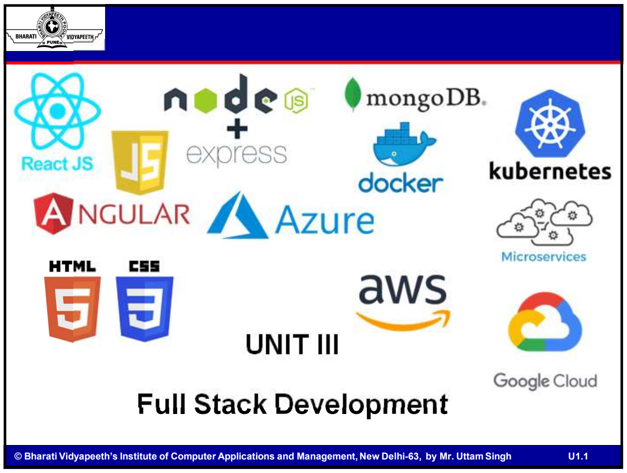


UNIT III

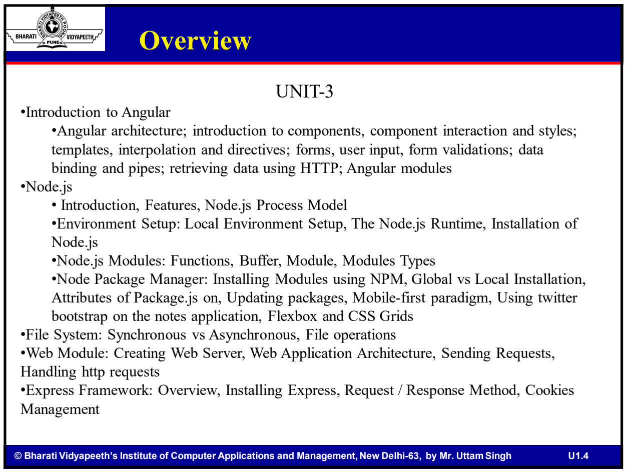
Full Stack Development

Learning Resources

- Books
  - D. Brad, B. Dayley and C. Dayley, "Node.js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications", Addison-Wesley Professional, 2nd Edition, 2017
  - Web Links (Strictly Referred):
    - https://angular.io/
    - https://nodejs.org/
    - https://expressjs.com

Course Outcome

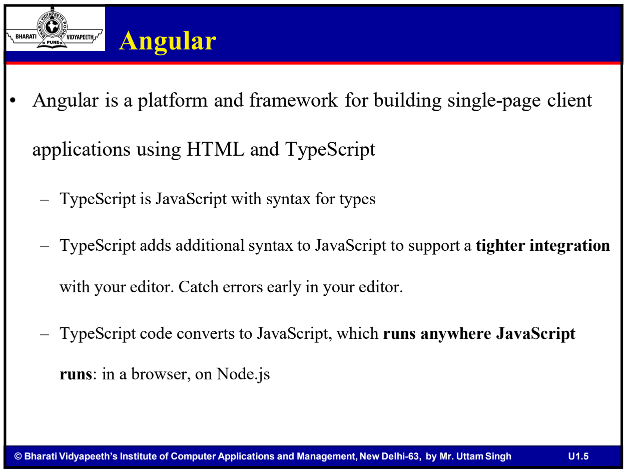
- CO1: Relate the basics of Javascript (JS) and ReactJS
- CO2: Apply the concepts of props and State Management in React JS
- CO3: Examine Redux and Router with React JS
- CO4: Appraise Node JS environment and modular development
- CO5: Develop full stack applications using MongoDB



Overview

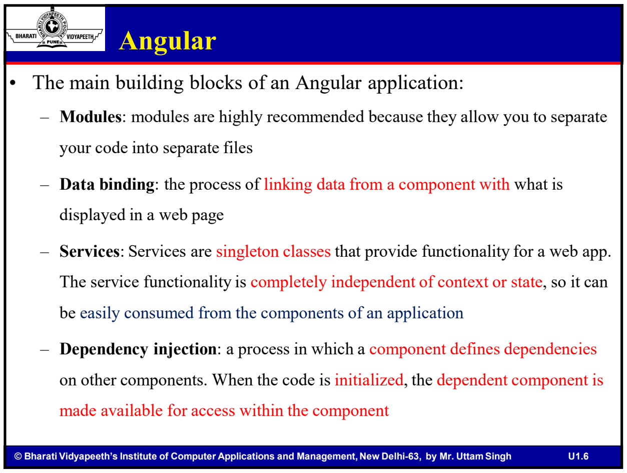
UNIT-3

- •Introduction to Angular
  - •Angular architecture; introduction to components, component interaction and styles; templates, interpolation and directives; forms, user input, form validations; data binding and pipes; retrieving data using HTTP; Angular modules
- •Node.js
  - • Introduction, Features, Node.js Process Model
  - •Environment Setup: Local Environment Setup, The Node.js Runtime, Installation of Node.js
  - •Node.js Modules: Functions, Buffer, Module, Modules Types
  - •Node Package Manager: Installing Modules using NPM, Global vs Local Installation, Attributes of Package.js on, Updating packages, Mobile-first paradigm, Using twitter bootstrap on the notes application, Flexbox and CSS Grids
- •File System: Synchronous vs Asynchronous, File operations
- •Web Module: Creating Web Server, Web Application Architecture, Sending Requests, Handling http requests
- •Express Framework: Overview, Installing Express, Request / Response Method, Cookies Management



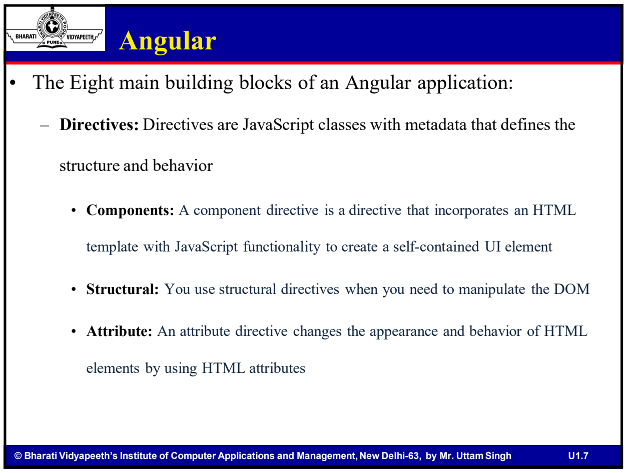
Angular

- Angular is a platform and framework for building single-page client applications using HTML and TypeScript
  - TypeScript is JavaScript with syntax for types
  - TypeScript adds additional syntax to JavaScript to support a **tighter integration** with your editor. Catch errors early in your editor.
  - TypeScript code converts to JavaScript, which **runs anywhere JavaScript runs**: in a browser, on Node.js



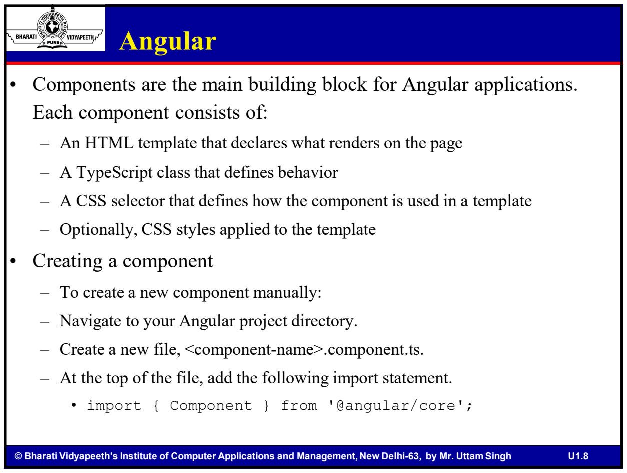
Angular

- The main building blocks of an Angular application:
  - **Modules**: modules are highly recommended because they allow you to separate your code into separate files
  - **Data binding**: the process of linking data from a component with what is displayed in a web page
  - **Services**: Services are singleton classes that provide functionality for a web app. The service functionality is completely independent of context or state, so it can be easily consumed from the components of an application
  - **Dependency injection**: a process in which a component defines dependencies on other components. When the code is initialized, the dependent component is made available for access within the component



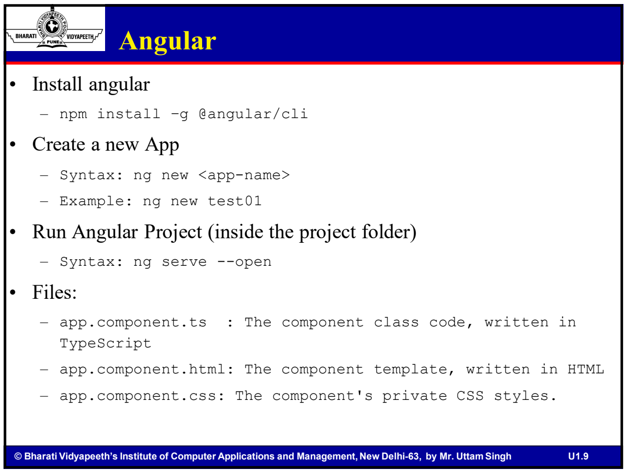
Angular

- The Eight main building blocks of an Angular application:
  - **Directives:** Directives are JavaScript classes with metadata that defines the structure and behavior
    - **Components:** A component directive is a directive that incorporates an HTML template with JavaScript functionality to create a self-contained UI element
    - **Structural:** You use structural directives when you need to manipulate the DOM
    - **Attribute:** An attribute directive changes the appearance and behavior of HTML elements by using HTML attributes



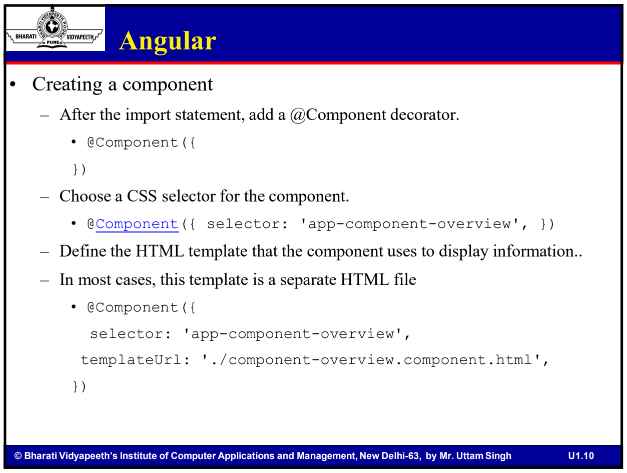
Angular

- Components are the main building block for Angular applications. Each component consists of:
  - An HTML template that declares what renders on the page
  - A TypeScript class that defines behavior
  - A CSS selector that defines how the component is used in a template
  - Optionally, CSS styles applied to the template
- Creating a component
  - To create a new component manually:
  - Navigate to your Angular project directory.
  - Create a new file, <component-name>.component.ts.
  - At the top of the file, add the following import statement.
    - `import { Component } from '@angular/core';`



Angular

- Install angular
  - `npm install -g @angular/cli`
- Create a new App
  - `Syntax: ng new <app-name>`
  - `Example: ng new test01`
- Run Angular Project (inside the project folder)
  - `Syntax: ng serve --open`
- Files:
  - `app.component.ts  : The component class code, written in TypeScript`
  - `app.component.html: The component template, written in HTML`
  - `app.component.css: The component's private CSS styles.`

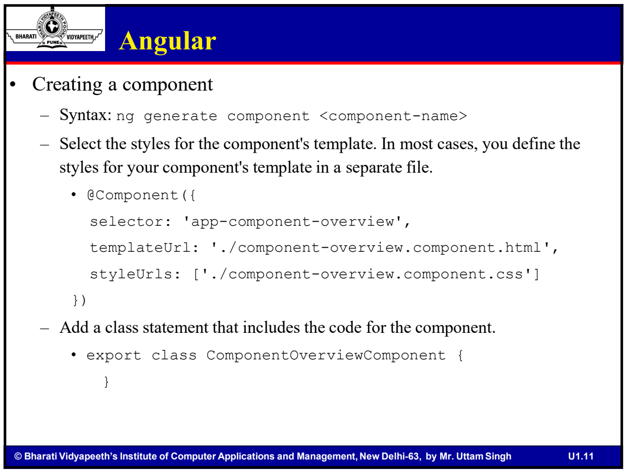


Angular

- Creating a component
  - After the import statement, add a @Component decorator.

    -
      ```
      @Component({
      })
      ```
  - Choose a CSS selector for the component.
    - `@Component({ selector: 'app-component-overview', })`
  - Define the HTML template that the component uses to display information..
  - In most cases, this template is a separate HTML file

    -
      ```
      @Component({
        selector: 'app-component-overview',
       templateUrl: './component-overview.component.html',
      })
      ```

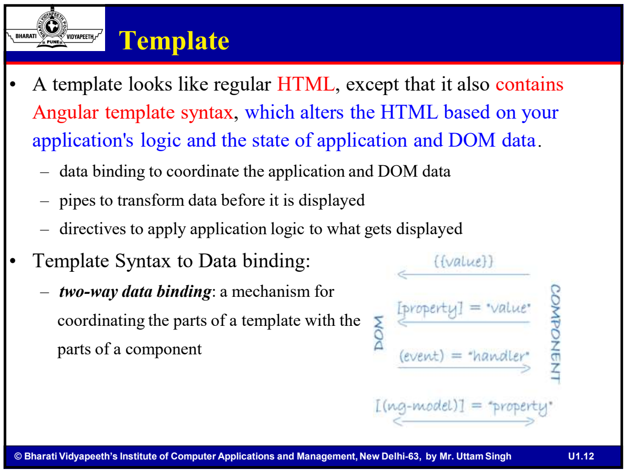


Angular

- Creating a component
  - Syntax: `ng generate component <component-name>`
  - Select the styles for the component's template. In most cases, you define the styles for your component's template in a separate file.

    -
      ```
      @Component({
        selector: 'app-component-overview',
        templateUrl: './component-overview.component.html',
        styleUrls: ['./component-overview.component.css']
      })
      ```
  - Add a class statement that includes the code for the component.

    -
      ```
      export class ComponentOverviewComponent {
          }
      ```



Template

- A template looks like regular HTML, except that it also contains Angular template syntax, which alters the HTML based on your application's logic and the state of application and DOM data.
  - data binding to coordinate the application and DOM data
  - pipes to transform data before it is displayed
  - directives to apply application logic to what gets displayed
- Template Syntax to Data binding:
  - ***two-way data binding***: a mechanism for coordinating the parts of a template with the parts of a component

Text interpolation

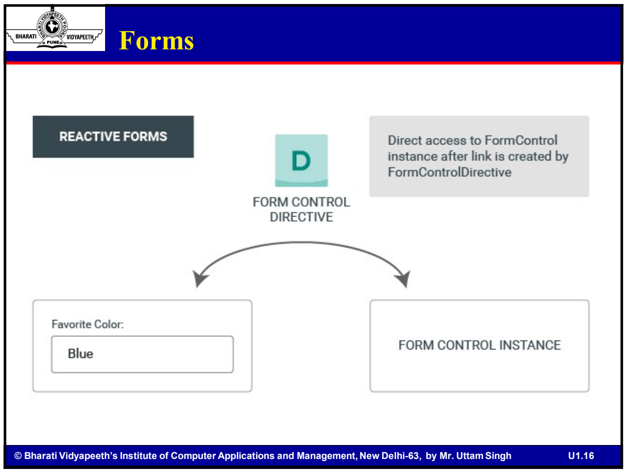
- Text interpolation lets you incorporate dynamic string values into your HTML templates
- Interpolation refers to embedding expressions into marked up text, uses the double curly brace `{{` and `}}` characters as delimiters
  - `currentCustomer = 'Maria';` // src/app/app.component.ts
  - `<h3>Current customer: {{ currentCustomer }}</h3>` // app.component.html
- Resolving expressions with interpolation
  - `<p>The sum of 1 + 1 is {{1 + 1}}.</p>`

Directive

- Directives are classes that add additional behavior to elements in your Angular applications.
- Built-in attribute directives
  - Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components
  - Built-in directives use only public APIs
- Adding and removing classes with NgClass

Forms

- Handling user input [https://angular.io/guide/forms-overview]
- Two types of approach
  - reactive : Provide direct, explicit access to the underlying forms object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
  - template-driven : Rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're straightforward to add to an app, but they don't scale as well as reactive forms. It is very useful for basic form requirements and logic that can be managed solely in the template.



Forms

REACTIVE FORMS

D

FORM CONTROL DIRECTIVE

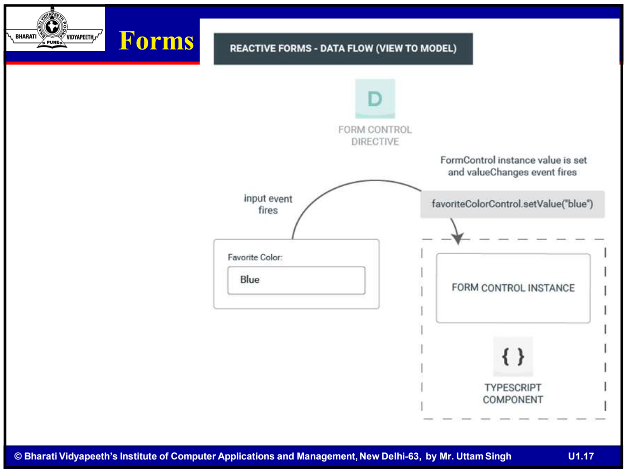
Direct access to FormControl instance after link is created by FormControlDirective

Favorite Color:

Blue

FORM CONTROL INSTANCE

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Mr. Uttam Singh U1.16



Forms

REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)

D

FORM CONTROL DIRECTIVE

FormControl instance value is set and valueChanges event fires

input event fires

favoriteColorControl.setValue("blue")

Favorite Color:

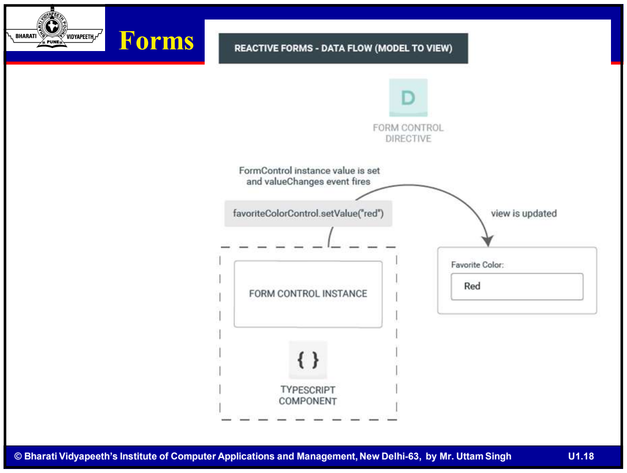
Blue

FORM CONTROL INSTANCE

{ }

TYPESCRIPT COMPONENT

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Mr. Uttam Singh U1.17



Forms

REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)

D

FORM CONTROL DIRECTIVE

FormControl instance value is set and valueChanges event fires

favoriteColorControl.setValue("red")

view is updated

FORM CONTROL INSTANCE

{ }

TYPESCRIPT COMPONENT

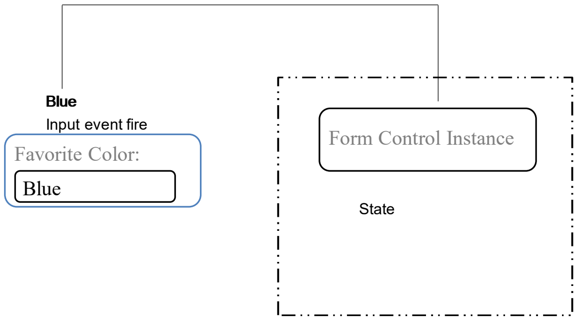
Favorite Color:

Red

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Mr. Uttam Singh U1.18

Forms

- Reactive form data flow View-to-model



Blue

Input event fire

Favorite Color:

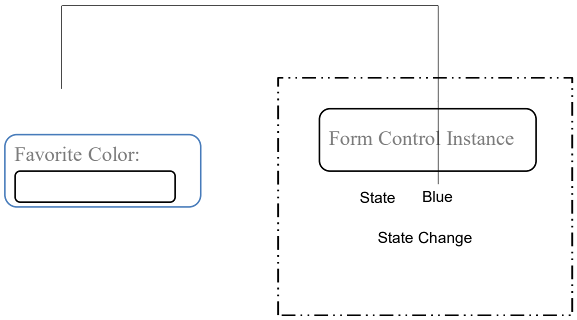
Blue

Form Control Instance

State

Forms

- Reactive form data flow model-to-view



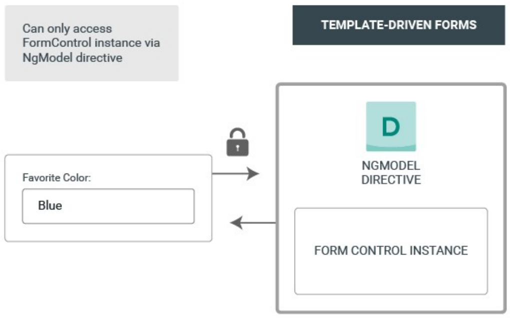
Favorite Color:

Form Control Instance

State Blue

State Change

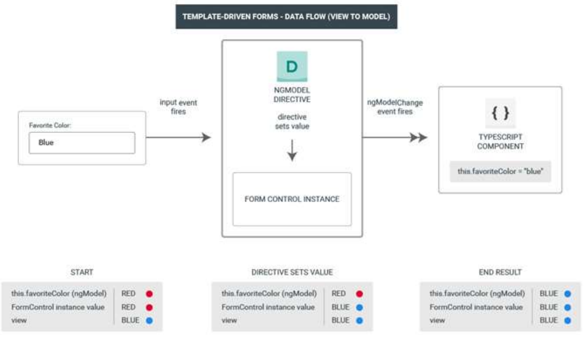
Forms



Can only access FormControl instance via NgModel directive

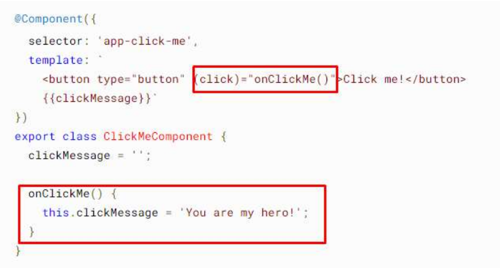
TEMPLATE-DRIVEN FORMS

Forms


TEMPLATE-DRIVEN FORMS - DATA FLOW (VIEW TO MODEL)
Favorite Color:
Blue
input event fires
D
NGMODEL DIRECTIVE
directive sets value
FORM CONTROL INSTANCE
ngModelChange event fires
{ }
TYPESCRIPT COMPONENT
this.favoriteColor = "blue"
START
this.favoriteColor (ngModel) RED
FormControl instance value RED
view BLUE
DIRECTIVE SETS VALUE
this.favoriteColor (ngModel) RED
FormControl instance value BLUE
view BLUE
END RESULT
this.favoriteColor (ngModel) BLUE
FormControl instance value BLUE
view BLUE

User Input

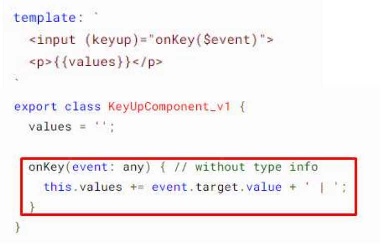
- User actions such as clicking a link, pushing a button, and entering text raise DOM events
  - Binding to user input events



```
@Component({
  selector: 'app-click-me',
  template: `
    <button type="button" (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

User Input

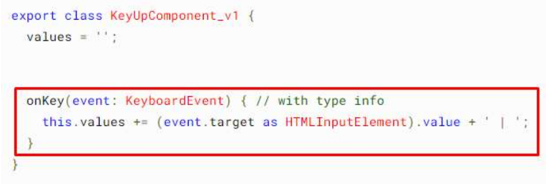
- User actions such as clicking a link, pushing a button, and entering text raise DOM events
  - Get user input from the $event object



```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

User Input

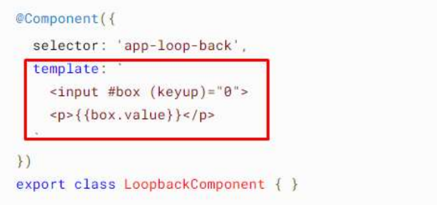
- User actions such as clicking a link, pushing a button, and entering text raise DOM events
  - Type the $event



```
export class KeyUpComponent_v1 {
  values = '';


  onKey(event: KeyboardEvent) { // with type info
    this.values += (event.target as HTMLInputElement).value + ' | ';
  }
}
```

User Input

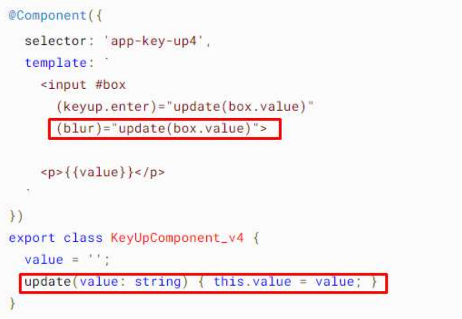
- User actions such as clicking a link, pushing a button, and entering text raise DOM events
  - Get user input from a template reference variable



```
@Component({
  selector: 'app-loop-back',
  template: `
    <input #box (keyup)="0">
    <p>{{box.value}}</p>
  `
})
export class LoopbackComponent { }
```

User Input

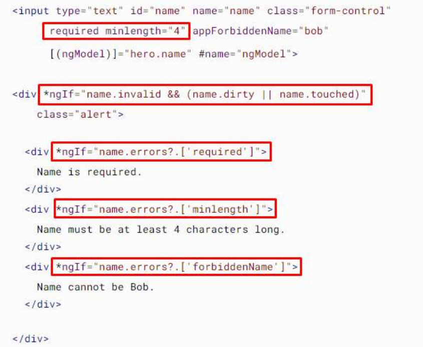
- User actions such as clicking a link, pushing a button, and entering text raise DOM events
  - On blur



```
@Component({
  selector: 'app-key-up4',
  template: `
    <input #box
      (keyup.enter)="update(box.value)"
      (blur)="update(box.value)">

    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v4 {
  value = '';
  update(value: string) { this.value = value; }
}
```

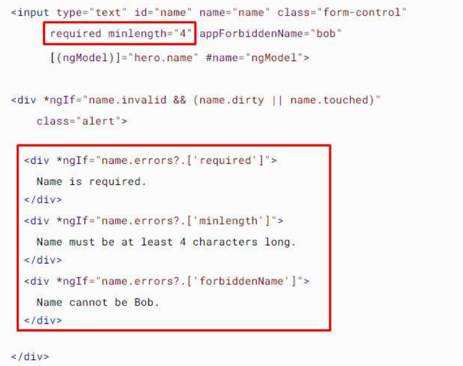
Form Validation

- Validating input in template-driven form



```
<input type="text" id="name" name="name" class="form-control"
      required minlength="4" appForbiddenName="bob"
      [(ngModel)]="hero.name" #name="ngModel">

<div *ngIf="name.invalid && (name.dirty || name.touched)"
    class="alert">

  <div *ngIf="name.errors?.['required']">
    Name is required.
  </div>
  <div *ngIf="name.errors?.['minlength']">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.['forbiddenName']">
    Name cannot be Bob.
  </div>

</div>
```

Form Validation



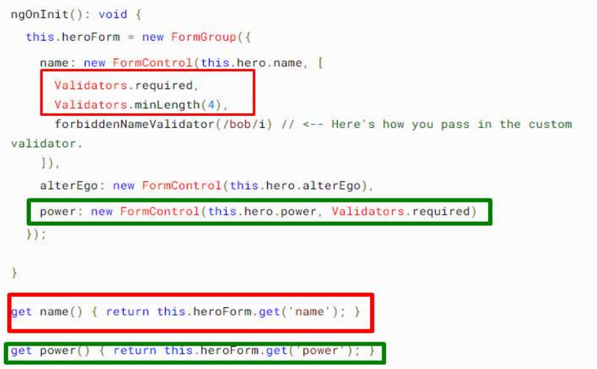
```
<input type="text" id="name" name="name" class="form-control"
      required minlength="4" appForbiddenName="bob"
      [(ngModel)]="hero.name" #name="ngModel">

<div *ngIf="name.invalid && (name.dirty || name.touched)"
    class="alert">

  <div *ngIf="name.errors?.['required']">
    Name is required.
  </div>
  <div *ngIf="name.errors?.['minlength']">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.['forbiddenName']">
    Name cannot be Bob.
  </div>

</div>
```

Form Validation

- Validating input in reactive forms

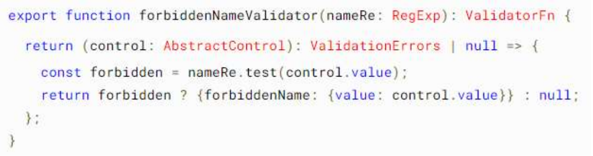
VALIDATOR TYPE	DETAILS
Sync validators	Synchronous functions that take a control instance and immediately return either a set of validation errors or null. Pass these in as the second argument when you instantiate a FormControl.
Async validators	Asynchronous functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. Pass these in as the third argument when you instantiate a FormControl.

Form Validation (Built-in validator functions)



```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    name: new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
validator.
    ]),
    alterEgo: new FormControl(this.hero.alterEgo),
    power: new FormControl(this.hero.power, Validators.required)
  });

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

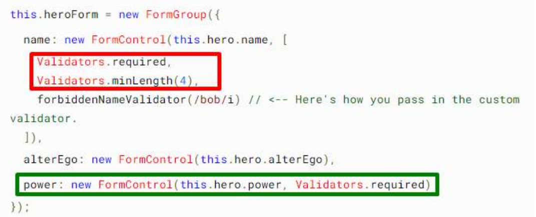
Form Validation (Defining custom validators)



```
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {forbiddenName: {value: control.value}} : null;
  };
}
```

Adding custom validators to reactive forms

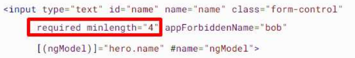
- add a custom validator by passing the function directly to the FormControl



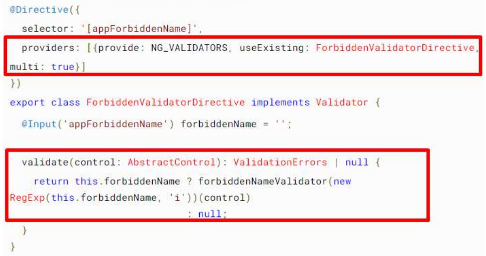
```
this.heroForm = new FormGroup({
  name: new FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
    forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
validator.
  ]),
  alterEgo: new FormControl(this.hero.alterEgo),
  power: new FormControl(this.hero.power, Validators.required)
});
```

Adding custom validators to template-driven forms

- In template-driven forms, add a directive to the template, where the directive wraps the validator function.



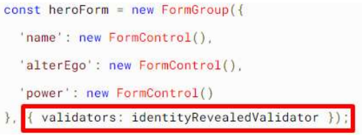
```
<input type="text" id="name" name="name" class="form-control"
     required minlength="4" appForbiddenName="bob"
      [(ngModel)]="hero.name" #name="ngModel">
```



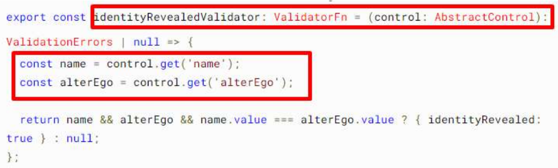
```
@Directive({
  selector: '[appForbiddenName]',
  providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective,
multi: true}]
})
export class ForbiddenValidatorDirective implements Validator {
  @Input('appForbiddenName') forbiddenName = '';

  validate(control: AbstractControl): ValidationErrors | null {
    return this.forbiddenName ? forbiddenNameValidator(new
RegExp(this.forbiddenName, 'i'))(control)
                                : null;
  }
}
```

Cross-field validation

- A cross-field validator is a custom validator that compares the values of different fields in a form and accepts or rejects them in combination.



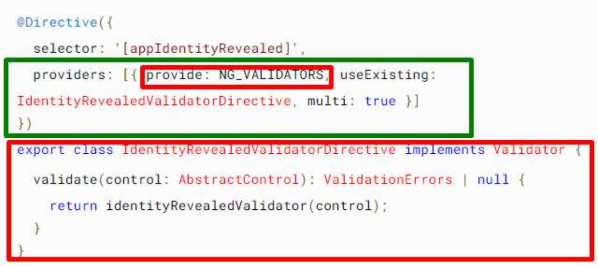
```
const heroForm = new FormGroup({
  'name': new FormControl(),
  'alterEgo': new FormControl(),
  'power': new FormControl()
}, { validators: identityRevealedValidator });
```



```
export const identityRevealedValidator: ValidatorFn = (control: AbstractControl):
ValidationErrors | null => {
  const name = control.get('name');
  const alterEgo = control.get('alterEgo');

  return name && alterEgo && name.value === alterEgo.value ? { identityRevealed:
true } : null;
};
```

Adding cross-validation to template-driven forms

- For a template-driven form, you must create a directive to wrap the validator function. You provide that directive as the validator using the NG_VALIDATORS token.

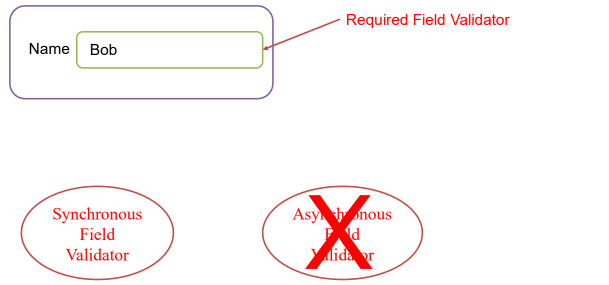


```
@Directive({
  selector: '[appIdentityRevealed]',
  providers: [{ provide: NG_VALIDATORS, useExisting:
IdentityRevealedValidatorDirective, multi: true }]
})
export class IdentityRevealedValidatorDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    return identityRevealedValidator(control);
  }
}
```

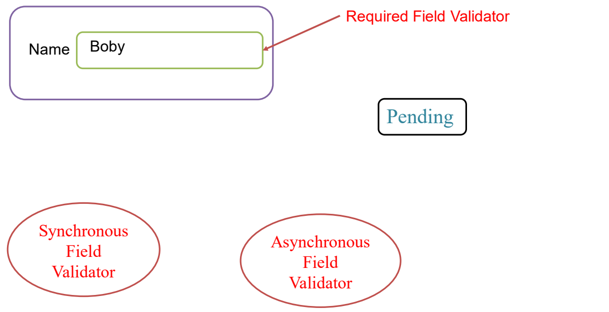
Asynchronous validators

- Asynchronous validators implement the AsyncValidatorFn and AsyncValidator interfaces.
- Differ than synchronous
  - The validate() functions must return a Promise or an observable,
  - The observable returned must be finite, meaning it must complete at some point. To convert an infinite observable into a finite one, pipe the observable through a filtering operator such as first, last, take, or takeUntil.
- Asynchronous validation is performed after the synchronous validation and only if the synchronous validation is successful.
- After asynchronous validation begins, the form control enters a pending state. Inspect the control's pending property and use it to give visual feedback about the ongoing validation operation.

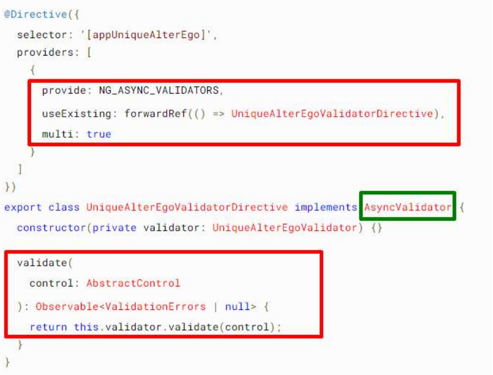
synchronous validators works



synchronous validators works



Adding async validators to template-driven forms



```
@Directive({
  selector: '[appUniqueAlterEgo]',
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: forwardRef(() => UniqueAlterEgoValidatorDirective),
      multi: true
    }
  ]
})
export class UniqueAlterEgoValidatorDirective implements AsyncValidator {
  constructor(private validator: UniqueAlterEgoValidator) {}

  validate(
    control: AbstractControl
  ): Observable<ValidationErrors | null> {
    return this.validator.validate(control);
  }
}
```

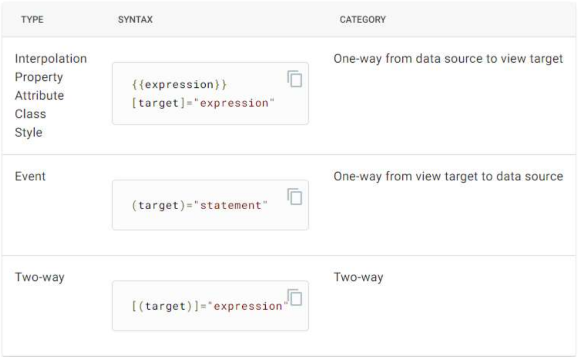
Data Binding & Pipes

- Data binding automatically keeps your page up-to-date based on your application's state in Angular.
- Data binding works with properties of DOM elements, components, and directives, not HTML attributes
- Attributes initialize DOM properties and you can configure them to modify an element's behavior

Types of data binding

- Angular provides three categories of data binding according to the direction of data flow:
  - From source to view
  - From view to source
  - In a two-way sequence of view to source to view

Types of data binding



TYPE	SYNTAX	CATEGORY

Interpolation
Property
Attribute
Class
Style
	`{{expression}}`
	`[target]="expression"`
		One-way from data source to view target

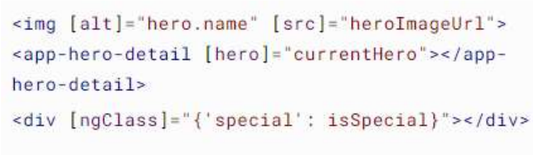
Event	`(target)="statement"`	One-way from view target to data source
Two-way	`[(target)]="expression"`	Two-way

Types of data binding

- Binding types other than interpolation have a target name to the left of the equal sign.
- The target of a binding is a property or event, which you surround with square bracket [ ] characters, parenthesis ( ) characters, or both [( )] characters.
- The binding punctuation of [], (), [()], and the prefix specify the direction of data flow.
- Summary:
  - Use [] to bind from source to view
  - Use () to bind from view to source
  - Use [()] to bind in a two way sequence of view to source to view

Binding types and targets

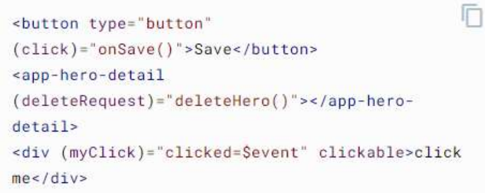
- Type: Property
- Target:
  - Element property
  - Component property
  - Directive property



```
<img [alt]="hero.name" [src]="heroImageUrl">
<app-hero-detail [hero]="currentHero"></app-
hero-detail>
<div [ngClass]="{'special': isSpecial}"></div>
```

Binding types and targets

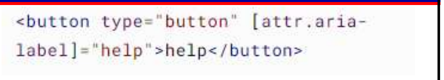
- Type: Event
- Target:
  - Element event
  - Component event
  - Directive event



```
<button type="button"
(click)="onSave()">Save</button>
<app-hero-detail
(deleteRequest)="deleteHero()"></app-hero-
detail>
<div (myClick)="clicked=$event" clickable>click
me</div>
```

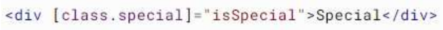
Binding types and targets

- Type: Attribute
- Target:
  - Attribute



```
<button type="button" [attr.aria-
label]="help">help</button>
```

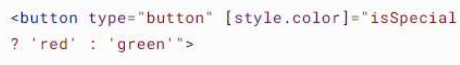
- Type: Class
- Target:



```
<div [class.special]="isSpecial">Special</div>
```

  - class property

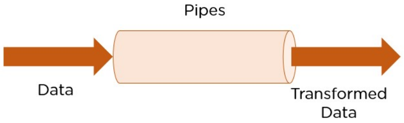
- Type: Style
- Target:
  - style property



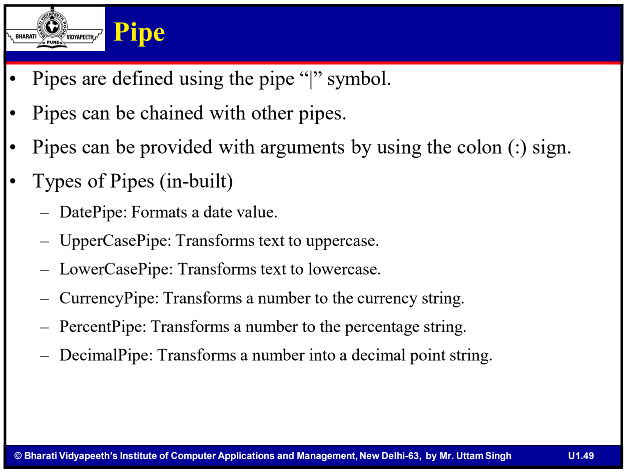
```
<button type="button" [style.color]="isSpecial
? 'red' : 'green'">
```

Pipe

- Decorator that marks a class as pipe and supplies configuration metadata.
- Angular Pipes allows its users to change the format in which data is being displayed on the screen.

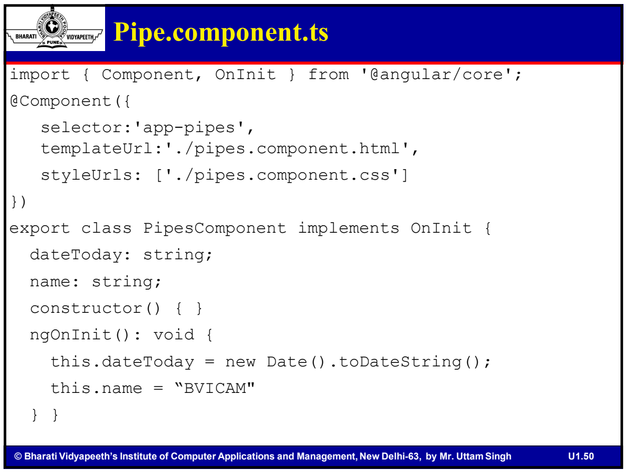


- Angular pipes are simple functions designed to accept an input value, process, and return a transformed value as the output.



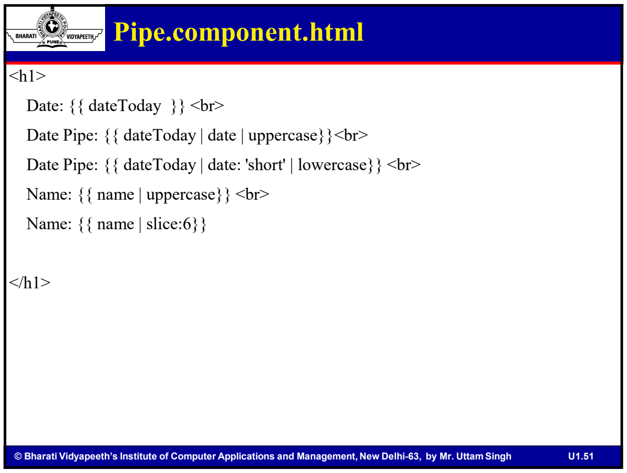
Pipe

- Pipes are defined using the pipe “|” symbol.
- Pipes can be chained with other pipes.
- Pipes can be provided with arguments by using the colon (:) sign.
- Types of Pipes (in-built)
  - DatePipe: Formats a date value.
  - UpperCasePipe: Transforms text to uppercase.
  - LowerCasePipe: Transforms text to lowercase.
  - CurrencyPipe: Transforms a number to the currency string.
  - PercentPipe: Transforms a number to the percentage string.
  - DecimalPipe: Transforms a number into a decimal point string.



Pipe.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
   selector:'app-pipes',
   templateUrl:'./pipes.component.html',
   styleUrls: ['./pipes.component.css']
})
export class PipesComponent implements OnInit {
  dateToday: string;
  name: string;
  constructor() { }
  ngOnInit(): void {
    this.dateToday = new Date().toDateString();
    this.name = “BVICAM"
  } }
```

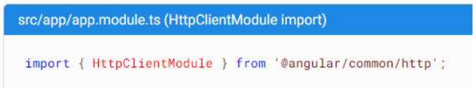


Pipe.component.html

```
<h1>
  Date: {{ dateToday  }} <br>
  Date Pipe: {{ dateToday | date | uppercase}}<br>
  Date Pipe: {{ dateToday | date: 'short' | lowercase}} <br>
  Name: {{ name | uppercase}} <br>
  Name: {{ name | slice:6}}

</h1>
```

Getting data from backend

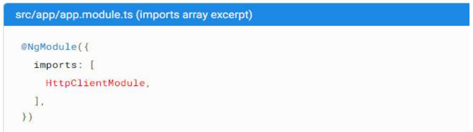
- Enable HTTP services
  - HttpClient is Angular's mechanism for communicating with a remote server over HTTP.
  - Make HttpClient available everywhere in the application in two steps. First, add it to the root AppModule by importing it:



src/app/app.module.ts (HttpClientModule import)

```
import { HttpClientModule } from '@angular/common/http';
```

  - Next, still in the AppModule, add HttpClientModule to the imports array:



src/app/app.module.ts (imports array excerpt)

```
@NgModule({
  imports: [
    HttpClientModule,
  ],
})
```

Angular Fetch Data from API Using HttpClientModule

- Step 1: Add HttpClientModule Into the Imports Array and Import it.
- Step 2: Create an Instance of HttpClient and Fetch the Data Using It
- Step 3: Create a Student Interface To Cast the Observables
- Step 4: Subscribe the Data From the Service in the Component

Angular Modules

- Angular applications are modular and Angular has its own modularity system called NgModules.
- NgModules are containers for a cohesive block of code dedicated to
  - an application domain,
  - a workflow, or
  - a closely related set of capabilities
- They can contain
  - components,
  - service providers, and
  - other code files whose scope is defined by the containing NgModule.

Angular Modules

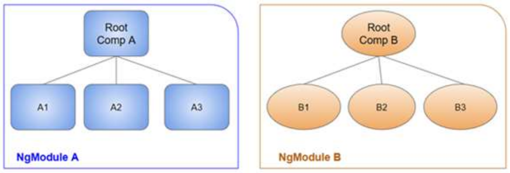
- An NgModule is defined by a class decorated with @NgModule().
- The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module
- Properties are:
  - declarations: -The components, directives, and pipes that belong to this NgModule.
  - exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
  - imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
  - providers: Creators of services that this NgModule contributes to the global collection of services
  - bootstrap: The main application view, called the root component, which hosts all other application views.

Example of simple root module (app.module.ts)

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

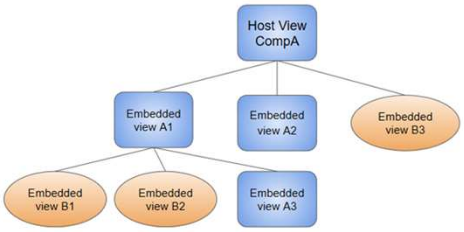
NgModules and components

- NgModules provide a compilation context for their components.
- A root NgModule always has a root component that is created during bootstrap
- Any NgModule can include any number of additional components, which can be loaded through the router or created through the template



NgModules and components

- A component and its template together define a view. A component can contain a view hierarchy, which allows you to define arbitrarily complex areas of the screen that can be created, modified, and destroyed as a unit.



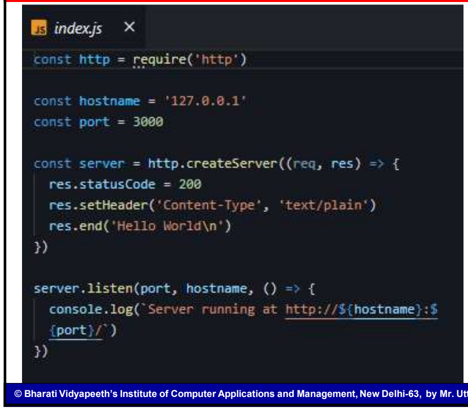
Node JS

- Node.js is an open-source and cross-platform JavaScript runtime environment.
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.
- A Node.js app runs in a single process, without creating a new thread for every request
- Node.js provides a set of asynchronous I/O primitives in its standard library
- When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread, Node.js will resume the operations when the response comes back

Node JS

- Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency
- npm with its simple structure helped the ecosystem of Node.js proliferate, and now the npm registry hosts over 1,000,000 open source packages you can freely use

Node JS



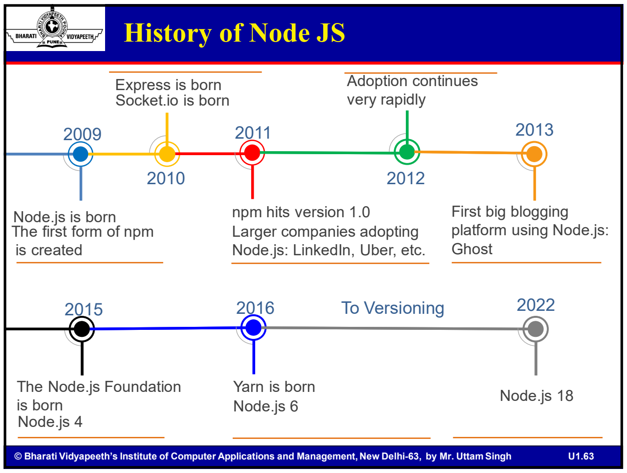
index.js

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:$
  {port}/`)
})
```

Node.js Frameworks and Tools

- AdonisJS: A TypeScript-based fully featured framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.
- Express: It provides one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
- hapi: A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.
- Loopback.io: Makes it easy to build modern applications that require complex integrations.
- Micro: It provides a very lightweight server to create asynchronous HTTP microservices.

History of Node JS



Express is born
Socket.io is born

Adoption continues
very rapidly

2009

2011

2013

2010

2012

Node.js is born
The first form of npm
is created

npm hits version 1.0
Larger companies adopting
Node.js: LinkedIn, Uber, etc.

First big blogging
platform using Node.js:
Ghost

2015

2016

To Versioning

2022

The Node.js Foundation
is born
Node.js 4

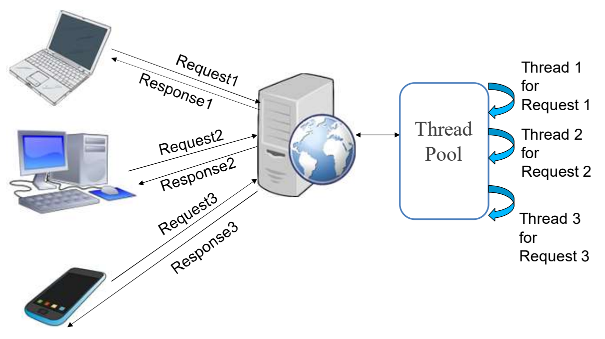
Yarn is born
Node.js 6

Node.js 18

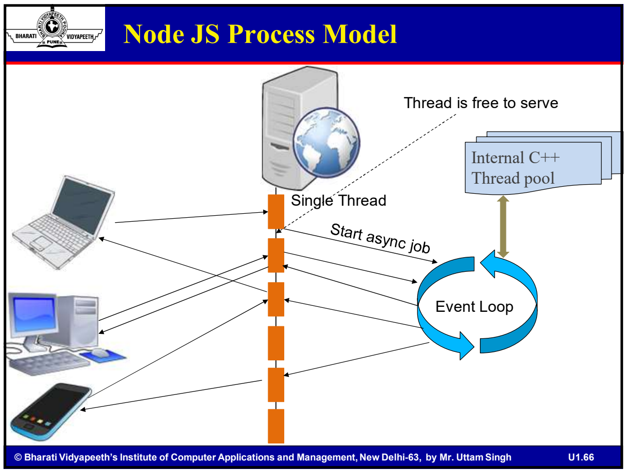
Node.js Frameworks and Tools

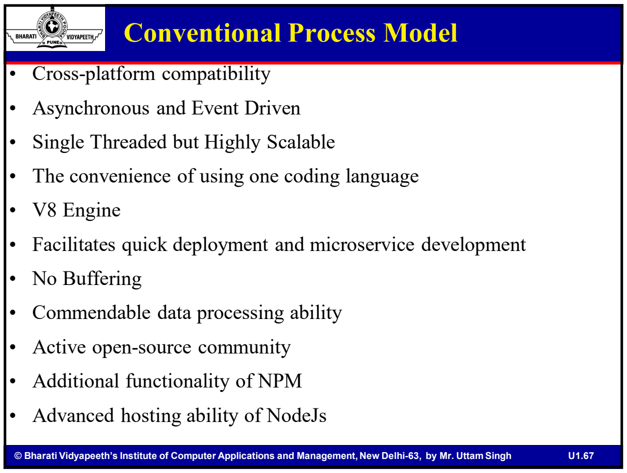
- Cross-platform compatibility
- Asynchronous and Event Driven
- Single Threaded but Highly Scalable
- The convenience of using one coding language
- V8 Engine
- Facilitates quick deployment and microservice development
- No Buffering
- Commendable data processing ability
- Active open-source community
- Additional functionality of NPM
- Advanced hosting ability of NodeJs

Conventional Process Model



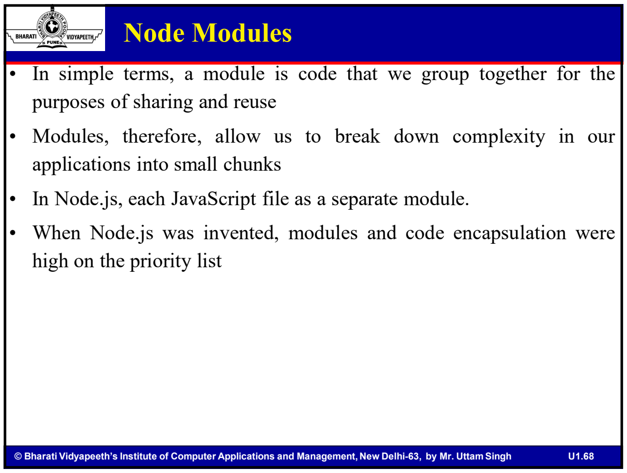
Node JS Process Model





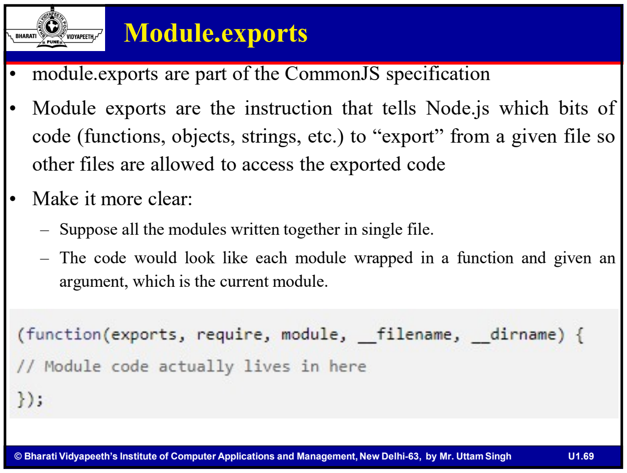
Conventional Process Model

- Cross-platform compatibility
- Asynchronous and Event Driven
- Single Threaded but Highly Scalable
- The convenience of using one coding language
- V8 Engine
- Facilitates quick deployment and microservice development
- No Buffering
- Commendable data processing ability
- Active open-source community
- Additional functionality of NPM
- Advanced hosting ability of NodeJs



Node Modules

- In simple terms, a module is code that we group together for the purposes of sharing and reuse
- Modules, therefore, allow us to break down complexity in our applications into small chunks
- In Node.js, each JavaScript file as a separate module.
- When Node.js was invented, modules and code encapsulation were high on the priority list

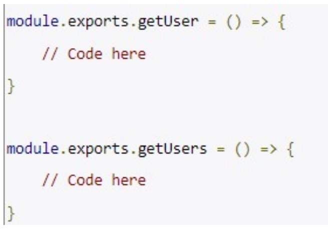


Module.exports

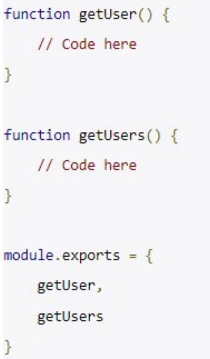
- module.exports are part of the CommonJS specification
- Module exports are the instruction that tells Node.js which bits of code (functions, objects, strings, etc.) to “export” from a given file so other files are allowed to access the exported code
- Make it more clear:
  - Suppose all the modules written together in single file.
  - The code would look like each module wrapped in a function and given an argument, which is the current module.

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

Exporting Module



```
module.exports.getUser = () => {
    // Code here
}

module.exports.getUsers = () => {
    // Code here
}
```

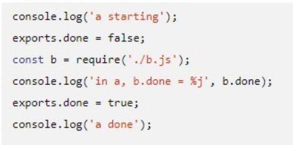


```
function getUser() {
    // Code here
}

function getUsers() {
    // Code here
}

module.exports = {
    getUser,
    getUsers
}
```

Cycle

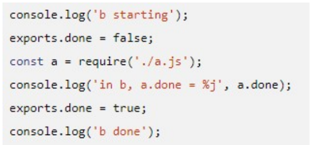
- When there are circular require() calls, a module might not have finished executing when it is returned.

a.js:



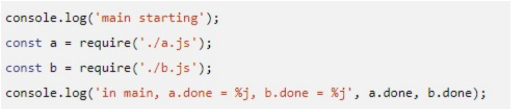
```
console.log('a starting');
exports.done = false;
const b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

b.js:



```
console.log('b starting');
exports.done = false;
const a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

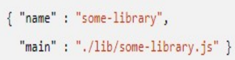
main.js:



```
console.log('main starting');
const a = require('./a.js');
const b = require('./b.js');
console.log('in main, a.done = %j, b.done = %j', a.done, b.done);
```

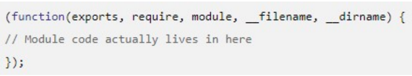
Modules

- If the exact filename is not found, then Node.js will attempt to load the required filename with the added extensions: .js, .json, and finally .node.
- There are three ways in which a folder may be passed to require() as an argument.
  - The first is to create a package.json file in the root of the folder, which specifies a main module.

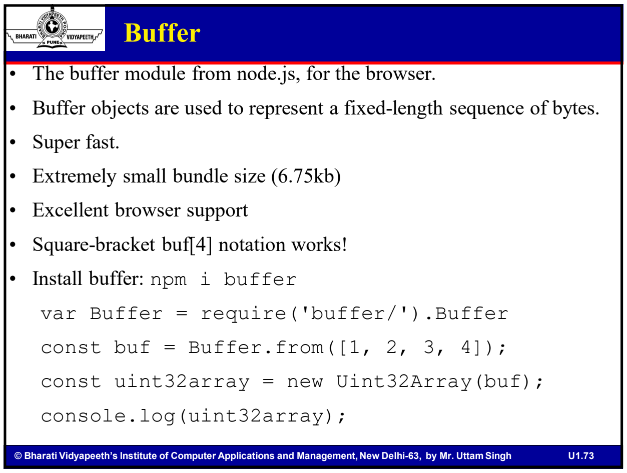


```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

- Loading from node_modules folders
- Loading from the global folders
  - Get a path from ENV Variables
- The module wrapper



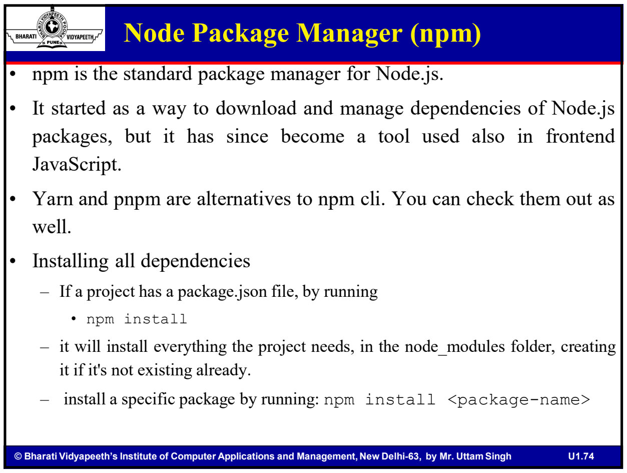
```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```



Buffer

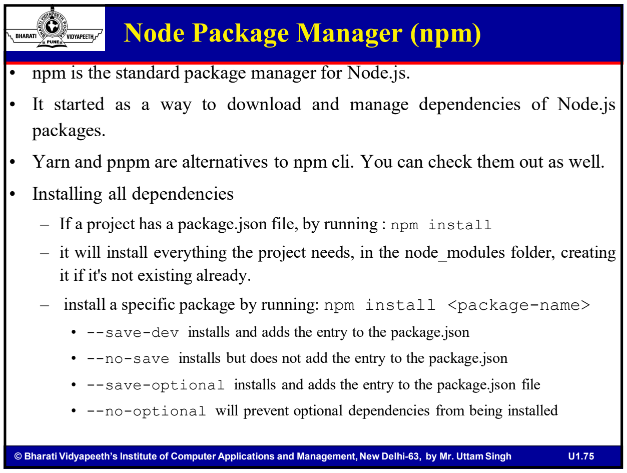
- The buffer module from node.js, for the browser.
- Buffer objects are used to represent a fixed-length sequence of bytes.
- Super fast.
- Extremely small bundle size (6.75kb)
- Excellent browser support
- Square-bracket buf[4] notation works!
- Install buffer: `npm i buffer`

```
var Buffer = require('buffer/').Buffer
const buf = Buffer.from([1, 2, 3, 4]);
const uint32array = new Uint32Array(buf);
console.log(uint32array);
```



Node Package Manager (npm)

- npm is the standard package manager for Node.js.
- It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.
- Yarn and pnpm are alternatives to npm cli. You can check them out as well.
- Installing all dependencies
  - If a project has a package.json file, by running
    - `npm install`
  - it will install everything the project needs, in the node_modules folder, creating it if it's not existing already.
  - install a specific package by running: `npm install <package-name>`



Node Package Manager (npm)

- npm is the standard package manager for Node.js.
- It started as a way to download and manage dependencies of Node.js packages.
- Yarn and pnpm are alternatives to npm cli. You can check them out as well.
- Installing all dependencies
  - If a project has a package.json file, by running : `npm install`
  - it will install everything the project needs, in the node_modules folder, creating it if it's not existing already.
  - install a specific package by running: `npm install <package-name>`
    - `--save-dev` installs and adds the entry to the package.json
    - `--no-save` installs but does not add the entry to the package.json
    - `--save-optional` installs and adds the entry to the package.json file
    - `--no-optional` will prevent optional dependencies from being installed

Node Package Manager (global / local)

- **local packages** are installed in the directory where you run npm install <package-name>, and they are put in the node_modules folder under this directory
- **global packages** are all put in a single place in your system (exactly where depends on your setup), regardless of where you run npm install -g <package-name>
- A package **should be installed globally** when it provides an executable command that you run from the shell (CLI), and it's reused across projects.
- All projects have their own local version of a package, even if this might appear like a waste of resources, it's minimal compared to the possible negative consequences.
- You can also install executable commands locally and run them using npx, but some packages are just better installed globally.

Node Package Manager (package.json)

- The package.json file is kind of a manifest for your project.
- It's a central repository of configuration for tools
- It's also where npm and yarn store the names and versions for all the installed packages.

Node Package Manager (package.json)

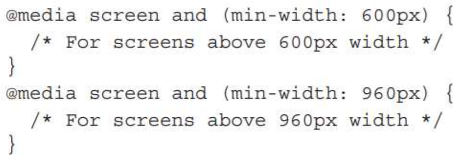
- JSON file Structure
  - `version` indicates the current version
  - `name` sets the application/package name
  - `description` is a brief description of the app/package
  - `main` sets the entry point for the application
  - `private` if set to true prevents the app/package to be accidentally published on npm
  - `scripts` defines a set of node scripts you can run
  - `dependencies` sets a list of npm packages installed as dependencies
  - `devDependencies` sets a list of npm packages installed as development dependencies
  - `engines` sets which versions of Node.js this package/app works on
  - `browserslist` is used to tell which browsers (and their versions) you want to support

Node Package Manager

- Update All Packages
- Install the npm-check-updates package globally:
  - `npm install -g npm-check-updates`
- Now run **npm-check-updates** to upgrade all version hints in package.json, allowing installation of the new major versions
  - `ncu -u`
- Finally, run a standard install
  - `npm install`

Mobile-first paradigm

- responsive web design
- accommodates the screen size and other device attributes
- mobile-first, mean that the design an application first to work well on a mobile device and then move on to devices with larger screens.
- In Stylesheet, required Media queries where, for certain sized screens, the styles defined for mobile devices are overridden to make sense for devices with larger screens.



```
@media screen and (min-width: 600px) {
  /* For screens above 600px width */
}
@media screen and (min-width: 960px) {
  /* For screens above 960px width */
}
```

Mobile-first paradigm

- At least target these device scenarios:
- **Small**: This includes iPhone 4.
- **Medium**: This can refer to tablet computers, or the larger smart phones.
- **Large**: This includes larger tablet computers, or the smaller desktop computers.
- **Extra-large**: This refers to larger desktop computers and other large screens.
- **Landscape/portrait**: You may want to create a distinction between landscape mode and portrait mode. Switching between the two of course changes viewport width, possibly pushing it past a breakpoint

Using Twitter Bootstrap on the Notes application

- Setting it up
- `npm i bootstrap`
- `import ‘./node_modules/bootstrap/dist/css/bootstrap.min.css’ ;`
- `import ‘./node_modules/bootstrap/dist/js/bootstrap.min.js’;`

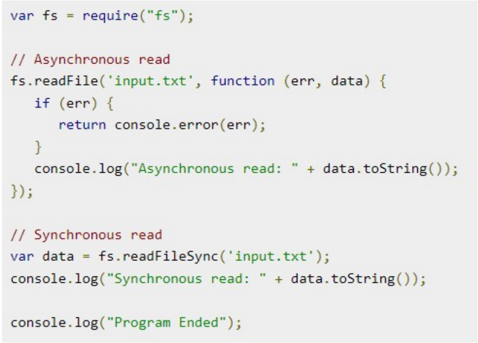
CSS Flexbox and Grid CSS

- Grid vs. Flexbox
  - Grid is made for two-dimensional layout while Flexbox is for one. **Flexbox** can work on **either row or columns at a time**, but **Grids** can work on **both**.
  - Flexbox, gives you more flexibility while working on either element (row or column). HTML markup and CSS will be easy to manage in this type of scenario.
  - GRID gives you more flexibility to move around the blocks irrespective of your HTML markup.
  - Flex Direction allows developers to align elements vertically or horizontally while, In CSS Grid, auto-keyword functionality to automatically adjust columns or rows.

Synchronous vs Asynchronous

- `npm i fs –save`
- Every method in the **fs** module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Synchronous vs Asynchronous



```
var fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());

console.log("Program Ended");
```

https://www.tutorialspoint.com/nodejs/nodejs_file_system.htm

File operations

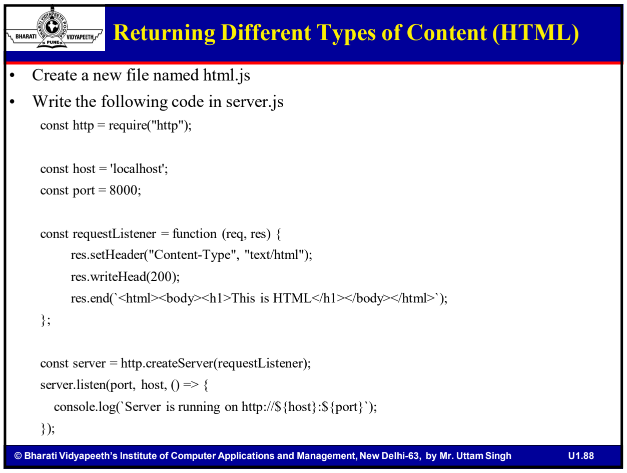
- **r**: Open file for reading.
- **r+:** Open file for reading and writing
- **rs:** Open file for reading in synchronous mode.
- **w:** Open file for writing.
- **wx:** Like 'w' but fails if the path exists.
- **w+:** Open file for reading and writing.
- **a:** Open file for appending.
- **a+:** Open file for reading and appending.

Creating a Basic HTTP Server

- First create a folder to manage server
  - mkdir http-server
- Create a new file named server.js
- Write the following code in server.js

```
const http = require("http");
const host = 'localhost';
const port = 8000;
const requestListener = function (req, res) {
   res.writeHead(200);
   res.end(“First HTTP server!");
};
const server = http.createServer(requestListener);
server.listen(port, host, () => {
   console.log(`Server is running on http://${host}:${port}`);
});
>node server.js
```

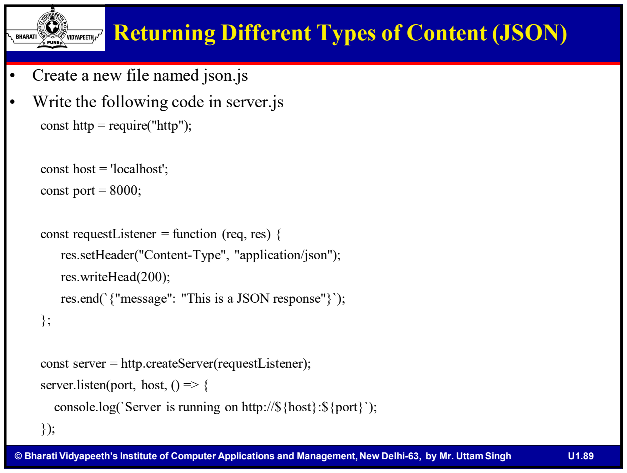
https://www.digitalocean.com/community/tutorials/how-to-create-a-web-server-in-node-js-with-the-http-module



Returning Different Types of Content (HTML)

- Create a new file named html.js
- Write the following code in server.js

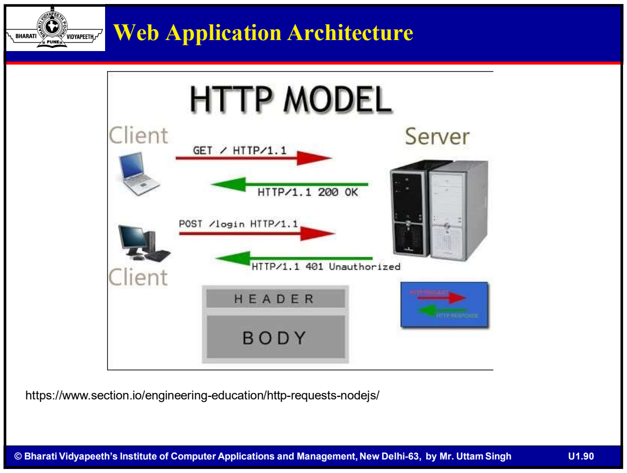
```
const http = require("http");

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {
        res.setHeader("Content-Type", "text/html");
        res.writeHead(200);
        res.end(`<html><body><h1>This is HTML</h1></body></html>`);
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```



Returning Different Types of Content (JSON)

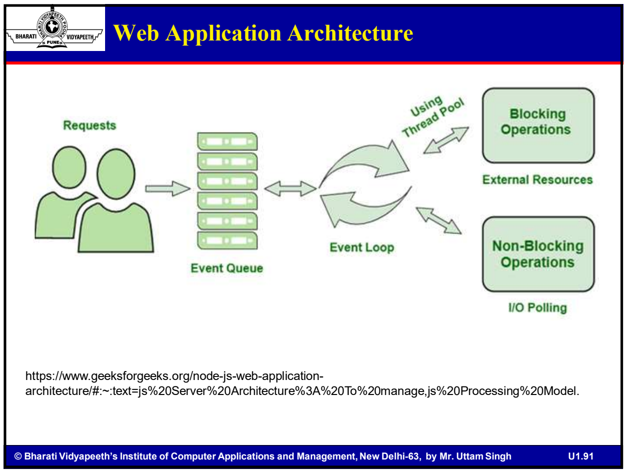
- Create a new file named json.js
- Write the following code in server.js

```
const http = require("http");

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {
      res.setHeader("Content-Type", "application/json");
      res.writeHead(200);
      res.end(`{"message": "This is a JSON response"}`);
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```



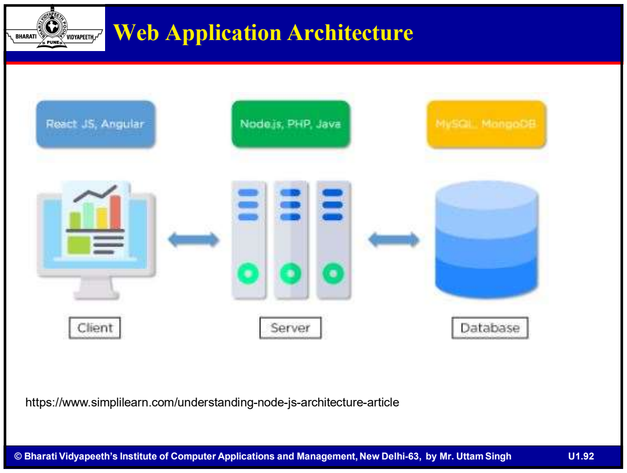
Web Application Architecture

https://www.section.io/engineering-education/http-requests-nodejs/



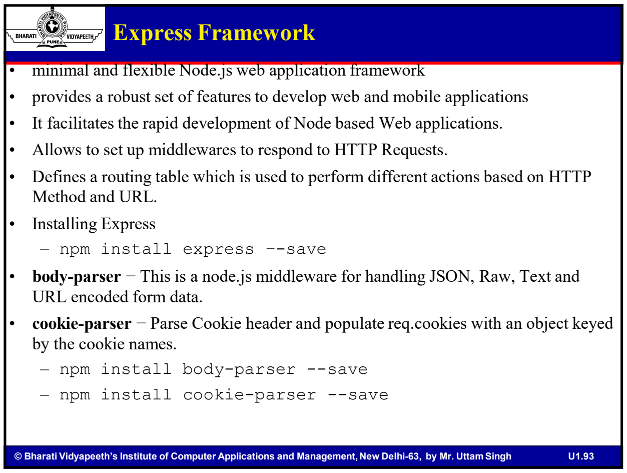
Web Application Architecture

https://www.geeksforgeeks.org/node-js-web-application-architecture/#:~:text=js%20Server%20Architecture%3A%20To%20manage,js%20Processing%20Model.



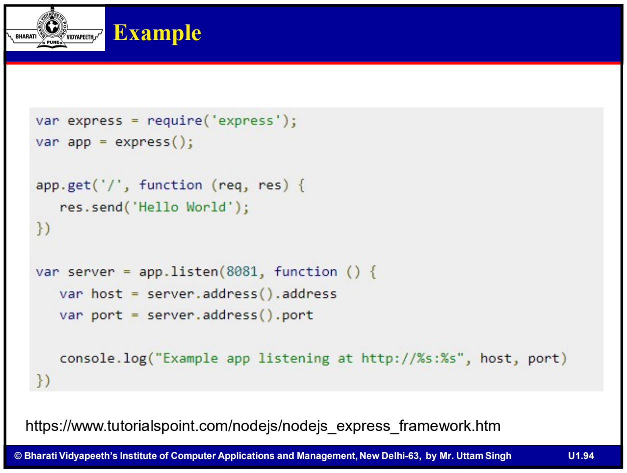
Web Application Architecture

https://www.simplilearn.com/understanding-node-js-architecture-article



Express Framework

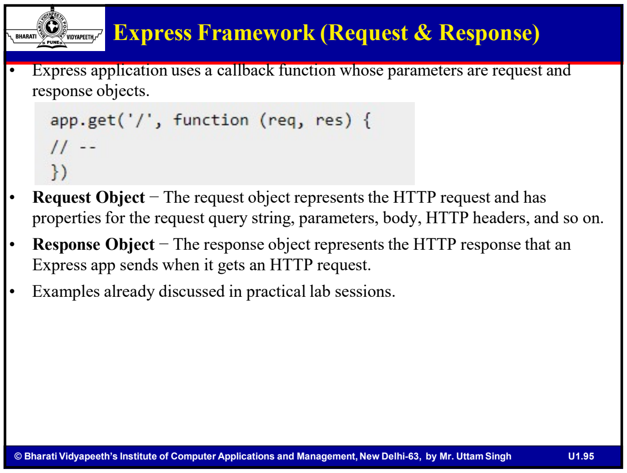
- minimal and flexible Node.js web application framework
- provides a robust set of features to develop web and mobile applications
- It facilitates the rapid development of Node based Web applications.
- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Installing Express
  - `npm install express --save`
- **body-parser** − This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser** − Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
  - `npm install body-parser --save`
  - `npm install cookie-parser --save`



Example

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host, port)
})
```

https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

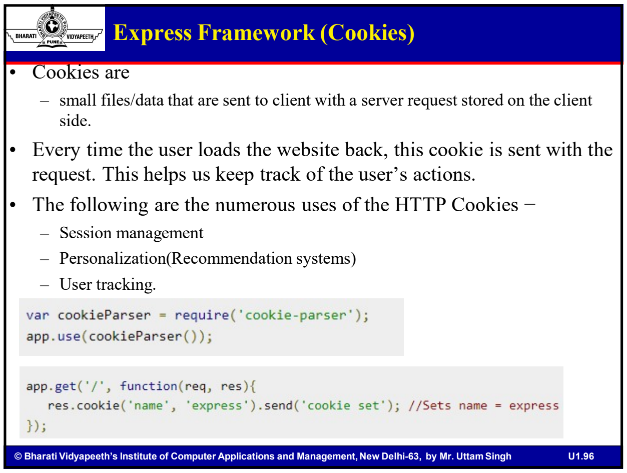


Express Framework (Request & Response)

- Express application uses a callback function whose parameters are request and response objects.

```
app.get('/', function (req, res) {
// --
})
```

- **Request Object** – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- **Response Object** – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.
- Examples already discussed in practical lab sessions.

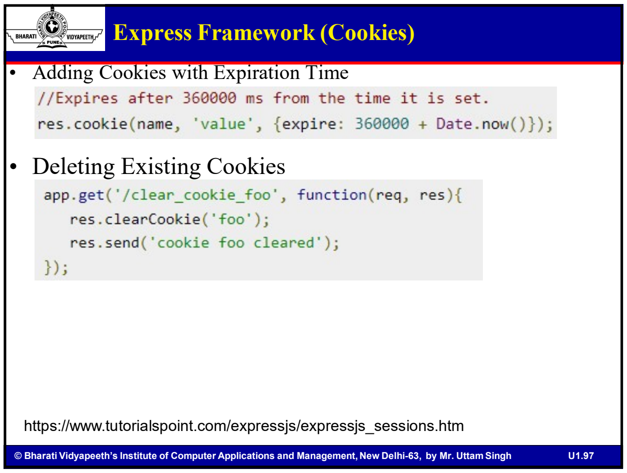


Express Framework (Cookies)

- Cookies are
  - small files/data that are sent to client with a server request stored on the client side.
- Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user's actions.
- The following are the numerous uses of the HTTP Cookies –
  - Session management
  - Personalization(Recommendation systems)
  - User tracking.

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

```
app.get('/', function(req, res){
   res.cookie('name', 'express').send('cookie set'); //Sets name = express
});
```



Express Framework (Cookies)

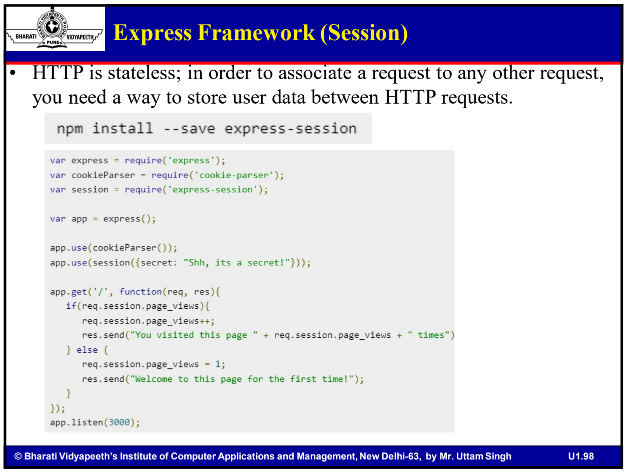
- Adding Cookies with Expiration Time

```
//Expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

- Deleting Existing Cookies

```
app.get('/clear_cookie_foo', function(req, res){
   res.clearCookie('foo');
   res.send('cookie foo cleared');
});
```

https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

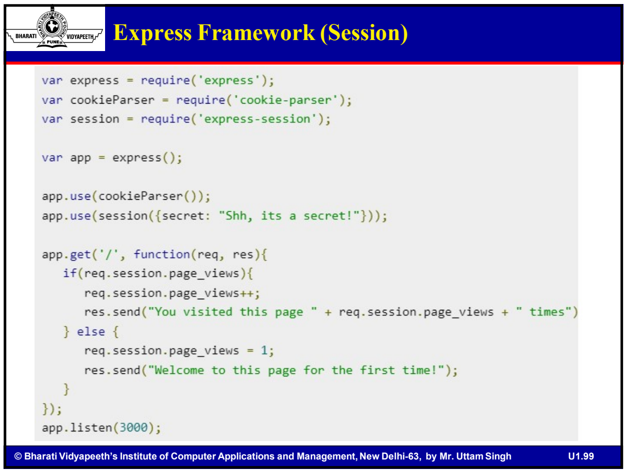


Express Framework (Session)

- HTTP is stateless; in order to associate a request to any other request, you need a way to store user data between HTTP requests.

```
npm install --save express-session
```

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');

var app = express();

app.use(cookieParser());
app.use(session({secret: "Shh, its a secret!"}));

app.get('/', function(req, res){
   if(req.session.page_views){
      req.session.page_views++;
      res.send("You visited this page " + req.session.page_views + " times")
   } else {
      req.session.page_views = 1;
      res.send("Welcome to this page for the first time!");
   }
});
app.listen(3000);
```



Express Framework (Session)

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');

var app = express();

app.use(cookieParser());
app.use(session({secret: "Shh, its a secret!"}));

app.get('/', function(req, res){
   if(req.session.page_views){
      req.session.page_views++;
      res.send("You visited this page " + req.session.page_views + " times")
   } else {
      req.session.page_views = 1;
      res.send("Welcome to this page for the first time!");
   }
});
app.listen(3000);
```