# Prerequisites-

Sound understanding of HTML, CSS, JavaScript and Webpack

Interest to learn

# Agenda

- JavaScript / ES6+ features
- Webpack
- MVC Architecture
- AngularJS Environment Setup
- AngularJS
  - Modules
  - Controllers
  - Directives
  - Filters
  - Services
- Data Binding
- Form Data
- Testing
- Redux Integration

# JavaScript - Concepts

## Scope chaining

- Inner scope → outer Scope

## Prototyping

- Object blueprint

## Closures

- Process of binding the external variables with the inner functions

# ES6+ Features

- Arrow functions
- Destructuring
- Rest/Spread Operators
- Maps & Sets
- Classes & Modules
- Template Literals
- Block Scoping
- Object.assign()
- Optional Parameter

# Webpack

Static module bundler for modern JavaScript applications.

How many people are still using jQuery to get/update their HTML document data?

**There is a better way**

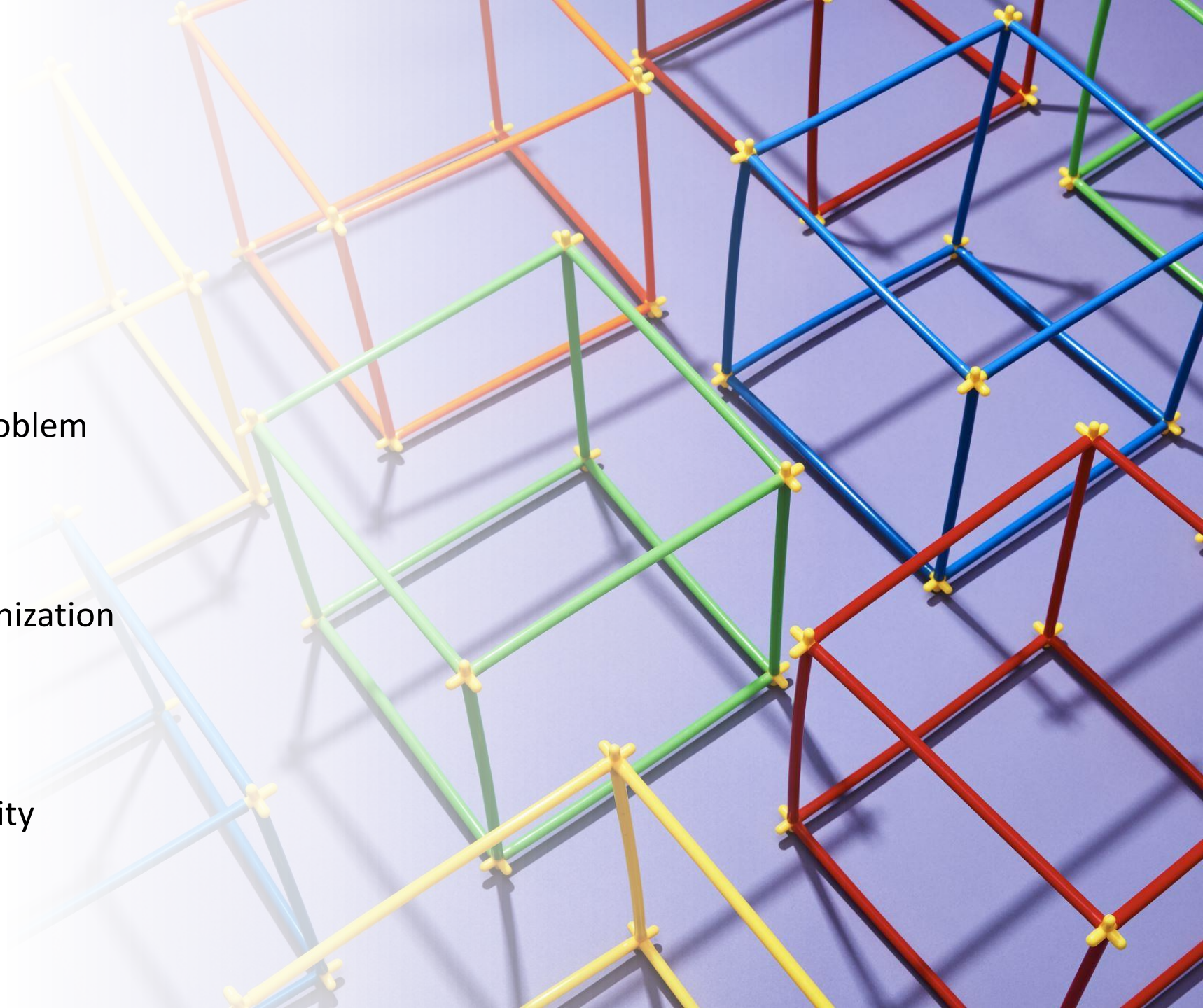
# AngularJS

*AngularJS is a JavaScript MVC library developed by Google that lets you build well structured, easily testable, and maintainable front-end applications.*

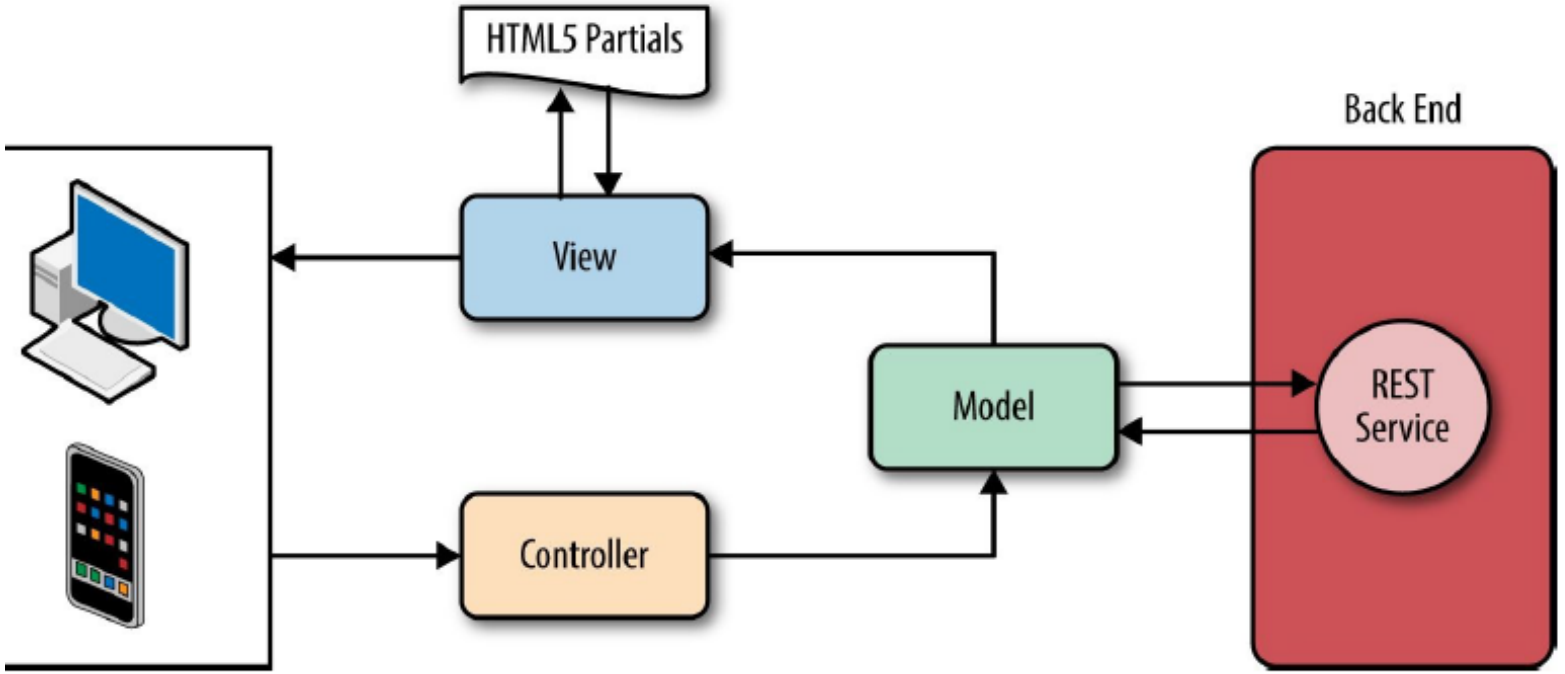
# Why AngularJS?

AngularJS addresses the root problem that HTML was not designed for dynamic views :

- Structure, Quality and Organization
- Lightweight & Free
- Separation of concern
- Modularity
- Extensibility & Maintainability
- Reusable Components



# MVC Architecture




- The *Model* is the truth and driving force of the application.
- The *View* is the UI that the user sees and interacts with.
- The *Controller* is the business logic and presentation layer.

# Environment Set-up

- Any preferred editor
  - VS Code / Sublime / Notepad++
- Updated Browser
  - Chrome / Firefox / IE
- Node Installer
- Permission to download libraries

# Angular Building Blocks

# Modules

Where we write pieces of our Angular application

Modules can define Controllers, Services, Factories & Directives

Makes our code more maintainable, testable and readable

Where we define the dependencies of our app.

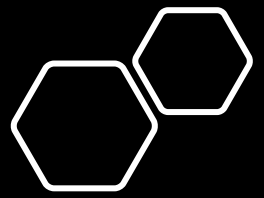
Modules can use other existing modules

# {{ Expressions }}

Angular expressions are JavaScript-like code snippets that are mainly placed in interpolation bindings

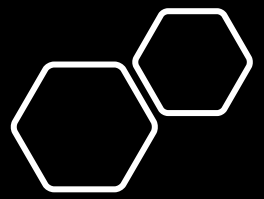
Allow you to insert dynamic values into your HTML

Shouldn’t use expressions to implement any higher-level logic.



# Controllers

- A Controller is defined by a JavaScript **constructor function** that is used to augment the Angular Scope.
- Controllers are where we define our app's behavior by defining functions and values
- Controllers are associated to tags, not accessible outside the boundary.
- Controller can be used to Set up the initial state, Adding behaviour etc.

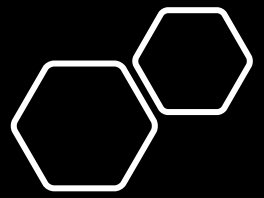


# Directives

- Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class)
- Attach specified behaviour to the DOM element
- Transforms the DOM element and its children
- Angular **normalizes** an element's tag and attribute name to determine which elements match which directives

# Few built-in Directives

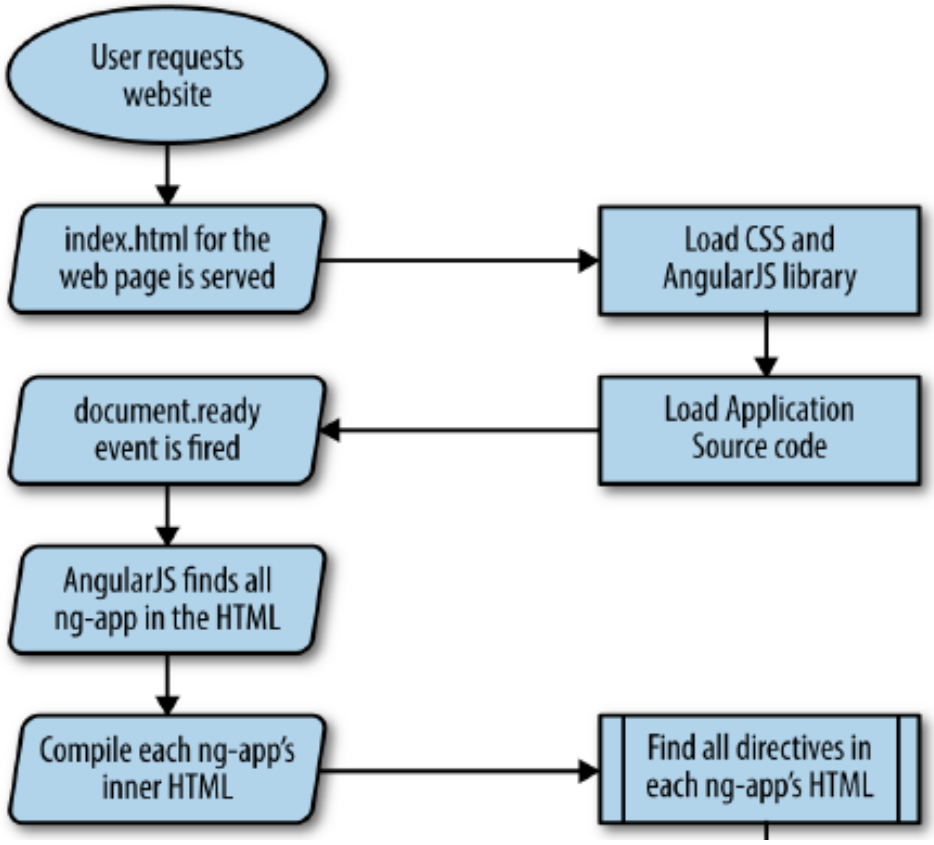
- ng-bind
- ng-show
- ng-hide
- ng-repeat
- ng-scope
- ng-class
- ng-click
- ng-cloak
- ng-model
- ng-src

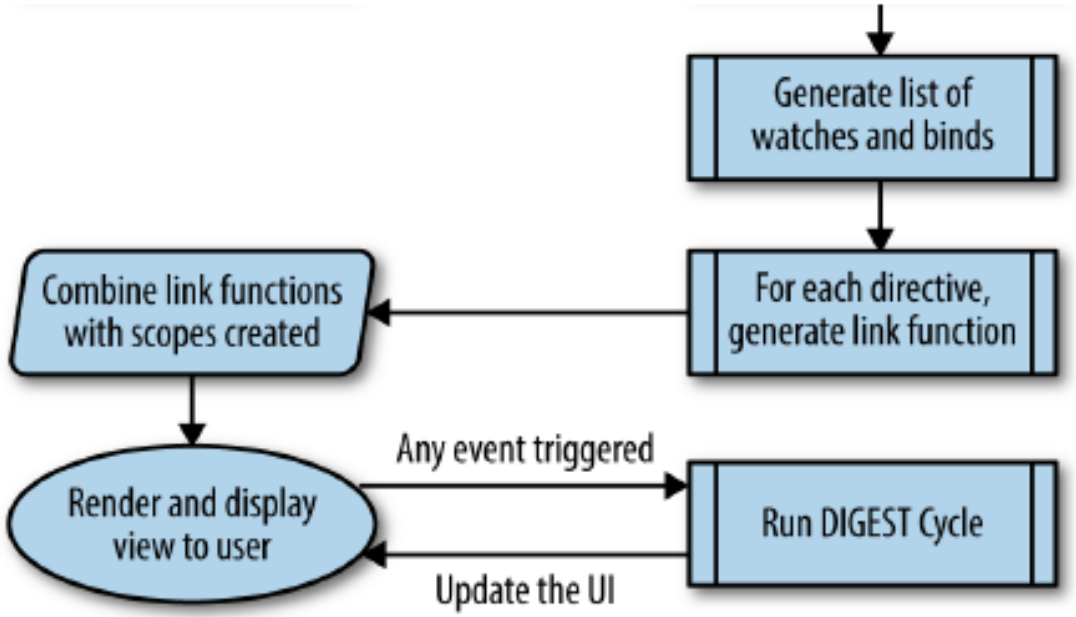


# Filters

- Filters format the value of an expression for display to the user.
- Filtered values are shown to the user but do not modify the original value on which they are applied.
- Can be used in view templates, controllers or services
- Attaching the word “**Filter**” after any AngularJS filter allows us to inject it into our controllers or services.
- Angular comes with a collection of built-in filters :
  - currency
  - lowerCase
  - upperCase
  - date
  - limitTo
  - orderBy

# Life Cycles in AngularJS

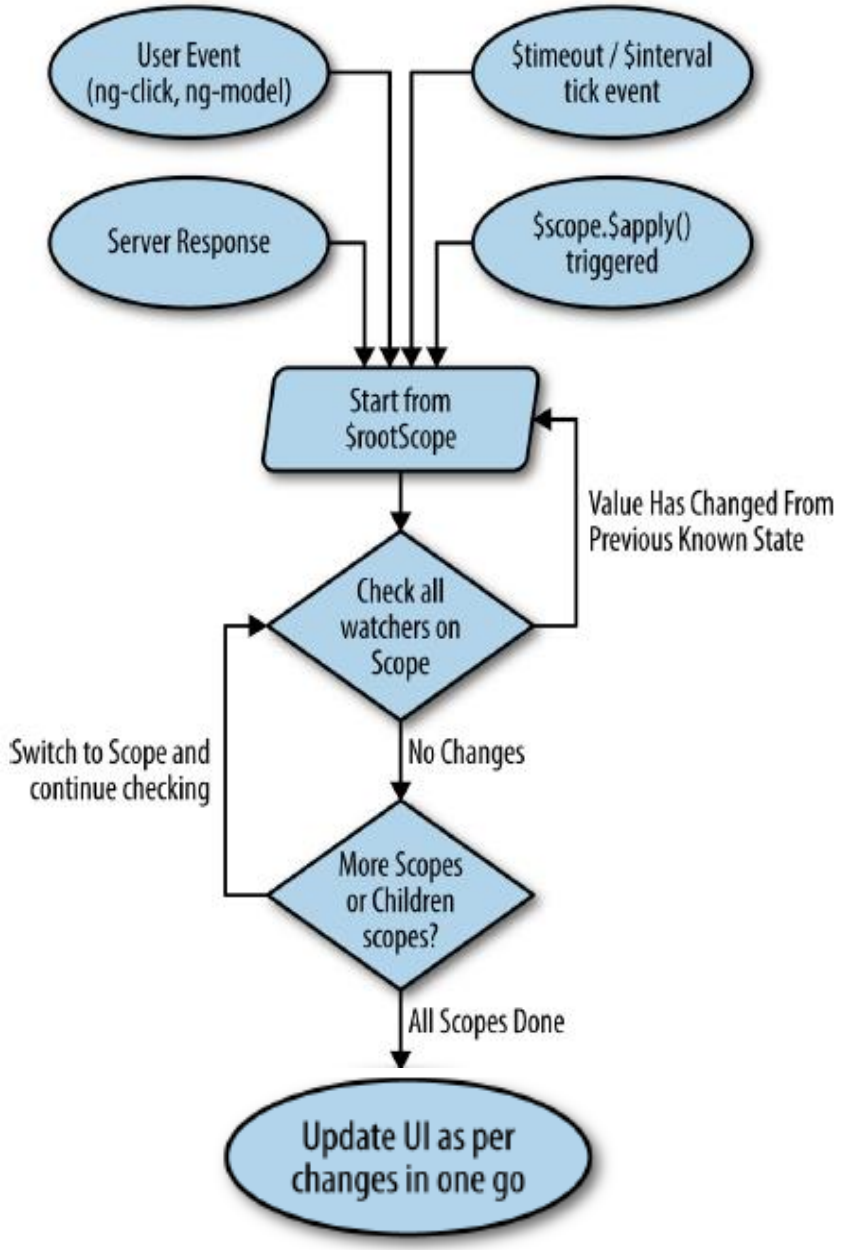


# Life Cycles in AngularJS

1. The HTML page is loaded:
    a. The HTML page loads the AngularJS source code (with jQuery optionally loaded before).
    b. The HTML page loads the application's JavaScript code.
2. The HTML page finishes loading.
3. When the document ready event is fired, AngularJS bootstraps and searches for any and all instances of the ng-app attribute in the HTML
4. Within the context of each ng-app, AngularJS starts its magic by running the HTML content inside ng-app through the compile step.
5. AngularJS takes the link function and combines it with scope. The scope has the variables and contents that need to be displayed in the UI. This combination generates the *live view that the user sees and interacts with.*
6. At the end of this, we have a live, interactive view with the content filled in for the user.

# The Digest Cycle



# The Digest Cycle

1. When any HTML is loaded, AngularJS runs its compilations step, and keeps track of all the watchers and listeners that are needed for the HTML . When linked with the scope, we get the actual current values that are then displayed in the UI.
2. AngularJS also keeps track of all the elements that are bound to the HTML for each  scope.
3. When one of the events mentioned in the previous section (such as a user click) happens, AngularJS triggers the digest cycle.
4. In the digest cycle, AngularJS starts from $rootScope and checks each watcher in the scope to see if the current value differs from the value it's displaying in the UI.
5. If nothing has changed, it recurses to all the parent scopes and so on until all the scopes are verified.
6. If AngularJS finds a watcher at any scope that reports a change in state, AngularJS stops right there, and *reruns the digest cycle.*
7. The digest cycle is rerun because a change in a watcher might have an implication on a watcher that was already evaluated beforehand. To ensure that no data change is missed, the digest cycle is rerun.
8. AngularJS reruns the digest cycle every time it encounters a change until the digest cycle stabilizes.
9. When the digest stabilizes, AngularJS accumulates all the UI updates and triggers them at once.

# $watch

- The $scope.watch() function creates a watch of some variable.
- When you register a watch you pass two functions as parameters to the $watch() function:
  - A value function
  - A listener function
- The value function should return the value which is being watched.
- AngularJS can then check the value returned against the value the watch function returned the last time. That way AngularJS can determine if the value has changed.

# $digest

- The $scope.$digest() function iterates through all the watches in the $scope object. When $digest() iterates over the watches, it calls the value function for each watch. If the value returned by the value function is different than the value it returned the last time it was called, the listener function for that watch is called.
- The $digest() function is called whenever AngularJS thinks it is necessary. For instance, after a button click handler has been executed, or after an AJAX call returns.

# $apply

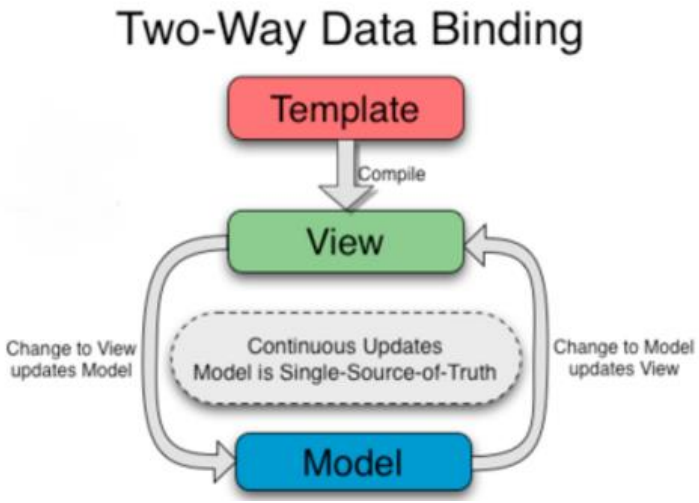
- The $scope.$apply() function takes a function as parameter which is executed and after that $scope.$digest() is called internally. That makes it easier for you to make sure that all watches are checked, and thus all data bindings refreshed.
- Sometimes AngularJS does not call the $digest() function for you. You will usually detect that by noticing that the data bindings do not upate the displayed values. In that case, call $scope.$digest() and it should work (you can perhaps use $scope.$apply() instead)

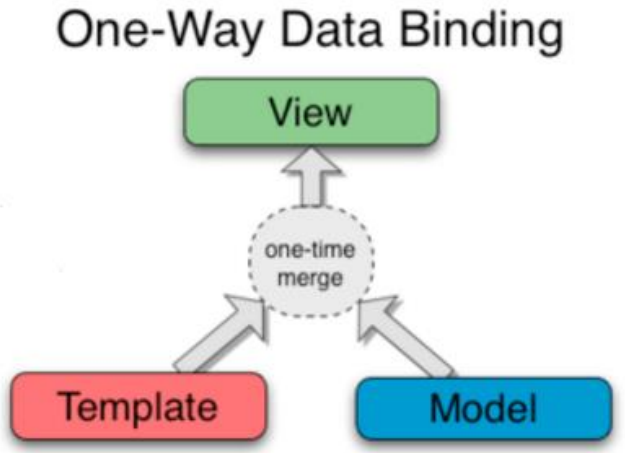
# Working with Forms

- Most applications have user interaction, and parts where the user has to feed in data.
- Angular provides the **ng-model** directive to deal with inputs and two-way data-binding.
- Angular benefit us when working with forms, especially with validation and various states of the forms and inputs.

# Data Binding

- Data-binding in Angular apps is the automatic synchronization of data between the model and view components.
- When the model changes, the view reflects the change, and vice-versa



# Form States

| Form State | Description |
| --- | --- |
| $invalid | AngularJS sets this state when any of the validations (**required, ng-minlength, and others**) mark any of the fields within the form as invalid. |
| $valid | The inverse of the previous state, which states that all the validations in the form are currently evaluating to correct. |
| $pristine | All forms in AngularJS start with this state. This allows you to figure out if a user has started typing in and modifying any of the form elements. |
| $dirty | The inverse of $pristine, which states that the user made some changes (s/he can revert it, but the $dirty bit is set). |
| $error | This field on the form houses all the individual fields and the errors on each form element. |

# Dependency Injection (DI)

- Dependency Injection is a concept to propagate reuse, modularity, and testability of code.
- Dependency Injection states that instead of creating an instance of a dependent service when we need it, the class or function should ask for it instead.
- Service concept in AngularJS is heavily dependent on and driven by its Dependency Injection system
- Any service known to AngularJS (internal or our own) can be injected into other service, directive, or controller by stating it as a dependency.

# Services

AngularJS services are functions or objects that can hold behaviour or state across our application

Each AngularJS service is instantiated only once, so each part of our application gets access to the same instance of the AngularJS service.

Repeated behaviour, shared state, caches, factories, etc. are all functionality that can be implemented using AngularJS services.

# Service vs. Controller

| Controllers | Services |
| --- | --- |
| Presentation logic | Business logic |
| Directly linked to view | Independent of view |
| Drives the UI | Drives the application |
| One-off, specific | Reusable |
| Responsible for decisions like what data to fetch, what data to show, how to handle user interactions, and styling and display of UI | Responsible for making server calls, common validation logic, application-level stores, and reusable business logic |

# Service Types

| | |
|---|---|
| Factory | Register the service factory function for the Angular module |
| Service | No need to create an additional service object and a new member function can be added directly to service definition. Shares instances of methods and objects. |
| Provider | It helps you in initializing the configuration before any service call. Shares methods and objects (like a Factory), but allows for configuration. |
| Value | The simplest service recipe used for sharing a value that is used throughout your app repeatedly. |
| Constant | Supplies a constant. Shares a value within application configuration. |

# Server Communication using $http

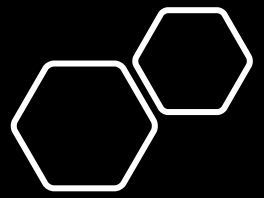
- $http is a core AngularJS service that allows us to communicate with server endpoints using *XHR*.
- $http Service is used to make HTTP requestd to remote server.
- $http service is a function that has single input parameter i.e. *configuration object*.
- AngularJS XHR API follows the ***Promise interface***.

# Promises

- Promises in AngularJS are standardized, convenient way of dealing with asynchronous calls in JavaScript.
- The Promise API was designed to solve this nesting problem and the problem of error handling.
- Promise functioning -
  - Each asynchronous task will return a promise object.
  - Each promise object will have a then function that can take two arguments, a success handler and an error handler.
  - The success or the error handler in the then function will be called only once, after the asynchronous task finishes.
  - The then function will also return a promise, to allow chaining multiple calls.
  - Each handler (success or error) can return a value, which will be passed to the next function in the chain of promises.
  - If a handler returns a promise (makes another asynchronous request), then the next handler (success or error) will be called only after that request is finished

# Routing & Navigation

- Hash URL vs. Non-hash URL
- The ngRoute module provides routing and deep-linking services and directives for angular apps.
- Steps for routing :
  - Include the routing module source code in the application's HTML : **angular-route**.
  - Include the module as a dependency of our main AngularJS app module : **ngRoute**.
  - Mark which section of the page AngularJS should change when the route changes.
  - Define our routes in the **config** section using the **$routeProvider** service.
  - Show the route template using **ng-view**



# Custom Directives :

- Dealing directly with the UI or the HTML that the user sees
- Directives are of two major types in AngularJS :
  - Behaviour Directives
  - Reusable Components

# Directive Life Cycle

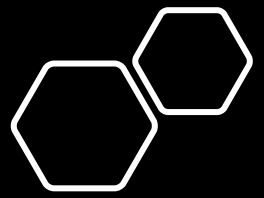
When the application loads, the directive definition object is triggered. This happens only once.

Next, when the directive is encountered in the HTML the very first time, the template for the directive is loaded.

This template is then compiled and AngularJS handles the other directives present in the HTML. This generates a link function that can be used to link the directive to a scope.

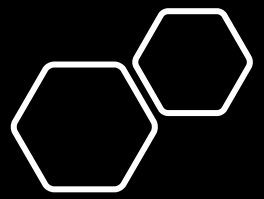
The scope for the directive instance is created or acquired. This could be the parent scope, a child of the parent scope, or an isolated scope

The link function (and the controller) execute for the directive.



# Unit Testing

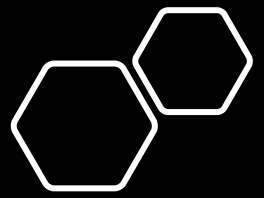
Unit testing is the concept of taking a single function or part of our code, and writing assertions and tests to ensure that it works as intended.



# Karma

- Karma is the test runner that makes running tests painless and fast.
- Karma uses **NodeJS** and **SocketIO** to facilitate tests in multiple browsers at fast speed.

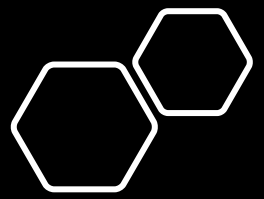
| STEPS | DESCRIPTION | COMMAND |
|---|---|---|
| 1 | Install NodeJS | `http://nodejs.org` |
| 2 | Install Karma CLI | `npm install –g karma-cli` |
| 3 | Install Karma Locally | `npm install karma` |
| 4 | Install Jasmine plugin & browser launcher | `npm install karma-jasmine jasmine-core karma-chrome-launcher` |



# Jasmine

Jasmine framework uses behaviour-driven style of writing tests. Instead of writing a bunch of functions and asserts, we describe behaviours and set expectations.

| # | Jasmine Syntax | Description |
|---|---|---|
| 1 | describe | a container for multiple unit tests.<br>write a describe for a controller, for a service etc. |
| 2 | beforeEach | function you pass to ***beforeEach*** will be executed before each individual ***it*** block. |
| 3 | afterEach | the ***afterEach*** block gets executed after the individual ***it*** blocks are completed. |
| 4 | it | These are the unit tests.<br>Each it block should be self-contained, and independent of all the other it blocks |
| 5 | expect | These are the Jasmine equivalents of assert statements.<br>Each expect takes a value, and then you can use one of the built-in matchers (or create your own) to check its value. |

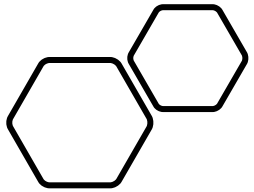


# Useful Jasmine Matchers

| Matchers | Description |
|---|---|
| toEqual | takes a second value and does a deep equality check between the two objects. |
| toBe | checks for reference, and expects both items passed to the expect and the matcher to be the exact same object reference. |
| toBeTruthy | Checks the value passed to the matcher to pass the JavaScript concept of true |
| toBeFalsy | Checks the value passed to the matcher to pass the JavaScript concept of false. |
| toBeDefined | Ensures the reference passed to the ***expect*** is defined |
| toBeUndefined | Checks if the reference passed to the ***expect*** is undefined or not set. |
| toBeNull | Checks if the reference passed to the ***expect*** is null. |
| toContain | Checks if the array passed to the ***expect*** contains the element passed to the matcher, ***toContain***. |
| toMatch | Used for regular expression checks when the first argument to the expect is a string that needs to match a specific regular expression pattern. |

# Redux

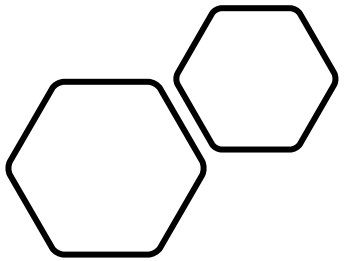
A Predictable State Container for JavaScript Apps



# Should I use Redux?

- You have reasonable amounts of data changing over time
- You need a single source of truth for your state
- You find that keeping all your state in a top-level component is no longer sufficient

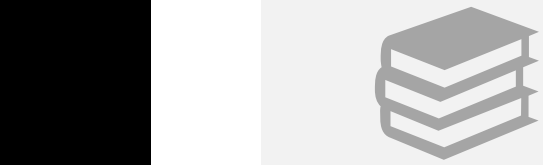
# Redux



- Predictable state container for JavaScript Apps
- It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.
- It provides a great developer experience, such aslive code editing combined with a time traveling debugger.

# Three Principles' of Redux

Single Source of Truth



State is read-only

Changes are made with Pure functions

# Actions

*Actions* are payloads of information that send data from your application to your *store*.

They are the only source of information for the store.

You send them to the store using store.dispatch().

# Reducers

The *reducer* is a pure function that takes the previous *state* and an *action*, and returns the next state.

*Actions* only describe *what happened*, but don't describe how the application's state changes.

**Reducers** specify how the application's state changes in response to *actions* sent to the *store*.

# Reducers : Don’ts

Mutate its arguments.

Perform side effects like API calls and routing transitions.

Call non-pure functions, e.g. Date.now() or Math.random().

# Store

The Store is the single object that has following responsibilities –

- Holds application state.
- Allows access to state via getState().
- Allows state to be updated via dispatch(action).
- Registers listeners via subscribe(listener).
- Handles unregistering of listeners via the function returned by subscribe(listener).

# Data Flow

- Redux architecture revolves around a **strict unidirectional data flow**.
- The data lifecycle in any Redux app follows these 4 steps:
  - You call `store.dispatch(action)`
  - The Redux store calls the reducer function you gave it
  - The root reducer may combine the output of multiple reducers into a single state tree
  - The Redux store saves the complete state tree returned by the root reducer

# Redux Middleware - Thunk

Thunks are middlewares for basic Redux side effects logic including complex synchronous operation that needs access to the store and/or async operation like AJAX requests.

Redux Thunk middleware allows you to write action creators that return a function instead of an action.

The thunk can be used to delay the dispatch of an action or to dispatch only if a certain condition is met. The inner function receives the store methods **dispatch** and **getState** as parameters.

# References

## Web -

- http://javascript.info
- https://redux.js.org
- https://stackoverflow.com
- https://www.npmjs.com
- https://nodejs.org
- https://docs.angularjs.org/api
- https://webpack.js.org/

## Book –

- AngularJS – Up & Running by Shyam Seshadri & Brad Green