controller cannot be accessed in another controller. If the controller is nested, the child controller will have access to the parent controller's methods and data, but not vice-versa.

There is also the concept of the $rootScope object in AngularJS. An AngularJS application can have only one root scope. All other scope objects are child objects for it and will be able to access the root scope's properties and methods.

```
<!DOCTYPE html>
<html>
<head>
  <script src="~/LearningRootScope.js"></script>
</head>
<body ng-app="randomApplication">
<h2>Learning AngularJS $rootScope : </h2>
  <div ng-controller="parentController">
    What Controller is this?: {{nameOfController}} <br />
    Text from Root Scope: {{text}} <br /><br />
    <div ng-controller="childController">
```

```
      Controller Name: {{nameOfController}} <br />
      Text from Root Scope: {{text}} <br /><br />
    </div>
  </div>
  <div ng-controller="cousinController">
    What Controller is this?: {{nameOfController}} <br />
    Text from Root Scope: {{text}} <br />
  </div>
  <script>
    var ngApp = angular.module('randomApplication', [ ]);
    ngApp.controller('parentController', function ($scope, $rootScope) {
      $scope.nameOfController = "parentController";
      $rootScope.text = "Root Scope Message!";
    });
    ngApp.controller('childController', function ($scope) {
      $scope.nameOfController = "childController";
    });
    ngApp.controller('cousinController', function ($scope) {
    });
  </script>
</body>
</html>
```

Output



You'll see in the example above that data (text message) from the root scope is available to all other scopes for use.

3.12 AngularJS Dependency Injection

Dependency Injection is a built-in functionality for AngularJS which allows you to divide the application you're creating into multiple kinds of components that can be injected into each other as "dependencies".

Dependency Injection modularizes your application and increases reusability, configurability and makes testing easier.

Let us understand the basic objects and components you will need to know before we look at an example of them:

1. value
2. factory
3. service
4. provider
5. constant

3.12.1 Value

Value is a simple object that can be a string or a numeric character or even an object. Its primary use is to pass values in factories, services or controllers.

```
//define a module in AngularJS
var randomModule = angular.module("randomModule", [ ]);
//create a "Value" object in AngularJS and passing data to it
randomModule.value("numericValue", 1994);
randomModule.value("stringValue", "Tushar Agarwal");
randomModule.value("objectValue", { key1: 1994, key2: "Tushar Agarwal"} );
```

Values are defined with the function "value". This function has two parameters: 1) the name of the value; and 2) the value.

These values can now be invoked using the name assigned to them. To "inject" a value, do the following:

```
var randomModule = angular.module("randomModule", [ ]);
randomModule .value("numericValue", 1994);
randomModule.controller("randomController", function($scope, numericValue) {
console.log(numericValue);
});
```

The above code has successfully injected the value into the controller by assigning it to the function when invoking the controller.

3.12.2 Factory

Factory is just a function used to return the "Value". When a service/controller needs the factory to inject a Value, a Value is created right then and there on demand. The "Factory" function then processes and returns the value.

Let us look at an example.

```
var randomModule = angular.module("randomModule", []);
randomModule .factory("randomFactory", function()
```

```
{
  return "My name is Tushar Agarwal! This is a Value!";
});
randomModule .controller("randomController", function($scope, randomFactory)
{
  console.log(randomFactory);
});
```

In the code above, you see a factory created first over a module and then the value created by the factory is injected using a controller.

3.12.3 Service

In AngularJS, service is a JavaScript object which contains a set of functions to perform certain tasks. Services are created by using the service() function on a module and then injected into controllers.

```
//define an AngularJS module
var myApplication= angular.module("myApplication", [ ]);
... rest of the program ... this is just a section of the code ...
//create an AngularJS service to create a method that returns the square of a numeric input.
myApplication.service('CalculatationService', function(MathematicsService){
  this.square = function(x) {
    return MathematicsService.multiply(x,x);
  }
});
// inject "CalculatationService" into AngularJS controller
myApplication.controller('CalculationController', function($scope CalculatationService, defaultInputNumber) {
  $scope.number = defaultInputNumber;
  $scope.answer = CalculatationService.square($scope.number);
   $scope.square = function() {
    $scope.answer = CalculatationService.square($scope.number);
  }
});
```

3.12.4 Provider

A "provider" in AngularJS provides (creates) services and factories. A provider is basically a very simple factory method that has a "get ( )" function that returns the value or service or factory.

```
//define an AngularJS module
var myApplication= angular.module("myApplication", [ ]);
... rest of the program ... this is just a section of the code ...
```

```
//create an AngularJS service using a "provider" that defines the method "divide"to return the division of two numbers.
myApplication.config(function($provide) {
 $provide.provider('MathematicsService', function() {
  this.$get = function() {
   var factory = { };
   factory.divide = function(x, y) {
    return x / y;
   }
   return factory;
  };
 });
});
```

3.12.5 Constants

Values cannot be injected into the module.config ( ) function. In such a case, during the config phase, constants are used.

```
myApplication.constant("configParameter", "constantValue");
```

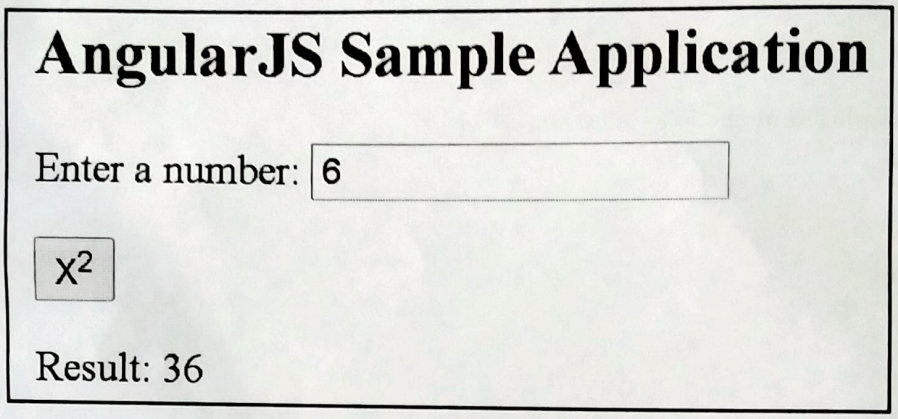
We will understand this better in the next example where all the above mentioned directives will be used in one application:

```
<!DOCTYPE html>
<html>
 <head>
  <title>AngularJS Dependency Injection</title>
 </head>
 <body>
  <h2>AngularJS Sample Application</h2>

  <div ng-app = "randomApplication" ng-controller = "CalculationsController">
   <p>Enter a number: <input type = "number" ng-model = "number" /></p>
   <button ng-click = "squareFunction()">X<sup>2</sup></button>
   <p>Result: {{result}}</p>
  </div>

  <script src = "~/LearningDependencyInjections.js"></script>
```

```
<script>
 var randomApplication= angular.module("randomApplication", []);

 randomApplication.config(function($provide) {
  $provide.provider('MathematicsService', function() {
   this.$get = function() {
    var factory = {};

    factory.multiplicationFunction = function(a, b) {
     return a * b;
    }
    return factory;
   };
  });
 });
 randomApplication.value("defaultInput", 10);
 randomApplication.factory('MathematicsService', function() {
  var factory = {};
  factory.multiplicationFunction = function(a, b) {
   return a * b;
  }
  return factory;
 });
 randomApplication.service('CalculationsService', function(MathematicsService){
  this.squareFunction = function(a) {
   return MathematicsService.multiplicationFunction(a,a);
  }
 });
randomApplication.controller('CalculationsController', function($scope, CalculationsService, defaultInputVariable)
{
  $scope.number = defaultInputVariable;
  $scope.result = CalculationsService.squareFunction($scope.number);
   $scope.squareFunction = function() {
   $scope.result = CalculationsService.squareFunction($scope.number);
```

```
    }
  });
  </script>
 </body>
</html>
```

Output



3.13 AngularJS Services

Services in AngularJS are functions or objects that are available to your AngularJS application. AngularJS has many built-in services but also allows you to create your own customized service.

We will look at a few built-in services first and then build a custom service to help you better understand the concept.

3.13.1 The $location Service

The service "$location" has built-in methods that return information about the location of the current web page.

```
<!DOCTYPE html>
<html>
<script src="~LearningLocationService.js"></script>
<body>
<div ng-app="randomApp" ng-controller="randomController">
<p>The URL for this web page is as follows:</p>
<h4>{{myUrl}}</h4>
</div>
<p>Here, we used the built-in $location service to get the URL for this web page.</p>
<script>
var application = angular.module('randomApp', []);
application.controller('randomController', function($scope, $location) {
```

```
  $scope.myUrl = $location.absUrl();
});
</script>
</body>
</html>
```

Output

The URL for this web page is as follows:

https://www.w3schools.com/angular/tryit.asp?filename=try_ng_services

Here, we used the built-in $location service to get the URL for this web page.

3.13.2 The $http Service

The $http is a very simple service that simply creates a request for the server and lets the application you created handle the response.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="randomApp" ng-controller="randomController">
<h3>{{myVariable}}</h3>
</div>
<p>$http service has created a request for a page to the tusharagarwal.htm server and the response sent back is the value of the variable "myVariable" stored over there which is "Hello AngularJS Students".</p>
<script>
var myApplication = angular.module('randomApp', []);
myApplication.controller('randomController', function($scope, $http) {
 $http.get("tusharagarwal.htm").then(function (result) {
    $scope.myVariable = result.data;
 });
});
</script>
</body>
</html>
```

Output

Hello AngularJS Students

$http service has created a request for a page and the response sent back is the value of the variable "myVariable".

3.13.3 The $timeout Service

This is a built-in service that just after passage of the timeout period in milliseconds, invoked a function specified under timeout.

```
<!DOCTYPE html>
<html>
<script src="~/LearningTimeouts.js"></script>
<body>
<div ng-app="randomApp" ng-controller="randomController">
<p>The header will automatically change after 5 seconds</p>
<h1>{{myHeader}}</h1>
</div>
<p>The $timeout service will invoke a function after the timeout period has passed.</p>
<script>
var myApplication = angular.module('randomApp', []);
myApplication.controller('randomController', function($scope, $timeout) {
 $scope.myHeader = "Message Before Timeout";
 $timeout(function () {
    $scope.myHeader = "Message After Timeout";
 }, 5000);
});
</script>
</body>
</html>
```

Output Before Timeout

The header will automatically change after 5 seconds

Message Before Timeout

The $timeout service will invoke a function after the timeout period has passed.

Output After Timeout

The header will automatically change after 5 seconds

Message After Timeout

The $timeout service will invoke a function after the timeout period has passed.

3.13.4 The $internal Service

The $internal service invokes the function specified under it every specified number of milliseconds.

Let us look at an example

```
<!DOCTYPE html>
<html>
<script src="~/LearningInternalService.js"></script>
<body>
<div ng-app="randomApp" ng-controller="randomController">
<p>The current time is:</p>
<h1>{{currentTime}}</h1>
</div>
<p>$interval service updates the current time for every one second. This gives the illusion of an actual clock while in reality, the time is simply updated every one second using $internal service.</p>
<script>
var app = angular.module('randomApp', []);
app.controller('randomController', function($scope, $interval) {
 $scope.currentTime = new Date().toLocaleTimeString();
 $interval(function () {
    $scope.currentTime = new Date().toLocaleTimeString();
 }, 1000);
});
</script>
</body>
</html>
```

Output

The current time is:

1:18:04 PM

$interval service updates the current time everyone second. This gives the illusion of an actual clock while in reality, the time is simply updated every one second using $internal service.

3.13.5 Custom Services

To create and use your custom service, first create a service and connect it to the module. Ater that, add it as a dependency when you define the controller.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="randomApplication" ng-controller="myController">
<p>The USD value of 15,000 INR is:</p>
<h2>{{dollarAount}}</h2>
</div>
<p>We have defined a custom service that coverts INR to USD.</p>
<script>
var myApplication = angular.module('randomApplication', []);
myApplication.service('dollarConvert', function() {
  this.customFunction = function (x) {
    return x / 75;
  }
});
app.controller('myController', function($scope, dollarConvert) {
 $scope.dollarAount = dollarConvert.customFunction(15000);
});
</script>
</body>
</html>
```

Output

The USD value of 15,000 INR is:

200

We have defined a custom service that coverts INR to USD.

3.14 AngularJS Form Validation

Form Validation ensures if the input entered in the form components used in your applications conforms with the input expected by the programmer.

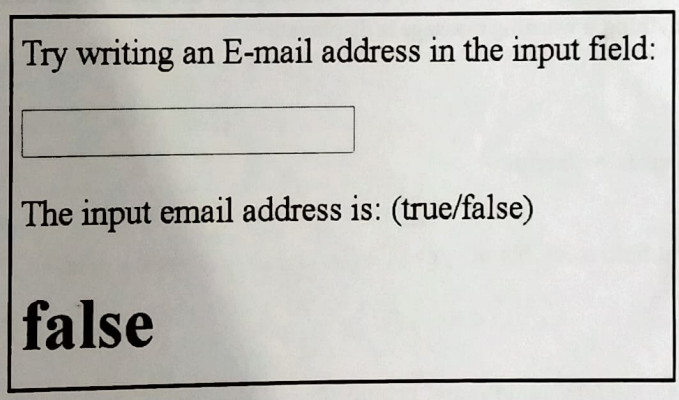
AngularJS can help monitor the state of the form and the input fields and then can notify the user if something is wrong. The state of the form refers to whether the form has been touched/clicked or not, whether it has been modified and whether the input information matches expectations.

Standards attributes exist built-in for form validation and custom validation can also be performed.

Let us learn two such form validations together. In the next example, the built-in validations will check if an email is valid and will throw a default value of false because the "required" validation is also added to it. Let us look at the code first and then understand them better.

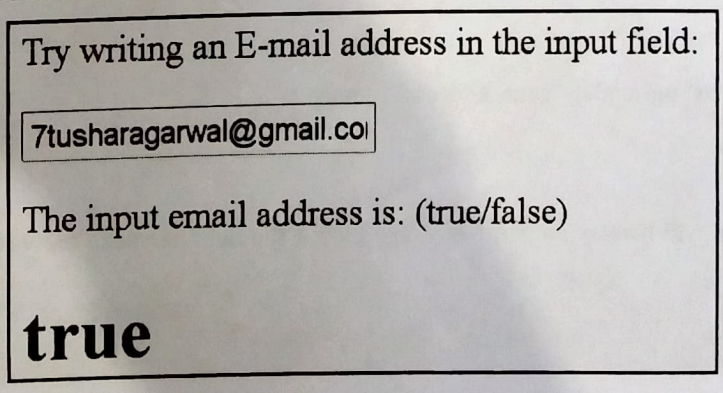
```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body ng-app="">
<p>Try writing an E-mail address in the input field:</p>
<form name="learningForms">
<input type="email" name="someInput" ng-model="someInput" required>
</form>
<p>The input email address is: (true/false)</p>
<h1>{{learningForms.someInput.$valid}}</h1>
</body>
</html>
```

Output without entering any text

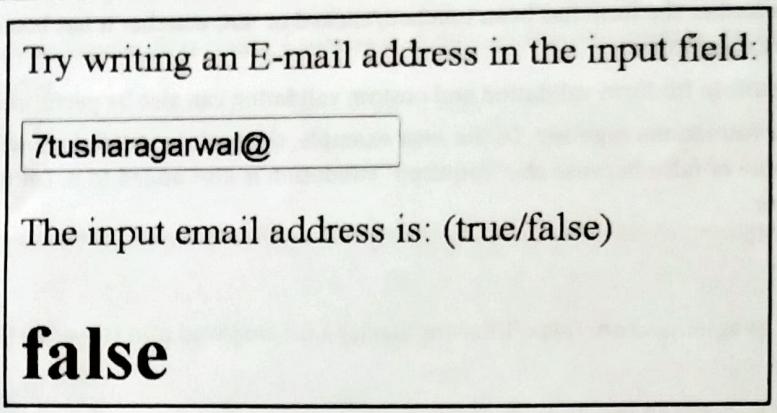


This is because of the "required" tag used for form validation. If the "required" tag was removed, the default answer above would be "true".

Output after you enter a valid email address



Output after you enter an invalid email address



We have used the state $valid in the example above. Input fields can possess any of the states shown below:

- **$untouched** - The form field hasn't been touched by the cursor yet
- **$touched** - The form field has been touched by the cursor already
- **$pristine** - The form field has not been modified/updated/written on yet
- **$dirty** - The form field has been modified/updated/written on already
- **$invalid** - The form field's content entered by the user is invalid
- **$valid** - The form field's content entered by the user is valid

Form states are used to pass on meaningful information/error messages to the user. In the code below, if a field if required and the user hasn't entered anything, a warning message is displayed:

```
<!DOCTYPE html>
<html>
<script src="~/LearningForms.js"></script>
<body ng-app="">
<p>Error when the first input field is left blank:</p>
<form name="learningForm">
<p>Name:
<input name="someName" ng-model="someName" required>
<span ng-show="learningForm.someName.$touched && learningForm.someName.$invalid">The name is a required field in this form.</span>
</p>
<p>Address:
<input name="someAddress" ng-model="someAddress" required>
</p>
</form>
<p>The ng-show directive will display an error message only if the field has been touched by the cursor and it is empty.</p>
</body>
</html>
```

Output

You will see that the error appears as soon as the cursor touches the name filed and leaves. This same error does not appear for the address.

Error when the first input field is left blank:

Name: The name is a required field in this form.

Address:

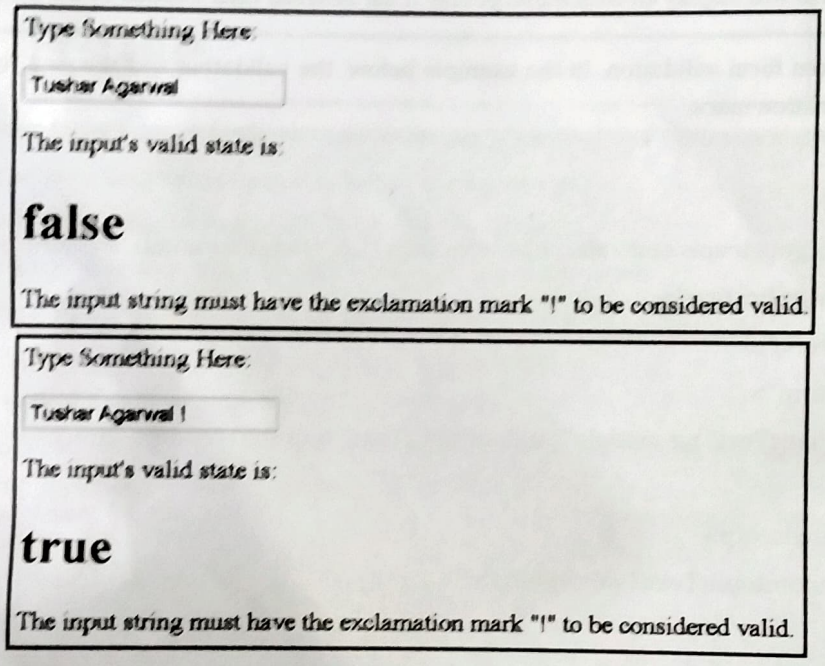
The ng-show directive will display an error message only if the field has been touched by the cursor and it is empty.

Let us now look at a custom form validation. In the example below, the validation will throw a "true" message only if the entered text has an exclamation mark.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body ng-app="randomApplication">
<p>Type Something Here:</p>
<form name="randomForm">
<input name="randomInputText" ng-model="randomInputText" required random-directive>
</form>
<p>The input's valid state is:</p>
<h1>{{randomForm.randomInputText.$valid}}</h1>
<script>
var randomApplication = angular.module('randomApplication', []);
randomApplication.directive('randomDirective', function() {
  return {
    require: 'ngModel',
    link: function(scope, element, attr, mCtrl) {
      function myValidation(value) {
        if (value.indexOf("!") > -1) {
          mCtrl.$setValidity('charExclamation', true);
        } else {
          mCtrl.$setValidity('charExclamation', false);
        }
        return value;
      }
      mCtrl.$parsers.push(myValidation);
    }
```

```
  };
});
</script>
<p>The input string must have the exclamation mark "!" to be considered valid.</p>
</body>
</html>
```

Output



3.15 AngularJS Routing using ng-route

AngularJS has a functionality to navigate to different pages in your application package without having to reload the page. This is called a Single Page Application. This can be achieved using the "ng-route" module. This module routes your application during runtime itself to different pages.

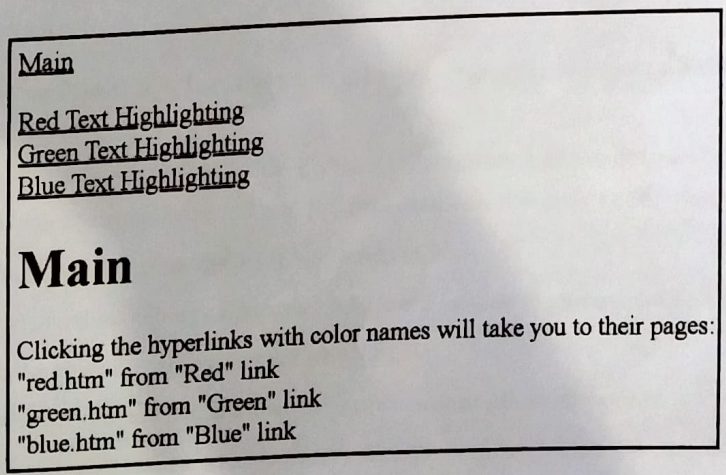
To incorporate and allow routing, we must include the AngularJS Route module in our code (second script).

After that, we add ng-route as a dependency. The $routeprovider now configured different routes within the application.

```
<!DOCTYPE html>
<html>
<script src="~/angular.min.js"></script>
<script src="~/angular-route.js"></script>
<body ng-app="randomApplication">
<p><a href="#/!">Main</a></p>
<a href="#!red">Red Text Highlighting </br></a>
<a href="#!green">Green Text Highlighting </br></a>
```

```
<a href="#!blue">Blue Text Highlighting </br></a>
<div ng-view></div>
<script>
var app = angular.module("randomApplication", ["ngRoute"]);
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.htm"
  })
  .when("/red", {
    templateUrl : "red.htm"
  })
  .when("/green", {
    templateUrl : "green.htm"
  })
  .when("/blue", {
    templateUrl : "blue.htm"
  });
});
</script>
<p>Clicking the hyperlinks with color names will take you to their pages:</br>
"red.htm" from "Red" link</br>
"green.htm" from "Green" link</br>
"blue.htm" from "Blue" link</p>
</body>
</html>
```

Default Output



Output when Red/Blue/Green Text Highlighting is Clicked (the text is highlighted in the color mentioned):

Main

Red Text Highlighting
Green Text Highlighting
Blue Text Highlighting

Red

Clicking the hyperlinks with color names will take you to their pages:
"red.htm" from "Red" link
"green.htm" from "Green" link
"blue.htm" from "Blue" link

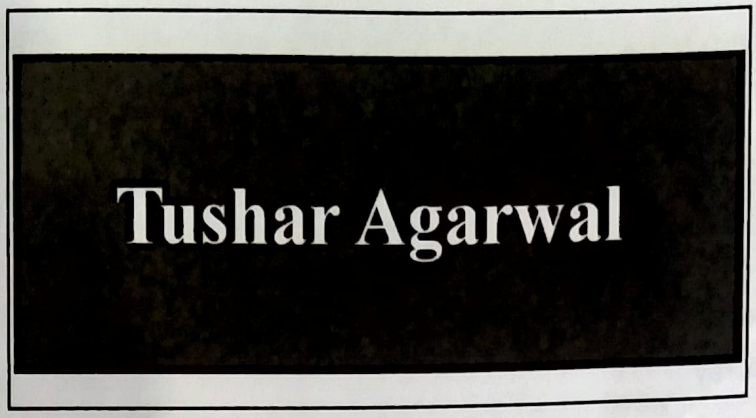
3.16 The ng-style Directive

The Directive "ng-style" is a style attribute for the HTML elements in your page and they are basically an object or an expression that returns an object.

The object contains the style or speaking in terms of web programming, the CSS properties and values that dictate the style of the web page (front end of your application that the user interacts with). The CSS properties and values are all key value pairs.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body ng-app="randomApplication" ng-controller="myController">
<h1 ng-style="styleObject">Tushar Agarwal</h1>
<script>
var randomApplication = angular.module("randomApplication", []);
randomApplication.controller("myController", function($scope) {
 $scope.styleObject = {
  "background-color" : "black",
  "color" : "white",
  "padding" : "100px",
  "font-size" : "75px"
 }
});
</script>
</body>
</html>
```

Output



3.17 AngularJS Built-In Helper Functions

AngularJS, like other scripting languages, also has a suite of built-in functions that are exceedingly useful little shortcuts to achieve the most common functionalities needed by most developers.

We will look at many examples of such built-in functions:

3.17.1 angular.copy

The copy function in AngularJS takes any object, or array or variable and creates a deep copy of it. Deep copy means that the new copy is assigned its own space in memory and changing the copy will not update the original. In a soft copy, updates to the copy will also update the original because they share the same memory block.

Example

```
let object1 = {name: "Tushar", occupation: "Product Manager"};
let x= angular.copy(object1);
x.name = "copyOfObject1"
// object1 = {name: "Tushar", occupation: "Product Manager"}
// copyOfObject1= {name: "Tushar", occupation: "Product Manager"}
```

3.17.2 angular.equals

This function compares two objects or values and performs a deep comparison of the two and returns a true if atleast one of the following conditions is mat:

The angular.equals function [angular.equals(value1, value2)] compares and determines if 2 objects or values are equal, angular.equals performs a deep comparison and returns true if and only if at least 1 of the following conditions is met.

1. The objects or values must clear the === (deep comparison).
2. Both the objects or values belong to the same data type and their properties are also equal.
3. Values of both the objects are programmatically NaN.
4. Values of both the objects are equal to the result of the same regular expression.

Examples

```
angular.equals(1994, 1994) // true
angular.equals(1994, 1995) // false
angular.equals({ }, { }) // true
angular.equals({x: 1}, {x: 1}) // true
angular.equals({x: 1}, {x: 2}) // false
angular.equals(NaN, NaN) // true
```

3.17.3 angular.isArray

This function returns a boolean True if and only if the object that is passed to it as a parameter is an array. It is written as: angular.isArray(parameter)

Examples

```
angular.isArray([ ]) // true
angular.isArray([1994, 1995]) // true
angular.isArray({ }) // false
angular.isArray(1994) // false
```

3.17.4 angular.isDate

This function returns a boolean True if and only if the object passed to it as parameter is of the data type "Date". It is written as: angular.isDate(value)

Examples

```
angular.isDate("Tushar Agarwal") // false
angular.isDate(1994-12-29) // true
angular.isDate(new Date( )) // true
```

3.17.5 angular.merge

This function merges the source object by extending the destination object. It is written as: angular.merge(destination, source)

Examples

```
angular.merge({ }, { })
// { }
angular.merge({name: "Tushar"}, {password: "Agarwal"})
// {name: "Tushar", password: "Agarwal"}
angular.merge({x: 1}, {y: {z: {w: 2}}})
// {"x":1,"y":{"z":{"w":2}}}
```

3.17.6 angular.isDefined and angular.isUndefined

The isDefined function checks if the passed parameter to it has been defined. If it has been, it returns a boolean True, else False. The isUndefined function is the exact opposite. It checks if the passed parameter to it has been defined. If it has been, it returns a boolean False, else Truex

They are written as:

```
angular.isDefined(parameter)
angular.isUndefined(parameter)
```

Examples

```
angular.isDefined(1994) // true
angular.isDefined([1994, 1995]) // true
angular.isDefined(undefined) // false
angular.isDefined(null) // true
angular.isUndefined(1994) // false
angular.isUndefined(undefined) // true
```

3.17.7 angular.toJson

This function serializes an object and converts it into the JSON format. Any parameter that begins with $$ will be removed because $$ indicates internal AngularJS properties. It is written as: angular.toJson(object passed as parameter)

Examples

```
angular.toJson({name: "Tushar", job: "Product Manager", $$randomProperty: isNumber})
// " {"name":"Tushar","job":"Product Manager"} "
angular.toJson(1994)
// " 1994 "
angular.toJson([1994, "1", 1995, "0"])
// " [1994,"1",1995,"0"] "
var someFunction= function(value) {  return value  }
angular.toJson(someFunction)
// undefined, functions have no representation in JSON
```

3.18 Using AngularJS with TypeScript

AngularJS is an absolute champ at front-end development frameworks and brings several benefits like being robust and very well designed.

In this last section of the chapter, we will build a basic AngularJS application using TypeScript that will have just one button called "Get my Music!" and upon clicking it, a list of music will be displayed. Simple enough?

Create a new folder called "myApplication" in the root of your project in the IDE and now this is where we will add all the code for our application.

Creating Interfaces

AngularJS in our application will communicate with the backend to get the track list. To keep the code modularized, we will create a new folder inside "myApplication" called interfaces and here, we will create a blank TypeScript file and add this code:

```
module angularJSWithTypeScript.Interfaces {
  export interface myPlaylistService {
    getPlaylist: () => Array < myTrack >;
  }
  export interface myTrack {
    songTitle: string;
    songArtist: string;
    songRating: number;
  }
}
```

The above code will create a new TypeScript module called "angularJSWithTypeScript". Two interface definitions have been added here.

1. **myPlaylistService** - it has a method called "getPlaylist" and the method shows that it accepts no parameter and returns an array "myTrack". This interface just ensures that no random objects get passed to the code and prevents bugs.
2. **myTrack** - defines the structure of the actual data for your playlist of songs.

Implementing Interfaces

After having defined the interfaces, now we must implement them. We will create another folder inside the "myApplication" folder and name this new folder "services". Here, we add a new TypeScript file called "myPlaylistService.ts" and to it, we add this code:

```
module angularJSWithTypeScript.Services {
  export class PlaylistService implements angularJSWithTypeScript.Interfaces.myPlaylistService {
    httpService: ng.IHttpService
    static $inject = ["$http"];
    constructor($http: ng.IHttpService) {
      this.httpService = $http;
    }
    getPlaylist = ( ) => {
      // To make life easy, this code has hardcoded values.
      // In the real world, GET service would be used with REST APIs to get the data.
```

```
    var result: Array<angularJSWithTypeScript.Interfaces.myTrack> = [
      { songTitle: "Leave Out All The Rest", songArtist: "LP", songRating: 5 },
      { songTitle: "Baby", songArtist: "JB", songRating: 1.2 },
      { songTitle: "Scientist", songArtist: "CP", songRating: 3.9 },
      { songTitle: "Airplanes", songArtist: "B.o.B.", songRating: 3.0 }
    ];
    return result;
  }
 }
 angular.module("angularJSWithTypeScript").service("angularJSWithTypeScript.Services.PlaylistService",
PlaylistService);
}
```

As can be seen in the code above, our service has been encapsulated within "angularJSWithTypeScript.Services". The keyword "export" here allows the service to be called from outside this module using the name of the service "angularJSWithTypeScript.Services".

The service we've created must be able to interact with backend APIs and hence, it has "$http".

You will also see the following line in the code above:

angular.module("angularJSWithTypeScript").service("angularJSWithTypeScript.Services.PlaylistService", PlaylistService);

This is the standard AngularJS code which adds the created class to the AngularJS module as a service. But, we have not yet created the ANgularJS module, so let's do that next.

Creating the AngularJS Module

We will now create a new TypeScript file called "myApplication.module.ts" in the "myApplication" folder and use the code below:

```
(():void=> {
  var myApplication = angular.module("angularJSWithTypeScript", ['ngRoute']);
  myApplication.config(angularJSWithTypeScript.Routes.configureRoutes);
})()
```

In the code above, we first created a module with the "ng-route" dependency and then we configured our application with the routing information by using a reference to a Class (angularJSWithTypeScript.Routes) that has not yet been created. We will do that next.

Defining the Routes

Let us now add another file to the "myApplication" folder called "myApplication.routes.ts" and add the following code to it:

```
module angularJSWithTypeScript {
  export class Routes {
    static $inject = ["$routeProvider"];
    static configureRoutes($routeProvider: ng.route.IRouteProvider) {
```

```
        $routeProvider.when("/home", { controller: "angularJSWithTypeScript.controllers.typescriptDemoController",
templateUrl: "/app/views/playlist.html", controllerAs: "playlist" });
//playlist.htm is the view that'll show the actual webpage
        $routeProvider.otherwise({ redirectTo: "/home" });
    }
  }
}
```

The "controllerAs" parameter you see for the first route shows that the value will be accessible with the "playlist" alias. We can access the controller's properties and methods using this alias.

The controller "angularJSWithTypeScript.controllers.typescriptDemoController" has been mapped to the route we will create next.

Creating the Controller

The controller will be created by adding a new folder to the "myApplication" folder called "controllers" and adding a new file called "typescriptDemoController.ts" to it.

```
module angularJSWithTypeScript.controllers {
  export class TypeScriptDemoController {
    playListService: angularJSWithTypeScript.Interfaces.IPlaylistService;
    static $inject = ["angularJSWithTypeScript.Services.PlaylistService"];
    constructor(playlistService: angularJSWithTypeScript.Interfaces.IPlaylistService) {
      this.playlistService = playlistService;
    }
    favorites: Array<angularJSWithTypeScript.Interfaces.ITrack>;

    getFavourites = ( ) => {
      this.favorites = this.playlistService.getPlaylist( );
    }
  }
angular.module("angularWithTypeScript").controller("angularWithTypeScript.controllers.typeScriptDemoController
", TypeScriptDemoController);
}
```

A new class is created above called "TypeScriptDemoController" with a dependency service called "angularJSWithTypeScript.Services.PlaylistService".

We also have a property called "favorites" with an array:

"Array<angularJSWithTypeScript.Interfaces.ITrack>"

We will use this property to bind to the View. The "getFavourites" function populates the "favorites" property with the result of the call. This will be the click-event handler.

The next step is the View.

Creating the View

We will create a new folder within the "myApplication" folder called "views" and add an empty html file to it called "playlist.html". Here, we add the following program:

```
<div class="displayUI">
  <ul class="list-group">
    <li ng-show="playlist.favorites" class="list-group-item" ng-repeat="x in playlist.favorites">
      {{x.title}}
    </li>
  </ul>
  <button class="button1 button1-block" ng-click="playlist.getFavourites()">Get Your Favorites Songs</button>
</div>
```

Combining Them All

Now, let's bring it all together by adding an "index.html" file to the "myApplication" folder. and ensure that all CSS files are added as well to the content folder in your project folder.

```
<!DOCTYPE html>
<html ng-app="angularJSWithTypeScript">
<head>
  <title>Program on AngularJS With TypeScript</title>
  <link href="content/css/bootstrap.min.css" rel="stylesheet" />
  <link href="content/css/bootstrap.custom.min.css" rel="stylesheet" />
</head>
<body>
  <div class="container">
    <div class="page-header">Click the following button to get your favorite tracks.</div>
    <div class="row">
      <div class="col-md-4 col-md-offset-4">
        <div ng-view=" ">
        </div>
      </div>
    </div>
  </div>
  <script src="scripts/angular.js"></script>
  <script src="scripts/angular-route.js"></script>
  <script src="app/app.routes.js"></script>
  <script src="app/app.module.js"></script>
```

```
  <script src="app/services/playlistService.js"></script>
  <script src="app/controllers/tsDemoController.js"></script>
</body>
</html>
```

Output

When you click the button, the table appears. I have kept the CSS files empty, but you can add any styles you'd like for the User Interface.

Click the following button to get your favorite tracks.

Get Your Favorites Songs

Leave Out All The Rest	LP	5
Baby	JB	1.2
Scientist	CP	3.9
Airplanes	B.O.B.	3

Review Questions

Q. 1 Explain the following : a) MVC; b) DOM Model **(10 Marks)**

Q. 2 Write a simple AngularJS application to add 3 numbers. Draw a mock UI of the output. **(10 Marks)**

Q. 3 State the built-in directives of AngularJS. State without explanation 7 advantages of AngularJS. **(5 Marks)**

Q. 4 What is an AngularJS dependency injection? State 6 examples of AngularJS built-in helper function. **(5 Marks)**

Q. 5 Explain the isDefined and isUndefined functions in AngularJS. **(5 Marks)**

Q. 6 True or False? **(5 Marks)**

(a) angular.equals({ }, { })

(b) angular.isArray([1994, 1995])

(c) angular.isArray({ })

(d) angular.isDate(1994-12-29)

(e) angular.isDate(new Date( )

□□□

4 MongoDB and Building REST API using MongoDB

Syllabus

MongoDB: Understanding MongoDB, MongoDB Data Types, Administering User Accounts, Configuring Access Control, Adding the MongoDB Driver to Node.js, Connecting to MongoDB from Node.js, Accessing and Manipulating Databases, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js, Using Mongoose for Structured Schema and Validation.

REST API: Examining the rules of REST APIs, Evaluating API patterns, Handling typical CRUD functions (create, read, update, delete), Using Express and Mongoose to interact with MongoDB, Testing API endpoints

Self-learning Topics : MongoDB vs SQL DB

4.1 MongoDB Overview

4.1.1 MongoDB Introduction

MongoDB is a document-database designed for the development of modern applications that require fast scaling. The architecture allows the program to scale rapidly by adding new nodes to share the responsibilities (computational load). We will understand what that means over the course of the chapter.

It is essentially an open source & document-oriented DB that can be classified as a NoSQL database. NoSQL tool means that it doesn't utilize the usual rows and columns.

4.1.2 MongoDB Features/Essentials

MongoDB Documents: These are the documents in a Document Database. MongoDB stores data in the form of JSON documents.

The Document Data Model allows for natural object-oriented nature in the code, making it simple to use and maintain.

The fields in the JSON document can vary and are not as straightforward as using a relational database, where adding more data means adding to a new row for all existing columns or adding a column to every existing row.

Documents can be nested for hierarchies to store data in arrays.

JSON structure

```
{
    "car": "mercedes"
    "year": "2016"
```

```
"Style": {
            "convertable"
             "red"
             "leather"
          ]
 "engine": {
              "power": "12 ohp"
               "liters": 78"
           }
}
```

The document model and the supported hierarchies provide programmers with the flexibility to scale and work on complex and ever-changing data sources and quickly adapt with growing needs.

MangoDB can even convert documents into Binary JSON or BSON for quicker access and more data type support. MongoDB however, on the front end, still remains a JSON database.

MongoDB Collections: A group of documents in MongoDB is called a collection. Think of a collection in MongoDB as a table. Collections, however, can be more flexible. They don't enforce any schema on the programmer to comply with and documents part of one collection can have different fields too. Every collection is associated with one MongoDB database.

MongoDB Replica Sets: To ensure high availability, duplicates of the data are stored in MongoDB. Having more than one copy ensures that data is always available and not lost if something happens to one copy. The system automatically creates two copies of your data and this is the replica set. This offers redundancy protection and offers availability in times of maintenance or failure.

MongoDB Sharding: Sharding refers to distributing data across machines to ensure queries can be handled faster using copies on the closest machines.

MongoDB shards or breaks down the data and then distributes it across the cluster of machines and the result is a highly scalable architecture.

MongoDB Indexes: MongoDB supports indexing strategies to offer faster execution. This is because queries can scan indexes first before going through the entire data and reading every document to speed up execution.

MongoDB Aggregation Pipelines: MongoDB has a framework for creating and utilizing data pipelines called aggregation of pipelines. It enables programmers to process, transform and analyze data at scale using >150 operators and expressions.

4.1.3 Differences between RDBMS and MongoDB

Table 4.1.1

Sr. No.	RDBMS	MongoDB
1	Relational Database	Non-relational, hierarchical, document-oriented database
2	Not suited to nested data storage	Suited for nested data storage
3	Scaled by increasing RAM	Scaled by increasing number of machines

Sr. No.	RDBMS	MongoDB
4	Predefined schema	Dynamic schema
5	**Primary Properties :** Atomicity, Consistency, Isolation, and Durability	**Primary Properties :** Consistency, Availability, and Partition tolerance
6	Slow	Fast (100 times faster than RDBMS)
7	Supports complex joins	No support for joins
8	Supports only SQL	Supports JSON querying language and SQL too

4.1.4 Sample MongoDB Document

The following is the structure of an example document:

```
{
 _id: ObjectId(12axsbt28xbn)
 title: 'Learning MongoDB',
 description: 'MongoDB by Tushar Agarwal',
 by: 'Tushar Agarwal',
 url: 'http://www.abc.com',
 tags: ['mongo', 'database', 'db', 'nosql'],
 likes: 50,
 comments: [
  {
   user:'user123',
   message: 'Hello World'
  },
  {
   user:'user456',
   message: 'Some comment'
  }
 ]
}
```

4.1.5 MongoDB Data Types

MongoDB supports the following data types :

- **String** : The most widely used and common data type.
- **Integer** : Stores numerical values and can be both 32 or 64 bit depending on how much load your server can handle.
- **Boolean** : Stores true or false values.
- **Double** : Stores floating point values.

- **Min/ Max Keys** : Used to compare the largest and the smallest Binary JSON elements.
- **Arrays** : Stores lists of values in one key.
- **Timestamp** : Records when a document has been created or updated.
- **Object** : Stores embedded documents.
- **Null** : Stores null values.
- **Symbol** : Identical to strings but only used for languages that have a symbol data type.
- **Date** : Stores current date in UNIX format. Can also store user created dates if passed the date elements.
- **Object ID** : Stores a document's ID.
- **Binary** : Stores any kind of binary information.
- **Code** : Stores JavaScript code.
- **Regular Expression** : Stores regular expressions.

4.1.6 MongoDB User Accounts & Access Control

User management in database management is of utmost importance to ensure security and understand who is accessing the database and whether they have the requisite permissions for it. User access can get messy in the absence of proper user accounts and access control.

MongoDB makes use of RBAC or Role-Based Access Control to implement the required controls and credentials.

So, what is Role Based Access Control ? Role Based Access Control or RBAC is an approach that restricts the use of resources (system) to only those authorized to use it.

Managing an individual's rights and permissions is as simple in MongoDB as assigning an appropriate role for a task and placing that role in the user's account.

The primary rules regarding RBAC are:

1. **Role Assignment :** Every subject must have at least one role.
2. **Role of Authorization :** The subject must have an active role authorized.
3. **Permission Authorization :** Commands are executed only if permission is authorized on the active role.

MongoDB User Roles

MongoDB has a few different kinds of roles that we will now discuss:

A) Built-in Roles

Role based authorization is enforced via built-in roles that give you several levels of access to the database. Roles can give several privileges to perform tasks for a resource that is available. They are of several categories:

- **User Database** : Thai role offers simple read or read and write permissions for the database.
- **Database Administration** : This role offers admin privileges like updating the schema and objects within it.
- **Cluster Administration** : This role can monitor the whole MongoDB system, including the replica sets and the shards spread all over the cluster.
- **Backup and Restoration** : This role is responsible for all backup and maintenance activities and has access to cache data for restoration.

- **All-Database Roles** : This role has access to all the databases within the schema. Does not have access to config files.
- **Superuser** : Can grant any level of access to any user and for any database. This is root access.

B) User Defined Roles

MongoDB allows us to create our own roles according to our needs and the needs of the roles who would use the data within the database. db.createRole() is the function within Mongo that can create new roles and assign them names and privileges. db.dropRole() removes a role and db.getRole() gives us information on all existing roles.

What kind of actions come under the purview of Access Control? All actions performed by a user on a resource. They are called privilege actions.

They are of many kinds :

- **Database Management Actions:** Changing passwords, creating new collections, creating indexes, etc.
- **Querying Actions:** Data manipulation like inserts, deletes and all updates.
- **Deployment Management Actions:** Allow changes to all DB configurations.
- **Replication Actions:** Set, execute and manage replica sets in MongoDB.
- **Server Administration Actions:** Maintain the operating system and the logs and ensure that all systems in the cluster are working.
- **Sharding Actions:** Ability to add and delete shards.
- **Session Actions:** Create, maintain and kill user sessions.
- **Diagnostic Actions:** Diagnose stats to know the health of the database for stability and latency issues.
- **Free Monitoring Actions:** Monitor the database for any untoward events and track live metrics.

4.2 Setting up MongoDB

For an in depth installation guide for all kinds of operating systems,

Please visit : *https://docs.mongodb.com/manual/installation/*

4.2.1 Adding MongoDB Driver to Node.js

As a prerequisite, we need to make sure that a supported version of Node.js has been installed.

AfterInstall the MongoDB Node.js Driver

The MongoDB Node.js Driver lets programmers connect to and interact with the MongoDB databases using Node.js applications. For the next set of actions described, we will need to connect to the database. To install the driver, use the command:

npm install mongodb

Once installed, the following command will give you the version number:

npm list mongodb

The next step is getting the Mongo database. We will use a MongoDB-managed database service called Atlas.

You can visit *mongodb.com/atlas/database* to create a new free tier cluster. A cluster is the set of nodes where all copies of your data are stored.

Head over to Atlas and create a new cluster in the free tier. At a high level, a cluster is a set of nodes where copies of your database will be stored. Once your tier is created, load the sample data.

After that, sample data will be loaded. To load some sample data into the cluster, do the following :

1. Go to "Database Deployments" view in Atlas and then, in the left navigation pane in Atlas, click on "Database".
2. Open the dialog titled, "Load Sample Dataset" and now find the cluster you just created to load Atlas' sample datasets in it. Click on the triple dot ellipsis button (...) to find your cluster.
3. In the dialog that will then appear, click on "Load Sample Dataset". Atlas will now load some sample datasets into the selected cluster.
4. To see the sample data, just click your cluster's "Browse Collections" button. You will see multiple sample databases in your cluster like mflix, airbnb, geospatial, etc.

4.2.2 Connecting to MongoDB from Node.js

The next step is to prepare the newly created cluster to establish the connection.

Go to your cluster just created in Atlas and click on "Connect". A wizard will now appear. It will ask you to enter your IP Address to the access list to whitelist it and create a MongoDB user. Note down/remember your username and password.

Next, this Wizard will ask for a connection method. Choose "Connect your Application" and then after entering the node.js version you have, copy the connection string the Wizard gives you.

Everything is now set up and it's time to write a piece of code to make sure the connection is working correctly and as expected. We will write a simple script now in node.js that connects to the database and lists what you have in your cluster.

There are few steps here that we must follows to achieve the desired result :

Import MongoClient : MongoDB exports MongoClient, to be more clear, an instance of MongoClient and thai will assist us in connecting to the MongoDB database.

```
const { MongoClient } = require ( 'mongodb' );
```

Main Function : Now, we create an asynchronous function called main ( ) and using it, we will connect to the MongoDB cluster, call the functions that query into our database.

```
async function main()
{
    // code will be written here soon
}
```

Inside the main ( ) function, we have to first create a constant for our connection URI. This is the string I asked you to copy early on in the installation procedure. Here, we will also update the username and password for the user we created earlier. The connection string also has a *dbname* placeholder. We will use Atlas' AirBNB database as a sample here, so *dbname* will be replaced with the name "sample_airbnb".

```
const uri = " mongodb+srv:// <username>:<password> @ <your-cluster-url> /test ?
retryWrites = true & w = majority ";
```

Now that the URI has been added, we will create an instance of MongoClient.

```
const client = new MongoClient ( uri );
```

We are now ready to connect to the cluster. The following command will be used for it:

```
await client.connect();
```

Client.connect ( ) will return a promise and the await keyword will wait for further execution until this operation has been completed.

Let us now build a function. This function will print the names of all of Atlas' inbuilt databases that we imported earlier. Let's call this function "listDatabases".

```
await listDatabases(client);
```

Let us also go ahead and add a try and catch block to catch any unwanted errors and notify us of them.

```
try {
  await client.connect ( );
  await listDatabases ( client );
 } catch ( exception err ) {
  console.error( err );

}
finally {
  await client.close();

}
```

The "*finally*" statement at the end will close the connection to the cluster. We never want concurrent updates to the same cluster by multiple connections.

Now, we call the main function:

```
    main ( ).catch ( console.error );
```

Written all together, it now would look like this :

```
async function main ( ) {
const uri = " mongodb+srv:// <username>:<password> @ <your-cluster-url> /test ?
retryWrites = true & w = majority ";
  const client = new MongoClient ( uri );
  try {
    await client.connect( ); //connect to the cluster
    await  listDatabases( client ); //make the needed call (function will be implemented later)
  } catch ( exception err ) {
    console.error(err);
  } finally {
    await client.close( );
  }
}
main( ).catch( console.error );
```

Let us now write the listDatabases() function to get the list of databases in the cluster and print it on the console:

```
async function listDatabases( client ) {
  databasesList = await client.db( ).admin( ).listDatabases( );
  console.log("List of Databases:");
  databasesList.databases.forEach (db => console.log (` - ${db.name}` ) );
};
```

Let us now save the file with the name "listdata.js" and then invoke the query from the command line :

```
node listdata.js
```

Output :

List of Databases :

```
- sample_airbnb
- sample_geospatial
...
- sample_mflix
- sample_supplies
...
- admin
- local
```

4.2.3 Accessing and Manipulating Data (Documents) using Node.js

4.2.3(A) Create a Database

Let us create a database first:

```
var MongoClient = require ( 'mongodb' ).MongoClient;
var my_url= "mongodb://localhost:12212/mydb";

MongoClient.connect(my_url, function(error, db1) {
 if (error) throw error;
 console.log("Your database has now been created !!");
 db1.close();
});
```

Once the code has been saved in a file (say "demo_db_creation.js) and run:

```
C:\Users\Computer_Name>node demo_db_creation.js
```

Output :

```
Your database has now been created !!
```

4.2.3(B) Create a Collection

Use the createCollection ( ) method to create a collection in MongoDB. Here, we create one called "mycustomers".

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://localhost:12212/";

MongoClient.connect(my_url, function(error, database) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 db1.createCollection("mycustomers", function(error, res) {
```

```
 if ( error ) throw err;
 console.log( "Your collection has been created !!" );
 database.close();
 ));
});
```

Once the code has been saved in a file (say "demo_collection_creation.js) and run :

```
C:\Users\Computer_Name>node demo_collection_creation.js
```

Output :

```
Your collection has now been created !!
```

4.2.3(C) Insert into Collection

To insert a document into a MongoDB collection, we need to use the insertOne ( ) method. A document in MongoDB is the equivalent of a record in RDBMS.

Let us see how it works by inserting a document into the "mycustomers" collection :

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://localhost:12212/";

MongoClient.connect(my_url, function(error, database) {
 if ( error ) throw error;
 var db1 = database.db( "mydb" );
 var myobj = { name: "ABC Company", address: "123 Main Street" };
 db1 .collection("mycustomers").insertOne(myobj, function(error, res) {
  if ( error ) throw error;
  console.log( "Document successfully inserted into collection !!" );
  database.close();
 });
});
```

Parameter one for the function is an object with the name and the value of each field in the document that we aim to insert into the collection. The callback function is where we can handle any errors that may arise.

Now, save this code in a file say "insert_record.js" and run it:

```
C:\Users\Computer_Name>node insert_record.js
```

Output :

```
Document successfully inserted into collection !!
```

4.2.3(D) Find in Collection

The findOne ( ) method can be used to find any items in the documents already inserted into a collection. It will return only the first occurrence.

The first parameter in this method is the query object. We will pass an empty query object which will return all documents but the first one will be returned.

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://localhost:12212/";
MongoClient.connect(my_url, function( error, database) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 db1.collection( "mycustomers" ).findOne({ }, function( error, result ) {
  if ( error ) throw error;
  console.log( result.name );
  database.close();
 });
});
```

Now, save this code in a file say "find.js" and run it:

```
C:\Users\Computer_Name>node find.js
```

Output :

```
ABC Company
```

4.2.3(E) Query the Result

To find documents in a collection, you can filter a result using a query object. The first parameter of the find ( ) method is the query object and it filters your search based on your need.

Let's look for documents that have the address "123 Main Street".

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://localhost:12212/";

MongoClient.connect(my_url, function( error, database) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 var query = { address: "123 Main Street" };
 dbo.collection( "mycustomers" ).find( query ).toArray( function( error, result) {
  if ( error ) throw error;
  console.log( result );
  database.close();
 });
});
```

Now, save this code in a file say "query.js" and run it:

```
C:\Users\Computer_Name>node query.js
```

Output :

```
[
 { _id: 63hd7u8nnd8n3xun3icn394 , name: 'ABC Company', address: '123 Main Street' }
]
```

4.2.3(F) Delete a Record/Collection

The deleteOne ( ) method is used to delete a record or document from a collection. The first parameter is the query object defining what needs to be deleted.

On a similar note, the drop ( ) function can be used to drop an entire collection. Let us look at both of them in the same program next.

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://localhost:12212/";

MongoClient.connect(my_url, function(error, database) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 //first drop a record from the collection
 var my_query = { address: '1050 Main Street' };
 //no record with this address exists in the collection but assume that there is one.
 db1.collection("mycustomers").deleteOne(my_query, function( error, obj)
 //now delete the whole collection
 var db2 = db.db("mydb");
 db2.collection("mycustomers").drop (function ( error, deleteOK ) {
  if ( error ) throw error;
  console.log("Specified document has been deleted !!");
  if (deleteOK) console.log( "Your collection has been successfully deleted !!");
  database.close();
 });
});
```

Now, save this code in a file say "delete_record_and_collection.js" and run it:

```
C:\Users\Computer_Name>node delete_record_and_collection.js
```

Output :

```
Specified document has been deleted !!
Your collection has been successfully deleted !!
```

4.2.3(G) Update the Record

We make use of the updateOne ( ) method to update a record that already exists in MongoDB. The first parameter again is the query object defining the document that we want to update. If more than one record exists, the first one is updated. The second parameter defines the new values for the record we want to update.

Let us update the record with the address "123 Main Street" with the name "Ram" and address "123 Prime Avenue".

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://128.0.0.1:12212/";

MongoClient.connect(my_url, function( error , database) {
```

```
 if ( error ) throw error;
 var db1 = database.db("mydb");
 var myquery_1 = { address: "123 Main Street" };
 var newvalues = { $set: {name: "Ram", address: "123 Prime Avenue" } };
 db1.collection( "mycustomers" ).updateOne(myquery_1, newvalues, function(error, res) {
  if ( error ) throw error;
  console.log( "Your document has been updated !!" );
  database.close();
 });
});
```

Now, save this code in a file say "update_record.js" and run it:

```
C:\Users\Computer_Name>node update_record.js
```

Output :

```
Your document has been updated !!
```

4.2.3(H) Limit the Collection

To limit the results in the output, we use the limit ( ) method, which has just one parameter: defining the documents to be returned.

Assume that you have a collection of customers already in MongoDB:

```
mycustomers
[
 {_id: 4237r23nfd2394bu42324cn, name: Tushar, address: '123 Main Street'},
 {_id: jd242i42nc49242f7234249d, name: Ram, address: '123 Prime Avenue'},
 {_id: 346bd63rvgd63hks93nms9 , name: 'John', address: '123 Bellevue Ave'},
 {_id:  nsdjwi291ms92ns2nd8892 , name: 'Jill', address: 123 Mountain Street'}
]
```

Let us now only return 2 documents:

```
var MongoClient = require('mongodb').MongoClient;
var my_url = "mongodb://localhost:27017/";

MongoClient.connect ( my_url, function( error, database ) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 db1.collection("mycustomers").find().limit(2).
                toArray(function( error, result) {
  if ( error ) throw error;
  console.log ( result );
```

```
  database.close();
 });`
});
```

Now, save this code in a file say "limit.js" and run it:

```
C:\Users\Computer_Name>node limit.js
```

Output :

```
[
 { _id: 4237r23nfd2394bu42324cn, name: Tushar, address: '123 Main Street'},
 { _id: jd242i42nc49242f7234249d, name: Ram, address: '123 Prime Avenue'}
]
```

4.2.3(I) Join Collections

A left outer join can be performed in MongoDB even though MongoDB is not a relational database using "$lookup". It lets you decide which collection you want to join with the current collection and what field(s) have to be matched.

Assume that you have two collections: "myorders" and "myorderitems"

Collection 1: "myorders"

```
[
 { _id: 1, item_id: 111, status: 1 },
 { _id: 2, item_id: 222, status: 1 }
]
```

Collection 2: "myorderitems"

```
[
 { _id: 111, itemname: 'Tootsie Rolls' },
 { _id: 222, itemname: 'Cacao Nibs' },
 { _id: 222, itemname: 'Headphones' },
]
```

Now, let's join the "myorderitems" to the "myorders" collection:

```
var MongoClient = require( 'mongodb' ).MongoClient;
var my_url = "mongodb://128.0.0.1:12212/";

MongoClient.connect(my_url, function( error, database) {
 if ( error ) throw error;
 var db1 = database.db("mydb");
 db1.collection( 'orders' ).aggregate ( [
  { $lookup:
    {
     from: myorderitems,
```

```
    localField: 'product_id',
    foreignField: '_id',
    as: 'myorderdetails'
   }
  }
 ]).toArray(function(error, res) {
 if ( error ) throw error;
 console.log(JSON.stringify(res));
 database.close();
 });
});
```

Now, save this code in a file say "join.js" and run it:

```
C:\Users\Computer_Name>node join.js
```

Output :

```
[
 { "_id": 1, "product_id": 111, "status": 1, "myorderdetails": [
  { "_id": 111, "name": "Tootsie Rolls" } ],
 "_id": 1, "product_id": 222, "status": 1, "myorderdetails": [
  { "_id": 222, "name": "Cacao Nibs" },
  { "_id": 222, "name": "Headphones" } ]
 }
]
```

4.3 Using Mongoose for Structured Schema and Validation

Having understood the native node.js driver, we will now make the jump top Mongoose. Mongoose is an Object Document Model that pipes additional functionality onto the native driver. It simply applies a more structured schema to a collection and allows for additional validation and type casting of variables.

The builder object that Mongoose lets you create allows you to pipe commands for finding, updating, saving, removing, aggregating and many other functionalities. Mongoose won't replace the native node.js driver. It will only enhance this driver with greater functionalities.

4.3.1 Advantages of Mongoose

1. A schema can be created for your documents.
2. Documents in the model can be successfully validated.
3. Data can be typecast as needed.
4. Business logic can be hooked via Middleware.
5. Mongoose is easier to learn than the node.js driver.

4.3.2 Disadvantages of Mongoose

1. A schema is mandatory which is a hassle when one is not needed.
2. Operations like storing data are performed with more efficiency by the node.js driver than by Mongoose.

4.3.3 New Objects Introduced by Mongoose

Mongoose is situated right above the node.js driver and adds some new objects to it.

The first is the Schema object which defines the needed structure for the documents in a given collection. It aids in the definition of fields and data types to add indexing or any validation to them.

The Model object is just a representation of the documents present in a collection.

The Document object is the representation of any individual document in a collection.

4.3.4 Connecting to the MongoDB Database

For connections to Mongoose, we use a similar connection string to node.js.

Let us now see the code that will import Mongoose, connect to the database, wait for open events and then display the connections present in the database before disconnecting.

```
var mongoose = require( 'mongoose' );
mongoose.connect( 'mongodb://localhost/mywords' );
mongoose.connection.on('open', function ( ) {
console.log(mongoose.connection.collection);
mongoose.connection.db.collectionNames( function( error, names) {
console.log( mynames );
mongoose.disconnect ( );
});
});
```

Defining A Schema

The main requirement for making use of Mongoose is the presence of a Schema. It will define for us the fields and the field data types for the documents made use of in a collection. This comes in handy to validate and typecast any objects to match the needs of the predefined schema. A schema is defined for every document type in a collection. We define a data type for each field in a schema.

The data types available are :

- String
- Number
- Boolean or Bool
- Array
- Buffer
- Date
- ObjectId or Oid
- Mixed

Understanding Mongoose Paths

Paths define access to fields in the document hierarchy. If a document has a field name, which has subdocuments called firstname, lastname and age, then the paths to the subdocuments are as follows:

- name.firstname
- name.lastname
- name.age

Creating a Mongoose Schema

To create a schema, we need a new instance of the Schema object. It accepts an object describing the schema as the first parameter and the second parameter are the options available when defining a schema as shown in Table 4.3.1.

Table 4.3.1

Option	Description
autoIndex	Boolean - when true autoIndex is turned OFF. Default is true.
BufferCommands	Boolean – When true, commands cannot be completed due to buffering server is sum. Default is true.
Capped	Max number of documents supported in a capped collection.
Collection	Collection name for schema model. Mongoose connected to this collection when compling model.
Id	Boolean – when true documents will have in id greater than the id value default is true
Read	Specify replica read preferences like primary, primary preferred secondary preferred or nearest
Safe	Boolean - when true mongoose applies write concern to request that update to data base. Default is true.
Strick	Boolean – when true any attributes passed that are not parts of the schema are not saved.

Let us create a schema called students, which has a name field (string), a gpa field (number) and a marks field (array of numbers).

```
var student_schema = new Schema ( {
 name: String,
 gpa: Number,
 marks: [ Number ] },
 {
  collection: 'students'
 } ) ;
```

Adding Index to a Schema

Indices can be added to a field that is frequently used to speed up execution time of queries. We can add an index to a schema object when defining the schema or we can use the index command.

Let's have a look at both :

```
var myschema = new Schema ( {
name: {
 type: String,
index: 1 }
} );
// OR
var myschema = new Schema ( {
 name: String
} );
schema.index ( { name: 1 } ) ;
```

To get the list of all indexed fields, we can use the method *schema.indexes( )*

Implementing Unique Fields

For validation that includes specifying that the value of some field must always be unique in the collection, we can use the "unique" property in the Schema definition.

```
var myschema = new Schema ( {
 name: { type: String,  index: 1,  unique: true }
} );
```

Forcing Required Fields

Think of this as a Primary Key from Relational Databases. Any new instance must have this field included. By default we do not have to specify any field, but forced fields must be specified. We achieve this using the "Required" property.

```
var myschema = new Schema ( {
 name: { type: String,  index: 1,  unique: true,  required: true }
} );
```

To get the list of all required fields, we can use the method *schema.requiredPaths( )*

Adding Methods to the Schema

In Mongoose, we can add methods to the Schema object that become available to the Document objects within it. They can now calle these methods. Methods are added to the Schema object by assigning a function to the Schema.methods property of the Schema object.

The function is just a basic JavaScript function that belongs to the Document object and this Document object is accessed using the "this" keyword.

The example below is a function that will combine the first and last names to provide the full name:

```
var myschema = new Schema (
 {
```

```
  first: String,
  last: String
});
schema.methods.fullName = function ()
{
  return  this.first + " " + this.last;
};
```

Mongoose Validators

Mongoose gives us some excellent builtin Schema validators and properties that we will now discover in this section.

Validators are used to validate the values provided before they are committed to the database.

Table 4.3.2

Property	Description
match	Regular Expression - it creates a validator that examines the provided value with the expression.
enum	Array - it creates a validator to test if the provided value is present in the array.
minlength	Number - it creates a validator to check if the value length is less than the provided number in the property.
maxlength	Number - it creates a validator to check if the value length is greater than the provided number in the property.

Let us see how these validators are used :

```
let demoSchema = new Schema ({
  someText: {
    type: String,
    required: true,
    uppercase: true,
    minLength: 10,
    maxLength: 20
  }
});
```

Let us look at two number validators :

Table 4.3.3

Property	Description
min	Number - it creates a validator to check if the value is greater than the provided minimum in the property.
max	Number - it creates a validator to check if the value is less than the provided maximum in the property.

Let us look at a full fledged example :

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

let demoSchema = new Schema({
  _id: new Schema.Types.ObjectId,
  myname: {
    type: String,
    required: true,
    uppercase: true,
    trim: true,
    minLength: [10, 'Your name is far too short !!'],
    maxLength: [20, 'Your name is far too long!!']
  },
  email: {
    type: String,
    required: true,
    uppercase: true
  },
  mobile: {
    type: Number,
    required: [true, 'Enter your mobile number please: ']
  }
},{
  collection: 'myusers'
})

const Myuser= mongoose.model(Myuser, demoSchema);

module.exports = Myuser;
```

We have not only used validators here, but also thrown alerts if the validation fails. As you see in the minLength and maxLength properties, if the validation is not a match, an alert is displayed to the user.

The last statement: the model ( ) function is a part of the Mongoose model. The function uses the syntax - model (name, [schema]). The first parameter is a string that is used to find the model and the second parameter is the Schema object.

4.4 REST API

4.4.1 Overview of REST API

We have already seen how to set up a MongoDB database, but for now, we have been interacting with it via the shell. In this chapter, we will learn to build a REST API so that we can interact with the database using HTTP requests and perform basic CRUD functions, i.e. Create, Read, Update and Delete.

We will start with understanding the rules of the REST API before evaluating API patterns and the actual CRUD tasks. As we move forward, we will learn a whole lot more about Mongoose and some node.js programming.

Let us first recap what the REST API is.

REST stands for REpresentational State Transfer, an architectural style which is stateless, meaning it has no idea of any current user or their history of interactions.

API stands for Application Program Interface, which allows applications to "talk" or interact with each other.

The REST API is the stateless interface to the database and allows a far for applications to work on the data. REST APIs have a set of standards. They are not mandatory but are highly recommended. It is always best to have online support and doing it the right way ensures that if you are stuck, there is always help available.

The REST API takes an incoming HTTP request, performs its processing and always sends back an HTTP response.

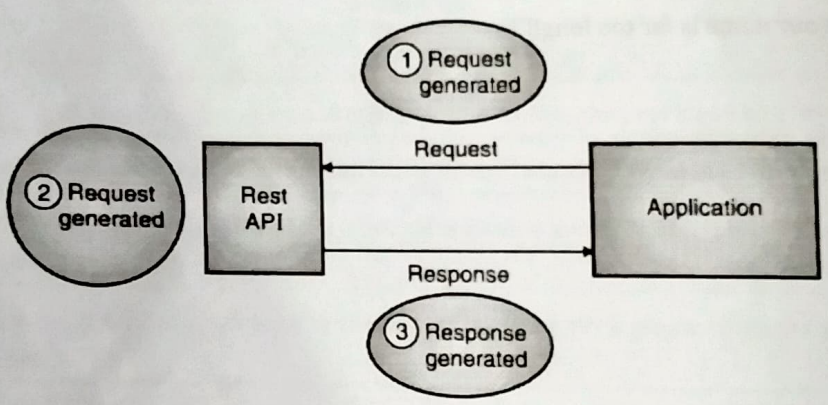


Fig. 4.4.1

4.4.2 REST API Standards

4.4.2(A) Request URLs

Request URLs for a REST API have a simple standard. Following this standard will make your API easy to maintain in case of issues.

Think about the collections in your database. You may have a set of URLs for each collection and for the subdocuments within the collection. Each URL will have the same base path and then a bunch of other parameters.

Within these URLs, you want to cover and maintain a number of actions (usually CRUD). The common actions you'd want to cover are the creation of new items, reading one or a list of items, updating an item or deleting an item.

Let's see how they might look :

URL to create/read a new location: *http://abc.com/api/randomlocation*

URL to read/update/delete a specific location: *http://abc.com/api/randomlocation/123location*

From the URLs above, you will see that multiple jobs can have the same URL. On a specific location, to read, update or delete, they have the same parameter (123location). How is this possible? The answer lies in the request methods.

4.4.2(B) Request Methods

HTTP requests can have different methods that notify the server of what action needs to be taken. The most widely used one is the GET request, used to enter the URL from the address bar in the browser. Another is the POST, used to submit data. Let's look at the methods that use the API and their most common use cases.

Table 4.4.1

Request Method	Use	Response
POST	To create data in the database	The new data object created
GET	To read data from the database	Data object that answered the request
PUT	To update data in the database	The newly updated data
DELETE	To delete data from the database	Nothing

The first column you see is that a different method is used for each of the CRUD operations. A well designed API should have the same URL for different actions. Here, it's the method that tells the server what to do.

Let us again look at the request URLs. Below, you saw earlier that the URLs are the same for different actions. These actions are mapped to different request methods shown above and that's how the server knows what to do.

URL to create/read a new location: *http://abc.com/api/randomlocation*

URL to read/update/delete a specific location: *http://abc.com/api/randomlocation/123location*

4.4.2(C) Response and Status Codes

A good API does not leave you hanging. If you go in for a handshake and the other person refuses to acknowledge it, that's not a good colleague. If you make a request, a good, well-designed API will always respond.

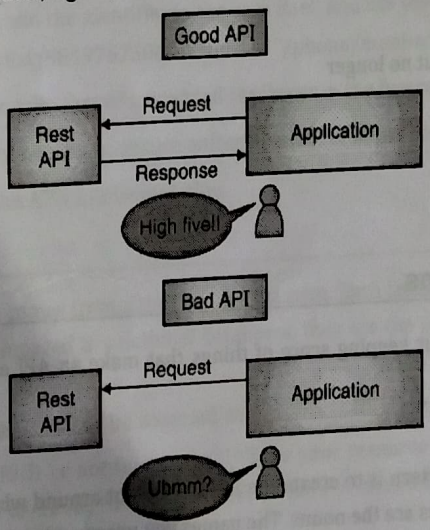


Fig. 4.4.2

Standardizing the API response is just as important as the API request format. The response has two parts - the returned data and the HTTP status codes. To move forward after the user gets the API response, they need the information gleaned from both parts of the response.

It is important that the API returns data in a consistent format and the typical data formats are XML and JSON because they naturally fit in with the MEAN stack architecture. JSON is more compact than XML and is usually a better format for faster responses.

The response will have one of the three things :

- The actual response data
- Error data, or
- A null response

The other thing a good, well-designed API should send back are HTTP status codes. The most famous one that we all know is "404" - Page Not Found. This happens when the server has been requested for a page that can't be found by the server.

404 may be the most prevalent code known on the internet, but there are many more the API can throw back like "400" - Bad Request or "401" - User Not Authorized (to make the request) or "500" - Internal Server Error.

A list of such known codes :

- **200**: Successful Request
- **201** : Successful Request after Create
- **204** : Successful Request with no Content Returned
- **301** : Permanently Redirect
- **400** : Bad request
- **401** : Unauthorized
- **403** : Forbidden
- **404** : Not Found
- **410** : Gone, existed earlier but no longer
- **429** : Too Many Requests
- **500** : Server Error
- **503** : Service Unavailable

4.5 Evaluating API Patterns

Understanding API patterns is all about keeping score of things that make an API easy to design, understand, build and maintain.

Using Friendly Endpoint Names

A very typical way to design an API pattern is to create the API's endpoint around what resources are available. The HTTP methods are the verbs and the resources are the nouns. The names you use must be as descriptive as you can make them.

Let us say that the API is about cooking so you can name the API endpoint as "recipes". Do not, however, name it as "getRecipes". Why? Because sending data to "getRecipes" does not convey the correct intentions.

If you want to add any new recipe here, we can use POST and to get a whole list of what's already in the database, we can use GET. To get anything specific, we can use specific identifiers like a number associated with each specific dish.

GET vs. POST

When the API is being designed, we want to make use of the HTTP methods to showcase the main reason for any call. It is because of this that we do not want to use POST only to retrieve data. The same way, you do not want to use GET to create or delete any data. The use case of the call should determine what is to be used. Many other factors will also make this decision for you :

1. GET requests may be cached
2. GET requests can be called any number of times to get the same result
3. GET requests cannot use sensitive data
4. GET requests always have to factor in output length restrictions
5. GET requests must be utilized only to retrieve data
6. POST requests cannot be cached
7. POST requests cannot be called any number of times with the same result
8. POST requests do not have to factor in output length restrictions

Choosing the right method is an essential part of REST API patterns.

Best Practices for API URLs

The URLs describe resources and the HTTP verbs describe what actions can be performed. So, the URLs must always contain nouns and never verbs. The nouns will describe the domain model and always pay attention to the below points:

1. The nouns should not be unnecessarily long or complicated.
2. If nouns are two different words, separate them with a hyphen.
3. Use common terminologies.
4. Use lowercase to remove confusion around case sensitivity.
5. To access specific resources, add the identifier in the path itself and not the query.
   For ex: use "/phone/numberlist/9819767506" instead of "/phone/numberlist?no=9819767506"
6. Try to use nouns in plurals to take care of any added resources in the future.
7. Avoid more than two levels of nesting to ensure enforcement of best-practice API patterns.
8. URLs should not give away the APIs implementation.

4.6 CRUD

CRUD is a very common acronym that comes from database management. Each letter in CRUD is a different type of action - Create, Read, Update and Delete. If compared to relational databases, they are the equivalent of Insert, Select, Update and Delete.

Every URL points to a set of data and the data can be accessed with different HTTP verbs like GET, POST, PUT and DELETE.

How do you decide whether to use CRUD or not for your API? Take your resource (file/document/etc) and put it in this sentence: "Using this API, I want users to create, read, update and delete x", where x is the resource. If the sentence fits your purpose, then CRUD is a good option for your API.

Let us look at the HTTP verbs that are at the core of CRUD :

4.6.1 POST

POST is the verb used to create new data (resources). It can create a parent resource or even a subordinate resource within a parent resource. POST can assign a new ID (Uniform Resource Identifier) to the newly added resource.

If a resource gets created successfully, the HTTP status 201 is returned to signify success and a location header with a link to the resource newly created.

The POST method is neither safe nor idempotent. Not being idempotent means that every time POST is run, the same response is not guaranteed. Multiple calls will result in duplicates.

Examples :

- POST http://www.abc.com/employees
- POST http://www.abc.com/employees/tushar/names

4.6.2 GET

The GET method, as we will now learn about, is used to read or retrieve a resource. If it works error-free, it returns an XML or a JSON representation of the data and an HTTP status code 200, which means "OK". If the call results in an error, it returns the status code 404 which stands for "Not Found" or 400, which means "Bad Request".

GET requests are used only to read data and never to change it. When used this way, they are considered to be very safe since there is no risk of modification or corruption to data. Calling GET 1000 times is the same as calling it once, as long as nothing changes between the calls. And so, GET is considered idempotent. Each request will return the same value as if they come from the same request.

Examples :

- GET http://www.abc.com/employees/tushar
- GET http://www.abc.com/employees/tushar/names

4.6.3 PUT

This HTTP method is used for updating resource values. The request will contain the new value for the original resource to be updated.

PUT can however, create a resource on a limited basis. This happens when the identifier is chosen by the client instead of the server. This means that the request contains an identifier that doesn't exist. Because of this confusion, this certain way of using PUT should be avoided as far as possible.

If the PUT request goes through successfully, status code 200 (Successful Request) is returned. If there is no content to be returned, then a code 204 (Successful Request with no Content Returned) is sent back.

The PUT method is not a safe operation. This is because it makes updates to the data and thus the state of the server side. The PUT request however, is idempotent. If you make the same PUT call again to create or update a resource, the resource is still there on the server side and it still has the same state, so there is no change. But if you increment or decrement a counter using PUT, the call won't be idempotent.

Examples :

- PUT http://www.abc.com/employees/tushar
- PUT http://www.abc.com/employees/tushar/names/suraj

4.6.4 DELETE

This is a simple HTTP method to understand. It is used to delete a resource identifier by the identifier.

If a resource has been deleted successfully as a result of the DELETE operation, the HTTP status code 200 (OK) is returned with the response body if needed. The response body may have the new representation of the data after deletion of the requested resource, but we avoid sending a response back because it could take up too much bandwidth. The status code 204 (Successful Request with no Content Returned) can also be returned if there is nothing to return.

DELETE requests are idempotent because once a resource has been deleted, it has been deleted and cannot be undone. Calling the same DELETE request on the same resource over and over again will not provide a different response. However, in special cases, if the DELETE request is used to decrement some counter to check the number of resources left after the current call, then the request is no longer idempotent because the counter value goes down with each call. Using DELETE for such work is not recommended.

There is a caveat to the idempotence of DELETE however. If a resource has been deleted by one DELETE request, a second request would return 404 status, which means "Not Found".

This leads some people to say that the responses are different and this makes DELETE non-idempotent. However, the state of the resource in question stays the same (deleted), and thus, it can be called idempotent.

Examples :

- DELETE http://www.abc.com/employees/tushar
- DELETE http://www.abc.com/employees/tushar/names

4.7 Using Express and Mongoose to Interact with MongoDB

Any project that runs with Express as a Web Application will eventually come to require a database. Initially, one may make do with some sample data where data can be manipulated without a database, but for scaling an application, at some point, a database will have to be created.

In this section, we will discuss connecting an Express application to MongoDB using Mongoose. Mongoose can be downloaded and installed using easily available download guides online, after which this section can be started.

MongoDB With Mongoose In Express Installation

To connect Express applications to MongoDB, we will make use of an ORM to seamlessly convert data from the database to a JS application without the need of any SQL.

What is ORM? ORM is an acronym for Object Related Mapping, which is a technique used by coders to convert information between incompatible data types. ORMs basically just mimic a database to the programmer's preference, so that they can operate using their own preferred language without using the database's querying language/ The flipside to doing this is additional code abstraction, which is why many people don't like it. But if you have complex querying needs and don't know the language, this becomes essential.

Here, Mongoose will be the ORM. Mongoose gives us an easy-to-use API to deal with the MongoDB all the way to execution right from the setup. We will therefore, first install Mongoose using the command line interface of the node.js application.

```
npm install mongoose --save
```

Let us also understand a few basics before moving forward.

Database Models, Schemas and Entities

This exercise will have a database for the application to interact with. This database will have two entities: User and Message.

Database Schema: A schema explains the implementation details and tells the developers how an entity looks like as it resides within the database. Every instance of an entity is a separate row in the table. A schema tells us the field and relationships that belong to the entity. Each field is a column in the database. A schema is akin to a database blueprint.

Database Model: A model gives us the conceptual framework on what methods and manipulation tactics are available and how they are to be used to interact with the entities. Models are implemented with ORMs (in our case, it's Mongoose).

Database Entity: An entity is an instance of an item in the database created as per the schema. Each entity is a row in the database and the field of the entity is a column in the database.

Before we start with the actual code, let us first examine the relationships between the entities and how the data passes between these entities. A Unified Modelling Language or UML diagram displays clearly the relationships between entities that can be easily understood and referenced.

An example of UML diagram :

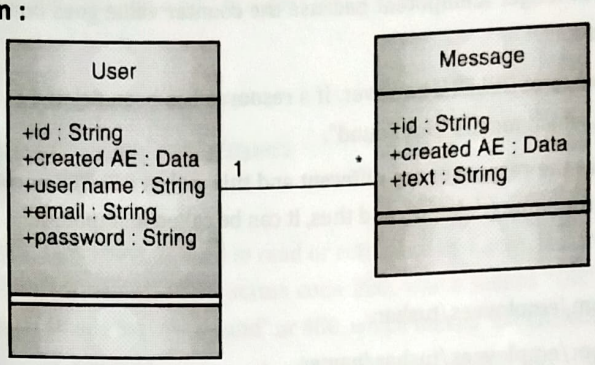


Fig. 4.7.1

User and Message above are the entities in our database. They both have their own fields and are joined by a relationship. In your application, the src/models/ folder will contain all the files for the model.

Every model has a schema that defines for the programmer the fields and the relationships. Let us say that we have two models here in the src/models/ folder with the file name model_name.js

The first is the User Model with the path - src/models/user.js

```
import mongoose from 'mongoose';

const userSchema  =  new mongoose.Schema (
 {
  username : {
   type:       String,
   unique:     true,
   required:   true,
  },
 },
 { timestamps : true },
);
const User = mongoose.model('User', userSchema);
export default User;
```

The username as seen is of the data type:string. We also have additional validation here to ensure that duplicate usernames cannot exist in the database. The keyword "unique" is the attribute that accounts for it. We also want the username to be mandatory and that's why the keyword "required" has also been used. The timestamp we added will create two new fields - createdAt and updatedAt for auditing.

Now, we can implement some methods too. In the future, say an email field is also added. We can add a method to find a user by the login term, which could be the email or even the username. We can write a method to find the user using either.

```
import mongoose from 'mongoose';


const userSchema = new mongoose.Schema(
```

```
 {
  username: {
   type :      String,
   unique :    true,
   required :  true,
  },
 },
 { timestamps : true },
);

userSchema.statics.findRecordByLogin = async function (login) {
 let requestingUser = await this.findOne({
  username: login,
 });

 if ( ! requestingUser ) {
  requestingUser = await this.findOne({ emailAddress: login });
 }
 return requestingUser ;
};

const User = mongoose.model('User', userSchema);
export default User;
```

The message model will be very similar. We will not add any custom methods and the fields will be very straightforward.

```
import mongoose from 'mongoose';

const userMessageSchema = new mongoose.Schema(
 {
  text: {
   type :      String,
   required :  true,
  },
 },
 { timestamps : true },
);

const userMessage = mongoose.model('User's Message', userMessageSchema);
export default userMessage;
```

However, we may want to associate the message with a user:

```
import mongoose from 'mongoose';

const userMessageSchema = new mongoose.Schema(
 {
  text: {
   type :      String,
   required :  true,
  },
  user: { type : mongoose.Schema.Types.ObjectId, ref: 'User' },
 },
 { timestamps: true },
);

const userMessage = mongoose.model('Message', userMessageSchema);
export default userMessage;
```

Let's say a user was deleted. We want to delete all of the messages of this user. To do this, we extend a schema with what is called a hook. We add a hook to the schema and remove all messages of a deleted user.

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema(
 {
  username: {
   type :      String,
   unique :    true,
   required :  true,
  },
 },
 { timestamps: true },
);

userSchema.statics.findByLogin = async function (login) {
 let requestingUser = await this.findOne( {
  username: login,
 } );

 if ( ! requestingUser ) {
  requestingUser = await this.findOne({ email: login });
 }
```

```
 return requestingUser ;
};

userMessageSchema.pre('remove', function(next) {
 this.model('User's Message').deleteMany( { requestingUser : this._id }, next);
} );

const requestingUser = mongoose.model('User', userMessageSchema);

export default requestingUser ;
```

Now, in the src/models/index.js file, import and combine the models and export them as a unified model interface :

```
import mongoose from 'mongoose';

import requestingUser from './user';
import userMessage from './message';

const connectToDatabase = ( ) => {
 return mongoose.connect(process.env.DATABASE_URL);
};

const models = { requestingUser , userMessage };
export { connectToDatabase };
export default models;
```

A connection function is also passed to the database with a mandatory URL parameter. We can use the environment variable over here. You will see the database URL on the command line when you start MongoDB.

Lastly, all we need is a function in the Express application to connect to the database.

```
import express from 'express';
...

import models, { connectToDatabase } from './models';

const myApplication = express( );

...

connectToDatabase ( ).then(async ( ) => {
 myApplication .listen(process.env.PORT, ( ) =>
  console.log(`This application is listening on port ${process.env.PORT} !`),
 );
} );
```

That's all! The database models have been defined for the Express application and to connect to the MongoDB when the application starts.

4.8 Testing API Endpoints

The API is the most important part of the application and must be thoroughly tested to determine functional and non-functional performance. To avoid any issues at the end of the project, we test early to ensure that the API endpoints are stable, safe and display expected behavior.

4.8.1 Functional API Endpoint Testing

Functional testing is aimed at verifying that the request, state and response of the endpoint is working as expected as per feature specifications.

Test HTTP Status Codes: verify if the correct status codes are being returned in response to different types of requests.

Test Response Objects: verify that the response objects in the JSON body are all correct and appear as expected. This involves the name & value paris, field names, error responses and so on.

Test Response Headers: verify the HTTP server headers to ensure that security and protocol level requirements are being met.

Test Happy Paths: verify that the API responses are correct for valid requests. This means checking the response for correctness.

Negative Testing: verify that the endpoint is handling invlid scenarios the right way. The input could be valid, but not acceptable, like repeat email addresses.

Destructive Testing: try to break the API and check how it reacts. Send a large amount of data or upload a massive file or work on a supremely slow internet connection to see what happens.

Combine API with UI Testing: verify if the UI can handle all kinds of unusual responses received from unusual requests.

Test Security and Authorization: verify role-based accesses, data leaks and other security protocols to ensure that they are working as intended.

4.8.2 Non-functional (Performance) Testing

Non-Functional testing of an API endpoint focuses on the speed, stability and scalability of the API. Benchmarks like response time, CPU usage, etc. are measured against.

Check Response Times : measure response times against expected benchmarks.

Check CPU Utilisation : measure CPU resources being utilized against expected benchmarks.

Check Memory Utilisation : measure how much memory is used for I/O operations against expected benchmarks.

Check Disk Time : measure for how long the disk remains busy against expected benchmarks.

4.8.3 API Endpoint Load Testing

Load testing is used to measure how well the API works when a massive number of API requests hit the server together. Will the API be able to process all of them simultaneously? Or will it crash and burn? Will some of those requests get satisfied and the rest be thrown out? Load testing answers all of these questions before they become a reality. The following are its advantages:

Performance-ready API : load testing will give you the needed control over website performance under extreme traffic conditions to ensure it can handle them when the time comes.

Evaluate Specific Servers : load testing can direct the load to specific servers to see if each server in the network is working as expected and none of them is becoming a bottleneck.

Cost and Time Effectiveness : any identified issue in load testing and save performance and maintenance efforts in the long run. And since load testing is not resource intensive, it gives you more bang for your buck!

For basic MongoDB commands that may come in handy during lab work, Please visit:

https://codingonly4u.blogspot.com/2018/06/mongodb-examples.html

https://gist.github.com/theRemix/7305403e1ab6fc8674f0

Review Questions

Q. 1 Briefly explain the essential MongoDB features. **(5 Marks)**

Q. 2 Differentiate between MongoDB and RDBMS. State 5 MongoDB data types **(10 Marks)**

Q. 3 Write a sample MongoDB document with no explanation and self-explanatory variable names that uses one array, textual information, numeric information and a nested block. Create your own example. **(10 Marks)**

Q. 4 What are the built-in roles in MongoDB for access privileges? **(5 Marks)**

Q. 5 What are the pros and cons of MongoDB? How do MongoDB and RDBMS differ? **(10 Marks)**

Q. 6 What does REST stand for? Explain REST API with a basic flow diagram. **(10 Marks)**

Q. 7 Explain the following: a) Request URLs in REST; b) Request Methods in REST. **(10 Marks)**

Q. 8 What are response and status codes? What are the three things they can have? List a few examples of status codes. **(10 Marks)**

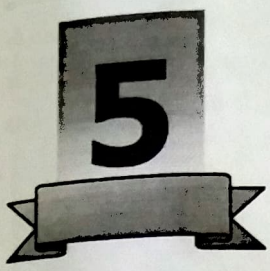
Q. 9 What are some best practices for API URLs? **(5 Marks)**

Q. 10 Explain GET, PUT and DELETE. **(10 Marks)**

Q. 11 Briefly explain the following : **(10 Marks)**

(a) Functional API Endpoint Testing (b) API Endpoint Load Testing

□□□



Flask

Syllabus

Introduction, Flask Environment Setup, App Routing, URL Building, Flask HTTP Methods, Flask Request Object, Flask cookies, File Uploading in Flask

Self-learning Topics : Flask Vs Django

5.1 Flask Introduction

Flask is a lightweight, micro web framework which provides programmers with the libraries to build and maintain web applications in Python. It is based on the WSGI (Web Server Gateway Interface) toolkit. Flask also makes use of a web template engine that makes use of data to render dynamic web pages.

So what is WSGI? The Web Server Gateway Interface is a standard for Python application development which is the universal specification for interfacing between the web server and the application.

What is a Web Framework? A web application framework or a web framework is a collection of libraries and modules within a programming language that allow programmers to write applications without having to take care of intricacies like protocol handling, security management, threading, etc. Flask is one such web application framework. Flask is a Python module that allows us to create web applications with relative ease. Flask has a small and easily extensible core: it does not have an ORM (Object Relational Manager) or any such complicated features.

Another such framework is Django, but unlike Django, Flask is very "Pythonic" and thus, has a very shallow learning curve. A simple "Hello World" program can be written with a few lines of code.

```
from flask import Flask
app = Flask ( __name__ )

@app.route ( ' / ' )

def hello_world ( ) :
  return ' Hello World ! '

if __name__ == ' __main__ ' :
  app.run ( )
```

Flask is also very explicit and easy to read. Django is not part of the syllabus, but it is deemed important in the syllabus to understand how Flask is different from Django. Flask was basically created a few years after Django when programmers started asking for more flexibility than Django was willing to provide. Django doesn't allow for as much alteration to its modules as Flask does.

Table 5.1.1

Sr. No.	Flask	Django
1	Suitable for single application development	More suitable for large scale application development
2	Uses WSGI (Web Server Gateway Interface) for less structured implementation	Full stack development framework that works off of the Model-View-Controller paradigm
3	Lightweight, open-source and needs minimum coding to get started	Needs more coding experience to master and work with
4	No support for dynamic HTML pages	Supports dynamic HTML pages
5	Supports APIs	No API support
6	Allows a large variety of databases to connect	Very limited database support

And the above reasons are why the syllabus covers Flask in some detail but not Django.

5.2 Flask Environment Setup

Python 2.7 or higher is a prerequisite to installing Flask on your system. The first step is to choose and install your Python version. How you install Python depends on your Operating system. For Windows, the simplest way is to install it using the official installer from the Python Software Foundation. Python x86 (32-bit version) is usually the better choice here because it has better compatibility with third-party packages. Since we'll be importing Flask, Flask is basically that third-party package.

The next step is having a text editor ready to write code in. If you already have one, that's great and if not, here's a few options to consider from: Sublime Text, Pycharm, Virtual Studio Code and Vim.

Installing the virtual environment is the next step in the process. Before you begin with any Python project, you will need to make a virtual environment for it. Virtual environments or "virtualenv" separate Python's dependencies from the actual operating system and ensure that any changes you make don't break some part of the OS. If you have Python 3.3 or above, you won't have to install anything more - the standard library provides virtualenv under a module named "venv". If not, you can install it via the command given below.

```
$ pip install virtualenv
```

"Virtualenv" is the virtual environment builder in Python which can be used to create more than one virtual environment simultaneously.

Install it using the following command:# Create a new virtualenv named "demoproject"

```
# Create a new virtualenv named "demoproject"

# Ideally Python 3.3+
$ python -m venv demoproject

# Python if you don't have 3.3+
$ virtualenv demoproject
New python executable in myproject/bin/python
Installing setuptools, pip, wheel...done.

# Now we activate the virtual environment in Linux
$ source demoproject/bin/activate

# Activate the virtual environment in Windows
$ demoproject\Scripts\activate
```

Now, we install Flask on the command line as shown.

```
$ pip install flask
```

Once the installation is done, we can pick up a simple code from the Flask website given below and run it to check if all is working as expected:

```
from flask import Flask
app = Flask ( __name__ )

@app.route ( " / " )
def hello ( ) :
 return " Hello World ! "

if __name__ == " __main__ " :
 app.run ( )
```

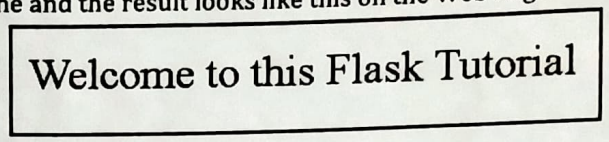
Save the above code in a file called "first_app.py" and run the Flask application with the command "python first_app.py:. If you now open the localhost in your browser, you will see the "Hello World !" response over there. On my machine, the link for localhost is "http://localhost:5200", but for most machines, it is "http://localhost:5000". Try both to see what works and use that.

5.3 First Basic Flask Application

We will build a very basic Python website using the Flask framework in this section. We can work on any text editor for this tutorial. Write the following code and save the file as basic_script.py

```
from flask import Flask

app = Flask(__name__) # this creates a Flask Class object

@app.route('/')
def home():
  return "Welcomes to this Flask Tutorial";

if __name__ =='__main__':
  app.run(debug = True)
```

Execute this file on the command line and the result looks like this on the Web Page:



This is confusing, yes, but it will become clear as you move through this chapter and understand Flask better. In the above code, to build a Python application, we need to import the Flask module and create an object of it. We pass the name of the current module (__name__) as a parameter to the Flask constructor. The route ( ) method defines the URL mapping of the function with the following syntax:

```
app.route ( rule, options )
```

Rule : represents the URL binding with the function.

Options : represents the parameters that are associated with the rule object above.

The "/" URL is bound to the main function and it returns the server's response. Here, the output is a string to be printed on the browser. Lastly, the run method of Flask runs the application on the local server.

Syntax :

```
app.run ( host, port, debug, options )
```

Table 5.3.1

Option	Explanation
host	The default host name is 127.0.0.1 (localhost)
Port	Port number on which server is listening default is 5000
debug	Default is false. Provides debug info is set to true
Options	Contains information to be forwarded to server

5.4 Flash App Routing

App Routing in Flask maps a URL to an associated function which then performs a set of functions. In the last section, the basic application we created uses the URL "/" which is associated with the home function and it returns a string that is printed on the browser.

Basically, if we visit the URL which is mapped to some function, that function's output will be rendered on the browser window.

Every web framework starts off with understanding what it means to serve required content at any URL. Routes are the URL patterns of an application like say "tusharapplication.com/home" or "tusharapplication.com/about". View, as we have learned in earlier chapters already, stands for the content that must be provided by these URLs and this could be a webpage, some string, an API response, a file, or anything else.

A simple application in Flask to display "Hello world" shows a route that is listening at the root of the application and executing a view function: "home ( )":

```
from flask import Flask
app = Flask ( __name__ )

@app.route( " / " )
def home ( ) :
  return " Hello World ! "
```

@app.route (" / ") is a Python "decorator" which is used by the web framework Flask to assign a URL in the application to a function. This decorator is directing our @app (application) that if a visitor to our application hits the home domain (tusharapplication.com) at the .route ( ), the home( ) function must be executed. Decorators are basically just logic that "wraps" around functions.

Flask even allows programmers to add variables to the URL. Programmers can reuse these variables by utilizing them as a parameter in the View.

Let us look at the following example:

```
from flask import Flask
app = Flask(__name__)

@app.route('/home/<myname>')
def home(myname):
  return "Hello,"+myname;

if __name__ =="__main__":
  app.run(debug = True)
```

Assuming that I already have a function ready at the URL which returns my name (Tushar) as a response, the following result will be printed on the browser:

Hello, Tushar

The URL can be used to display variables and a converter can be used to convert the variable into the required data type.

Let's have a look at the following example:

```
from flask import Flask
app = Flask ( __name__ )
```

```
@app.route ( '/home/<int:myage>' )
def home ( myage ):
  return " My Age is %d " %myage ;

if __name__ =="__main__":
  app.run(debug = True)
```

The output is the variable converted and displayed as an integer.

My Age is 50

The following are the converters available to us for use. They can be used to convert the default variable data type, which is string into the needed data type.

- ***string*** : default data type for variables in Flask
- ***int*** : converts the string data type into integer
- ***float*** : converts the string data type into float

Applications these days are not so static. What we have seen so far is only status rote creation. A dynamic URL with a variable can be used for dynamic routing. The variable influences the output from the URL. Consider an application for social media users that has user profiles. You want to show a user's profile based on their unique ID and if you have thousands of users, thousands of static URLs may not be the best idea. How fast would you scale? So, we create a route for each user using variables in the URL.

The dynamic parts of the URL will be marked using a < variable_name >. This will then be passed on as an argument to the view function.

```
@app.route( ' / user / <id> / ' )
def user_profile ( id ) :
  return "The profile page being displayed is of user # { }".format ( id )
```

The "id" placeholder that you see in the code above will map anything that comes after the URL "/user/". If you want to visit the profile with user 123, this is the response you'll get:

The profile page being displayed is of user #123

5.5 URL Building

Flask creates URLs using a function from the Flask package called "url_for". Hardcoding in programming is generally a last resort and almost always a very bad practice. Let's say that you wanted to change the URL structure of the posts for a user on your website from "/<id>/<your-post-titles>/" to "/<id>/post-library/<your-post-titles>/". If the URL was hardcoded, you would have to go to every template & view function and make that change. With the "url_for" function, these changes can be made instantaneously.

This function accepts the endpoint and as output, returns the URL as a string. The endpoint refers to the unique name the programmer gave the URL. Usually, it's the same name as the function.

```
#_
@app.route ( ' / ' )
def index ( ) :
  return render_template ( ' index.html ' , name = ' Tushar ' )
#_
```

To get back the root URL from the url_for ( ) function, invoke the function as url_for (' index '). The output now would be " / ".

```
#_
from filename import app
from flask import url_for
with app.test_request_context ( '/api' ):
  url_for ( 'index' )
#_
```

If the function url_for ( ) is unable to create a URL, it throws a BuildError exception.

Let's look at a bit more complicated Flask script to understand how building URLs dynamically works:

```
from flask import *

app = Flask ( __name__ )

@app.route( '/admin' )
def admin ( ) :
  return 'The current user is an admin'

@app.route ( '/teacher' )
def teacher ( ) :
  return 'The current user is a teacher'

@app.route( ' /student' )
def student ( ) :
  return 'The current user is a student'

@app.route ( ' / user / <name> ' )
def user ( name ) :
  if name == 'admin':
    return redirect ( url_for ( 'admin' ) )
  if name == 'teacher':
    return redirect ( url_for ( teacher ) )
```

```
    if name == 'student':
        return redirect ( url_for ( 'student' ) )

if __name__ == '__main__' :
    app.run ( debug = True )
```

The code above is a simulation of a very simple library management system. It accepts three kinds of users: admins, teachers and students. A function "user ( )" recognizes the kind of user and redirects them to the exact function which contains the implementation for that user.

If you the the user is localhost:5200 and you call "http://localhost:5200/user/admin", the word "admin" would be value of the variable "name" in the URL, you would be redirected to "http://localhost:5200/admin" and the output on the browser would now be:

The current user is an admin

Advantages of dynamic URL creation are :

1. No hardcoding of URLs, so in the long term, it saves implementation and maintenance effort.
2. Can handle escaping special characters and unicode symbols.
3. The paths created are absolute and there is no unexpected behavior.

5.6 Flask HTTP Methods

Hypertext Transfer Protocol or HTTP is the base from which all other transfer protocols rise up on the world wide web. Flask, like all other frameworks, provides several HTTP methods for data transfer.

The supported HTTP methods in Flask are as shown in Table 5.6.1.

Table 5.6.1

Method	Explanation
GET	Most common method used to send unencrypted data to the server
POST	Used to send form data to server. Server does not cache data received via this method
DELETE	Used to delete current representation of a resource specified in the URL

By default, all requests in Flask are handled by the GET ( ) method. If you want to utilize another, you must specify it.

5.6.1 HTTP POST Method

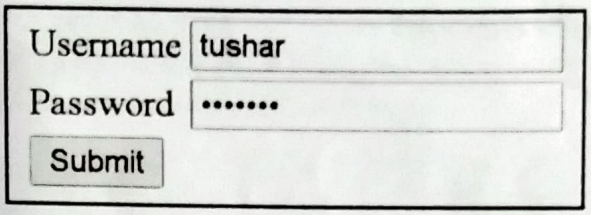
To understand how the HTTP POST method works, let us first create a form to get some kind of data from it at the client side (using the browser) and we will then access that data on the server side using the POST method.

The following is the code for a sample form page saved as "login_form.html"

```
<html>
  <body>
    <form action = "http://localhost:5200/login" method = "POST">
      <table>
```

```
    <tr> <td> Username </td>
    <td> <input type = "text" name ="username"> </td> </tr>
    <tr> <td> Password </td>
    <td> <input type = "password" name = "password"> </td> </tr>
    <tr> <td> <input type = "submit"> </td> </tr>
    </table>
   </form>
  </body>
</html>
```

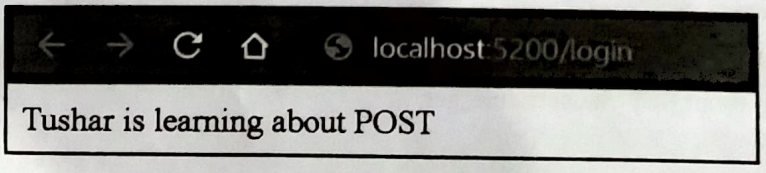
What the form looks like :



Now, we'll make use of the data from this form in the next python file saved as "learning_post.py"

```
from flask import *
app = Flask ( __name__ )

@app.route ( '/login', methods = [ 'POST' ] )
def login ( ) :
    username_post = request.form [ 'username' ]
    password_post = request.form [ 'password' ]
    if username_post == "tushar" and password_post == "agarwal":
      return "%s is learning about POST"  %username_post

if __name__ == '__main__':
  app.run ( debug = True )
```

Once you start the server by running the Flask script and then open the HTML page to run it, clicking the "Submit" button now will generate for you the following text:

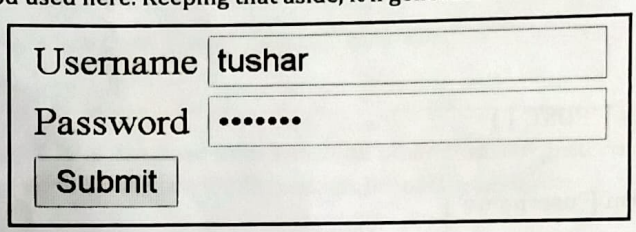


5.6.2 HTTP GET Method

We can look at the same example again to understand the HTTP GET method. There are just a few changes in the data retrieval code on the server side. The following is the code for the sample form page as was shown above saved as "login_form.html"

```
<html>
  <body>
    <form action = "http://localhost:5200/login" method = "GET">
      <table>
      <tr> <td> Username </td>
      <td> <input type = "text" name ="username"> </td> </tr>
      <tr> <td> Password </td>
      <td> <input type = "password" name = "password"> </td> </tr>
      <tr> <td> <input type = "submit"> </td> </tr>
   </table>
    </form>
  </body>
</html>
```

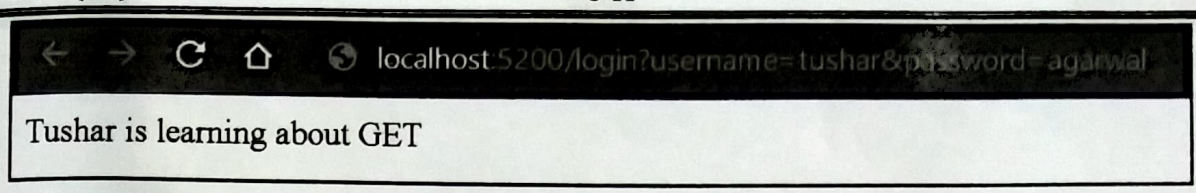
Notice the different HTTP method used here. Keeping that aside, it'll generate the same form as the earlier one.



Now, we create the python script and save it as "learning_get.py"

```
from flask import *
app = Flask ( __name__ )

@app.route ( '/login' , methods = [ 'GET' ] )
def login ( ) :
    username_get = request.args.get ( 'username' )
    password_get = request.args.get( 'pass' )
    if username_get == "tushar" and password == "agarwal":
      return "%s is learning about GET" %username_get

if __name__ == '__main__':
  app.run ( debug = True )
```

Now, once the server code is run and the HTML page is opened and the submit button is clicked, the username and password will be verified and the following text will be generated :



The data sent by the get ( ) method is retrieved on the server side using the following line:

```
username_get = request.args.get ( 'username' )
```

"Args" in this line is a dictionary object that contains a list of all the pairs of parameters and values present in the HTML form.

The one major difference between GET and POST is that in POST, the data sent to the server is not shown. The URL when you click on the submit button in POST does not show what you entered in the form while the URL in GET does show the information. This is a very simplified example of showing you how it works.

5.7 Flask Request Object

On the client side of the architecture, a request object stores all the data that needs to be sent from the client side to the server side. On the server side then, as we have already seen, the data can then be retrieved using HTTP methods.

Here, we will discuss in brief what the request object looks like and how it works. First, let's look at what the request object's most important attributes are in the Table 5.7.1.

Table 5.7.1

Attribute	Explanation
From	Dictionary object with key-value pairs of from parameters and values
Avgs	Parsed from URL. It is the part in the URL after the question mark
Cookies	Dictionary object with cookie names and values. Saved at client side to track after session
Files	Contains data related to upload file (ρ)
Method	Current request method (get) post.

Let us understand how the entire architecture works with an example.

Example : Data Retrieval from a Form

This example has three files:

An HTML file called "customer_form.html" which has a form for the customer to enter all their information. The data that is input here is POSTed to the /success URL that triggers the print_my_data ( ) in the next Flask file.

A "learning_request_object.py" Flask file with the actual method and implementation.

An HTML file by the name "print_results.html" to output the final results onto a browser.

File 1 : customer_form.html

```
<html>
  <body>
    <h4>Please fill this form for the customer !</h4>
    <form action = "http://localhost:5200/success" method = "POST">
      <p>Customer Name <input type = "text" name = "customername" /></p>
      <p>Customer Email <input type = "email" name = "customeremail" /></p>
      <p>Customer Contact Number <input type = "text" name = "customercontact" /></p>
      <p>Customer Security Pin <input type ="text" name = "customerpin" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>
```

File 2 : learning_request_object.py

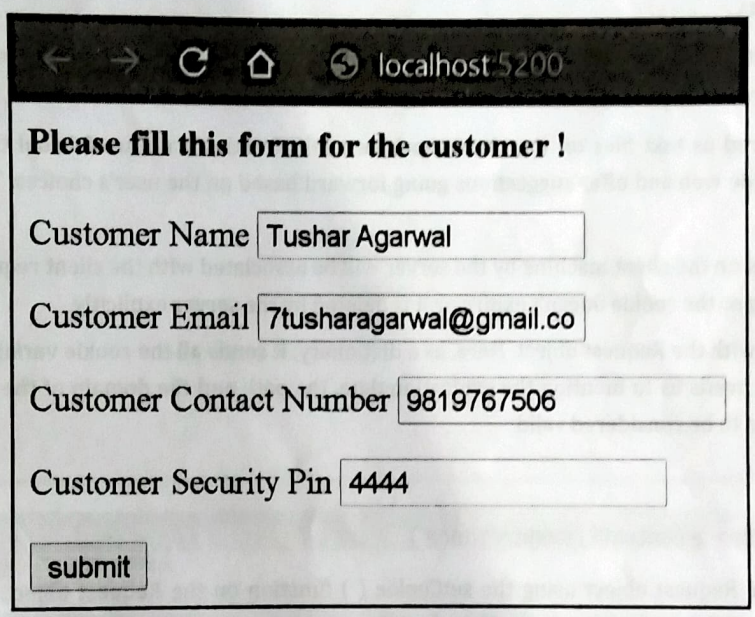
```
from flask import *
app = Flask ( __name__ )

@app.route ( ' / ' )
def mycustomer ( ) :
  return render_template ( ' customer_form.html ' )

@app.route ( '/success' , methods = [ ' POST ' , ' GET ' ] )
def print_data ( ) :
  if request.method == ' POST ' :
    customerResult = request.form
    return render_template ( " print_results.html " , result = customerResult )

if __name__ == ' __main__ ' :
  app.run ( debug = True )
```

File 3 : print_results.html

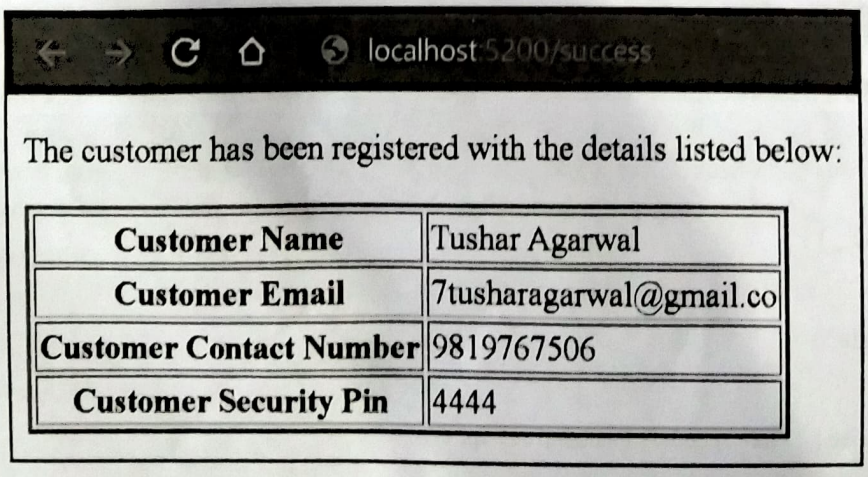
```
<!DOCTYPE html>
<html>
  <body>
    <p>The customer has been registered with the details listed below:</p>
    <table border = 2>
```

```
    {% for key, value in result.items() %}
      <tr>
        <th> { { key } } </th>
        <td> { { value } } </td>
      </tr>
    {% endfor %}
  </table>
 </body>
</html>
```

The input form looks like this:



Once the submit button is clicked and there are no errors, the output is generated as follows:



5.8 Flask Cookies

The pages we have created and seen up to this point have been quite simplistic. The browser makes a request to the server and the server responds to it. Hypertext Transfer Protocol is a stateless protocol and that means that HTTP has no way to let the server know if two requests have come in from the same user. So, the server doesn't know if you requested the same page for the first or the millionth time. How then do e-commerce and other websites give you tailored recommendations based on browsing history when you visit after the first time? The answer here lies with cookies.

What is a cookie? A cookie is essentially data about the user/requestor which the server maintains on the client machine. How does it work ?

Step 1 : The client browser makes a request to the server.

Step 2 : The server sends what is needed with one or more cookies.

Step 3 : When the client gets the response, it renders the webpage from the response and saves the cookie on the local machine (client).

Step 4 : All new requests from now on will include data from the cookie in the header until the cookie expires after which it is removed from the machine.

In Flask, cookies are stored as text files on the client machine itself. Cookies are a useful tool that can track the user's activities on the world wide web and offer suggestions going forward based on the user's choices. This enhances the user's experience.

Cookies that have been set on the client machine by the server will be associated with the client request to that server for all future requests for as long as the cookie doesn't expire or it is deleted by the server explicitly.

We associate the cookies with the Request object. Here, as a dictionary, it sends all the cookie variables and values from the client to the server. Flask needs us to mention the expiration date, the path and the domain of the website they are meant for in order for the cookies to be considered valid.

Syntax :

```
response.setCookie ( <title> , <content> , <expiry time> )
```

The cookies are set to the Request object using the setCookie ( ) function on the Request object. The response object is created using the make_response ( ) function in the View function.

We can read the cookies using the get ( ) function of the cookies attribute which is associated with the Request object.

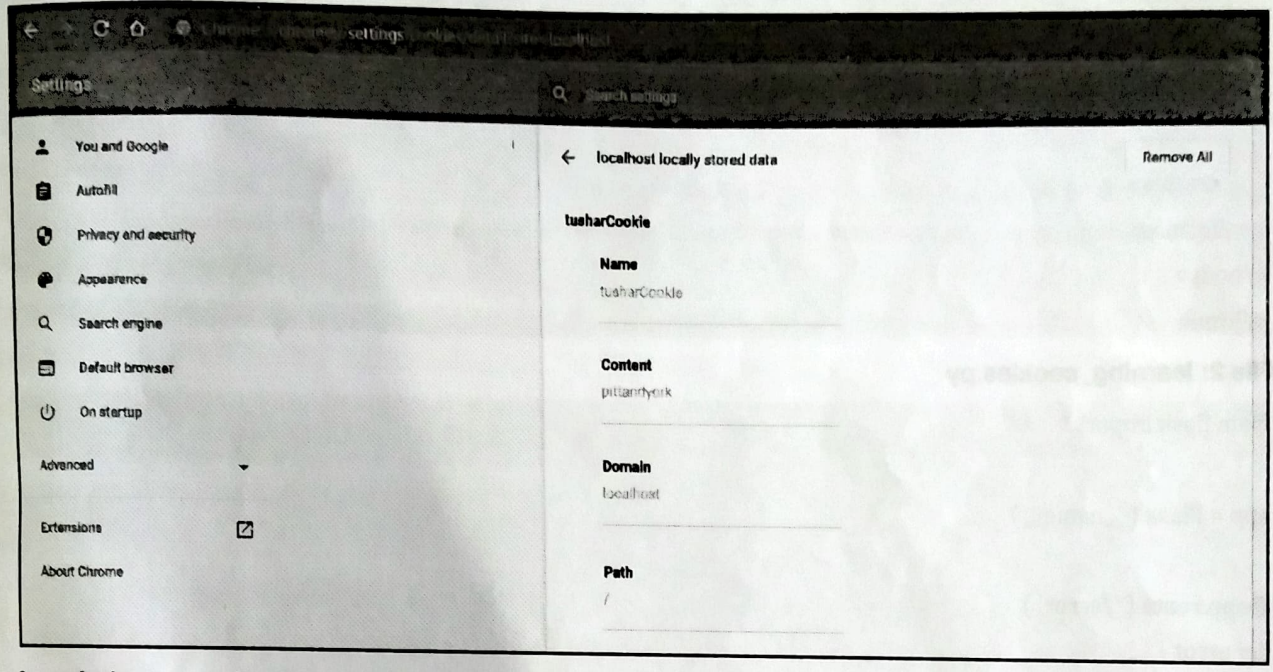
```
request.cookies.get ( <title> )
```

We will now see a simple example below.

```
from flask import *

app = Flask ( __name__ )

@app.route( ' /cookie ' )
def cookie ( ) :
  res = make_response ( " <h2> A cookie has been set up. </h2> " )
  res.set_cookie ( 'tusharCookie' , 'pittandyork' )
```

```
    return res

if _name_ == '_main_':
    app.run ( debug = True )
```

The script above will set up a cookie on the local machine (client) with the name "tusharCookie" and the content "pittandyork" for the localhost:5200 (my machine).



Let us look at a more detailed example to see how cookies in Flask work.

Example : Flask Login Application

We will create a new login application with a few different files:

An HTML file called "login_page.html" where the user is prompted to input their username and password. If the password is "Agarwal", the application will redirect the user to a success page (another HTML file). If not, the user will be directed to an error page.

The controller for this application is a Flask script saved as "learning_cookies.py" which handles the application's behavior. The has the View functions for the different cases to be handled. The username will be stored on the client side in the browser in the form of a cookie. If the correct password ("Agarwal") is entered, the application stores the username as a cookie and then reads it later on to display a message on the browser.

File 1: login_page.html

```
<html>
<head>
  <title> Login Page </title>
</head>
<body>
```

```
<form method = "post" action = "http://localhost:5200/success">
  <table>
    <tr>
      <td> Username</td> <td> <input type = 'text' name = 'username'> </td>
    </tr>
    <tr>
      <td> Password </td><td><input type = 'password' name = 'password'> </td>
    </tr>
    <tr>
      <td> <input type = "submit" value = "Submit"> </td>
    </tr>
  </table>
 </form>
</body>
</html>
```

File 2: learning_cookies.py

```
from flask import *

app = Flask ( __name__ )

@app.route ( '/error' )
def error ( ) :
  return "<p> Your password is incorrect :(( </p>"

@app.route ( ' / ' )
def login ( ) :
  return render_template ( " login_page.html " )

@app.route ( '/success', methods = [ ' POST ' ] )
def success():
  if request.method == " POST " :
    username = request.form [ ' username ' ]
    password = request.form [ ' password ' ]

  if password=="jtp":
    resp = make_response(render_template('success_page.html'))
    resp.set_cookie ( 'username', username )
    return resp
```

```
  else:
    return redirect ( url_for ( ' error ' ) )

@app.route ( '/viewprofile' )
def profile ( ) :
  username = request.cookies.get ( ' username ' )
  resp = make_response ( render_template ( ' profile_page.html ', name = username ) )
  return resp

if _name_ == "_main_":
  app.run(debug = True)
```

The code above handles both scenarios: when the password is incorrect and an error message needs to be displayed and when the password is correct and a success page must be displayed on the browser with information from the cookie.

File 3: success_page.html

This is the page you'll be redirected to in a ***success*** scenario.

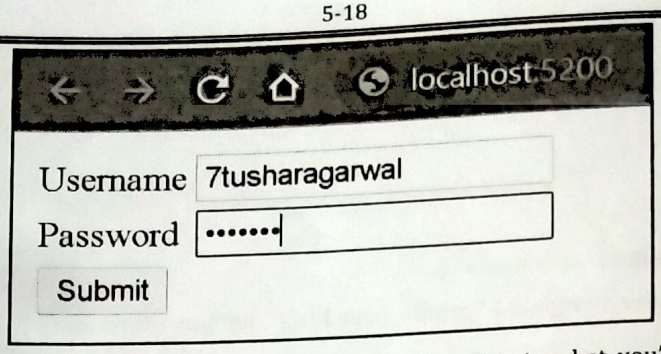
```
<html>
<head>
<title> Login Success </title>
</head>
  <body>
    <h2> You have successfully logged in ! </h2>
    <a href="/viewprofile"> View User Profile At This Link </a>
  </body>
</html>
```

File 4 : profile_page.html

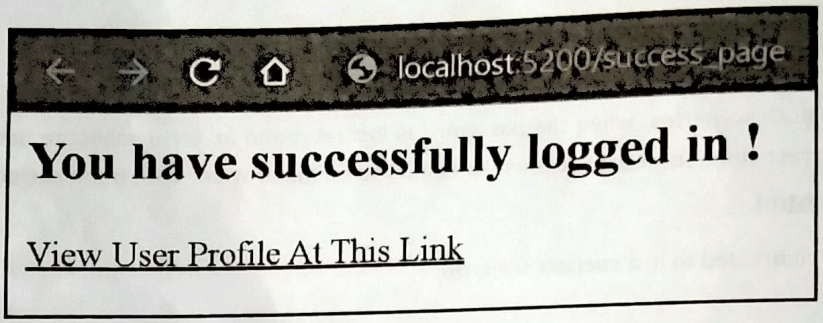
This is the page you'll be redirected to if you click on the profile link in the success scenario.

```
<html>
<head>
  <title> User Profile </title>
</head>
  <body>
    <h2> This profile belongs to {{name}} </h2>
  </body>
</html>
```

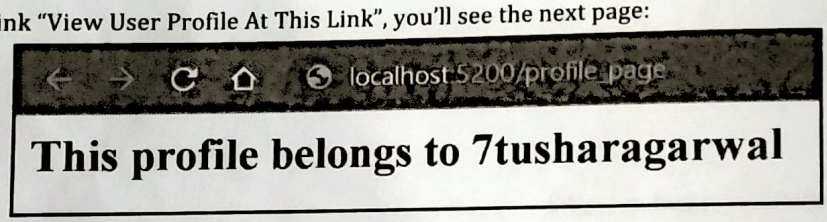
To start the server, we execute the python script first using the command "python login_page.py" and then go to "localhost:5200" (for my machine, yours may be different) on the browser as given in the next snapshot. The login page will look as shown below:



Let's assume that the right credentials were filled in and are a match. This is what you'll see next when you click on "Submit":



When you click on the link "View User Profile At This Link", you'll see the next page:



How do you explicitly delete cookies ? We call the set_cookie ( ) method with the name and value of the cookie and set a max_age parameter inside it to "0". Let's look at the code to delete the cookie we created earlier:

```
#...
@app.route ( ' /delete-cookie/ ' )
def delete_cookie():
  result = make_response("Cookie Deletion Successful !!")
  result.set_cookie('tusharCookie', 'pittandyork', max_age=0)
  return res
#...
```

5.8.1 Drawbacks of Using Cookies

Cookies have their own shortcomings too :

1. **Cookies are not a secure way to store data :** The data is visible to anyone, so using it to store any sensitive data or PII (personally identifiable information) is not advisable.
2. **Cookies can be easily disabled :** Most browsers today give their users the option to simply disable cookies. If they are disabled, the client gets no warnings or errors and any cookie sent by the server is simply discarded. Writing a JavaScript program to force it is possible, but is often not liked by the users. Who likes those pesky pop ups, right?

3. **Data storage issues :** Cookies can store only up to 4 KB of data. The browser can also impose a limit on how many cookies a server (website) can send. This limits how much cookies can achieve.

5.9 File Uploading in Flask

Flask has an easy way to handle uploading any files. It only needs an HTML form where the programmer must set the enctype attribute to "multipart/form-data". This allows posting the file to a URL and the URL handler can then get the file with a "request.files[ ]" object and save said file to the required directory.

Every file that is uploaded is saved first to a temporary location on the server before being diverted to its real location / directory.

We can define the default path for the upload folder and the maximum size of the file that can be uploaded in the configuration settings for the Flask object.

It is possible to define the path of default upload folder and maximum size of uploaded file in config settings of Flask object.

app.config [' UPLOAD_FOLDER ']

// This defines the path

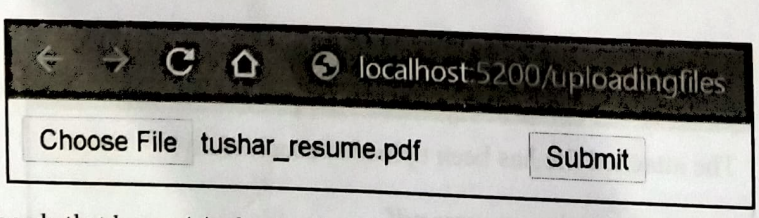
app.config [' MAX_CONTENT_PATH ']

// This defines the max size of the file

Let's understand the concept in more detail by looking first at an HTML file titled "uploadingfiles.html" :

```
<html>
  <body>
    <form action = "http://localhost:5200/uploadingfiles" method = "POST"
      enctype = "multipart/form-data">
      <input type = "file" name = "file" />
      <input type = "submit"/>
    </form>
  </body>
</html>
```

'uploadingfiles.html' here has a file chooser button and a submit button. In the output below, you'll see an input type HTML option (file chooser) to upload a file, to which I've attached my resume. If you click the "Submit" button after choosing a file, the HTML form's HTTP POST method will call the ' /upload_file ' URL which will be managed in the Flask code we will soon see.

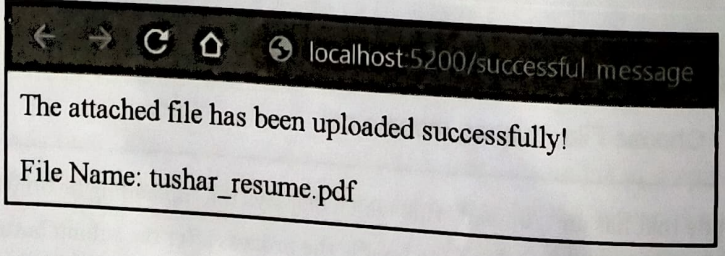


We will now see the Flask code that has an ' /upload ' rule which displays the "uploadingfile.html" page above and another ' /upload-file' rule which invoked an upload function ( ) to handle the process after the submit button has been clicked.

```
from flask import Flask, render_template, request
import secure_filename

app = Flask ( __name__ )

@app.route ( ' /upload ' )

def upload_file ( ) :
  return render_template ( ' uploadingfiles.html ' )

@app.route ( '/uploader ' , methods = [ ' GET ' , ' POST ' ] )

def upload_file ( ) :
  if request.method == ' POST ' :
    file_to_upload = request.files [ ' file ' ]
    file_to_upload .save ( secure_filename ( f.filename ) )
    return render_template("successfule_message.html", name = file_to_upload .filename)

if __name__ == ' __main__ ' :
  app.run ( debug = True )
```

If the file has been uploaded successfully, the function will return an HTML page titled "successful_message.html" which will deliver the message to the user on the browser screen. The code for that final page is as follows:

```
<html>
  <head>
    <title>Successful Message</title>
  </head>
    <body>
      <p> The attached file has been uploaded successfully! </p>
      <p> File Name: { { name } } </p>
    </body>
</html>
```



Review Questions

Q. 1 Write a brief on Flask and how it differs from Django. **(10 Marks)**

Q. 2 Explain the following: **(10 Marks)**

a) App Routing in Flask

b) Flask HTTP Methods

Q. 3 What are cookies in Flask? **(10 Marks)**

Q. 4 Explain cookies in Flask. How are they lacking? **(10 Marks)**

Q. 5 How many parameters does the route method have in Flask? Write one line about each parameter. How many data types does Flask have? State them. **(5 Marks)**

Q. 6 Write a small program in Flask to send Username and Password from an HTML page to flask server using GET or POST. Avoid writing incidental code. **(10 Marks)**

Q. 7 What is the Flask Request Object? Explain in brief. **(10 marks)**

Q. 8 What is Flask URL Building? **(5 marks)**

❑❑❑

6 Rich Internet Application

Syllabus

AJAX : Introduction and Working, Developing RIA using AJAX Techniques : CSS, HTML, DOM, XML HTTP Request, JavaScript, PHP, AJAX as REST Client Introduction to Open Source Frameworks and CMS for RIA: Django, Drupal, Joomla

Self-learning Topics : Applications of AJAX in Blogs, Wikis and RSS Feeds

6.1 AJAX Introduction and Working

AJAX stands for **A**synchronous **J**avaScript and **X**ML.

It is a technique used to create extremely fast and agile web pages that have the capability to update any part of the web page in a dynamic fashion without the need to reload the entire page.

This is possible because AJAX allows the client to asynchronously deliver and get back small packets of data. This makes it possible to update small parts of the web page without reloading the entire page.

Google Maps updating directions, Facebook updating videos and adding new comments asynchronously and YouTube updating recommended videos on the fly are all successful use cases that make use of AJAX.

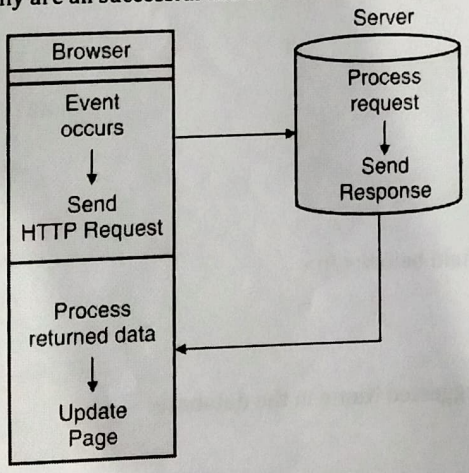


Fig. 6.1.1

AJAX uses certain well-known internet standards to accomplish all of this :

1. XMLHTTP Request Object to deliver and receive data from the server.
2. JavaScript / Document Object Model (DOM) to display information and interact with it.
3. CSS to style the web pages as per needs, and
4. XML as the data exchange format standard.

6.2 Using AJAX with HTML/DOM Client and PHP Server

The code below will make use of AJAX on the front end with HTML and a PHP function in the back end to show you how AJAX dynamically updates web pages :

File 1 : HTML file with AJAX

```
<!DOCTYPE html>
<html>
<head>
<script>
function displayClosestName ( str ) {
 if (str.length == 0) {
  document.getElementById ( " userInputCharacter " ).innerHTML = " ";
  return;
 } else {
  var requestToServer= new XMLHttpRequest();
  requestToServer.onreadystatechange = function() {
   if (this.readyState == 4 && this.status == 200) {
    document.getElementById ( " outputText " ).innerHTML = this.responseText;
   }
  }
  requestToServer.open("GET", "phpFunctionToGetSuggestedName.php?q="+inputString, true);
  requestToServer.send();
 }
}
</script>
</head>
<body>

<p>Start typing a name in the input field below:</p>
<form action="">
 <label for = " name ">
  Type Some Character to See the Suggested Name in the database:
 </label>
 <input type = " text " id = " name" name = " name "
  onkeyup = " displayClosestName ( this.value ) ">
</form>
<p>
  Possible Name from database : <span id = " outputText "> </span>
</p>
```

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

File 2 : PHP Server Program

This server contains recommendations

```
<?php
// An array of names
$namesArray [ ] = " Arpit " ;
$namesArray [ ] = "Babita " ;
$namesArray [ ] = " Chakra " ;
$namesArray [ ] = " Dhruv " ;
$namesArray [ ] = " Ekshit " ;
$namesArray [ ] = " Falguni " ;
$namesArray [ ] = " Gaurav " ;
$namesArray [ ] = " Hardik " ;

$q = $_REQUEST [ " q " ] ;

$userInput = " ";

if ($q !== " ") {
 $q = strtolower ( $q ) ;
 $len = strlen ( $q ) ;
 foreach ( $namesArray as $name ) {
  if ( stristr ( $q , substr ( $name , 0 , $len ) ) ) {
   if ( $userInput == " " ) {
    $userInput = $name;
   } else {
    $userInput .= ", $name";
   }
  }
 }
}

// Send back " No Name Found! " if no name was found in the array for the input
echo $userInput === "" ? " No Name Found! " : $userInput;
?>
```

Please note that I have only added a few names from A to H here to keep the code short. You can add as many as you would like.

This program first checks if the input field given to the user has been filled or if it is empty. If it is empty, the suggested name placeholder will be cleared.

If the field is not empty and the user has entered something :

1. An XMLHTTPRequest object will be created.
2. A function on the client has been created to be executed when something is entered. Name of function: "requestToServer"
3. A request is made to the php file. Name of file: "phpFunctionToGetSuggestedName.php"
4. You will notice a query parameter on the client side HTML page (called the "q" parameter). The string in this parameter takes the content from the input to the server.

Output :

Since the array in the PHP code only has name recommendation from A to H, if you enter any character from A to H, you will see the following:

Type a character below to see a name recommendation

Type Some Character to See the Suggested Name in the database: H

Possible Name from database : Hardik

If you change the H to A in the text box, the name recommendation will automatically change without having to reload the page. The output will be as shown :

Type a character below to see a name recommendation

Type Some Character to See the Suggested Name in the database: A

Possible Name from database : Arpit

If you type a character other than A to H, you will see the following output:

Type a character below to see a name recommendation

Type Some Character to See the Suggested Name in the database: T

Possible Name from database : No Name Found!

6.3 Developing a Rich Internet Application Using AJAX Client

A Rich Internet application (RIA) is a kind of Web application that is created to produce the same features and functionalities that are seen in regular high-powered desktop applications. RIAs conquer and divide by splitting the computing across the entire network by rendering the user interface and all other related work and computations on the client machine, and the information manipulation and other calculations on the server machine.

An RIA normally can execute inside a Web browser and in most cases, does not need additional software installation on the client machine to execute flawlessly. In some cases though, RIAs may work only with specific browsers. To ensure proper security, RIAs run the client side of the code within an isolated separate area of the client machine called a sandbox to limit visibility to the actual data on the server.

Advantages of Rich Internet Applications :

1. Offer users of the website a more rich feel, a more engrossing user experience and makes it easier to achieve their task on the site.
2. Keep in tune with rising expectations of your website's user's expectations. Everyone wants the best all the time. Amazon isn't the king of the market because of low prices, it's the king because it is fun to shop on Amazon.
3. Higher retention and loyalty rates as compared to old and slow applications.
4. Supports very computationally heavy data visualizations like dynamic charts and tables.

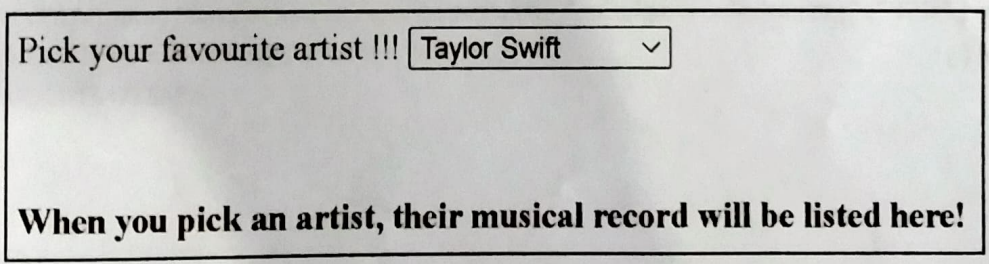
In this section, we want to understand the creation of a front end AJAX Client that uses HTML, CSS, DOM, XML, HTTP Request and Response, JavaScript and PHP

Since a lot of technologies are in play here, we will start understanding them and before you know it, you will realize that a simple application can make use of almost all of these technologies.

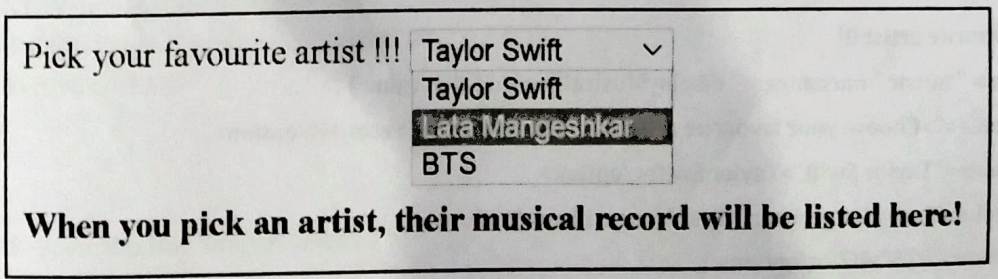
The AJAX client will make use of JavaScript to send HTTP requests using GET. The server (PHP) will load and access an XML document using a user value selected from a dropdown in the client HTML page. This XML will be sorted to get an answer based on user selection and return the HTTP response back to the client web page. CSS is just a styler for the page and can be used or removed as needed.

Interactive communication with the server can be performed using an XML file for data transport. The next example will show us how an XML file can be used to display data from the server on to the web pages. The server in this case is also PHP.

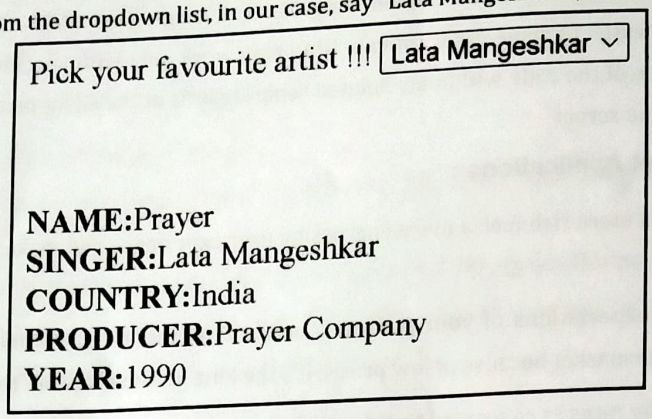
The HTML page looks like this :



When you click the on the drop down list, this is what you see :



When you select an option from the dropdown list, in our case, say "Lata Mangeshkar", this is the outpu t:



Let us now look at all the files needed to create this functionality.

File 1 : HTML Client Page

```
<html>
<head>
<script>
function displayMusicalRecord ( userString ) {
 if ( userString == " " ) {
  document.getElementById ( " formOption " ).innerHTML = " ";
  return;
 }
 var request = new XMLHttpRequest();
 request.onreadystatechange = function ( ) {
   document.getElementById ( " formOption  " ).innerHTML = this.outputRecord;
 }
 request.open ( " GET " , " getmusicalrecord.php?q=" + str , true ) ;
 request.send ( ) ;
}
</script>
</head>
<body>

<form>
Pick your favourite artist !!!
<select name = " music " onchange = " displayMusicalRecord ( this.value ) ">
  <option value="">Choose your favourite artist to get their musical record!</option>
  <option value="Taylor Swift">Taylor Swift</option>
  <option value="Lata Mangeshkar">Lata Mangeshkar</option>
  <option value="BTS">BTS</option>
```

```
</select>
</form>
<br><br>
<div id = " textForUser ">
 <b>
  When you pick an artist, their musical record will be listed here!
 </b>
</div>

</body>
</html>
```

The displayMusicalRecord created above does the following :

1. Creates an XML HTTP request object.
2. Creates a function that is invoked when an option is selected from the drop down list.
3. Send the request to the server. This is done using a query parameter just like the last example.

Before we look at the PHP server, let's see what the XML file looks like which the server will load to get the musical records.

File 2 : XML Musical Records File: "musical_records.xml"

The file can have any number of records. To keep it short, I've shown you below the three records we need for the selections above :

```
<LIST>

<RECORD>
        <NAME>Some Love Song</NAME>
        <SINGER>Taylor Swift</SINGER>
        <COUNTRY>United States of America</COUNTRY>
        <PRODUCER>Romantic Song Company</PRODUCER>
        <YEAR>2015</YEAR>
</RECORD>

<RECORD>
      <NAME>Prayer</NAME>
      <SINGER>Lata Mangeshkar</SINGER>
      <COUNTRY>India</COUNTRY>
      <PRODUCER>Prayer Company</PRODUCER>
      <YEAR>1990</YEAR>
</RECORD>

<RECORD>
      <NAME>Some Hip Hop Song</NAME>
      <SINGER>BTS</SINGER>
```

```
        <COUNTRY>South Korea</COUNTRY>
        <PRODUCER>Some Pop Company</PRODUCER>
        <YEAR>2020</YEAR>
    </RECORD>

    <RECORD>
     ...
    </RECORD>

    <RECORD>
     ...
    </RECORD>

    </LIST>
```

File 3 : PHP Server File

The script above has invoked a PHP server file called "getmusicalrecord.php". The server (PHP) will load an XML document titled "musical_records.xml" that contains information about musical records of the artists above and return that result :

```
<?php
$q = $_GET [ " q " ] ;

$xmlRecordDocument = new DOMDocument ( ) ;
$xmlRecordDocument -> load ( " musical_records.xml " ) ;

$a = $xmlRecordDocument -> getElementsByTagName ( ' SINGER ' ) ;

for ( $x = 0 ; $x <= $x -> length - 1 ; $x++ ) {

 if ( $a -> item ( $x ) -> nodeType == 1 ) {
  if ( $a -> item ( $x ) -> subNodes -> item( 0 ) -> nodeText == $q ) {
   $b = ( $a -> item( $x ) -> mainNode );
  }
 }
}

$cd = ( $b -> subNodes ) ;

for ( $i = 0 ; $i < $cd -> length ; $i++ ) {

 if ( $cd -> item( $i ) -> nodeType == 1) {
  echo ( " <b> " . $cd -> item( $i ) -> nodeName . ":</b> ");
  echo ( $cd -> item( $i ) -> subNodes -> item( 0 ) -> nodeText );
```

```
  echo ( " <br> " ) ;
 }
}
?>
```

When the query is sent from the client script over to the server (PHP), the following steps take place in order:

1. An XML DOM object is created to be able to access the XML Document.
2. Find all the <SINGER> elements that match the name sent from the client.
3. Open the musical record information and it to the "textForUser" placeholder on the client side.

You will see that the files above cover all of the technologies outlined for the creation of a rich internet application except CSS. CSS, as a styling tool, does not add any functionality and so, adding it from the beginning itself would increase the length of the code and create further confusion. Assuming that you know how CSS works, this book will not be detailing and teaching you how CSS works, but simply explain what an added CSS style block would do and make a very, very simple style block just to showcase how CSS can be combined with the above application.

Let us take the File 1 (HTML Client page) and add a simple style block with a header in a specific color and a new paragraph with some text. Any and all modifications to this block can be made. Tables with padding, formatting, colors, highlights and text, forms with more complicated styling features, borders, etc. can all be added to this block.

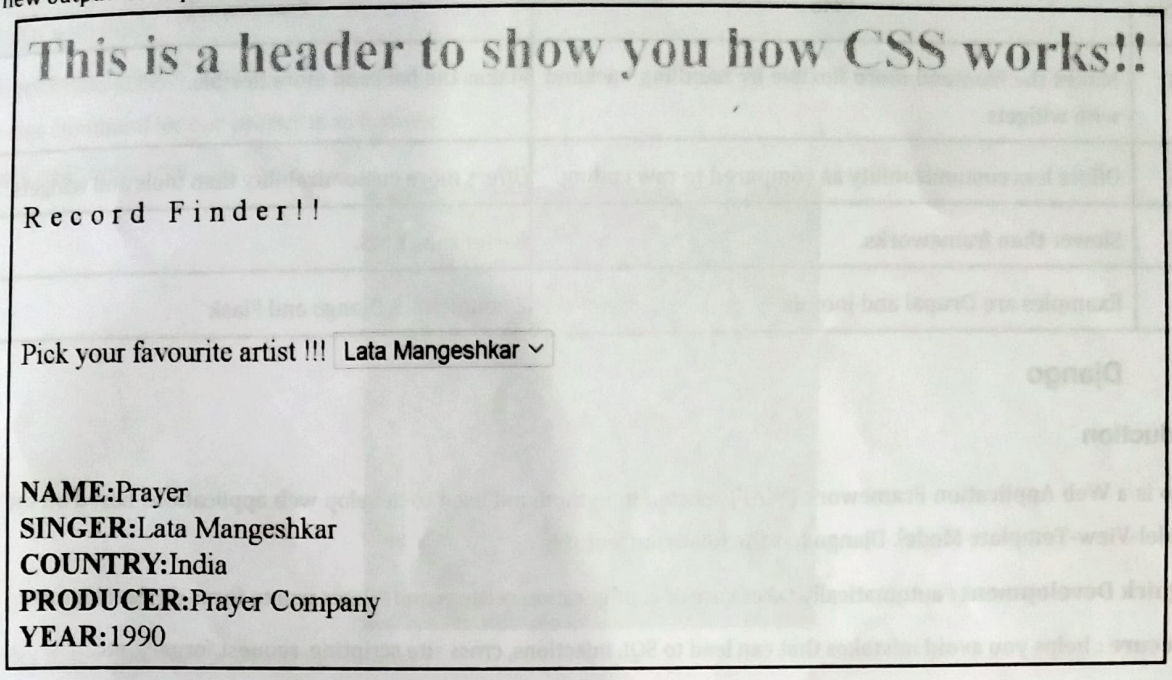
However, the syllabus does not cover the details of CSS. It simply covers a student's understanding of how CSS can be added to an application.

New File 1 with CSS : HTML Client Page

```
<html>
<head>

<style>
div {
 border: 2px green
 padding: 4px;
}

h1 {
 text-align: left
 text-transform: uppercase;
 color: #4DBF60;
}

p {
 letter-spacing: 5px;
}

a {
 text-decoration: none;
```

```
 color: #019ACB;
}
</style>

<script>
function displayMusicalRecord ( userString ) {
 if ( userString == " " ) {
  document.getElementById ( " formOption " ).innerHTML = " ";
  return;
 }
 var request = new XMLHttpRequest();
 request.onreadystatechange = function ( ) {
   document.getElementById ( " formOption  " ).innerHTML = this.outputRecord;
 }
 request.open ( " GET " , " getmusicalrecord.php?q=" + str , true ) ;
 request.send ( ) ;
}
</script>
</head>
<body>

<form>
Pick your favourite artist !!!
<select name = " music " onchange = " displayMusicalRecord ( this.value ) ">
   <option value="">Choose your favourite artist to get their musical record!</option>
   <option value="Taylor Swift">Taylor Swift</option>
   <option value="Lata Mangeshkar">Lata Mangeshkar</option>
   <option value="BTS">BTS</option>
</select>
</form>
<br><br>
<div id = " textForUser ">
 <b>
   When you pick an artist, their musical record will be listed here!
 </b>
</div>

</body>
</html>
```

The new output now upon selecting Lata Mangeshkar will be as shown below:



I reiterate once again that the syllabus does not cover details of CSS, just a rudimentary understanding of what CSS is and how it can be added to the AJAX Client side and used in conjunction with other AJAX techniques.

6.4 Introduction to Open Source Frameworks and CMS for a Rich Internet Application

Under this section, we'll be studying the basics of a few different technologies that help developers create richer and more dynamic web applications. The syllabus covers the basics of these technologies only and not an in-depth study, so being able to create and run basic applications & answering fundamental questions about them is part of the syllabus, but the nitty-gritties of these technologies are not.

A CMS or a content management system, is a tool or software that assists the users of a website in making, managing, and updating any content on the website without having to know advanced technical know-how. Basically, you can create a website using CMS even if you don't know the code from scratch. The CMS will handle all basic infrastructure and the backend for a website so you can focus on what the website needs to look like.

So how is it different from a framework like Flask or Django? A framework gives us an easy way to create and deploy applications by providing the common pieces of code and allowing software developers to modify and customize according to their needs. Frameworks help improve code reusability, but need a basic amount of technical expertise to create from scratch.

Table 6.4.1 : Difference between CMS and Framework

Sr. No.	CMS	Framework
1	Can create and modify digital content seamlessly.	A generic set of functionality that can be modified as per needs.
2	Easy to learn with little to no technical know-how.	Needs more time to learn with some prerequisite technical know-how.

Sr. No.	CMS	Framework
3	Makes the frontend more flexible by handling backend with widgets.	Makes the backend more flexible.
4	Offers less customizability as compared to raw coding.	Offers more customizability than tools and widgets.
5	Slower than frameworks.	Faster than CMS.
6	Examples are Drupal and Joomla	Examples are Django and Flask

6.4.1 Django

Introduction

Django is a Web Application Framework (WAF) written in Python and used to develop web applications based on the MVT or Model-View-Template Model. Django has the following features :

1. **Quick Development :** automatically takes care of configuration settings and allows you to focus on development.
2. **Secure :** helps you avoid mistakes that can lead to SQL injections, cross site scripting, request forgery, etc.
3. **Scalable :** easy context switching makes Django great at scaling from small to large applications.
4. **Feature Heavy :** a large collection of libraries to handle everything from authentication to content administration and everything in between.
5. **Versatile :** can build applications for any domain.
6. **Open Source :** public software available for free.
7. **Vast Community :** lots of help available online.

Django Installation

To install Django, just go to the official Django website (www.djangoproject.com) and download the Django executable file.

After that, Django needs "pip" to initiate installation. "Pip" is a system that manages data packages and is used to install, configure and manage data packages written in python. "Pip3" is the newer version, so use the latest that is compatible with your Python version.

The installation command on the command line for my Django version is as follows :

```
$ install django == 2.0.3
```

You can use the following command to make sure it got installed as expected:

```
print ( django.get_version ( ) )
```

First Project

We can create a project in the /home directory using the following command:

```
$ django-admin startproject tusharsProject
```

Go to your project using "cd":

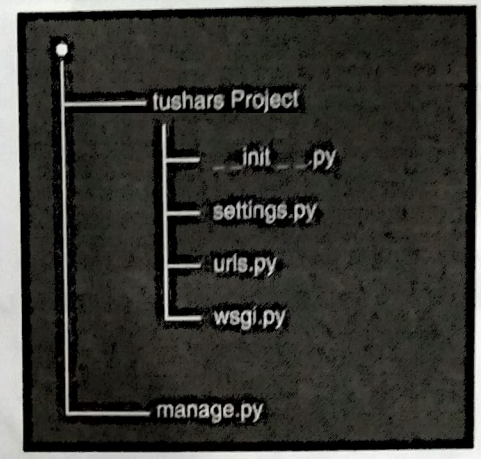
```
cd tusharsProject
```

To go through all the files and folders in a Django project, use the tree command to see the structure. If you don't have it, install it with this:

apt-get install tree

The tree command for our project is as follows:

tusharsProject tree



The files in the directory above are as follows :

- **manage.py:** command line utility to interact with and manage the application.
- **tusharsProject:** this is the module we will import. It is the actual application package.
- **__ init __.py:** empty file to tell Python to consider this directory as a Python package.
- **settings.py:** configure application settings like database connections and static linking.
- **urls.py:** contains all listed URLs in the application. We write the URLs here with the corresponding action.
- **wsgi.py:** all WSGI-compatible web servers go through here.

These are all the default files already present in any new application.

When you run the application using the following command, you will see the local port at which the server started. So, when you run . . .

$ python3 manage.py runserver

. . . you will see something like this:

http://127.0.0.2 : 8200/

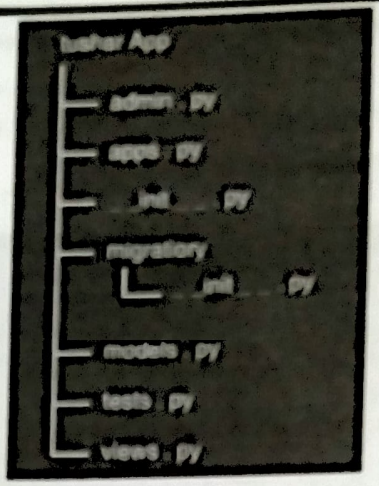
This means that the server started at port 8200. If you go to localhost: 8200 on your browser, you will see *"Success"*. Meaning the server connection has been initiated.

Once a project has been created, we will create an application inside the project. Django will create the base directories for it automatically, so we can focus on the code.

A Django project is a bunch of configuration files and an application is a piece of programming meant to perform a certain task under the configurations of the project it has been created within.

To create a Django application within *tusharsProject*, use the following command:

$ python manage.py startapp tusharApp



Right above is the directory structure for the app. Initially, all these files are empty. Let's write some code for it to see how execution works.

File 1 : "views.py"

```
from django.shortcuts import render
# We make our views here in this file
from django.http import HttpResponse
def printsomething ( request ) :
   return HttpResponse ( " <h1> Hi, my name is Tushar! </h1> " )
```

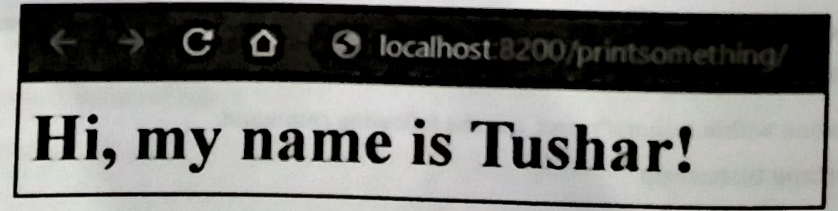
File 2 : "urls.py"

```
from django.contrib import admin
from django.urls import path
from tusharApp import views
urlpatterns = [
  path ( ' admin/ ' , admin.site.urls),
  path ( ' hello/ ' , views.printsomething ),
]
```

Now, we run the server with the following command

```
python manage.py runserver
```

Now, you open your port on the browser and navigate to the link "localhost:8200/printsomething /" to see the following output:



Django Model-View-Template

The MVT in Django is a software design pattern where the model handles access to the data, the template is a presentation layer to look after the UI and the View is used to perform business logic operations as needed and get data from the model and render a template for the user. Control is handled by Django itself. The end user sends a request to Django, which works as a controller to check for what is needed in the URL. If the URL maps successfully, a View is called that gets the data from the Model and then renders the Template. Django then responds back to the user and sends the template as a response.

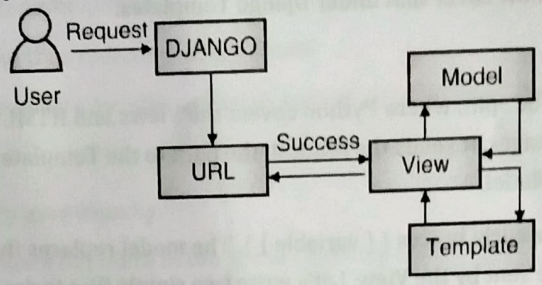


Fig. 6.4.1

Django Model

The Model in Django is a Class that hosts the fields and methods. Each Model maps to one table in the database. Each field of the Model's class is a column in the table. Django gives us the ability to create, interact with, retrieve, update and delete via the Model. Models are defined in the "models.py" file. This one file holds all needed models.

Let's create a model called "Employee" in the "models.py" file with the following code:

```
class Employee ( models.Model ) :
  first_name = models.CharField ( max_length = 50 )
  last_name = models.CharField ( max_length = 50 )
  phone_number = models.IntegerField ( )
  email_address = models.EmailField ( max_length = 25 )
  age = models.IntegerField ( )
```

To commit it, run the following command. This is called migration in Django.

```
python manage.py makemigrations tusharsApp
```

This will create a table internally called "tusharsapp_employee".

Django View

Views are a key part of Django's framework. Views get response objects from the server, update those objects if necessary, render data, return HTML pages, etc. We must connect a View to a URL in order to see the web page.

Let me now create a sample View in "tusharsApp" to say " Hi, my name is Tushar! "

File 1 : "views.py"

```
from django.http import HttpResponse
def printsomething( request ) :
  text = """<h1> Hi, my name is Tushar! </h1>"""
  return HttpResponse ( text )
```

Here, we will use the HTML response to render a web page which has been hardcoded inside the HTML header tag.

Django supports the MVT model however, so it should ideally be like this where "printsomething.html" is the Template :

```
from django.shortcuts import render
def printsomething ( request ) :
 return render ( request, " tusharsApp/template/printsomething.html ", { } )
```

What is the "render" function? Let's now cover that under Django Templates.

Django Template

With Django, HTML and Python can be split, where Python covers the Views and HTML covers the Templates. Django relies on "render" to connect the two languages. It sends the request, the path to the Template and a dictionary of parameters that has all the needed variables for the Model.

A variable is written between double curly braces { { variable } }. The model replaces the variable in the third parameter of the render function with the variable sent by the View. Let's write two simple files to see how Templates work:

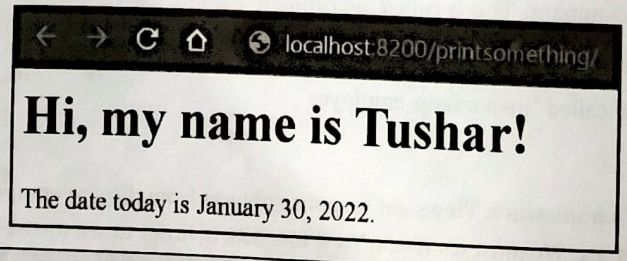
File 1 : "views.py"

```
from django.shortcuts import render
def renderFunction ( request ):
  today = datetime.datetime.now ( ) . date( )
  return render ( request , " printsomething.htm l ", { " today " : today } )
```

File 2 : "printsomething.html"

```
<html>
 <body>
  Hi, my name is Tushar!! <p>The date today is { { today } }. </p>
 </body>
</html>
```

Output :



Note from Author : This section is insufficient to teach you the details of how Django works, but has been deemed sufficient as per University syllabus and thus, this section will not go into further detail and confuse you. A rudimentary understanding of Django is sufficient as per syllabus.

6.4.2 Drupal

Drupal is an open source CMS or Content Management System that helps programmers organize and publish content on the world wide web. It is built on PHP and requires an understanding of HTML and CSS before you proceed. Both are considered prerequisites before studying this subject as per syllabus.

A CMS is a piece of software that stores data of all types (text, images, audio, video, documents. etc.) and makes it available and accessible to a website for use.

Advantages of Drupal

1. Flexible in allowing all kinds of content to be stored and used, irrespective of whether the content is structured or unstructured.
2. Provides ready templates for web development.
3. Makes it easy to reuse content and thus reduces storage needs.
4. Has almost a thousand plugins to boost your site and being open source, you can also create your own.

Disadvantages of Drupal

1. The UI (user interface) is not very user friendly.
2. Not compatible with a lot of software out there because it is relatively new.
3. Lower performance compared to other content management systems.

Download Drupal

Step 1 : Download Drupal's latest version of the available zip files from the official website as per your computer's specifications and run the executable file: *www.drupal.org/project/drupal*

Step 2 : Drupal requires a MySQL database. Have an SQL database ready for connection. Let's say that the username is "root" and so is the password.

Step 3 : In my case, my Drupal path is localhost/drupal_folder_name. Keep this link.

Step 4 : On the database configuration page of the installer, enter MySQL for type of database and add other details as needed.

Step 5 : Add other information like site name, email address, username, password, etc. and when installation is done, you will see a link to visit your new site which will be like an empty placeholder for a website. This will be the Drupal homepage.

Drupal Architecture :

Drupal's architecture has the following layers to it:

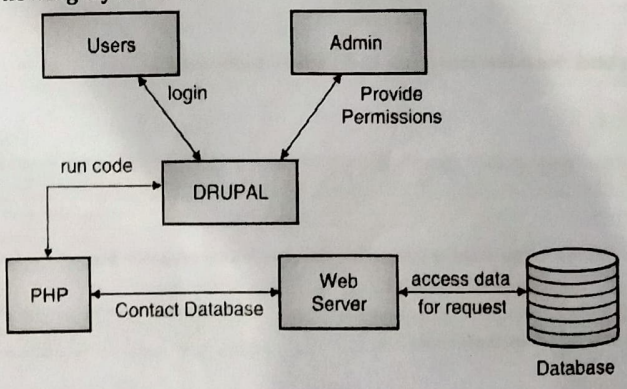


Fig. 6.4.2

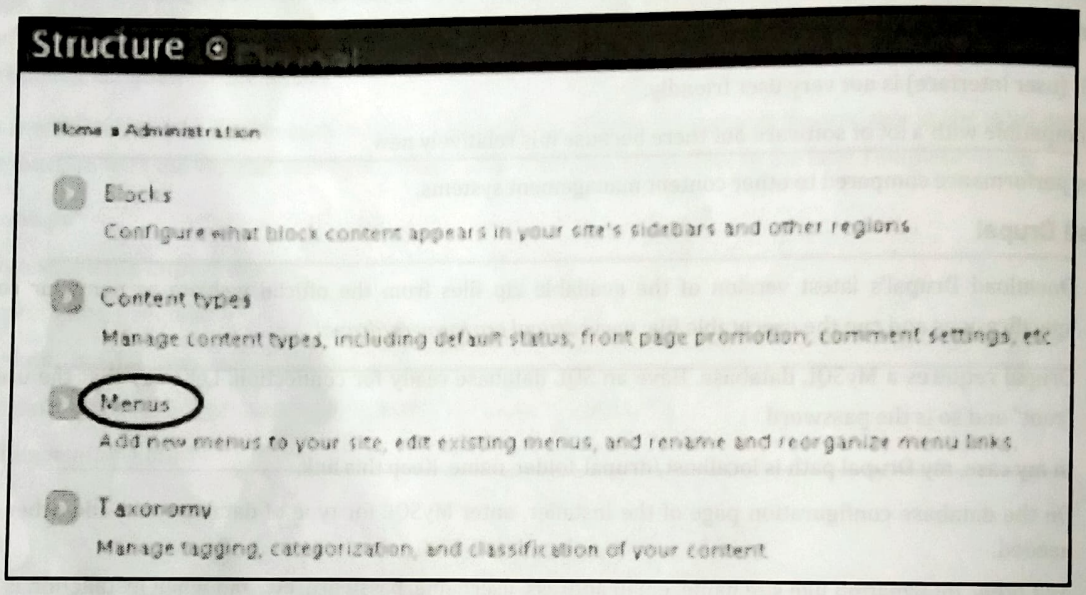
- **Users** : The actual users of Drupal who send a request and expect a response - the client side.
- **Administrator** : Provides access and authorization by managing privileges for monitoring content.
- **Drupal** : The free open source CMS that works on a PHP environment to create powerful and complex websites.
- **PHP** : The scripting language used to fetch data from the server, i.e. database.

- **Web Server** : Processes requests via HTTP and serves as the communication platform between the client and the server.
- **Database** : Stores all the information and allows modification, updates and deletion.

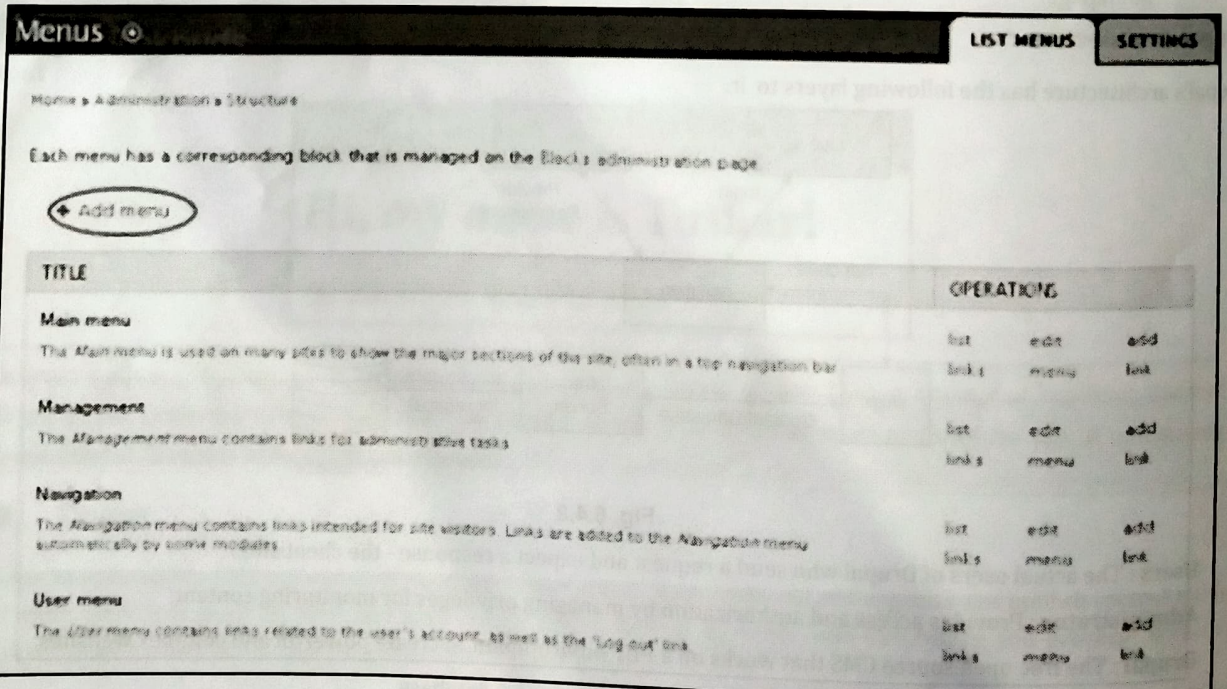
Drupal Menus

Let us now learn how to create a basic menu in Drupal. Menus are necessary to navigate a website by providing links within. Follow these simple steps:

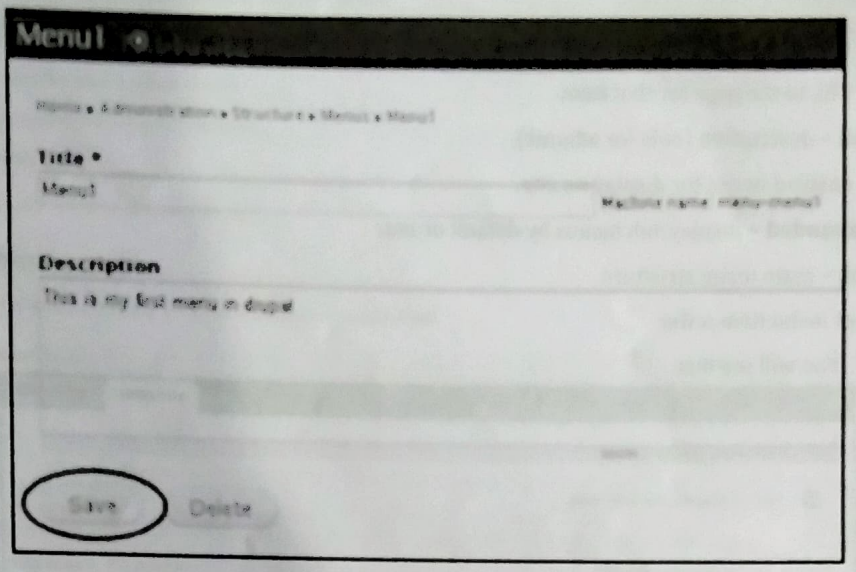
Click on "Structure" and then "Menus" as shown :



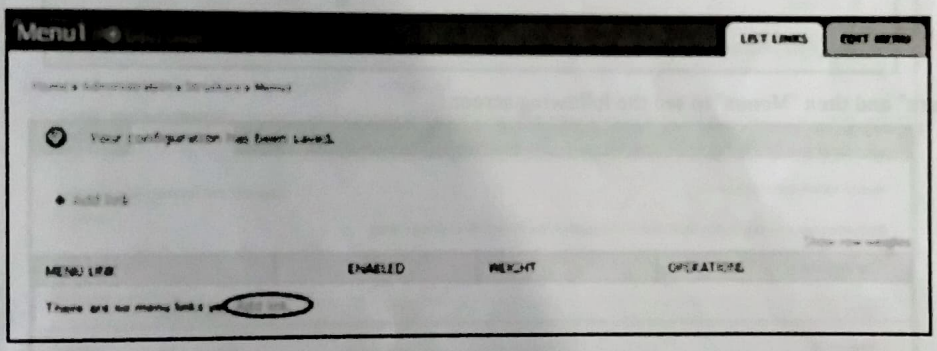
Now, click on the "Add" menu option :



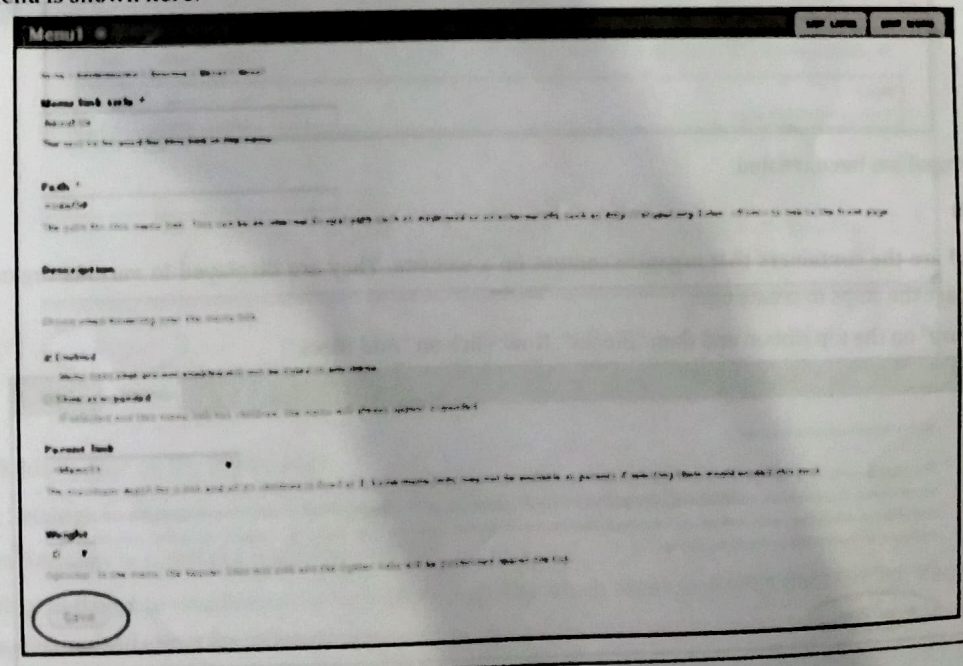
Fill your details :



Fill the title and description for the menu and click on "Save". Once you click on "Save", the following screen will be shown to you. Now, define the path link to the new menu page. Click on "Add link" as shown below:



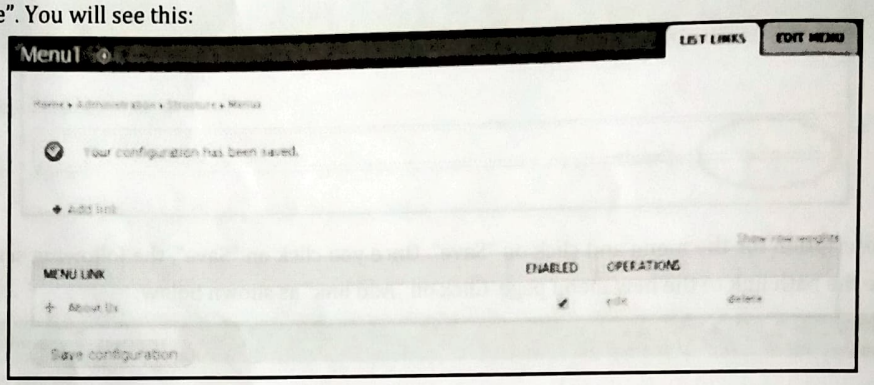
The created Menu is shown here:



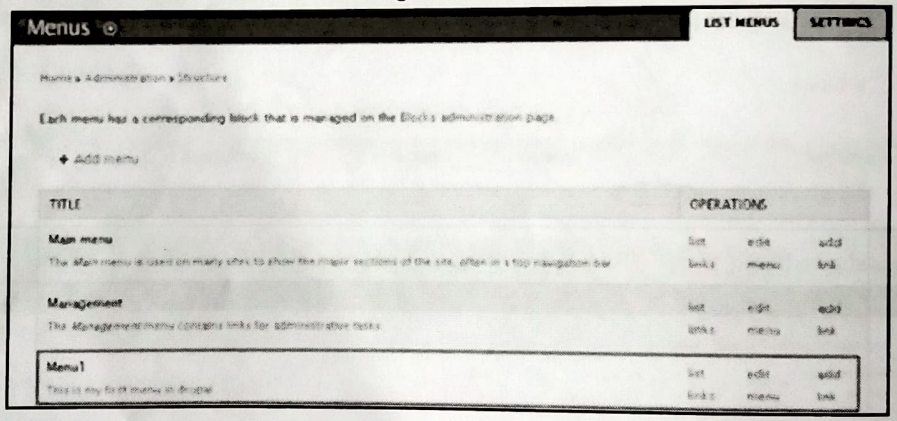
It has the following fields :

- **Menu link title**– name of a menu item.
- **Path** – the URL to the page for that item.
- **Description** – description (only for admins).
- **Enabled** – enabled or not for display on site.
- **Show as expanded** – display sub menus by default or not.
- **Parent Link** – main menu structure.
- **Weight** – set menu item order.

Now, click on "Save". You will see this:



Click on "Structure" and then "Menus" to see the following screen:

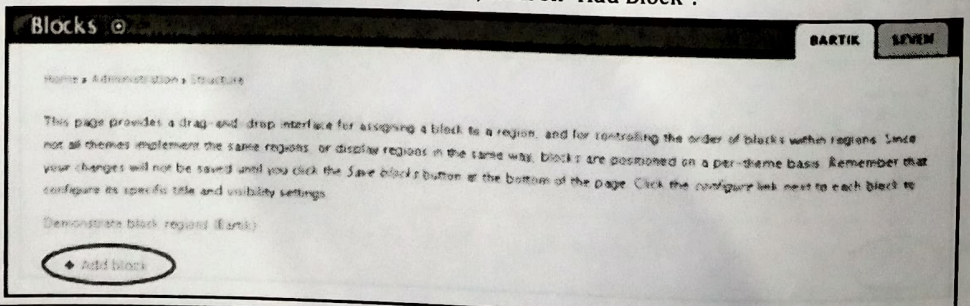


First Menu in Drupal has been created.

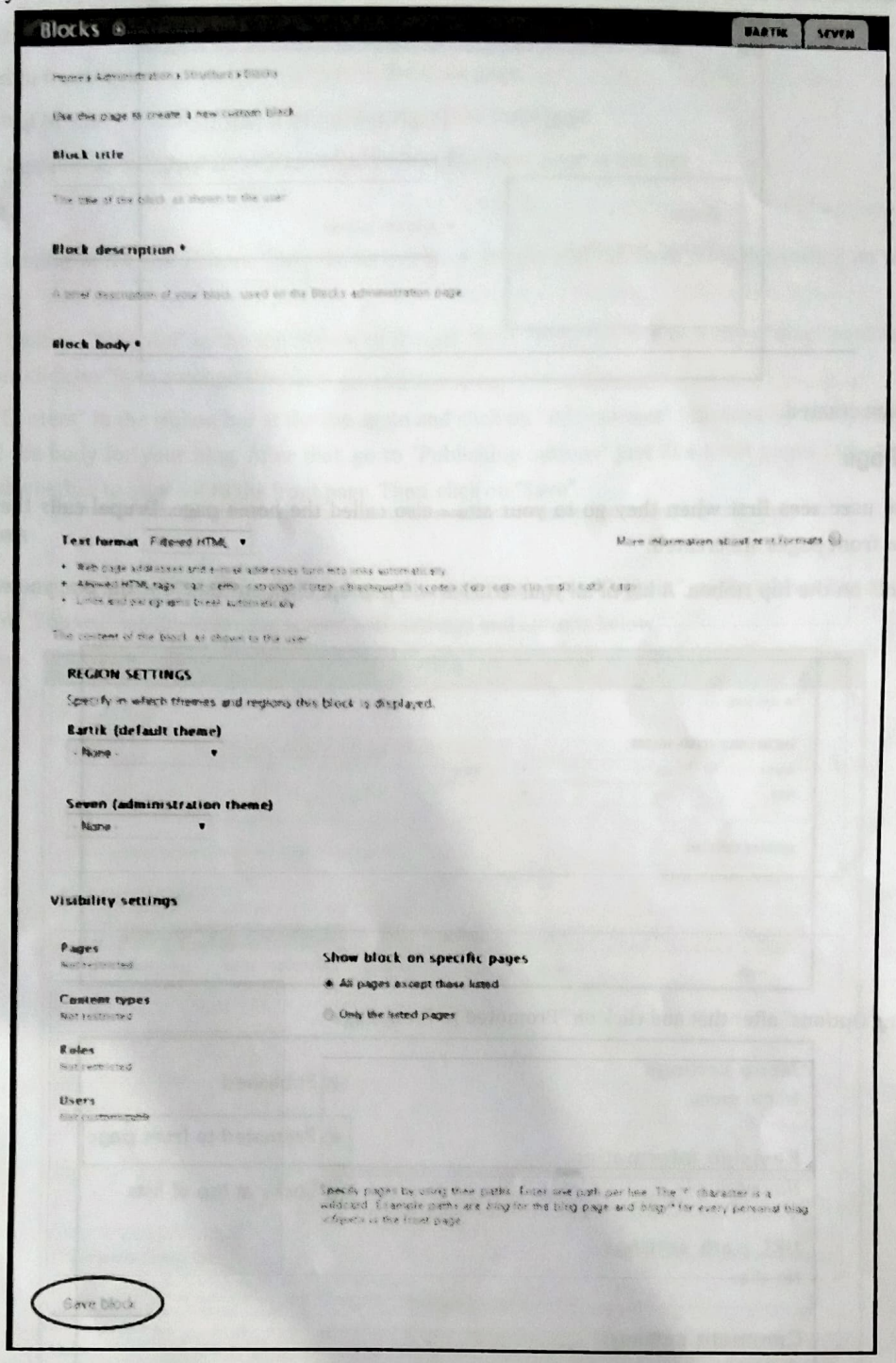
Drupal Blocks

Blocks in Drupal are the containers that organize content on a website. They are displayed in various regions of the web page. Following are the steps to create one:

Clink on "Structure" on the top ribbon and then "Blocks". Now, click on "Add Block":



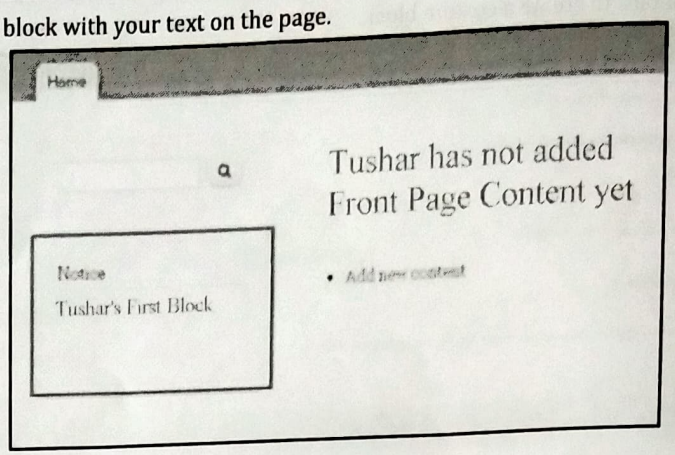
A page is displayed to fill in details to create a custom block.



The following fields appear on the Blocks page.

- **Block Settings** to enter the title, description, body text and format for the text.
- **Region Settings** to select the region for display
- **Visibility Settings** to decide more advanced settings like which pages to display the block on, who can access the blocks, customized views for different users, etc.

Now, click on "Save" to see a block with your text on the page.

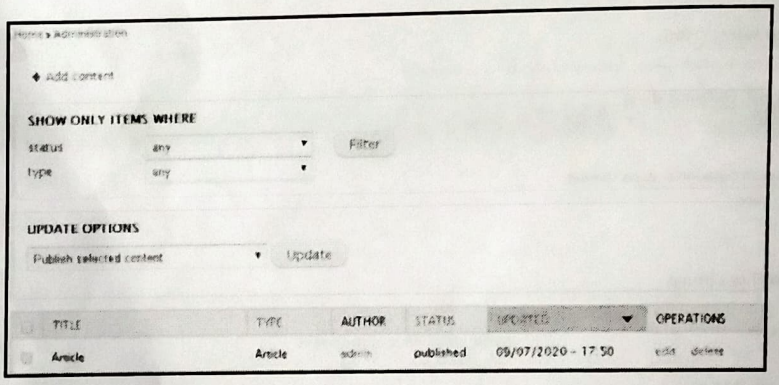


The block has been created.

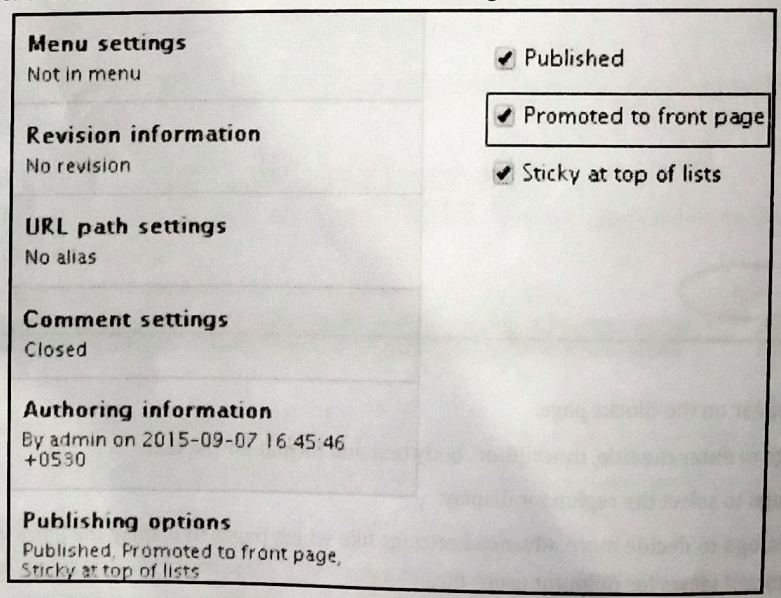
Drupal Front Page

This is what the user sees first when they go to your site - also called the home page. Drupal calls them front pages. Following is how front pages are created:

Click on "Content" on the top ribbon. A list of all your articles will pop up. Click on "Edit" for the one you want for a front page.



Go to "Publishing Options" after that and click on "Promoted to Front Page".



Available options :

- **Published** – makes the article live
- **Promoted to front page** – attaches the article to the front page.
- **Sticky at top of lists** – attaches your article to the top of the front page.

If you check all three options, the article will now be listed on the front page at the top:

Drupal Blogs

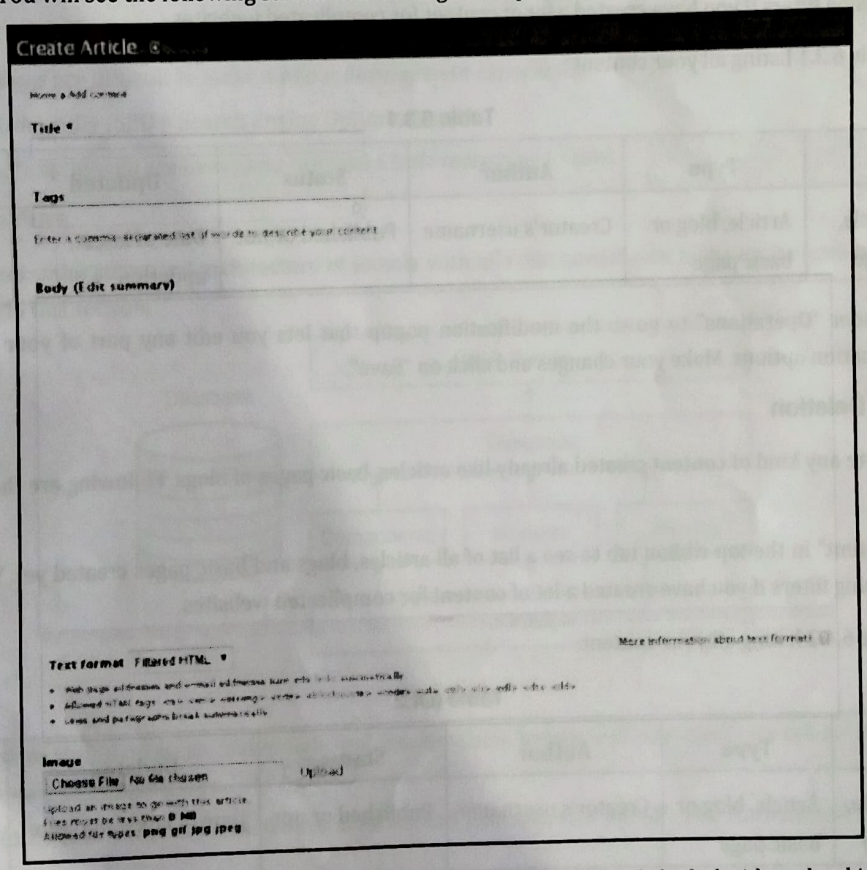
Blogs help get content to the site visitors. They can be public or private and can have roles depending on who is allowed to access them.

To create one, click on "Modules" in the top ribbon of Drupal. Now, check the box that says "Blog" next to it to install the module and then click on "Save configuration".

Now, click on "Content" in the ribbon bar at the top again and click on "Add content" followed by "Blog entry". You can now add a title and the body for your blog. After that, go to "Publishing options" just like front pages and choose whether to make it live and whether to attach it to the front page. Then, click on "Save".

Drupal Articles

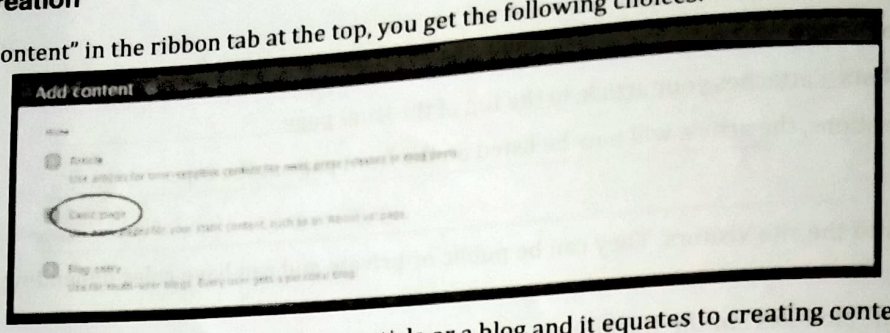
Articles help site visitors know more about the website. To create one, go to "Content" on the ribbon bar at the top and click on "Add content". You will see the following screen with settings and options below.



You can add the title, any relevant tags, the body of the article, the format for it (which decides what kind of text goes in - normal text or HTML if you want to design the text yourself) and any image if you want one. After that, go to "Publishing options" at the bottom just like front pages and choose whether to make it live and whether to attach it to the front page. Then, click on "Save".

Drupal Content Creation

When you click on "Content" in the ribbon tab at the top, you get the following choices:



In Drupal, creating a basic page is just like creating any article or a blog and it equates to creating content.

You enter the title, body, text format and choose your publishing options and finally click on "Save" and create content for your website.

Drupal Content Modification

Drupal lets us **modify** any kind of content created already like articles, basic pages of blogs.

Following are the simple steps to achieve this:

First, click on "Content" in the top ribbon tab to see a list of all articles, blogs and basic pages created yet. You also have an option to search using filters if you have created a lot of content for complicated websites.

You will see a Table 6.3.1 listing all your content :

Table 6.3.1

Title	Type	Author	Status	Updated	Operations
Name of the article, blog or basic page	Article, blog or basic page	Creator's username	Published or not	Date of latest updation	*Edit* *Delete*

Click on "Edit" under "Operations" to go to the modification popup that lets you edit any part of your content or their settings and publication options. Make your changes and click on "Save".

Drupal Content Deletion

Drupal lets us delete any kind of content created already like articles, basic pages of blogs. Following are the simple steps to achieve this:

First, click on "Content" in the top ribbon tab to see a list of all articles, blogs and basic pages created yet. You also have an option to search using filters if you have created a lot of content for complicated websites.

You will see a Table 6.4.2 listing all your content:

Table 6.4.2

Title	Type	Author	Status	Updated	Operations
Name of the article, blog or basic page	Article, blog or basic page	Creator's username	Published or not	Date of latest updation	*Edit* *Delete*

Click on "Delete" under "Operations" to go to the modification popup that lets you delete your selected content after a second confirmation. A notification popup in Green will appear stating that the content you deleted has been deleted successfully.

Note from Author : This section is insufficient to teach you the details of how Drupal works, but has been deemed sufficient as per University syllabus and thus, this section will not go into further detail and confuse you. A rudimentary understanding of Drupal is sufficient as per syllabus.

6.4.3 Joomla

Joomla is an open-source, content management system that can build complex websites and online applications. It is built on an MVC or Model-View-Controller framework. It is based on PHP and MySQL, and the templates can support HTML and CSS, so knowledge of these as prerequisites will make understanding Joomla easier.

Advantages of Joomla

- Free and openly sourced.
- Simple to install and set up.
- Uses the What-You-See-Is-What-You-Get (WYSIWYG) Editor, so content editing is easier.
- Has almost universal browser support.
- Provides free and easy to use templates for quick deadlines.
- Can work with almost any database.

Disadvantages of Joomla

- Has internal stability and compatibility issues when multiple modules and plugins are installed and run together.
- Complex designs are difficult to make without development experience.
- Not very SEO-friendly (SEO = Search Engine Optimization).
- Websites made by Joomla are heavy and can take a little more time to load.

Joomla Architecture

The Fig. 6.4.3 shows the structural architecture of Joomla with all components that make up its framework. They will be explained briefly in this section.

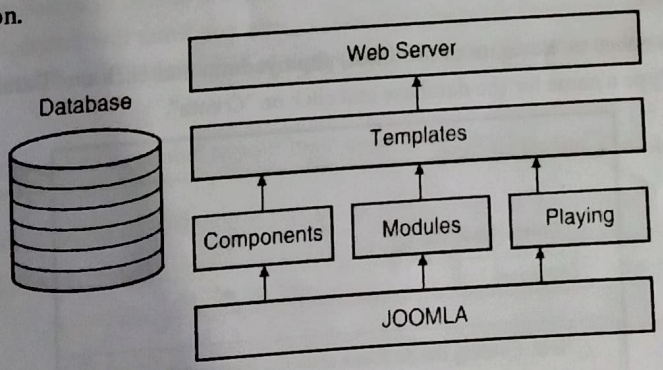


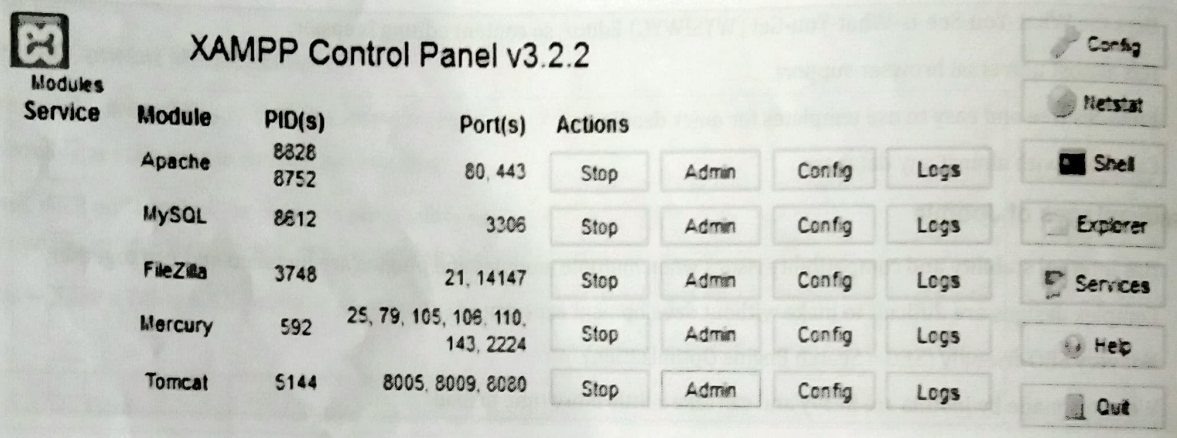
Fig. 6.4.3

- **Database:** Uses MySQL to store, manipulate and process data.. Images and documents cannot be stored. Also handles administration and authorization.
- **Joomla Framework:** Collection of software libraries and packages that are all open sourced and make up the Joomla CMS.
- **Components:** When a page needs to be loaded, a component is invoked to render the page's body. It has two parts: 1) ***administrator components***, which manage the many aspects of the component and 2) ***site components***, which render a page when a request is made.

- **Modules:** Lightweight extensions of Joomla to extend capabilities of web page rendering. Managed by components.
- **Plugin:** Used to extend Joomal's existing framework that triggers in specific events. Usually used to format the output from components or modules.
- **Templates :** Used to manage the look of the web page. It is of two types: 1) ***front-end templates***, which manage what the user sees, and 2) ***back-end templates***, which manage what the administrator sees.
- **Web Server :** It communicates with the database to get what is needed using HTTP methods.

Joomla Installation

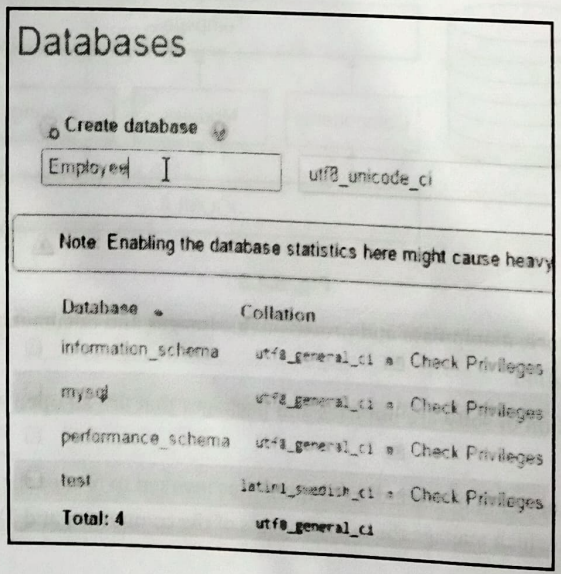
We will start with a server stack package called XAMPP. Install XAMPP from www.apachefriends.org/index.html and then install the executable file just like any regular software. You can activate it by launching it after installation and starting the server by clicking on "Start" next to "MySQL". After you click on "Start", the button name changes to "Stop". I have started all servers simultaneously here:



To install Joomla in localhost, download the zipped file from downloads.joomla.org/ and extract it and copy all files into the "htdocs" folder available in the XAMPP directory. I've created another subdirectory inside it for better management:

C: > xampp > htdocs > Joomla

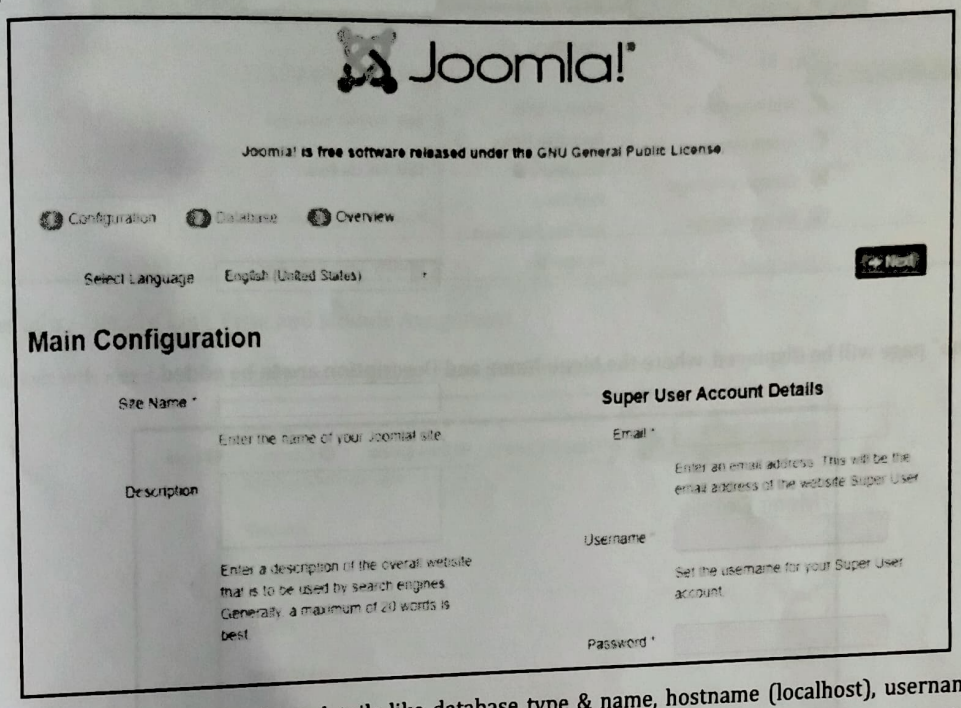
Now, create a database to connect to. Navigate to localhost/phpmyadmin and click on "Databases". This is assuming you already have PHP installed. Type a name for the database and click on "Create".



To get a live server ready, go to the Control Panel and then Databases > MySQL Databases section. Add the database name, user id and password. The name will be needed later for Joomla.

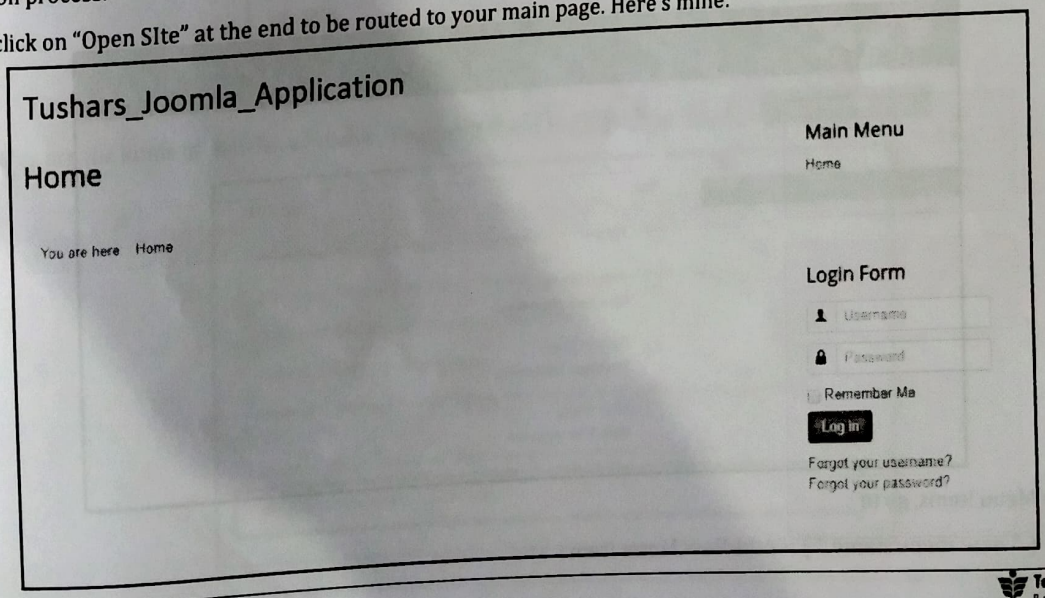
On your browser, search for http://localhost or http://localhost/abc where abc is the subdirectory inside the "htdocs" folder.

Once you get to the Joomla web installer page, you must add your basic information to get to the next step - information like language, site name, description, admin email, username and password, etc. After that, click the "Next" button for Database configuration.



The next page will ask for your database details like database type & name, hostname (localhost), username (root) and password (leave blank). Once entered, will move you along to the next page for Overview where you go through all the entered information and click on "Install". At the end of the wizard, click on "Remove Installation Folder" to finish the installation process.

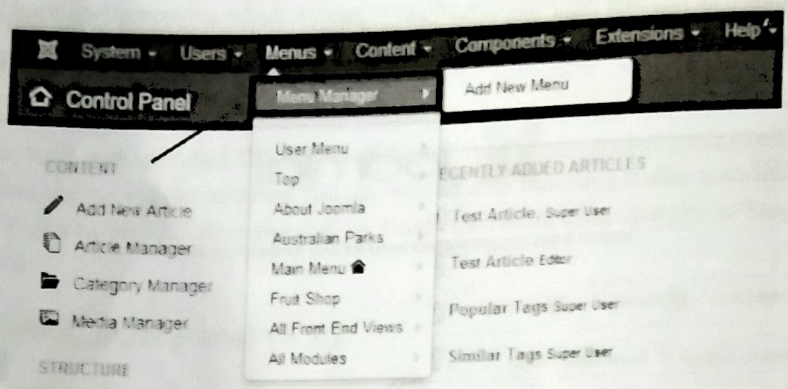
You can click on "Open SIte" at the end to be routed to your main page. Here's mine:



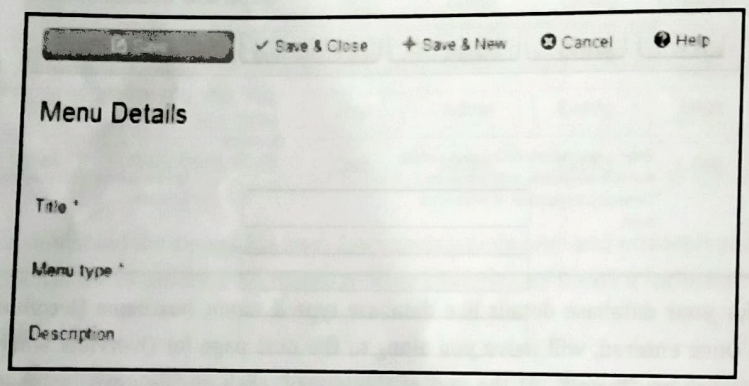
Joomla Menus

Menus are essential to a website. They help you navigate the entire site. Here's the steps to create one for Joomla:

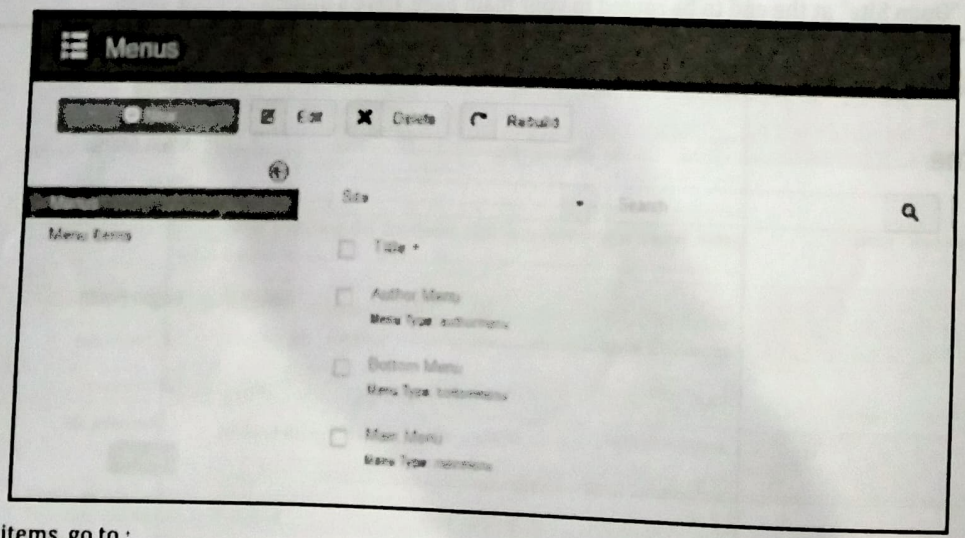
Click on "Menus" on the admin side of Joomla in a ribbon on the Control Panel and go to "Menu Manager" followed by "Add New Menu".



The "Add Menu" page will be displayed, where the Menu Name and Description are to be added.



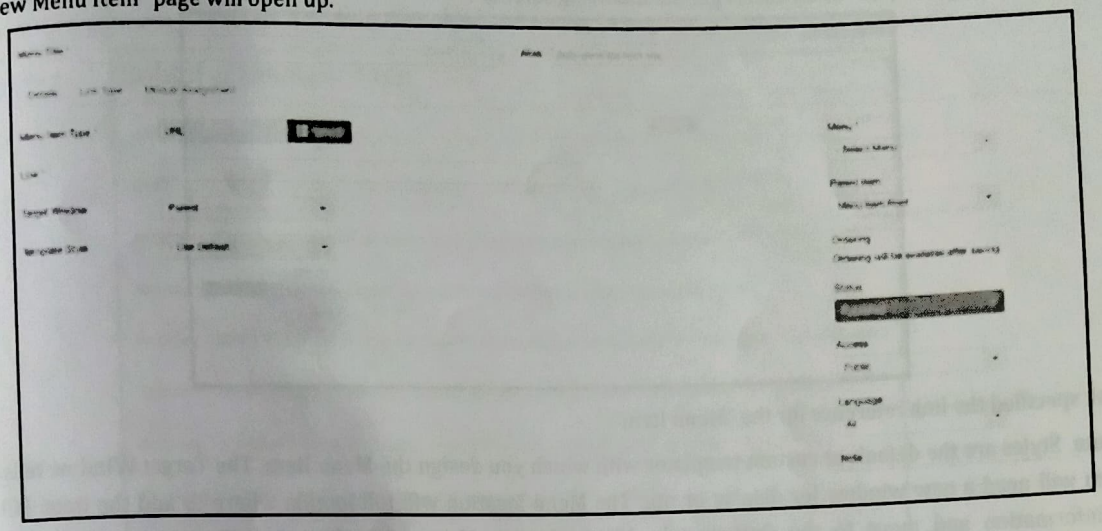
After that, click on "Save & Close". After successful addition, you will be routed to the Menu Manager, which will show all the content you have created, including the new Menu:



To create Menu items, go to :

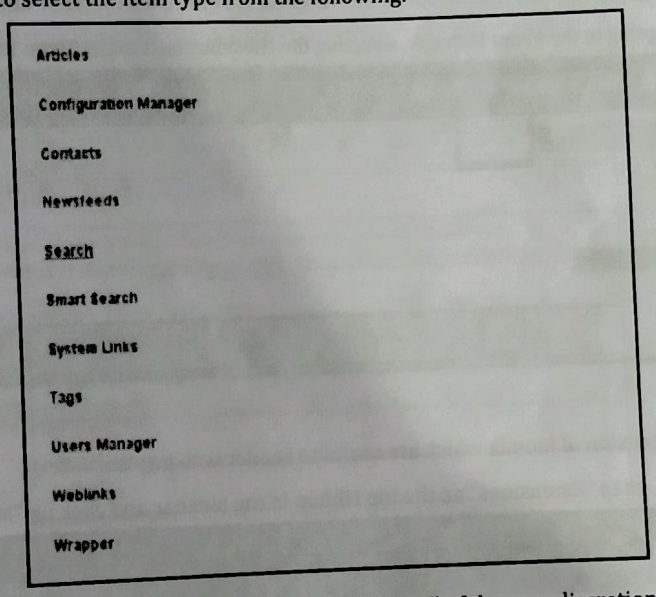
Menus > (* new menu name *) > Add New Menu Item

The "New Menu Item" page will open up.

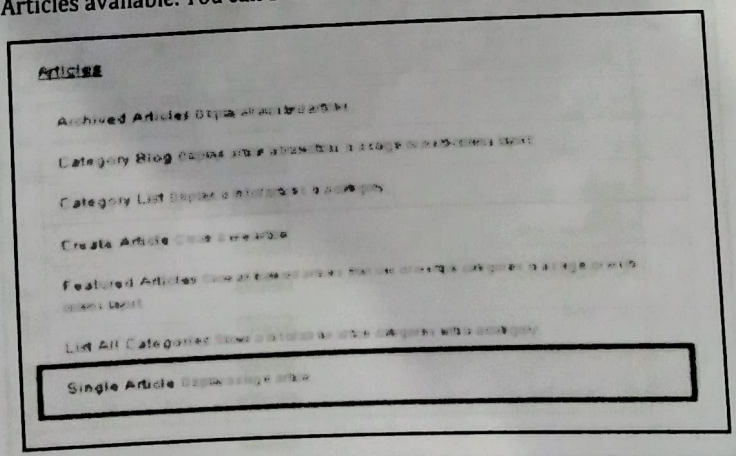


It has three tabs - Details, Link Type and Module Assignment.

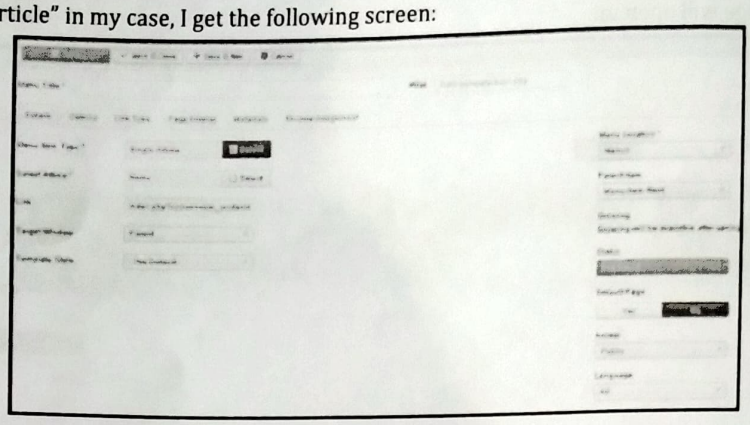
The Details tab will need you to select the item type from the following:



Following are the kinds of Articles available. You can browse through all of them on discretion.



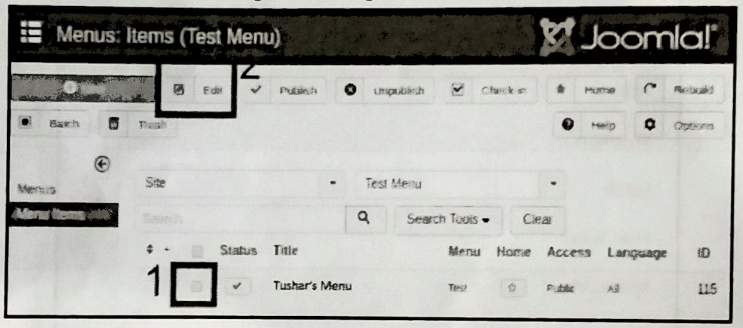
After selecting "Single Article" in my case, I get the following screen:



The link specified the link reference for the "Menu Item".

Template Styles are the default or custom templates with which you design the Menu Item. The Target WIndow tells you if the item will need a new window for display or not. The Menu location will tell Joomla where to add the item. Fill in all other information and move to the Options tab. After selecting Menu Item Type it will generate an Options tab automatically. Choose appropriate options as per needs. This tab will also have an option to choose the page location of the article, any custom CSS design for it, page title, headers, etc. After all is done, click on "Save and Close".

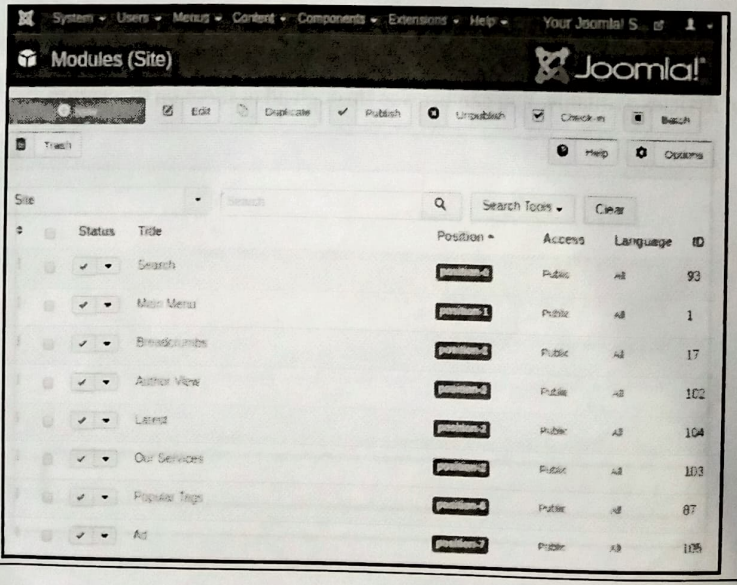
You can edit menu items by going to the Menu Manager, selecting the checkbox next to the Menu Item and clicking on "Edit":



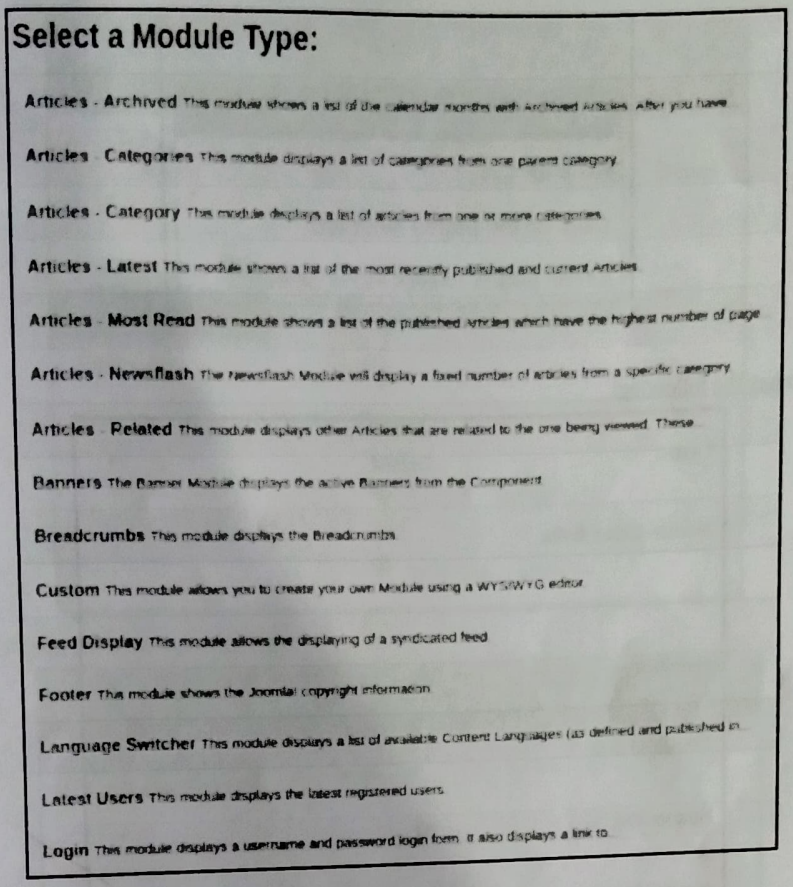
Joomla Modules

Modules are a lightweight extension of Joomla which are useful to render web pages.

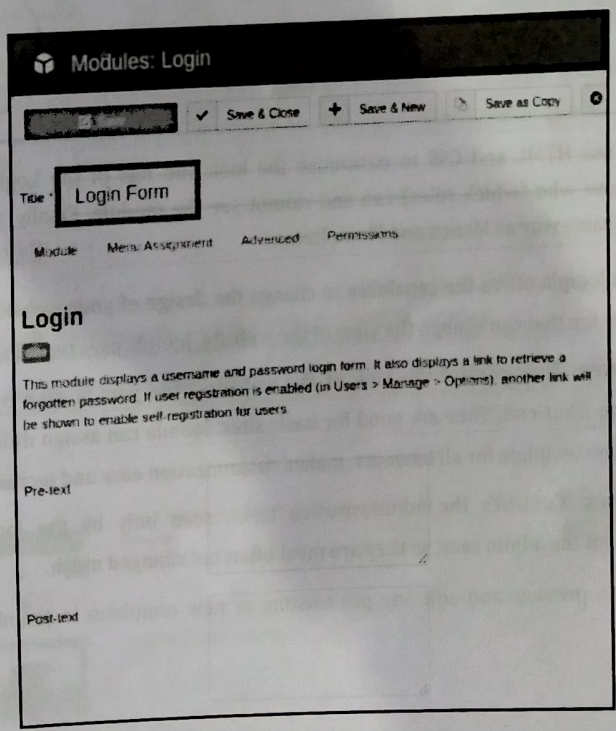
To create a Module in Joomla, go to "Extensions" on the top ribbon in the taskbar and click on "Modules".



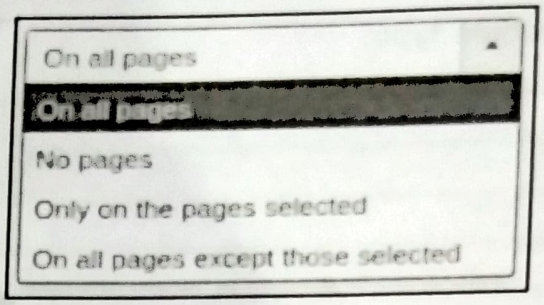
Click on the green "New" button at the top left in the Joomla toolbar, which will take you to a list of all available module types. You can select the one you need:



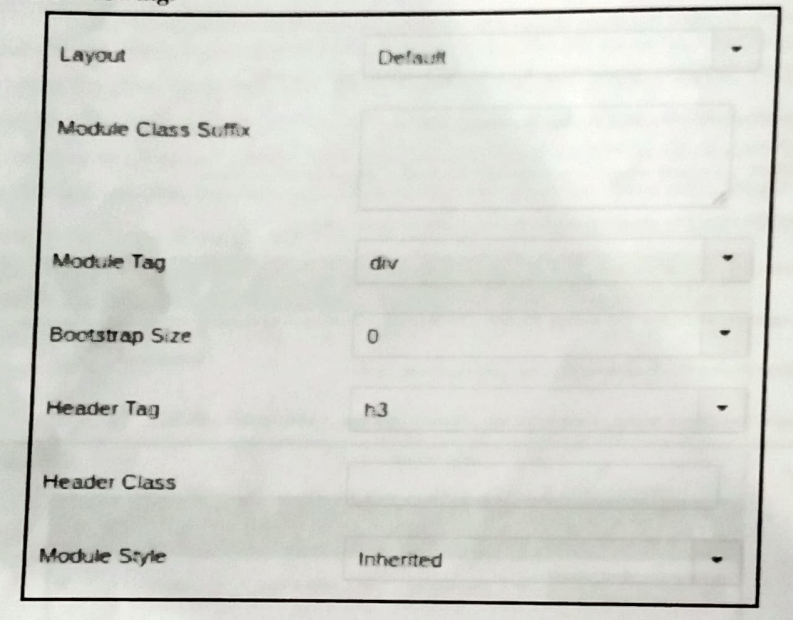
I chose the Module "Login".



Fill in the mandatory details in the "Module" tab here and move forward to "Menu Assignment" where you can choose which pages to display the Login module on from the following options:



Next, the Advance tab has the following:



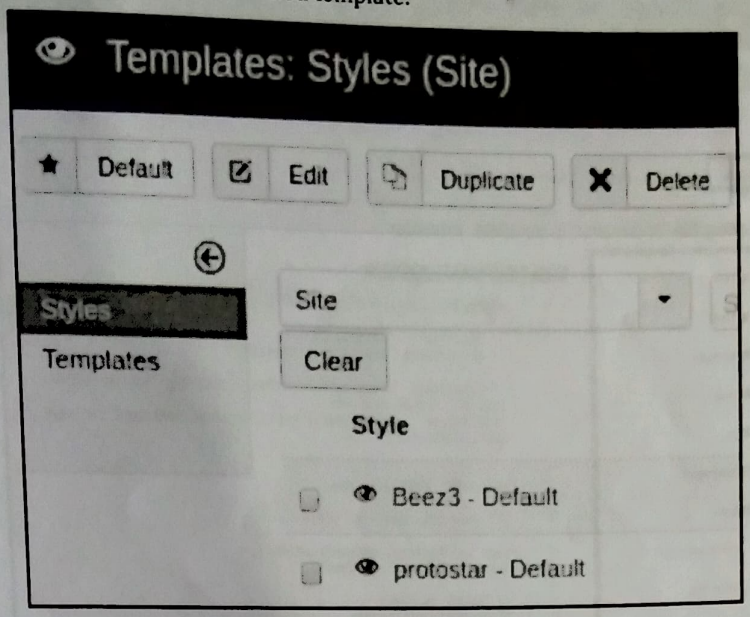
These options allow you to use HTML and CSS to customize the look and feel of the Login module. The last Module Permissions tab lets you choose who (which roles) can and cannot see the module. Finally, you click on "Save". Module information can be edited the same way as Menus and Menu Items.

- **Joomla Templates** Joomla offers the capability to change the design of your website without having to code. A template is an extension that can change the view of the website. Joomla uses two kinds of templates:
- **Front-End Templates:** They control the look of the website - the part is seen by the users. Most available templates are for the front-end. They are good for basic sites. Joomla can assign different templates to different Menu Items. Using one template for all however, makes customization easy and increases uniformity.
- **Back-End Templates:** Control's the administrative tasks, seen only by the Joomla administrator. These templates change what the admin sees, so they are most often not changed much.

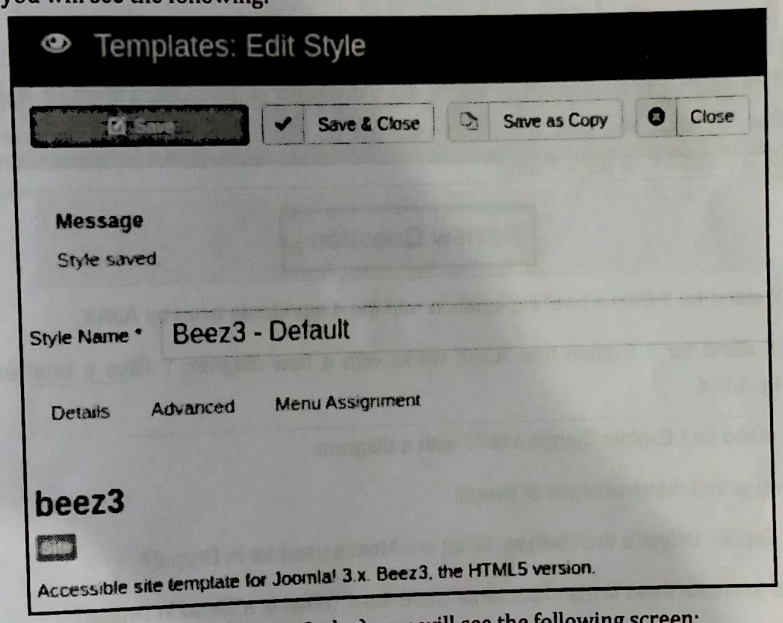
A Template Manager is used to preview and edit any pre-existing or new templates in Joomla. To access the Template Manager, go to:

Extensions > Templates

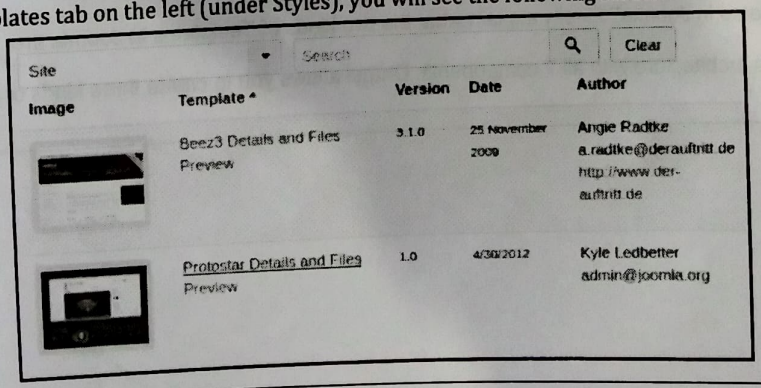
Once you click on "Templates", you will be taken to the Template Manager's style tab. You will see a list of all pre-installed templates here. Beez3 is one such default, pre-installed template:



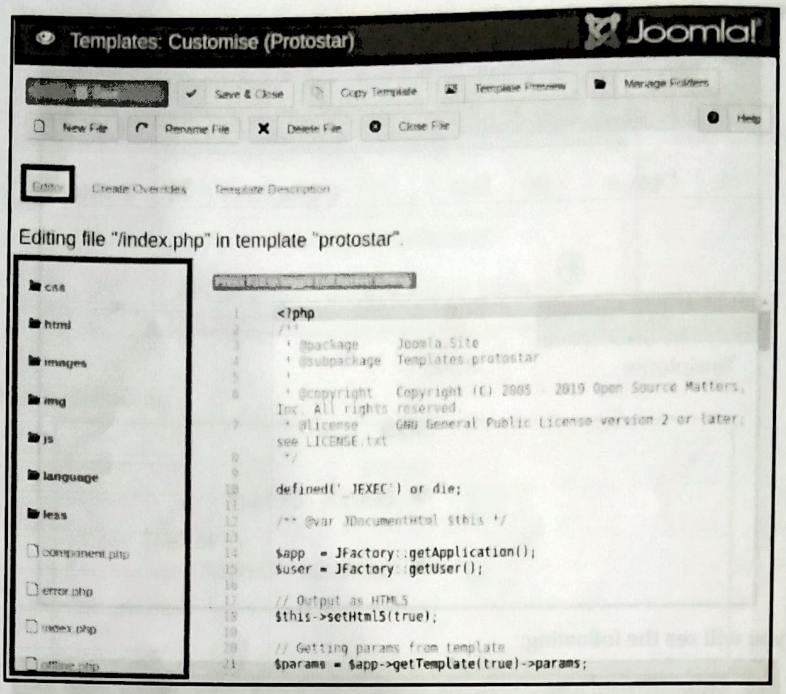
If you click on "Beez3", you will see the following:



If you click on the Templates tab on the left (under Styles), you will see the following screen:



If you click on the Template and stay in the "Editor" tab, you will be able to make any updates to the pre-installed template per your requirements:



Note from Author : This section is insufficient to teach you the details of how Joomla works, but has been deemed sufficient as per University syllabus and thus, this section will not go into further detail and confuse you. A rudimentary understanding of Joomla is sufficient as per syllabus.

Review Questions

Q. 1 What does AJAX stand for ? Give a brief explanation with the 4 standards used by AJAX. **(5 Marks)**

Q. 2 What does AJAX stand for ? Explain how AJAX works with a flow diagram ? Give a brief explanation with the 4 standards used by AJAX. **(10 Marks)**

Q. 3 What does MVT stand for? Explain Django's MVT with a diagram. **(10 Marks)**

Q. 4 State the advantages and disadvantages of Drupal. **(5 Marks)**

Q. 5 What is a CMS? Explain Drupal's architecture. What are Menus used for in Drupal? **(10 Marks)**

Q. 6 Drupal allows you to create three kinds of contents: state them. What is a Menu in Drupal? **(5 Marks)**

Q. 7 What are Templates in Joomla? Briefly explain what the two kinds of Templates in Joomla are. **(5 Marks)**

Q. 8 Explain Joomla's architecture with all 7 components. Drupal allows you to create three kinds of contents: state them. **(10 Marks)**

❑❑❑