Angular Compiler
Redux State Management

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

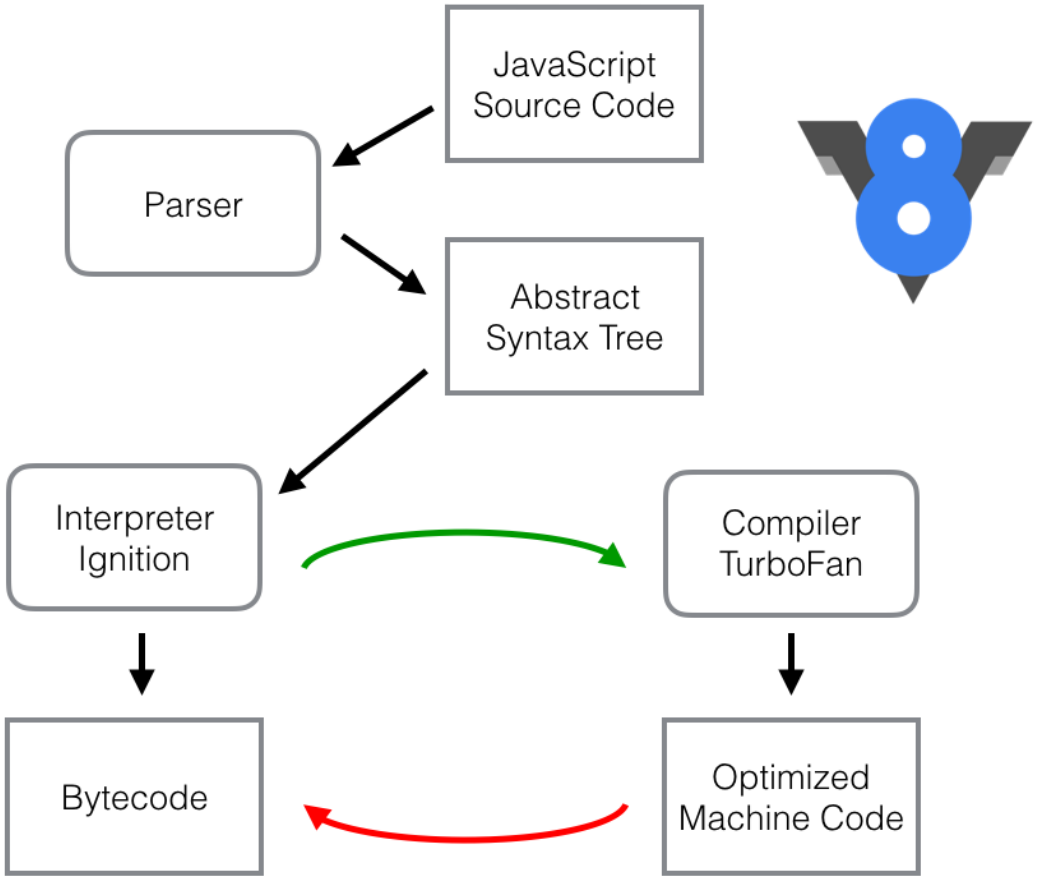
Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

JavaScript Performance

We almost expect scripts to be immediately parsed and executed as soon as the parser hits a <script> tag. But this isn't quite the case. Here's a simplified breakdown of how V8 works:

Once the browser has downloaded our page's scripts it then has to parse, interpret, optimize & run them.

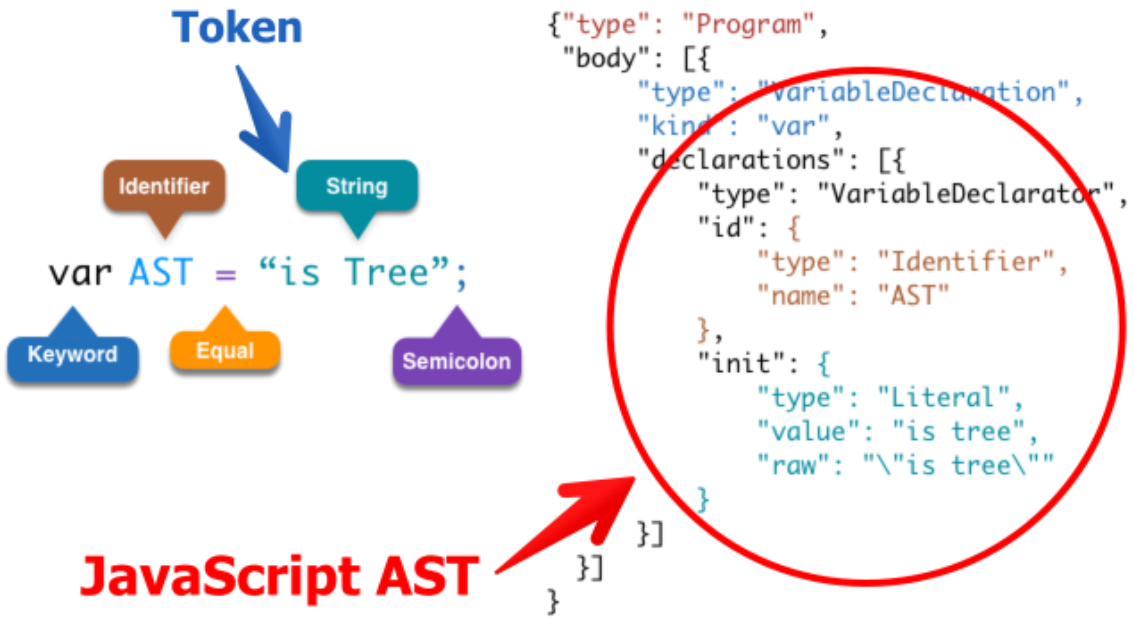


Google

@fhinkel

Abstract Syntax Tree (AST)

Abstract Syntax Tree (**AST**) is a **tree** representation of the **abstract syntactic** structure of source code written in a programming language. Each node of the **tree** denotes a construct occurring in the source code.



Why Do We Need a Angular Compiler?

The TS compiler will convert your TS code (types, decorators and classes) to ES 5/6 friendly code.

But these function constructors will have templates written in Angular language and need to be converted to JS functionality to handle all the property bindings, change detection and the directives.. etc

The Angular code **needs to be compiled** into functional JS logic (component factories when called it will generate the DOM elements)

Finally after compilation the JS code is then sent to V8 to be: parsed into AST, from AST we generate bytecode, this code will be optimized and recompiled before being sent to the CPU.

Angular Compiler

We need to compile the templates of the components to JavaScript classes. These classes will have methods that contain logic for change detection, bindings and rendering the user interface.

Angular **generates VM-friendly code at runtime or build time**. This allows the JavaScript virtual machine to perform property access caching and execute the change detection/rendering logic much faster.

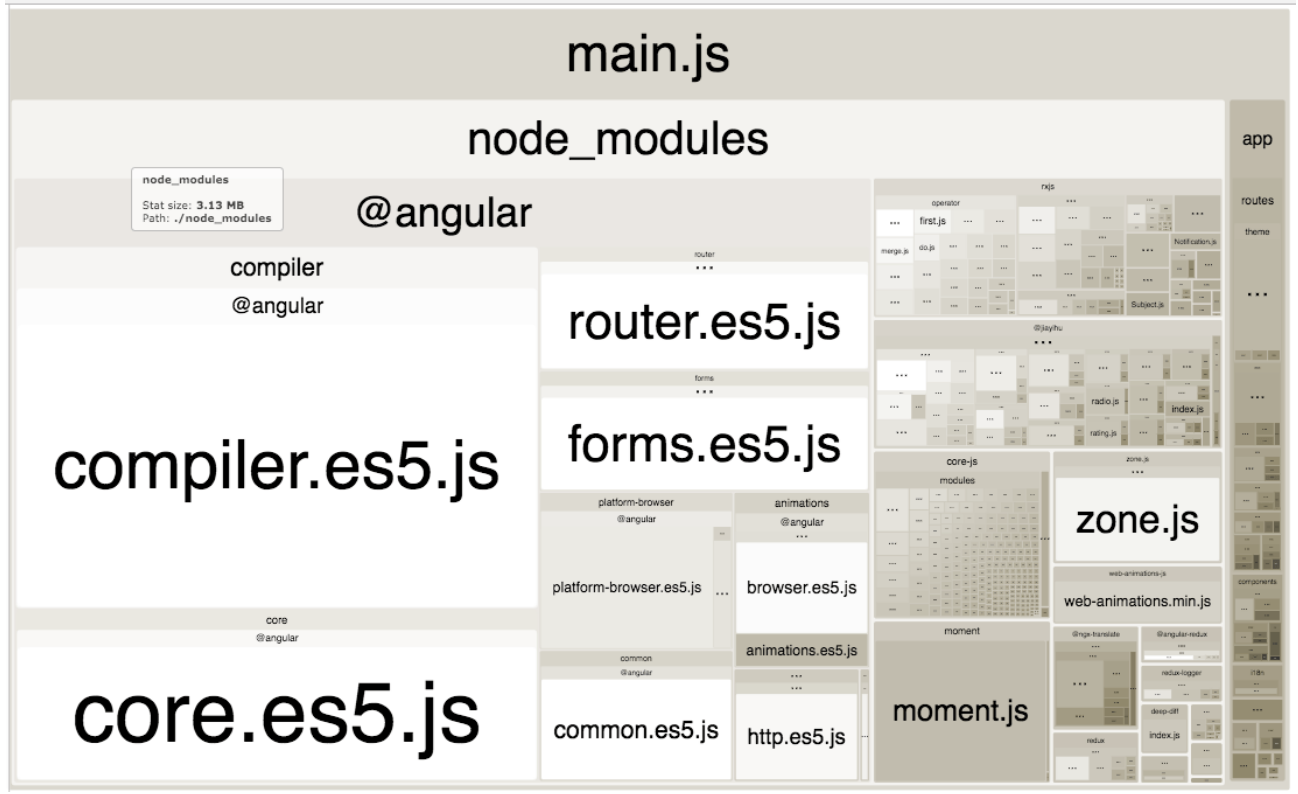
Just-in-time (JIT) compilation

It's known as **dynamic translation**, the compilation is done during execution of a program at runtime. When the user opens the browser, the following steps are performed:

- Download all the JavaScript assets (including the compiler code)
- Angular bootstraps
- Angular goes through the JiT compilation process generating all the JavaScript for each component in the application
- The application gets rendered.

Source Code Shipped to Users (JiT)

```
// npm i source-map-explorer -g
source-map-explorer bundle.js bundle.js.map
```



jiayihu.net

Ahead of Time compilation (AOT)

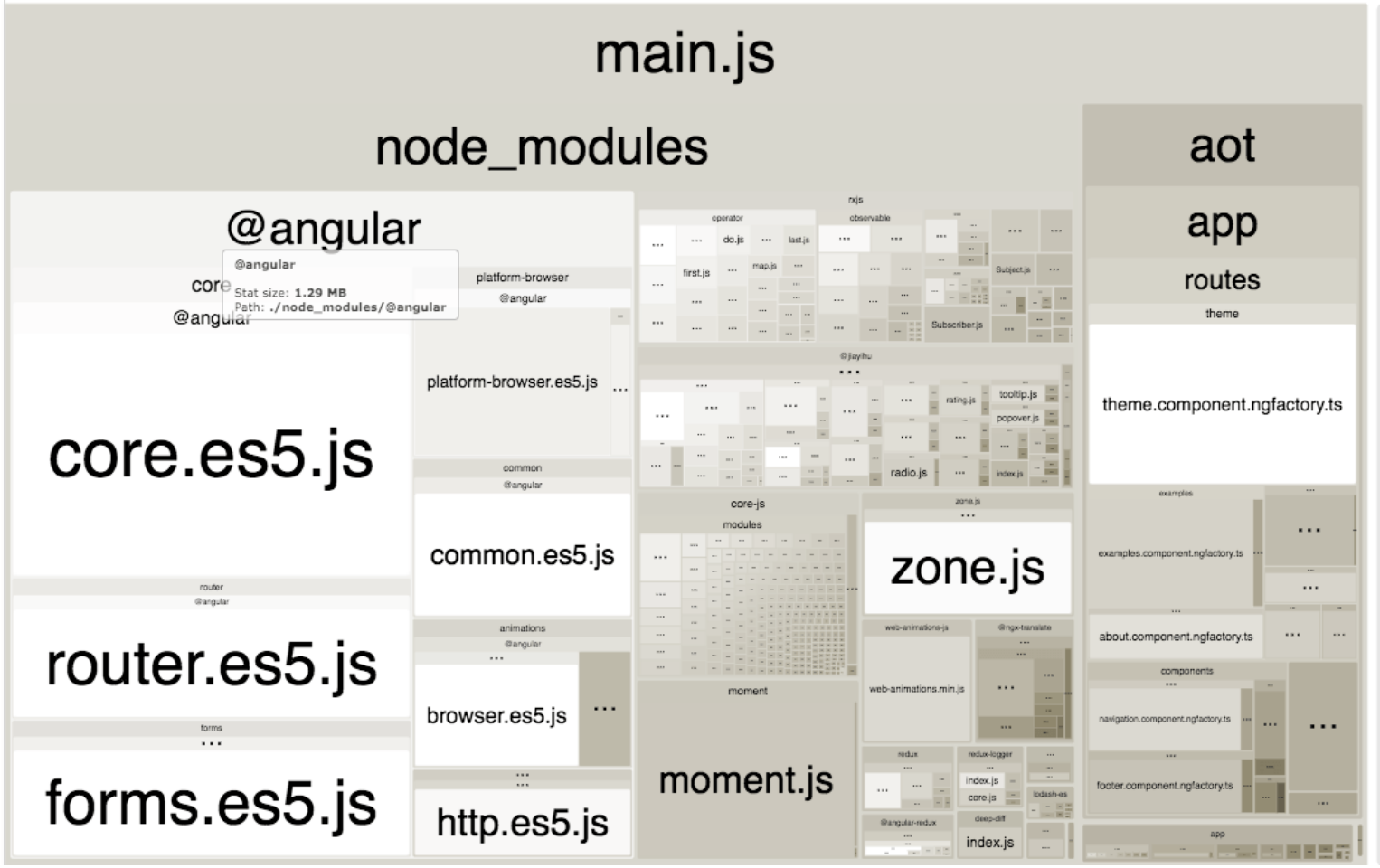
We no longer have to ship the compiler to the client

Since the compiled app doesn’t have any HTML, and instead has the generated TypeScript code, the TypeScript compiler can analyze it to produce type errors. In other words, your templates are type safe.

Bundlers can tree shake away everything that is not used in the application. The bundler will figure out which components are used, and the rest will be removed from the bundle.

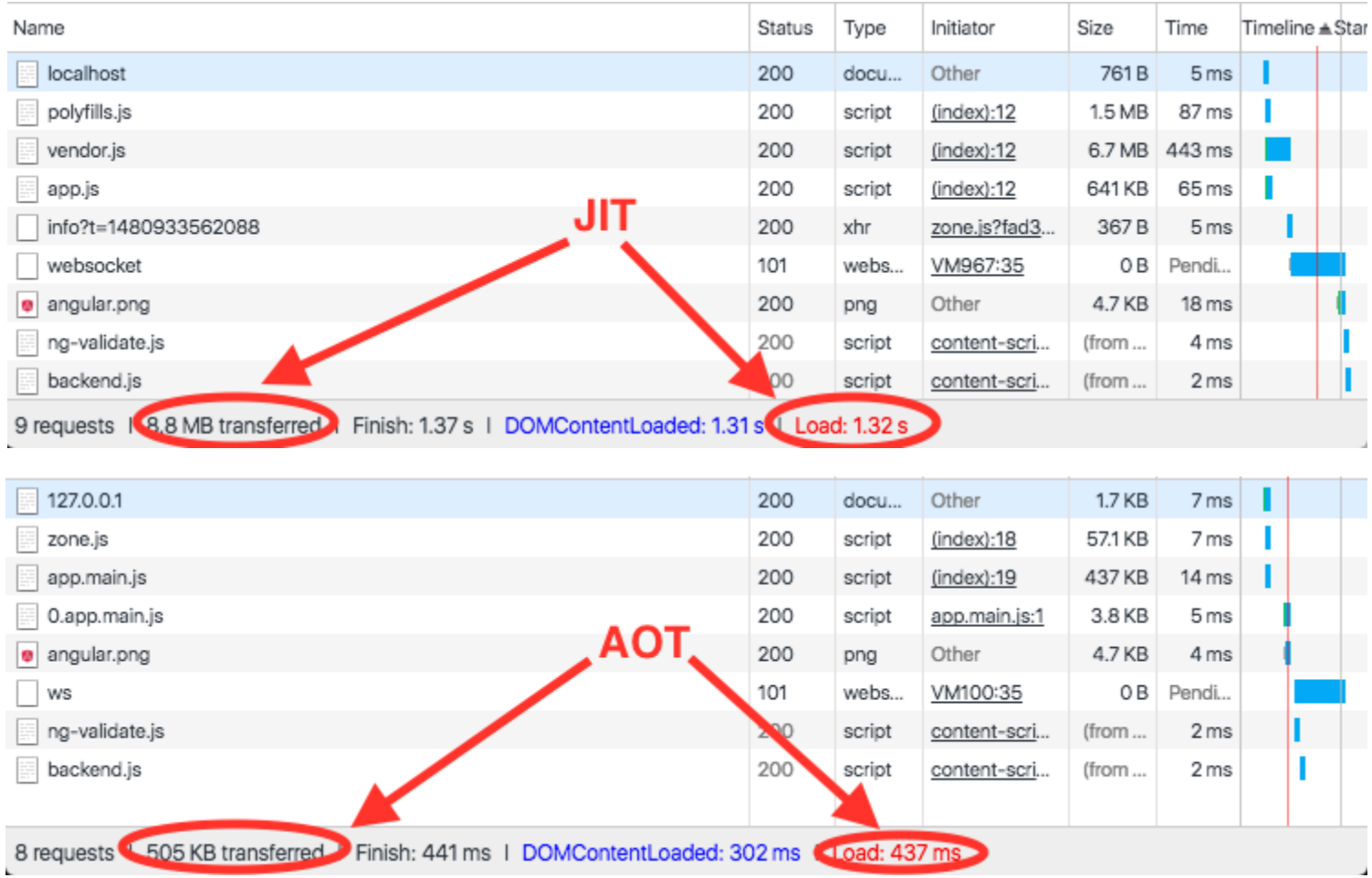
Since the most expensive step in the bootstrap of your application is compilation, compiling ahead of time can significantly improve the bootstrap time.

Source Code Shipped to Users (AoT)



1 / 25 requests | 1.1 MB / 2.8 MB transferred | Finish: 1.14 s | DOMContentLoaded: 1.05 s | Load: 1.16 s

JiT vs AoT



dzurico.com

AOT implications

To make AOT work the application has to have a clear separation of the **static and dynamic data** in the application. And the compiler has to be built in such a way that it **only depends on the static data**.

The information in the decorator is known statically. Angular knows the selector and the template of your component. It also knows that the component has an inputs and outputs.

Since Angular knows all the necessary information ahead of time, it can compile the application components without actually executing any application code, as a build step.

Server Side Rendering

Browsers have to do a heavy work before it can display the first pixel on the screen.

Flow for **server pre-rendering**:

- Generate static HTML with build tool
- Deploy generated HTML to a CDN
- Server view served up by CDN
- Server view to client view transition

https://universal.angular.io

Flow for **server re-rendering**:

- HTTP GET request sent to the server
- Server generates a page that contains rendered HTML and inline JavaScript for Preboot
- Server view to client view transition

SSR: Server View to Client View Transition

Browser receives initial payload from server, user sees **Server View.**

Preboot creates hidden div that will be used for client bootstrap and starts recording events.

Browser makes async requests for additional assets.

Once external resources loaded, Angular client bootstrapping begins.

Client view is rendered to the hidden div created by **Preboot.**

Preboot events replayed in order to adjust the application state to reflect changes made by the user before Angular bootstrapped.

Preboot switches the hidden **Client View** div for the visible **Server View** div and performs some cleanup on the visible **Client View** including setting focus.

The Big Picture

Developer writes code in TS, run **`ng build`**:

Just-In-Time:
Empty HTML + JS bundle with Angular code (Templates) + compiler

Ahead-Of-Time:
Empty HTML + JS bundle with pure JS classes (no templates)

Server-Side-Rendering:
HTML fully rendered component (server view) and fetch client view in background.

Main Points

The central part of Angular is its compiler.

The compilation can be done just in time (at runtime) and ahead of time (as a build step).

The AOT compilation creates smaller bundles, tree shakes dead code, makes your templates type-safe, and improves the bootstrap time of your application.

The AOT compilation requires certain metadata to be known statically, so the compilation can happen without actually executing the code.

Deploy for Production

```
> ng build --prod // When you run the ng build command, it creates a /dist folder.
```

Adding the production flag reduce the bundle size nearly an 83% reduction:

- Removes unwanted white space by minifying files.
- Uglifies files by renaming functions and variable names.
- AoT compilation

The app will work if you're uploading it to the root public folder, such as website.com, but if it's within a sub folder such as website.com/subfolder, you can specify the **`--base-href`** flag during the build process based on the folder structure of where the app will be placed.

If you want to use /app as an application base for router and /public as base for your assets:

```
> ng build --prod --base-href /app --deploy-url /public/
```

Differential loading

Provides two groups of bundles: one is based on ES5 and addresses older browsers, the other is based on a ES6 version (ECMAScript 2015), and offers modern browsers with faster running code.

1. Set an upper bar for the ECMAScript versions to be supported in the **`tsconfig.json`** as follows: **`"target": "es2015"`**
2. Set a lower bar within **`browserslist`**. It is a file that identifies many browsers to be supported:

```
> 0.5%
last 2 versions
Firefox ESR
not dead
IE 9-11
```

Adding CSS and JavaScript to a Project

Adding external files directly to your index.html file is not a good practice, because files won't get bundled. What we ultimately want to end up with is code that we can ship all as one unit.

If you've already done a build, you should have a folder named "**`dist`**" with all the files you need to run your application. What we want to end up with is ALL of the files we need for our application in that directory all bundled and minified.

When CSS and HTML files get compiled into our bundles, they all end up as JavaScript.

Adding Bootstrap to a Project

```
npm install bootstrap --save-dev
```

Update **angular.json** file to tell the CLI to these files installed:

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

Alternative Way

An alternate way is to place **`@import`** statements in your application level CSS file (**`styles.css`** in an Angular CLI project). This works best if you are referencing external files (URLs) or files that you can access from your resources directory.

```
@import "~bootstrap/dist/css/bootstrap.min.css";

OR

@import url('https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css');
```

Referencing Global Objects in Angular

Referencing global browser objects like document or window directly from within your code is possible, but **not encouraged and considered bad practice**. Especially Angular isn't only designed to run within your browser, but also on mobiles, the server or web workers where objects like window may not be available. (also AoT and SSR won't be possible)

Therefore the suggested approach is to wrap such objects and inject them through the dependency injection mechanism. This way it is possible to change the concrete runtime instance of a given object based on the environment the Angular application is running.

Modern Web Applications State Management

Introduction

For many Angular projects we can manage state in a fairly direct way: We tend to grab data from services and render them in components, passing values down the component tree along the way.

Managing our apps in this way works fine for smaller apps, but as our apps grow, having multiple components manage different parts of the state becomes cumbersome.

The need for state management

As the requirements for JavaScript single-page applications have become increasingly complicated, our code must manage more state than ever before. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.

Managing this ever-changing state is hard. If a model updates another model, then a view updates a model, which updates another model, and this might cause another view to update. At some point, you no longer understand what happens in your app as you have lost control over the when, why, and how of its state.

Problems!

Passing all of our values down our component tree suffers from the following downsides:

- Intermediate property passing
- Inflexible refactoring (coupling between parent and child components)
- State tree and DOM tree don’t match as data is passing through the tree
- State throughout our app is not possible

Application State

We have to differentiate between three kind of states, and find solutions for syncing between them:

- Server state (Restful - no state)
- Persisted state (DB)
- App state (Angular app)
- UI state (DOM)

What's Redux?

Redux is a tiny library used to handle Application state and bind it to the User Interface in a very effective way.

Following in the steps of **Flux**, Redux attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the three principles of Redux:

1. Single source of truth
2. State is read-only
3. Changes are made using pure functions (Reducers)

Three Principles

Single source of truth

The state of your whole application is stored in an object tree within a single store.

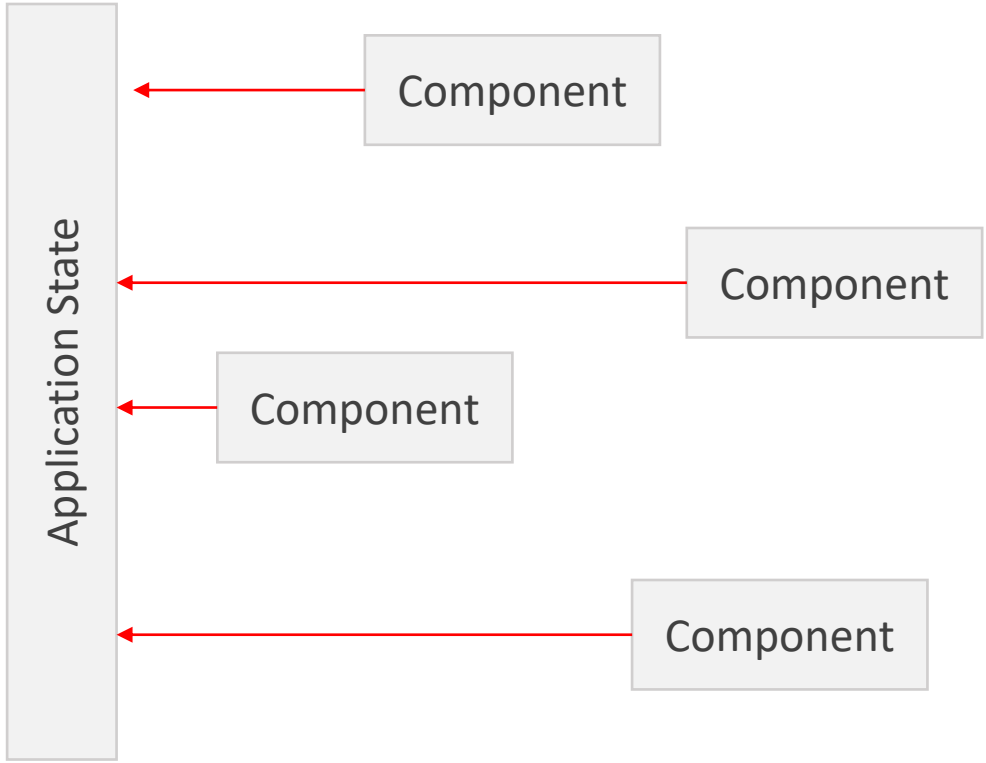
State is read-only

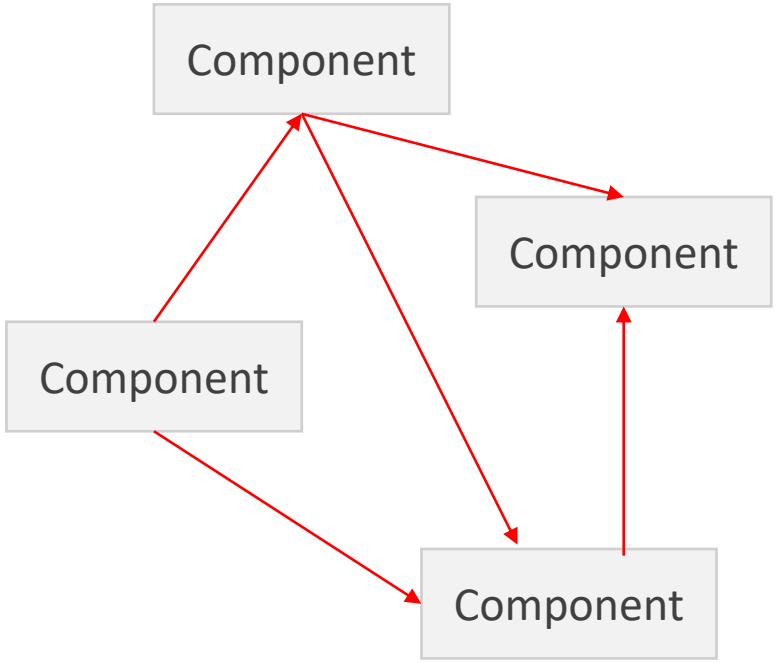
The only way to change the state is to emit an action, an object describing what happened.

Changes are made using pure functions (Reducers)

To specify how the state tree is transformed by actions, you write pure reducers.

Single Source of Truth





Why Avoiding Mutations?

Working with immutable data structures can have a positive impact on performance:

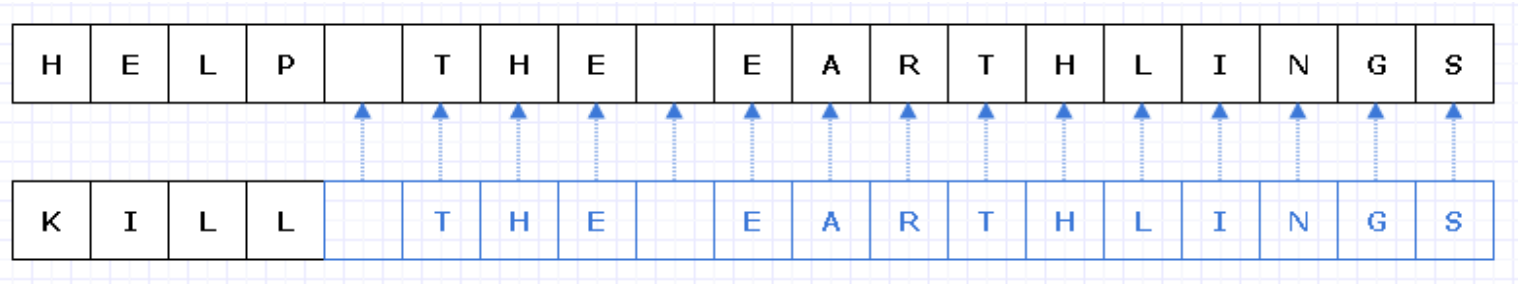
We can avoid unnecessary re-rendering of the app if the data did not change. To achieve that, we need to compare the next state of your app with the current state.

With immutable objects you can compare states by performing a shallow equality checking (immutableObject1 === immutableObject2) . In plain old JavaScript objects, you would need to deep compare in order to see if any property inside the objects changed.

Redux represents your application state as frozen object snapshots so you can save your state, or reverse state, and generally have more accounting for all state changes. (filter an array and reverse to the unfiltered array)

Immutable Data Structure

In general, most of a data structure is not copied, but *shared*, and only the changed portions are copied. It is called a **persistent data structure**. Most implementations are able to take advantage of persistent data structures most of the time. The performance is close enough to mutable data structures that functional programmers generally consider it negligible.



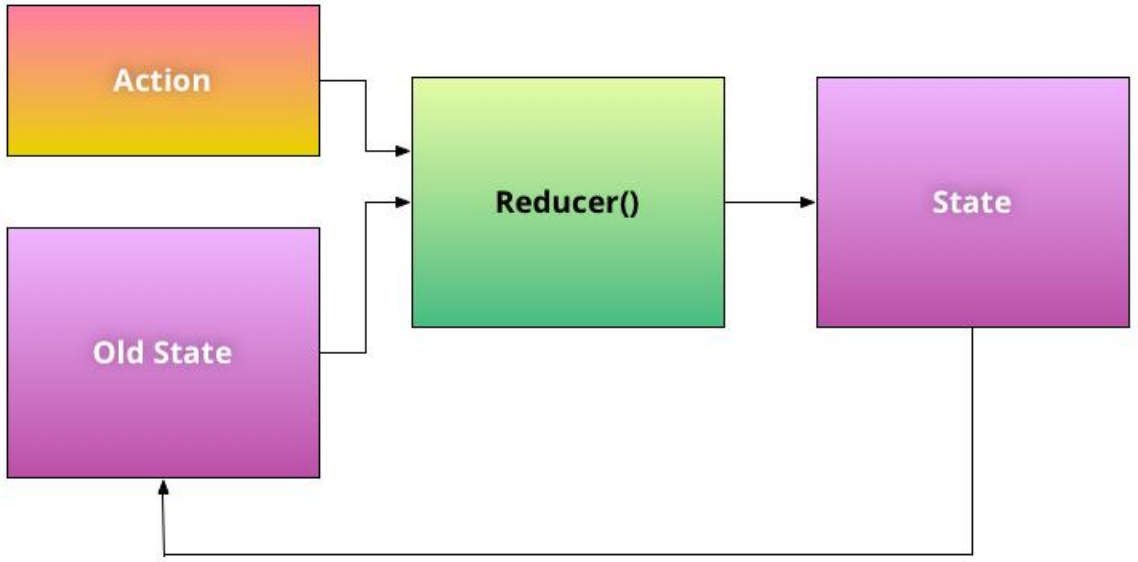
Key Ideas

All of your application's data is in a single data structure called the **state** which is held in the store

This store is never mutated directly

User interaction fires **actions** which describe what happened

A new state is created by combining the old state and the action by a function called the **reducer**.



Actions (Events)

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({action}).`**

Actions are plain JavaScript objects describing a change and using a **`type`** property as identifier:

```
{
  type: 'UPDATE_USER',
  payload: {id: 5}
}
```

Reducers

Actions describe the fact that something needs to be happened, but don't specify how the application's state changes in response. This is the job of reducers.

Reducers specify how the state changes in response to Actions. All Reducers must be pure functions:

- They produce the same output given the same input
- They don’t mutate state

Application Store

Actions represent the fact about "what to be happened" and the reducers update the state according to those actions. The Store is the object that brings them together and holds the application state.

The Application Store is central to Redux and offers an API to:

- **Dispatch actions**: `appStore.dispatch(action)`
- **Register listeners** for change notification: `appStore.subscribe(callback)`
- **Read the Application State**: `appStore.getState()`

There should only be a single store in your app

Creating a Store

`createStore(reducer, [initialState], [enhancer])`

reducer (Function): A reducing function that returns the next state tree, given the current state tree and an action to handle

[**initialState**] (any)

[**enhancer**] (Function)

Returns (**Store**): An object that holds the complete state of your app. The only way to change its state is by dispatching actions. You may also subscribe to the changes to its state to update the UI.

Example

```
const { createStore } = require('redux');
const defaultState =
        { topics: [
                { name: 'Learning Node', topic: 'Node', },
                { name: 'Learning MongoDB', topic: 'MongoDB', },
                { name: 'Learning Angular', topic: 'Angular', }
        ]};
function reducer(state, action) {
        switch(action.type) {
                case 'ADD_TOPIC':
                        return Object.assign({}, state,
                                        { topics: [...state.topics, action.payload] });
                default: return state; }
}
const store = createStore(reducer, defaultState);
function renderView(f) {
        f(defaultState);
        store.subscribe(() => { f(store.getState()); });
}
renderView((state) => {console.log(`There are ${state.topics.length} topics in the course`)});
store.dispatch({
        type: 'ADD_TOPIC',
        payload: { name: 'Learning Redux', topic: 'Redux', }
});
```

Never Mutate!

If your state is a plain object, make sure you never mutate it.

In reducer, instead of returning something like this:

```
Object.assign(state, newData)
```

use: `Object.assign({}, state, newData)`

This way you don't override the previous state.

Another way to return a new object is:

```
return { ...state, ...newData }
```

Spread Operator

Triple dots also called as spread operator, it is used as syntatic sugar for **`Object.assign()`**. What it simple does is take properties of source object (first parameter) and updates its properties based on subsequent parameters, later properties overriding earlier ones and returns new target object (to avoid mutation).

Middleware

Redux middleware solves different problems than Express middleware, but in a conceptually similar way. It provides a third-party extension point **between dispatching an action, and the moment it reaches the reducer.** People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more.

Action -> Middleware(s) -> Reducer

Middleware Example

```
const { createStore, applyMiddleware } = require('redux');

const defaultState = { ... };

function reducer(state, action) { ... }

const logger = store => next => action => {
        console.log('dispatching ', action);
        let result = next(action);
        console.log('state after action ', store.getState());
        return result;
}

const store = createStore(reducer, defaultState, applyMiddleware(logger));
```

`combineReducers(reducers)`

As your app grows more complex, you'll want to split your reducing function into separate functions, each managing independent parts of the state.

The `combineReducers` helper method turns an object whose values are different reducing functions into a single reducing function you can pass to `createStore`.

The resulting reducer calls every child reducer, and gathers their results into a single state object. The shape of the state object matches the keys of the passed reducers.

As a result, the state object will look like this:

```
{
    reducer1: ...
    reducer2: ...
}
```

CombineReducers Example

The **`concat()`** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
export default function todos(state = [], action) {
        switch (action.type) {
                case 'ADD_TODO': return state.concat([ action.payload ])
                default: return state }
}
```

reducers/todos.js

```
export default function counter(state = 0, action) {
        switch (action.type) {
                case 'INCREMENT': return state + 1
                case 'DECREMENT': return state - 1
                default: return state }
}
```

reducers/counter.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import counter from './counter'
export default combineReducers({ todos, counter })
```

reducers/index.js

CombineReducers Example

```
import { createStore } from 'redux'
import reducer from './reducers/index'

let store = createStore(reducer)

console.log(store.getState())
{ counter: 0, todos: [] }

store.dispatch({ type: 'ADD_TODO', payload: 'Finish Redux Homework' })

console.log(store.getState())
{ counter: 0, todos: [ 'Finish Redux Homework' ] }
```

App.js

Integration with Angular

There are several attempts to use Redux or create a Redux-inspired system that works with Angular.

ngrx is a Redux-inspired architecture that is heavily observables-based.

Integrating Redux with Angular

1. Create **AppState**
2. Create **Actions**
3. Create **Reducers**
4. Create **Store**
5. Use Redux Store in your **component**

Create AppState

```
export interface ITopic {
        id: number;
        name: string;
}
```

store/topic.ts

```
import { ITopic } from './topic'

export interface IAppState{
        data : ITopic[];
}
```

store/state.ts

Create Actions

```
export const ADD_TOPIC = 'ADD_TOPIC';

export function addTopicAction(topicText: String){
        return {
                type: ADD_TOPIC,
                payload: topicText
        }
}
```

store/actions.ts

Create Reducers

```
import { IAppState } from './state'
import { ADD_TOPIC } from './actions'
const initialState: IAppState = { data: [
        { id: 0, name: 'Node JS', },
        { id: 1, name: 'MongoDB', },
        { id: 2, name: 'TypeScript', } ] }
let nextId = 3;

function addTopic(state, action): IAppState{
        return Object.assign({}, state,
        { data: [...state.data, { id: nextId++, name: action.payload }] })
}

export function reducer(state:IAppState = initialState, action){
        switch(action.type){
                case ADD_TOPIC: return addTopic(state, action);
                default: return state; }
}
```

store/reducer.ts

Create Store

```
import { createStore } from 'redux'
import { reducer } from './reducer'

export const store = createStore(reducer);
```

store/store.ts

Use Redux in Component

```
import { Component, ViewChild } from '@angular/core';
import { store, addTopicAction } from './redux-store';

@Component({ selector: 'app-root', template: `
        <input #topicText>
        <button (click)="addTopic(#topicText.value)">Add</button>
        <ul> <li *ngFor="let topic of topics">{{topic.id}} - {{topic.name}}</li> </ul> ` })
export class AppComponent {
        topics; subscription;

        ngOnInit(){
            this.topics = store.getState().data;
            this.subscription = store.subscribe(()=>{this.topics = store.getState().data; })
        }
        ngOnDestroy(){ this.subscription.unsubscribe(); }
        addTopic(value){
                store.dispatch(addTopicAction(value));
        }
}
```