CLEAN Architecture Workshop :: Angular, NestJS, and Nx

Matt Vaughn

CLEAN Architecture Workshop :: Angular, NestJS, and Nx

Matt Vaughn

This book is for sale at
http://leanpub.com/clean-architecture-workshop-with-angular-nestjs-and-nx

This version was published on 2021-12-05

Also By Matt Vaughn

Angular Architecture Patterns

Effective Angular Code Reuse Strategies and Techniques

Angular Libraries - The Complete Guide

Contents

Architecture

Architecture is not a single discipline, nor is it only a structural artifact for the team. It involves all facets of design, planning, execution, and maintenance of software solutions. Therefore, it is a continual consideration for all project endeavors.

Facet	Description
Design and Planning	Any endeavor that includes development and ends with deployment requires proper design and planning appropriate to the context and significance of the item. This is mostly a thinking exercise that gathers enough information to know and understand the specific goal and objective before deciding on `how?` to do something. Design and planning should be deliberate with the objective of understanding and sharing through collaboration. The result of design and planning should be specific and within context - it needs to have an actionable outcome.
Tools and Materials	The selection of tools and materials is essential so that they align with the goals and design of the feature and/or system. Consider the tool/material capabilities and how they match to the capabilities of the actual developers. It is essential that each developer understand and to know how to use the tools effectively to support the design and plan.
Execution	The execution of architecture is the deliberate use of the *selected* tools and materials to deliver a solution that corresponds to the specifications of the design and plan. It requires that all developers and team members understand the overall goal and objective of the solution, the scheme[1] or strategy, developer principles, and their tools/materials at a high-level.

Principles

It is much easier to learn and understand a few principles than memorize a large set of rigid rules that may or not be within context of the specific situation. There are many developer, software, and architectural principles to learn and follow. However, if you could only bring a few to a deserted island - Separation of Concerns and Single Responsibility have to be on the list.

[1] scheme.md#scheme

Separation of Concerns (SoC)

Most enterprise applications contain a set of concerns - and most of them happen in a well-defined sequence. Our application concerns will be implemented using a layered-architectural approach using the following layers. Each layer has a specific concern and set of responsibilities. Each layer provides access endpoints to consumers of the layer - this will primarily be Angular *services* that are injectable and participate in the Dependency Injection mechanism of Angular.

Single Responsibility

The *thing* under consideration should do one thing and do it well. May include:

- encapsulation
- open-closed principle

Inversion of Control

To improve application testability and manage dependencies effectively, Most if not all dependencies will be provided to a target using dependency injection. There are few or a couple of use-cases that allow for an object to initialize or create a new object for their use. Here are the foreseeable exceptions.

- initializing new Observables and Subject for streaming data
- initializing new data models
- initializing new business actions

Angular Applications

We will use the Angular CLI and Angular Workspace to generate and manage new application projects. All application members should be contained within well-defined modules. The responsibility of the *AppModule* is to bootstrap the application and coordinate the loading of all required framework, 3rd-party, core, shared, and site modules.

Application configuration will make use of the our configuration cross-cutting concern library and the specific *environment* constants defined for the application.

Angular Libraries

During the analysis and design of new features we should give consideration to how the target feature will be used. If the feature is a candidate for code sharing and reuse it should be implemented as a library project within the Angular workspace. The following list is a sample of the different types of library projects that fit well within the Angular Workspace and Architecture capabilities.

Library	Description	Types
cross-cutting concerns	A *cross-cutting concern* library provides a feature that is not specific to a domain - but cross several boundaries.	Configuration, Logging, Exception Handling, HTTP Service
feature	A *feature* library may implement a domain-feature. However, the feature is shared and reused by multiple applications.	Shopping Cart, Login, Password Management, Password Reset, etc.
components	A *component* library provides a set of components that are not specific to a single domain or application. They may provide structural elements or they could be generic controls.	Generic/structural components like: panels, cards, banners, date pickers, text editors, pills, etc.
foundational	A *foundational* library will provide base classes and/or interfaces for specific Angular elements like Components and Services. The allow for using the Object Oriented capabilities of Typescript for inheritance, polymorphism, and encapsulation. They usually have common implementations that are shared by all instances of each class type.	Base classes for Components, Services, Business Actions and Business Providers

Library	Description	Types
core domain service	A *core domain service* library implements the business logic and data access for a specific application domain. The entry-point is typically a service that the application will use within the application project. It provides a way to encapsulate the business logic and to create a distinct layer separate from the UI/Presentational layer. This library is a good candidate for code reuse when you have multiple applications and/or other domain service libraries to support a specific domain.	A single core domain service could be used by a consumer facing application, internal administrative application, and also a B2B application at the same time - a single source of truth for the implementation of the business logic.
framework	A *framework* library provides a mechanism and typically an API to implement customized solutions based on a single structure. Occasionally, these libraries may include some default implementations that will be common to most use cases.	The *business action* provides and algorithm to implement customized business logic in a consistent and testable manner. It takes advantage of a *rule engine* - which also another framework library to create simple or complex business rules.

Library	Description	Types
micro-application	A *micro-application* library is a full or more comprehensive implementation of feature(s). It will contain all of the UI/Presentational elements as well as the business logic implement. The entire library encapsulates an entire application part. The application is consumed by Angular application projects by lazy-loading the module. The micro-application is an *NgModule* that has all of its components and routes defined.	Security, Shopping Cart, Admin Utilities

Architecture Layers

Layer Definitions

The following is a list of application layers (top-to-bottom) that represent the logical separation of concerns for each layer and their specific responsibilities.

- User interface and presentational concerns
- what to show the user
- how to display information
- how to create, edit, or remove data/information
- security: authentication and authorization

- UI Service

- providing data to components
- retrieve and persisting data using core domain services
- managing state for UI/presentation concerns
- abstracting the business logic from the presentational concerns
- acts as a mediator between the UI and Core Domain Service

- Core Domain Services
  - provides an entry point or API to handle domain-business logic
  - has not UI or presentational concerns
  - encapsulates all business logic for the specified domain item
- Business Logic Layer
  - encapsulated within a Core Domain Service
  - provides a business logic layer for the specific domain
  - a coordinating *BusinessProvider* service will provide the interface/API for business logic operations.
  - all business logic operations will be implemented as units-of-work as business actions.
  - business actions provide a consistent mechanism to validate input, evaluate business rules, and execute business logic
  - return objects, if any, of business actions will be *Observables*
  - all business rules will be implemented using a rule engine to provide a consistent and testable implementation of simple and complex business rules
  - business actions will have access to specific members of the business provider (e.g., HTTP services, ServiceContext, etc.)
- Data Access
  - The application will implement specific data access providers to persist and retrieve data from data stores.
  - All HTTP operations will use the HTTP service (cross-cutting concern library) to execute HTTP requests
  - All HTTP data providers will return a defined API Response object that contains the response payload and additional meta information required to handle the response - including providing messages to users and/or the application.

UI (Presentation)

The UI layer will contain site-level presentational elements that are not related to a specific domain. These will mostly likely be organized in a *SiteModule* that is shared/exported to other feature-specific modules.

- Sidebar components for navigation
- Header
- Footer
- Menu components

The UI will also contain one or more feature modules that organize all of the items related to a feature within a single module. These modules will define their own routes - most likely including a default route that could be a target from the application-level lazy-loaded route(s). Feature modules provide a code organization strategy that enables encapsulation and a good separation of concerns.

- NgModule
- routes
- components (container and presentational)
- pipes
- directives
- services
- models

Search Form (Example)

The following page/view is the home page for the new application. The home page view is a *container component* that contains a set of other *presentational* components to make up the display - like the *Keyword Search* form.

> **Container/Presentational Components**: Understanding the concept and principles are more important than a pure/strict adherence to this pattern. It is an organizational technique that is inline with a *reactive* programming flow. More information about container/presentation components:
>
> - https://indepth.dev/container-components-with-angular/[2]
> - [https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0])https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Most of the time *container components* are top-level entry point components that are the targets of the application routes. This is not a rule. Because depending on the structure of the view, there may be more than one container component responsible for coordinating the display using presentational components.

Keyword Search form

Presentational components usually get data from their parent *container* component. These components take advantage of inputs and outputs to allow communication between the container and specified presentational component.

[2] https://indepth.dev/container-components-with-angular/

Container component with outline of components

Search Form and Validation

The search form is using Angular's Reactive Forms. We are using default validators and a custom *async* validator to check for any invalid characters.

```
initializeForm() {
  this.loggingService.log(this.componentName, Severity.Information, `Preparing to in\
itialize the keyword search form.`);
  this.keywordSearchForm = this.formBuilder.group({
    keyword: new FormControl(null, {
      validators: [Validators.required, Validators.minLength(3), Validators.maxLengt\
h(20)],
      asyncValidators: [this.invalidCharacterValidator],
    }),
  });
}
```

The `invalidCharacterValidator` implementation is simplified by using a UI Service. The UI Service provides a location for component logic code that is shared by all components within the specified UI module. The UI Service can coordinate with other services to provide validation and data operations for the components. This abstraction should allow each component to have less code and be easier to test.

```
  invalidCharacterValidator: AsyncValidatorFn = (control: FormControl): Observable<V\
alidationErrors | null> => {
    return this.uiService.validateKeyword(control);
  };
```

Invalid Form and Notifications

If there are any validation errors, the services provide enough information back to the components for user notification.

BrandTrace Keyword Search form with validation errors.

UI Service (Mediator)

The *service* mentioned in the feature module above is the UI Service. In many applications, this service is overloaded with too many concerns and responsibilities. This service will be used to support UI concerns like state management and abstracting data access operations. This service will consume *core domain services* to perform data and business logic operations. It will have the responsibility to provide data to the components by way of Observables using one-way reactive streams of data.

This means that components cannot directly communicate with Subject types (i.e., BehaviorSubject, ReplaySubject) to retrieve data streams. It is not the responsibility of components to subscribe to data streams - the UI service will abstract these operations by using the core domain services.

The *UI Service* will act as a mediator for the components to the core domain service. It will not perform or initiate any HTTP/API calls directly. In fact, using the layered-architecture approach there will no HTTP/API calls originating from the UI/Presentational layer. These will be implemented and encapsulated within the core domain service libraries.

UI Service (Example)

This is a minimal example of *some* of the responsibilities that a UI Service could have.

- coordinate async form validation(s)
- coordinate data retrieval and persistence with the domain service
- manage state for data elements displayed by the components

```
import { Injectable } from '@angular/core';
import { ServiceBase } from '@tc/foundation';
import { LoggingService } from '@tc/logging';
import { KeywordValidationService } from '../services/keyword-validation/keyword-val\
idation.service';
import { FormControl } from '@angular/forms';
import { debounceTime, switchMap, map } from 'rxjs/operators';
import { of } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class UISearchService extends ServiceBase {
  constructor(private keywordValidatorService: KeywordValidationService, loggingServ\
ice: LoggingService) {
    super(loggingService, `UISearchServiceService`);
```

```
  }

  /**
  - Use to coordinate the validation of the [keyword] value for the UI.
  - @param control
   */
  validateKeyword(control: FormControl) {
    if (control.value == null) {
      return of(null);
    }
    return of(debounceTime(1200)).pipe(
      switchMap(() => {
        return this.keywordValidatorService.validateCharacters<boolean>(control.valu\
e).pipe(
          map(response => {
            return response.IsSuccess === true ? null : { invalidCharacters: true };
          })
        );
      })
    );
  }
}
```

Motivation for UI Services

The implementation of the async validator returns the result a helper service (*KeywordValidatorService*). The UI Service provides more flexibility for the implementation details for the components. A component's main responsibility is to collect and display information within the designated view. Therefore, removing the *validation* logic outside of the component removes the responsibility to a more testable location and also reduces the responsibilities of the component.

```
validateKeyword(control: FormControl) {
    if (control.value == null) {
      return of(null);
    }
    return of(debounceTime(1200)).pipe(
      switchMap(() => {
        return this.keywordValidatorService.validateCharacters<boolean>(control.valu\
e).pipe(
          map(response => {
            return response.IsSuccess === true ? null : { invalidCharacters: true };
          })
        );
```

```
      })
    );
  }
```

The testing of validation rules in a service is a smaller foot-print than performing all of the tests from component configurations.

```
yarn test --test-file=keyword-validation.service.spec.ts --project=brand-trace --wat\
ch=true
yarn run v1.15.2
$ ng test --test-file=keyword-validation.service.spec.ts --project=brand-trace --wat\
ch=true
 PASS  apps/brand-trace/src/app/services/keyword-validation/keyword-validation.servi\
ce.spec.ts
  KeywordValidationService
    √ should be created (21ms)
    √ should be valid with alpha-only value [brand] (10ms)
    √ should be valid with alpha-numeric value [brand1234] (4ms)
    √ should be valid with alpha-numeric value with space [brand 1234] (5ms)
    √ should be valid with numeric-only value [1234567890] (4ms)
    √ should be valid with alpha-numeric value with hyphen [brand-1234] (5ms)
    √ should be valid with alpha-only value with hyphen [brand-trace] (5ms)
    √ should be valid with numeric-only value with hyphen [1234-5678] (4ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        3.999s
Ran all test suites matching /keyword-validation.service.spec.ts/i.
```

The test setup and configuration is straight-forward and doesn't have any unnecessary *TestBed* configuration for:

- providers
- imports
- exports
- declarations

```
import { TestBed } from '@angular/core/testing';
import { ApiResponse } from '@tc/common';
import { KeywordValidationService } from './keyword-validation.service';
import { LoggingService } from '@tc/logging';

describe('KeywordValidationService', () => {
  beforeEach(() =>
    TestBed.configureTestingModule({
      providers: [LoggingService],
    })
  );

  it('should be created', () => {
    const service: KeywordValidationService = TestBed.get(KeywordValidationService);
    expect(service).toBeTruthy();
  });

  it('should be valid with alpha-only value [brand]', () => {
    const service: KeywordValidationService = TestBed.get(KeywordValidationService);

    const response = service.validateCharacters<boolean>('brand');
    let apiResponse: ApiResponse<boolean>;
    response.subscribe(
      result => {
        apiResponse = result;
      },
      error => console.log(error)
    );

    expect(apiResponse.IsSuccess).toEqual(true);
    expect(service.validateCharacters('')).toBeTruthy();
  });
});
```

Core Domain Services

The core domain services are libraries that expose a service API for a specific domain item. A service within this library provides an accessible API to allow operations on the specified domain. There is a complete encapsulation of the business logic within this library. This library does not contain any items or operations related to UI or presentation concerns. It is a complete separation of business logic from the UI layer.

The core domain service will be provided (dependency injected) into any feature modules contained in the application project. Typically, the scope of this service will be at the feature module level. However, there could be use cases where the domain service is globally available from the application to any other feature module that may need to use its operations.

Domain Service Example (Search)

The CLI command will create a new `search` library project in the `libs/brandtrace` folder. This project will contain the business logic for the *Search* feature of the BrandTrace application.

```
ng generate library brandtrace/search --publishable --dry-run
? In which directory should the library be generated?
CREATE libs/brandtrace/search/ng-package.json (167 bytes)
CREATE libs/brandtrace/search/package.json (154 bytes)
CREATE libs/brandtrace/search/README.md (158 bytes)
CREATE libs/brandtrace/search/tsconfig.lib.json (519 bytes)
CREATE libs/brandtrace/search/tslint.json (188 bytes)
CREATE libs/brandtrace/search/src/index.ts (48 bytes)
CREATE libs/brandtrace/search/src/lib/brandtrace-search.module.ts (172 bytes)
CREATE libs/brandtrace/search/src/lib/brandtrace-search.module.spec.ts (400 bytes)
CREATE libs/brandtrace/search/tsconfig.json (126 bytes)
CREATE libs/brandtrace/search/tsconfig.spec.json (236 bytes)
CREATE libs/brandtrace/search/jest.config.js (278 bytes)
CREATE libs/brandtrace/search/src/test-setup.ts (30 bytes)
UPDATE package.json (8235 bytes)
UPDATE angular.json (41393 bytes)
UPDATE nx.json (1468 bytes)
UPDATE tsconfig.json (1696 bytes)
```

Search Service

This service is the interface or public API for the domain library. The responsibility of this service is to coordinate the request and response for the specified requests.

> Note: This service is the *facade* of the Facade Pattern for this domain service. It encapsulates the actual implementation details of the business logic. There should be no *business logic* details or implementation within this service.

```
import { Injectable, Inject } from '@angular/core';
import { ServiceBase } from '@tc/foundation';
import { LoggingService } from '@tc/logging';
import { BusinessProviderService } from './business/business-provider.service';
import { Observable, of } from 'rxjs';
import { SearchResult } from '@tc/brandtrace/common';
import { ApiResponse } from '@tc/common';

@Injectable({
  providedIn: 'root',
})
export class SearchService extends ServiceBase {
  constructor(@Inject(BusinessProviderService) private businessProvider: BusinessPro\
viderService, loggingService: LoggingService) {
    super(loggingService, 'SearchService');
    this.initializeBusinessProvider();
  }

  /**
  - Use to initialize the business provider. Required to use
  - the same [ServiceContext] throughout the stack.
   */
  private initializeBusinessProvider() {
    this.businessProvider.serviceContext = this.serviceContext;
  }

  search<T extends SearchResult[]>(keyword: string): Observable<ApiResponse<T>> {
    return this.businessProvider.search<T>(keyword);
  }
}
```

Add a service that will act as an entry point to the domain service.

```
ng g service search --flat=false --project=brandtrace-search
CREATE libs/brandtrace/search/src/lib/search/search.service.spec.ts (333 bytes)
CREATE libs/brandtrace/search/src/lib/search/search.service.ts (135 bytes)
```

Domain Common Library

Create a library project for the specific application's *common* domain information.

- models and/or entities

- shared classes and enums
- shared rules
- regular expression constants

```
ng g library brandtrace/common --publishable
? In which directory should the library be generated?
CREATE libs/brandtrace/common/ng-package.json (167 bytes)
CREATE libs/brandtrace/common/package.json (154 bytes)
CREATE libs/brandtrace/common/README.md (158 bytes)
CREATE libs/brandtrace/common/tsconfig.lib.json (519 bytes)
CREATE libs/brandtrace/common/tslint.json (188 bytes)
CREATE libs/brandtrace/common/src/index.ts (48 bytes)
CREATE libs/brandtrace/common/src/lib/brandtrace-common.module.ts (172 bytes)
CREATE libs/brandtrace/common/src/lib/brandtrace-common.module.spec.ts (400 bytes)
CREATE libs/brandtrace/common/tsconfig.json (126 bytes)
CREATE libs/brandtrace/common/tsconfig.spec.json (236 bytes)
CREATE libs/brandtrace/common/jest.config.js (278 bytes)
CREATE libs/brandtrace/common/src/test-setup.ts (30 bytes)
UPDATE package.json (8235 bytes)
UPDATE angular.json (42590 bytes)
UPDATE nx.json (1519 bytes)
UPDATE tsconfig.json (1768 bytes)
```

API Response Model

The following command creates a shared API response model for the *keyword search* request.

```
ng g class search/searchResult --skipTests --project=brandtrace-common
CREATE libs/brandtrace/common/src/lib/search/search-result.ts (30 bytes)
```

Business Logic

Create a new service within the domain library that will coordinate business logic for the domain service.

```
ng g service business/businessProvider --project=brandtrace-search
CREATE libs/brandtrace/search/src/lib/business/business-provider.service.spec.ts (38\
4 bytes)
CREATE libs/brandtrace/search/src/lib/business/business-provider.service.ts (145 byt\
es)
```

The business provider will initialize and execute business actions to process the request.

```
import { Injectable, Inject } from '@angular/core';
import { BusinessProviderBase, ServiceBase } from '@tc/foundation';
import { LoggingService } from '@tc/logging';
import { HttpSearchRepositoryService } from './http-search-repository.service';
import { Observable, of } from 'rxjs';
import { SearchAction } from './actions/search.action';
import { SearchResult } from '@tc/brandtrace/common';
import { ApiResponse } from '@tc/common';

@Injectable({
  providedIn: 'root',
})
export class BusinessProviderService extends BusinessProviderBase implements IBusine\
ssProviderService {
  constructor(@Inject(HttpSearchRepositoryService) public apiService: HttpSearchRepo\
sitoryService, loggingService: LoggingService) {
    super(loggingService);
  }

  search<T extends SearchResult[]>(keyword: string): Observable<ApiResponse<T>> {
    const action = new SearchAction<T>(keyword);
    action.Do(this);
    return action.response;
  }
}

export class BusinessProviderServiceMock extends ServiceBase implements IBusinessPro\
viderService {
  constructor(@Inject(HttpSearchRepositoryService) public apiService: HttpSearchRepo\
sitoryService, loggingService: LoggingService) {
    super(loggingService, 'BusinessProviderServiceMock');
  }

  search<T extends SearchResult[]>(keyword: string): Observable<ApiResponse<any>> {
    const data = <T>new Array<SearchResult>();
```

```
36
37      const apiResponse = new ApiResponse<T>();
38      apiResponse.IsSuccess = true;
39      apiResponse.Message = 'Successfully processed keyword search.';
40      apiResponse.Data = data;
41
42      return of(apiResponse);
43    }
44  }
45
46  export interface IBusinessProviderService {
47    search<T extends SearchResult[]>(keyword: string): Observable<ApiResponse<T>>;
48  }
```

Business Actions

Business actions provide the *units of work* that execute and process business logic for the application.

```
ng generate class business/actions/businessActionBase --skipTests --project=brandtra\
ce-search
CREATE libs/brandtrace/search/src/lib/business/actions/business-action-base.ts (36 b\
ytes)
```

Business actions use a base class to provide *context* to each of the actions. Basically, we are providing the *BusinessProvider* and *LoggingService* along with a shared ServiceContext that aggregates information from any other actions.

```
1  import { LoggingService } from '@tc/logging';
2  import { ActionBase } from '@tc/foundation';
3  import { Observable } from 'rxjs';
4  import { BusinessProviderService } from './../business-provider.service';
5  import { ApiResponse } from '@tc/common';
6
7  /**
8   * A helper class to provide the action with required dependencies and
9   * starting the execution of the action life-cycle pipeline.
10  */
11 export abstract class BusinessActionBase<T> extends ActionBase {
12   showRuleMessages = true;
13   hideRuleMessages = false;
14
15   businessProvider: BusinessProviderService;
```

```
  loggingService: LoggingService;
  actionName: string;
  public response: Observable<ApiResponse<T>>;

  constructor(actionName: string) {
    super();
    this.actionName = actionName;
  }

  /**
  - Use the [Do] method to perform the action. Also uses [inversion of control]
  - and provides the action the same instance of the [service context] and
  - [logging service].
   */
  Do(businessProvider: BusinessProviderService) {
    this.businessProvider = businessProvider;
    this.serviceContext = businessProvider.serviceContext;
    this.loggingService = businessProvider.loggingService;

    this.execute();
  }
}
```

Search Action

Use the CLI to create a new action in the domain library.

```
ng g class business/actions/search.action --skipTests=false --project=brandtrace-sea\
rch
CREATE libs/brandtrace/search/src/lib/business/actions/search.action.spec.ts (182 by\
tes)
CREATE libs/brandtrace/search/src/lib/business/actions/search.action.ts (31 bytes)
```

The implementation includes setting up validation rules and executing the business logic in the `performAction()` method.

```
import { BusinessActionBase } from './business-action-base';
import { Severity } from '@tc/logging';
import { SearchResult } from '@tc/brandtrace/common';
import { KeywordIsValidRule } from '@tc/brandtrace/common';

/**
 * Action implements business logic of performing a keyword search.
 */
export class SearchAction<T extends SearchResult[]> extends BusinessActionBase<T> {
  constructor(private keyword: string) {
    super('SearchAction');
  }

  preValidateAction() {
    this.validationContext.addRule(
      new KeywordIsValidRule('KeywordIsValid', 'The keyword is not valid.', this.key\
word, this.showRuleMessages)
    );
  }

  performAction() {
    this.loggingService.log(this.actionName, Severity.Information, `Preparing to exe\
cute a keyword search for [${this.keyword}].`);
    this.response = this.businessProvider.apiService.search<T>(this.keyword);
  }
}
```

Business Rules

The rule-engine library project contains the framework and helper classes to execute validation and business rules. There is a pre-defined set of rules read to use. You can compose your validation logic using any of the rules.

Validation of a Single String Value (Example)

The following example shows the implementation of an async validator using the *rule-engine* library. The validation of the a string value for the following cases:

- value is not null or undefined
- value length is within 3 and 20 characters
- string matches the specified regular expression

Use the *ValidationContext* as a container for the specified rules to evaluate. The `addRule()` method allows you to add any rule that *extends* a `RulePolicy`. The following rule-set is implemented inline within the method. However, the *rule-engine* allows you to create (2) types of rules that enable you to encapsulate the rule-set into a single rule.

1. Simple
2. Composite

```
validateCharacters<T>(value: any): Observable<ApiResponse<T>> {
    const validationContext = new ValidationContext();
    const expression: RegExp = /^[a-zA-Z0-9\-\s]*$/;
    validationContext.addRule(
      new StringIsNotNullEmptyRange(
        'KeywordStringIsValid',
        'The keyword value is not valid. Must be within 3 and 20 characters.',
        value,
        3,
        20,
        this.displayValidationMessages
      )
    );

    validationContext.addRule(
      new StringIsRegExMatch(
        'KeywordContainsValidCharacters',
        'The keyword contains invalid characters. The keyword may start and end with\
 alpha-numeric characters; may include a hyphen ("-") within the alpha-numeric chara\
cters.',
        value,
        expression,
        this.displayValidationMessages
      )
    );

    validationContext.renderRules();

    this.loggingService.log(this.serviceName, Severity.Information, `Preparing to va\
lidate characters in the keyword: ${value}`);
    const response = new ApiResponse<T>();
    response.IsSuccess = validationContext.isValid;
    response.Message = validationContext.isValid ? 'The keyword is valid.' : 'The ke\
yword is not valid.';
```

```
    if (!validationContext.isValid) {
      validationContext.results.map(result => {
        const message = new ApiMessage();
        message.isDisplayable = this.displayValidationMessages;
        message.message = result.message;
        message.messageType = ApiMessageType.Error;

        response.Messages.push(message);
      });
    }
    response.Timestamp = new Date();

    return of(response);
  }
```

Business Rule (Shared)

An alternative to adding rules to a `ValidationContext` is to create a shared rule. Add the rule to the barrel file (*index.ts*) for the library to expose it to other consumers that may need to use the rule.

```
import { CompositeRule, StringIsNotNullEmptyRange, StringIsRegExMatch } from '@tc/ru\
le-engine';

export class KeywordIsValidRule extends CompositeRule {
  constructor(name: string, message: string, private keyword: string, isDisplayable:\
 boolean = true) {
    super(name, message, isDisplayable);
    this.configureRules();
  }

  configureRules() {
    const expression: RegExp = /^[a-zA-Z0-9\-\s]*$/;
    this.rules.push(
      new StringIsNotNullEmptyRange(
        'KeywordStringIsValid',
        'The keyword value is not valid. Must be within 3 and 20 characters.',
        this.keyword,
        3,
        20,
        true
      )
    );

```

```
    this.rules.push(
      new StringIsRegExMatch(
        'KeywordContainsValidCharacters',
        'The keyword contains invalid characters. The keyword may start and end with\
 alpha-numeric characters; may include a hyphen ("-") within the alpha-numeric chara\
cters.',
        this.keyword,
        expression,
        true
      )
    );
  }
}
```

Data Repository

The application stack will contain a specific layer for HTTP calls to the application's API.

```
ng g service business/httpSearchRepository --project=brandtrace-search
CREATE libs/brandtrace/search/src/lib/business/http-search-repository.service.spec.t\
s (405 bytes)
CREATE libs/brandtrace/search/src/lib/business/http-search-repository.service.ts (14\
9 bytes)
```

The `HttpSearchRepositoryService` provides an implementation for making HTTP calls using the shared library `HttpService`.

```
import { Injectable } from '@angular/core';
import { ServiceBase } from '@tc/foundation';
import { LoggingService, Severity } from '@tc/logging';
import { Observable, of } from 'rxjs';
import { ApiResponse } from '@tc/common';
import { KeywordSearchRequest, SearchResult, DomainItem } from '@tc/brandtrace/commo\
n';
import { HttpService, HttpRequestMethod } from '@tc/http-service';
import { Guid } from 'guid-typescript';

@Injectable({
  providedIn: 'root',
})
export class HttpSearchRepositoryService extends ServiceBase implements IHttpSearchR\
```

```
epositoryService {
  constructor(private httpService: HttpService, loggingService: LoggingService) {
    super(loggingService, 'HttpSearchRepositoryService');
  }

  // search<T extends SearchResult>(keyword: string): Observable<ApiResponse<T>> {
  search<T extends SearchResult>(searchRequest: KeywordSearchRequest): Observable<an\
y> {
    const requestUrl = `http://brandtracewebapi.stgapi.tcdevops.com/Search`;
    this.loggingService.log(this.serviceName, Severity.Information, `Preparing to ca\
ll keyword search API: ${searchRequest.searchTerm}`);

    const options = this.httpService.createOptions(
      HttpRequestMethod.post,
      this.httpService.createHeader(),
      requestUrl,
      searchRequest,
      false
    );

    return this.httpService.execute(options);
  }
}

export interface IHttpSearchRepositoryService {
  search<T extends SearchResult>(searchRequest: KeywordSearchRequest): Observable<Ap\
iResponse<T>>;
}
```

HTTP Error Handling

Notice that there is no error handling in the *repository* class for the HTTP calls. The `http-service` library contains an HTTP interceptor to catch any errors. This interceptor will determine the error type and process accordingly - with the intent of providing the consumer of the API a response that can be consumed with error details.

```
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent, HttpErrorResponse } f\
rom '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';
import { ApiResponse, ApiErrorMessage } from '@tc/common';

export class HttpErrorInterceptor implements HttpInterceptor {
  displayToUser = true;
  doNotDisplayToUser: false;

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>\
> {
    return next.handle(request).pipe(
      retry(1),
      catchError((error: any) => {
        return this.handleError(error);
      })
    );
  }

  protected handleError(error: HttpErrorResponse): Observable<any> {
    const apiErrorResponse = new ApiResponse();
    apiErrorResponse.isSuccess = false;

    // Use the base error object to determine if the error type is a general or an a\
ll-purpose error.
    if (error.error instanceof ErrorEvent) {
      // A client-side or network error occurred. Handle it accordingly.
      apiErrorResponse.messages.push(new ApiErrorMessage(`A client-side or network e\
rror occurred.`, null));
      return throwError(apiErrorResponse);
    } else {
      // The API returned an unsuccessful response (failure status code).
      if (error instanceof ApiResponse) {
        /**
        - A known error response format from the API/Server; rethrow this response.
         *
        - Throwing the error sends the Observable to the subscriber of the response.
        - The subscriber or consumer should handle the response and display of messa\
ges.
         */
        return throwError(error);
      } else {
```

```
        // An unhandled error/exception - may not want to display this information t\
o an end-user.
        apiErrorResponse.messages.push(
          new ApiErrorMessage(`The API returned an unsuccessful response. ${error.st\
atus}: ${error.statusText}. ${error.message}`, null)
        );
        return throwError(apiErrorResponse);
      }
    }
  }
}
```

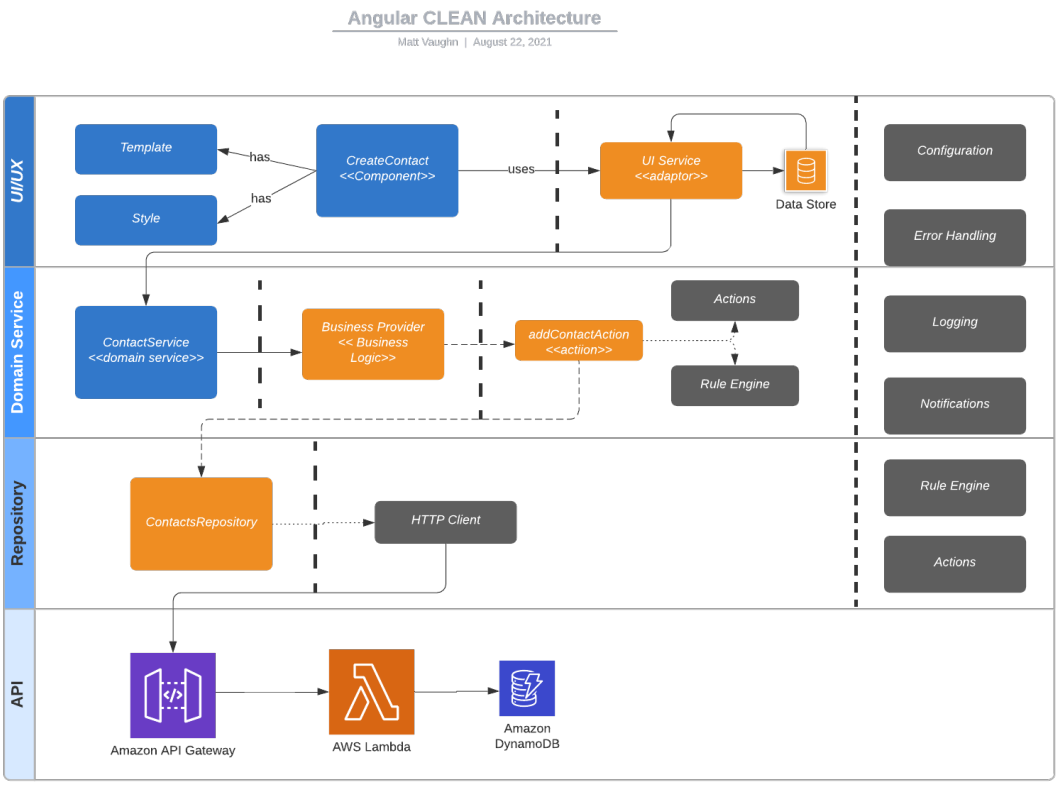
Introduction

Workshop

This workshop will be a single-day of development to create a fullstack application using Angular and NestJS along with Nx.dev tools.

Goals/Objectives

Use a CLEAN architecture principles and approach to create a fullstack application.



- Create Nx Workspace
- Add Angular Application (Portal)
  - workspace configuration
  - Nx commands (lint, test, build, and serve)
- Add Template for Layout and Style
  - main layout
  - header and footer
  - navigation
- Add Cross-Cutting Concern Libraries
- Add Configuration to Portal Application
- Inject Cross-Cutting Libraries
- Add Schematics/Generators
- Add Contact Feature Library
  - Add Create Contact
  - View Contacts (List)
  - Edit Contact
  - Remove Contact
- Add Contact Domain Service Library
- Add API Project (NestJS)
- AWS
- Serverless

Source Code

Git Repository: https://github.com/buildmotion/angular-clean-architecture-workshop[3]

Workshop Repository

```
1 git clone https://github.com/buildmotion/angular-clean-architecture-workshop.git
```

Repository Branches

[3] https://github.com/buildmotion/angular-clean-architecture-workshop

```
git checkout 1_setup/create-nx-workspace
git checkout 1_setup/add-angular-application
git checkout 2_setup/add-cross-cutting-libraries
git checkout 2_setup/add-code-to-x-concern-libs
git checkout 3_app/configure-application
git checkout 4_tools/generators-and-schematics
git checkout 5_accounts/create-ui-library
git checkout 5-2/accounts/add-new-account-form
git checkout 5-3/accounts/add-new-account-template
git checkout 5-4/accounts/new-accounts-ui-service
git checkout 6-1/accounts/domain-service
git checkout 6-2/accounts/handle-api-response
git checkout 7-1/accounts/add-api-project
git checkout 7-2/accounts/add-accounts-controller
git checkout 7-3/accounts/integrate-accounts-api
git checkout 7-4/accounts/api-debugging-tools
```

Setup

Prerequisites

- Visual Studio Code
- Node Version Manager (nvm)
- Git
- AWS Account

Visual Studio Code

Download and install the latest version of Visual Studio Code at https://code.visualstudio.com/download[4].

download-vsc

[4] https://code.visualstudio.com/download

Node

You will need to install Node version `14.17.4` on your machine. There are different ways to do this. I recommend using Node Version Manager (nvm).

Node Version Manager (nvm)

Install Node Version Manager (nvm) on you machine. This will allow you to target specific versions of Node and to change them without requiring a complete removal and install of Node.

- Mac OS installation: https://tecadmin.net/install-nvm-macos-with-homebrew/[5] for instructions.
- Max Big Sur installation: https://andrejacobs.org/macos/installing-node-js-on-macos-big-sur-using-nvm/[6]
- Windows installation: https://dev.to/skaytech/how-to-install-node-version-manager-nvm-for-windows-10-4nbi[7]

```
# install nvm
nvm use 14.17.4
node -v > .nvmrc
node -v #verify version
```

> After installing nmv, use the `nvm install 14.17.4` command to install and use this specific version.

Use the `node -v` command on your terminal to verify Node exists and the version

Git

Use the `git --version` on your terminal to verify Git exists and the version.

```
git version 2.30.1 (Apple Git-130)
```

AWS Account

Create a new free tier account or use your existing AWS account.

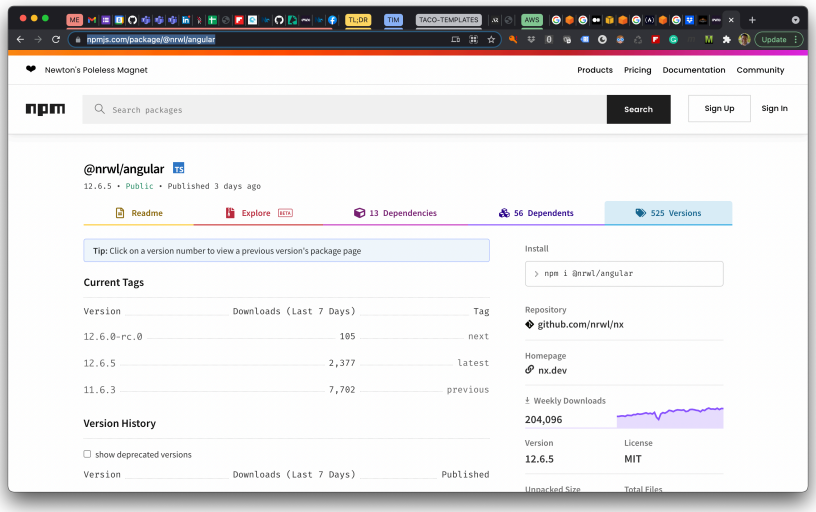
> See: AWS Free Tier - Create a free account[8]

[5] https://tecadmin.net/install-nvm-macos-with-homebrew/
[6] https://andrejacobs.org/macos/installing-node-js-on-macos-big-sur-using-nvm/
[7] https://dev.to/skaytech/how-to-install-node-version-manager-nvm-for-windows-10-4nbi
[8] https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all

Create New Workspace

Using the Nx CLI to create a new workspace will create the workspace using the latest version of the Nrwl package. To target a specific package version you want to go to npmjs.com[9] to determine the specific version number you want to use.

> https://www.npmjs.com/package/@nrwl/angular



https://www.npmjs.com/package/@nrwl/angular

We will target version 11 for our new workspace. We want to use the latest minor/patch release of this major version 11 (e.g., 11.6.3). Use the *npx* command to create an empty workspace for our projects. Now we can add the version number to our `npx` command to specify the version we want.

```
1  npx create-nx-workspace@11.6.3 workspace --npm-scope=buildmotion
```

> Use the `--npm-scope` to create a scope that identify your organization and/or the package group (e.g., @angular/core, @angular/common, @angular/forms).

The output of the CLI command should look similar to the output listed below.

[9] npmjs.com

```
npx create-nx-workspace@11.6.3 workspace --npm-scope=buildmotion
npx: installed 66 in 2.68s
? What to create in the new workspace empty             [an empty workspace with a l\
ayout that works best for building apps]
? Use Nx Cloud? (It's free and doesn't require registration.) No

>  NX  Nx is creating your workspace.

  To make sure the command works reliably in all environments, and that the preset i\
s applied correctly,
  Nx will run "npm install" several times. Please wait.


>  NX   SUCCESS  Nx has successfully created the workspace.
```

Nx Workspace Structure

- multi-project environment
- shared package.json
- shared/inherited configurations
- tools
  - specification tests
  - E2# tests
- Nx CLI

package.json

```
{
  "name": "workspace",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "nx": "nx",
    "start": "nx serve",
    "build": "nx build",
    "test": "nx test",
    "lint": "nx workspace-lint && nx lint",
    "e2e": "nx e2e",
    "affected:apps": "nx affected:apps",
    "affected:libs": "nx affected:libs",
    "affected:build": "nx affected:build",
```

```
    "affected:e2e": "nx affected:e2e",
    "affected:test": "nx affected:test",
    "affected:lint": "nx affected:lint",
    "affected:dep-graph": "nx affected:dep-graph",
    "affected": "nx affected",
    "format": "nx format:write",
    "format:write": "nx format:write",
    "format:check": "nx format:check",
    "update": "nx migrate latest",
    "workspace-generator": "nx workspace-generator",
    "dep-graph": "nx dep-graph",
    "help": "nx help"
  },
  "private": true,
  "dependencies": {
    "tslib": "^2.0.0"
  },
  "devDependencies": {
    "@nrwl/tao": "11.6.3",
    "@nrwl/cli": "11.6.3",
    "@nrwl/workspace": "11.6.3",
    "@types/node": "14.14.33",
    "dotenv": "8.2.0",
    "ts-node": "~9.1.1",
    "typescript": "~4.0.3",
    "prettier": "2.2.1"
  }
}
```

workspace.json

```
{
  "version": 2,
  "projects": {},
  "cli": {
    "defaultCollection": "@nrwl/workspace"
  }
}
```

TypeScript Configuration

```
// workspace/tsconfig.base.json (b16f50f)
{
  "compileOnSave": false,
  "compilerOptions": {
    "rootDir": ".",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "importHelpers": true,
    "target": "es2015",
    "module": "esnext",
    "lib": ["es2017", "dom"],
    "skipLibCheck": true,
    "skipDefaultLibCheck": true,
    "baseUrl": ".",
    "paths": {}
  },
  "exclude": ["node_modules", "tmp"]
}
```

nx.json

```
// nx.json
{
  "npmScope": "buildmotion",
  "affected": {
    "defaultBase": "master"
  },
  "implicitDependencies": {
    "workspace.json": "*",
    "package.json": {
      "dependencies": "*",
      "devDependencies": "*"
    },
    "tsconfig.base.json": "*",
    "tslint.json": "*",
    ".eslintrc.json": "*",
    "nx.json": "*"
  },
  "tasksRunnerOptions": {
    "default": {
```

```
      "runner": "@nrwl/workspace/tasks-runners/default",
      "options": {
        "cacheableOperations": [
          "build",
          "lint",
          "test",
          "e2e"
        ]
      }
    }
  },
  "projects": {}
}
```

.prettierrc

```
// .prettierrc
{
  "singleQuote": true
}
```

Add New Angular Application

Prerequisites

The Nx Workspace will require the `@nrwl/angular` package that contains the *generators* to create a new Angular application.

```
yarn add @nrwl/angular@11.6.3 -D
```

CLI Command

> The Nx CLI Is a tool that contains a set of generators. Each generator or schematic contains templates that generate code, applications, services, or other things in the workspace. Use the generators to quickly scaffold your application and libraries for the workspace.

Use the following command(s) to create a new application project in your workspace.

- The application name is *black-dashboard*.

```
nx generate @nrwl/angular:application black-dashboard
```

The output of the application generate command.

```
CREATE jest.config.js
CREATE jest.preset.js
CREATE apps/black-dashboard/tsconfig.editor.json
CREATE apps/black-dashboard/tsconfig.json
CREATE apps/black-dashboard/src/favicon.ico
CREATE apps/black-dashboard/.browserslistrc
CREATE apps/black-dashboard/tsconfig.app.json
CREATE apps/black-dashboard/src/index.html
CREATE apps/black-dashboard/src/main.ts
CREATE apps/black-dashboard/src/polyfills.ts
CREATE apps/black-dashboard/src/styles.scss
CREATE apps/black-dashboard/src/assets/.gitkeep
CREATE apps/black-dashboard/src/environments/environment.prod.ts
CREATE apps/black-dashboard/src/environments/environment.ts
```

```
CREATE apps/black-dashboard/src/app/app.module.ts
CREATE apps/black-dashboard/src/app/app.component.scss
CREATE apps/black-dashboard/src/app/app.component.html
CREATE apps/black-dashboard/src/app/app.component.spec.ts
CREATE apps/black-dashboard/src/app/app.component.ts
CREATE .eslintrc.json
CREATE apps/black-dashboard/.eslintrc.json
CREATE apps/black-dashboard/jest.config.js
CREATE apps/black-dashboard/src/test-setup.ts
CREATE apps/black-dashboard/tsconfig.spec.json
CREATE apps/black-dashboard-e2e/cypress.json
CREATE apps/black-dashboard-e2e/src/fixtures/example.json
CREATE apps/black-dashboard-e2e/src/integration/app.spec.ts
CREATE apps/black-dashboard-e2e/src/plugins/index.js
CREATE apps/black-dashboard-e2e/src/support/app.po.ts
CREATE apps/black-dashboard-e2e/src/support/commands.ts
CREATE apps/black-dashboard-e2e/src/support/index.ts
CREATE apps/black-dashboard-e2e/tsconfig.e2e.json
CREATE apps/black-dashboard-e2e/tsconfig.json
CREATE apps/black-dashboard-e2e/.eslintrc.json
UPDATE workspace.json
UPDATE package.json
UPDATE .vscode/extensions.json
UPDATE nx.json
```

Available Options

Each *template* has a specific set of options that can be applied when you run the command.

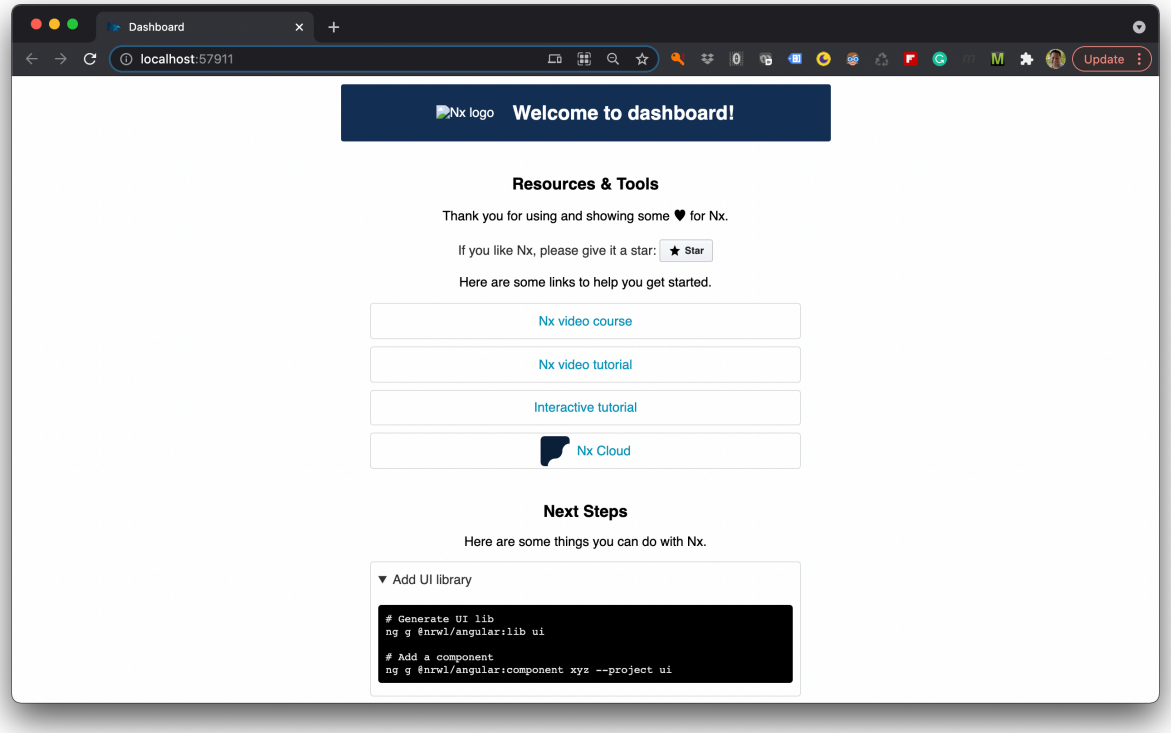
```
nx generate @nrwl/angular:application black-dashboard --help
```

Option	Description
–name	The name of the application.
–directory	The directory of the new application.
–style	The file extension to be used for style files. (default: css)
–routing	Generates a routing module.
–inlineStyle	Specifies if the style will be in the ts file.
–inlineTemplate	Specifies if the template will be in the ts file.
–viewEncapsulation	Specifies the view encapsulation strategy.
–enableIvy	Create a new app that uses the Ivy rendering engine. (default: true)

Option	Description	
–prefix	The prefix to apply to generated selectors.	
–skipTests	Skip creating spec files.	
–skipFormat	Skip formatting files	
–skipPackageJson	Do not add dependencies to package.json.	
–unitTestRunner	Test runner to use for unit tests (default: jest)	
–e2eTestRunner	Test runner to use for end to end (e2e) tests (default: cypress)	
–tags	Add tags to the application (used for linting)	
–linter	The tool to use for running lint checks. (default: eslint)	
–backendProject	Backend project that provides data to this application. This sets up	proxy.config.json.
–strict	Creates an application with stricter type checking and build optimization	options.
–dryRun	Runs through and reports activity without writing to disk.	
–skip-nx-cache	Skip the use of Nx cache.	
–help	Show available options for project target.	

Serve the New Application

```
nx serve dashboard
```



serve-new-app.png

Angular Template

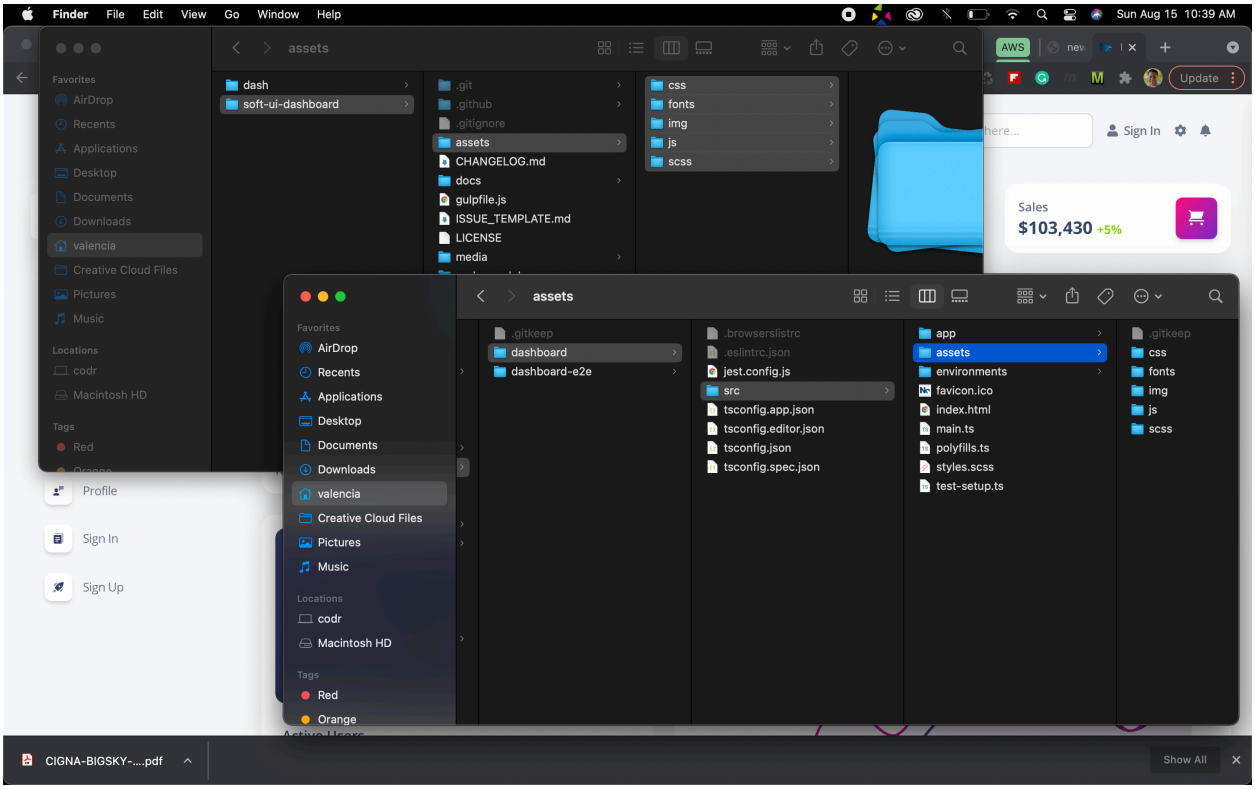
This workshop will take advantage of a template to provide the styles and layout for our application.

- demo: Dashboard Template[10]
- template: Template[11]
- Github.com: https://github.com/creativetimofficial/soft-ui-dashboard[12]

[10] https://demos.creative-tim.com/soft-ui-dashboard-pro/pages/dashboards/default.html?_ga=2.216831702.1978166807.1628202168-1527944961.1625616027

[11] https://www.creative-tim.com/product/soft-ui-dashboard

[12] https://github.com/creativetimofficial/soft-ui-dashboard



copy-assets

Template Setup

The original template contains several layout options and widgets.

The workshop template will be updated to just contain a layout and a few component elements.

- header
- footer
- left navigation

Styles

```
{
  "glob": "**/*",
  "input": "libs/shared/assets/src/assets",
  "output": "assets"
},
```

Update JavaScript References

```
yarn add @popperjs/core@2.9.2
```

Add Not Found Component

> nx g @nrwl/angular:module –name=not-found –project=black-dashboard –module=/app.module –route=not-found –routing –dry-run

```
nx g @nrwl/angular:module --name=not-found --project=black-dashboard --module=/app.m\
odule --route=not-found  --routing
CREATE apps/black-dashboard/src/app/not-found/not-found-routing.module.ts
CREATE apps/black-dashboard/src/app/not-found/not-found.module.ts
CREATE apps/black-dashboard/src/app/not-found/not-found.component.css
CREATE apps/black-dashboard/src/app/not-found/not-found.component.html
CREATE apps/black-dashboard/src/app/not-found/not-found.component.spec.ts
CREATE apps/black-dashboard/src/app/not-found/not-found.component.ts
UPDATE apps/black-dashboard/src/app/app-routing.module.ts
```

Cross-Cutting Concerns

> Angular Architecture: A book on applying enterprise patterns and practices in Angular. This sample chapter talks about cross-cutting conerns:
>
> 1. what they are
> 2. benefits
> 3. how to implement your own
> 4. how to provide configuration to cross-cutting concern
>
> See the book on Leanpub.com - Angular Architecture by Matt Vaughn[13]

What is a cross-cutting concern? It is a concern that crosses the boundaries of your application layers and domain verticals. They are typically agnostic to domain features of the application. They provide useful features to applications like configuration, logging, and error handling. Most applications have requirements to log application events, handle errors and log those errors to a centralized repository. This allows administrators to monitor the health of the application.

A typical cross-cutting concern is something that is not specific to the domain of your application. For example, if you are building a learning management system that include features for authors, students, courses, and videos - it would be something so generic, yet functional, that is used within all or quite a few domain features.

For example, instead of implementing logging directly within an application, we can abstract and separate the cross-cutting concern from the application. I recommend implementing these as library projects. This allows other applications and libraries within an Angular Workspace to use the specified cross-cutting concern library.

Cross-cutting concerns are like many of the infrastructure items that are often not thought about, discussed, or included in the early architectural decisions. Let's say your company has spent 2 or more years of development on a specific project - the current project doesn't have any centralized error handling or logging of information, events, or even errors. How can you quantify that the application is working properly? How will you know if there is a problem with the latest build or deployment? This may seem like an extreme example. However, as a consultant I have seen this very scenario more than once.

It is easier to determine what cross-cutting concerns the application requires early and most likely before a single line of domain feature code is written. I like to think of cross-cutting concerns as required infrastructure concerns that provide value during the development, maintenance, and runtime of the application.

[13] https://leanpub.com/angular-architecture-the-unofficial-guide

These concerns are like the electrical, plumbing, and air systems that are part of a home. Each room of a house has a specific concern and set of features - however, most likely each of the rooms require one if not all of the infrastructure elements listed above. If someone built a new home and these cross-cutting concerns were not addressed before construction, where and how would they be put in retroactively? The home would be an odd mess. We normally follow convention in many parts of our lives, in building homes and other things. Do not let the excitement of building out the features of a new application outweigh the importance of having a good foundation for your application which includes cross-cutting concerns.

> ***Pro Tip***: The minimal cross-cutting concerns for an enterprise Angular application are logging and error handling. I cannot stress the value and importance of these (2) concerns. If there are any issues in your code, you know early if you log and handle errors.

Identify Candidates for Reuse

A good rule to follow when developing applications, is to identify candidates to share with multiple applications or libraries. Implement these candidates as stand-alone library projects. The Angular Workspace with library projects gives us the ability to share and reuse code effectively. Consider it thoughtful development when you think through the solutions with this level of detail.

Putting reusable code in their own libraries is a great organizational strategy. It provides a single source of truth for the specific feature. If the feature needs an enhancement or improvement due to a defect, then you have one place to make that change and then all of the other applications or libraries benefit. This eliminates the need to go to many different applications or many different areas within an application to fix something that should have been implemented as a reusable library.

> ***Pro Tip***: Some or just enough analysis, design, and planning can make a difference of success or failure. Look back at some failed projects or features to determine if a little more preparation could have made a difference.

It is interesting to think that setting up a new development environment could have so much code and features without a single line of the domain application code. I have found over the years that most applications, if not all of them, require these very common cross-cutting concerns. Therefore, implementing them as reusable libraries makes a lot of sense. It is more efficient and effective in terms of managing your code, improving the code, and allowing other applications and libraries to share the code.

Creating reusable libraries in the Angular Workspace is a relatively new topic. The Angular Workspace is now a capability since version 6 of Angular. If you or your team are not taking advantage of the Angular Workspace by creating reusable libraries, then you are missing out on one of the biggest game changers in Angular and web development. These are new capabilities that many web development teams have not had the luxury. Using cross-cutting concern libraries simplifies the implementation of your domain code. The code is much cleaner, the signatures of

constructors and methods are specific to the domain and do not include any cross-cutting concern items.

> Separate early and often. Do not implement things that should be shared directly in an application project.

I take advantage of a configuration library that provides or pushes configuration to each of the cross-cutting concerns during runtime. Many of these cross-cutting concern libraries contain services that can be injected into your application (i.e., services or components). Thus, we get the ability to use dependency injection and to have the ability to use these cross-cutting concerns very easily throughout our Angular application projects.

The reason why I bring up these cross-cutting concerns when we are talking about implementing a feature module, is that every feature module requires these things. Therefore, it is very important that the architect or team lead take the time to define the patterns (recipes) for implementing cross-cutting concerns in feature modules. The architect determines and demonstrates how each feature module service or component logs information, how they handle errors, and how they provide information back to the user. The team lead or architect is also responsible for establishing the different layers of the application within a feature module. They define how each layer communicates with other layers within the feature module.

It may seem odd to a teammate to see so much defined up front. Some team members may not understand why everything needs to be defined up front before developing the feature module. However, if the patterns, practices, and recipes to create each of the feature module elements are well defined, there is more consistency in the implementation. Therefore the code is more consistent and maintainable throughout the lifetime of the application. I have seen entire code bases for large applications to have such consistency; that you cannot tell which team member implemented what part of the code.

> Remember that any application that is deployed to production is an application that requires maintenance.

I have heard some developers say that with so much structure, planning, and design - that it inhibits innovation and creativity. This is not true. To truly innovate and be creative a person needs a full-understanding and knowledge of how something works, not a superficial knowledge. This type of understanding also requires experience over time or at least a strong analysis of the domain. With this level of understanding and knowledge a developer can truly be creative and innovative. Without it, it is just guess work with getting lucky occasionally. Do not confuse the occasional success as full understanding and knowledge of how something works if you didn't do a good-enough analysis of the domain item.

> It is not sufficient to know how to write code and compile it. You need to understand the domain of the application along with its concerns, why it is important and beneficial, who uses it, the business rules and workflow. This takes discipline and commitment.

Putting thought, design and a plan in place early allows for easier maintenance and extensibility later. For example I propose that services and components should all extend from a base class. This base class provides the structure and opportunity to provide common elements, features, methods and properties that all components or services can share. It is an excellent extensibility point.

Logging

Before we write any domain code for the feature we might want to think about how are we going to handle logging or writing information about events that happen in the application. It is important for the developers of the application to know when and where and also in what sequence things take place when business logic is executed. Therefore our application requires a logging service so that we can persist this information.

The simplest way to do this in Angular is to create a service that can be injected into our components and services. This service at the minimum logs information to the console of the application browser or Visual Studio Code environment. This allows you to see information about events, errors, and details about the application during development.

Use the following CLI command to create a new library project within your Angular Workspace. This allows other projects to use the features of the logging module/service.

```
ng g library logging --publishable
```

LoggingService

The following LoggingService is a typical implementation that I use to provide the capture of log events for an application. It only does a few things.

- retrieves configuration for logging from the configuration service
- uses a logEntries$ Observable to publish log events
- the log() method is used by consumers of the LoggingService to add a new log event

> Note: the logging service requires some configuration. Please see the Configuration section for more information on how to provide configuration to cross-cutting concern modules and services.

```
import { Injectable, Optional } from "@angular/core";

import { Severity } from "./severity.enum";
import { IConfiguration } from "@angularlicious/configuration";
import { ConfigurationService } from "@angularlicious/configuration";
import { LogEntry } from "./log-entry";
import { ReplaySubject, Observable } from "rxjs";
import { ILogEntry } from "./i-log-entry";
import { LoggingConfig } from "@angularlicious/configuration";
import { Guid } from "guid-typescript";
import { take } from "rxjs/operators";

@Injectable()
export class LoggingService {
  serviceName = "LoggingService";
  source: string;
  severity: Severity;
  message: string;
  timestamp: Date = new Date();
  applicationName: string;
  version: string;
  isProduction: boolean;
  config: LoggingConfig;
  id: Guid = Guid.create();

  private logEntriesSubject: ReplaySubject<ILogEntry> = new ReplaySubject<
    ILogEntry
  >(1);
  logEntries$: Observable<ILogEntry> = this.logEntriesSubject.asObservable();

  /**
   * The [LoggingService] constructor.
   */
  constructor(@Optional() public configService: ConfigurationService) {
    this.log(
      this.serviceName,
      Severity.Information,
      `Starting logging service [${this.id.toString()}] at: ${this.timestamp}`
    );
    this.initializeService(configService);
  }

  /**
```

```
44     * Use to initialize the logging service. Retrieves
45     * application configuration settings.
46     *
47     * @param configService contains the configuration settings for the application
48     */
49    private initializeService(configService: ConfigurationService) {
50      if (configService) {
51        this.configService.settings$
52          .pipe(take(1))
53          .subscribe(settings => this.handleSettings(settings));
54      }
55    }
56
57    /**
58     * Use to handle settings from the configuration service.
59     * @param settings
60     */
61    handleSettings(settings: IConfiguration) {
62      if (settings) {
63        this.config = settings.loggingConfig;
64
65        this.applicationName =
66          this.config && this.config.applicationName
67            ? this.config.applicationName
68            : "Angular";
69        this.version =
70          this.config && this.config.version ? this.config.version : "0.0.0";
71        this.isProduction =
72          this.config && this.config.isProduction
73            ? this.config.isProduction
74            : false;
75      }
76    }
77
78    /**
79     * Use this method to send a log message with severity and source information
80     * to the application's logger.
81     *
82     * If the application environment mode is [Production], the information will
83     * be sent to a centralized repository.
84     *
85     * @param source
86     * @param severity
```

```
87      * @param message
88      */
89    log(source: string, severity: Severity, message: string, tags?: string[]) {
90      this.source = `${this.applicationName}.${source}`;
91      this.severity = severity;
92      this.message = message;
93      this.timestamp = new Date(Date.now());
94
95      if (tags) {
96        tags.push(`LoggerId:${this.id.toString()}`);
97      } else {
98        tags = [`LoggerId:${this.id.toString()}`];
99      }
100
101     const logEntry = new LogEntry(
102       this.applicationName,
103       this.source,
104       this.severity,
105       this.message,
106       tags
107     );
108     this.logEntriesSubject.next(logEntry);
109   }
110 }
```

Log Entry Items

Notice that there is no code in the LoggingService to actually log or write an event to a repository or to the application console. The logging service publishes log entries using an Observable. This allows a separation of concerns to capture log entries and the actual mechanism to write them. How and where to write a log entry is up to you. All we need is a log writer that is responsible for logging events to a console or a Web API. This log writer subscribes to the logEntries$ Observable and then write new log events from the data stream of new log items. The application creates log items and the LoggingService just publishes them.

Here is the ILogEntry interface that provides the definition of a log entry.

```
import { Severity } from "./severity.enum";

export interface ILogEntry {
  source: string;
  application: string;
  severity: Severity;
  message: string;
  timestamp: Date;
  tags?: string[];
}
```

When you create a new log entry, you can define the Severity level of the log item. The Severity allows you to designate the level of the log entry. During development, you may want to be verbose in your logging and log all entries. You might have a different strategy for production deployments. For example, you may only want to log entries that are Warning or Error designation only. You can add logic to your writers that log information based on the environment and the severity level of the log entry.

I'll use the `Severity.Information` enum option to define log entries throughout the application to see a sequence of events and operations. I rely on these to show and display information relevant to the specific operation and feature. Here is an example of an informational log entry.

```
this.loggingService.log(
  this.componentName,
  Severity.Information,
  `Preparing to load the provider(s) for authentication.`
);
```

Here is a sample Severity enum that includes some options that may be useful for your implementation.

```
export enum Severity {
  Information = 1,
  Warning = 2,
  Error = 3,
  Critical = 4,
  Debug = 5
}
```

Log Entry Writer

So far, we've created a logging service that captures log events from an application. However, these log entries are not very useful unless we can view and monitor the log events. We can create a

helper for this cross-cutting concern to actual write the log entries. Now we have to determine the destination of the log entries.

The easiest destination is the application console. Use the CLI to create a ConsoleWriter service in the Logging library project. Below is an example of a console writer. This implementation is a little unique. Notice that the service extends a LogWriter class. This LogWriter is a TypeScript abstract class[14] that provides the structure and implementation details to implement any number of log writers for the application.

Remember that a log writer only needs to subscribe to the LoggingService and then handle any published log entry events. The writer is a service that is decorated with @Injectable which allows it to be provided to the application when it initializes, just like any other provider. Therefore, you have some control as to what log writers you want to provide for the application.

- The log writer has a LoggingService injected into the constructor.
- Subscribes to the logEntries$ Observable
- handles published log events
- writes the log item to the specified target (console)

```
1  import { LogWriter } from "./log-writer";
2  import { ILogEntry } from "../i-log-entry";
3  import { Severity } from "../severity.enum";
4  import { Injectable } from "@angular/core";
5  import { LoggingService } from "./../logging.service";
6
7  /**
8   * Use this writer to log information to the browser console.
9   */
10 @Injectable()
11 export class ConsoleWriter extends LogWriter {
12   constructor(private loggingService: LoggingService) {
13     super();
14     this.loggingService.logEntries$.subscribe(logEntry =>
15       this.handleLogEntry(logEntry)
16     );
17   }
18
19   handleLogEntry(logEntry: ILogEntry) {
20     this.targetEntry = logEntry;
21     this.execute();
22   }
23
```

[14] https://www.typescriptlang.org/docs/handbook/classes.html#abstract-classes

```
24    /**
25     * No setup required for the console writer.
26     */
27    public setup(): void {}
28
29    /**
30     * Implementation of the abstract method. This will perform the
31     * actual `write` action for the specified writer.
32     */
33    public write(): void {
34      switch (this.targetEntry.severity) {
35        case Severity.Debug:
36          console.debug(this.targetEntry);
37          break;
38        case Severity.Information:
39          console.info(this.targetEntry);
40          break;
41        case Severity.Warning:
42          console.warn(this.targetEntry);
43          break;
44        case Severity.Error:
45          console.error(this.targetEntry);
46          break;
47        case Severity.Critical:
48          console.error(this.targetEntry);
49          break;
50        default:
51          break;
52      }
53    }
54  }
```

Log Writer Abstract Class

The LogWriter is an abstract class. This means that class cannot be instantiated directly - but it can provide the overall structure for any class to become a log writer. Only classes that extend this abstract class can be initialized. Abstract classes are unique in that they can define a set of abstract members (e.g., methods, properties) that require implementation from classes that extend from them. They can also provide implementation members - these members are consumed and available by classes that extend the abstract class. So they provide abstract members like an interface but have the capability to provide implementations for any public members.

Abstract classes are useful to provide common behaviors and a structure for any sub-classes that

extend from it. It is like a super base class that is extensible. They are like a miniature blue-print for a specialized class. Abstract classes are not part of the JavaScript specification - they are built into TypeScript and support many Object-Oriented design principles.

> **Pro Tip**: Use abstract classes and the template method[15] design pattern. It is a powerful pattern to provide a consistent implementation for performing the same operation but using a distinct implementation. Can you think of anything in Angular that uses this pattern? Wait for it...the Angular Component has a life-cycle of events and methods - same pattern.

The template method abstract class implements the ILogWriter interface which has an `execute()` method that requires implementation. You can name your entry point method anything you want - the important thing is that it provides a wrapper around the set of template methods.

```
1  import { ILogEntry } from "../i-log-entry";
2
3  export interface ILogWriter {
4    execute(): void;
5  }
```

The `execute()` method is the entry point into the template method pattern for our writers. We have an entry method that is a controller for a set of methods that define the template or recipe. The recipe in this example is set of methods that provide the implementation of writing log entries. Here is the list or our methods that work together to provide this log writing capability.

- setup()
- validateEntry()
- write()
- finish()

The following is the actual template method that demonstrates the flow of the template methods. Some call this the pipeline or lifecycle of something. It is interesting to note that when you use the RxJS `pipe()` method, you are creating a pipeline of methods that run in a specified sequence when you add any of the RxJS operator methods within the pipe.

[15] https://en.wikipedia.org/wiki/Template_method_pattern

```
  execute(): void {
    this.setup();
    if (this.validateEntry()) {
      this.write();
    }
    this.finish();
  }
```

The LogWriter class below provides default implementation for any classes that extend from it. The template method pattern is very extensible. You can add more methods to the template and provide a default implementation and then all other classes that extend from this abstract class now have that behavior.

The `validateEntry()` method provides a default validation of the information required to log an entry whether it is to the console or to a Web API. If there are no rule violations, the writer invokes the specific writer's `write()` method. In our current code example, it writes the entry to the application's console.

> Note: I use another cross-cutting concern that provides the ability to use default and/or create custom rules for validation. I can reuse the rules-engine in services, components, business logic layers - anywhere there is a need to validate things in a consistent and reliable manner.

```
import { ILogWriter } from "./i-log-writer";
import {
  ValidationContext,
  IsTrue,
  IsNotNullOrUndefined,
  StringIsNotNullEmptyRange
} from "@angularlicious/rules-engine";
import { ILogEntry } from "../i-log-entry";

// @Injectable()
export abstract class LogWriter implements ILogWriter {
  hasWriter: boolean; // = false;
  targetEntry: ILogEntry;

  /**
   * Use this method to execute the write process for the
   * specified [Log Entry] item.
   *
   * Using the [template method] design pattern.
```

```
20     */
21    execute(): void {
22      this.setup();
23      if (this.validateEntry()) {
24        this.write();
25      }
26      this.finish();
27    }
28
29    /**
30     * Use to perform an setup or configuration of the [writer].
31     * The [setup] method runs on all executions of the writer - and
32     * is called before the [write] method.
33     */
34    public abstract setup(): void;
35
36    /**
37     * Use to validate the [log entry] before attempting to write
38     * using the specified [log writer].
39     *
40     * Returns a [false] boolean to indicate the item is not valid.
41     */
42    public validateEntry(): boolean {
43      const validationContext = new ValidationContext();
44      validationContext.addRule(
45        new IsTrue(
46          "LogWriterExists",
47          "The log writer is not configured.",
48          this.hasWriter
49        )
50      );
51      validationContext.addRule(
52        new IsNotNullOrUndefined(
53          "EntryIsNotNull",
54          "The entry cannot be null.",
55          this.targetEntry
56        )
57      );
58      validationContext.addRule(
59        new StringIsNotNullEmptyRange(
60          "SourceIsRequired",
61          "The entry source is not valid.",
62          this.targetEntry.source,
```

```
        1,
        100
      )
    );
    validationContext.addRule(
      new StringIsNotNullEmptyRange(
        "MessageIsValid",
        "The message is required for the [Log Entry].",
        this.targetEntry.message,
        1,
        2000
      )
    );
    validationContext.addRule(
      new IsNotNullOrUndefined(
        "TimestampIsRequired",
        "The timestamp must be a valid DateTime value.",
        this.targetEntry.timestamp
      )
    );

    return validationContext.renderRules().isValid;
  }

  /**
   * Use to implement the actual write of the [Log Entry].
   */
  public abstract write(): void;

  /**
   * Use to finish the process or clean-up any resources.
   */
  public finish(): void {}
}
```

Centralized Log Repository

Now that we have a LoggingService and a recipe for creating log writers we can add different writers that target a different destination (i.e., Web API/database). This is where logging get interesting. When a single page application is released into production, the application runs on a web browser: the client. Any activity or events that you want to log need to be stored in a centralized repository or location. We do not have access to the browser console anymore.

Adding log events to the browser's console does not provide any insight into what the application is doing or its current health. Therefore, we need (2) things:

1. a centralized repository to store and persist log information
2. a way to view log items - which may include searching and filtering

There are several cloud-based solutions that allow you to store application log information and to use their reporting tools to view the log events. I use Loggly[16]. Usually, the cloud-based providers have entry-level offerings that provide basic features for free or at a low cost.

The snippet below is an implementation of our LogWriter for Loggly.

- configuration information is injected into the constructor (ConfigurationService)
  - contains the API key for your Loggly account
  - a boolean indicator to determine if the log item should be written to the console (sendConsoleErrors)
- the writer subscribes to the logEntries$ Observable in the LoggingService
  - when the logging service publishes a new log entry, this writer prepares it for Loggly
- the writer formats the message and pushes the new item onto the LogglyService to send the log information to the cloud-based repository

```
1  import { LogWriter } from "./log-writer";
2  import { ILogEntry } from "../i-log-entry";
3  import { ConfigurationService } from "@angularlicious/configuration";
4  import { Optional } from "@angular/core";
5  import { LogglyService } from "ngx-loggly-logger";
6  import { LoggingService } from "../logging.service";
7  import { IConfiguration, LogglyConfig } from "@angularlicious/configuration";
8
9  export class LogglyWriter extends LogWriter {
10   config: LogglyConfig;
11
12   constructor(
13     @Optional() private configService: ConfigurationService,
14     private loggingService: LoggingService,
15     private loggly: LogglyService
16   ) {
17     super();
18     if (this.configService && this.loggingService) {
19       this.configService.settings$.subscribe(settings =>
```

[16] https://www.loggly.com

```
20          this.handleSettings(settings)
21        );
22        this.loggingService.logEntries$.subscribe(entry =>
23          this.handleLogEntry(entry)
24        );
25      }
26    }
27
28    handleSettings(settings: IConfiguration) {
29      if (settings) {
30        this.config = settings.logglyConfig;
31        this.hasWriter = true;
32        console.log(`Initializing Loggly writer for messages.`);
33      }
34    }
35
36    handleLogEntry(entry: ILogEntry) {
37      if (this.hasWriter) {
38        this.targetEntry = entry;
39        this.execute();
40      }
41    }
42
43    /**
44     * This method is part of the [execute] pipeline. Do not call
45     * this method outside of the context of the execution pipeline.
46     *
47     * Use to setup the [Loggly] writer with an [apiKey] from the
48     * configuration service.
49     *
50     * It will use the configuration service to configure and initialize
51     * and setup a new call to log the information to the writer.
52     */
53    public setup(): void {
54      if (this.hasWriter) {
55        try {
56          this.loggly.push({
57            logglyKey: this.config.apiKey,
58            sendConsoleErrors: this.config.sendConsoleErrors
59          });
60
61          if (this.targetEntry.tags && this.targetEntry.tags.length > 0) {
62            const tags = this.targetEntry.tags.join(",");
```

```
63            this.loggly.push({ tag: tags });
64          }
65        } catch (error) {
66          const message = `${this.targetEntry.application}.LogglyWriter: ${{
67            ...error
68          }}`;
69          console.error(message);
70        }
71      }
72    }
73
74    /**
75     * This method is part of the [execute] pipeline - it will be called if the
76     * current [Log Entry] item is valid and the writer is initialized and ready.
77     */
78    public write(): void {
79      this.loggly.push(this.formatEntry(this.targetEntry));
80    }
81
82    /**
83     * Use this function to format a specified [Log Entry] item. This should be moved
84     * to a specific [formatter] service that can be injected into the specified
85     * writer.
86     * @param logEntry
87     */
88    formatEntry(logEntry: ILogEntry): string {
89      return `application:${logEntry.application}; source:${
90        logEntry.source
91      }; timestamp:${logEntry.timestamp.toUTCString()}; message:${
92        logEntry.message
93      }`;
94    }
95  }
```

The image below shows the browser console with log events. This same information is sent to Loggly. The HTTP request contains the following payload.

```
{
  "text": "application:angularlicious; source:angularlicious.GuideComponent; timesta\
mp:Sat, 16 Nov 2019 13:02:43 GMT; message:Preparing to set [Google Analytics] page v\
iew for [/custom-angular-modules].",
  "sessionId": "9e71eb86-2352-42e5-959e-8efbf60946f5"
}
```

Web page with console logs.

You can use the Loggly website to view, filter, and search for specific log items. This is very practical when you need to determine if there are any health or diagnostic concerns for your application.

Loggly event item.

Error Handling

Another thing to be concerned about is error handling. Angular provides a default error handler.The default error handler writes events to the console on the developer tools in the browser. So we also see a relationship between error handling and logging events in our application. We can provide a custom error handler for our application that does much more than just write the event to the console.

Errors and exceptions happen. Period. There are too many factors and dependencies to prohibit any errors from happening. Therefore, they need to be handled. And systems, users, and application administrators need notifications about errors. It is better to know about an error or exception during development than after a deployment to production.

> It would be an error on our part to not consider adding a custom Error Handler to your Angular Workspace. You need it. This is not one of those things that you put into the category of we'll add this later when we have time.

Here are some things to consider for your error handling scenarios.

1. Error Handling
    - Determine where error handling should take place in the application - responsibility.
    - Should there be a single source of error handling?
    - What do you do with the error details and source?

    - Do you deliver a generic error message, “Oops!” to the user?
    - How do you handle different types of errors?
        - HttpClient
        - Application
        - 3rd-party library
        - API/Server
2. Error Notification
    - Determine if the end user should be notified of the error.
    - Should application/system administrators be notified - how?
3. Error Logging (Tracking)
    - Determine what is required for logging/tracking.
    - Need to understand the context of the error.
    - Do not log too little - need relevant information.
    - When did it occur? What additional information should be included in the log message?
4. Custom Error Classes
    - instanceOf
    - extending Error Classes
    - adding rich meta data

Error Sources

We can categorize `error sources` in (3) groups.

1. External
2. Internal
3. Application

External Errors

External errors are `external` from the running application. In our case, they are external to our Angular application running in a client browser. These occur on servers or APIs outside of our application’s runtime environment. Server errors happen while attempting to process the request or during processing on the server.

- database connection errors
- database errors
- server exceptions
- application not available

Server

Most Angular applications use some kind of back end API(s) or server to perform additional application processing. Even if the Angular application is `serverless` - meaning that it doesn't have its own specific server associated to the application. In this case the Angular application may use several APIs and functions that are hosted by 3rd-party providers (think: APIs for MailChimp, Contentful, Firebase, Medium, or Google Cloud Platform etc.).

Regardless of the source of these `external` errors, an Angular application needs to handle them gracefully.

- 500 Errors: The server failed to fulfill a request.

  Response status codes in the 500-range indicate cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and indicate whether it is a temporary or permanent condition. Likewise, user agents should display any included entity to the user. These response codes are applicable to any request method.

Here is an example of some of the types of `500` Server Errors that can happen.

- **500 Internal Server Error**: A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.[62]
- **501 Not Implemented**: The server either does not recognize the request method, or it lacks the ability to fulfil the request. Usually this implies future availability (e.g., a new feature of a web-service API).[63]
- **502 Bad Gateway**: The server was acting as a gateway or proxy and received an invalid response from the upstream server.[64]
- **503 Service Unavailable**: The server is currently unavailable (because it is overloaded or down for maintenance). Generally, this is a temporary state.[65]

Internal Errors

The error may be due to invalid input provided by the request, failed business rules or failed data validation. The request may be mal-formed - and therefore, cannot be processed (wrong media-type, payload too large, invalid permissions).

- 400 Errors

  This class of status code is intended for situations in which the error seems to have been caused by the client. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents should display any included entity to the user.

Client (Browser) - JavaScript

JavaScript has an Error[17] object that all errors in JavaScript derive from. The standard available properties for an error are as follows:

- columnNumber
- fileName
- lineNumber
- message
- name
- stack

This is the information that we see in the Console of the browser's developer tools. Here is a list of specialized types of errors that can occur.

- EvalError (not thrown any more (remains for compatibility)[18]
- InternalError[19]
- RangeError[20]
- ReferenceError[21]
- SyntaxError[22]
- TypeError[23]
- URIError[24]

Application Errors

Believe it or not, applications can also be the source of `errors`. These could be exceptional - meaning that they are unanticipated. However, when they do happen, the current flow of application flow is redirected to a registered provider for handling the error.

Developers, coders, and software engineers cannot write perfect code. The vast number of conditions, processing paths, algorithms, calculations, and other things that happen during the runtime of an application make it impossible to anticipate all scenarios.

Therefore, errors happen and we see them in the following cases:

1. Business Rule Violations
2. Data Validation Errors
3. Application Exceptions

[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
[18] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/EvalError
[19] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/InternalError
[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RangeError
[21] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError
[22] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError
[23] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError
[24] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/URIError

Handle Errors in Angular

Regardless the origination of an error, an Angular application needs the ability to handle errors. Angular has a default `ErrorHandler` that is provided to the application when the application is `initialized`. This `ErrorHandler` catches and handles all unanticipated errors.

Handling errors means:

- handling the error gracefully
- not allowing the error to disable the progress of the application
- storing information about the error event in a centralized repository
- providing notifications to interested parties.

It is really nice that they Angular platform includes such a feature.

```
1  /**
2   * @license
3   * Copyright Google Inc. All Rights Reserved.
4   *
5   * Use of this source code is governed by an MIT-style license that can be
6   * found in the LICENSE file at https://angular.io/license
7   */
8
9  import {
10   ERROR_ORIGINAL_ERROR,
11   getDebugContext,
12   getErrorLogger,
13   getOriginalError
14 } from "./errors";
15
16 export class ErrorHandler {
17   /**
18    * @internal
19    */
20   _console: Console = console;
21
22   handleError(error: any): void {
23     const originalError = this._findOriginalError(error);
24     const context = this._findContext(error);
25     // Note: Browser consoles show the place from where console.error was called.
26     // We can use this to give users additional information about the error.
27     const errorLogger = getErrorLogger(error);
28
```

```
29      errorLogger(this._console, `ERROR`, error);
30      if (originalError) {
31        errorLogger(this._console, `ORIGINAL ERROR`, originalError);
32      }
33      if (context) {
34        errorLogger(this._console, "ERROR CONTEXT", context);
35      }
36    }
37
38    /** @internal */
39    _findContext(error: any): any {
40      if (error) {
41        return getDebugContext(error)
42          ? getDebugContext(error)
43          : this._findContext(getOriginalError(error));
44      }
45
46      return null;
47    }
48
49    /** @internal */
50    _findOriginalError(error: Error): any {
51      let e = getOriginalError(error);
52      while (e && getOriginalError(e)) {
53        e = getOriginalError(e);
54      }
55
56      return e;
57    }
58  }
59
60  export function wrappedError(message: string, originalError: any): Error {
61    const msg = `${message} caused by: ${
62      originalError instanceof Error ? originalError.message : originalError
63    }`;
64    const error = Error(msg);
65    (error as any)[ERROR_ORIGINAL_ERROR] = originalError;
66    return error;
67  }
```

The actual code for the Angular `ErrorHandler` contains comments and an example.

> Provides a hook for centralized exception handling.

> The default implementation of `ErrorHandler` prints error messages to the `console`. To intercept error handling, write a custom exception handler that replaces this default as appropriate for your app.

The recommendation, although embedded in a comment, is to create our own implementation of the `ErrorHandler` to provide a more direct and customized handling of errors. The default error handler writes errors to the application's console. This is great during development. But it doesn't solve a requirement of having a centralized repository of error messages.

> Angular applications are single-page applications where each application instance is on a browser. We do not have access to a user's browser when the application is published. We want a centralized repository for this information.

The code sample shows that we can create our own class that implements the `ErrorHandler` interface. A custom handler
needs to override and provide a concrete implementation of the `handleError()` method. Additionally, add the `@Injectable` decorator to allow the provider to participate in Angular dependency injection.

> There are (2) ways to provide a singleton service[25] for your application.
>
> 1. use the `providedIn` property, or
> 2. provide the module directly in the AppModule of the application

Since we want to override the default ErrorHandler for the Angular application, please remove the `providedIn` configuration

```
@Injectable({
  //providedIn: "root"
})
class MyErrorHandler implements ErrorHandler {
  handleError(error) {
    // do something with the exception
  }
}
```

Custom Error Handler

Here is a sample Error Handling service that will log error messages using a LoggingService. This implementation requires some configuration to be provided to the service. The `handleError()` determines the type of error and processes accordingly.

[25] https://angular.io/guide/singleton-services

Note: If the error type is a HttpErrorResponse (i.e., response status code is 400s or 500s) type and is **not** an ErrorEvent (i.e., generalized JavaScript error) - there is no operation or logging performed. The HTTP service (a different cross-cutting concern) handles error during the execution of HTTP calls. If there are HTTP errors or an HTTP response contains a known API error response body, the HTTP service will throw an Observable error to allow the consumer to handle it using the Observable pattern.

- the configuration provides a `includeDefaultErrorHandling` boolean property to indicate if logging to the application console should be performed. Normally this is set to false during for production environment deployments.
- a LoggingService is injected into the constructor to allow the service to log error information

Note: the logging service requires some configuration. Please see the Configuration section for more information on how to provide configuration to cross-cutting concern modules and services.

```
@Injectable({
  providedIn: "root"
})
export class ErrorHandlingService extends ErrorHandler {
  serviceName = "ErrorHandlingService";
  config: ErrorHandlingConfig;
  hasSettings: boolean;

  constructor(
    private configService: ConfigurationService,
    private loggingService: LoggingService
  ) {
    super();

    this.init();
  }

  init() {
    this.config = new ErrorHandlingConfig();
    this.config = {
      applicationName: "Angular",
      includeDefaultErrorHandling: true
    };
    this.config.applicationName = "ErrorHandlerService";
    this.config.includeDefaultErrorHandling = false;
    console.warn(`Application [ErrorHandler] is using default settings`);
```

```
27
28      this.configService.settings$
29        .pipe(take(1))
30        .subscribe(settings => this.handleSettings(settings));
31    }
32
33    handleSettings(settings: IConfiguration) {
34      if (settings && settings.errorHandlingConfig) {
35        this.config = settings.errorHandlingConfig;
36        this.hasSettings = true;
37
38        this.loggingService.log(
39          this.config.applicationName,
40          Severity.Information,
41          `Application [ErrorHandler] using configuration settings.`
42        );
43      }
44    }
45
46    /**
47     * Use to handle generalized [Error] items or errors from HTTP/Web
48     * APIs [HttpErrorResponse].
49     *
50     * @param error
51     */
52    handleError(error: Error | HttpErrorResponse): any {
53      if (this.config.includeDefaultErrorHandling) {
54        // use the [super] call to keep default error handling functionality --> conso\
55  le;
56        super.handleError(error);
57      }
58
59      if (this.hasSettings) {
60        // A. HANDLE ERRORS FROM HTTP
61        if (error instanceof HttpErrorResponse) {
62          if (error.error instanceof ErrorEvent) {
63            // A.1: A client-side or network error occurred. Handle it accordingly.
64            const formattedError = `${error.name}; ${error.message}`;
65            this.loggingService.log(
66              this.config.applicationName,
67              Severity.Error,
68              `${formattedError}`
69            );
```

```
70          } else {
71            // A.2: The API returned an unsuccessful response (i.e., 400, 401, 403, et\
72  c.).
73            /**
74             * The [HttpService] should return a response that is consumable by the ca\
75  ller
76             * of the API. The response should include relevant information and error \
77  messages
78             * in a format that is known and consumable by the caller of the API.
79             */
80            noop();
81          }
82        } else {
83          // B. HANDLE A GENERALIZED ERROR FROM THE APPLICATION/CLIENT;
84          const formattedError = `${error.name}; ${error.message}`;
85          this.loggingService.log(
86            this.config.applicationName,
87            Severity.Error,
88            `${formattedError}`
89          );
90        }
91      }
92    }
93  }
```

There is a lot of logic in the `handleError()` method. First, it will use the configuration information to determine if it should log the event using the default base class. This will essentially log the item to the browser console.

```
1  if (this.config.includeDefaultErrorHandling) {
2    // use the [super] call to keep default error handling functionality --> console;
3    super.handleError(error);
4  }
```

Next, there is a check to determine the type of error to handle. There are (2) possible sources and the processing paths are different based on the type.

A. HANDLE ERRORS FROM HTTP
B. HANDLE A GENERALIZED ERROR FROM THE APPLICATION/CLIENT;

A. HANDLE ERRORS FROM HTTP

If the source of the error is from an HTTP operation, the error could originate from JavaScript while preparing the request or when handling the response. We can check the error to determine if it is an instanceof `ErrorEvent`. This type of error is most-likely unanticipated and should be logged.

```
if (error.error instanceof ErrorEvent)
```

The second type of HTTP error is based on the HTTP Status Code of the HTTP response. If the status code is in the 400 or 500 categories, the response is considered to be in an error state. However, since we are most likely working with the application's Web API, we want to defer to the processing logic of our HTTP service (See the Handle HTTP Errors section). There are few reasons for this.

- the web API could be sending a valid response - it just has a specified status code that indicates an error of some sort
- the web API may include and probably should include a payload that provides information about the specified request and any error messages. See API Response Schema/Model section.
  - perhaps sending a list of validation error messages
  - or, it may be a failed business rule
  - the consumer of the web API did not receive a successful response and needs to provide information to the user about the status of the request.
- the consumer of the web API response is expecting an Observable event to handle and process
  - the HTTP service can return the response with the payload that includes the error information as an Observable (application validation or business rule failure)
  - if the web API response payload is not known (server error), the HTTP service can wrap a generic error message in a response format that the consumer is expecting and return it as an Observable.

In this scenario, we will do nothing and allow the HTTP service to handle the error.

```
// A.2: The API returned an unsuccessful response (i.e., 400, 401, 403, etc.).
/**
 * The [HttpService] should return a response that is consumable by the caller
 * of the API. The response should include relevant information and error messages
 * in a format that is known and consumable by the caller of the API.
 */
noop();
```

B. HANDLE A GENERALIZED ERROR FROM THE APPLICATION/CLIENT

This is truly an unexpected JavaScript error of type `Error`. We want the application's custom Error Handler to handle the error by creating a new log event item with a severity status of *Error*.

```
const formattedError = `${error.name}; ${error.message}`;
  this.loggingService.log(
    this.config.applicationName,
    Severity.Error,
    `${formattedError}`
  );
```

Provide Custom Error Handler

Provide the custom error handler in the applications root module `AppModule`, use the `providers` configuration and the `useClass` property with the type of the new `ErrorHandler`. This basically injects our custom class into the application as the new default error handler provider. Providing it at the root level of the application makes the `MyErrorHandler` globally available to the entire application.

```
@NgModule({
  providers: [
    {
      provide: ErrorHandler,
      useClass: MyErrorHandler
    }
  ]
})
class AppModule {}
```

The Angular `ErrorHandler` is initialized very early in the application's load life cycle. Therefore, you need to initialize a custom provider early.

Error Handling References

- Error Handling and Angular[26]
- HTTP Status Codes[27]
- JavaScript Error Object[28]
- Exceptional Exception Handling in JavaScript[29]
- Angular ErrorHandler (error_handler.ts)[30]
- Angular HttpClient :: Error Handling[31]
- Angular HttpResponse[32]

[26] https://medium.com/@aleixsuau/error-handling-angular-859d529fa53a
[27] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#4xx_Client_errors
[28] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
[29] https://www.sitepoint.com/exceptional-exception-handling-in-javascript/
[30] https://github.com/angular/angular/blob/master/packages/core/src/error_handler.ts
[31] https://angular.io/guide/http
[32] https://angular.io/api/common/http/HttpErrorResponse

- Angular HttpResponseBase[33]
- Chocolate in my Peanut Butter[34]

Configuration

Most Angular applications that require configuration will take advantage of the *environment* constant where you can set application-wide configuration items for each of the environments. Angular provides a *fileReplacement* mechanism to assist in this regard. Therefore, we have a convenient mechanism to provide environment-specific configuration for an application.

We can take advantage of the same pattern and create configuration for our cross-cutting concern providers (i.e., services) that need environment and application specific configuration.

Provide Configuration to Cross-Cutting Concern Libraries

Library projects cannot reference or import anything from an application project. If this was possible it would immediately create a circular dependency for the cross-cutting libraries. Therefore, we need a way for a library to get configuration information during the runtime of an application instance.

Define the Configuration

Create a *constant* that provides the structure for your configuration. In this example, an *interface* defines the members of the configuration container. Notice that each of the cross-cutting concerns that need configuration has its own definition (schema/interface) - and it should specific to the environment as well.

- logging
- error handling
- loggly log writer (cloud-based) repository for log items

Each cross-cutting library (module) that needs configuration has a corresponding interface to define the specific configuration members of that item. This separation of concerns keeps things organized and reduces the chance of using the wrong configuration.

[33] https://angular.io/api/common/http/HttpResponseBase
[34] https://www.youtube.com/watch?v=DJLDF6qZUX0

```
import { ILoggingConfig } from './config/i-logging-config';
import { IErrorHandingConfig } from './config/i-error-handling-config';
import { ILogglyConfig } from './config/i-loggly-config';

export interface IConfiguration {
  applicationName: string;
  loggingConfig: ILoggingConfig;
  errorHandlingConfig: IErrorHandingConfig;
  logglyConfig: ILogglyConfig;
}
```

The TypeScript *const* contains a concrete implementation for the application configuration.

```
import {
  IConfiguration,
} from '@angularlicious/configuration';
import {} from '@angularlicious/error-handling';

export const AppConfig: IConfiguration = {
  applicationName: 'Angularlicious.LMS',
  loggingConfig: {
    applicationName: 'Angularlicious.LMS',
    isProduction: false,
  },
  errorHandlingConfig: {
    applicationName: 'Angularlicious.LMS',
    includeDefaultErrorHandling: true,
  },
  logglyConfig: {
    applicationName: 'Angularlicious.LMS',
    apiKey: '11111111-1111-1111-1111-111111111111',
    sendConsoleErrors: true,
  };
}
```

Transform the Configuration (File Replace)

Most of the time, there are subtle differences between development and production environments. Some enterprise applications may have additional environments. We can use the *configuration* node in the architect|build section (see: angular.json file) to configure any file replacements that the configuration requires. The *fileReplacements* is an array of `{replace, with}` items.

> Note: The build configuration for all of the workspace projects is in the *angular.json* file. Find the specific project and update the configuration|production|fileReplacements section.

```
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "apps/lms-admin/src/environments/environment.ts",
        "with": "apps/lms-admin/src/environments/environment.prod.ts"
      },
        {
          "replace": "apps/lms-admin/src/assets/app-config.ts",
          "with": "apps/lms-admin/src/assets/app-config.production.ts"
        }
    ],
  }
}
```

Load the Configuration

Angular applications take advantage of dependency injection to share instances of providers. A provider may be a service, class, or value that is loaded into an *injector*. We also need to provide the configuration which is currently a *TypeScript Object Literal*, essentially an object with data. Each Angular application contains a single root injector that is responsible for injecting all things provided at the root-level.

It is recommended that providers are added to the root-level injector by default if at all possible. This is exactly what needs to happen for the application's configuration and cross-cutting concern services (i.e., logging, error handling, etc.). The application only needs a single instance of these providers.

A root-level injector implies that there can be other levels or branches of injectors within an application. For example, when a module is lazy-loaded a new injector is created for that module. This impacts the use of *providers* from a shared module. It is a common practice to create a *shared* module to group related things together; as well as provide a set of services. When a lazy-loaded module would like to use the items provided by a shared module it has no way of accessing those items. Why? It is because they are scoped to the shared module at this time. However, if we could just provide those items from a shared module to the root-level injector.

In this example, the shared module to configures and provides all of the cross-cutting providers (i.e., services like configuration, error handling, logging, and log writers). When the lazy-loaded module requires a provider with the same Key as one that is loaded in the root-level injector, you would

think by default that the lazy-loaded module would get the provider from the root-level injector. This is *not* the case! What? The lazy-loaded module has its own injector and will initialize its own set of providers required by the module that have the same key/name. But there is another, way that is.

> Note: You may need to create a unique identifier or timestamp for these providers if you want to see that they are actually different instances than the singletons contained in the root-level injector of the application.

In some cases, there may not be any side effects from this behavior. However, if you are expecting all modules lazy-loaded or not to use the same provider (i.e., as a singleton) instance throughout the application we have to do something a little different.

> Learn more about Shared Modules and Dependency Injection[35] at Rangle.io[36].

We will not add items to the *providers* array in the shared `@NgModule` decorator because this will basically provide them using an injector that is specific to the shared module and not the application's root-level injector. Instead, a static `forRoot()` is created to allow a consumer of this *shared* module and its providers to basically *push* the providers to the root-level injector by calling this method.

To provide items from a *shared* module to the root-level injector, use the `forRoot()` static method when importing the module in the *AppModule*. Learn more about Sharing the Same Dependency[37] at Rangle.io. Now that the providers (all of our cross-cutting concern services) are contained in the root-level injector of the application, they will *also* be available to all lazy-loaded modules - magic!

```
@NgModule({
  imports: [
    CrossCuttingModule.forRoot(),
    SharedModule,
    SiteModule,
    AppRoutingModule,
    BrowserAnimationsModule,
  ],
  declarations: [AppComponent, AdminLayoutComponent],
  exports: [],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

The return object of the forRoot() method is an object literal that returns the shared module and all of its providers - including the ones that requires some special handling with the *APP_INITIALIZER*.

[35] https://angular-2-training-book.rangle.io/modules/shared-modules-di
[36] https://rangle.io
[37] https://angular-2-training-book.rangle.io/modules/shared-di-tree

```
/**
 * The factory function to initialize the logging service and writer for the
 * application.
 *
 * @param loggingService
 * @param consoleWriter
 */
export function initializeLogWriter(consoleWriter: ConsoleWriter) {
  console.log(`Initializing [Console Writer] from [AppModule]`);
  return () => {
    return consoleWriter;
  };
}
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    ErrorHandlingModule,
    LoggingModule,
    ConfigurationModule.forRoot({ config: AppConfig }),
    SecurityModule,
  ],
  providers: [
    // DO NOT ADD PROVIDERS HERE WHEN USING [SHARED] MODULES; USE forRoot();
  ],
})
export class CrossCuttingModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: CrossCuttingModule,
      providers: [
        ConfigurationService,
        LoggingService,
        ConsoleWriter,
        LogglyWriter,
        {
          provide: APP_INITIALIZER,
          useFactory: initializeLogWriter,
          deps: [LoggingService, ConsoleWriter, LogglyWriter],
          multi: true,
        },
        {
          provide: ErrorHandler,
```

```
            useClass: ErrorHandlingService,
            deps: [ConfigurationService, LoggingService],
          },
          AuthenticationService,
        ],
      };
    }
}
```

Push the Configuration

There is now a mechanism to define configuration and to load providers at the root-level injector of the application. There is one last step to this entire process. We still need to get the configuration to each provider that has a configuration concern. Leverage the capabilities of Angular and RxJS to push the configuration when it is available.

In the *CrossCuttingModule*, the import of the *ConfigurationModule* calls the `forRoot()` method to provide the configuration to the module. What does this mean? It basically does what we were doing for the shared module - it provides the configuration to the root-level dependency injector. Now the configuration is available to the application.

```
}
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    ErrorHandlingModule,
    LoggingModule,
    ConfigurationModule.forRoot({ config: AppConfig }),
    SecurityModule,
  ],
  providers: [
    // DO NOT ADD PROVIDERS HERE WHEN USING [SHARED] MODULES; USE forRoot();
  ],
})
```

The *ConfigureContext* is provided to the application with the *AppConfig* that contains all of the required configuration information. Now that the configuration is provided and available to access, the ConfigurationService can use the configuration.

```
import { NgModule, ModuleWithProviders } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ConfigurationContext } from './configuration-context';

@NgModule({
  imports: [CommonModule],
})
export class ConfigurationModule {
  static forRoot(configContext: ConfigurationContext): ModuleWithProviders {
    console.log(`Preparing to handle configuration context.`);
    return {
      ngModule: ConfigurationModule,
      providers: [
        {
          provide: ConfigurationContext,
          useValue: configContext,
        },
      ],
    };
  }
}
```

The *ConfigurationContext* is injected into the constructor of the *ConfigurationService*. The service will now take advantage of publishing the configuration via a *readonly* Observable. Notice that the Observable `settings$` accessiblility is read-only. Consumers of the configuration are not allowed to publish any changes to the configuration. Changes are published once when the configuration is available using the *ReplaySubject* of 1.

```
private settings: Subject<IConfiguration> = new ReplaySubject<IConfiguration>(1);
public readonly settings$: Observable<IConfiguration> = this.settings.asObservable();
```

The *ConfigurationService* is a humble service but its job is very important. Its job is to act like a mediator between the application and the providers to push the configuration to any providers that subscribe to the *settings$* Observable.

> Note: the *ConfigurationService* pushes the configuration to the subscribers. This is the Angular way. There is no need for each provider to retrieve its own configuration - the dependency injection pattern is to provide what is asked for and to not allow consumers to create or provide their own dependencies.

```
import { Injectable, Optional } from '@angular/core';
import { Subject, ReplaySubject, Observable } from 'rxjs';
import { IConfiguration } from './i-configuration';
import { ConfigurationContext } from './configuration-context';

@Injectable({
  providedIn: 'root',
})
export class ConfigurationService {
  private settings: Subject<IConfiguration> = new ReplaySubject<IConfiguration>(1);
  public readonly settings$: Observable<IConfiguration> = this.settings.asObservable\
();

  constructor(@Optional() context: ConfigurationContext) {
    if (context) {
      this.settings.next(context.config);
    }
  }
}
```

Configuration Summary

Configuration is a very important aspect of an enterprise application. The more that we make use of libraries and cross-cutting concerns - the more important it is to have the capability to provide configuration to different parts of the application. These may also include core domain libraries that contain application specific business logic for that specified domain. Therefore, create a reliable, consistent, and repeatable way to:

- define configuration schema/interface for a specific concern
- create a concrete configuration based on the interface
- push the configuration to any subscribers
- provide the configuration and cross-cutting concerns to the root-level dependency injector

API Response Schema/Model

Many applications communicate with 3rd-party APIs or the application's own API hosted on a server. Therefore, one main aspect of the application is the ability to make HTTP requests and to properly handle the HTTP response. I think it is important to define a schema for the HTTP response. The schema is a contract between the client and back end Web APIs. This works when you are building the back end Web APIs to support the application and can coordinate what the *shape* or *schema* of the response will look like.

This response schema should include an indicator of success. An API scheme should also include a payload property that contains the data returned by the back end server. If the API has any messages that it would like to provide to the user or the consuming application, it should provide these in a message list in the API response schema.

Having a well defined and well known API schema is very important for the consuming application to handle and process the responses from the back end server. It allows the client application to have a consistent and reliable format and mechanism for processing the response of any API request.

Why Define an API Response

It might seem like an added layer of complexity at the moment. Or, it may seem like it isn't that important. You may be satisfied with making an HTTP/Web API call and dealing with everything in an optimistic mindset as if there will never be any errors in the processing of a request.

The truth is that error will happen. Not all requests are successful and return the payload of data. Additionally, if something goes wrong would it be nice to have the capability to display information to the user to let them know what happened or what they may do to correct the problem.

To put it into perspective, I recently worked on a web form that literally had 6 different formats for various responses (i.e., asynchronous validators and form submission). Displaying any relevant information to the user was an exercise in chaos. The original implementation just displayed the word *Error* without any other information for the user. Very sad! By the way, this application contains about 150 HTTP/Web API calls and each one of them is a *snowflake* (i.e., each request has a different implementation on handling the response and any errors). Many of the response came back as success, status code 200/OK, but only contained cryptic error messages. This is confusing and unnecessary with a little design and planning. Keep it consistent by plan.

> ***Pro Tip***: Define an API Response that is reliable. It will save you hours of frustration during development - and it will allow for a consistent mechanism to communicate information with the users. It supports consistent error handling and processing of HTTP requests.

A Base Response

The `ApiResponse<T>` is an abstract class that provides the basic structure for any HTTP response. This is one that I am currently using. However, I recommend that you work with the Web API developers and agree upon a response schema. There are quite a few styles for a response schema. Please do some research and determine what is most appropriate for you application and context.

- is an *abstract* class to allow for a success and error type response sub-classes
- uses *Generics* to allow you to indicate the expected type for the model/entity
- *IsSuccess* to indicate if the response is successful or not; seems arbitrary at the moment, but is very useful in determine the processing path of the response if it contains any error messages
- *Timestamp*: use to indicate the date and time the response was issued by the server

```
export abstract class ApiResponse<T> {
  IsSuccess: boolean;
  Message: string;
  StatusCode: number;
  Timestamp: Date;
}
```

Success API Response

It is likely that most of the responses you get from your application's Web API are successful and return with a payload of data. The `SuccessApiResponse<T>` class just extends the base class so that it also has common information - no matter if it is successful or not.

- contains a `Data` property of type `T` to allow the client to indicate what the data payload type should be.
- contains a list of *ApiMessage* items to allow the Web API to return any information useful to the application or to display to the user.

```
import { ApiResponse } from './api-response';
import { ApiMessage } from './api-message';

/**
 * Use to define a successful API response. A successful response will
 * most likely include a payload of data (i.e., use the Data property).
 */
export class SuccessApiResponse<T> extends ApiResponse<T> {
  Data: T;
  Messages: ApiMessage[];
}
```

Error API Response

In the unlikely event, the application's Web API returns an error or failure response it should be in a well-defined format to allow the application to process it and retrieve any messages from the response.

> Note that this response does not contain a *Data* property. The expected payload is not available or provided by the API when the operation is in an error state.

You get a list of *messages* that provide valuable information to the application and also to the user. If you need a consistent mechanism to provide information to the user from the back end of you application this is the way to do it.

```
import { ApiResponse } from './api-response';
import { ApiMessage } from './api-message';

/**
 * Use to provide error information from an API. You can also
 * use this class to create a response with errors while doing
 * error handling.
 *
 * Errors: is a list om [ApiErrorMessage] items that contain specific
 * errors for the specified request.
 */
export class ErrorApiResponse<T> extends ApiResponse<T> {
  Errors: ApiMessage[] = [];
}
```

API Messages

This is the equivalent of two tin cups and a string that attaches them together - it is a way to communicate a message from one end of the application to the other end (client). If the application's Web API needs to communicate any information to the application and/or the user of the application it needs a message.

This is a message format that is generic enough to provide messages that are informational, warning, or indicate an error of some sort.

- contains an *ApiMessageSeverity* enum property to indicate the level of severity
- use the *isDisplayable* boolean indicator to determine whether the message is intended for the user or not

```
import { ApiMessageSeverity } from './api-message-severity.enum';

export class ApiMessage {
  id?: string;
  message: string;
  severity: ApiMessageSeverity;
  isDisplayable: boolean;

  /**
   * Use to create a new [ApiErrorMessage]
   * @param message The error from the API.
   * @param displayable Use to indicate if the error should be displayed to the user.
   * @param id An optional identifier for the error.
```

```
14     */
15   constructor(message: string, displayable: boolean, id: string | null) {
16     this.message = message;
17     this.isDisplayable = displayable;
18     if (id) {
19       this.id = id;
20     }
21   }
22 }
```

The *ApiMessageSeverity* is useful if you need to style or format the display of the information based on the severity level. It is nice to be able to provide error messages, but it also a nice feature to provide messages that informative to the user.

```
1  export enum ApiMessageSeverity {
2    Information = 'Information',
3    Warning = 'Warning',
4    Error = 'Error',
5  }
```

HTTP Service

Most applications require the use of Web APIs or a dedicated back end to retrieve and persist information. If the application makes HTTP requests and processes HTTP responses to provide useful data or information, it will need to use the Angular *HttpClient*. In my experience, I have seen many different (i.e., *how*) implementations of the HttpClient to perform HTTP related concerns. Also, *where* these operations are implemented is interesting - never in the same place.

> ***Pro Tip***: When using a layered architecture determine where and how to implement HTTP requests. It should be consistent throughout all of the domain verticals and/or specified layer of the domain item. This is definitely one item where variance, deviation from acceptable patterns/recipes, and chaos could make an application virtually unmaintainable. The long-term effects of *snowflake* implementations is a pile of technical debt that is not easy to overcome. The best thing to do is to eliminate technical debt with proper planning and implementation.

Each domain section will have its own HTTP service that is specific to the API operations that is requires. However, the way or mechanism to construct, execute and handle an HTTP request should be consistent and maintainable. Therefore, make use of a library project to create a new cross-cutting concern for HTTP-related things.

> **Goal**: To enable HTTP/Web API calls to be constructed, managed, and executed using a repeatable and reliable mechanism. Most if not all HTTP requests should be constructed using the same pattern/recipe. There should be little or no reason to not use such a mechanism. When you want to extend or add new features - there will be a single-location to make such changes. The current application I'm working on has over 150 variations of processing HTTP requests.

HTTP Service Responsibilities

The HTTP Service has some basic responsibilities and concerns.

- create the HTTP request **options**
  - set the request *method* (i.e., post, get, put, etc.)
  - add *header* information to the request if required
  - set the target *URL* for the request
  - include an optional *body* payload
- **execute** the request
- handle any **errors**

The following *HttpService* class is basically using and wrapping some concerns using the *HttpClient* from `@angular/common/http`. It may seem trivial at first to create such a class. Consider, where HTTP calls should be make within your application - location matters. Consider how they are implemented and really who or what should have the responsbility or concern. When you think about it, an HTTP call is one of the last tasks you perform within an application for a specific operation. The request is executed and the application has to wait for some kind of response.

Create a new *HttpService* library project and service with the Angular CLI:

```
ng g library httpService
ng g service httpService --project=http-service
```

The sample class below is only doing a few things. Mostly, creating the *options* for an HTTP request. The *options* is a container for the information required to execute an HTTP request. The `execute<T>()` makes use of the Angular HttpClient. Unless you practice the mystical art of *optimistic* programming, there will be errors and exceptions during HTTP operations - please do not ignore these, but handle them.

- **`createOptions()`**: a primary method to construct HTTP options using a specific recipe
- **`createHeader`**: a helper method to add header information to the HTTP options. Modify to suite your needs
- **`execute<T>()`**: a primary method of the service to wrap the HttpClient request
- **`handleError()`**: use to handle errors when they happen

```
import { Injectable } from '@angular/core';
import { HttpRequestMethod } from './http-request-methods.enum';
import {
  HttpHeaders,
  HttpClient,
  HttpResponse,
  HttpErrorResponse,
} from '@angular/common/http';
import { HttpRequestOptions } from './http-request-options';
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
import { ApiResponse } from '@angularlicious/foundation';
import { ErrorApiResponse } from '@angularlicious/foundation';
import { ApiErrorMessage } from '@angularlicious/foundation';

@Injectable()
// { providedIn: 'root', }
export class HttpService {
  constructor(private httpClient: HttpClient) {}

  /**
   * Use to create [options] for the API request.
   * @param method Use to indicate the HttpRequest verb to target.
   * @param headers Use to provide any [HttpHeaders] with the request.
   * @param url Use to indicate the target URL for the API request.
   * @param body Use to provide a JSON object with the payload for the request.
   * @param withCredentials Use to indicate if request will include credentials (coo\
kies), default value is [true].
   */
  createOptions(
    method: HttpRequestMethod,
    headers: HttpHeaders,
    url: string,
    body: any,
    withCredentials: boolean = true
  ): HttpRequestOptions {
    let options: HttpRequestOptions;
    options = new HttpRequestOptions();
    options.requestMethod = method;
    options.headers = headers;
    options.requestUrl = url;
    options.body = body;
    options.withCredentials = withCredentials;
```

```
44      return options;
45    }
46
47    /**
48     * Use to create a new [HttpHeaders] object for the HTTP/API request.
49     */
50    createHeader(): HttpHeaders {
51      let headers = new HttpHeaders();
52      headers = headers.set('Content-Type', 'application/json');
53      return headers;
54    }
55
56    /**
57     * Use to execute an HTTP request using the specified options in the [HttpRequestO\
58  ptions].
59     * @param requestOptions
60     */
61    execute<T>(requestOptions: HttpRequestOptions): Observable<ApiResponse<T>> {
62      console.log(
63        `Preparing to perform request to: ${requestOptions.requestUrl}`
64      );
65      return this.httpClient
66        .request<T>(
67          requestOptions.requestMethod.toString(),
68          requestOptions.requestUrl,
69          {
70            body: requestOptions.body,
71            headers: requestOptions.headers,
72            reportProgress: requestOptions.reportProgress,
73            observe: requestOptions.observe,
74            params: requestOptions.params,
75            responseType: requestOptions.responseType,
76            withCredentials: requestOptions.withCredentials,
77          }
78        )
79        .pipe(
80          retry(1),
81          catchError((errorResponse: any) => {
82            return this.handleError(errorResponse);
83          })
84        );
85    }
86
```

```
87    /**
88     * Use to handle errors during HTTP/Web API operations. The caller expects
89     * an Observable response - this method will either return the response from
90     * the server or a new [ErrorApiResponse] as an Observable for the client to
91     * handle.
92     *
93     * @param error The error from the HTTP response.
94     */
95    protected handleError(error: HttpErrorResponse): Observable<any> {
96      const apiErrorResponse = new ErrorApiResponse();
97      apiErrorResponse.IsSuccess = false;
98      apiErrorResponse.Timestamp = new Date(Date.now());
99      apiErrorResponse.Message = 'Unexpected HTTP error.';
100
101     if (error.error instanceof ErrorEvent) {
102       // A client-side or network error occurred. Handle it accordingly.
103       apiErrorResponse.Errors.push(
104         new ApiErrorMessage(
105           `A client-side or network error occurred. Handle it accordingly.`,
106           true,
107           null
108         )
109       );
110       return throwError(apiErrorResponse);
111     } else {
112       // The API returned an unsuccessful response.
113       if (error instanceof ErrorApiResponse) {
114         // A known error response format from the API/Server; rethrow this response.
115         return throwError(error);
116       } else {
117         // An unhandled error/exception - may not want to lead/display this informat\
118 ion to an end-user.
119         // TODO: MIGHT WANT TO LOG THE INFORMATION FROM error.error;
120         apiErrorResponse.Errors.push(
121           new ApiErrorMessage(
122             `The API returned an unsuccessful response. ${error.status}: ${error.sta\
123 tusText}. ${error.message}`,
124             false,
125             null
126           )
127         );
128         return throwError(apiErrorResponse);
129       }
```

```
    }
  }
}
```

Using the HTTP Service

If the processing of business rules and/or data validation goes well in the business layer of the application the next task to follow is typically an HTTP request to either retrieve, save, or update some information. Each domain vertical probably has a distinct set of Web API calls. Create a new service to handle the specific HTTP requests. An *@Injectable* service for HTTP calls is injected into a business provider within the specific domain service. A service of this type could also be the recipient of configuration information that includes *base URL* or other information required to execute the request.

The sample domain-specific HTTP service below contains an injected *HttpService*. The *LoggingService* cross-cutting concern is also available for this service. Reusing code creates such a good feeling, right? There are only a few things to do to perform an HTTP operation:

1. configure the ***URL*** endpoint for the request
2. setup the ***options*** for the request (uses the URL)
3. ***execute*** the request using the new HttpService.

Note the simplicity in the following example of an application's HTTP service for the *ThingsToDo* domain feature. Consistent code is maintainable code - it is also easier to identify any deviations early.

```
import { Injectable, Inject } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';

import { ApiResponse, ServiceBase } from '@tc/foundation';
import { HttpService, HttpRequestMethod } from '@tc/http-service';
import { LoggingService } from '@tc/logging';
import { IThingsToDoHttpService } from './i-things-to-do-http.service';

@Injectable({
  providedIn: 'root',
})
export class ThingsToDoHttpService extends ServiceBase
  implements IThingsToDoHttpService {
  baseUrl = 'http://mybackend.com/api/';
  noCredentials = false;
  credentialsRequired = true;
```

```
18
19    constructor(
20      @Inject(HttpService) public httpService: HttpService,
21      public loggingService: LoggingService
22    ) {
23      super(loggingService, 'ThingsToDoHttpService');
24    }
25
26    RetrieveThingsToDo<T>(): Observable<ApiResponse<T>> {
27      const requestUrl = this.baseUrl.concat('things');
28      const options = this.httpService.createOptions(
29        HttpRequestMethod.get,
30        this.httpService.createHeader(),
31        requestUrl,
32        null,
33        this.noCredentials
34      );
35      return this.httpService.execute<T>(options);
36    }
37  }
```

The use and implementation of an interface *IThingsToDoHttpService* is totally optional. There may be a case to use such a feature during initial development when the real API is not available or online.

```
1  import { Observable } from 'rxjs';
2  import { ApiResponse } from '@tc/foundation';
3
4  export interface IThingsToDoHttpService {
5    /**
6     * Use to retrieve things to do.
7     */
8    RetrieveThingsToDo<T>(): Observable<ApiResponse<T>>;
9  }
```

An interface-based approach allows you to create a fake HTTP service to provide an implementation that returns fake data. There are many different approaches to using and creating fake data from an API. Interfaces may make the implementation a little easier.

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

import { Observable, of, BehaviorSubject, throwError } from 'rxjs';
import { ThingToDo } from '../models/thingToDo.model';
import { IThingsToDoHttpService } from './i-things-to-do-http.service';

import {
  ApiResponse,
  ServiceBase,
  SuccessApiResponse,
  ErrorApiResponse,
  ApiErrorMessage,
} from '@angularlicious/foundation';
export class ThingsToDoFakeHttpService implements IThingsToDoHttpService {
  things: ThingToDo[] = [];

  RetrieveThingsToDo<T>(): Observable<ApiResponse<T>> {
    const response = new SuccessApiResponse();
    response.IsSuccess = true;
    response.Message = 'Successfully processed request for things.';
    response.Timestamp = new Date(Date.now());
    response.Data = this.loadThings();

    const subject: BehaviorSubject<any> = new BehaviorSubject(response);
    return subject.asObservable();
  }

  loadThings() {
    this.things = [];
    this.things.push(
      new ThingToDo(
        1,
        'Denver Taco Festival',
        'RiNo',
        'Eat tacos with your friends and some many Chihuahuas.'
      )
    );
    this.things.push(
      new ThingToDo(
        2,
        'Smooth Jazz Concert',
        'Arvada, CO',
        'Listen to Kenny G play his favorites.'
```

```
      )
    );

    return this.things;
  }
}
```

The feature module can provide the correct HTTP service based on the *environment* of the application runtime. The sample below demonstrates providing a service with the *useClass* and a ternary operator to determine which service to load: fake or real?

```
@NgModule({
  declarations: [VisitorIndexComponent, ThingsToDoComponent],
  imports: [
    CommonModule,
    SharedModule,
    SiteComponentsModule,
    VisitorRoutingModule,
  ],
  providers: [
    // ThingsToDoService, // NOT PROVIDED HERE, EACH COMPONENT REQUIRES A DISTINCT I\
NSTANCE
    ThingsToDoBusiness,
    {
      provide: 'IThingsToDoHttpService',
      useClass: environment.production
        ? ThingsToDoHttpService
        : ThingsToDoFakeHttpService,
    },
  ],
  schemas: [CUSTOM_ELEMENTS_SCHEMA],
})
export class VisitorModule {}
```

Handle HTTP Errors

Errors fall into the category of ***when*** and not ***if***. It is only a matter of time and circumstance that there will be an error during the operation of an HTTP request. Handling errors is a common practice and discipline in programming. We can *hope* that errors never occur- however, *hope* is never a good strategy for most things in life. A little preparation and forward-thinking will make the application safer.

Many times, the error is from the back end of an application - the Web API. If an error occurs on the back end, you typically get a 400 or 500 status code from the server. However, during the processing

of an HTTP request, it is possible to get an error related to the processing of the HTTP request or the response. Basically, there is a lot of opportunity for things to go wrong.

> Note: This is a specialized handling of errors to use the Observable pattern while performing HTTP operations.

Do not be tempted to use `HttpClient` calls directly in your application components. We need a more reliable, consistent, and maintainable mechanism to handle errors while performing actions that use `HTTP` and Observables. It is also not an ideal situation to display `raw` error information to users.

For example, when using `HttpClient` you can call the `request()` method. Using the RxJs `pipe()` you can also use the `catchError()` which returns an `HttpErrorResponse` to be handled. Handling an error in a single location (not in the component Observable response) allows for greater control - it is also contained in a single location. I know this from experience, A current project that I joined after almost 3 years of development has approximately 200 variations of making HTTP calls and handling responses. The number of variations of error handling in this code is mind-boggling considering that only 2 developers implemented the API calls. Therefore, please consider the following principles in regard to error handling.

- single-responsibility
- DRY: don't repeat yourself
- separation of concerns

> ***Pro Tip***: There are many ways to handle errors. Some of the implementation details may be specific
> to the technology. Become familiar with error and exception handling practices.
>
> Here are the goals for handling the error:
>
> 1. ***inspection***: proactively expect exceptions and provide a mechanism to catch them
> 2. ***interpretation***: provide some information local to the error that gives meaning and context to the error
> 3. ***resolution***: catch and handle unexpected errors or exceptions; provide information to user or application that > helps mitigate or resolve an issue

The sample code below expect that there may be an error while processing the request. If an HTTP error happens, it will attempt to retry the operation at least once with `retry(1)` in the Observable pipeline. If there continues, it will be caught with the `catchError()` RxJS pipeline operation and handled. Handling HTTP errors is a specialized process because the process needs to determine the source of the error to engage the proper handling path. More information about Handling HTTP Errors.

```
execute<T>(requestOptions: HttpRequestOptions): Observable<HttpResponse<ApiResponse<\
T>>> {
    try {
      return this.httpClient.request<T>(
        requestOptions.requestMethod.toString(),
        requestOptions.requestUrl,
        {
          headers: requestOptions.headers,
          observe: requestOptions.observe,
          params: requestOptions.params,
          reportProgress: requestOptions.reportProgress,
          withCredentials: requestOptions.withCredentials
        }
      ).pipe(
        retry(1)
        catchError((errorResponse: any) => {
          return this.handleError(errorResponse);
        })
      );
    } catch (error) {
      this.handleError(error);
    }
  }
```

The `HttpErrorResponse` contains details to determine the **source** of the error. Was it from the server/http or from within the application. This helps us to determine what type of information to provide the user, if any. At a minimum, you could log this information to help monitor the health of the application and determine if any improvements should be made.

HttpErrorResponse[38]: A response that represents an error or failure, either from a non-successful HTTP status - an error while executing the request, or some other failure which occurred during the
parsing of the response.

I updated the signature of the `handleError()` method to include either type of `Error` or type of `HttpErrorResponse` - this allows for specialized handling based on the type of error.

[38] https://angular.io/api/common/http/HttpErrorResponse

```
protected handleError(error: Error | HttpErrorResponse): Observable<any> {
  if(error.error instanceof ErrorEvent)  {
    // A client-side or network error occurred. Handle it accordingly.
  } else {
      // The API returned an unsuccessful response.
  }
  // handler returns an RxJS ErrorObservable with a user-friendly error message.
  // Consumers of the service expect service methods to return an Observable of
  // some kind, even a "bad" one.
  //
  // return throwError(error);
  return throwError(`Hey, you got my chocolate in your peanut butter.`);
}
```

Notice that the `HttpErrorResponse` type implements `Error`. Therefore, it contains information about the HTTP Request and also error information. These classes are already part of the Angular error handling eco-system. We can leverage the use of these types to create a more robust error handling flow for our applications.

```
class HttpErrorResponse extends HttpResponseBase implements Error {
  constructor(init: {...})
  get name: 'HttpErrorResponse'
  get message: string
  get error: any | null
  get ok: false

  // inherited from common/http/HttpResponseBase
  constructor(init: {...}, defaultStatus: number = 200, defaultStatusText: string = \
'OK')
  get headers: HttpHeaders
  get status: number
  get statusText: string
  get url: string | null
  get ok: boolean
  get type: HttpEventType.Response | HttpEventType.ResponseHeader
}
```

The abstract base class for the `HttpResponse` provides the structure for other **HTTP Response** classes:

- HttpErrorResponse
- HttpHeaderResponse
- HttpResponse

```
abstract class HttpResponseBase {
  constructor(init: {...}, defaultStatus: number = 200, defaultStatusText: string = \
'OK')
  get headers: HttpHeaders
  get status: number
  get statusText: string
  get url: string | null
  get ok: boolean
  get type: HttpEventType.Response | HttpEventType.ResponseHeader
}
```

HTTP Error Processing

As mentioned previously, processing HTTP errors during Web API operations involve Observables. We will use the RxJS `catchError()` and then handle the error. The source of the error may be from the status code of the HTTP response or it may be a more generalized JavaScript error while attempting to send the request or handle the response. Therefore, we will need to determine the source of the error.

The `handleError()` implementation will determine what to do with the response. If the HTTP response status code is in an error state and the body of the response is in the expected ErrorApiResponse format, we can simply use the RxJS `throwError()` and send it on its way for handling by the consumer of the API.

If the error doesn't contain a body in our expected format, we can wrap the error information into the expected format (generic message) and send it forward.

```
import { Injectable } from "@angular/core";
import { HttpRequestMethod } from "./http-request-methods.enum";
import {
  HttpHeaders,
  HttpClient,
  HttpResponse,
  HttpErrorResponse
} from "@angular/common/http";
import { HttpRequestOptions } from "./http-request-options";
import { Observable, throwError } from "rxjs";
import { catchError, retry } from "rxjs/operators";
import { ApiResponse } from "@angularlicious/foundation";
import { ErrorApiResponse } from "@angularlicious/foundation";
import { ApiErrorMessage } from "@angularlicious/foundation";

@Injectable()
// { providedIn: 'root', }
```

```
18  export class HttpService {
19    constructor(private httpClient: HttpClient) {}
20
21    /**
22     * Use to create [options] for the API request.
23     * @param method Use to indicate the HttpRequest verb to target.
24     * @param headers Use to provide any [HttpHeaders] with the request.
25     * @param url Use to indicate the target URL for the API request.
26     * @param body Use to provide a JSON object with the payload for the request.
27     * @param withCredentials Use to indicate if request will include credentials (coo\
28  kies), default value is [true].
29     */
30    createOptions(
31      method: HttpRequestMethod,
32      headers: HttpHeaders,
33      url: string,
34      body: any,
35      withCredentials: boolean = true
36    ): HttpRequestOptions {
37      let options: HttpRequestOptions;
38      options = new HttpRequestOptions();
39      options.requestMethod = method;
40      options.headers = headers;
41      options.requestUrl = url;
42      options.body = body;
43      options.withCredentials = withCredentials;
44      return options;
45    }
46
47    /**
48     * Use to create a new [HttpHeaders] object for the HTTP/API request.
49     */
50    createHeader(): HttpHeaders {
51      let headers = new HttpHeaders();
52      headers = headers.set("Content-Type", "application/json");
53      return headers;
54    }
55
56    /**
57     * Use to execute an HTTP request using the specified options in the [HttpRequestO\
58  ptions].
59     * @param requestOptions
60     */
```

```
  execute<T>(requestOptions: HttpRequestOptions): Observable<ApiResponse<T>> {
    console.log(
      `Preparing to perform request to: ${requestOptions.requestUrl}`
    );
    return this.httpClient
      .request<T>(
        requestOptions.requestMethod.toString(),
        requestOptions.requestUrl,
        {
          body: requestOptions.body,
          headers: requestOptions.headers,
          reportProgress: requestOptions.reportProgress,
          observe: requestOptions.observe,
          params: requestOptions.params,
          responseType: requestOptions.responseType,
          withCredentials: requestOptions.withCredentials
        }
      )
      .pipe(
        retry(1),
        catchError((errorResponse: any) => {
          return this.handleError(errorResponse);
        })
      );
  }

  /**
   * Use to handle errors during HTTP/Web API operations. The caller expects
   * an Observable response - this method will either return the response from
   * the server or a new [ErrorApiResponse] as an Observable for the client to
   * handle.
   *
   * @param error The error from the HTTP response.
   */
  protected handleError(error: HttpErrorResponse): Observable<any> {
    const apiErrorResponse = new ErrorApiResponse();
    apiErrorResponse.IsSuccess = false;
    apiErrorResponse.Timestamp = new Date(Date.now());
    apiErrorResponse.Message = "Unexpected HTTP error.";

    if (error.error instanceof ErrorEvent) {
      // A client-side or network error occurred. Handle it accordingly.
      apiErrorResponse.Errors.push(
```

```
104           new ApiErrorMessage(
105             `A client-side or network error occurred. Handle it accordingly.`,
106             true,
107             null,
108             null
109           )
110         );
111         return throwError(apiErrorResponse);
112       } else {
113         // The API returned an unsuccessful response.
114         if (error instanceof ErrorApiResponse) {
115           // A known error response format from the API/Server; rethrow this response.
116           return throwError(error);
117         } else {
118           // An unhandled error/exception - may not want to lead/display this informat\
119 ion to an end-user.
120           // TODO: MIGHT WANT TO LOG THE INFORMATION FROM error.error;
121           apiErrorResponse.Errors.push(
122             new ApiErrorMessage(
123               `The API returned an unsuccessful response. ${error.status}: ${error.sta\
124 tusText}. ${error.message}`,
125               false,
126               null,
127               error.status.toString()
128             )
129           );
130           return throwError(apiErrorResponse);
131         }
132       }
133     }
134 }
```

Cross-Cutting Concern Libraries

TODO: ADD INFORMATION ABOUT CROSS-CUTTING CONCERNS

Create library projects.

Use the `--help` to list the available options for this CLI command.
`--publishable` configures the project for publishing to package registries (public and private)

```
nx g @nrwl/angular:library configuration    --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/configuration
nx g @nrwl/angular:library rule-engine      --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/rule-engine
nx g @nrwl/angular:library logging          --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/logging
nx g @nrwl/angular:library error-handling   --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/error-handling
nx g @nrwl/angular:library actions          --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/actions
nx g @nrwl/angular:library common           --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/common
nx g @nrwl/angular:library http-service     --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/http-service
nx g @nrwl/angular:library foundation       --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/foundation
nx g @nrwl/angular:library validation       --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/validation
nx g @nrwl/angular:library notifications    --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/notifications
nx g @nrwl/angular:library version-check    --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/version-check
nx g @nrwl/angular:library analytics        --simpleModuleName --linter=eslint --imp\
ortPath=@buildmotion/analytics
```

Move code into the library projects.

cp -R ./configuration/src ../../../../../work/github/angular-clean-architecture-workshop/workspace/libs/configuration

```
cp -R ./configuration/src   ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/configuration
cp -R ./rule-engine/src     ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/rule-engine
cp -R ./logging/src         ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/logging
cp -R ./error-handling/src  ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/error-handling
cp -R ./actions/src         ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/actions
cp -R ./common/src          ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/common
cp -R ./http-service/src    ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/http-service
cp -R ./foundation/src      ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/foundation
cp -R ./validation/src      ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/validation
cp -R ./notifications/src   ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/notifications
cp -R ./version-check/src   ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/version-check
cp -R ./analytics/src       ../../../../../work/github/angular-clean-architecture-wo\
rkshop/workspace/libs/analytics
```

Configuration and Cross-Cutting Concerns

Need to create configuration for the application that can be provided to the injectable services from the libraries.

1. add configuration
2. setup and provide the cross-cutting concerns

```
nx g @nrwl/angular:module crossCutting --flat
CREATE apps/black-dashboard/src/app/cross-cutting.module.ts
```

DataDog Logging and Analytics

Create Application Id/Key:

Key: 92719441096c12bebde673af4216f72fb59d6cb8

```
KEY ID
aec645ca-5972-4168-8ffc-f8674bf7c33f
KEY
92719441096c12bebde673af4216f72fb59d6cb8
```

Email Information:

> More information here: https://docs.datadoghq.com/events/guides/email/?tab=json

```
event-hoz8bn44@dtdg.co
```

Nx Generators :: Schematic Tools to Scaffold

Here is a sequence of implementation details that can implement a target domain feature.

1. create a UI library to implement a specific feature with UI/UX components
2. add top-level feature components to the UI library
    1. lazy-loaded route
3. add UI service for feature component(s)
4. create domain library to implement the business logic for a specific feature
    1. the UI services will use these domain libraries
5. add business actions to the domain library to implement discreet business logic items

Create Feature UI Component Library with Modules

Use the Nx CLI command to create an Angular Library project That contains UI elements for a specific domain feature.

> Common use is for a feature UI component library with Modules. Use the module template
> to add component/modules (SCAM) to the project.

```
nx generate @nrwl/angular:library --name=chat-ui --style=scss --directory=chat --imp\
ortPath=@buildmotion/chat/chat-ui --lazy --linter=eslint --routing --simpleModuleNam\
e
nx generate @nrwl/angular:library --name=notifications --style=scss --directory=shar\
ed --importPath=@buildmotion/notifications --lazy --linter=eslint --routing --simple\
ModuleName
```

Add Component Modules to Feature UI Library

The following is an example of adding a single-component with a module to an existing library project called *store*.

```
nx g @nrwl/angular:module --name=cart --project=store --module=/store.module --route\
=cart --routing --dry-run
nx g @nrwl/angular:module --name=products --project=store --module=/store.module --r\
oute=products --routing --dry-run
```

UI Service for Component Module | Component Set

The `UI Service` is an essential part of the Angular CLEAN Architecture pattern. It is responsible for coordinating data and business logic operations between the UI/UX and the *domain service*.

> **Syntax**: nx workspace-schematic ui-service [name] –path=[to-component-folder] –project=[name-of-project]

Option	Description
[name]	Indicate the simple name of the UI service. The schematic will append a `UIService` to the name (e.g., a name of "campaign-list" will create the class name *CampaignList**UIService***)
`path`	Use to indicate the path to the component folder relative to the library's `src/lib` folder.
`project`	Use to provide the name of the library project. See the `nx.json` file for a list of project names or the `workspace.json` file.

Here is a sample of running the Nx CLI generator command to create a new UI Service.

```
nx workspace-schematic ui-service campaignList --path=campaign-list --project=portal\
-campaigns
yarn run v1.22.10
$ relatient360/workspace/node_modules/.bin/tsc -p /relatient360/workspace/tools/tsco\
nfig.generated.json
   Done in 2.10s.
>  NX  Executing your local schematic: ui-service
CREATE libs/portal/campaigns/src/lib/campaign-list/campaign-list-ui.service.ts (398 \
bytes)
```

Domain Library

Adds to an existing library project. Use the Domain Library command to create a new domain library that includes a *domain service.*

```
nx workspace-schematic domain-library --directory=security --importPath=@buildmotion\
/security --name=security-service --dry-run
```

Output for CLI command:

```
nx workspace-schematic domain-library --directory=security --importPath=@buildmotion\
/security --name=security-service --dry-run
nx workspace-schematic domain-library --directory=shared --importPath=@buildmotion/n\
otifications --name=notifications --dry-run
```

Item	Description
nx	Name of the CLI
workspace-schematic	Name of the Nx CLI package used to run Schematics or Generators
domain-library	The name of the *custom* Schematic with templates and options.
–directory	The *directory* option creates or uses a folder in *libs* for the new library project
–importPath	The *importPath* option creates the *@scope name and path* for the new library project.
–name	The *name* option provides a unique name for the library project.
–dry-run	The *dry-run* option allows you to run the CLI command without making any file changes.

NX Executing your local schematic: domain-library

```
nx workspace-schematic domain-library --directory=security --importPath=@buildmotion\
/security --name=security-service --dry-run

>  NX  Executing your local schematic: domain-library

? Which stylesheet format would you like to use? SASS(.scss)  [ http://sass-lang.com\
   ]
CREATE libs/security/security-service/README.md (174 bytes)
CREATE libs/security/security-service/tsconfig.lib.json (465 bytes)
CREATE libs/security/security-service/src/index.ts (47 bytes)
CREATE libs/security/security-service/src/lib/security-service.module.ts (171 bytes)
CREATE libs/security/security-service/tsconfig.json (200 bytes)
CREATE libs/security/security-service/jest.config.js (754 bytes)
CREATE libs/security/security-service/src/test-setup.ts (30 bytes)
CREATE libs/security/security-service/tsconfig.spec.json (236 bytes)
CREATE libs/security/security-service/.eslintrc.json (705 bytes)
CREATE libs/security/security-service/src/lib/security-service.service.ts (662 bytes)
CREATE libs/security/security-service/src/lib/business/business-provider.service.ts \
(1150 bytes)
CREATE libs/security/security-service/src/lib/business/http-security-service-reposit\
ory.service.ts (1245 bytes)
CREATE libs/security/security-service/src/lib/business/i-business-provider.service.t\
s (251 bytes)
CREATE libs/security/security-service/src/lib/business/i-http-security-service-repos\
itory.service.ts (196 bytes)
CREATE libs/security/security-service/src/lib/business/actions/business-action-base.\
ts (1208 bytes)
UPDATE package.json (2831 bytes)
UPDATE workspace.json (13784 bytes)
UPDATE nx.json (1091 bytes)
UPDATE tsconfig.base.json (1562 bytes)
UPDATE jest.config.js (769 bytes)
```

Domain Action

Use the following to create new Business Action classes in domain service libraries.

```
nx workspace-schematic domain-action "retrieveConversation" --project=chat-service
```

Domain Service

Use to add a domain service with the backing `BusinessProvider`, Actions, and HTTP Repository for API calls.

> Adds to an existing library project. Use the Domain Library command to create a new domain library that includes a *domain service*.

```
nx workspace-schematic domain-service --name=<name-of-project> --project=state-machi\
ne --dry-run
```

New Generator (a.k.a. Schematic) project.

```
nx generate @nrwl/workspace:workspace-generator my-generator
```

Add UI/UX library

Setup Accounts UI Library

git checkout 5_accounts/create-ui-library

L

ibrary Project(s)]Generate [Accounts] Library Project(s)

- [ ] generate `accounts-ui` library project

```
nx generate @nrwl/angular:library --name=accounts-ui --style=scss --directory=accoun\
ts --importPath=@buildmotion/accounts/accounts-ui --lazy --linter=eslint --routing -\
-simpleModuleName
```

- [ ] create a default target module to load when the module is lazy-loaded by an application route.

```
nx g @nrwl/angular:module --name=home --project=accounts-ui --module=/accounts-ui.mo\
dule --route=home --routing --dry-run -d
```

- [ ] update component template

```
<!-- libs/accounts/accounts-ui/src/lib/home/home.component.html -->
<div class=" content">
  <div class=" row">
    <!-- ADD STUFF FOR ROW 1 -->
    <h1>Home works</h1>
  </div>
  <div class=" row">
    <!-- ADD STUFF HERE FOR ROW 2 -->
    <h4>More stuff here...</h4>
  </div>
</div>
```

Update Route to the Library Module (Library)

- [ ] add route to the application; lazy-load accounts-ui module

```
// apps/black-dashboard/src/app/app-routing.module.ts
{
  path: 'accounts',
  loadChildren: () => import('@buildmotion/accounts/accounts-ui').then((m) => m.Acco\
untsUiModule)
}
```

- [ ] Update the side bar menu

```
// apps/black-dashboard/src/app/components/sidebar/sidebar.component.ts
  {
    path: '/accounts',
    title: 'Accounts',
    type: 'link',
    icontype: 'tim-icons icon-chart-pie-36',
    rtlTitle: ''
  }
```

C

omponent(s)]Add [Account] Component(s)

```
nx g @nrwl/angular:module --name=new-account      --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=new-account
nx g @nrwl/angular:module --name=verify-account   --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=verify-account
nx g @nrwl/angular:module --name=login            --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=login
nx g @nrwl/angular:module --name=logout           --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=logout
nx g @nrwl/angular:module --name=change-password  --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=change-password
nx g @nrwl/angular:module --name=forgot-password  --project=accounts-ui --module=/ac\
counts-ui.module --routing --route=forgot-password
```

- [ ] add new single component module to create new account (e.g., new-account)
  - [ ] use CLI:
- [ ] extends ComponentBase
- [ ] add reactive form to the component
  - [ ] update template with form and inputs
  - [ ] initialize FormGroup with configuration and validation of controls

Add Validation Service

- [ ] add `ValidationService`

```
nx g @nrwl/angular:service --name=validation --project=validation -d
```

Generate a new library to share data types between Angular and NestJS projects:

```
nx g @nrwl/workspace:library accounts/types
```

Add New Account Form (Component Setup)

- git checkout 5-2/accounts/add-new-account-form

1. add `form: FromGroup` to the component
2. inject `FormBuilder` into the constructor
3. update component class to implement `OnInit`
4. add `this.initializeForm()` to the `ngOnInit()` method.
5. initialize form using `FormBuilder`
    1. set each control that will collect information

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { ComponentBase } from '@buildmotion/foundation';
import { LoggingService, Severity } from '@buildmotion/logging';

@Component({
  selector: 'buildmotion-new-account',
  templateUrl: './new-account.component.html',
  styleUrls: ['./new-account.component.css']
})
export class NewAccountComponent extends ComponentBase implements OnInit {

  form: FormGroup;

  // shared password validators;
  private passwordValidators = [Validators.required, Validators.minLength(8), Valida\
tors.maxLength(128)];

  constructor(
    private formBuilder: FormBuilder,
    loggingService: LoggingService,
    router: Router
```

```
24    ) {
25      super('NewAccountComponent', loggingService, router);
26    }
27
28    ngOnInit(): void {
29      this.loggingService.log(this.componentName, Severity.Information, `Preparing to \
30  initialize component.`);
31      this.initializeForm();
32    }
33
34    initializeForm() {
35      this.loggingService.log(this.componentName, Severity.Information, `Preparing to \
36  initialize the create account form.`);
37      this.form = this.formBuilder.group({
38        emailAddress: new FormControl(undefined, {
39          validators: [Validators.required, Validators.maxLength(100)],
40          asyncValidators: [
41            // ADD ASYNC EMAIL VALIDATION HERE;
42          ],
43          updateOn: 'blur',
44        }),
45        password: new FormControl(undefined, {
46          validators: [...this.passwordValidators],
47          asyncValidators: [],
48        }),
49        passwordConfirm: new FormControl(undefined, {
50          validators: {...this.passwordValidators},
51          updateOn: 'change',
52        }),
53      });
54    }
55  }
```

- [ ] add getters for the form control items

```
  get emailAddress(): AbstractControl {
    return this.form.get('emailAddress') as FormControl;
  }

  get passwordConfirm(): AbstractControl {
    return this.form.get('passwordConfirm') as FormControl;
  }

  get password(): AbstractControl {
    return this.form.get('password') as FormControl;
  }
```

- [ ] update the `NewAccountModule` to import:

1. FormsModule
2. ReactiveFormsModule

```
// libs/accounts/accounts-ui/src/lib/new-account/new-account.module.ts
@NgModule({
  declarations: [
    NewAccountComponent
  ],
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    NewAccountRoutingModule,
    RouterModule.forChild(routes)
  ]
})
export class NewAccountModule { }
```

Update New Account Template

git checkout 5-3/accounts/add-new-account-template

```
<!-- libs/accounts/accounts-ui/src/lib/new-account/new-account.component.html -->
<div class="content">
  <div class="row">
    <!-- REGISTER/CREATE ACCOUNT -->
    <div class="col-md-6">
      <form id="RegisterValidation" [formGroup]="form" (ngSubmit)="onSubmit()" noval\
idate class="bv-content">
        <div class="card">
          <div class="card-header">
            <h4 class="card-title">Register Form</h4>
          </div>
          <div class="card-body">
            <!-- EMAIL ADDRESS -->
            <div class="form-group has-label" [ngClass]="{
                'has-danger': !form.valid && emailAddress.touched && emailAddress.in\
valid,
                'has-success': form.valid && emailAddress.touched && emailAddress.va\
lid
              }">
              <label> Email Address * </label>
              <input class="form-control" name="email" formControlName="emailAddress\
" type="email" />
              <label class="error" *ngIf="!form.valid && emailAddress.touched && ema\
ilAddress.hasError('required')">The email address is required.</label>
              <label class="error" *ngIf="!form.valid && emailAddress.touched && ema\
ilAddress.hasError('maxlength')">The email address max length is 100 characters.</la\
bel>
              <label class="error" *ngIf="!form.valid && emailAddress.touched && ema\
ilAddress.hasError('invalidEmailAddress')">The email address format is not valid.</l\
abel>
            </div>

            <!-- PASSWORD -->
            <div class="form-group has-label" [ngClass]="{
                'has-danger': !form.valid && password.touched && password.invalid,
                'has-success': form.valid && password.touched && password.valid
              }">
              <label> Password * </label>
              <input class="form-control" id="registerPassword" formControlName="pas\
sword" name="password" required=""
                type="password" />
              <label class="error" *ngIf="!form.valid && password.touched && passwor\
d.hasError('required')">This password is required.</label>
```

```
44              <label class="error" *ngIf="!form.valid && password.touched && passwor\
45 d.hasError('minlength')">This password minimum length is 8 characters.</label>
46              <label class="error" *ngIf="!form.valid && password.touched && passwor\
47 d.hasError('maxlength')">This password max length is 128 characters.</label>
48              <label class="error" *ngIf="!form.valid && password.touched && passwor\
49 d.hasError('PasswordStrength')">This password strength is not valid - must have alph\
50 a, numeric, and special characters!</label>
51
52            </div>
53
54            <!-- PASSWORD CONFIRM -->
55            <div class="form-group has-label" [ngClass]="{
56                'has-danger': !form.valid && passwordConfirm.touched && passwordConf\
57 irm.invalid,
58                'has-success': form.valid && passwordConfirm.touched && passwordConf\
59 irm.valid
60              }">
61              <label> Confirm Password * </label>
62              <input
63                class="form-control"
64                formControlName="passwordConfirm"
65                id="registerPasswordConfirmation"
66                name="password_confirmation"
67                type="password" />
68              <label class="error" *ngIf="!form.valid && passwordConfirm.touched && \
69 passwordConfirm.hasError('required')">This confirmation password is required.</label\
70 >
71              <label class="error" *ngIf="!form.valid && passwordConfirm.touched && \
72 passwordConfirm.hasError('passwordMismatch')">This passwords do not match.</label>
73            </div>
74            <div class="category form-category">* Required fields</div>
75          </div>
76          <div class="card-footer text-right">
77            <div class="form-check pull-left">
78              <label class="form-check-label">
79                <input class="form-check-input" formControlName="acceptTermsConditio\
80 ns" name="optionCheckboxes" type="checkbox" />
81                <span class="form-check-sign"> </span> Accept the terms and conditio\
82 ns
83              </label>
84              <label class="error" *ngIf="acceptTermsConditions.touched && acceptTer\
85 msConditions.hasError('required')">You must accept the terms and conditions.</label>
86            </div>
```

```
            <button class="btn btn-danger" type="submit">Register</button>
          </div>
        </div>
      </form>
    </div>
  </div>
  <div class="row">
    <!-- ADD STUFF HERE FOR ROW 2 -->
  </div>
</div>
```

Form

- `formGroup` matches the name of the property in the component
- `onSubmit()` handler for the `ngSubmit` event/click

```
  <form id="RegisterValidation" [formGroup]="form" (ngSubmit)="onSubmit()" novalidat\
e class="bv-content">
```

Input Template

- label changes display when invalid
- input `formControlName` value matches name of control in FormBuilder setup
- error message displays when the form, input and error criteria evaluate to error state

```
<!-- EMAIL ADDRESS -->
<div class="form-group has-label" [ngClass]="{
    'has-danger': !form.valid && emailAddress.touched && emailAddress.invalid,
    'has-success': form.valid && emailAddress.touched && emailAddress.valid
  }">
  <label> Email Address * </label>
  <input class="form-control" name="email" formControlName="emailAddress" type="emai\
l" />
  <label class="error" *ngIf="!form.valid && emailAddress.touched && emailAddress.ha\
sError('required')">The email address is required.</label>
  <label class="error" *ngIf="!form.valid && emailAddress.touched && emailAddress.ha\
sError('maxlength')">The email address max length is 100 characters.</label>
  <label class="error" *ngIf="!form.valid && emailAddress.touched && emailAddress.ha\
sError('invalidEmailAddress')">The email address format is not valid.</label>
</div>
```

Account UI Service

git checkout 5-4/accounts/new-accounts-ui-service

- [ ] implement submit on the component

NOTE: NEED TO INSTALL

- [ ] @angular-devkit/core@11.2.0
- [ ] Update workspace.json version to “1”

See: https://nx.dev/l/n/core-concepts/configuration#version
When the version of workspace.json is set to 2, targets, generators and executor properties are used instead of the version 1 properties architect, schematics and builder.

```
nx workspace-schematic ui-service new-account --path=new-account --project=accounts-\
ui -d
CREATE libs/accounts/accounts-ui/src/lib/new-account/new-account-ui.service.ts (392 \
bytes)
```

- [ ] Update the component to use the new UI service in the `onSubmit()` method.

```
const newAccount: NewAccount = { ...this.form.value };
this.uiService.createAccount(newAccount);
```

- [ ] inject the UI Service in the constructor of the component

```
// libs/accounts/accounts-ui/src/lib/new-account/new-account.component.ts
 private uiService: NewAccountUIService,
```

- [ ] Update the UI Service to include a `createAccount()` method to handle the request

```
// libs/accounts/accounts-ui/src/lib/new-account/new-account-ui.service.ts
import { NewAccount, NewAccountResponse } from '@buildmotion/accounts/types';
import { AccountsService } from '@buildmotion/accounts/accounts-service'

@Injectable()
export class NewAccountUIService extends ServiceBase {

  private isSendingSubject: BehaviorSubject<boolean> = new BehaviorSubject<boolean>(\
false);
  public readonly isSending$: Observable<boolean> = this.isSendingSubject.asObservab\
le();

  constructor(
    private accountsService: AccountsService,
    loggingService: LoggingService, serviceContext: ServiceContext) {
    super('NewAccountUIService', loggingService, serviceContext);
  }

  createAccount(newAccount: NewAccount) {
    this.loggingService.log(this.serviceName, Severity.Information, `Preparing to cr\
eate new account for [${newAccount.emailAddress ?? 'n/a'}]`);
    // this.accountsService.createAccount<NewAccountResponse>(newAccount).subscribe(
      // (response) => this.handleCreateAccountResponse<NewAccountResponse>(response\
),
      // (error) => this.handleCreateAccountError(error),
      // () => this.finishCreateAccount()
    );
  }
}
```

Domain Service

git checkout 6-1/accounts/domain-service

- [ ] generate `accounts-service` library project

Generate accounts-service project.

```
nx workspace-schematic domain-library --directory=accounts  --importPath=@buildmotio\
n/accounts --lint=eslint --name=accounts-service --dry-run
nx workspace-schematic domain-library --directory=security  --importPath=@buildmotio\
n/security --lint=eslint --name=security-service --dry-run
nx workspace-schematic domain-library --directory=products  --importPath=@buildmotio\
n/security --lint=eslint --name=security-service --dry-run
nx workspace-schematic domain-library --directory=orders    --importPath=@buildmotio\
n/security --lint=eslint --name=security-service --dry-run
nx workspace-schematic domain-library --directory=shipments --importPath=@buildmotio\
n/security --lint=eslint --name=security-service --dry-run
```

Add an *action* to implement the business logic with business rules and/or validations.

```
nx workspace-schematic domain-action "createAccount" --project=accounts-service -d
```

Angular Architecture :: Business Actions

Angular-Actions is a framework to build amazing business logic. It is built using Typescript and compliments the Angular-Rules-Engine[39] on npmjs.com[40] or GitHub[41].

Note: v2.0.x requires Angular 5.x packages.

Another Framework - Really??

The word ‘ framework ‘ is not bad. We all need structure in our lives. There is structure all around us in everything we do. Knowing what to expect or how to interact with things in real life is nothing new to us, we do it everyday without even thinking about it, right? Sometimes, we need a little coaxing and training. You all know the sandwich shop where you start by indicating the type of sandwich, the bread, toasted or not toasted, cheese, toppings, etc. compared to ordering a ‘ Big Mac ‘ at McDonald’s. McDonald’s doesn’t ask you if you want your bun toasted or what kind of cheese you want - I guess you can say they are opinionated in how the build a sandwich. Of course, you can ask for ‘ no pickles ‘ and hope it comes out that way.

You may not like it at first, but you will soon learn the ways. You do not say ‘ large ‘ at Starbucks - you will soon be corrected with ‘ Venti ‘. It is just how it works. We work with it and we get what we want.

Opinionated frameworks are like these companies. We learn how to work with each one and know what to expect. Many things in real life are not much different from using a framework to provide some structure around a given process, like developing software.

So, let’s not get hung up on the word ‘ framework ‘ thinking that it will hinder your creativity and brilliance. It is only putting a little structure around what we already do - to allow us to focus on the important things while we create our next masterpiece. I encourage you to investigate the ‘ angular-actions ‘ framework, however, I encourage you more so to use well-known design patterns to implement your business logic. You and your team should have the benefit of a plan, process, and structure for implementing the most important code of the application. If you leave it up to the individual developer, you will get as many variations as there are developers and even more depending on how each deverloper *feels* on any given day. Do you really want to ‘ hope ‘ that your business code works - or would you rather have confidence of quantifiable quality of this code.

[39] https://www.npmjs.com/package/angular-rules-engine
[40] https://www.npmjs.com/package/angular-rules-engine.
[41] https://github.com/buildmotion/angular-rules-engine

What do you want when you develop your applications?

We want to focus on the solution and the value of the application when it is in the hands of our users. We want to know what to expect and to have things familiar to us - so we can just simply write code that is:

- readable (beutiful code)
- maintainable
- extensible
- testable
- focused on the domain

In order to get ‘ jobs ‘ done in life, we use tools. We get familiar with them, hopefully become experts with them and then we create amazing things and/or we just get the job done! Regardless, if we are mowing lawns, baking cakes, writing an email, or building software. What if there was a `tool` that would help you to create amazing business code? What if it was as simple as implementing a well-known ‘ design pattern ‘ that has been around for decades; and has real-life examples, works in the real world, etc.

Good for us that a lot of smart people before us created the ‘ wheel ‘ - we do not need to do that. The design pattern is called ‘ template method `. We use this pattern all the time in real life. So, let's not be afraid of the scary name and the words` design pattern ‘. Right now, you are probably thinking, “Ugh, not a design pattern!”. Relax, you got this.

A design pattern is just a name given to something that is so repeatable and defined that it deserves a name. Learning design patterns isn’t something that you only do by reading a book and you are done. First you learn the concept and then you have some higher-level of understanding and knowledge of how the pattern works. Then over time and using the pattern, you will begin to appreciate its goodness and how you can use the pattern with variations to implement your solutions. You will find that you are already using patterns, you just didn’t realize there is an established name for it.

Template Method - Design Pattern

It was mentioned ealier that this pattern exists everywhere in real life. It is really an abstraction, for example, we can say:

- make a cake
- make coffee
- deposit money
- shop for groceries
- make a taco

What do these things have in common? They are things that are performed, have some steps and/or procedures that usually occur in a defined and repeatable sequence. So we can make a cake by calling an ‘ Execute() ‘ method that returns a ‘ Cake ‘. But what happens internally is:

```
public Execute() : cake {
    // 1. get ingredients
    retrieveIngredients();
    // 2. mix ingredients
    mixIngredients();
    // 3. fill cake pan
    fillCakePan();
    // 4. pre-heat oven
    preHeatOven();
    // 5. bake cake
    bakeCake();
    // 6. decorate cake
    decoreateCake();
    // 7. deliver cake
    return cake;
}
```

Isn't much easier to just say ' Execute() ' and the magic just happens? This is what the template method pattern does for us. We abstract all of the ' methods , the sequence ' of steps by using a single method to execute the ' template . You define the template for what you need done - it is that simple. Everytime, the cake Execute() ' method is called it will use the defined ' template ' to process the request. You get consistent and repeatable results. Is it really possible for ' business logic ' to be implemented using the pattern?

Use the Force, Luke!

Components with page lifecycle events, methods, and hooks are not a new thing. They have been around for a long time. I remember learning about JSPs in 2000 and using ASP.NET Web Forms in 2001. Each of these frameworks had a defined and specific way, some today would call this an ' opinion ', to implement web pages to display interesting things.

I know that I just said the word ' opinion . When most people hear this word, it sometimes invokes negative feelings, or maybe we feel I have my own opinions ' so I do not have to listen to and/or believe opinions of others. Recently, I have heard technology people talk about some frameworks being opinionated, like Angular[42], and it is a good thing. Some opinions are very acceptable because they offer many benefits. Can you think of some Angular opinions that are beneficial?

- Uses ' Typescript ' to allow for type safety.
  * Allows us to write higher quality code with less defects.

[42] https://angular.io

- Provides ‘ structural elements ‘ to build applications (i.e., modules, components, directives, services).
  * Provides consistency in how things are used and arranged.
- Uses ‘ Dependency Injection ‘ to provide things to the other things that need it.
  * Reduces code complexity.
  * Increases the testability of our code - less dependencies.

There is a name for this pattern. It is called the ‘ template method ‘ pattern. It allows you to create an instance of something and call a single method to begin execution of a series/sequence of events and pipeline methods. The sequence is well-defined and always executes consistently. Therefore, many programmers rely on and use this pattern to create a specific flow that is reliable and consistent.

The ‘ @buildmotion/actions ‘ framework is using the same ‘ template method ‘ design pattern that all of these component framework architects have been using for decades. It is a simple, and yet powerful pattern. We have the use of this pattern in UI display elements mentioned previously. However, there really has not been too much in terms of frameworks or tools for implementers of business logic. We have to agree that business logic is really the heart of the application. Without business logic we do not have an application - no matter how pretty the UI looks, right?

When we think about business logic, we are trying to make determinations on what to do. We need to do things like persisting and retrieving data. We might have concerns about the current user - are they authenticated? Is the specified user authorized to do the specified operation? Are there specific business rules that need to be satisfied in order to allow the request to continue? What about data validation? Does the data collected by the application have to meet certain validation rules in order to allow the request to continue? What if you have a team of application developers. Are they allowed to implement code to cover these concerns according to how they feel that day? Is there a way for the entire team to have a structure, framework, or tool that would allow them to do all of these things consistently and ‘ still ‘ allow them to be flexible in their designs and coding tasks?

Why yes, there is! The ‘ @buildmotion/actions ‘ framwork provides a consistent and maintainable mechanism for implementing business logic. More importantly, it provides the ability for 100% code coverage when you unit test. It is often a struggle for teams to unit their business logic. One of the main complaints is that it is too difficult. Why? This is due to the inconsistency of implementation and/or an architecture that was not designed to support unit testing from the beginning. If you are building an enterprise application and you have a team of developers, the application deserves a consistent implementation of business logic. This isn’t the time for any developers to start singing, “I just gotta be me!”. During my 2 decades of building enterprise applications, I have learned (sometimes, the hard way) that you need consistency. For example, I remember applications that had not only a consistent horizontal layered architecture with clear separation of concerns - they also had consistent vertical implementations of the domain. Sometimes more than a dozen vertical domains implemented consistently. As a developer, if you learned and understood the layered architecture, and you also learned and understood a single vertical domain implementation - you could work on any part of the application. It was that consistent.

One of the consulting companies I worked for allowed me to use and define a framework for implementing business logic using the ‘ template method ‘ pattern. One of the other benefits, is that the implementations are consistent between other projects and applications. You have long-term consistency. This has allowed my current team that includes junior developers the opportunity to become productive faster.

Why Use Angular-Actions?

Business logic is the heart of your application. It deserves as much attention if not more than the visible parts of the application. Many times, the architecture or design of the business layer will determine the success of the application. For most business or enterprise applications you will want to strive for the following:

+ Testability:
+ Extensibility:
+ Maintainability:
+ Performance:

There are many concerns for an application beyond just the business logic. Many Angular developers will also need to consider how to implement and integrate the following into their new Angular application. Most tutorials give you the “Hello World” and “Tour of Heroes” as examples of the technology. However, you still need to apply well-established principles and patterns to create an application that has the following:

+ Logging
+ Exception Handling
+ Data Validation
+ Business Rules
+ Authorization (Permission-based)
+ Interaction with Angular Services
+ Custom Services
+ Core Angular Services (i.e., Http, Route)

`angular-actions` is an NPM package that provides a framework for implementing business logic for Typescript applications. The action framework is an object-oriented set of classes that provide a mechanism to create testable, extensible, and maintainable business logic code. ‘ Business Actions ‘ developed using this framework provides a consistency in the implementation of business logic that allows developers to be more productive - as well as enabling developers to become familiar with the code base faster.

The framework implements a `Template Method Design Pattern` [43] - which when combined with the `Angular-Rules-Engine` [44] provides a productive and intuitive development environment to

[43] http://www.dofactory.com/net/template-method-design-pattern
[44] https://www.npmjs.com/package/angular-rules-engine

create complex business logic.

NOTE: The ‘ angular-actions ‘ framework is a port from the ‘ BuildMotion Framework ‘ - which is a Microsoft .NET framework for building .NET Web APIs and Domain Services with a rich business logic layer. It uses ‘ actions ‘ and a ‘ rule engine ‘ to build extensible and maintainable business logic and rules. Available on Nuget at: https://www.nuget.org/packages/BuildMotion/[45] with thousands of downloads combined (Vergosity[46]/BuildMotion).

NPM: https://www.npmjs.com/package/angular-actions[47]

IAction

All actions implement the following ‘ interface . `The` execute() ‘ method is used as the entry point to start the processing pipeline of an action.

```
1 export interface IAction {
2         execute();
3 }
```

The pipeline is a set of methods that execute in a predefined sequence. The framework requires the implementation of only (2) methods: processAction and validateActionResult.

1. start
2. audit
3. preValidateAction
4. evaluateRules
5. postValidateAction
6. preExecuteAction
7. processAction
8. postExecuteAction
9. validateActionResult
10. finish

As you can see from the code below, that the ‘ processAction ‘ will be performed if there are no rule violations. The ‘ evaluateRules() ‘ method will set the action’s ‘ allowExecution ‘ boolean property to false if there are any failed rule evaluations. You can add additional logic in your application to set this property to indicate if the actual business logic contained in the ‘ processAction() ‘ method should be executed.

[45] https://www.nuget.org/packages/BuildMotion/
[46] https://www.nuget.org/packages/Vergosity.Framework/
[47] https://www.npmjs.com/package/angular-actions

```
import {ValidationContext} from 'angular-rules-engine/validation/index';
import {ValidationContextState} from 'angular-rules-engine/validation/index';
import {IAction} from './IAction';
import { ActionResult } from './index';

/**
 * This is the framework Action class that provides the pipeline of pre/post
 * execution methods. This class implements the [Template Method] pattern.
 *
 * The pre-execute functions that can be implemented are:
 *               1. start();
 *               2. audit();
 *               3. preValidateAction();
 *               4. evaluateRules();
 *               5. postValidateAction();
 *               6. preExecuteAction();
 *
 *If the status of action is good, the business logic will be executed using the:
 *               1. processAction();
 *
 * The post-execution functions that can be implemented are:
 *               1. postExecuteAction();
 *               2. validateActionResult();
 *               3. finish();
 */
export class Action implements IAction {
    allowExecution = true;
    _validationContext: ValidationContext = new ValidationContext();
    actionResult: ActionResult = ActionResult.Unknown;

    constructor() {}

    get validationContext(): ValidationContext {
        return this._validationContext;
    }

        /**
         * Use this method to execute a concrete action. A concrete action must implement
         * the [processAction] and the [validateActionResult] functions to be a valid
         * action.
         */
    execute() {
        console.log('Preparing to execute action.');
```

```
44          this.processActionPipeline();
45      }
46
47      private processActionPipeline() {
48          this.startAction();
49          if (this.allowExecution) {
50              this.processAction();
51          }
52          this.finishAction();
53      };
54
55      private startAction() {
56          console.log('Starting action.');
57          this.start();
58          this.audit();
59          this.preValidateAction();
60          this.evaluateRules();
61          this.postValidateAction();
62          this.preExecuteAction();
63      }
64
65      private finishAction() {
66          console.log('Finishing action.');
67          this.postExecuteAction();
68          this.validateActionResult();
69          this.finish();
70      }
71
72
73      private processAction() {
74          console.log('Processing action.');
75          this.performAction();
76      }
77
78          /**
79           * All action must implement this function. This is where your
80           * [business logic] should be implemented. This function is called if
81           * there are no validation rule exceptions.
82           */
83      performAction() {
84          throw new Error('Not implemented. Requires implementation in concrete action\
85  .');
86      }
```

```
87
88          /**
89           * Override/Implement this function to perform an early operation in the action pip\
90  eline.
91           * This function belongs to the pre-execute functions of the action pipeline.
92           */
93      start() {
94          console.log('Starting action.');
95      }
96
97          /**
98           * Implement this function to perform any auditing features during the pre-exectuio\
99  n of the
100          * business logic.
101          */
102     audit() {
103         console.log('Auditing action.');
104     }
105
106         /**
107          * Use this function to setup any validation rules before the validation happens. T\
108 his
109          * function is called before [evaluateRules].
110          */
111     preValidateAction() {
112         console.log('Pre-validating action.');
113     }
114
115         /**
116          * Use this function to implement the execution of the validation and business rule\
117 s. This
118          * function is called after [preValidateAction].
119          */
120     evaluateRules(){
121         console.log('Evaluating action rules.');
122         let context = this.validateAction();
123         if (context.isValid) {
124             this.allowExecution = true;
125             this.validationContext.state = ValidationContextState.Success;
126         } else {
127             this.allowExecution = false;
128             this.validationContext.state = ValidationContextState.Failure;
129         }
```

```
130     }
131
132         /**
133          * Use to determine or handle the results of the rule evaluation. This
134          * function is called after the [evaluateRules].
135          */
136     postValidateAction() {
137         console.log('Post-Validation of action.');
138     }
139
140         /**
141          * Use this function to perform any setup before the action is executed.
142          */
143     preExecuteAction() {
144         console.log('Pre-execution of action.');
145     }
146
147         /**
148          * Use this function to evaluate the action after the the business logic within
149          * the [performAction] has executed.
150          */
151     postExecuteAction() {
152         console.log('Post-execution of action');
153     }
154
155         /**
156          * This function requires implementation to determine the state and result of the a\
157 ction.
158          * Use this opportunity to validate the results.
159          */
160     validateActionResult(): ActionResult {
161         throw new Error('Concrete actions required to implement this method.');
162     }
163
164         /**
165          * Use this function to perform any cleanup, logging, or disposing of resources used
166          * by the action. This is the last function called during the pipeline.
167          */
168     finish() {
169         console.log('Finish action.');
170     }
171
172         /**
```

```
173          * Implement this function to perform validation of business rules and data.
174          */
175     validateAction() {
176         console.log('Validating the action.');
177         return this.validationContext;
178     }
179 }
```

A typical implementation of an ‘ action ‘ would be a concrete business action that ‘ extends ‘ from a base action class. The base action class extends from the framework’s ‘ action ‘ class. This structure allows for the concrete actions to focus on the specific business logic (using Single-Responsibility principle) while having common or shared implementation of the pipeline methods in the base class.

As you can see from the code snippet below - the responsibility of the action is to get a ‘ TimeSpan ‘ between two different data objects. This ‘ GetTimeSpanAction ‘ action extends the base class ‘ ActionBase . `The code in this action is clean. Basically, we are elevating business logic from implementations in` methods ‘ of a class to its own class. Implementing business logic with actions allow you to use all of the goodness of object-oriented programming, such as: inheritance, abstraction, encapsulation, and polymorphism. OOP allows for the use of common design patterns that simplify many programming efforts.

Note the implementation of the business rules in the ‘ preValidateAction ‘ method. This allows the rules to be setup and ready for evaluation when the ‘ evaluateRules() ‘ method is called using the framework’s action pipeline. The solution now has a consistent mechanism to implement rules, evaluate rules, and retrieve rule results. Consistency improves quality and the ability to isolate issues using unit and specification tests.

```
1  export class GetTimeSpanAction extends ActionBase {
2
3      response: TimeSpan;
4
5      /**
6       * Constructor for the action.
7       * @param startDate Required.
8       * @param endDate Required.
9       */
10     constructor(
11         private startDate: DateTime,
12         private endDate: DateTime) {
13         super(new LoggingService());
14         this.actionName = 'GetTimeSpanAction';
15     }
16
17     /**
```

```
18      * Use this action pipeline method to add validation or business rules to the Va\
19  lidationContext before the
20      * [evaluateRules] action pipeline method is called.
21      */
22      preValidateAction() {
23          // both dates need to be valid and not null/undefined;
24          this.validationContext.addRule(new rules.IsNotNullOrUndefined('StartDateIsNo\
25  tNull', 'The start date is not valid. Cannot be null or undefined.', this.startDate,\
26   true));
27          this.validationContext.addRule(new rules.IsNotNullOrUndefined('EndDateIsNotN\
28  ull', 'The end date is not valid. Cannot be null or undefined.', this.endDate, true)\
29  );
30      }
31
32      performAction() {
33          const span = DateTime.between(this.startDate, this.endDate);
34          console.log(`the timespan between ${this.startDate.toUniversalTime.toJsDate(\
35  )} and ${this.endDate.toUniversalTime.toJsDate()} is ${span.minutes} minutes.`);
36          this.response = span;
37      }
38
39      /**
40       * Use this action pipeline method to validate the result of the action - based \
41  on rule violations.
42       */
43      validateActionResult(): ActionResult {
44          this.loggingService.log(this.actionName, LoggingSeverity.Information, `Runni\
45  ng [validateActionResult] for ${this.actionName}.`);
46          if (this._validationContext.hasRuleViolations()) {
47              this.businessProvider.loggingService.log(this.actionName, LoggingSeverit\
48  y.Error, `The ${this.actionName} contains rule violations.`);
49              this.actionResult = ActionResult.Fail;
50          }
51          this.actionResult = this.serviceContext.isGood() ? ActionResult.Success : Ac\
52  tionResult.Fail;
53          return this.actionResult;
54      }
55  }
```

The base class ‘ ActionBase ‘ provides the opportunity to implement shared/common functionality of all actions that extend from this class. It allows access to ‘ infrastructure ‘ concerns but doesn’t dirty the concrete implementations of concrete actions.

You can implement any of the ‘ angular-actions ‘ pipeline methods, like:

- validateAction()
- postValidateAction()
- postExecuteAction()

You can also implement shared helper functions/methods that each of the actions will have access to:

- compareLogItemEventDateTime()
- setRegulation()

The implementation of the ‘ Do() ‘ method is a nice extension-like method that allows for the injection of infrastructure services to all actions that extend from this base action class. It also has the added benefit of simplifying the signatures of the concrete actions to contain only the members that the action is concerned with. We all hate method signatures that contain useful but non-relevant information to the concern of the business logic. Things like:

- logging services
- data access
- connection strings
- user information
- ...

```
export class ActionBase extends Action {
    loggingService: LoggingService;
    serviceContext: ServiceContext;
    businessProvider: BusinessProvider;
    actionName: string;

    // Use the following property(s) to provide contextual rule/regulation informati\
on in validation responses.
    RuleSet: string = '';
    Regulation: Regulation = new Regulation();
    HasRegulation: boolean = false;//default for action implementation; set to true \
if a regulation is setup

    constructor(loggingService: LoggingService) {
        super();
        this.loggingService = loggingService;
    }

    /**
```

```
20      * Use the [Do] method to perform the action.
21      */
22      Do(businessProvider: BusinessProvider) {
23          this.businessProvider = businessProvider;
24          this.serviceContext = businessProvider.serviceContext;
25          this.loggingService = businessProvider.loggingService;
26
27          this.execute(); //entry point into the pipeline of methods (template method \
28  pattern)
29      }
30
31      /**
32       * A helper method to setup regulation information for actions that are processi\
33  ng federal regulatory rules.
34       * @param regulationId: A unique identifier for the regulation.
35       * @param regulationText: A description of the regulation.
36       * @param url Use to indicate the regulation source online.
37       */
38      setRegulation(regulationId: string, regulationText: string, url: string = '') {
39          if (regulationId && regulationText) {
40              this.HasRegulation = true;
41              this.Regulation.RegulationId = regulationId;
42              this.Regulation.Regulation = regulationText;
43              this.Regulation.Url = url;
44          }
45          else {
46              this.loggingService.log(this.actionName, LoggingSeverity.Error, `Cannot \
47  setup regulation information without a valid identifier and regulation text.`);
48          }
49      }
50
51      /**
52      * This is a required implementation if you want to render/execute the rules tha{\
53  t j
54      * are associated to the specified action.
55      */
56      validateAction(): ValidationContext {
57          return this.validationContext.renderRules();
58      }
59
60      postValidateAction() {
61          this.loggingService.log(this.actionName,
62              LoggingSeverity.Information,
```

```
63              `Preparing to determine if the action contains validation errors in ${th\
64  is.actionName}`);
65
66          if (this.validationContext.hasRuleViolations()) {
67              this.loggingService.log(this.actionName, LoggingSeverity.Information, `T\
68  he target contains validation errors in ${this.actionName}`);
69
70              // Load the error/rule violations into the ServiceContext so that the in\
71  formation bubbles up to the caller of the service;
72              this.validationContext.results.forEach((e) => {
73                  if (!e.isValid && e.rulePolicy.isDisplayable) {
74                      let serviceMessage = new Message(e.rulePolicy.name, e.rulePolicy\
75  .message, MessageType.Error);
76                      serviceMessage.DisplayToUser = true;
77                      serviceMessage.Source = this.actionName;
78                      serviceMessage.RuleSet = this.RuleSet;
79                      if (this.HasRegulation) {
80                          serviceMessage.Regulation = this.Regulation;
81                      }
82                      this.serviceContext.Messages.push(serviceMessage);
83                  }
84              });
85          }
86      }
87
88      postExecuteAction() {
89          if (this.actionResult === ActionResult.Fail) {
90              this.serviceContext.Messages.forEach((e) => {
91                  if (e.MessageType === MessageType.Error) {
92                      this.loggingService.log(this.actionName, LoggingSeverity.Error, \
93  e.toString());
94                  }
95              });
96          }
97      }
98
99      /**
100      * Use this method to sort a LogItem by the [EventDateTime] property.
101      * @param a
102      * @param b
103      */
104     protected compareLogItemEventDateTime(a: LogItem, b: LogItem) {
105         if (a.EventDateTime < b.EventDateTime) {
```

```
106             return -1;
107         }
108         if (a.EventDateTime > b.EventDateTime) {
109             return 1;
110         }
111         return 0;
112     }
113 }
```

Using the action shown above is easy. You instantiate the action, pass in the required parameters. Call the ‘ Do(this) ‘ method of the action and return the response. Notice how clean and consistent the code is for calling business actions implemented using the ‘ angular-actions ‘ NPM package.

```
1 getTimeSpan(startDate: DateTime, endDate: DateTime): TimeSpan {
2     let action = new actions.GetTimeSpanAction(startDate, endDate);
3     action.Do(this);
4     return action.response;
5 }
```

Testable

Extensible

Maintainable

Performant

[48] http://www.buildmotion.com
[49] https://www.linkedin.com/in/matt-vaughn-857a982?trk=profile-badge

Business Rules and Validation

Angular Rules Strategy

Today, most web applications are not simple - especially when they include forms, APIs and a database. Modern web applications now contain complex business rules and data validation requirements. Most business applications collect a fair amount of data. This means that the application or software must verify or validate that the information is correct before making expensive API calls. In addition to this, there may be other business rules that must be evaluated to ensure that the business logic is correct.

Therefore, what is your strategy to implement business rules and data validation? Most likely if you have five developers on your development team you will have five different ways to manage business rules and data validation. Over a period of time, the variance in the implementation of this type of code will create technical debt that will be difficult to overcome. If the application supports revenue generation then it is essential that the code is conventional, consistent, repeatable, and for sure testable.

Game Plan and Strategy

> Hope is not a good strategy. If you hope the code will just work and the business rules will be fine without any thought or planning, good luck!

There are different sources in your application that may require validation. The validation and business rule strategy may include the Angular form validators. However, this is just the first layer of validation and business rule processing of the application. As the code progresses towards making an API call there are other opportunities for validation and business rule processing. Most likely there will be different types of validation using different mechanism.

I like to imagine a set of safety nets that are layered depending on the layer of the application (UI/UX, UI Services, domain service layers, business logic, and data access/HTTP calls). This is where it gets interesting. In lower layers of your application you cannot use the reactive form `Validators` that implement the `ValidatorFn` *interface*. Therefore you might want to consider using a rule engine to create a more conventional way to manage validation and business rules.

Type of Validation and Rules

1. form input (synchronous)
2. asynchronous validation of form input that may include an API operation
3. business logic with business rules
    1. simple rules
    2. complex rules
4. data validation before an API call

Occasionally you might need to create a custom validator for a form input. This custom validator might be synchronous or async. Either way, you will need to write some custom code. The validator must also be testable to verify the results. It should be a pure function[50] - see an example here[51].

Motivation

Two core principles of great software design are Separation of Concerns (SoC)[52] and Single Responsibility[53]. Business rules and validation are an integral part of most business applications. There are rules and validations that must occur during processing of business logic. Most applications will combine the business logic with rules and data validation - when this happens, testing and maintaining applications becomes more difficult.

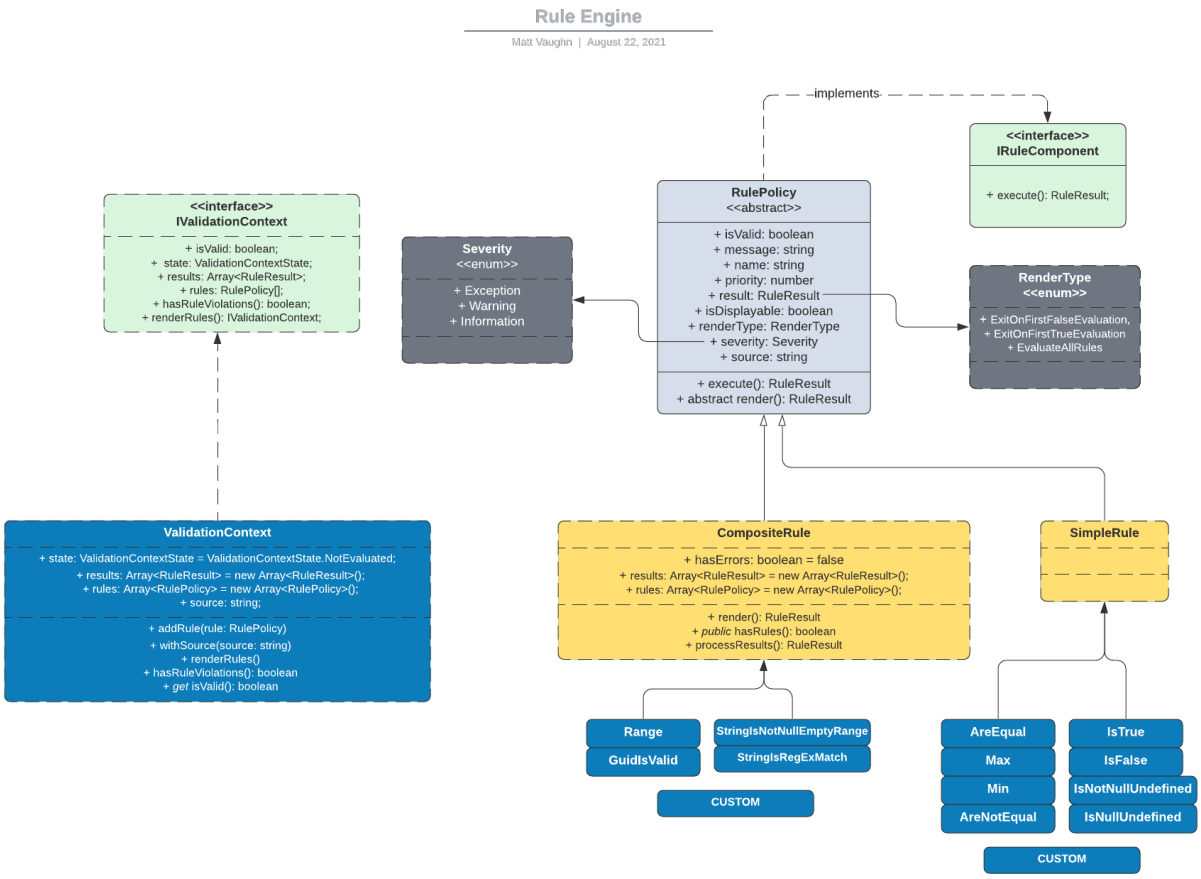
A rule engine allows the application to have a good Separation of Concerns (SOR). A good rule engine allows you to:

+ Quickly start using out-of-the-box rules that are already implemented.
+ Create custom rules that are either simple or composite.
+ Create rules that can be reused throughout the application. Code reuse eliminates copy/paste of common rules.
+ Use a single ValidationContext to add rules, execute rules, and evaluate the rule results.
+ Use a consistent pattern and mechanism to implement your business rules and data validation.
+ Integrate rules with Reactive Form custom validators (async or sync).

The following diagram shows a strongly-typed Typescript rule engine that allows applications to implement simple or sophisticated business rules as well as data validation. It contains a set of common rules ready for use; as well as a framework and set of classes to create custom rules for Angular applications, components, services, and library projects.

> Do not worry about the complexity of the diagram below. The diagram shows the internal design of the rule engine. The `ValidationContext` provides a simple API to add rules, execute rules, and to retrieve rule results.

[50] https://betterprogramming.pub/what-is-a-pure-function-3b4af9352f6f
[51] https://gist.github.com/wesleygrimes/f5bdf06c5c85fd3c8cb849111b5b3de2
[52] https://en.wikipedia.org/wiki/Separation_of_concerns
[53] https://en.wikipedia.org/wiki/Single_responsibility_principle



rule engine

The rule engine is using a simple design pattern called Composite. This was the first library I created using TypeScript in 2016. This was a turning point for me. I was currently using AngularJS. However the new version of Angular 2 was just released and I was interested in its capabilities. One of my concerns was that the new version of angular was using TypeScript. Imagine that was a real concern that I had. I was a little apprehensive about adding a new programming language to my current list.

My thought was that if I could create a reusable library using TypeScript then I would learn Angular 2 along with TypeScript but with a focus on creating reusable libraries for my enterprise web applications.

International JavaScript Conference (New York 2021)

Well, five years later I am still focused on reusable libraries and Angular 2 or rather just Angular. Next month I will present at the International JavaScript Conference in New York (September 27-30, 2021) on the topic of Custom Angular Libraries[54].

[54] https://javascript-conference.com/angular/custom-angular-libraries/

 [55]

Why use a rule engine?

+ Provides a consistent way to implement business and validation rules and provide a consistent mechanism to retrieve the results.
+ You can use the existing set of rules already implemented.
- AreEqual
- AreNotEqual
- GuidIsValid
- IsFalse
- IsTrue
- IsNullOrUndefined
- IsNotNullOrUndefined
- Max
- Min
- Range
- StringIsRegExMatch
- StringIsNotNullEmptyRange
- StringIsNullEmptyRange
+ You can create a reusable library of custom rules and use them in one or more applications.
+ Combine default with one or more custom rules to create a `CompositeRule` - a rule that contains other rules (rule set).
+ Each rule has a `Priority` property to execute rule sets in a specified sequence.
+ Take advantage of Typescript classes to quickly create `simple` or `composite` (nested) rules using the API that is part of the framework.
+ Use the `ValidationContext` to simply add, execute, and retrieve rule results.
+ Code faster using Fluent API style syntax - be more productive.
+ Using the `CompositeRule` base class, you can create a rule that contains other rules of either `simple` or `composite` types. The rule execution algorithm manages the complexity - now you can create rules and reuse rules to match your business logic.

ValidationContext

The ValidationContext is the container object for rules. It allows the developer to add, execute and retrieve the results of the evaluated rules.

+ Add rules by calling the `addRule()` function.
+ Execute rules by calling the `renderRules()` function.
+ Retrieve the results (a list of RuleResult items) using the `results` public property.

[55] https://javascript-conference.com/angular/custom-angular-libraries/

The following code snippet shows import statements and initialization of the `ValidationContext` as a member in the class.

```
import { ValidationContext } from '@buildmotion/rules-engine';
import { ValidationContextState } from '@buildmotion/rules-engine';
...
export class myClass {
    validationContext: ValidationContext = new ValidationContext();

    // implementation details here;
}
```

The following shows the entire `ValidationContext` class with its implementation details. It is straightforward, you make the calls in the following sequence:

1. `addRule(..)`: Add rules that you want to evaluate.
2. `renderRules()`: Renders all rules added to the ValidationContext.
3. Determine the state of the validation by using either: `hasRuleViolations()` or `isValid()`. Each returns a boolean value indicating the status of the validation context.
4. Retrieve the `ValidationContext.results` which is a array of `RuleResult` items.

ValidationContext

The `ValidationContext` is the entry point into using the rule engine. It is small but powerful.

```
export class ValidationContext implements IValidationContext {
    state: ValidationContextState = ValidationContextState.NotEvaluated;
    results: Array<RuleResult> = new Array<RuleResult>();
    rules: Array<RulePolicy> = new Array<RulePolicy>();
    source: string;

    /**
    * Use this method to add a new rule to the ValidationContext.
    */
    addRule(rule: RulePolicy) {
        if (this.source) {
            rule.source = this.source;
        }
        this.rules.push(rule);
        return this;
    }

```

```
18      /**
19      * Use this method to execute the rules added to the [ValidationContext].
20      */
21      renderRules(): ValidationContextBase {
22          this.results = new Array<RuleResult>();
23          if (this.rules && this.rules.length < 1) {
24              return this;
25          }
26          this.rules.sort(r => r.priority).forEach(r => this.results.push(r.execute())\
27  );
28          return this;
29      }
30
31      /**
32      * Use to determine if the validation context has any rule violations.
33      */
34      hasRuleViolations(): boolean {
35          var hasViolations = false;
36          if (this.rules && this.rules.filter(r => r.isValid === false)) {
37              hasViolations = true;
38          }
39          return hasViolations;
40      }
41
42      /**
43          * *Use to indicate if the validation context is valid - no rule violations.
44          * @returns {}: returns a boolean.
45          */
46      get isValid(): boolean {
47          var isRuleValid: boolean = true;
48          if (this.rules) {
49              var invalidRulesCount = this.rules.filter(r => r.isValid === false).leng\
50  th;
51              if (invalidRulesCount > 0) {
52                  isRuleValid = false;
53              }
54          }
55          return isRuleValid;
56      }
57  }
```

Adding Rules to the ValidationContext

Using an initialized [ValidationContext] object, you can add rules using a Fluent API syntax. The following example uses existing rules.

A rule requires:

+ Name: the name of the rule.
+ Message: the text to display if the rule fails.

```
  this.validationContext
      .withSource(this.actionName)
      .addRule(new rules.AreEqual('ThingsAreEqual', 'The things are not equal.', 'th\
is', 'that', false))
      .addRule(new rules.IsTrue('ThisIsTrue', 'This is not true', this.isDone, true))
      .addRule(new rules.IsTrue('Really?', 'Is it really true?', false))
      .addRule(new rules.StringIsNotNullEmptyRange('StringIsGood', 'The string is no\
t valid.', 'Hi', 3, 10));
```

Here is an example of a business action (unit of work) for adding a `contact`. It uses the `ValidationContext` to add a set of specific rules to validate the action. If all of the rules evaluate without any errors, the API is called with the contact DTO object.

> Focus on the `preValidateAction()` method. This is where the rules are implemented for adding a new contact to the application.

```
import { BusinessActionBase } from './business-action-base';
import { StringIsNotNullEmptyRange, IsNotNullOrUndefined } from '@buildmotion/rules-\
engine';
import { Severity } from '@buildmotion/logging';
// tslint:disable-next-line:nx-enforce-module-boundaries
import { ContactDto, EmailAddressFormatIsValidRule } from '@buildmotion/quicken/doma\
in/common';

/**
 * Use this action to perform business logic with validation and business rules.
 */
export class AddContactAction<T> extends BusinessActionBase<T> {
  constructor(private contact: ContactDto) {
    super('AddContactAction');
  }

```

```
  /**
   * Use the [preValidateAction] to add any business or validation rules that
   * are required to pass in order to perform the action.
   *
   * Use the [ValidationContext] item of the action to add rules. The ValidationCont\
ext
   * uses a Fluent API to allow for chained rules to be configured.
   */
  preValidateAction() {
    this.validationContext.addRule(
      new IsNotNullOrUndefined('ContactDtoIsValid', 'The contact information is not \
valid.', this.contact, this.showRuleMessages)
    );

    if (this.contact) {
      this.validationContext
        .addRule(
          new StringIsNotNullEmptyRange(
            'Address1IsValid',
            'The address information 1 is required. Cannot be greater than 60 charac\
ters.',
            this.contact.address1,
            3,
            60,
            this.showRuleMessages
          )
        )
        .addRule(
          new StringIsNotNullEmptyRange(
            'Address2IsValid',
            'The address 2 information is required. Cannot be greater than 60 charac\
ters.',
            this.contact.address2,
            0,
            60,
            this.showRuleMessages
          )
        )
        .addRule(
          new StringIsNotNullEmptyRange(
            'CityIsValid',
            'The city is required. Cannot be greater than 60 characters.',
            this.contact.city,
```

```
            1,
            60,
            this.showRuleMessages
          )
        )
        .addRule(
          new StringIsNotNullEmptyRange(
            'CompanyIsValid',
            'The company is required. Cannot be greater than 60 characters.',
            this.contact.company,
            1,
            60,
            this.showRuleMessages
          )
        )
        .addRule(
          new StringIsNotNullEmptyRange(
            'EmailAddressIsValid',
            'The email is required. Cannot be greater than 80 characters.',
            this.contact.emailAddress,
            5,
            80,
            this.showRuleMessages
          )
        )
        .addRule(
          new EmailAddressFormatIsValidRule(
            'EmailAddressFormatIsValid',
            'The email address format is not valid.',
            this.contact.emailAddress,
            this.showRuleMessages
          )
        )
        .addRule(
          new StringIsNotNullEmptyRange(
            'FirstNameIsValid',
            'The first name value is required. Cannot be greater than 45 characters.\
',
            this.contact.firstName,
            1,
            45,
            this.showRuleMessages
          )
```

```
103         )
104         .addRule(
105           new StringIsNotNullEmptyRange(
106             'LastNameIsValid',
107             'The last name value is required. Cannot be greater than 45 characters.',
108             this.contact.lastName,
109             1,
110             45,
111             this.showRuleMessages
112           )
113         )
114         .addRule(
115           new StringIsNotNullEmptyRange(
116             'PhoneIsValid',
117             'The phone value is required. Cannot be greater than 25 characters.',
118             this.contact.phone,
119             10,
120             25,
121             this.showRuleMessages
122           )
123         )
124         .addRule(
125           new StringIsNotNullEmptyRange(
126             'PostalCodeIsValid',
127             'The postal code value is required. Cannot be greater than 25 characters\
128 .',
129             this.contact.postalCode,
130             5,
131             25,
132             this.showRuleMessages
133           )
134         )
135         .addRule(
136           new StringIsNotNullEmptyRange(
137             'StateIsValid',
138             'The state value is required. Cannot be greater than 45 characters.',
139             this.contact.state,
140             2,
141             45,
142             this.showRuleMessages
143           )
144         );
145     }
```

```
146   }
147
148   /**
149    * Use the [performAction] operation to execute the target of the action's busines\
150 s logic. This
151    * will only run if the rules and validations are successful.
152    */
153   performAction() {
154     this.loggingService.log(this.actionName, Severity.Information, `Preparing to cal\
155 l API to complete action.`);
156     this.response = this.businessProvider.apiService.addContact<T>(this.contact);
157   }
158 }
```

Custom Rules

Notice that the rules for adding a new `Contact` contains an `EmailAddressFormatIsValidRule` rule.

> Custom rules inherit from either a `CompositeRule` or a `SimpleRule`. All rules inherit from the `RulePolicy` class and are executed and evaluated the same way. A `RuleResult` is returned when a simple or composite rule is evaluated.

```
1  import { CompositeRule, StringIsNotNullEmptyRange, StringIsRegExMatch } from '@build\
2  motion/rules-engine';
3  import { RuleConstants } from './rule-constants';
4
5  /**
6   * Use to validate the format of an email address. Expects:
7   *
8   * 1. string is not null or undefined
9   * 2. string length is within specified value
10  * 3. string value matches RegEx
11  *
12  *
13  * Resource: https://emailregex.com/
14  */
15 export class EmailAddressFormatIsValidRule extends CompositeRule {
16   constructor(name: string, message: string, private emailAddress: string, isDisplay\
17 able: boolean = true) {
18     super(name, message, isDisplayable);
19     this.configureRules();
20   }
```

```
21
22    configureRules() {
23      this.rules.push(
24        new StringIsNotNullEmptyRange(
25          'EmailAddressStringIsValid',
26          'The email address value is not valid. Must be within 5 and 100 characters.',
27          this.emailAddress,
28          5,
29          100,
30          true
31        )
32      );
33
34      this.rules.push(
35        new StringIsRegExMatch(
36          'EmailAddressContainsValidCharacters',
37          'The email address format is not valid.',
38          this.emailAddress,
39          RuleConstants.emailAddressFormatRegEx,
40          true
41        )
42      );
43    }
44  }
```

Rules with Rules (Composite)

> A *composite* rule is a collection of simple and composite rules.

Notice that this composite rule contains another composite rule: `StringIsRegExMatch`. Rules can be an Object-Graph of rules - a rule chain. Where each rule or group of rules (Composite) must evaluate without any errors for the rule or context to be valid. If any rule fails within the ValidationContext the `ValidationContextState.State` is `Failure`.

```
import { CompositeRule } from './CompositeRule';
import { IsNotNullOrUndefined } from './IsNotNullOrUndefined';
import { IsTrue } from './IsTrue';

/**
 * Use this rule to determine if the string value matches the specified
 * regular expression.
 */
export class StringIsRegExMatch extends CompositeRule {
  /**
   * The constructor for the [IsNotNullOrUndefined] rule.
   * @param name The name of the rule.
   * @param message The message to display when the rule is violated.
   * @param target The target that the rules are evaluated against.
   * @param isDisplayable: Indicates if the rule violation is displayable. Default v\
alue is [false].
   */
  constructor(name: string, message: string, private target: string, private express\
ion: RegExp, isDisplayable: boolean) {
    super(name, message, isDisplayable);
    this.configureRules();
  }

  /**
   * Use to configure the rules to be evaluated.
   */
  private configureRules() {
    const showRuleViolations = true;
    const doNotShowRuleViolation = false;

    // determine if the target is a valid object;
    this.rules.push(
      new IsNotNullOrUndefined('StringIsNotNullOrUndefined', 'The target value is nu\
ll or undefined.', this.target, doNotShowRuleViolation)
    );
    if (this.target) {
      this.rules.push(
        new IsTrue('StringIsRegExpMatch', 'The target value is not a match.', this.e\
xpression.test(this.target), doNotShowRuleViolation)
      );
    }
  }
}
```

Executing Rules

When you have added one or more rules to an instance of a `ValidationContext`, you are ready to execute the rules. The
`renderRules()` method will return the `ValidationContext` - each of the rules are evaluated against their specified
targets. You are now ready to evaluate the rule results.

```
this.validationContext.renderRules();
```

Evaluation Rule Results

After the rules are executed, you can examine the rule results. Each rendered rule will have a result - either valid or not valid,
based on the rule, criteria, and target value(s).

Many times it is useful to filter or extract the failed rules from the `ValidationContext`. The following code snippet shows how
you would extract failed rules that are marked as displayable (i.e., `e.rulePolicy.isDisplayable`) into a list of `ServiceMessage` items.

```
// Load the error/rule violations into the ServiceContext so that the information bu\
bbles up to the caller of the service;
this.validationContext.results.forEach( (e) => {
    if(!e.isValid && e.rulePolicy.isDisplayable) {
        let serviceMessage = new ServiceMessage(e.rulePolicy.name, e.rulePolicy.mess\
age, MessageType.Error);
        this.serviceContext.addMessage(serviceMessage);
    }
} );
```

RulePolicy

The `RulePolicy` is the base class for all rule types. The angular-rules-engine contains (2) types of rule implementations:

1. Simple
2. Composite.

These rule types form the basis of all rules in the rule engine. The rule engine uses the Composite Design Pattern - click here for more information about this pattern[56]. All rules have the following properties of information.

+ **isValid**: Use to indicate the status of the rule after evaluation.
+ **message**: Use to provide a message for failed rules.
+ **name**: Use to create a name that identifies the specified rule.
+ **priority**: Use to assign a numeric value to the rule. Rules are sorted by priority and executed in the same sort sequence.
+ **result**: The output of an executed rule. It contains the result and `RulePolicy` information.
+ **isDisplayable**: Use to indicate if the rule result is displayable to the caller. Default value is `false`. You must explicitly provide a [true] value for this when initializing a new rule.
+ **renderType**: Currently the only option is `RenderType.EvaluateAllRules`.
+ **severity**: Use to indicate the severity (Exception, Warning, or Information) if the rule evaluation is not valid.
+ **source**: Use to indicate the source or location of the rule.

The following `RulePolicy` class is the base class for all rules. Each of the default rules also extend either the `SimpleRule` or the `CompositeRule`. And each of these classes extend from the `RulePolicy` class. The allows *all* rules to have common behavior and execution strategies.

```
1  export class RulePolicy implements IRuleComponent {
2      isValid: boolean = true;
3      message: string;
4      name: string;
5      priority: number;
6      result: RuleResult;
7      isDisplayable: boolean;
8      renderType: RenderType = RenderType.EvaluateAllRules;
9      severity: Severity = Severity.Exception;
10     source: string;
11
12     constructor(name: string, message: string, isDisplayable: boolean);
13     constructor(name: string, message: string, isDisplayable: boolean = false, sever\
14 ity: Severity = Severity.Exception, priority: number = 0) {
15         this.name = name;
16         this.message = message;
17         this.isDisplayable = isDisplayable;
18         this.priority = priority;
19         this.severity = severity;
20     }
21
22     execute(): RuleResult {
```

[56] https://en.wikipedia.org/wiki/Composite_pattern

```
        console.log('Begin execution of RulePolicy: ' + this.name);
        return this.render();
    }

    /**
     * Each rule must implement this function and return a valid [RuleResult].
     */
    render(): RuleResult {
        throw new Error('Each concrete rule must implement this function and return \
a valid Result.');
    }
}
```

IRuleComponent

This interface is just infrastructure for the rule engine. It provides the contract that all rules will contain an `execute()` method - this provides a consistent mechanism to begin the process of all rules that implement this interface.

```
export interface IRuleComponent {
    execute(): RuleResult;
}
```

RuleResult

The output of an executed rule is a `RuleResult` object that contains the rule (`rulePolicy`), an indicator for the rule's state (`isValid`), and a message to be used if the rule has failed.

```
import {RulePolicy} from './index';
import {CompositeRule} from './index';

export class RuleResult {
    isValid: boolean = false;
    rulePolicy: RulePolicy;
    message: string;
    target: any;

    constructor(rulePolicy: RulePolicy, target: any);
    constructor(rulePolicy: CompositeRule);
    constructor(rulePolicy: RulePolicy, target?: any) {
        if (rulePolicy != null) {
            this.rulePolicy = rulePolicy;
            this.isValid = rulePolicy.isValid;
            this.message = rulePolicy.message;
```

```
        }
        this.target = target;
    }
}
```

Simple Rules

The main difference between `SimpleRule` and a `CompositeRule` is how they are rendered during their execution. A simple rule has a single evaluation with a single result.

```
import {RulePolicy} from './RulePolicy';

/**
 * Use this class as a base [extends] class for simple rules. A simple contains
 * a single rule and target to evaluate.
 *
 * If you require a rule that will contain more than one rule, you should
 * use extend the [CompositeRule] class.
 */
export class SimpleRule extends RulePolicy {

    /**
     * The constructor for the simple rule.
     * @param name: The name of the rule.
     * @param message: The message to display if the rule is violated.
     */
    constructor(name: string, message: string, isDisplayable: boolean) {
        super(name, message, isDisplayable);
    }
}
```

The following code is the `IsTrue` rule. This rule evaluates the target and creates a new `RuleResult` in the `render()` method. Basically, the result is based on the evaluation of the target value. This `render()` method returns a single result. This is much different from a composite rule discussed later that has to return a `RuleResult` for each rule in a list of rules.

```
export class IsTrue extends SimpleRule {
    target: boolean;

    constructor(name: string, message: string, target: boolean, isDisplayable: boole\
an = true) {
        super(name, message, isDisplayable);
        this.target = target;
    }

    render() {
        this.isValid = true;
        if (this.target === false) {//if(not true)-->false;
            this.isValid = false;
        }
        return new RuleResult(this, this.target);
    }
}
```

Composite Rules

A composite rule is a rule that contains a list of rules to be evaluated. A rule in this list can be a rule that extends from either `SimpleRule` or `CompositeRule`. This allows for a more complex implementation of rules - it is a very powerful pattern. You can have a rule that contains a list of rules, where one of those rules may be a `CompositeRule`, where one of those rules in the composite rule is a composite side-by-side with other simple and complex rules.

You are creating a rule-tree where all rules will have to evaluate to valid for the container rule to be valid. This pattern allows a developer to create new custom rules and then use those rules with the default rules to orchestrate a rule implementation against a target object or value.

The `CompositeRule` extends from `RulePolicy` which has the responsible of calling `render()` on each rule. In case of a composite rule, this method will iterate through the list of rules and call the the `execute()` method of each rule. Then the results are processed to determine if *any* of the rules failed.

```
import {RulePolicy} from './RulePolicy';
import {RuleResult} from './RuleResult';

export class CompositeRule extends RulePolicy {
    hasErrors: boolean = false;
    results: Array<RuleResult> = new Array<RuleResult>();
    rules: Array<RulePolicy> = new Array<RulePolicy>();

    constructor(name: string, message: string, isDisplayable: boolean) {
```

```
        super(name, message, isDisplayable);
    }

    render(): RuleResult {
        this.rules.sort(s => s.priority).forEach(r => this.results.push(r.execute())\
);
        return this.processResults();
    }

    public hasRules(): boolean {
        if (this.rules && this.rules.length > 0) {
            return true;
        }
        return false;
    }

    processResults(): RuleResult {
        if (this.results.filter(r => (r.isValid === false)).length > 0) {
            this.isValid = false;
            this.hasErrors = true;
        }
        return new RuleResult(this);
    }
}
```

The following shows an implementation of a composite rule. Basically, this rule is using (2) default rules, both of which are also
composite rules. All rules within each composite must evaluate to true for this rule to be valid.

```
import dCompareResult = require('typescript-dotnet-commonjs/System/CompareResult');
import CompareResult = dCompareResult.CompareResult;
import dCompare = require('typescript-dotnet-commonjs/System/Compare');
import Compare = dCompare;

import {CompositeRule} from './index';
import {RuleResult} from './RuleResult';
import {Primitive} from './index';
import {IsNotNullOrUndefined} from './index';
import {Range} from './index';

/**
 * Use this rule to validate a string target. A valid string is not null or undefine\
d; and it
```

```
 * is within the specified minimum and maximum length.
 */
export class StringIsNotNullEmptyRange extends CompositeRule {
    maxLength: number;
    minLength: number;
    target: Primitive;

    /**
     * The constructor for the [StringIsNotNullEmptyRangeRule].
     * @param name: The name of the rule.
     * @param message: The message to display when the rule is violated.
     * @param target: The target that the rule(s) will be evaluated against.
     * @param minLength: The minimum allowed length of the target value.
     * @param maxLength: The maximum allowed length of the target value.
     */
    constructor(name: string, message: string, target: Primitive, minLength: number,\
 maxLength: number, isDisplayable: boolean = false) {
        super(name, message, isDisplayable);
        this.target = target;
        this.minLength = minLength;
        this.maxLength = maxLength;

        this.configureRules();
    }

    /**
     * A helper method to configure/add rules to the validation context.
     */
    configureRules() {
        this.rules.push(new IsNotNullOrUndefined('StringIsNotNull', 'The string targ\
et is null or undefined.', this.target));
        if (this.target != null) {
            this.rules.push(new Range('TargetLengthIsWithinRange', 'The string value\
 is not within the specified range.', this.target.toString().length, this.minLength,\
 this.maxLength));
        }
    }
}
```

The `Range` rule used in the composite rule above uses (2) simple rules to form the composite. Each rule has to evaluate to true for the entire rule to be valid.

```
import dCompareResult = require('typescript-dotnet-commonjs/System/CompareResult');
import CompareResult = dCompareResult.CompareResult;
import dCompare = require('typescript-dotnet-commonjs/System/Compare');
import Compare = dCompare;

import {CompositeRule} from './index';
import {RuleResult} from './RuleResult';
import {Primitive} from './index';
import {IsNotNullOrUndefined} from './index';
import {Min} from './index';
import {Max} from './index';

/**
 * Use this rule to determine if the specified target is within the specified range \
(start and end) values.
 *
 * The range values are inclusive.
 *
 * Ex: 1 is within 1 and 3. The target is valid.
 * Ex: 2 is within 1 and 3. The target is valid.
 * Ex: 0 is not within 1 and 3. The target is not valid.
 * Ex: 4 is not within 1 and 3. The target is not valid.
 */
export class Range extends CompositeRule {
    end: number;
    start: number;
    target: Primitive;

    /**
     * Constructor for the [Range] rule.
     * @param name: The name of the rule.
     * @param message: A message to display if the rule is violated.
     * @param target: The target object that the rules will be applied to.
     * @param start: The start range value - the lowest allowed boundary value.
     * @param end: The end range value - the highest allowed boundary value.
     * @param isDisplayable: Indicates if the rule violation may be displayed or vis\
ible to the caller or client.
     */
    constructor(name: string, message: string, target: Primitive, start: number, end\
: number, isDisplayable: boolean = false) {
        super(name, message, isDisplayable);
        this.target = target;
        this.start = start;
```

```
44          this.end = end;
45          this.isDisplayable = isDisplayable;
46
47          this.rules.push(new IsNotNullOrUndefined('TargetIsNotNull', 'The target is n\
48  ull or undefined.', this.target));
49
50          if (this.target != null) {
51              this.rules.push(new Min('MinValue', 'The value must be equal to or great\
52  er than the start range value.', this.target, this.start));
53              this.rules.push(new Max('MaxValue', 'The value must be equal to or less \
54  than the end range value.', this.target, this.end));
55          }
56      }
57  }
```

Conclusion

There are lots of details in the implementation of a rule engine. However, remember that you only need to initialize a `ValidationContext`, add rules, and then call the `renderRules()` to evaluate the rule set and provide a list of `RuleResult` items. It is that simple. Happy rule rendering.

Resources

- The Composite Pattern - Design Patterns Meet the Frontend[58]
- Composite Pattern (Wikipedia)[59]
- Composite Pattern using C#[60]

[57] https://www.AngularArchitecture.com
[58] https://dev.to/coly010/the-composite-pattern-design-patterns-meet-the-frontend-445e
[59] https://en.wikipedia.org/wiki/Composite_pattern
[60] https://www.dofactory.com/net/composite-design-pattern

API: NestJS

Generate NestJS Project

> git checkout 7-1/accounts/add-api-project

- [ ] add pre-requisite packages: `yarn add -D @nrwl/nest@11.6.3`
- [ ] generate `accounts-api` project using CLI

```
nx g @nrwl/nest:application accounts-api -d
```

Add API Controller

> git checkout 7-2/accounts/add-accounts-controller

- [ ] Add module, controller, and service for the API project.

```
nx g @nrwl/nest:module accounts --project=accounts-api
nx g @nrwl/nest:controller accounts --project=accounts-api -d
nx g @nrwl/nest:service accounts --project=accounts-api -d
```

Integrate Accounts API with Application

> git checkout 7-3/accounts/integrate-accounts-api

API Debugging Tools

> git checkout 7-4/accounts/api-debugging-tools

Crete a new shared/workspace library for common types between API and application projects.

```
nx g @nrwl/workspace:library accounts/types -d
```

Fix tests and linting errors

Integrate application to use the API.

- update configuration
- update HTTP repository to use new endpoint (not mock)

Request payload should be:

```
{"emailAddress":"joe@email.com","password":"..Joe2021","passwordConfirm":"..Joe2021"\
,"acceptTermsConditions":true}
```

Response:

```
{
    "isSuccess": true,
    "message": "Successfully created new account",
    "messages": [],
    "timestamp": "2021-11-24T21:31:17.575Z",
    "data": {},
    "id": "9e2035d0-5d12-6531-082f-862b1e8c595d"
}
```

Attach to API

Launch API: https://blog.davidjs.com/2021/05/debugging-nestjs-app-in-nrwl-nx-workspace/
NodeJS debugging: https://code.visualstudio.com/docs/nodejs/nodejs-debugging

```
 {
    // debug: https://blog.davidjs.com/2021/05/debugging-nestjs-app-in-nrwl-nx-works\
pace/
    "name": "Accounts API Launch",
    "type": "pwa-node",
    "request": "launch",
    "args": [
        "apps/accounts-api/src/main.ts"
    ], // Path to main entry file
    "runtimeArgs": [
```

```
        "--require",
        "ts-node/register",
        "--require",
        "tsconfig-paths/register",
        "--experimental-modules"
    ],
    "cwd": "${workspaceRoot}",
    "trace": true,
    "restart": true,
    "internalConsoleOptions": "openOnSessionStart",
    "env": {
        "NODE_ENV": "local",
        "NODE_PORT": "3333",
        "TS_NODE_PROJECT": "apps/accounts-api/tsconfig.app.json", // Specify the tsc\
onfig to use. See content of it below.
        "IS_DEBUG_MODE": "true" // Custom env variable to detect debug mode
    },
    "sourceMaps": true,
    "console": "internalConsole",
    "outputCapture": "std",
    "resolveSourceMapLocations": [
        "${workspaceFolder}/**",
        "!**/node_modules/**" // Disable the "could not read source map" error for n\
ode_modules
    ]
    }
```