ANGULAR ARCHITECTURE PATTERNS

Apply Enterprise Principles and Patterns to Build Amazing Applications

AngularArchitecture.com

build something amazing

MATT VAUGHN

Angular Architecture Patterns

Apply Enterprise Principles and Patterns to Build Amazing Applications

Matt Vaughn

This book is for sale at http://leanpub.com/angular-architecture-the-unofficial-guide

This version was published on 2020-09-30

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

This book is dedicated to all seeking to build something better.

Table of Contents

1 Introduction

Architecture is a big topic. This book distills this down to the capabilities that we now have with Angular, Typescript, and some great patterns from enterprise applications. The book includes specific examples of a Clean Architecture and RxJS. RxJS allows a **reactive** approach to simplify the flow of data. We explore additional object-oriented capabilities of Typescript to write clean code and enhance the extensibility and testability of our applications.

Last but not least, if you ever wondered where exactly should you put your business logic, this book is for you. The implementation of business logic should be consistent and maintainable; and more importantly testable. This book provides specific guidelines for business logic, data validation, and processing business rules.

This book provides the missing guidance to build enterprise-level Angular applications using Clean Architecture principles and patterns. Simplify your code and focus on what matters by leveraging specific architectural patterns, designs, and principles.

1.1 Angular Version

The samples used in this book are using Angular 8.

```
     _                      _
    / \   _ __   __ _ _   _| | __ _ _ __
   / △ \ | '_ \ / _` | | | | |/ _` | '__|
  / ___ \| | | | (_| | |_| | | (_| | |
 /_/   \_\_| |_|\__, |\__,_|_|\__,_|_|
                |___/

Angular CLI: 8.0.1
Node: 10.16.0
OS: win32 x64
Angular:

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.800.1
@angular-devkit/core         8.0.1
@angular-devkit/schematics   8.0.1
@schematics/angular          8.0.1
@schematics/update           0.800.1
rxjs                         6.4.0
```

1.2 BETA Book

This book is a work in progress that will include updates with additional chapters. You will receive notifications of updates for free updates and downloads.

1.3 About the Author

Matt Vaughn

I build enterprise business applications. I focus on quality using a pragmatic approach to software design and implementation. I use SOLID principles and design patterns to create solutions that are testable, extensible, maintainable, and performant. For example, one solution was deployed in 4 different regions of the world and would sustain thousands of requests per second - this application continues to have 100% availability and uptime for over 10 years. Most of my career has focused on building robust frameworks and backend APIs for web and mobile applications. I now want to leverage this experience for Angular applications.

I am self-taught (but aren't we all?) when it comes to technology. I have had the privilege of working with some really smart people over the years that were also willing to share and provide guidance. I have built a few code generators, rule engines, and application frameworks using C#/.NET and now I am applying many of these tools and architectures to Angular applications.

I love solving problems, providing solutions, and being innovative. Now, I'm giving back.

It is my hope that this book can motivate, teach, encourage, and provide some guidance on your endeavor to learn. I believe applications can be built right the first time - but you need the right people, tools, and process. I hope you find the information and resources to your liking. Enjoy!

If you have a question or comments. I'm here on Twitter at @angularlicious.

- website: https://www.AngularArchitecture.com
- podcast: https://www.AngularArchitecture.com/podcast
- blog: https://medium.com/@angularlicious

1.4 GitHub.com

The source code and other resources for this book are located in GitHub:

angular-architecture

1.5 Acknowledgements

No one person achieves anything alone. I think I have a good career as a software developer. It takes a lot of sacrifice and discipline. Keeping up with technology over the last 20 years has not been easy. It feels like the “two steps forward, and one step back” saying many times. Sometimes you feel on top of the world - everything is working! And other times you question that you are a developer of any sort - how do you even have a job writing code?

Throughout the ups and downs, there has been only one individual that has been there for every moment. My wife, Wendy. I tell her all the time that she is so much smarter than me. She has so many more talents than me - I wish I could be more like her. I couldn’t have done anything without her support and all of the other amazing things she does.

I would also like to acknowledge everyone that I have worked with over the years. I really have learned something from everyone good and bad. I say that because it has been a learning experience for me continually. I am not perfect. Many times, I have created the learning experience by doing something stupid or without thinking of the consequences.

Doug Gregory is an amazing Solutions Architect with extreme dedication to the craft of software development. We had amazing whiteboard discussions back in the day. He may not know it but one night after work in the office one discussion changed my career. I learned that I didn’t have to wait for someone to tell me to do something that needed to be done or to do something that I wanted to do. Thanks Doug.

Back in 2005, I met Chris Morgan. Chris and I had the pleasure of working in one of the oldest office buildings in downtown Denver. We shared what used to be a broom closet on the top floor of the Sugar Building. We heard a continuous drip of water from the coolers on the roof. He taught me about pair-programming and other XP (Extreme Programming) principles. He also gifted me a copy of The Pragmatic Programmer book. This book and Chris’s mentorship was just what I needed at that time. Chris is a very thoughtful developer. I wonder how many individuals he has influenced.

One last person that I would like to acknowledge is Lev Vigdorov. Lev and I worked together to build an amazing suite of applications in the early 2000's. His dedication and discipline was an inspiration to me - no task was too small or too hard for Lev. He was tenacious. Lev had an unparalleled work ethic. I see that it has served him well over the years.

2 Architecture

Why do we need to understand architecture? This is an important question. We see and use architecture every day. It is all around us. It would be difficult to go anywhere without seeing architecture in place. Architecture not only defines the functionality of what something does and how it's done, but it is also in the aesthetics or how it is perceived or viewed. Architecture is not solely defined by functionality. Many times, people recognize the beauty in architecture. I think it is the combination of beauty and functionality that makes architecture so special.

Certainly, architecture is concerned with proportion, design, and the science of how things work. I find it interesting that Leonardo Da Vinci, one of the world's most famous architects, was fascinated with proportion, scale, and the science of how things work. He devoted his entire life to understanding how things work together so that they could be applied to architecture. There is definitely an artistic aspect to architecture but it also includes a scientific one as well. I think it depends on what you are building; if it has more science or more creativity. But they are both there and both rely on each other.

I think many people may think that software engineering is a scientific practice. However, when we consider how architecture is involved, we certainly need to add creativity to this scientific endeavor. In fact most of the time it may require more creativity to solve a problem than science. Therefore, we need to understand the key principles of architecture so that we can be more effective when we build our software solutions. We need to have an understanding of the functionality and how, where, and when others use it. Before we can design anything that is useful, we need to understand *why* something is needed. If we can understand *who* is going to use the application and why they are going to use it, then we have most of the answers and a good understanding of what it needs to do. It is this understanding that allows us to be more effective and to build better software.

Architecture is closely related to tangible things that are built - mostly buildings, homes, and other structures used by people in their everyday lives (i.e., shops, bridges, community and municipal buildings, houses, or skyscrapers). Today, we live in a very technology-based world. Many of the things built today require technology and the same principles founded in architecture. Let’s discuss 3 elements of effective architecture.

2.1 Quick Guide of Effective Architecture

Architecture is much more than just design. The process to create a design and a plan is just the beginning of architecture. You could have the best design and plans for a project, however, if the wrong tools and materials are chosen the outcome of the design might be failure. Also important is the execution of the design. If you only had 2 of the 3, without proper execution of the plan, the chances of success may be limited. It is the combination of all 3 together that creates *effective* architecture and provides the best potential for success.

Design and Plan

Architecture includes the **design** of something. Things could be built with only functionality in mind. There would not be anything special about these items. However, if the design was innovative or unique and had a special appeal to individuals then it is something special. Perhaps it is a combination of functional and aesthetics. It does what users want, and it also appeals to the eye. From the design, **plans** emerge as the instructions on how it works, function - but also on the specifications of how it is built.

Materials and Tools

Architecture also includes the selection of **materials**. The selection of materials depends on the purpose of their use. They may be chosen for their special characteristics. These materials may be stone, iron, wood, or anything else that provides the medium to create the solution. There would also be a corresponding set of **tools** to work with the specific material. The designers and workers would also need to know the capabilities of these tools.

Execution

Architecture also includes the **execution** of the plan. Someone can design, plan, and select the materials for a project. However, if the execution of that plan is not followed or there is no one skilled enough to work with the tools or the materials, then there is no finished product. There is nothing to admire. Just another failure.

Therefore, architecture has many facets or characteristics. There are many nuances to architecture in general. However, for our discussion of software and Angular, the main things to focus on are the following:

1. design and planning
2. tools and materials
3. execution of the plan

Do You See Architecture

Throughout a typical day in our lives think about the many things that we come into contact with. Look at the things that you *use* every day. The common things that you use without even thinking about it. You may start that ask questions like:

- who designed this?
- where did it come from?
- how was it made?
- what materials were used to make this item?
- how is its functionality organized?
- what about this item makes it appealing to use?

There are other things in life, that we rarely take notice of. It could be the home or building that we live in, the appliances in our homes; it could be the electronics and even the software that we use on a daily basis. We can ask the questions above about almost everything we come into contact with.

Therefore with these questions in mind, take time each day or throughout your day to look at something and to examine it. Try to answer the questions above about the specific item. Learn to appreciate the architecture that was involved in creating that thing. There may be a combination of Science and Technology with creativity. Think about what tools were required or what types of craftsmanship would be required to make such a thing. Lastly, think about the people involved in the actual building or making of the item. It is the execution of the design, the plan, and using the materials and tools properly that make all these things possible.

Ask yourself, “Do you see architecture?”

3 Effective Software Architecture

Effective Software Architecture doesn't have to be a buzz-word or a dream. It is possible. Being effective in anything is a process and an endeavor that requires a commitment to excellence. It isn't something that happens overnight. When individuals reach excellence and effectiveness in a given field, it is almost like they do it without thinking - by intuition. It isn't that they are not thinking, they have muscle memory (mind memory). It is this ability to do something at the highest level. When this happens you will know it and understand it. It happens in almost anything we do well. Ask someone who prepares your favorite dish, how they just did that. It may be hard for them to explain, but they just did it. The point is that it takes time and practice. Each and every day that we create software or applications we are practicing, we are getting better. Do you think about where the keys are when you are typing your code?

Effective architecture is the practice of different disciplines, processes, and adherence to principles - not rules. Therefore, if done properly, there are always ways to be more effective. Technology changes pretty fast. The things we did 20, 10, or even 5 years ago may not be effective anymore. There must be continuous evaluation and adjusting the course to make sure we are on the right path.

> Effective architecture is the practice of different disciplines, process, and adherence to principles

I have distilled Effective Architecture into 3 elements. They cannot stand on their own. All are required to work together to be *effective*. Some may appear to be more valuable on their own merits, however, for a successful result, each one is important and has a specific responsibility to play in application architecture.

Experience, Essentials, Execution

- Experience
- Essentials
- Execution

Some questions to consider that demonstrate the need for all degrees of effectiveness to be present in order for a team to win.

- Would a team win a championship without strong leadership from players, coaches, and owners?
- If the team had the best equipment and experienced leaders, would they win if they never practiced or executed the playbook?
- What if the team had excellent leadership and practiced diligently, however, their equipment, tools, and the playing field was in shambles - would they still win?
- What if a team member ran a different play than her teammates during the game - would they still win?

The following information demonstrates that there cannot be effective architecture without all elements present. If your application projects are failing, take too long to deliver, are not maintainable over time, and are not

enjoyable to work on - then you might be missing one or more elements of effective architecture.

3.1 Powerful Effect of Experience

Some say that there is no substitute for experience. The good thing about experience is that all you need is to be there and learn from the event. I'm not sure if I agree with the *length of time* definition below. I've seen some company employees have decades of time with a company - however, they do nothing different than they did 20 or 30 years ago. How much *experience* do they really have?

> *Experience should also include taking advantage of the knowledge or skills to improve or to effect a difference or positive outcome.*

Here is a common definition of experience:

> **Experience**: the process of doing and seeing things and of having things happen to you; skill or knowledge that you get by doing something; the length of time that you have spent doing something (such as a particular job).

Experience Effects Designs and Plans

When we are talking about the plan for a design and what needs to be implemented; we are talking about a specific set of steps to accomplish. The plan is not some general concept. It is a set of discreet items and precise instructions.

Many teams start development without a specific architectural plan. This means that *what* they are developing evolves as time moves forward. Over time the developers implement different items with several deviations or alternate ways of doing the same thing. Without a plan, development for even a few weeks creates technical debt. Technical debt happens fast. Developers require a specific plan with enough details to work without any questions. Ask yourself, "Would I rather do it right the first time or would I rather struggle with the effects of technical debt throughout the life of the application?"

> A failure is not always a mistake, it may simply be the best one can do under the circumstances. The real mistake is to stop trying. - B. F. Skinner

Do not stop trying to do the right thing or doing it right the first time. We have to condition ourselves to see the benefits and outcomes of doing it right. The long-term effect. Be a teammate, play for your team members.

Technical Leadership is Responsible for Design

This means that a technical lead or software architect prepares the architecture before the project begins. Many times a technical lead or architect already has designs and plans that can be leveraged or used to begin the discussion. However, depending on the application, there might be opportunities to refine the design to match the current business goals or objectives.

Without an architectural plan or design in place, it is difficult to execute the delivery of the solution. There will be a continuation of unexpected events that come up during the development process. There will be delays, excuses, late nights, and most likely weekends. There may be elements missing from features. One of the most egregious mistakes is to miss any infrastructure or cross-cutting concerns until late in the development process.

> In Real Life: Recently joined a project after 2.5 years of development. The application had many features. However, there was no implementation for handling errors, providing notifications to users, logging information, and/or errors to a central repository. These concerns had to be added retro-actively with much difficulty. Why? Because there was no plan in place when the project began.

When you think of application architecture, you need to consider design and planning. Part of the planning stage of your application involves understanding what this application needs to do. When you understand what your application needs to do you can organize each of the features into specific groups. You also need to understand the infrastructure concerns not

related to the domain of the application. These things also require thought as to the design and plan. A new house is not built without a strong foundation, framing, plumbing, electrical, or air systems in place. Many of these concerns are not seen, but they are there and play an important role. You wouldn't build a house without these concerns, so don't build software without them either.

Code Organization by Features

Code organization is a key element in effective architecture. When I say organize, I mean you need to categorize and group related things together. Angular provides several mechanisms to help us organize our code and applications. Let's talk about the Angular module. An Angular application is actually a JavaScript module. In fact, the Angular application has an `AppModule` that contains all of the other application modules, components, services and other application things like routes. We could put everything that the application is going to do into this single module. However, this would not be a good organization.

Angular provides us with everything we need in order to organize code well. What is code organization? One aspect of code organization is the ability to categorize the things in your application. You might categorize them by their relationships with other things. For example, you might group everything related to orders in the *OrdersModule*. All things that belong to shipments are grouped in the *ShipmentModule*. So, put related things together in an Angular module.

> Some say that the component is the key element in Angular. I think we need to think bigger. Think modular. Put components into modules, it is much cleaner.

Most of the time, applications have several features. It is a good idea to organize these features into their own modules. Think of modules as containers of related things. In our Angular applications, these modules can contain services, components, pipes, and directives. Therefore, if we have a feature module we can put all things related to that feature in this specific feature module. It is good to recognize that the Angular platform is built on

the base structure of a module. These modules are key elements of your Angular application. Therefore understanding how they work and how you can use them to organize your code is very important.

Components are also an organization feature. Components are more granular than a module. What goes into a component depends on the feature and the granularity of composition. The same thought process is used to define and design components within an Angular application. Group related things together. Compose application views using one or many components. There are specific patterns that relate to this topic: Container/Presentation Component patterns.

Take the time necessary to understand what the application features and capabilities are. When we understand what the application needs to do in terms of its features, then we are able to categorize and organize our code using the right container so that we can work more effectively. The exercise of designing and planning what the application is going to do should take enough time as required to fully understand the requirements. One of the habits of highly successful or effective people is to "Seek to understand, and then to be understood.". Understand what you are going to build through proper:

1. analysis
2. design
3. planning

These tasks are essential for effective architecture. It is the design, the innovation, and the creativity of going through the ***thought process*** of how you are going to do something. Great minds throughout time took the time and energy to do these things so that their ideas came to life. I have learned that this is a great team exercise. Take turns on who moderates these discussions. Work and learn together.

A Plan is Not Optional

When we speak of plans we are talking about putting together a sequence of actions that would fulfill the design. Many times there's a specific order or sequence of things that need to take place. Some of these actions or things

that need to take place have dependencies on other things. Therefore it is necessary to describe the things and their dependencies in an artifact that the team can reference.

In today's world, there are many tools to organize and plan software projects. However, I think that many of these applications manage task-based work and they are great for that purpose. However, using only a pencil and paper is sometimes the most effective way to figure something out. When we use paper and pencil we are not distracted by technology. It allows us to focus on our thoughts and ideas and to express them very quickly and efficiently.

> Alternate: use a whiteboard and sticky-notes. You can move sticky-notes easier. Take photos of the boards throughout the process. Use the photos and sticky-notes to drive further deign and even documentation. Don't get hung up on proper UML. If you can use UML and describe it, fine. However, simple diagrams with lines/arrows are usually enough to convey the ideas and capture the essence of the design.

Putting your ideas on paper is the start to the design and planning process. You can later use these artifacts to create different types of artifacts to move the process forward. What I'm trying to emphasize is that you do not need expensive or elaborate tools to do proper design and planning. It is the actual *thinking* part that is most important. Use your brain.

> Practice the art of thinking through ideas, designs, and plans.

As a consultant and employee at many different types of companies over the years, I found that many teams and individuals are so excited about building something new that they jump right in and start building parts of the application without any proper design or planning. Now if you have several smart individuals on your team the outcome might be a success. However, I have found that the lack of planning and proper design, even with smart people, has a negative effect on the delivery and quality of the solution. I think being enthusiastic or the over-enthusiasm of some

individuals disrupt the proper process of building good architecture and applications that stand the test of time.

> In Real Life: A prototype application causes problems when it is perceived as a solution that contains the implementation of proper planning, design, and execution.

Ask the Right Questions for the Right Answer

We forget that it is the finished solution that really matters. We need to deliver software that works and solves the correct problem. The right solution at the right time. Therefore, before beginning any software endeavor it is important to understand the goals of the application. It is even more important to understand and know *who* is going to be using the application.

- Why are they going to use the application?
- What problems are they trying to solve?
- When are they going to use the application?
- Where are they going to use the application?
- Who is using the application?
- Who has a stake in the success of the application?

If you have the answers and understanding to these questions, then you are ready to provide the right solution. I hope you noticed that there is no question about, how?

The Most Unimportant Question is How.

These are all important questions. I find that most developers focus on the ***how*** they are going to do something. What I see in many design and planning meetings is a bunch of developers talking about *how* they are going to provide the solution. Instead, they should be focusing on what, who, why, where, and when.

After you answer all of these important questions about your application, you have a better understanding of what it needs to do; and understand *how* you can properly build the application. Categorizing and grouping related

things together now makes sense because the domain of the application is well-understood. Now a proper design and plan is in order.

Being able to write the code and deliver the solution is the reward for going through the difficult design and planning processes. I find most developers and teams do not want to go through these exercises. They feel like their job is to write code and check it in.

> In Real Life: A developer is hired to deliver a working solution. Software that works. Software that can be delivered to a customer. Software that solves the right problem at the right time. Software that can be easily maintained over time.

A Need for Creativity and Innovation

I also find that many software developers today express the idea that following a process or a specific design stifles their creativity. This attitude is disturbing. Why? It is because we are hired and paid very well to deliver something very specific - a working solution. Although our jobs require us to be creative and innovative and to do all the necessary things to deliver the best solution. We still have to deliver the solution. Therefore it is important, no matter who you are on a development team, to follow the design, the plan, and the process as a team. The team needs to be effective and to deliver the solution for your customers and for your company.

This sentiment may not be popular by some individuals who read this. However, if we want to be effective, then we need to have proper design and planning. We also need to have team members that are willing to follow the process and to execute the exact plan. It is not effective to have one or more team members doing their own thing without regard to the team or defined architecture. This is when the excuse is made, “I did it quick and dirty, solved the problem. We can fix it later when we have more time.”. There is no later when you need to do it right the first time! These types of deviations of the plan cause more problems than they solve.

> In Real Life: What do you do with those who do not want to be on the team? Or, be a team player? If it was professional sports, this person is

benched and/or traded somewhere else.

Consistency by Design

When we think about the historical architecture achievements of mankind. There are many that stand out. Some of these may have taken decades or years to build. Think about it, if you were a worker or builder on one of these projects you likely did not see the completed solution. In fact, there were probably many individuals and groups of individuals that came and went during the building process. But it is important to note, that no matter the individual or when they were on the project, in order for the plan to be successful it required the talents of many individuals to execute that plan explicitly over a long period of time without any deviations.

Imagine if, during the construction of the great pyramids of Giza, certain individuals decided that there was a different way to build a Pyramid? So they decide to cut stones in different shapes or use different materials to finish the Pyramid. Do you think the pyramids in Egypt would still be an architectural fascination? Would you be able to notice the deviations from the original plan? This is not a crazy question.

I see in many software applications and development teams that there is deviation throughout the entire process. In fact, you could categorize the design and plan as total chaos and deviation because there is no plan being followed during the development process.

We live in a very mobile world today. Many software development teams do not have the luxury of having the same individuals or teams from start to finish. People come and go. However, if there were a proper design and plan for all developers to follow throughout the building of the software and they also followed the plan, then there is a greater potential for success. I see that when change occurs with any development team that many plans and designs are modified and changed.

Chaos by No Design

In fact, one company I recently worked at, had an application that was developed over a 2 year time period. When you look at the code, it was as if

100 or more developers worked on this application. You could see at least 3 or 4 different architectural approaches in the code. There was no consistency. Consistency, even if it is not optimal is better than code chaos. This caused many problems for developers and new team members to work on this application. It literally took days to find where to fix a defect. And the fix was mostly likely removing code from the application. If I was paid by the lines of code written, I would owe them money.

The lack of consistency and the lack of following a specific plan of execution caused this application to fail. The company was not able to recover from the lack of a plan. It was decided to stop the development of any new features and sunset the application within a year.

The Unfortunate Consultant

It is also unfortunate that many consulting companies take advantage of companies like this and promise many things, but do not deliver. These companies supply many *Warm Bodies*, many of these very intelligent. However, some of these developers or teams do not have nor want to follow a specific plan of execution. It is not their focus.

> Success is not an outcome based on the number of developers you throw at a project. Success comes from building the right things at the right time for the right users - following a well-defined design and plan without deviation. It requires technical leadership to be alert and awake to guard the code.

Many consulting companies that deliver software solutions are really focused on billing as many hours as they can. Some may invent problems and situations to encourage their clients to add even more developers to the project. This explicitly compounds the problem because you are adding more man-days, months, or years of technical debt. Many companies are unable to recover from this kind of development process. It is not effective. It is not efficient. Therefore, if you want to deliver the best solutions for your company and your clients then you need to do proper design, planning, and organization of your code. But more importantly, the execution of this plan needs monitoring throughout the entire project.

I will not name the consulting company - they know who they are and what their real objectives are. I can guarantee, that it is not to deliver the best solution to their clients. Otherwise, this major player in the online map business would not have to stop active development on one of the company's most valuable assets. Sad.

Software Development Is Not Factory Work

There appears to be a move in the industry to contract talent for specific projects. There are some business practices that make this a more attractive option than to hire full-time employees. Development managers have the thought that software development is like processing an insurance claim - within and out baskets. Some believe that building web applications, pages, components is the same as building widgets in a factory line.

> If software and web development was so easy, everyone would be doing it, right?

Application development always requires the most important asset: our thinking ability. Our innovation and creativity to create new business services and solve problems can never be written down as a set of instructions for someone to use like building a widget in a factory.

Where is the Technical Leadership?

Many development teams do not have the proper direction. What I mean by this is that there is no leader in terms of technology for the team. Yes, they may understand the business objectives and the goals of the company and the software. However, there is no one providing the guidance as to what the specific architecture is - how do you go from a concept to concrete implementation? What do developers have to guide them when actually writing the code?

It is the responsibility of technical leadership to provide direction and guidance for software developers. Therefore, I do not think that the problem solely rests with the attitudes and capabilities of software developers. It is mostly a lack of technical leadership.

If you are building a software solution, your company requires technical leadership. The technical leaders have the responsibility to architect and design systems and solutions. They should be able to document and provide the specific recipes of how their team needs to accomplish these tasks. They also need to put patterns and practices in place to enable consistency and maintainability.

Guardians of the Code

These individuals also need to guard the code. Guarding the code means that you do not let any code or parts of the application be put into the system that deviates from the plan or design. It is this chaos that causes many problems throughout the development lifecycle but also in the delivery and duration of a software application.

> There should be no excuse to not do it right the first time. We have the best tools, frameworks, libraries, and work environments since the beginning of software development. There is no excuse.

Leaders That Do Not Lead

This is not a problem solely with consulting. Recently, I worked for a private company where the manager the development team told developers that they should to write code that makes them happy. Sorry. It was more important for him to have happy employees than to deliver the right thing. This is just another example of a lack of technical leadership and the problems that it can cause. This particular application had a team of 5 or 6 individuals over one year. However, they were still not able to deliver the application. Imagine that you have 5 man-years of development and still can’t deliver a solution? This application had 5 forms, mostly name, and address type stuff. This is pathetic and is one of the main problems in the software industry - a lack of technical leadership.

> In Real Life: People show up to work and do the job they were hired to perform. Deliver. In most jobs, if you do not deliver or perform, you will soon be unemployed.

Effective architecture requires strong technical leadership. These leaders must use their experience to guide their teams. Create happy teams by making it easy for them to win and deliver effective solutions.

3.2 Essentials of the Plan

- Materials
- Tools

Material Selection is an Architectural Decision

When we think of the great architectural feats of mankind throughout history, Many times what is extraordinary about the solution are the materials. What also makes the architecture incredible is the tools that were used along with the materials. Many times the materials were selected because of their special characteristics. Perhaps it is their beauty, or how long the materials will last and hold up over time. Whatever the reasons are for selecting the specific materials, it is evident that the selection of materials was an architectural decision.

Tools Specific to the Materials

We are now going to talk about the importance of tools and materials in architecture. Let's discuss the importance of selecting the right materials and tools but more importantly how understanding their capabilities and characteristics may impact the success of a project.

> Pro Tip: Consider the selection of tools and materials early. They can and will influence the design and planning of the application.

Many times, builders used special tools to work with the selected materials. In some cases, these tools were invented for use with the selected materials. To be effective with the materials, the builders had to have mastery over the tools. If the architects or the builders did not understand the capabilities of the tools, then how would they know how to effectively work with materials? Therefore, it is reasonable to conclude that knowledge of the tools and their capabilities along with the materials and their characteristics go hand-in-hand. The designers of such structures had to completely understand the materials as well as the tools that were going to be used.

Today, it is the same. when we set out to build a new application. The selection of materials and tools has to work with a specific plan from the designers. Many times, development teams do not select the materials or frameworks for the project. The company may be using specific technology stacks in order to manage and maintain consistency in their technical environments. Regardless, even if the main technology stack is already chosen, there may be opportunities when new materials and/or tools need to be introduced.

However, if you are starting a new project or a new need emerges, then it is necessary to wisely choose which types of materials and tools would be well-suited for the specific solution or project. The specific goals and objectives of the project may dictate or provide additional requirements during the selection process.

It is vital to the success of a project, to select the appropriate and correct tools and materials. It is also just as important for all the users of those materials and tools to understand their capabilities and their characteristics and to use them in the right way.

Tools and Materials: Build or Buy?

One of the things to consider during the material selection process is to determine whether you need to build or buy. Many times, the off-the-shelf solution provides just enough or more of the features that you need. In this case, you do not need to build tools. You can purchase them and they will be ready to use for the specific need. I have seen many teams or individuals within the team that want to build their own tools or materials. Since some teams have the capabilities and the resources as well as the skills to do such a thing, the question is, “Is it really necessary to do so?

> Do not turn unnecessary things into science projects. Ask yourself, “Do you feel lucky?”

Think about the scope of the problem. If building the tool or materials is really not necessary or part of your business goal or objective, then you probably should not build it. You should select a solution or purchase a

product that solves that specific need. I have seen teams waste valuable time building tools and materials that could have been purchased. If your project has a specified delivery date, you need to make sure that your delivery date also allows for you to build the tools and materials that you need for your project. Otherwise, you may not deliver on time and/or deliver the wrong thing.

> In Real Life: Spending money on purchasing tools and materials is many times less expensive than the cost to build a custom solution and maintain it over time. There is that phrase: Pennywise, but pound foolish.

Wrong Tool Horror Stories

About one year ago, I needed to find a new job. The company that I was working for was undergoing some severe financial problems. During the previous 2 years, our team had built many new services and applications to support new business opportunities. However, doing these things was a little too late. For example, I learned that 2 members of the team that had been with the company for many years, used valuable company resources to build things that they could have got for free. They built an object-relational mapping tool or ORM, instead of using one of the many suitable enterprise-level solutions available at the time. This little endeavor cost the company about 2 man-years of development. I am sure that there were many other projects just like this during the years before I joined the team.

Here is another example of selecting the wrong tool. A consulting company in Denver had the task of rebuilding an application for a client. they could either choose AngularJS or the new version of Angular. Let us fast forward a few years, The current application was in such a state that it could not be refactored or salvaged. Therefore, the client has decided to not do any new development and to sunset the application. It is sad to note that this application was once the main provider of specialized information on the internet 20 years ago. This 2-year project, or rather supplying warm bodies to a client, lacked technical leadership. The company assumed they were getting leadership. What they got was about 10 man-years of code that couldn’t run all of the tests (most turned off due to lack of maintenance),

contained many different patterns and developer one-offs and lost the status of being the company's flagship product.

> Question: What would be the result if the right tools and materials were used by capable developers with proper leadership during this time?

Selecting the Right Tools and Materials

Many times selecting the right tools and materials can make all the difference in a successful project versus one that is riddled with problems or that results in failure. There are many things to consider when making a selection of tools and materials. Some of the considerations may be the skills, assets, or resources that you have available on your team. For example, one company I worked at was mainly a Microsoft development shop. Many of the team members had years of experience using C#. That was just one consideration in choosing the latest version of the Angular framework - with Typescript. The capabilities of Typescript along with the knowledge and experience of our .NET developers made it an excellent choice for our team.

Sometimes there are non-technical factors to consider. For example, what is the community like behind the potential framework? What are the available training resources for the specified framework? Is there good support from the community for this framework? What is the likelihood that this framework lasts for many years?

Today, you can build web applications entirely made up of Open Source projects and packages. Therefore, if you do not have a budget or money to spend you still have an opportunity to build a successful project. Many times cost and licensing fees are a factor in the choice of tools. If you are selecting a tool or materials from a third-party provider, make sure that the product has good support and product updates, along with an active community supporting the product. There are situations that a purchased product has a lower lifetime cost due to its reliability and support.

I have found that there are many free resources available to web developers. Many of these tools and materials provide more than adequate tools, sometimes even better than third party providers. It is a good time to be a web developer.

Understand the Capabilities

Another process to go through to help in the selection of tools and materials is to understand the capabilities of these tools and materials. In order to do this, you need to understand what it is that you are actually building or need to build. Consider the goals and objectives of the application. When you have this information, you can look at the capabilities of the tools and materials to determine if they meet or exceed expectations.

For example, if your application requires the ability to use object-oriented programming techniques or design patterns, you need to select a programming language that supports these capabilities. An emerging feature in many web environments is the ability for *type safety*. If this is your requirement, then you need to consider only languages that support strong types.

Think about the depth and breadth of the features of any framework candidate. If the framework does not have a feature that is a core requirement of your application, then you most likely have to come up with a different solution. There are additional costs and risks associated with building your own tools and materials. I find it useful to create a list of the required features (weight them by importance to the team). Then create a comparison matrix of the framework candidates. Give each framework a score for each comparison item using the weighted factor. Some candidates may have deficiencies where others do not, however, their overall score may indicate the best choice.

> Math doesn't lie - at least most of the time, right?

I have seen teams or team members be very emotional about the selection of a tool or framework. It is best to be objective about the decision. Focus on the desired outcomes and the characteristics during the selection process.

3.3 Execute the Plan

Alright, now we have a great *design* and *plan* along with the selection of great materials and tools. What do we do next? Remember that there are 3 requirements for effective architecture. You need to have the experience to design and plan. You require the experience to create the recipes for the developers to follow. Additionally, we require *essentials*. The essential tools and materials to implement the design. Last but not least, we require proper ***execution***.

Know Execution, Know Success

The execution portion of the architecture is the most important. You may have a great plan and design, along with great tools and materials. However, you will have limited or no success if the execution fails.

In order to execute a plan, there needs to be a plan. If the team of developers does not have a plan or a set of recipes to implement the designs of the application, the project will contain chaos and deviation throughout the project. Without guidance, each developer ultimately decides for themselves how to implement the feature. It is possible to have the same number of deviations of days in a week *x* number of developers. One of the main characteristics of incredible architecture and technology is that is it is consistent in terms of how it was built - there is little to no deviation in the workmanship.

> Is it possible that 20% of development by the team is causing some sort of technical debt? Consider the amount of debt by a team working on a project for 6 months. That is 120 man-days of *technical debt*. If it was only 10% of development, it is still 60 man-days of technical debt. How do you recover from that?
>
> const techDebt = (developerCount * daysOfDevelopment)/.20

Software Development is a Team Sport

For just a moment let us think about a professional sports team. Perhaps a sports team that has won multiple world championships. How do they accomplish this year after year? It is reasonable to conclude that these teams have a system; a playbook or a scheme that they follow. The coaches and the players most likely study and learn the system so that they can be effective during practices, but more importantly during the game against their opponents.

Most likely, it took much effort to design and create a playbook of this sort. It is reasonable to conclude that the players would have a disregard for the playbook and choose to do something different when a specific play was called.

> Play as a team, win as a team. Sacrifice and commitment are not optional.

Software development is a team sport. There is no room for deviation and chaos during the implementation or execution phase of a project. Most teams have individuals or roles that are responsible for creating or executing the playbook. In software development, we require the same roles.

There has to be somebody performing the architectural role on a team. Who is doing the design and planning of the application? If the team does not have this type of resource available, then, as a team, they need to do this together so that they at least have a plan that they can follow.

- Is it documented?
- Does everyone know what the playbook is?
- Does everyone understand the architecture, design, and plans?
- Are there examples and/or recipes to implement the designs?
- Are there specific patterns to use and follow?
    - Does everyone understand them and use them?
- Can each team member articulate the play or architecture?
    - If you cannot explain it, how do you know you really understand it?

During the execution phase, we need to ensure code quality. This is most likely a team lead or a Tech lead that has the responsibility of ensuring the quality of code that is committed to the repository - but it should be every member of the team. This individual has the responsibility to make sure that the correct patterns or recipes are being used to implement the code. I think of this as a play that has been called by a coach during a game. Somebody needs to make sure the play is executed. If we watch sports, it is evident that there are individuals performing this role. It should be everyone's job.

There are other ways to help and ensure code quality. I liked to have informal and formal code reviews where individuals can review the code with the developer who created it. I have seen many teams go through the motions of a *pull request* or *merge request* process, and think that this is a code review. This is not a code review. You need to see working code, a walk-through the code in debug mode, and a set of passing tests exercising the feature. It also involves communication with the developer - asking questions to understand the design decisions or motivations.

This takes time and effort.

> In Real Life: Does it take time and effort for a professional athlete to train, study, practice, and work with the team? Yes, it does. I guess that is why they call it work. Athletes get paid well, as do developers.

I have been on teams where you can open up different files within an application and not be able to tell which developer worked on that code. This is a team-first approach. A team member should be thinking about her teammates that will maintain and support this code in the future. This is certainly a level of professionalism that should exist on all teams today. Each member of the team needs to be accountable and responsible for following the playbook and to run the correct play. Do not wish for quality code, *make* it happen.

> Guardians of the code - is everyone's responsibility.

With the number of tools and resources that we have today, there is no excuse to not produce the highest level of code quality than ever before. There should be no excuses like, “Let’s come back to this code later when we have more time.” There should be no excuses like, “I didn’t have enough requirements or understand the requirements.” We all have the ability to communicate and to ask questions when we do not understand something. Most of these issues are not technical issues they are people or communication issues. People failing to communicate or to follow a process or a playbook or a defined way of doing something. If the application is not working properly, has unanticipated defects, or it is taking way too long to deliver - that is most likely on the entire team, management, and lack of leadership, planning, and communication.

It is reasonable to conclude that new patterns and ways of doing things emerge during the development process. However, they need to be handled in a way that maintains and keeps the integrity of code, as well as the quality. There is a time and place to introduce new concepts or ways of doing things.

Code Chaos is the Enemy

During a recent code of you, I saw a single *class* that had 5 different methods to retrieve data from an API/HTTP call. Each of the 5 methods had a different way of handling the response and any error messages returned by the API. I am going to assume that the same developer implemented each of the 5 methods. So the question is, why are they all different? There is no need to have this type of deviation in a single file - now consider the deviation throughout the entire code base. It makes it more difficult to understand and to read. Code that is difficult to understand is also difficult to maintain over time. Who enjoys working in a codebase like this?

> In Real Life: After 2 plus years of development, the application still requires proper error handling and logging. If you only want to handle error messages from the backend API, there are over 200 places in the code that need to be refactored to do this! There was no plan for error handling in the application? Many of them just eaten by the code and maybe there is a `console.log`. Completely unacceptable.

My current project involves the internationalization of the entire application. This includes information that comes from the API and the client application. All of the static content in the application required *i18N* internationalization attribute tags in the HTML. After 2 plus years of development, the team did not have a process in place to translate the content or a workflow to manage new or changes to content. There were many issues surrounding the formatting of the HTML and translating messages from the back end. If there was a specific plan in place from the beginning for internationalization as well as guarding the code for code quality, then we would have eliminated a lot of work during the last several weeks.

Obviously, there has been a lack of technical leadership or a lack of a plan from the beginning. This causes delays when you are trying to release the new application to customers. Things would be much different if the original plan or a plan existed with a following:

- error handling
- logging (centralized)
- user notifications
- proper data validation
- proper business rule implementation
- a single mechanism for HTTP requests
    - single pattern
- proper code organization
- i18n (internationalization)
- proper use of *NgRx*

Developer Workflow

> Execution involves more than code that just compiles and performs the anticipated behavior.

During the execution of a plan, there are most likely many processes or tasks that fall into the category of repetitious. How many hours would be spent if each developer had to perform these tasks manually throughout the

development effort? Some of these activities are not directly related to feature development. Does this make them less important? Here is a list of things to think about?

- Source Control
  - Feature, BugFix, Chore branches
  - GitFlow
  - Frequency of Commits
  - Frequency of code pushes
  - Frequency of integration with *develop* branch
- Building Application
  - create scripts
- Running Unit Tests
  - scripts
- Formatting Code
  - use tools to configure code by type(HTML, JSON, TypeScript, CSS)
  - pre-commit hooks to format code before push to repository
- Continuous Integration
  - verify `aot` and `prod` build
  - verify lint rules
  - verify unit tests
  - package and deploy
- Continuous Deployment
  - specific environments (test, stage)

A technical lead and/or team should be aware of these concerns before and during the execution phase of the project. Effectiveness requires efficiency at scale. If we can automate builds, testing, deployments, linting, code formatting, and any other processes it allows us to focus on the specific solution.

4 Angular Toolbox

4.1 What Does Angular Give Us

Angular had a confusing beginning. This is due to the original beginnings of AngularJS - often referred to as 1.x versions. Angular 2.0 was announced in October of 2014. There were many changes. In fact, the only thing Angular 2.0 and 1.x versions shared was the name *Angular* - unfortunate. However, it was the next evolution in the framework. I think it took courage to do a complete rewrite of the framework from the ground up, leveraging the lessons learned from the previous version.

Version 2

Angular 2 was just the beginning of an amazing framework and community. Recently, I was invited to be a mentor at an ngGirls.org event in Denver, Colorado. My student and I were talking about web frameworks and comparing some of the features and capabilities. There are many great frameworks out there - and each individual and team needs to weigh many factors to determine which one suits them the best. I mentioned that Angular is like buying a home that is fully furnished with everything you need to move in. This home has all the rooms finished, including air conditioning, heat, hot/cold water, entertainment system, network, video monitoring, landscaping, etc.

Other frameworks, maybe provide only a limited set of features for building enterprise applications. A comparison home, in this case, maybe a home that is structurally sound - however, there is only a rough-frame of the home. You need to finish the home by adding the remaining features that make it livable. This approach works if you have the skills, resources, and time to bring in the other parts.

In 2016 I was asked to select a framework for a new set of enterprise web applications, some of my decision points were:

- community
- technical support and documentation
- number of users
- training resources

- capabilities of the framework
- capabilities of language (Typescript)

  Version 2 Release Facts:

  - Beta: December 2015
  - Release Candidate: May 2016
  - Final Version: September 2016

Angular provides a rich *component-based* system where code and application elements are organized in *Modules*. Since the original release of version 2, we have a rich development environment that is often referred to now as a ***platform*** rather than just a framework. Here are some of the capabilities.

- Typescript
- Components
- Modules
- Services
- Dependency Injection
- Environment Configuration
- Router
  - Router lifecycle, Guards and Resolvers.
- HttpClient
  - Http Interceptors
- Directives
- Pipes
- RxJS driven
  - Reactive Forms
- Support for Progressive Web Apps and Material Design
- Angular CLI with updated Schematics
- CDK Components
- CLI Workspaces
- Library support for custom library development, sharing, and publishing
- Animations

- Dynamic imports for lazy routes
- Build and performance enhancements
- Support for internationalization (i18n) and localization (l10n)

I thought the capabilities were impressive in 2016. However, in just a few years, Angular has changed the way modern web applications are developed. I feel the most important capabilities related to software architecture is *code organization*.

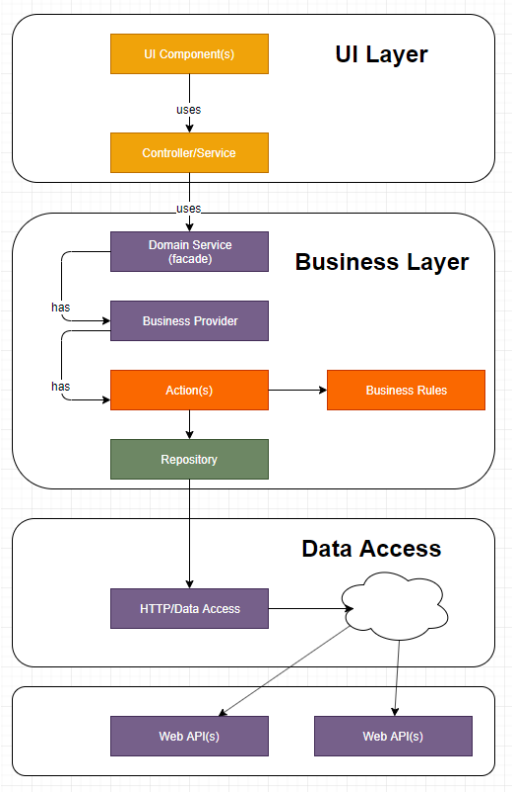
Code Organization

There is no argument that Angular is an opinionated framework. Being opinionated is not bad - especially if they are good opinions based on standard practices and patterns. The Angular environment provides all of the elements we need to organize our code - that enables the implementation of enterprise patterns. Many of the established architecture patterns focus on the principles:

- Separation of Concerns
- Single Responsibility
- Inversion of Control
- Use of Interfaces and Abstractions for boundaries

Application operations flow up and down a specific stack or layers. Many times, these layers are sliced vertically to group related items into specific domain features. Imagine we are building a Learning Management System (LMS). We would have the verticals for:

- Students
- Instructors
- Videos
- Courses



We have exactly what we need to organize our code using very standard enterprise patterns and principles. Basically, we are creating horizontal layers and verticals to represent domain features. We use the capabilities of Angular and Typescript to implement the boundaries (e.g., separation of concerns, single responsibility, etc.).

Let’s face the facts. Angular gives us a lot. In a few short years, the

- Progressive Web Applications
- Native mobile applications using Cordova, Ionic, or NativeScript

4.2 What's in the Angular Toolbox?

The Angular toolbox is more like one of the big box home improvement stores. Imagine if you could walk into one of those stores and select everything you need for your next project. You can select materials, tools, and anything else you need. Oh, and by the way there is no cost. It is all free. That probably never happens in the home improvement business. However, things are much different in the technical community.

Many tools and frameworks to build web applications are available at no cost. However, even when frameworks and tools are free to developers it does not mean that there was no cost in the creation of these things. If you had to purchase Visual Studio Code, how much would you pay for it? If the Angular framework was a commercial product, how much would you be willing to pay for it? Interesting questions.

If you want to be an effective developer, you need to know and understand the capabilities of your tools. Just having the tools in your garage does not mean that you are a good car mechanic. We can download and install tools and frameworks all day long. It is much different to use these tools and frameworks skillfully to provide effective technical solutions. Therefore, you need to get familiar with:

- Visual Studio Code
  - Documentation
  - Extensions
  - Getting Started Videos

Visual Studio Code

I recommend using Visual Studio Code as the main development tool for developing Angular applications. These are just a few features that I use almost on a daily basis:

- use the CLI at a terminal
  - to create workspaces
  - add modules, components, and services to applications

    - creating custom libraries
- build and serve the applications
- test applications
- view my code and other files using an outline
- use Git to manage source control repositories and branches
    - view and compare changes and versions of files
- integrate an amazing set of extensions for
    - code formatting
    - IntelliSense
    - spelling
    - manage imports of modules and other dependencies
- Live Share features
- Debugging and attaching to node applications
    - Watches
    - Code Stack
- code refactoring
- View and navigate to the usage of classes, methods, properties, and other class members

Angular CLI: Command-line Power Tools

Code Generator and Terminator: Schematics

4.3 Typescript

Object-Oriented Programming

Design Patterns Still Relevant

4.4 Angular Platform

Is Dependency Injection Necessary?

Building Blocks of Angular Applications

Modules

Components

Services

Routing with No Traffic Jams

HTTP

4.5 Code Organization Containers

Workspace

Applications

Libraries

Modules

Components

Services

5 Workspace: Where You Work Matters

Most everyone agrees that a nice workspace is not only enjoyable but essential to being effective. Angular gave us an impressive gift with version 6. This chapter provides an overview of the game-changing work environment for developing web applications. For years, other development environments (i.e., .NET, Java, etc.) have had rich IDEs that provide environments to create solutions with multiple projects of different types.

Programmers require the ability to organize code and projects related to each other. Today, the requirements for web applications require the same capabilities.

Web developers typically have a folder and the entire codebase for the project is in that folder. There is no concept of sharing and reusing libraries in a solution (or workspace). If you want to create a JavaScript library, it is typically developed as a separate project, published to a package manager, and then consumed by other JavaScript applications. This is a workflow that has many steps - not a very efficient or effective workflow for rapid development. I know, because my team did this for Angular 2 applications up through version 5 for about 11 or 12 projects. Some of our packages had dependencies on other custom libraries. A small change in one library would require us to build and deploy several other libraries due to our dependencies. This was very time-consuming.

So, what if there was a work environment that enabled you to manage multiple web application projects, create libraries, and share/reuse those libraries with any of the web applications and/or the other libraries without the overhead of publishing and consuming packages via npm? It was quite a surprise when Angular 6 was released with support for a Workspace environment that does all of these things.

The Angular Workspace at first appearance may not seem like much. The ability to add multiple projects is significant. However, the new supported

project type for library projects may provide the biggest impact. It allows us to build modern web applications with new architectural patterns that simplify and make our code more powerful.

> This book will focus on the changes the new Angular Workspace provides to implement strong enterprise-level architectures.

The Angular Workspace simplifies the management of multi-project environments. All projects now share the same package dependency configuration, the same Angular version, a single configuration for building, serving, linting, and testing projects. We got all of this with Version 6 of Angular!

> We can have great web application architecture that is clean without the Angular Workspace. However, the additional CLI tooling along with the ability to share and reuse code via libraries is the biggest game-changer to web development workflows and work environments. Individuals and teams that leverage these capabilities will certainly be more efficient and effective - this means a lot in the competitive software industry.

5.1 May the 4th Be With You

On, May 4th, 2018, Angular 6 was released. The Angular.io site actually says May 3rd. However, it appears that the press releases were pushed the following day. And I like the sound and reference to *using the force* with the date that most recognize as the public release date.

5.2 What Is So Special About Angular version 6?

The Angular CLI command to create a new project was dramatically upgraded. This command no longer creates a project. It creates a ***Workspace***.

```
1 ng new <my-workspace-name-here>
```

Most developers didn't notice much. The command above still worked - there were no breaking changes. I think there should have been because the change was so significant - I think the new capability flew under the radar for a while. This command created a new *Workspace* with a default application project where both had the same name. In fact, the default application project was in the same location as before. There was no indication that anything different was going on. Yes, there was a new *angular.json* file.

If you inspected this file, you noticed that there was a *projects* node with the default project containing a project type of *application*. Interesting.

```
1 "projectType": "application"
```

> Note: Angular version 7, allows for an *empty* workspace with the `ng new` command. You would use the `ng generate application <name>` to add a new application to the workspace.

5.3 Monorepo with Angular Workspace

We just got a monorepo. A single environment where we can now develop multiple projects. Not just single-page web applications, but also *library* projects (more on this later).

Is this a good thing? It is for several reasons. Think about all of the time that could be saved without the overhead of configuring development environments. A monorepo allows a team to continually improve the workspace environment - and everyone benefits. It is like a community garden that continually gets better and always produces fine results.

One of the main goals of Angular is to enable companies, teams, and individual developers to be as effective as possible. Effectiveness also includes efficiency and building the right things at the right time. Therefore, to enable and provide the necessary environment to develop effective Angular solutions, we require the implementation of a single *Workspace* for all things Angular. This includes:

- all Angular applications
- all Angular and TypeScript libraries (cross-cutting concerns, domain libraries, core, and foundational libraries)
- all tooling (Schematics)
- documentation
- linting
- testing
- build

This environment allows a team to establish, monitor, and improve patterns and practices as a collective. This does not mean the removal of specialization and/or creativity within the team or individual developers - however, it provides increased transparency, knowledge sharing, workflow, and collaboration for the many in a single work environment.

A good example for my current team is code formatting. This may seem like a trivial thing. However, HTML formatting became very important in regard to internationalization (i18n). We were able to use tools like Prettier

- with a single configuration that applies to all applications and libraries in our Workspace. We also added a pre-commit hook to format our Typescript files before commits to our repository. This level of consistency and maintainability is very impressive.

5.4 Angular Workspace Improves Developer Efficiency

When you are working on any project whether it is software, home, automobile, or other hobbies; the setup and breakdown of your work environment takes a lot of time.

- getting the tools together in a single location
- clearing out a space for work
- retrieving and putting things away
- cleaning up after the work

Although important, the setup and breakdown of a project are considered waste or non-productive time, because you are not building features. To solve this problem, a monorepo allows you to focus on a specific project - not setting up an environment for each new project.

There is a lot of overhead and maintenance of a project beyond the actual domain-specific implementation. Before Angular 6, if you had 3, 5, or 10 Angular applications, you had as many Angular projects. Depending on when they were created, they may vary with different versions of Angular, 3rd-party packages, or Typescript. Keeping all of your projects in-sync with version updates is/was a time-consuming process. I have seen older projects just left alone without any upgrades.

The new *Angular Workspace* solves this problem. The workspace is a single work environment where all projects share the same runtime and development dependencies (packages). They all use the same version of dependencies and they all share the same *package.json* version. When you upgrade the workspace, all projects are updated at the same time. If there are any problems or issues, you have early notification during the build of your applications.

> No project left behind.

Instead of setting up and managing each project independently, you have a single source of truth and a place to manage all of your Angular projects.

If you develop custom Angular libraries and publish them as packages, you doing a very noble thing. Sharing and reusing code is very effective. However, the process to build, publish, and consume *library* updates is not a very *efficient* process, right? There are many steps. There are even more steps if you have libraries that have dependencies on other custom libraries. The Angular Workspace solves this problem with the new project type: ***library***.

For more information on this topic, please read my blog post: Monorepo + Angular Packaged Libs :: You Can Have Your Cake and Eat it Too!!.

6 Workspace: A Container for Projects

The Angular Workspace is an environment to manage multiple projects. The `angular.json` file contains the configuration for each project in the `projects` node of the file. There are currently (2) supported project types.

1. application
2. library

Our reference workspace for this book contains a project named `lms` (learning management system). Using the CLI command creates the application project and files in the Workspace. The CLI updates the `angular.json` by adding the project definitions to the `projects` node.

```
1 ng generate application lms
```

6.1 Angular.json

Configuration for any application is a big topic. If you had 10 applications and 10 different teams, each application would most likely have a different configuration and quite possibly many variations or deviations depending on your perspective. What if there was a way to have a conventional and reliable configuration for your projects where there was no deviation between projects, teams, or even companies?

The `angular.json` configuration and CLI provide the answer. We do not need to spend time coming up with a configuration for building, testing, linting, serving, and even localizing (l10n/i18n) our applications. It is already done for us - focus on building something amazing.

The Angular.io website contains a nice overview of the Angular Workspace Configuration. The following properties, at the top level of the file, configure the workspace.

- *version*: The configuration-file version.
- *newProjectRoot*: Path where new projects are created. Absolute or relative to the workspace folder.
- *defaultProject*: Default project name to use in commands, where not provided as an argument. When you use ng new to create a new app in a new workspace, that app is the default project for the workspace until you change it here.
- *schematics* : A set of schematics that customize the ng generate sub-command option defaults for this workspace. See Generation schematics below.
- *projects* : Contains a subsection for each project (library or application) in the workspace, with the per-project configuration options.

There is another project type that doesn't fit into the category of either application or library. There is a **Schematic** project type that is used by the CLI and provide utility-like features to generate, update, and remove items from specific projects and Workspace environments.

6.2 Applications

Application projects are where most developers spend their time. These projects require a runtime element, are hosted, and typically run in a browser. The runtime elements of an Angular application are HTML, JavaScript, and CSS. How we get there is another story. There are (2) levels of configuration for an application project.

1. *angular.json*: provides a Workspace-level configuration for running builders, linters, i18n, and testers for the application. You can also define specific configurations that match to target environments.
2. *root project folder*: The application root folder contains configuration files to provide instructions for TypeScript compilation, Linting, Testing, and browser list targets.

Each Angular application has a conventional file structure.

> Learn about the application file structure and configuration files at: https://angular.io/guide/file-structure

Here is a partial sample of a web application configured in an `angular.json` file. Using JSON allows for a schema definition - a reliable configuration definition.

> Configure at will, but stay within the lines of the schema.

```
  "projects": {
    "lms": {
      "projectType": "application",
      "schematics": {
        "@nrwl/workspace:component": {
          "style": "scss"
        }
      },
      "root": "apps/lms",
      "sourceRoot": "apps/lms/src",
      "prefix": "lms",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
```

```
            "outputPath": "dist/apps/lms",
            "index": "apps/lms/src/index.html",
            "main": "apps/lms/src/main.ts",
            "polyfills": "apps/lms/src/polyfills.ts",
            "tsConfig": "apps/lms/tsconfig.app.json",
            "assets": ["apps/lms/src/favicon.ico", "apps/lms/src/assets"],
            "styles": [
              "apps/lms/src/styles.scss",
              "apps/lms/src/assets/demo/demo.css"
            ],
            "scripts": [
              "apps/lms/src/assets/js/core/popper.min.js",
              "apps/lms/src/assets/js/core/bootstrap.min.js",
              "apps/lms/src/assets/js/blk-design-system.min.js"
            ]
          },
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "apps/lms/src/environments/environment.ts",
                  "with": "apps/lms/src/environments/environment.prod.ts"
                }
              ],
              "optimization": true,
              "outputHashing": "all",
              "sourceMap": false,
              "extractCss": true,
              "namedChunks": false,
              "aot": true,
              "extractLicenses": true,
              "vendorChunk": false,
              "buildOptimizer": true,
              "budgets": [
                {
                  "type": "initial",
                  "maximumWarning": "2mb",
                  "maximumError": "5mb"
                }
              ]
            }
          }
        },
        "serve": {...},
        "extract-i18n": {...},
        "lint": {...},
        "test": {...}
    }
  }
```

Application Configuration (angular.json)

Angular application Workspace configuration is managed in the `angular.json` file. The following application assets in the `architect|build|options` node of the project configuration.

- assets: use to set the application `*.ico` and the target `assets` folder
- styles: use to define styles for the application
- scripts: use to define any required JavaScript files used by other packages

Application Module (AppModule)

An Angular application is based on modules - the application is bootstrapped using a default `AppModule`. This module references other required modules along with a set of routes. In the sample application files listed below, there are (4) modules in addition to the *AppModule*.

- CoursesModule (feature)
- CoreModule (core)
- SharedModule (shared)
- SiteModule (site)

```
app
├ features
│ ├ courses
│ │ ├ latest-courses
│ │ │ ├ latest-courses.component.css
│ │ │ ├ latest-courses.component.html
│ │ │ ├ latest-courses.component.spec.ts
│ │ │ └ latest-courses.component.ts
│ │ ├ courses-routing.module.ts
│ │ └ courses.module.ts
├ modules
│ ├ core
│ │ └ core.module.ts
│ └ shared
│ │ └ shared.module.ts
├ site
│ ├ footer
│ │ ├ footer.component.css
│ │ ├ footer.component.html
│ │ ├ footer.component.spec.ts
│ │ └ footer.component.ts
│ ├ navbar
│ │ ├ navbar.component.css
│ │ ├ navbar.component.html
│ │ ├ navbar.component.spec.ts
│ │ └ navbar.component.ts
│ ├ sidebar
│ │ ├ sidebar.component.css
│ │ ├ sidebar.component.html
│ │ ├ sidebar.component.spec.ts
│ │ └ sidebar.component.ts
│ └ site.module.ts
```

```
├ app-routing.module.ts
├ app.component.html
├ app.component.scss
├ app.component.spec.ts
├ app.component.ts
└ app.module.ts
```

Application Module

Each Angular application has an `AppModule` responsible for bootstrapping the application. A module is a container to group related items together. It is an organizational structure. The concern for the application module is the entry-point for the application. It is responsible for:

- bootstrapping the application
- loading other modules
- providing global (application-level) services

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { AppComponent } from "./app.component";
import { SiteModule } from "./site/site.module";
import { AppRoutingModule } from "./app-routing.module";
import { SharedModule } from "./modules/shared/shared.module";

@NgModule({
  declarations: [AppComponent],
  imports: [AppRoutingModule, BrowserModule, SharedModule, SiteModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

> Learn more about Angular modules at:
> https://angular.io/guide/ngmodules

Shared Module

We can use the Angular CLI to generate additional modules in our applications. The concern of the `SharedModule` is to manage Angular and other 3rd-party modules for the application.

Creating shared modules allows you to organize and streamline your code. You can put commonly used directives, pipes, and components into one module and then import just that module wherever you need it in other parts of your app.

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { RouterModule, Router } from "@angular/router";

@NgModule({
  declarations: [],
  imports: [CommonModule, RouterModule],
  exports: [RouterModule]
})
export class SharedModule {}
```

Core Module

The `CoreModule` is a module used to coordinate modules related to the domain of the application. There should only be a single instance of this module for the application. Use the `parentModule: CoreModule` to check for another instance of the module in the constructor.

```
import { NgModule, Optional, SkipSelf } from "@angular/core";
import { CommonModule } from "@angular/common";
import { SiteModule } from "../../site/site.module";

@NgModule({
  declarations: [],
  imports: [CommonModule, SiteModule],
  exports: [SiteModule]
})
export class CoreModule {
  /**
   * Use the check to determine if the [CoreModule] has been loaded in the
parentMod\
ule (AppModule root).
   */
  constructor(
    @Optional()
    @SkipSelf()
    parentModule: CoreModule
  ) {
    if (parentModule) {
      throw new Error(
        `CoreModule is already loaded. Import it in the AppModule only.`
      );
    }
  }
}
```

Site Module

In most applications, there are elements that are not specific to a feature. They are site-level. Therefore, I create a *SiteModule* to contain things that are specific to the site:

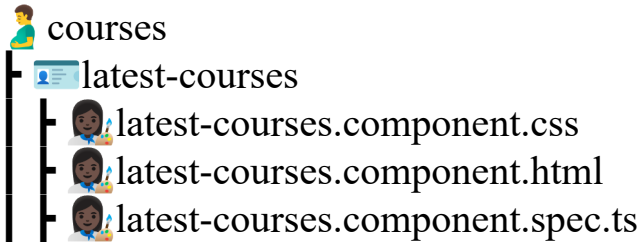
- navigation
- sidebar
- header
- footer

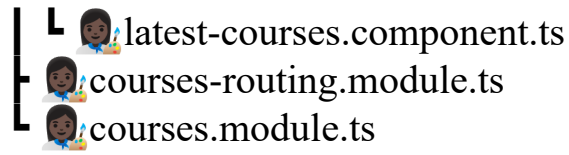
```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { NavbarComponent } from "./navbar/navbar.component";
import { SidebarComponent } from "./sidebar/sidebar.component";
import { FooterComponent } from "./footer/footer.component";
import { SharedModule } from "../modules/shared/shared.module";

@NgModule({
  declarations: [NavbarComponent, SidebarComponent, FooterComponent],
  exports: [NavbarComponent, SidebarComponent, FooterComponent],
  imports: [CommonModule, SharedModule]
})
export class SiteModule {}
```

Feature Modules

Feature modules are containers for all related items of a specific feature. You can encapsulate the components, routes, services for business logic, and more within the feature module. Feature modules can be configured for lazy-loading to increase application performance.

Consider a feature module to be a mini-application within your Angular application. Use feature modules to organize your code - create as many as you need for each of the domain features of your application. The example below is the `CoursesModule`, which is responsible for displaying course information to the users of the application.






```ts
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";

import { CoursesRoutingModule } from "./courses-routing.module";
import { LatestCoursesComponent } from "./latest-courses/latest-courses.component";

@NgModule({
  declarations: [LatestCoursesComponent],
  exports: [LatestCoursesComponent],
  imports: [CommonModule, CoursesRoutingModule]
})
export class CoursesModule {}
```

Routes

Components

Services

Models

Application Routes

6.3 Libraries

Most modern programming languages give you the ability to create reusable libraries. The question is how we do this with modern web applications? We need the ability to share and reuse code is if we're using compiled libraries. Modern web applications use JavaScript packages. These packages provide a library that can be used by installing the package in our application.

In order to consume these packages, JavaScript web applications need to consume these modules using a specific module loader. Depending on the type of application, different module loaders are used to initialize and use the specific JavaScript package. If you are an Angular developer and would like to consume custom libraries you need to develop separate JavaScript projects and compile them to specific package formats.

There are different package formats. The Angular team has created an Angular Package Format guide that helps us to create packages using typescript and angular that can be shared and reused by other applications. The reason for this is because the angular CLI did not have a way to compile and publish libraries to a package manager such as npm. Therefore you would need to implement a script that would create a package that could be consumed as a module by different web applications. Later this process was automated by a new package call ng-packagr.

Now the current version of the angular CLI has the capability to build libraries using this the ng-packagr package. when you add this new capability to the new angular workspace, it now provides us with the capability of creating libraries and sharing and reusing those libraries among other angular applications as well as other angular libraries. These capabilities have been available since angular version 6. You can now create a library project in your workspace using a CLI command. There are other options to provide if you would like the library to be publishable to a package manager like npm. Without this new tooling, the creation and management of publishing packages and consuming them within an angular application was very time-consuming and not a very efficient process.

> Note: our current angular development environment with the workspace allows us to be very effective when it comes to sharing and reusing code AS libraries.

At a recent developer conference, I spoke to a member of a team that was using Python and React for their web development environment. However, they were also using Angular libraries with TypeScript to create reusable libraries for their React applications. This means that libraries are not just for angular applications. These Angular libraries packaged using ng-packagr may be consumed by any JavaScript application client-side or server-side.

Libraries are essential to effective architecture. They allow applications to be built using libraries for specific concerns throughout an application. We can share and reuse code in the different layers of our application architecture. Using these libraries allows us to keep our code simple without copying and pasting code in different parts of the application. We do not repeat ourselves with this type of code, but we reuse one single source of truth contained in a single library.

There are different types of Library projects that we can create to make our Angular architecture more effective. here is a list of different types of libraries that we can use to simplify our architecture:

- Cross-cutting concerns
- Custom Frameworks
- Feature Libraries
- Component Libraries
- Common

Cross-Cutting Concerns

Cross-cutting concerns are elements within an application that are not specifically associated with a domain feature of that application. The responsibility of a cross-cutting library is to provide specific functionality that crosses many or all of the domains within the application - thus the name cross-cutting concern. If we use the same process to identify features of our application we can also group and categorize cross-cutting concerns

for our workspace. For example, most applications require logging and error handling. These are two cross-cutting concerns that can be implemented as a library. Implementing these concerns as libraries, allow us to reuse and share this code with other applications and library projects.

There are several benefits to implementing cross-cutting concerns as libraries. If we do this, we do not have to have logging or error handling code scattered throughout our applications and library projects. We now have a single source of truth for each of these cross-cutting concerns. Therefore any new logging features or changes to the logging functionality can be done within the logging library project.

> Note: Creating cross-cutting concerns as libraries is really a code organization technique that simplifies our application workspace.

Custom Framework Libraries

We can also create custom framework libraries for our applications and Library projects. A framework is really a set of features and functionality that provide some utilities and/or features to the consumer of that framework. You may find that there is a specific utility or feature in the domain of your business that could be shared and used by many other application projects. Instead of copying this code from project-to-project, create a library project, and implement the features and utilities in a single Library. This way any changes to that library are available to all of the consumers of that specific Library.

As a .NET/C# developer, I developed two frameworks to implement highly scalable business logic. Two of my first TypeScript projects were to create the same libraries as reusable TypeScript libraries for Angular applications. Let’s now look at each of these as examples of custom framework libraries.

- Business actions
- Rule engine

Business Actions

My main focus on developing web applications is on the business layer. I have specialized in implementing very complex business logic that includes the evaluation of simple and composite business rules along with data validation. I have built applications with very strong business layers that are 100% unit testable. Business logic is one of the most important elements of your application. Therefore, it is essential that it is consistent, testable, maintainable, and extensible.

Rule Engine

The Angularlicious Rules Engine is a JavaScript/Typescript based 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 for you to create any custom rule you need for your application.

- Provides a consistent way to implement business and validation rules and provide a consistent mechanism to retrieve the results.
- You can use the existing library of rules already implemented.
    - AreEqual
    - AreNotEqual
    - IsFalse
    - IsTrue
    - IsNullOrUndefined
    - IsNotNullOrUndefined
    - Max
    - Min
    - Range
    - StringIsNotNullEmptyRange
- You can create a reusable library of rules and use them in one or more applications.
- Combine default and one or more custom rules to create a `CompositeRule` - a rule that contains other rules (ruleset).
- 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.

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

A business rule engine allows the application to have a good Separation of Concerns (SOR). The Angular Rules 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.

Feature Libraries

You can also create feature libraries for your application. A feature library is specific to a domain within your application. For example, you may have a security feature library, or a library that is concerned with account management, or shopping carts, or anything else related to an e-commerce application.

A feature library candidate is a feature that many applications can share. For example, you may have a set of security feature that is shared by a suite of applications for your business. Another example could be a shopping cart

feature library that includes the components and services for all of your applications.

Component Libraries

You might find it useful to create a library for a specific set of components. This Library may have components that could be used by several applications. For example, you may have a notification component or a set of notification components that could be used by all of your applications. This library would include different kinds of components that are not specifically tied to a domain but could be used generically by different applications.

Here are some more examples.

- navigational or menus
- notifications
- popups
- cards
- tabs
- calendar
- form controls (inputs, dropdowns, checkboxes, radio, buttons)

This type of component library could use a 3rd-party package to abstract the implementation details of a specific control suite, such as:

- PrimeNg
- Material Design

Common

Another library project type could be for common elements. Common elements would include anything that is shared by most, if not all of your library or application projects. For example, you may have a specific model for your API responses. You will want all of your applications to conform to this schema provided by the API model. Therefore you can add an item like this to your common Library, and this Library could be made available

to all other libraries. Typically a common Library has no or very few dependencies on other libraries. This is to avoid recursive references.

- models or classes commonly used throughout all layers of applications
  - API response
  - service messages with severity
  - error messages
- items shared by all project types

API Response Model

A good practice is to have a consistent API response for all calls to the back end of your application. A consistent response is well-known and becomes familiar to the developers that need to work with the responses. Typically, a response is mostly successful and contains a payload of data. Otherwise, it is failed and should contain information or messages related to the failure.

In our example below, we have an abstract class that defines a response using a generic type. This means that we can have a single definition - but it can be specialized by specifying the data type of the response using the `<T>` generic. All API responses would then have a:

- specific response type
- `IsSuccess` indicator
- `Message` from the API
- `StatusCode` representing the HTTP Response code
- `Timestamp` that indicates the DateTime the response was created

Notice that the `ApiResponse<T>` is an abstract class. It cannot be instantiated but must be extended by sub-classes. The abstract class provides the default definition of a response model.

```
1 export abstract class ApiResponse<T> {
2   IsSuccess: boolean;
3   Message: string;
4   StatusCode: number;
5   Timestamp: Date;
6 }
```

> Note: Research and determine what data elements and format your team could use as a reliable API response model. It is an interesting discussion.

A **success** response (`SuccessApiResponse<T>`) would extend this abstract base class and provide an implementation for the `Data` payload of type `<T>` as indicated.

```
import { ApiResponse } from "./api-response";

/**
 * 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;
}
```

An **error** response (`ErrorApiResponse<T>`) would extend the same abstract base class and provide an implementation for a set of `ApiErrorMessage` items returned in the `Errors` property.

```
import { ApiResponse } from "./api-response";
import { ApiErrorMessage } from "./api-error-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: ApiErrorMessage[] = [];
}
```

7 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

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.

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.

7.1 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.

7.2 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.

```
1 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.

```ts
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);
  }

  /**
   * Use to initialize the logging service. Retrieves
   * application configuration settings.
   *
   * @param configService contains the configuration settings for the
application
   */
  private initializeService(configService: ConfigurationService) {
    if (configService) {
      this.configService.settings$
        .pipe(take(1))
        .subscribe(settings => this.handleSettings(settings));
    }
  }

  /**
   * 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 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)

```
import { LogWriter } from "./log-writer";
import { ILogEntry } from "../i-log-entry";
import { Severity } from "../severity.enum";
import { Injectable } from "@angular/core";
```

```
import { LoggingService } from "./../logging.service";

/**
 * Use this writer to log information to the browser console.
 */
@Injectable()
export class ConsoleWriter extends LogWriter {
  constructor(private loggingService: LoggingService) {
    super();
    this.loggingService.logEntries$.subscribe(logEntry =>
      this.handleLogEntry(logEntry)
    );
  }

  handleLogEntry(logEntry: ILogEntry) {
    this.targetEntry = logEntry;
    this.execute();
  }

  /**
   * No setup required for the console writer.
   */
  public setup(): void {}

  /**
   * Implementation of the abstract method. This will perform the
   * actual `write` action for the specified writer.
   */
  public write(): void {
    switch (this.targetEntry.severity) {
      case Severity.Debug:
        console.debug(this.targetEntry);
        break;
      case Severity.Information:
        console.info(this.targetEntry);
        break;
      case Severity.Warning:
        console.warn(this.targetEntry);
        break;
      case Severity.Error:
        console.error(this.targetEntry);
        break;
      case Severity.Critical:
        console.error(this.targetEntry);
        break;
      default:
        break;
    }
  }
}
```

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* 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.

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

  /**
   * Use to perform an setup or configuration of the [writer].
   * The [setup] method runs on all executions of the writer - and
   * is called before the [write] method.
   */
  public abstract setup(): void;

  /**
   * Use to validate the [log entry] before attempting to write
   * using the specified [log writer].
   *
   * Returns a [false] boolean to indicate the item is not valid.
   */
  public validateEntry(): boolean {
    const validationContext = new ValidationContext();
    validationContext.addRule(
      new IsTrue(
        "LogWriterExists",
        "The log writer is not configured.",
        this.hasWriter
      )
    );
    validationContext.addRule(
      new IsNotNullOrUndefined(
        "EntryIsNotNull",
        "The entry cannot be null.",
        this.targetEntry
      )
    );
    validationContext.addRule(
      new StringIsNotNullEmptyRange(
        "SourceIsRequired",
        "The entry source is not valid.",
        this.targetEntry.source,
        1,
        100
      )
    );
    validationContext.addRule(
```

```
68        new StringIsNotNullEmptyRange(
69          "MessageIsValid",
70          "The message is required for the [Log Entry].",
71          this.targetEntry.message,
72          1,
73          2000
74        )
75      );
76      validationContext.addRule(
77        new IsNotNullOrUndefined(
78          "TimestampIsRequired",
79          "The timestamp must be a valid DateTime value.",
80          this.targetEntry.timestamp
81        )
82      );
83
84      return validationContext.renderRules().isValid;
85    }
86
87    /**
88     * Use to implement the actual write of the [Log Entry].
89     */
90    public abstract write(): void;
91
92    /**
93     * Use to finish the process or clean-up any resources.
94     */
95    public finish(): void {}
96  }
```

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. 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

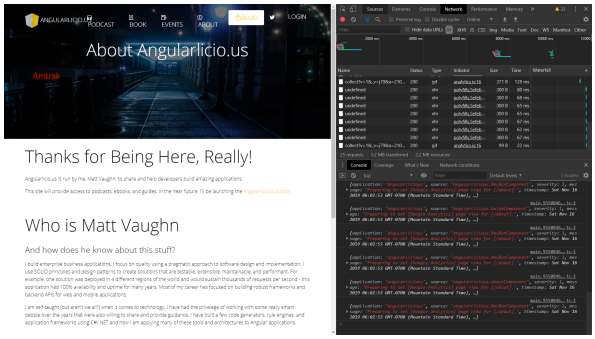
```
import { LogWriter } from "./log-writer";
import { ILogEntry } from "../i-log-entry";
import { ConfigurationService } from "@angularlicious/configuration";
import { Optional } from "@angular/core";
import { LogglyService } from "ngx-loggly-logger";
import { LoggingService } from "../logging.service";
import { IConfiguration, LogglyConfig } from
"@angularlicious/configuration";

export class LogglyWriter extends LogWriter {
  config: LogglyConfig;

  constructor(
    @Optional() private configService: ConfigurationService,
    private loggingService: LoggingService,
    private loggly: LogglyService
  ) {
    super();
    if (this.configService && this.loggingService) {
      this.configService.settings$.subscribe(settings =>
        this.handleSettings(settings)
      );
      this.loggingService.logEntries$.subscribe(entry =>
        this.handleLogEntry(entry)
      );
    }
  }

  handleSettings(settings: IConfiguration) {
    if (settings) {
      this.config = settings.logglyConfig;
      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  }
```

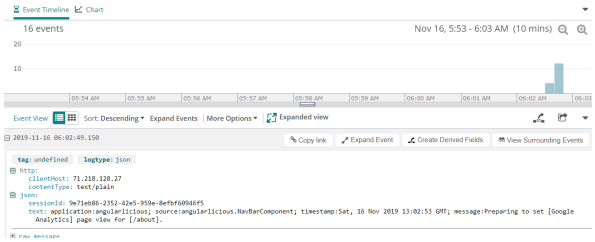
The image below shows the browser console with log events. This same information is sent to Loggly. The HTTP request contains the following payload.

```
1 {
2   "text": "application:angularlicious; source:angularlicious.GuideComponent;
timesta\
3 mp:Sat, 16 Nov 2019 13:02:43 GMT; message:Preparing to set [Google Analytics]
page v\
4 iew for [/custom-angular-modules].",
5   "sessionId": "9e71eb86-2352-42e5-959e-8efbf60946f5"
6 }
```



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.

7.3 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 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)
- InternalError
- RangeError
- ReferenceError

- SyntaxError
- TypeError
- URIError

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

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] 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.

> 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`);

    this.configService.settings$
      .pipe(take(1))
      .subscribe(settings => this.handleSettings(settings));
  }

  handleSettings(settings: IConfiguration) {
    if (settings && settings.errorHandlingConfig) {
      this.config = settings.errorHandlingConfig;
      this.hasSettings = true;

      this.loggingService.log(
        this.config.applicationName,
        Severity.Information,
        `Application [ErrorHandler] using configuration settings.`
      );
    }
  }

  /**
   * Use to handle generalized [Error] items or errors from HTTP/Web
   * APIs [HttpErrorResponse].
   *
   * @param error
   */
  handleError(error: Error | HttpErrorResponse): any {
    if (this.config.includeDefaultErrorHandling) {
      // use the [super] call to keep default error handling functionality -
-> conso\
le;
      super.handleError(error);
    }

    if (this.hasSettings) {
      // 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.

```
1 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
- HTTP Status Codes
- JavaScript Error Object
- Exceptional Exception Handling in JavaScript
- Angular ErrorHandler (error_handler.ts)
- Angular HttpClient :: Error Handling
- Angular HttpResponse
- Angular HttpResponseBase
- Chocolate in my Peanut Butter

7.4 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.

```
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 at Rangle.io.

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 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*.

```
/**
 * 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,
```

```
44            useClass: ErrorHandlingService,
45            deps: [ConfigurationService, LoggingService],
46          },
47          AuthenticationService,
48        ],
49      };
50    }
51  }
```

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.

```
1  }
2  @NgModule({
3    declarations: [],
4    imports: [
5      CommonModule,
6      ErrorHandlingModule,
7      LoggingModule,
8      ConfigurationModule.forRoot({ config: AppConfig }),
9      SecurityModule,
10   ],
11   providers: [
12     // DO NOT ADD PROVIDERS HERE WHEN USING [SHARED] MODULES; USE forRoot();
13   ],
14 })
```

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.

```
1  import { NgModule, ModuleWithProviders } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { ConfigurationContext } from './configuration-context';
4
```

```
@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

7.5 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

```
1 export abstract class ApiResponse<T> {
2   IsSuccess: boolean;
3   Message: string;
4   StatusCode: number;
```

```
5   Timestamp: Date;
6 }
```

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.

```
1 import { ApiResponse } from './api-response';
2 import { ApiMessage } from './api-message';
3
4 /**
5  * Use to define a successful API response. A successful response will
6  * most likely include a payload of data (i.e., use the Data property).
7  */
8 export class SuccessApiResponse<T> extends ApiResponse<T> {
9   Data: T;
10   Messages: ApiMessage[];
11 }
```

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.

```typescript
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

```typescript
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.
   */
  constructor(message: string, displayable: boolean, id: string | null) {
    this.message = message;
    this.isDisplayable = displayable;
    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  }
```

7.6 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:

```
1 ng g library httpService
2 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

```typescript
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;
    return options;
  }

  /**
   * Use to create a new [HttpHeaders] object for the HTTP/API request.
   */
  createHeader(): HttpHeaders {
    let headers = new HttpHeaders();
    headers = headers.set('Content-Type', 'application/json');
    return headers;
  }

  /**
   * Use to execute an HTTP request using the specified options in the
[HttpRequestO\
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        }
130      }
131    }
132 }
```

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;

  constructor(
    @Inject(HttpService) public httpService: HttpService,
    public loggingService: LoggingService
  ) {
    super(loggingService, 'ThingsToDoHttpService');
  }

  RetrieveThingsToDo<T>(): Observable<ApiResponse<T>> {
    const requestUrl = this.baseUrl.concat('things');
    const options = this.httpService.createOptions(
      HttpRequestMethod.get,
      this.httpService.createHeader(),
      requestUrl,
      null,
      this.noCredentials
    );
    return this.httpService.execute<T>(options);
  }
}
```

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.

```
import { Observable } from 'rxjs';
import { ApiResponse } from '@tc/foundation';

export interface IThingsToDoHttpService {
  /**
   * Use to retrieve things to do.
   */
  RetrieveThingsToDo<T>(): Observable<ApiResponse<T>>;
}
```

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.

```
 1 execute<T>(requestOptions: HttpRequestOptions):
Observable<HttpResponse<ApiResponse<\
 2 T>>> {
 3     try {
 4       return this.httpClient.request<T>(
 5         requestOptions.requestMethod.toString(),
 6         requestOptions.requestUrl,
 7         {
 8           headers: requestOptions.headers,
 9           observe: requestOptions.observe,
10           params: requestOptions.params,
11           reportProgress: requestOptions.reportProgress,
12           withCredentials: requestOptions.withCredentials
13         }
14       ).pipe(
15         retry(1)
16         catchError((errorResponse: any) => {
17           return this.handleError(errorResponse);
18         })
19       );
20     } catch (error) {
21       this.handleError(error);
22     }
23   }
```

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: 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.

```
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    */
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           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 }
```

8 CLEAN Architecture Layers

During the last several years, you might have heard about **CLEAN Architecture**. There are many blog posts, videos, and even a book dedicated to CLEAN architecture. It is using principles and patterns that have been around for decades. It provides guidance on separating concerns and creating boundaries between the concerns. It is definitely use-case driven and fits nicely with other development practices like Domain-Driven Design and Test-Driven Design, and others. It leverages the capabilities of Object-Oriented principles and patterns.

8.1 Why CLEAN Architecture for Angular?

There is not too much discussion on the internet about applying the principles of CLEAN architecture to Angular applications. Theses principles are easy to grasp and learn. However, they may not be as familiar to front end developers. Until recently web applications were at the mercy of the capabilities of plain JavaScript. Today, Angular leverages the strong-typing and object-oriented capabilities of TypeScript - JavaScript that scales.

Even with small applications or proof-of-concept (POC) applications can benefit from good programming practices that include the principles found in CLEAN architecture. The best fit for using CLEAN Architecture is for applications that are core to business success. Enterprise applications that need to be reliable; and easy to maintain and extend over time. If you are part of a development team working on larger projects, a CLEAN architecture provides other benefits as well.

At this point, you might have a few questions. We are going to discuss the main concepts and principles of CLEAN architecture. This includes discussions about the capabilities of Angular and Typescript and how these capabilities enable building applications that have:

- great separation of concerns
- single responsibility features
- well-defined boundaries between application layers
- domain-specific interfaces and abstractions for domain operations
- tests for the most important part of your application - business logic
- a consistent implementation of business rules and validation
- a consistent mechanism to pass information between layers

8.2 Getting Started with CLEAN Architecture

You might ask, “What exactly is CLEAN architecture?” Let’s break it down.

What is clean? When you think of the word clean, what do you think about? Is it a clean yard, room, car, or a garage? Can you park your car in your garage? Have you seen a dirty and cluttered garage that is so full that you cannot even park a car in it. We know what clean is. Clean is organized. Clean means no trash, no waste, and no clutter. There is nothing that is broken, unused or unwanted.

> CLEAN Architecture is:
>
> - organized
> - contains no trash, waste, or clutter
> - removes broken, unwanted or unused code

When we think of a clean garage, we think of something that is very organized - where everything is in its proper place. Related things are grouped together and may even be stored in special containers to keep things neat.

What does this have to do with software or architecture? Are there any benefits to being organized and clean with our code? Well, we need organization and clean in order to be effective and efficient. Clean architecture means that things are well organized in our applications. This also means that we do not have anything that is broken or unused in our codebase. We take care of the code for our applications and treat it as a valuable asset. This means that cleaning and keeping our codebase clean over time is very important. This is not something that is done once and you can forget about it.

> A clean room doesn’t always stay clean over time. It takes effort, diligence, and a plan to keep something clean and pristine. It can be done.

Therefore we would not keep code that is unused or broken in the code repository. We would remove it so that it does not dirty or complicate what is useful. We need to fix things that are broken. We need to organize and make sure that related things are grouped together.

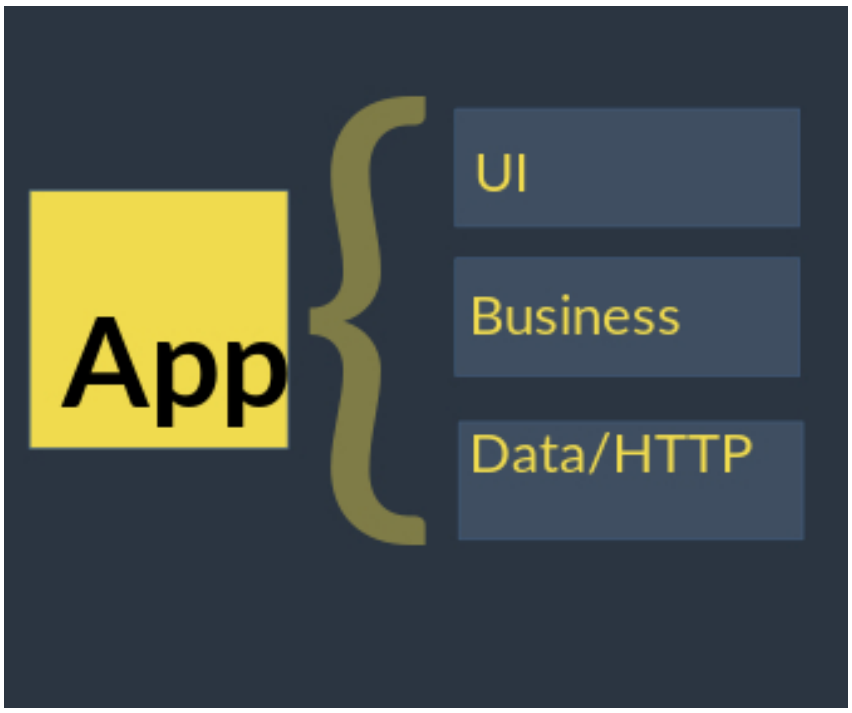
In regard to organizing code we would take the time to group and categorize related things together. This means that we group members (Components, Services, Business Logic, Business Rules, etc.) of application features together in modules. Organize your code in specific layers so that each layer can have a specific responsibility and concern.

> Take the time to categorize and group related things together. You won't be disappointed.

8.3 How to Organize Code

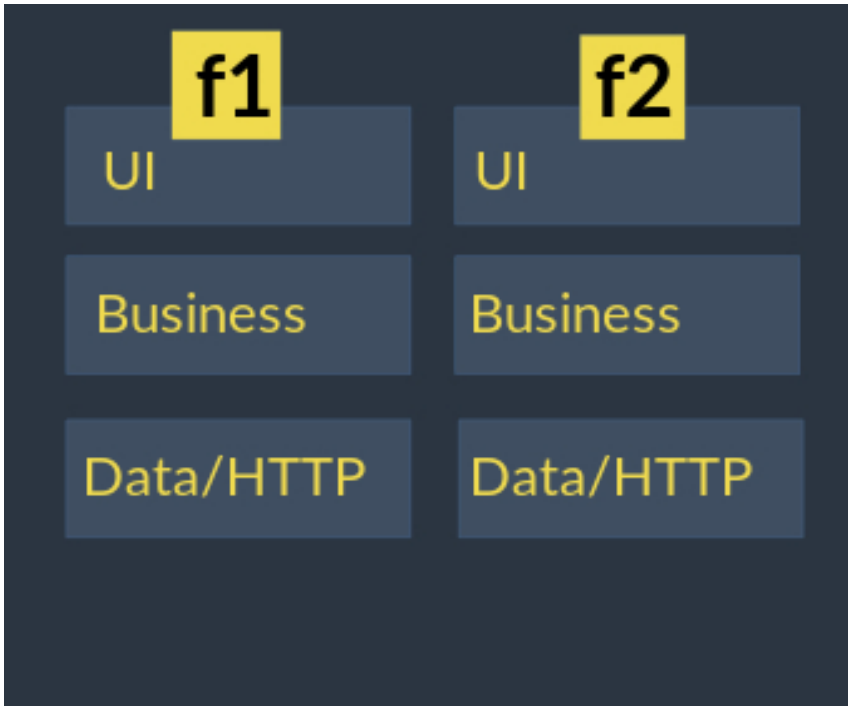
I think the big question is how do we organize our code? Or what do I need to do the have a good architecture for my application? The answer is very simple, in fact it is not a technical answer. It is the ability to categorize and organize our code effectively. So the question is how do we organize our code? Let's use a technique found in enterprise applications - horizontal layers and domain/feature verticals.

- layers
    - provide a specific concern (e.g., user presentation, business logic, data access)



Architecture Layers

- verticals
    - represent a specific domain feature (e.g., products, orders, shipments, etc.)
    - each vertical contains an implementation of the defined layers above



Architecture Verticals

Organize by Layers

All layers have a specific concern no matter what the actual domain feature vertical is. For example, the *user presentation* layer is concerned with collecting and displaying information to the user. It is the same concern no matter if the feature is for products, orders, shipments, or any other domain feature.

> Note: Our sample application demonstrates different types of layers for the architecture. There is no rule on how many layers you should have. This depends on the complexity of the application. There is a little more overhead with more granularity (more layers) - however, you get to define specific behaviors (concerns) for each of these. It is really about creating the boundaries between the layers and respecting those boundaries to get: single responsibility and separation of concerns at a high-level.

We are going to organize our code using horizontal layers along with core domain verticals. Here are the layers:

Layer	Concern
Presentation Layer	the collection and display of information to the user
Controller Service (mediator)	coordinates access to domain services and manages UI specific states; allows one ore more components to communicate and/or share information
Core Domain Service	a service dedicated to a specific domain feature, contains nothing related to UI; provides an API to core business operations, implemented as a facade; encapsulates business logic for the domain feature
Business Layer	a layer to implement business logic, contained within a core domain service, not directly accessible from consumers - requires use of domain service API
Data Repository	an abstraction to the data access layer; provides the interface/API for data persistence
Data Access	the actual implementation of a specific data access mechanism to persist data (HTTP/Web APIs, LocalStorage)
Data	the actual data store where application data is persisted

A layered approach allows each layer to have distinct concerns and very specific responsibilities. These are the building blocks of CLEAN architecture. Layers provide boundaries between these concerns. These boundaries provide the foundation of CLEAN architecture. The communication flow between the layers is top-down *only*. A layer cannot initiate communication with a layer directly above it.

The layers provide code organization by their specific concern.

> *I have seen many developers and teams create a **service** for the set of components or feature. This is a good start. However, these services*

> *are usually overloaded with UI concerns, core domain logic, data access, business rules, and authorization concerns. It is too much!*

Angular and Typescript already provide what we need to create a layered architecture. All we need now is a little guidance, sample code, and a reference application to demonstrate the simplicity and benefits of a layered approach.

Organize by Principles

The first principle is the principle the **single responsibility** principle. The other principle is the **separation of concerns** principal. If we use these principles to group, categorize and organizer code - it will provide about 80-90% of good architecture. It almost sounds too good to be true. Or too simple. It is the process of thinking about the relationships between things - which includes analysis and design, categorizing and grouping related things together.

Now that we have an idea about the main elements of CLEAN architecture it should be easier to review these layers one by one.

8.4 Start at the Top: Presentation Layer

Layers:

- **presentation layer**
- component service (mediator)
- core domain service
- business layer
- data repository
- data access
- data

So let’s start at the very top of our application. The presentation layer is responsible for collecting and displaying information to our users. This is the single responsibility of the presentation layer. Its concern is to collect and display information. Therefore the presentation layer would not be concerned with the actual mechanisms of storing information and processing business logic or business rules for application. These concerns belong to other parts of the application - different layers of the application. Therefore, the architecture uses specific layers to separate the concerns so that each of these layers have a specific responsibility. Each layer fulfills a different responsibility and concern for different parts of the application. Many applications retrieve and store information in a database. Therefore a lot needs to happen between the presentation layer and the actual data layer that stores this information. Let’s now take the time to discuss the layers and the boundaries between these layers so that we can specifically identify each concern.

8.5 Component Service - Mediator of UI and Core Domain

Layers:

- presentation layer
- **component service (mediator)**
- core domain service
- business layer
- data repository
- data access
- data

In most strong enterprise architectures there is a business layer that is responsible for implementing the domain of the application. This includes processing business logic and business rules as well as validation of data. Many times access to the business logic layers of an application are provided through a set of domain services. These services act as a facade or an API to the specific core domain features of the business logic.

In application we would not want the presentation layer to have direct access to the core business logic layer. Therefore the presentation layer uses a component service to interact with the business layer. In many Angular applications developers create a service that is responsible for communicating with the back end of the application or to the Web APIs.Let’s however take a different approach.

The presentation layer communicates with the core business layer via a mediator service that coordinates operations between the core domain services that implement the business logic and the presentation layer that is responsible for displaying and collecting information. Let’s call this a component service. The concern is to mediate the interactions between the core business layer and the presentation layer. It also provides support to the presentation layer to manage state and the observables that binds data for display. Many presentation layers have complex display logic and the

mediator component service helps or assist the state for displaying information for one or more components in the presentation layer.

This mediator service that is between the presentation layer and the core business layer has the responsibility to mediate between presentation and core business logic. Its concern is directly related to the presentation layer, but it also knows how to communicate with the core business layer.

So far we have discussed 3 different layers in architecture. We now have a presentation layer a mediator service and the core business layer.

8.6 Core Domain Service

Layers:

- presentation layer
- component service (mediator)
- **core domain service**
- business layer
    - data repository
- data access
- data

The core business layer is responsible for the implementation of business logic for the domain of the application. For example, if we have an e-commerce application we would have core domain services for shipping, products, shopping cart, catalog, and other things related to an e-commerce store. Each one of these items would be its own separate core domain service. So when we talk about architecture we think of layers in terms of concerns. However we can also separate these concerns by their features or domains within the application as distinct verticals. Now we have specific layers that are responsible for specific concerns and we also have separate verticals for the different parts of the application domain.

As previously mentioned the entry point into our Domain Service is a service. The service acts as a facade to our business layer. The service does not implement any business logic but acts as the mechanism to handle a request and provide a response to the consumer of the service. The real work is done in the business layer where the business logic is processed. The *domain service* allows us to encapsulate the business logic within the service so that we can provide a concrete implementation but abstract the details of that implementation. Now the consumers of the domain service know how to use the service; but do not necessarily know the details of how something is done. This allows you to have a rich API for a domain service and to keep the implementation details separate from the consumers of that service.

8.7 It is Just a Business Layer Decision, Right?

Layers:

- presentation layer
- component service (mediator)
- core domain service
- **business layer**
- data repository
- data access
- data

So far we have mentioned the presentation layer, the mediator service, and the domain service. Let's discuss the next layer of the application that is contained within the domain service. This is the *business layer*.

The business layer is a unique layer to the application. This layer is typically abstracted from the consumer of the domain services and cannot be directly accessed. This is where our rich business logic, workflows, and specific processes or intellectual property is implemented for our application. The business layer is also responsible for validating information before it is stored in the database. It is also responsible for implementing simple or complex business rules for the different operations of the domain. Typically the business layer is implemented as a set of specific actions - units of work. These units of work allows business logic to be very testable. Additionally you can compose complex operations by combining multiple actions together. Many times a single action contains one or more business rules and data validation rules or both.

Responsible for Processing Business Logic

Many times business actions in the business layer determine if the user is authenticated or authorized to perform certain operations. During the processing of business logic many times information needs to be provided to the consumer or to the user of the application. Therefore the business

actions and the business logic need to provide messages or information back to the user as part of the response for the specific request.

To summarize, we have talked about a presentation layer a mediator service that is a lair that coordinates the actions between a presentation layer and the core business service we also have the core business service lair. And within those domain services we have a business logic lair. The business logic layer includes a rich set of business actions and business rules. The next layer that we will talk about is a repository layer.

Where does business logic *NOT* belong?

8.8 Data Repository

Layers:

- presentation layer
- component service (mediator)
- core domain service
- business layer
- **data repository**
- data access
- data

The repository layer is really an abstraction between the business layer and the data access layer. It is good to provide an abstraction because many times the mechanisms or the providers to store and retrieve information change over time for your application. Sometimes using a repository pattern also assist in the implementation details and the testing of your application.

Repository pattern is usually implemented as an interface that provides an API for the data access operations that you will need to fulfill the domain of your application. For example in our products domain service, we will need operations to retrieve products by ID, retrieve a list of products, to create a new product, or update an existing product. When we use the repository pattern the business logic layer does not need to know the specific technology or details of how the data is accessed. It just needs to know what repository to use to store and retrieve information.

8.9 Data Access

Layers:

- presentation layer
- component service (mediator)
- core domain service
- business layer
- data repository
- **data access**
- data

The next layer in our application architecture is the actual data providers that implement the repository interface. For example, we need a data provider that implements the repository that connects to the Web API's of the application.

The repository pattern along with the data providers allow for a specific mechanism to retrieve and store data by using a simple interface or abstraction between the data provider and the business logic layer.

8.10 Data Layer

Layers:

- presentation layer
- component service (mediator)
- core domain service
- business layer
- data repository
- data access
- **data**

The last layer of her application is the data layer. The data layer is responsible for this specific mechanism of storing information. Most of the time it is a database that is accessed using Web APIs. It could also be a browser's local storage.

8.11 Review of Layers

So let's recap all of the layers that we have discussed so far. We have talked about the presentation layer, the component service layer that acts as a mediator between the presentation layer and the actual core domain service - which is a layer the 3rd layer and the core domain service includes internally a business logic layer. The business logic layer use as a repository layer to abstract the interface between the actual data storage and the mechanism used to persist and to retrieve data from that storage.

Using all of these different types of layers may sound very confusing and complicated. However when we think of the separation of concerns and single responsibility principles we see that these layers have their specific concern and the boundaries between them actually provide a clean separation of responsibility. This keeps our code well organized.

When we think of implementing R code using D specific layers then we would never have an HTTP call or ridge and aging from our presentation layer. The reason for this is that it is not the concern nor the responsibility of the presentation layer to make HTTP calls to a Web API. In fact many things have to happen before you would make such a call. For example you would want to make sure the user is authorized to perform the operation and/or the business rules all pass allowing the operation to be performed. Therefore we wouldn't want to put all of this code in the component or presentation layer of our angular application.

Many times I see teams or developers implement a service that their component can use. And this service contains really everything the API, processing business rules or data validation, as well as the HTTP operations to the Web API.These type of services in applications are doing way too much and violate the separation of concerns and single responsibility principles. Therefore it is essential that we use a layered approach to separate these concerns. As well as to create these domain verticals of rap occasion's that separate the specific features into modules vertically then implement the specific layers as well.

Key principles of CLEAN architecture have been around for about 7 or 8 years. Actually the principles have been around a lot longer than that but the concept of CLEAN architecture has been well defined over the last several years. Many enterprise back end architectures use CLEAN architecture principles. Basically these principles are there to help you to define the boundaries between each of these layers and how the communication or interaction between these layers should occur.

The capabilities of angular to organizer code using modules and services as well as the capabilities of pipe script to implement a specific design patterns and to create interfaces and abstractions allow us to implement CLEAN architecture. We need to take advantage of these capabilities. The front end development paradigm has been changed because the richness of the angular platform and the capabilities of the type script programming language enable us to use object oriented programming techniques, specific design patterns common and coat organization techniques then help us to be more effective in or implementation and designed.

Angular even provides a rich development environment that includes a monorepo workspace. Therefore we have the capability of managing multiple application projects as well as multiple library projects in the single environment.The capabilities of the workspace enable us to implement rich crosscutting concerns as reasonable and shareable libraries for our applications. This is yet another great example of a capability that allows us to organize our code at a higher level.

9 Angular Architecture(s)

Let's start by saying, "There is no single correct way to implement application architecture." OK, now that this is out of the way, let's talk about architecture.

The reason for this is that there are too many different factors involved. The application may require the same functionality - however, consider the age and skills of the users, the environment where the application is used, when it is used, and what the specific goals and objectives are. Other factors may include the reasons for building the application. How soon does it need to be in the market? One consideration that many forget to ask is, "Does the developer or team have the necessary skills to execute and deliver the solution *on time*?".

There are many considerations. Time and budget constraints are important as well. The old saying is true: You get what you pay for. Therefore, the best advice for implementing architecture (i.e., design and plans, tools and materials, and execution) is to use common sense and apply as many good programming principles as you can without sacrificing the delivery of the solution:

1. with the right features
2. for the right people
3. at the right time

There are many approaches and disciplines dedicated to the management of software projects. It is safe to say that architecture will impact all of them in some way. Even if you do not have a dedicated software architect or team lead providing architectural guidance, you can still have good architecture. It takes one or more individuals willing to take the time to ask the right questions to fully understand the goals, problems, and reasons for building the solution.

One of the things that I really like about Angular is that it is very opinionated about how things are done. The fact that there is a CLI that enables us to consistently create workspaces, projects, modules, components, and services is a great start to good architecture. We have a great tool in Visual Studio Code and a selection of materials in the Angular framework (or rather platform) that provide the foundation for the beginnings of a strong architecture. The CLI commands use templates to generate and help us quickly scaffold application elements. We also have the ability to create our own Schematics with templates to generate even more application elements.

> Pro Tip: Take the time to learn how to use the Angular CLI. Use the `--help` option to see lists of options for each command. Experiment with the commands using the `--dry-run` flag to execute the command without modifying any files.

Remember the (3) main things of an effective architecture - they are listed below if you do not recall. With the Angular platform, there isn't much decision about which tools and materials to use. I think one of the main concerns is to understand the capabilities of the CLI, Visual Studio Code, formatting tools, build tools, and the capabilities of the Angular framework. This is a continual learning process - we should have a specific plan to learn and understand our tools and materials.

1. design and planning
2. tools and materials
3. execution and delivery

One of the main concerns of architecture is the structural aspects of the design and how things will work together to provide the desired solution. This is a significant endeavor that will impact the implementation, delivery, and maintenance of the application for a greater amount of time than it took to perform the initial development. Therefore, choosing the right structural approach is very important. Some studies have shown that the cost to maintain an application is almost twice as expensive in time and money as it was to initially develop it. Choose wisely.

The next sections provide a review of different Angular architectures (structural) with their concerns and merits. There may be more or less different architectures depending on your experience. However, the list is broad enough to demonstrate some characteristics that allow us to determine how we might proceed within the context and boundaries of our specific needs.

9.1 Clean Architecture Map to Layered Architecture

To begin, let's start with the end in mind. This section presents a layered architecture using principles and concepts from CLEAN architecture. Which basically means, that we have well-defined layers that have specific concerns and responsibilities. Between each layer is a well-defined boundary that allows the layer above it to communicate with it. The communication flow is one-way, meaning that layers cannot initiate communication to the layers above it. These layers may only provide a way for the above layer to communicate with it and then to provide a response if any.

This layered architecture is key in following the Separation of Concerns (SoC) principle. The layers provide a granularity of functionality that enables great efficiency within each layer. Understand the concern of each layer and what the means in terms of the responsibilities of that layer. A layer becomes an important element for the application architecture. In our business application, each layer provides a specific and integral part of the overall solution. When you think about it, it is really just another code organization strategy to logically categorize and group related things together - in the case of layers, the category is related to the specific concern: presentation, business logic, or data access.

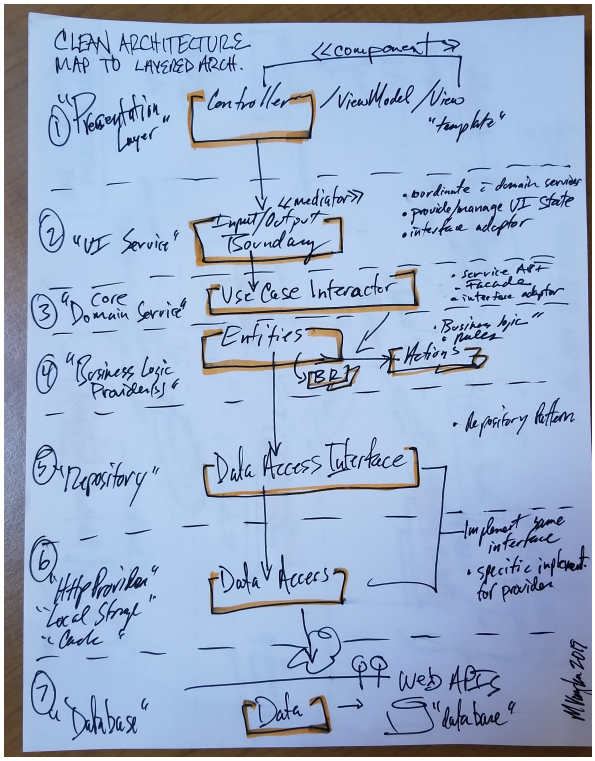
When we talk about CLEAN architecture, we want well-defined boundaries between each of the layers. The boundaries also include how one layer communicates or interacts with another layer. It also includes how we organize our code logically or physically. The location of the code matters and what structure we use to do that is also important. The Angular platform provides us with a lot of options for code organization strategies.

The list below shows that there may be many options in the number of layers that we implement if any, right? Your specific needs may not require this level of layer granularity or may even not require any application layers. The layered approach is just a structural and code organization technique that provides different benefits but may cost a little more in time and effort.

The following table maps the players noted in the CLEAN Architecture book to a layered architecture approach. The terminology may be somewhat different - however, the intent and specific responsibilities are fairly common in *n-tier* architecture patterns.

Layered	**Clean**	**Notes**
Presentation	Controller	Contains view model, view (i.e. Template)
UI Service	Input/Output Boundary	Responsible for coordinating with domain services. Provides and manages UI state. Acts as an interface adaptor between the *Presentation* layer and the core *domain* service.
Core Domain Service	Use Case Interactor	Provides an API for a core domain service. A facade that encapsulates the business logic layer. Consider as an interface adaptor.
Business Logic Provider	Entities	Provides the implementation of the application business logic. Includes processing business actions and rules to compose business logic behaviors. 100% unit testable.
Repository	Data Access Interface	Provides an API/Interface for data repository operations. Implements the enterprise Repository pattern. Implements the same interface as the concrete Data Access Provider. Abstracts the actual mechanism to persist/retrieve data from the business layer.
Data Providers	Data Access	The concrete implementations for the repository interface for data operations. Implements the specific mechanism (i.e., HTTP, LocalStorage, etc.).

Layered	Clean	Notes
Database	Data	Represents the data storage container for the application. An application may have more than one. Each Data Provider implementation provides storage access to specific data container. May be represented as a Web API.



Clean architecture map to a layered architecture.

9.2 Layered Architecture

Regardless of implementing a set of layers for different application concerns, most applications will have to perform and handle the same things. It is just a matter of where (code organization) and how you implement the concerns. Therefore, consider a layered architecture as just a convenient and logical way to organize your code. The organization allows each layer to provide the implementation of a specific concern - each with its own set of responsibilities.

Most applications have most if not all of the following concerns:

1. Collecting and displaying information to the user.
2. Coordinating the retrieval and persistence of information and how to get the information to the UI elements.
3. Business logic for different operations and features of the application.
4. Some element of verifying and/or determining that information/data is in a specific state.
5. The ability to store and retrieve information reliably - data storage.

> Note: There is no rule that you have to use a layered architecture. It is just a widely-accepted pattern that has a long track
> record of providing reliable solutions in many different technology platforms.

The terminology or layer names to describe these concerns may vary. Here is a list that I will use in our discussions about the different Angular architectures under review.

1. UI Presentation
2. UI Service (mediator/adaptor)
3. Core Domain Service
4. Business Logic
    - Business Actions
    - Business Rules
5. Repository

6. Data Provider (HTTP Data Access)
7. Web API (Data)

Horizontal Layers

Each layer is a horizontal layer where each layer is stacked. The top-level layer communicates with the layer beneath it and with no other layer(s) below that. A layer should not have direct access to any other layers, except with the one directly below it. Layers initiate communication with the layer below it. This allows for the communication flow to be one-way in terms of initiating requests. Each layer then has the capability to provide a response to the consumer of that layer. This layered approach maintains a dependency on the specific layer below it. Therefore, when a layer has a well-defined concern, the interface or API to that layer is more focused and well-defined. The entry-points to each layer define the boundaries between each layer. The layers and the boundaries are very important structural concerns of the architecture. The location of the code and how it is organized creates a physical boundary. If they are in separate modules the access is limited to what is exported or *exposed* by the module. Therefore, you have more control to define the endpoints or API of each layer.

- one-way communication flow
- layers cannot initiate communication with the layer above it
- fewer dependencies for each layer
- respect the boundaries between each layer - any exceptions or deviations could cause unanticipated negative effects

  As with anything in real life, there may be an exception or edge-case that may cause some sort of deviation or allowance of doing something atypical (not following the layered guidance). This most likely falls into the category that is more of a fundamental design issue rather than an issue with strictly following the layered-approach strategy. You must decide if the convenience of a deviation will affect your workflow, delivery, and maintenance of the code.

Benefits of Layered Architecture

Using layers to define different concerns within an application is really a code organization strategy. One aspect of architecture is to decompose and break things down into manageable items that can be categorized and logically grouped together. Understand all of the items under consideration and their relationship to other things is a critical step in this process - it is mostly a thinking activity and exercise for you and your team. It may not be easy to do at times. However, there are many benefits. Here is a shortlist of some benefits of a layered approach.

- Separation of Concerns
- Encapsulation of how things are implemented
- Refactor work has little or no impact to external consumers
- Simpler maintenance over time
- Easier to test in isolation - fewer dependencies.
- Fewer or no merge conflicts - rare that multiple developers are working on the same layer, feature, and/or item.
    - There are more opportunities for merge conflicts in non-layered design because certain areas of code are often overloaded with too much responsibility.

10 Architecture Options

As mentioned before, there are many options for web application architecture. Each option has its own merits. Therefore, choosing an architecture should be based on the context of:

- Why is the application required; what are the goals and objective of the application?
    - Is it an essential part of the business?
    - Does it drive revenue for the business?
    - Does it solve specific user needs that cannot be done in any other way?
- Who uses the application?
    - External customers?
    - Business partners?
    - Administrators or internal stakeholders?
    - Are there any accessibility (a11y) concerns?
    - Are there any internationalization (i18n) or localization (l10n):
- When is the application is used (frequency)?
    - Used frequently by a large number of people?
    - Used rarely?
    - Used by a large number of users, but only during specified peaks?
    - Are there usage peaks during specific seasons or business cycles?
- What are the requirements for reliability?
    - Is performance an issue?
    - What happens if the application is not available?
- Where is it used?
    - At work on PC/browsers?
    - Mobile?
    - Mobile and PC/browser?

There are more questions to ask. Do not leave any stone unturned. You will have most if not all of the information to deliver a fine solution - but it is

essential to understand, plan, and design the solution. This cannot be done without thorough analysis and asking great questions.

10.1 Default Architecture (#1)

Use the Angular CLI to create a new application project in the workspace. The application project by default contains a single *AppModule*. Some applications do not have the complexity to require a multi-layer architecture. We get this by a single CLI command:

```
1 ng generate application <MY-APP-NAME>
```

This command creates the application's source code structure, scripts to build, test, lint, and serve the application. It integrates the project into the Angular workspace using the *angular.json* configuration file.

The application's single *AppModule* will contain all of the applications required members.

- components
- services
- data models
- if any cross-cutting concerns

The Pros and Cons of Angular Default

This is the default architecture that we get with the default CLI template. No additional orchestration is required to create additional modules for *many* features. This application is singular in purpose and most likely has a single feature. If there are more feature they may be organized within a specific folder structure.

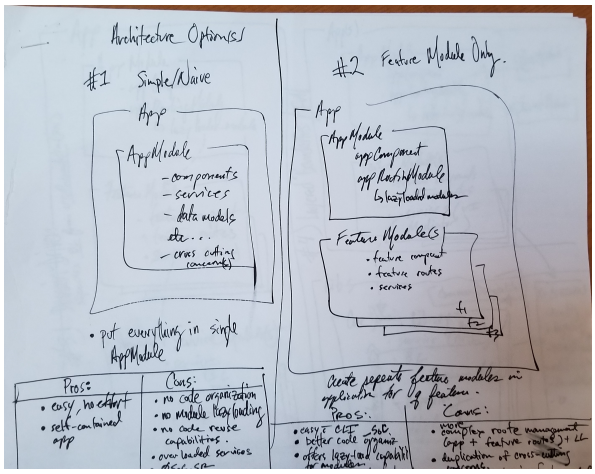
Pros:

- easy, little effort to set up and configure
- self-contained application with little dependencies

Cons:

- little or no code organization for more complex applications
- no support for module lazy-loading; may have performance issues

- no code reuse capabilities
- may suffer from overloaded services
- little or no Separation of Concerns/Single Responsibility



10.2 Feature Module Architecture (#2)

The *Feature Modules*-based application will most likely have several features. The size of the features as well as when or if they are accessed by users of the application, may require them to be lazy-loaded. Lazy-loading provides some performance improvements but may be optional depending on the application's performance and bundle size(s).

Create separate feature modules within the application using the Angular CLI. Angular applications have the capability to contain multiple *NgModules* - these modules can be used as containers of related things to support feature modules. Using this capability improves the code organization. Keeping related things together simplifies maintenance and may reduce dependencies. The Angular CLI will also update the module when you use the CLI to add new components.

```
1 ng generate module features/<MY-FEATURE-MODULE>
```

> Using modules to organize related code elements is a great strategy because it allows you to quickly identify things that do not belong. When this happens it is necessary to find the right home (location). This may require some refactoring or creating a new module. You may also identify things that are shared by multiple modules/consumers.

One of the major concerns is long-term support for this architecture. Usually, the services take on too many concerns and responsibilities. This is due to the fact that there are still a lot of concerns (validation, business logic, state management, and data access via HTTP) to implement, and currently the most logical choice is a service within the feature module. These services are also tightly-coupled with the presentation layer's components and they also provide the business logic and data access duties. They may also be responsible for UI state management. Things can get complicated fast depending on how big or complex the application is. Due to the fact that so much is implemented in these services, your team may experience more merge conflicts during development cycles.

Each identified feature may contain:

- feature-related components
- feature routes
- feature-related services (public); the services will most likely be provided in the application root as globally scoped.
    - by default services are global using the default: `providedIn: "root"` configuration.

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

@Injectable({
  providedIn: "root"
})
export class SampleService {}
```

The Pros and Cons of Feature Modules

There are more benefits to this architecture as compared to a single AppModule approach. The best thing about this architecture is the thinking involved to organize the code into the feature modules. This type of thinking will be the basis for implementing additional layers in more complex and refined designs.

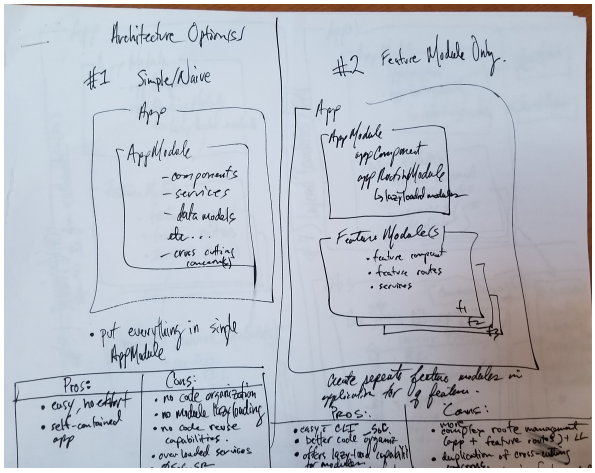
Pros:

- easy to create feature modules with Angular CLI
- better code organization capabilities
- allows for lazy-loaded modules (performance gains)
- services could be scoped to the specific module
    - allows for the use of encapsulated services within the feature module
    - other services can be globally provided in the application root.

Cons:

- more complex route management
    - application routes, feature routes, and lazy-loaded routes
- duplication of cross-cutting concerns (assuming no use of library projects)

- overloaded services in feature modules
    - performing the responsibilities that could be implemented in separate layers
- architecture does not provide or allow for extension



Simple/Default Angular architecture.

Most Angular teams are using this architecture for many production enterprise applications. It will become more difficult to maintain as more features are added. The lack of consideration for cross-cutting concerns is alarming. This architecture begins to provide better code organization - however, it does not provide many long-term benefits. Initial development may make product owners happy, but as time goes on, the amount of effort and time to fix broken things and/or add new features will become a serious issue.

How does the application provide a consistent implementation for:

- logging
- error handling
- configuration
- notifications; difficult to provide information to users of the application. Messages from the Web API may never get to the user
- HTTP/API calls - each call is a snow-flake with special handling, RxJS, inconsistent error handling for both Network and JavaScript sourced issues.

10.3 Core Domain Service Architecture (#3)

A domain service approach starts by a strong separation between the presentation and the business layers of the application. This architecture is going to take advantage of additional Angular and Typescript capabilities. The most important capability that supports the Domain Service Architecture is the new Angular ***library*** project type. Instead of creating new modules within the *application* project for implementation of *core domain* services - you create separate library projects.

> Note: Requires the use of Angular 6 or greater and the use of library project types.

There are some key characteristics of this approach. The burden of the business layer is no longer a concern of the presentation layer - isn't that great news? The feature modules can simply focus on the presentational aspects of the feature. The business side of things is now handled exclusively by the new library project for the specified core domain service. You might be thinking that this sounds like more work and more code. Remember, that the application has the same concerns - we are just implementing them in a different location and simplifying the concerns of the presentation layer.

The actual service of the (core domain) is *not* provided by the library project. Library projects typically do not provide public services that they contain as a convention/practice. The domain service is publicly accessible by consumers of the library. It is most common that the consuming application will provide the core domain service globally within the application - this allows it to be injected into any *presentational services* (within application feature modules) and/or any other *core domain services* that would need its services.

> Note: Most of our discussion thus far is about horizontal application layers. However, with the introduction of feature modules and core domain services we now have verticals that represent features and core

> domain items of the application. These *verticals* are participants in the horizontal layered-architectural approach.

Again, we can take advantage of the Angular CLI or Nrwl.io Nx Workspace enhancements.

```
1 ng generate library <NAME-OF-YOUR-LIBRARY>
```

Library projects are the magic that makes this architecture approach a reality. We could create libraries and JavaScript packages before. However, the overhead and developer workflow were not very efficient. Angular version 6 introduced library projects and the new Workspace as a monorepo. Therefore, we now have a single environment to develop reusable libraries and to integrate them into other applications and library projects. Amazing!!!

> Pro Tip: The Nrwl.io Nx Workspace has some advanced library project management features. As well as CLI options to create libraries configured to publish to NPM.

Pros and Cons of the Core Domain Service

A Core Domain Service approach is slightly more complex - taking advantage of advanced features of Angular and the new Angular Workspace (available since version 6). This is a great approach for large enterprise applications with many features and core domain items (e.g., products, shipments, orders, customer accounts, etc.). It is more sophisticated in terms of code organization and has the potential for additional layer implementations within each core domain service.

Pros:

- Best separation of concerns between business and presentational layers.
- Business logic is encapsulated within the library project (module) and operations are exposed via a service API
- No HTTP or data access originating from the presentation layer

    - The presentational services use the new core domain service to retrieve/persist data
- Easier unit/specification testing of UI layer elements
- Core domain services (as libraries) can be shared and reused by other libraries (i.e., other core domain services) and by other application projects (e.g., consumer, administrative, and B2B applications) simultaneously.

Cons:

- Need to use new Angular *library* project type (not really difficult)
- Need to define a service as an API that can be consumed by other applications and/or libraries.
- Although code organization is improved, the services may still be overloaded with many concerns (no UI concerns) but other business layer concerns:
    - data validation
    - business rule processing
    - data access
    - handling HTTP responses and errors
    - providing notification messages to consumers (presentational layer services)

10.4 Full-Layered Architecture (#4)

The *Full-Layered Architecture* approach builds upon the Core Domain Service architecture. It will continue to use core domain services implemented as library projects in the Angular Workspace. This architecture will include some additional layers that implement the remainder of the common application concerns. These remaining layers are encapsulated within each core domain service (basically Business Logic and Data Access layers).

This approach will reset the concerns of the core domain service. The core domain service now becomes a true API that receives requests and returns response - there is no business logic or implementation of the logic within any of the core domain services. Each core domain service will now interact with a new *business provider service* that coordinates the execution of business operations.

> Note: The new concern of the core domain service may act as a `controller` familiar to developers of Web APIs. Typically, they are are a pass-through to the underlying services that perform all of the heavy lifting and business logic. The take the request and return a response.
>
> It may also be considered a true service facade. It provides the interface or end-points to the specific domain operations. The consumers of the service/API do not know how the work is done - it just uses the service for what it needs. Simple.

Processing and implementing business logic is a key part of any serious application - and even more important to enterprise-level applications. The remaining layers could be a book of themselves. In fact, there are many books on each topic separately. Google the term business objects to see how much information is out there on the subject of business objects alone.

1. Business Logic
2. Data Access

Business Logic

About 15 years ago, I was given a very special role on a big project for one of our major clients. I was a consultant at the time, specializing in .NET/C# and ASP.NET applications. We were building a new version of a real estate listing application. It would need to retrieve and display all of the information required by agents and potential home buyers. It was a web-based application with many sets of XML Web Services - JSON was just beginning to emerge and was not yet standardized (2013).

My job was to write and implement unit tests for all of the business logic of the application. It was an important job and I was very excited. I knew all of the reasons why you need to do unit testing. It was my chance to shine. Instead of sunshine, I got rain and lots of it. More like a thunderstorm. You see, there were about 20 developers on the project and each one of them implemented the business logic differently. They were all snowflakes - each class, each method was so different from the others. There was no consistency in the implementation of business logic. There were hard-coded dependencies and instantiations of objects that were difficult to fake or mock. It was unrealistic.

At the time, I was an impressive ASP.NET developer and loved the framework and patterns of the ASP.NET page lifecycle. There were pre and post-processing events and hooks to do anything you required. Additionally, there were very conventional mechanisms that were very reliable. Consistency was baked into the very ASP.NET framework. It had to dynamically data-driven web pages and deliver the HTML as a response. The process was elegant. I wondered why we couldn't implement business logic in the same manner: consistent, conventional, with pre and post-processing events and hooks?

I started a quest to learn the patterns and the mechanism to implement business logic in the same conventional way with the intent of having 100% unit-testable code that was also extensible and maintainable. Was this even attainable? It consumed me for a long time, months. And for many years after that, I was able to create frameworks for .NET and Typescript applications that provided an elegant way to implement business logic. You might think that building an application framework for business logic

would be very complicated. At some point, I was able to review the actual source code of the ASP.NET Page Lifecycle. I learned that the *template method* pattern was used. I combined this pattern with a *composite* pattern and developed a rule engine.

The *Business Action* and *Rule Engine* frameworks were developed and put into use in many consultant and enterprise level projects. Since then, I have had 3 major releases of the framework and even ported it to Typescript. I was able to use it on a suite of 6 Angular applications and several cross-cutting concern libraries.

Here is a list of some of the features that the business action framework can do to implement business logic:

- business logic as business actions
    - discreet units of work that implement and execute a specific action
    - may include data/input validation
    - business rule processing
    - business actions will use Data Repository services for data operations

Let's briefly show some of the code for the business logic implementation.

Course Core Domain Service

The main entry point of the *Course* domain service provides the APIs for the consumers. It is showing some advanced features:

- implements a base service class
- has a logging service injected into the constructor
    - this is a cross-cutting concern library
- makes use of generics to indicate the response type (model)
- notice that the service delegates the request to a `BusinessProviderService`
    - the business provider coordinates the request
    - the business provider service is scoped to the library and is not accessible outside of the domain service library project

Note: the code has been reduced to show relevant items.

```
import { Injectable, Inject } from "@angular/core";
import { ServiceBase, ApiResponse } from "@angularlicious/foundation";
import { LoggingService } from "@angularlicious/logging";
import { Observable } from "rxjs";
import { BusinessProviderService } from "./business/business-
provider.service";
import { Course } from "@angularlicious/lms-common";

/**
 * The [CoursesService] is a member of the core domain business logic
implementation
 * for [Courses]. It provides the API for all course domain operations. The
domain
 * module is for business logic implementation only.
 *
 * Note: there are no UI concerns in this module or service.
 *
 * This service is provided at the scope of a feature module within the
application.\
 This
 * service has a dependency on the [BusinessProviderService] - which is
internal and\
 scoped
 * to the business domain module. The [LmsBusinessCoursesModule] provides
this servi\
ce and
 * all other internal/dependency services for the business domain
implementation.
 */
@Injectable()
export class CoursesService extends ServiceBase {
  constructor(
    @Inject(BusinessProviderService)
    private businessProvider: BusinessProviderService,
    loggingService: LoggingService
  ) {
    super("CoursesService", loggingService);
  }

  addCourse<T>(course: Course): Observable<T> {
    return this.businessProvider.addCourse<T>(course);
  }
}
```

Business Provider Service

The *BusinessProviderService* is the entry point into the business logic layer of the application. It is implemented as an Angular Service (it itself is Injectable).

- is a service that is scoped to the core domain service library - not accessible to outside consumers of the library

- it is an injectable service
- it has the specific data access service injected into its constructor
    - this data access service is public within the library and allows the business actions to use it for data operations
- has its own base class - takes advantage of class inheritance
- notice the `Course` model comes from a common core domain service library in the workspace

All of the business logic operations are executed using *Actions*. The *AddCourseAction* performs the operation of adding a new course to the application. Most actions are implemented with the conventional 3 lines of code.

1. initialize the action with the required inputs
2. call the `Do()` method to execute the action
3. return the response - an Observable that is part of the Reactive Programming paradigm.

```
 1 import { BusinessProviderBase } from "@angularlicious/foundation";
 2 import { LoggingService } from "@angularlicious/logging";
 3 import { AddCourseAction } from "./actions/add-course.action";
 4 import { FirestoreCourseRepositoryService } from "./firestore-course-
repository.serv\
 5 ice";
 6 import { Course } from "@angularlicious/lms-common";
 7
 8 /**
 9  * This is the coordinator of business operations for the core domain
module. It will
10  * compose business operations using one or more [Business Actions].
11  */
12 @Injectable()
13 export class BusinessProviderService extends BusinessProviderBase {
14   constructor(
15     @Inject(FirestoreCourseRepositoryService)
16     public apiService: FirestoreCourseRepositoryService,
17     loggingService: LoggingService
18   ) {
19     super("BusinessProviderService", loggingService);
20   }
21
22   addCourse<T>(course: Course) {
23     const action = new AddCourseAction<T>(course);
24     action.Do(this);
25     return action.response;
26   }
27 }
```

AddCourseAction: Business Action

As mentioned previously, a business action is a class that executes a single unit of work representing a business logic operation. An application can have as many as they need to implement business logic concerns. One of the main advantages of using a class is that it can participate in Object-Oriented behaviors like inheritance, abstraction, and polymorphism. It also encapsulates the implementation details of the business logic - which makes it a great candidate for unit testing. By the way, I have had over 90% unit test coverage of large applications with hundreds of actions.

- the actions extends from a base class
    - takes advantage of inheritance and polymorphism
- uses a constructor to pass in the input for the action
    - notice that there are no other cross-cutting concerns polluting the constructor
- cross-cutting concerns are injected using property injection in the `Do()` method
- makes use of the template method pattern
    - one of the methods in the pipeline of an action is `preValidation` - used to manage the execution of rules for the action
    - if there are no rule violations, the action pipeline will call the `performAction()` method to execute the specified business logic.

> Note: There will a chapter dedicated to business actions and business logic in the book. This information is to provide a sense of how business logic can be implemented.

```
1  import { BusinessActionBase } from "./business-action-base";
2  import { CourseIsValidRule } from "../rules/course-is-valid.rule";
3  import { Course } from "@angularlicious/lms-common";
4
5  export class AddCourseAction<T> extends BusinessActionBase<T> {
6    constructor(private course: Course) {
7      super("AddCourseAction");
8    }
9
10   preValidateAction() {
11     this.validationContext.addRule(
12       new CourseIsValidRule(
13         "CourseIsNotNull",
14         "The course information is not valid.",
```

```
        this.course,
        this.showRuleMessages
      )
    );
  }

  performAction() {
    this.response = this.businessProvider.apiService.addCourse<T>
(this.course);
  }
}
```

Notice, that the action used a *validationContext* to add rules to evaluate. The rule engine has a pre-defined set of rules that can be used to compose more complex rules. The rule engine allows for single and/or complex rules to be defined. In this example, we created a custom rule using the rule engine framework. All rules are constructed and executed the same way. The rule engine is conventional and provides a consistent mechanism to implement business rules for enterprise applications.

Data Access

We left off in the Business Action discussion with the action using the *apiService* to perform a data operation. The *Data Access Layer* is the last layer that we will implement in our application concerns list. This specific data access service is using Firebase/Firestore as its data provider.

- Data Repository services injected into each *business provider service* to provide
  - data access
  - handling HTTP responses and errors

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

import { AngularFirestore,
    AngularFirestoreDocument,
    AngularFirestoreCollection } from '@angular/fire/firestore';
import { LoggingService,
    Severity } from '@angularlicious/logging';
import { Observable, from, of } from 'rxjs';
import { Course, Video } from '@angularlicious/lms-common';
import { map } from 'rxjs/operators';

@Injectable()
export class FirestoreCourseRepositoryService extends ServiceBase {
  private COURSES = 'courses';
  private VIDEOS = 'videos';
```

```
17
18    private courseDocument: AngularFirestoreDocument<Course>;
19    private course$: Observable<any> = new Observable<Course>(null);
20
21    private courseCollection: AngularFirestoreCollection<Course>;
22    private courses$: Observable<any> = new Observable<Course[]>(null);
23
24    constructor(private firestore: AngularFirestore,
25      loggingService: LoggingService) {
26      super('FirestoreCourseRepositoryService', loggingService);
27    }
28
29    public addCourse<T>(course: Course): Observable<T> {
30      this.courseCollection = this.firestore.collection<Course>(this.COURSES);
31      this.courseCollection
32        .add({ ...course })
33        .then(response => {
34          this.loggingService.log(this.serviceName, Severity.Information,
`Preparing t\
35 o process response to adding a new course to the database.
${course.title}`);
36          if (response) {
37            return this.retrieveCourse(response.id);
38          }
39        })
40        .catch(error => this.handleError(error));
41      return of(null);
42    }
43
44    public retrieveCourse<T>(courseId: string): Observable<T> {
45      this.courseDocument = this.firestore.doc(`${this.COURSES}/${courseId}`);
46      this.course$ = this.courseDocument.valueChanges();
47
48      return this.course$;
49    }
```

The Pros and Cons of Full-Layered Architecture

A Full-Layered Architecture implements specific layers for business logic and data access operations.

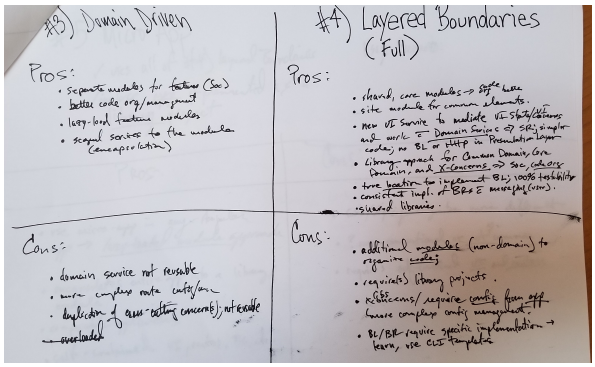
Pros:

- All of the pros of the core domain service architecture
- The feature module service concerns are specific to UI
    - is a mediator between UI concerns and the core domain service
- All application concerns are implemented in a specific layer
- Tighter control on how business logic is implemented
    - uses the business action framework
    - uses a rule engine for business rules and input validation
- All business logic actions are consistent and 100% unit testable.

- All business logic is encapsulated within the domain service library
- Can create new Schematics for Angular CLI to create: actions and rules

Cons:

- More layers require the discipline to manage all layers to ensure that each layer's concern is respected
- Need to learn the behaviors of the business action framework
- Need to learn the behaviors of the business rule engine



Domain-driven architecture - pros and cons.

11 Setup Reference Application

11.1 Nx Workspace

Create a new Angular Workspace with super powers using Nx from Nrwl.io. Use the npx CLI command to create a new Angular Workspace.

```
npx create-nx-workspace workspace --npm-scope=valencia
```

```
npx create-nx-workspace workspace --npm-scope=valencia
? What to create in the new workspace empty             [an empty workspace]
? CLI to power the Nx workspace       Angular CLI  [Extensible CLI for
Angular appli\
cations. Recommended for Angular projects.]
Creating a sandbox with Nx...
warning @angular/cli > universal-analytics > request@2.88.2: request has
been deprec\
ated, see https://github.com/request/request/issues/3142
warning @angular/cli > universal-analytics > request > har-validator@5.1.5:
this lib\
rary is no longer supported
warning Your current version of Yarn is out of date. The latest version is
"1.22.5",\
 while you're on "1.22.1".
new workspace "--npm-scope=valencia" --preset="empty" --interactive=false --
collecti\
on=@nrwl/workspace
CREATE workspace/nx.json (259 bytes)
CREATE workspace/tsconfig.json (509 bytes)
CREATE workspace/package.json (1156 bytes)
CREATE workspace/README.md (2694 bytes)
CREATE workspace/.editorconfig (245 bytes)
CREATE workspace/.gitignore (503 bytes)
CREATE workspace/.prettierignore (74 bytes)
CREATE workspace/.prettierrc (26 bytes)
CREATE workspace/angular.json (96 bytes)
CREATE workspace/apps/.gitkeep (1 bytes)
CREATE workspace/libs/.gitkeep (0 bytes)
CREATE workspace/tools/tsconfig.tools.json (218 bytes)
CREATE workspace/tools/schematics/.gitkeep (0 bytes)
CREATE workspace/.vscode/extensions.json (164 bytes)
√ Packages installed successfully.
    Directory is already under version control. Skipping initialization of
git.
PS D:\development\github\reference-loans>
```

Nrwl Angular Schematics

Install the following package for the workspace environment to provide the Angular schematics.

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

11.2 Create Application Project

Use the Angular CLI and the prompted options to create a new application projects for the workspace.

```
ng g application reference-contacts
? Which stylesheet format would you like to use? SASS(.scss)  [ http://sass-lang.com\
   ]
? Would you like to configure routing for this application? Yes
CREATE jest.config.js (250 bytes)
CREATE tslint.json (2311 bytes)
CREATE apps/reference-contacts/tsconfig.json (97 bytes)
CREATE apps/reference-contacts/src/favicon.ico (15086 bytes)
CREATE apps/reference-contacts/browserslist (429 bytes)
CREATE apps/reference-contacts/tsconfig.app.json (163 bytes)
CREATE apps/reference-contacts/tslint.json (248 bytes)
CREATE apps/reference-contacts/src/index.html (339 bytes)
CREATE apps/reference-contacts/src/main.ts (375 bytes)
CREATE apps/reference-contacts/src/polyfills.ts (2836 bytes)
CREATE apps/reference-contacts/src/styles.scss (80 bytes)
CREATE apps/reference-contacts/src/assets/.gitkeep (0 bytes)
CREATE apps/reference-contacts/src/environments/environment.prod.ts (51 bytes)
CREATE apps/reference-contacts/src/environments/environment.ts (662 bytes)
CREATE apps/reference-contacts/src/app/app.module.ts (417 bytes)
CREATE apps/reference-contacts/src/app/app.component.html (3017 bytes)
CREATE apps/reference-contacts/src/app/app.component.spec.ts (1053 bytes)
CREATE apps/reference-contacts/src/app/app.component.ts (226 bytes)
CREATE apps/reference-contacts/src/app/app.component.scss (2088 bytes)
CREATE apps/reference-contacts/jest.config.js (369 bytes)
CREATE apps/reference-contacts/tsconfig.spec.json (233 bytes)
CREATE apps/reference-contacts/src/test-setup.ts (30 bytes)
CREATE apps/reference-contacts-e2e/tslint.json (97 bytes)
CREATE apps/reference-contacts-e2e/cypress.json (432 bytes)
CREATE apps/reference-contacts-e2e/tsconfig.e2e.json (188 bytes)
CREATE apps/reference-contacts-e2e/tsconfig.json (137 bytes)
CREATE apps/reference-contacts-e2e/src/fixtures/example.json (80 bytes)
CREATE apps/reference-contacts-e2e/src/integration/app.spec.ts (424 bytes)
CREATE apps/reference-contacts-e2e/src/plugins/index.js (832 bytes)
CREATE apps/reference-contacts-e2e/src/support/app.po.ts (47 bytes)
CREATE apps/reference-contacts-e2e/src/support/commands.ts (1068 bytes)
CREATE apps/reference-contacts-e2e/src/support/index.ts (599 bytes)
UPDATE package.json (1988 bytes)
UPDATE angular.json (4586 bytes)
UPDATE nx.json (417 bytes)
\ Installing packages...
\
                                        \
√ Packages installed successfully.
```

Configure Chrome Launch Debugger

Configure the Chrome launch debugger using the command pallete.

Ctrl + P
Debug: Open launch.json

- update the port to `4200`
- update the name to the target project

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Reference Contacts",
      "url": "http://localhost:4200",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

Build and Serve

Use the CLI command below to build and serve the application. You can use the Chrome debugger to view the application using the `launch` configuration above - click `F5` to launch.

ng serve

Install Spark Design System

Install the required packages in the workspace - they will now be available to all application projects. Nice!

```
yarn add @sparkdesignsystem/spark @sparkdesignsystem/spark-angular -D
```

The following warning was displayed after installing the packages.

```
$ ngcc --properties es2015 browser module main --first-only --create-ivy-entry-points
Warning: Entry point '@sparkdesignsystem/spark-angular' contains deep imports into '\
D:/development/github/reference-loans/workspace/node_modules/lodash/uniqueId'. This \
```

```
4 is probably not a
5 problem, but may cause the compilation of entry points to be out of order.
```

Update the application project's `styles.scss` file with the `import` of the spark SCSS file.

> Note: If you are using an Angular Workspace the path will be different from the site documentation.

```scss
1 /* You can add global styles to this file, and also import other style files
*/
2 @import './../../../node_modules/@sparkdesignsystem/spark/_spark.scss';
```

11.3 Application Modules and Configuration

The application module is the main entry point into the application. It has the responsibility to
coordinate all of the required modules for the application. Therefore, instead of overloading the `AppModule` to import, configure, initialize, and provide services - we organize our modules for the application into the following, each with specific responsibilities.

Here are some examples of modules in the sample application.

> Learn more about NgModules at:
> https://angular.io/guide/ngmodules#ngmodules

Module	Responsibility
AppModule	The `AppModule` coordinates all of the required modules for the application. It is the responsible for bootstrapping the application.
AppRoutingModule	The `AppRoutingModule` provides the routing definition for the application. It can lazy-load any modules via a specific route.
BrowserAnimationsModule	Use to provide Angular animations - the illusion of motion for HTML elements. Well-designed animations can make your application more fun and easier to use, but they aren't just cosmetic. Animations can improve your app and user experience. More information at Angular.io
BrowserModule	Exports required infrastructure for all Angular apps. Included by default in all Angular apps created with the CLI new command. Re-exports CommonModule and ApplicationModule, making their exports and providers available to all apps. More information at Angular.io
CoreModule	The `CoreModule` provides the organization of other modules that are related to the specific domain of the application.
CrossCuttingModule	The `CrossCuttingModule` module organizes the cross-cutting concerns into a single module to provide initialization and configuration, as well as providing global services to the application for: configuration, logging and error handling.

Module	Responsibility
SharedModule	Use the `SharedModule` to manage Angular and other 3rd-party modules/libraries/packages used by the application. For example: MaterialDesignModule, RouterModule, ReactiveFormsModule

The reference application has a focus on a clean architecture with clearly-defined boundaries. The architecture also takes advantage of the SOLID principles using Object-Oriented Programming principles and patterns. The *presentation layer* relies on the list of modules above. I think it is interesting that before we even create a single domain-specific component or service that there is already so much code for the application. Let's start the journey by exploring these modules that really setup and bootstrap the entire application.

Core Module

The *CoreModule* provides a single instance module (singleton) for the Angular application for modules that only require a single instance.

```
 1 import { NgModule, SkipSelf, Optional } from '@angular/core';
 2 import { CommonModule } from '@angular/common';
 3 import { BrowserModule } from '@angular/platform-browser';
 4 import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
 5
 6 const MODULES = [CommonModule, BrowserModule, BrowserAnimationsModule];
 7
 8 @NgModule({
 9   declarations: [],
10   imports: [...MODULES],
11   exports: [...MODULES],
12 })
13 export class CoreModule {
14   constructor(@Optional() @SkipSelf() core: CoreModule) {
15     if (core) {
16       throw new Error(`Application requires single instance of
CoreModule.`);
17     }
18   }
19 }
```

Shared Module

The *SharedModule* is a convenience module to group a set of common modules that will be used by other application modules.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { RouterModule } from '@angular/router';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

const MODULES = [CommonModule, FormsModule, HttpClientModule,
HttpClientModule, Reac\
tiveFormsModule, RouterModule];
@NgModule({
  declarations: [],
  imports: [...MODULES],
  exports: [...MODULES],
})
export class SharedModule {}
```

Cross-Cutting Module

The *CrossCuttingModule* allows for a set of application services to be injected into the application's global DI container. All of the cross-cutting concerns are accessed via the singleton instance available to all other application modules from the global *injector*.

> This module is a must for enterprise-level application to provide cross-cutting
> concerns:
>
> - configuration
> - logging
> - error handling
> - notifications

The application has the capability to load different *writer* providers for the logging service. This allows logging in the console during development and logging to a centralized repository when deployed to production environments.

```
import { CommonModule } from '@angular/common';
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { APP_INITIALIZER, ErrorHandler, ModuleWithProviders, NgModule } from
'@angul\
ar/core';
import { ConfigurationModule, ConfigurationService, ConfigurationContext }
from '@an\
gular-architecture/configuration';
import { ErrorHandlingModule, ErrorHandlingService } from '@angular-
architecture/err\
or-handling';
import { HttpErrorInterceptor } from '@angular-architecture/http-service';
import { LogglyWriter, ConsoleWriter, LoggingModule, LoggingService } from
'@angular\
-architecture/logging';
import { NotificationService } from '@angular-architecture/notification';
// tslint:disable-next-line:nx-enforce-module-boundaries
import { AppConfig } from 'apps/reference-contacts/src/config/app-config';
import { LogglyService } from 'ngx-loggly-logger';

/**
 * The factory function to initialize the configuration service for the
application.
 * @param configService
 */
export function initializeConfiguration(configContext: ConfigurationContext,
configS\
ervice: ConfigurationService) {
  return () => {
    configService.settings = configContext.config;
    return configService;
  };
}

/**
 * The factory function to initialize the logging service and writer for the
 * application.
 *
 * @param loggingService
 * @param consoleWriter
 */
export function initializeLogWriter(loggingService: LoggingService,
consoleWriter: C\
onsoleWriter) {
  return () => {
    return consoleWriter;
  };
}

@NgModule({
  declarations: [],
  imports: [CommonModule, ConfigurationModule, ErrorHandlingModule,
LoggingModule],
  providers: [],
})
export class CrossCuttingModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: CrossCuttingModule,
      providers: [
        {
```

```
          provide: ConfigurationContext,
          useValue: { config: AppConfig },
        },
        ConfigurationService,
        LoggingService,
        ConsoleWriter,
        LogglyWriter,
        {
          provide: APP_INITIALIZER,
          useFactory: initializeConfiguration,
          deps: [ConfigurationContext, ConfigurationService],
          multi: true,
        },
        ConsoleWriter,
        LogglyService,
        LogglyWriter,
        LoggingService,
        {
          provide: ErrorHandler,
          useClass: ErrorHandlingService,
          deps: [ConfigurationService, LoggingService],
        },
        {
          provide: APP_INITIALIZER,
          useFactory: initializeLogWriter,
          deps: [LoggingService, ConsoleWriter, LogglyWriter],
          multi: true,
        },
        {
          provide: HTTP_INTERCEPTORS,
          useClass: HttpErrorInterceptor,
          multi: true,
        },
        NotificationService,
      ],
    };
  }
}
```

App Module

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { RouterModule } from '@angular/router';

import { SparkAngularModule } from '@sparkdesignsystem/spark-angular';
import { CrossCuttingModule } from './modules/cross-cutting/cross-cutting.module';
import { SharedModule } from './modules/shared/shared.module';
import { CoreModule } from './modules/core/core.module';

@NgModule({
  declarations: [AppComponent],
  imports: [
    CoreModule,
    SharedModule,
    CrossCuttingModule.forRoot(),
```

```
    BrowserModule,
    RouterModule.forRoot([], { initialNavigation: 'enabled' }),
    SparkAngularModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Configuration

```
yarn add ngx-loggly-logger
```

11.4 Code Formatting

The Angular Workspace contains a single configuration for code formatting. This ensures that all project code is formatted against a predefined set of formatting rules. Code formatting is enforced during the commit process using a `pre-commit` hook.

> This allows a team of several developers to maintain consistent code formatting.

Install the following packages.

```
"husky": "^4.2.5",
"pretty-quick": "^3.0.0",
```

```
yarn add husky pretty-quick
```

Add the `husky` configuration in the root of the *package.json* file.

```
"husky": {
    "hooks": {
      "pre-commit": "pretty-quick --staged  --pattern=\"**/*.*(ts|json)\" --verbose"
    }
  },
```

Verify your lint configuration.

```
yarn lint
```

Fix any formatting issues.

11.5 Create Angular Library Projects

The Angular Workspace supports *library* projects to share and reuse code throughout the Workspace in other application and library projects. This Workspace is enhanced with the Nrwl.io Nx developer tools - an Nx Workspace allows for libraries to be used without the overhead of:

1. building/compiling packages
2. rebuilding dependency packages
3. managing package versions
4. publishing packages
5. consuming packages
6. managing consumed package versions

> Any JavaScript solution with multiple custom library/packages will require
> a lot of maintenance to build and consume changes. Nx provides a no build/no install solution.

```
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=actions
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=components
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=configuration
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=error-handling
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=foundation
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=http-service
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=logging
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=notification
ng generate @nrwl/angular:library --publishable --simpleModuleName --style=scss --pr\
efix=aa --name=rules-engine
```

11.6 Create Micro-Frontend (Application Project)

Create a new library project that will have the responsibility of a *Contacts* UI. Using a library project type for UI implements provides an effective and efficient strategy to share and reuse UI code as *micro-applications* or *micro-frontends*. A micro-frontend allows for the following:

- multiple application projects can reuse a single micro-application
- application projects are composable using micro-applications as aggregates
    - single-source of truth
    - higher quality
    - no versioning required
    - eliminates scattered code throughout different applications
    - no copy/past coding
    - can be shared/reused by other micro-frontends
- single micro-frontend applications are modular
    - encapsulate UI and specific feature sets
- allows for shared application state, resources, styles, and assets
    - global services: security, logging, configuration
    - SCSS
    - images, fonts, configuration

Generate Micro-Frontend with CLI

Use the Angular CLI to generate a micro-frontend library project.

```
ng g library contactsApp --simple-module-name --
directory=reference/micro-apps --publishable
```

```
 1 ng g library contactsApp --simple-module-name --directory=reference/micro-
apps  --pu\
 2 blishable
 3 ? Which stylesheet format would you like to use? SASS(.scss)  [ http://sass-
lang.com\
 4    ]
 5 CREATE libs/reference/micro-apps/contacts-app/ng-package.json (193 bytes)
 6 CREATE libs/reference/micro-apps/contacts-app/package.json (196 bytes)
 7 CREATE libs/reference/micro-apps/contacts-app/README.md (186 bytes)
 8 CREATE libs/reference/micro-apps/contacts-app/tsconfig.lib.json (414 bytes)
```

```
 9 CREATE libs/reference/micro-apps/contacts-app/tsconfig.lib.prod.json (97
bytes)
10 CREATE libs/reference/micro-apps/contacts-app/src/lib/contacts-app.module.ts
(167 by\
11 tes)
12 CREATE libs/reference/micro-apps/contacts-app/tsconfig.json (129 bytes)
13 CREATE libs/reference/micro-apps/contacts-app/jest.config.js (413 bytes)
14 CREATE libs/reference/micro-apps/contacts-app/tsconfig.spec.json (239 bytes)
15 CREATE libs/reference/micro-apps/contacts-app/src/test-setup.ts (30 bytes)
16 UPDATE angular.json (20567 bytes)
17 UPDATE nx.json (973 bytes)
18 UPDATE tsconfig.json (1277 bytes)
19 √ Packages installed successfully.
```

Add Routing Module to Micro-Frontend

Angular UI modules need routing modules with routes to components. Add a routing module to the micro-frontend library project using the CLI.

```
ng g m appRouting --module=contacts-app.module --
project=reference-micro-apps-contacts-app
```

The routing module is created. The CLI also updates the library module to import the routing module. When the routing module is set up with routes - the micro-frontend application will initialize the routes with a `forChild()` call.

```
1 ng g m appRouting  --module=contacts-app.module --project=reference-micro-
apps-conta\
2 cts-app
3 CREATE libs/reference/micro-apps/contacts-app/src/lib/app-routing/app-
routing.module\
4 .ts (196 bytes)
5 UPDATE libs/reference/micro-apps/contacts-app/src/lib/contacts-app.module.ts
(254 by\
6 tes)
```

Add Components to Micro-Frontend

Run the following CLI commands to create the components in the micro-frontend library project.

> Pro Tip: Install the Nx Console extension for Visual Studio Code. It provides
> a UI to collect and configure a CLI command. You can capture the

command in a
terminal to see the execution results and command details.

```
ng generate @schematics/angular:module --name=landing --project=reference-micro-apps\
 -contacts-app --module=app-routing/app-routing.module --no-commonModule --lintFix --\
route=landing --routing
ng generate @schematics/angular:module --name=list --project=reference-micro-apps-co\
ntacts-app --module=app-routing/app-routing.module --no-commonModule --lintFix --rou\
te=list --routing
ng generate @schematics/angular:module --name=item --project=reference-micro-apps-co\
ntacts-app --module=app-routing/app-routing.module --no-commonModule --lintFix --rou\
te=item --routing
ng generate @schematics/angular:module --name=add --project=reference-micro-apps-con\
tacts-app --module=app-routing/app-routing.module --no-commonModule --lintFix --rout\
e=add --routing
```

See the sample output below for the `contact-item` component:

```
ng generate @schematics/angular:module --name=landing --project=reference-micro-apps\
 -contacts-app --module=app-routing/app-routing.module --no-commonModule --lintFix --\
route=landing --routing <
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing-routing.module\
.ts (348 bytes)
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing.module.ts (290\
 bytes)
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing.component.html\
 (22 bytes)
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing.component.spec\
.ts (635 bytes)
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing.component.ts (\
284 bytes)
CREATE libs/reference/micro-apps/contacts-app/src/lib/landing/landing.component.css \
(0 bytes)
UPDATE libs/reference/micro-apps/contacts-app/src/lib/app-routing/app-routing.module\
.ts (901 bytes)
```

Component UI Services

Each of the UI components will have a designated UI service to coordinate UI/UX state and any data operations with the domain service. The UI service

- data operations
  - handles the response
  - manages the publishing of data using Async Observables
  - manages the publishing of UI events using Async Observables

```
1 ng generate @schematics/angular:service --name=AddContactUI --
project=reference-micr\
2 o-apps-contacts-app --lintFix --skipTests --no-interactive --dry-run <
3
4 CREATE libs/reference/micro-apps/contacts-app/src/lib/add-contact-
ui.service.ts (141\
5  bytes)
```

The UI service extends from the `ServiceBase` class to provide common behavior for Angular services. A *Logging* provider is injected into the service via the constructor.

```
 1 import { Injectable } from '@angular/core';
 2 import { ServiceBase } from '@angular-architecture/foundation';
 3 import { LoggingService } from '@angular-architecture/logging';
 4
 5 @Injectable({
 6   providedIn: 'root',
 7 })
 8 export class AddContactUIService extends ServiceBase {
 9   constructor(loggingService: LoggingService) {
10     super('AddContactUIService', loggingService);
11   }
12 }
```

Add Routing to Micro-Frontend

Update the `AppRoutingModule` with routes and register the routes using the `RouterModule.forChild(routes)` call in the module's *import* section.

> Pro Tip: Use the Nx Console mentioned in the Add Components to Micro-Frontend section. This will automatically update application routing module with lazy-loaded routes to the SCAMs

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: 'landing', loadChildren: () =>
import('../landing/landing.module').then(m \
=> m.LandingModule) },
  { path: 'list', loadChildren: () => import('../list/list.module').then(m
=> m.List\
Module) },
  { path: 'item/edit:id', loadChildren: () =>
import('../item/item.module').then(m =\
> m.ItemModule) },
  { path: 'add', loadChildren: () => import('../add/add.module').then(m =>
m.AddModu\
le) },
];

@NgModule({
  declarations: [],
  imports: [CommonModule, RouterModule.forChild(routes)],
})
export class AppRoutingModule {}
```

Update the host application to load the micro-frontend application using lazy-loading.

```
import { NgModule } from '@angular/core';
import { Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'contacts/landing',
    loadChildren: () => import('@angular-architecture/reference/micro-
apps/contacts-\
app').then(m => m.ContactsAppModule),
  },
];

@NgModule({
  declarations: [],
  imports: [],
})
export class AppRoutingModule {}
```

11.7 Create Domain Library Projects

A domain library encapsulates all of the business logic for the specified *service*.

```
Executing task: ng generate @nrwl/angular:library --name=contactsService --
style=scs\
s --directory=reference/domain --publishable --simpleModuleName <

CREATE libs/reference/domain/contacts-service/ng-package.json (193 bytes)
CREATE libs/reference/domain/contacts-service/package.json (196 bytes)
CREATE libs/reference/domain/contacts-service/README.md (186 bytes)
CREATE libs/reference/domain/contacts-service/tsconfig.lib.json (414 bytes)
CREATE libs/reference/domain/contacts-service/tsconfig.lib.prod.json (97
bytes)
CREATE libs/reference/domain/contacts-service/tslint.json (254 bytes)
CREATE libs/reference/domain/contacts-service/src/index.ts (47 bytes)
CREATE libs/reference/domain/contacts-service/src/lib/contacts-
service.module.ts (17\
1 bytes)
CREATE libs/reference/domain/contacts-service/tsconfig.json (129 bytes)
CREATE libs/reference/domain/contacts-service/jest.config.js (413 bytes)
CREATE libs/reference/domain/contacts-service/tsconfig.spec.json (239 bytes)
CREATE libs/reference/domain/contacts-service/src/test-setup.ts (30 bytes)
UPDATE angular.json (22116 bytes)
UPDATE nx.json (1038 bytes)
UPDATE tsconfig.json (1383 bytes)
√ Packages installed successfully.
```

Common Domain Library

```
ng generate @nrwl/angular:library --name=common --style=scss --
directory=reference/d\
omain --publishable --simpleModuleName <

CREATE libs/reference/domain/common/ng-package.json (183 bytes)
CREATE libs/reference/domain/common/package.json (186 bytes)
CREATE libs/reference/domain/common/README.md (166 bytes)
CREATE libs/reference/domain/common/tsconfig.lib.json (414 bytes)
CREATE libs/reference/domain/common/tsconfig.lib.prod.json (97 bytes)
CREATE libs/reference/domain/common/tslint.json (254 bytes)
CREATE libs/reference/domain/common/src/index.ts (37 bytes)
CREATE libs/reference/domain/common/src/lib/common.module.ts (162 bytes)
CREATE libs/reference/domain/common/tsconfig.json (129 bytes)
CREATE libs/reference/domain/common/jest.config.js (393 bytes)
CREATE libs/reference/domain/common/tsconfig.spec.json (239 bytes)
CREATE libs/reference/domain/common/src/test-setup.ts (30 bytes)
UPDATE angular.json (24794 bytes)
UPDATE nx.json (1133 bytes)
UPDATE tsconfig.json (1525 bytes)
√ Packages installed successfully.
```

```
ng generate @nrwl/workspace:library --name=common --
directory=reference/domain <

CREATE libs/reference/domain/common/tslint.json (97 bytes)
CREATE libs/reference/domain/common/README.md (196 bytes)
CREATE libs/reference/domain/common/tsconfig.json (129 bytes)
CREATE libs/reference/domain/common/tsconfig.lib.json (178 bytes)
CREATE libs/reference/domain/common/src/index.ts (45 bytes)
CREATE libs/reference/domain/common/src/lib/reference-domain-common.ts (0
bytes)
CREATE libs/reference/domain/common/jest.config.js (280 bytes)
CREATE libs/reference/domain/common/tsconfig.spec.json (255 bytes)
UPDATE tsconfig.json (1525 bytes)
UPDATE angular.json (24199 bytes)
UPDATE nx.json (1133 bytes)
```

Contact DTO (Data Transfer Object)

Create a DTO for the contact information.

> Note that the *contact* item also contains an `options` list of numbers.

```
export class ContactDto {
  address1: string;
  address2: string;
  city: string;
  company: string;
  emailAddress: string;
  firstName: string;
  lastName: string;
  options: number[] = [];
  phone: string;
  postalCode: string;
  state: string;
}
```

Add a `Contact` model that contains the same information in the DTO, but also has an identifier for the a specific contact item.

```
import { ContactDto } from './contact.dto';

export class Contact extends ContactDto {
  contactId: string;
}
```

API Service for Domain Library

The library project will require a service as an entry point. Create a new service for the library project - the responsibility of the service is to provide an API for any consumers of *ContractsService*.

```
1 ng generate @schematics/angular:service --name=contacts --project=reference-
domain-c\
2 ontacts-service --lintFix --skipTests <
3 CREATE libs/reference/domain/contacts-service/src/lib/contacts.service.ts
(137 bytes)
```

11.8 Test

Use the following commands to run the test suites in each project

```
yarn run test --project=reference-contacts --watch
yarn run test --project=reference-domain-contacts-service --watch
yarn run test --project=actions --watch
yarn run test --project=components --watch
yarn run test --project=configuration --watch
yarn run test --project=error-handling --watch
yarn run test --project=foundation --watch
yarn run test --project=http-service --watch
yarn run test --project=logging --watchhttps
yarn run test --project=notification --watch
yarn run test --project=rules-engine --watch
```

The *package.json* contains scripts to run different types of tests

```
"test:libs": "ng test --project=actions && ng test --project=components && ng test -\
-project=configuration && ng test --project=error-handling && ng test --project=foun\
dation && ng test --project=http-service && ng test --project=logging && ng test --p\
roject=notification && ng test --project=rules-engine",
"test:reference": "yarn run test --project=reference-contacts && yarn run test --pro\
ject=reference-domain-contacts-service && yarn run test --project=reference-micro-ap\
ps-contacts-app",
```

Use the `--test-file` option to target a specific file.

```
ng test --project=reference-domain-contacts-service --test-file=add-contact.action.s\
pec.ts --watch
```

The following is a the output for a single test file. The *business action* provides the capability to verify the valdiation rules for in the inputs of a data operation.

```
ng test --project=reference-domain-contacts-service --watch
 PASS  libs/reference/domain/contacts-service/src/lib/business/actions/add-contact.a\
ction.spec.ts (8.583s)
  VerifyAccountsAction
    √ should create an instance (48ms)
    √ should contain validation error for MIN LENGTH first name (24ms)
```

```
    √ should contain validation error for MIN LENGTH last name (16ms)
    √ should contain validation error for MIN LENGTH company (16ms)
    √ should contain validation error for MIN LENGTH phone (16ms)
    √ should contain validation error for MIN LENGTH email address (24ms)
    √ should contain validation error for MIN LENGTH postal code (24ms)
    √ should contain validation error for MIN LENGTH state (16ms)
    √ should contain validation error for MIN LENGTH city (16ms)
    √ should contain validation error for MAX LENGTH first name (32ms)
    √ should contain validation error for MAX LENGTH last name (16ms)
    √ should contain validation error for MAX LENGTH company (16ms)
    √ should contain validation error for MAX LENGTH email address (8ms)
    √ should contain validation error for MAX LENGTH phone (24ms)
    √ should contain validation error for MAX LENGTH city (8ms)
    √ should contain validation error for MAX LENGTH state (16ms)
    √ should contain validation error for MAX LENGTH postal code (16ms)
    √ should contain validation error for MAX LENGTH address 2 (8ms)
    √ should contain validation error for MAX LENGTH address 1 (24ms)
    √ should contain validation error for MIN LENGTH address 1 (16ms)
    √ should contain validation error for UNDEFINED address 1 (17ms)
    √ should contain valid VALIDATION CONTEXT (15ms)
    √ should contain email address format validation rule (16ms)

Test Suites: 1 passed, 1 total
Tests:       23 passed, 23 total
Snapshots:   0 total
Time:        11.718s
```

11.9 Application Components

```
1 ng generate @schematics/angular:module --name=modules/site --
project=reference-conta\
2 cts --module=app.module
3 CREATE apps/reference-contacts/src/app/modules/site/site.module.ts (190
bytes)
4 UPDATE apps/reference-contacts/src/app/app.module.ts (854 bytes)
```

Add Site Layout Component

```
 1 ng generate @schematics/angular:component --name=modules/site/layout --
project=refer\
 2 ence-contacts --module=site.module --style=scss --changeDetection=OnPush --
lintFix -\
 3 -skipTests <
 4
 5 CREATE apps/reference-
contacts/src/app/modules/site/layout/layout.component.html (21\
 6  bytes)
 7 CREATE apps/reference-
contacts/src/app/modules/site/layout/layout.component.ts (357 \
 8 bytes)
 9 CREATE apps/reference-
contacts/src/app/modules/site/layout/layout.component.scss (0 \
10 bytes)
11 UPDATE apps/reference-contacts/src/app/modules/site/site.module.ts (422
bytes)
```

Add Site Header Component

```
 1 ng generate @schematics/angular:component --name=modules/header/layout --
project=ref\
 2 erence-contacts --module=header.module --style=scss --changeDetection=OnPush
--lintF\
 3 ix --skipTests <
 4
 5 CREATE apps/reference-
contacts/src/app/modules/header/layout/layout.component.html (\
 6 21 bytes)
 7 CREATE apps/reference-
contacts/src/app/modules/header/layout/layout.component.ts (35\
 8 7 bytes)
 9 CREATE apps/reference-
contacts/src/app/modules/header/layout/layout.component.scss (\
10 0 bytes)
11 UPDATE apps/reference-contacts/src/app/modules/header/header.module.ts (422
bytes)
```

Add Site Footer Component

```
ng generate @schematics/angular:component --name=modules/header/footer --
project=ref\
erence-contacts --module=header.module --style=scss --changeDetection=OnPush
--lintF\
ix --skipTests <

CREATE apps/reference-
contacts/src/app/modules/header/footer/footer.component.html (\
21 bytes)
CREATE apps/reference-
contacts/src/app/modules/header/footer/footer.component.ts (35\
7 bytes)
CREATE apps/reference-
contacts/src/app/modules/header/footer/footer.component.scss (\
0 bytes)
UPDATE apps/reference-contacts/src/app/modules/header/header.module.ts (422
bytes)
```

11.10 Common Module

```
ng generate @nrwl/angular:library --name=common --style=scss --publishable --simpleM\
oduleName <

CREATE libs/common/ng-package.json (156 bytes)
CREATE libs/common/package.json (171 bytes)
CREATE libs/common/README.md (136 bytes)
CREATE libs/common/tsconfig.lib.json (408 bytes)
CREATE libs/common/tsconfig.lib.prod.json (97 bytes)
CREATE libs/common/tslint.json (248 bytes)
CREATE libs/common/src/index.ts (37 bytes)
CREATE libs/common/src/lib/common.module.ts (162 bytes)
CREATE libs/common/tsconfig.json (123 bytes)
CREATE libs/common/jest.config.js (351 bytes)
CREATE libs/common/tsconfig.spec.json (233 bytes)
CREATE libs/common/src/test-setup.ts (30 bytes)
UPDATE angular.json (23365 bytes)
UPDATE nx.json (1078 bytes)
UPDATE tsconfig.json (1439 bytes)
√ Packages installed successfully.
```

12 Reference Application

The reference application demonstrates the principles and patterns of our desired architecture. I will use a simple *Contacts* application. It will display a list of contacts with the ability to add and remove contacts. The goal is to provide a reference that is simple enough to understand but also demonstrates realistic scenarios that show the benefits and capabilities of a clean architecture.

> Please note that the *purpose* of the reference application is to demonstrate the capabilities that we have with Angular, Typescript, and our Workspace environment. Use the reference application as a guide for considering different ways to approach many of the common problems we are trying to solve. Most of what is demonstrated is simply code organization strategies with the purpose of keeping things simple and testable with the goal of delivering high-quality solutions.

The reference application should promote:

- continued exploration of Angular and TypeScript capabilities
- considering reactive programming techniques with RxJS
- thinking about code organization strategies
    - simplification
    - code reuse
    - maintainability
    - testability
    - quality
    - readability
- learning about and using design patterns
    - facade
    - composite
    - template method
    - builder

    - strategy
- thinking about architecture as an ongoing-process to design, plan, and execute our solutions

Remember, that CLEAN Architecture is based on layers where each layer has a single responsibility and each layer has a well-defined boundary for separation of concerns. This boundary is respected through discipline, good code organization, and by following SOLID coding principles. With that said, there are going to be a lot of layers.

CLEAN Architectur e Layers

12.1 Essential Modules

Modules provide modularity. There is no dispute that modular-design has many benefits. The `NgModule` is a specialized element in the Angular landscape. It is a container for grouping related things together. We can take advantage of this container to organize our application code more effectively. This is what Angular.io has to say about the modularity of *NgModules*.

- Declares which components, directives, and pipes belong to the module.
- Makes some of those components, directives, and pipes public so that other module's component templates can use them.
- Imports other modules with the components, directives, and pipes that components in the current module need.
- Provides *services* that the other application components can use.

Let's spend some time talking about some essential modules for an Angular application. As you review these modules think about the benefits of organizing your application code into specific groups. Put related things together to keep our code clean.

> Please take the time to understand how modules work within an Angular application. They are the essential building block to organizing the application functionality. They are generic enough that we can use these containers for multiple purposes or concerns.

12.2 Application Modules

The application module is the main entry point into the application. It has the responsibility to
coordinate all of the required modules for the application. Therefore, instead of overloading the `AppModule` to import, configure, initialize, and provide services - we organize our modules for the application into the following, each with specific responsibilities.

Here are some examples of modules in the sample application.

> Learn more about NgModules at:
> https://angular.io/guide/ngmodules#ngmodules

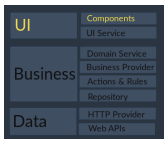
Module	Responsibility
AppModule	The `AppModule` coordinates all of the required modules for the application. It is responsible for bootstrapping the application.
AppRoutingModule	The `AppRoutingModule` provides the routing definition for the application. It can lazy-load any modules via a specific route.
BrowserAnimationsModule	Use to provide Angular animations - the illusion of motion for HTML elements. Well-designed animations can make your application more fun and easier to use, but they aren't just cosmetic. Animations can improve your app and user experience. More information at Angular.io
BrowserModule	Exports required infrastructure for all Angular apps. Included by default in all Angular apps created with the CLI new command. Re-exports CommonModule and ApplicationModule, making their exports and providers available to all apps. More information at Angular.io
CoreModule	The `CoreModule` provides the organization of other modules that are related to the specific domain of the application.

Module	Responsibility
CrossCuttingModule	The `CrossCuttingModule` module organizes the cross-cutting concerns into a single module to provide initialization and configuration, as well as providing global services to the application for: configuration, logging and error handling. These services are injected into the application root-level dependency injector.
SharedModule	Use the `SharedModule` to manage Angular and other 3rd-party modules/libraries/packages used by the application. For example: MaterialDesignModule, RouterModule, ReactiveFormsModule

The reference application has a focus on clean architecture with clearly-defined boundaries. The architecture also takes advantage of the SOLID principles using Object-Oriented Programming principles and patterns. The *presentation layer* relies on the list of modules above. I think it is interesting that before we even create a single domain-specific component or service that there is already a lot of code.

> In other industries, there is a lot of planning and infrastructure work implemented before the building of the specific target. Remember that thoughtful planning upfront will eliminate many future development problems. It is this type of due diligence that can make a difference between a flawless delivery or failure.

12.3 UI Layer



UI Components

The UI layer allows the application to display and collect information. That should be its only responsibility. Do not let the blog articles and sample applications out there sway you. The `component` part of the an Angular component was not meant to be the container for the entire implementation. Keep your components simple. Simple components are easy to test and maintain. I find that when things are not simple - tests are the first things to go. No tests means that you are quantifying your quality and that things are working.

Angular components are for display elements of your application. That is why they include a template (.html), style (.scss/css) files. The TypeScript portion of the *component* is the glue or binding data for display and collection.

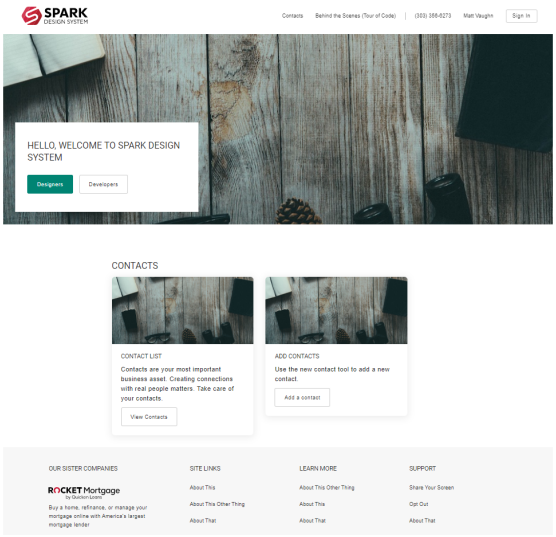
> If your component has a notion of HttpClient or Web APIs you have already created technical debt and violated (2) fundamental principles:
>
> - Single Responsibility
> - Separation of Concerns

Component Responsibility

Angular allows for sets of templates to compose the layout and display of the application details. The UI layer requires the ability to display information - however, it is not the responsibility for this layer to actually retrieve the data. The UI layer will respond to changes in data streams. Data streams come from RxJS as reactive data.

At the end of the day, the application is HTML and JavaScript running on a browser. Most of our discussion is within the context of a business application or software that is built to support a enterprise. This is most likely team-driven development that will last more than a few months. The application lifetime is also a consideration. Therefore, code organization (i.e., architecture) is a primary concern.

This is our reference application.



Site Layout with Header and Footer

Application Entry Module (AppModule)

Our architecture discussion will start with the application's entry-point module.

The application loads the AppModule - the AppModule coordinates the importing of the required application modules as mentioned in the table listed above. The app module coordinates the loading of all other required modules for the application. In the reference application only the app component is loaded by the app module. All other components will be loaded by feature modules of the application. This provides a simple entry point into the application.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { SparkAngularModule } from '@sparkdesignsystem/spark-angular';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CoreModule } from './modules/core/core.module';
import { CrossCuttingModule } from './modules/cross-cutting/cross-
cutting.module';
import { SharedModule } from './modules/shared/shared.module';
import { SiteModule } from './modules/site/site.module';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';

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

Application-Level Routes

For now, the application will take advantage of lazy-loaded routes. We define the routes with a path, a component, and a `loadChildren` function to Lazy load the target module. The component configuration allows us to wrap the target component with site level header and footer components - using the `LayoutComponent`.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LayoutComponent } from './modules/site/layout/layout.component';

const routes: Routes = [
  {
    path: '',
    component: LayoutComponent,
    loadChildren: () => import('@angular-architecture/reference/micro-
apps/contacts-\
app').then(m => m.ContactsAppModule),
  },
  {
    path: 'contacts/landing',
    component: LayoutComponent,
```

```
15     loadChildren: () => import('@angular-architecture/reference/micro-
apps/contacts-\
16 app').then(m => m.ContactsAppModule),
17   },
18   {
19     path: 'contacts/add',
20     component: LayoutComponent,
21     loadChildren: () => import('@angular-architecture/reference/micro-
apps/contacts-\
22 app').then(m => m.ContactsAppModule),
23   },
24   {
25     path: 'contacts/list',
26     component: LayoutComponent,
27     loadChildren: () => import('@angular-architecture/reference/micro-
apps/contacts-\
28 app').then(m => m.ContactsAppModule),
29   },
30 ];
31
32 @NgModule({
33   declarations: [],
34   imports: [
35     RouterModule.forRoot(routes, { enableTracing: true, useHash: false,
scrollPositi\
36 onRestoration: 'top', anchorScrolling: 'enabled' }),
37   ],
38 })
39 export class AppRoutingModule {}
```

Feature Module

Angular allows us to add additional modules for features. Feature modules provide a great code organization strategy to encapsulate specific feature components and logic together. You can add a feature module to the application project or you can also create a library project for the specific feature module. Typically feature modules are not shared between different projects. However, if you have a feature that will be shared by multiple projects then it is a good candidate to create a feature module as a *library* project. The only difference is where the code is located. A good rule to follow is to keep related things together. If the feature is a single domain feature of the application and is not shared/used by other applications, then it should be created and managed within the application project.

- Application project feature module: `ng generate module contacts`

Feature Module as a Library Project

I used the Nx Console to generate the CLI command that creates a feature module within a library project. Remove the `--dry-run` to create the library

project.

- Library project feature module: `ng generate @nrwl/angular:library --name=contacts --style=scss --directory=reference --lazy --routing --simpleModuleName --no-interactive --dry-run`

  Feature module library projects can be lazy-loaded as micro frontends by multiple application projects. This provides additional capabilities of sharing and reusing micro frontends and/or feature modules.

Here is the output of the `generate library` command for the feature module.

```
 1 ng generate @nrwl/angular:library --name=contacts2 --style=scss --
directory=referenc\
 2 e --lazy --routing --simpleModuleName --no-interactive --dry-run <
 3
 4 CREATE libs/reference/contacts/README.md (162 bytes)
 5 CREATE libs/reference/contacts/tsconfig.lib.json (411 bytes)
 6 CREATE libs/reference/contacts/tslint.json (251 bytes)
 7 CREATE libs/reference/contacts/src/index.ts (40 bytes)
 8 CREATE libs/reference/contacts/src/lib/contacts.module.ts (337 bytes)
 9 CREATE libs/reference/contacts/tsconfig.json (126 bytes)
10 CREATE libs/reference/contacts/jest.config.js (383 bytes)
11 CREATE libs/reference/contacts/tsconfig.spec.json (236 bytes)
12 CREATE libs/reference/contacts/src/test-setup.ts (30 bytes)
13 UPDATE package.json (3145 bytes)
14 UPDATE angular.json (22776 bytes)
15 UPDATE nx.json (1110 bytes)
16 UPDATE tsconfig.json (1897 bytes)
```

In the code sample below, noticed that the feature module contains its own routing module. A future module can define a set of child routes for the application.

```
 1 import { CommonModule } from '@angular/common';
 2 import { NgModule } from '@angular/core';
 3
 4 import { AppRoutingModule } from './app-routing/app-routing.module';
 5 import { RouterModule } from '@angular/router';
 6
 7 @NgModule({
 8   imports: [CommonModule, AppRoutingModule, RouterModule],
 9   declarations: [],
10 })
11 export class ContactsAppModule {}
```

Feature Module Routes

The feature module routes are lazy-loaded to allow for smaller bundle sizes and performance when not all components will be accessed by the user. Notice that we have modules contained within the feature module. These additional modules are component modules. They can be described as a Single-Component Application Module (SCAM). For example, let's say a user is interested in going to the landing page and perhaps seeing a list of contacts. They are not interested in adding a new contact or editing a contact. Therefore, we only load the components explicitly requested by the user using router links.The other modules are not loaded by the application (i.e., ItemModule or AddModule).

> Modules. The **AppModule** has a **AppRoutingModule** that points to a **ContactsAppModule** that
> contains a child routing **AppRoutingModule** with a set of lazy-loaded **single-component**
> **application modules**. Modules are just *containers* to keep related things organized.

```
 1 import { CommonModule } from '@angular/common';
 2 import { NgModule } from '@angular/core';
 3 import { RouterModule, Routes } from '@angular/router';
 4
 5 const routes: Routes = [
 6   { path: '', loadChildren: () => import('../landing/landing.module').then(m
=> m.La\
 7 ndingModule) },
 8   { path: 'contacts/list', loadChildren: () =>
import('../list/list.module').then(m \
 9 => m.ListModule) },
10   { path: 'contacts/item/edit:id', loadChildren: () =>
import('../item/item.module')\
11 .then(m => m.ItemModule) },
12   { path: 'contacts/new-contact', loadChildren: () =>
import('../add/add.module').th\
13 en(m => m.AddModule) },
14 ];
15
16 @NgModule({
17   declarations: [],
18   imports: [CommonModule, RouterModule.forChild(routes)],
19 })
20 export class AppRoutingModule {}
```

Single Component Application Module

Let us take a look at a single component application module. This is a component that is managed by a single module. this single component module contains a single route that defaults to the single target component. This technique or strategy is very similar to how components for Material Design are managed. In earlier releases of Material Design, all of the components were contained in a single module. However, today each component is contained in its own module to allow the application to specify which components it needs. This creates much smaller bundle sizes - because you only import what you are going to use.

Notice that the landing component is declared in the module however it is not exported. This means that the component is encapsulated within the module. the next section will explain how this component is loaded using a child route.

> Remember, a single-component application module is a pattern. There may be a case to allow more than one component in the module. For example, when there are container and presentational components that are required to compose the view.

Learn more about this topic from the article by Lars Gyrup Brink Nielsen "Single Component Angular Modules".

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

import { LandingRoutingModule } from './landing-routing.module';
import { LandingComponent } from './landing.component';
import { SparkAngularModule } from '@sparkdesignsystem/spark-angular';

@NgModule({
  declarations: [LandingComponent],
  imports: [LandingRoutingModule, SparkAngularModule],
})
export class LandingModule {}
```

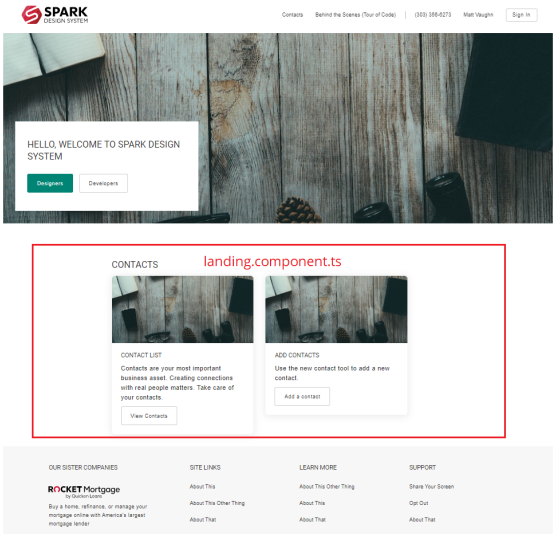
Default Routes for Component Module

The component module contains its own router module with a single route targeting the single component. In our example, the *LandingComponent* will load when the route is empty.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { LandingComponent } from './landing.component';

const routes: Routes = [{ path: '', component: LandingComponent }];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class LandingRoutingModule {}
```

Landing Component

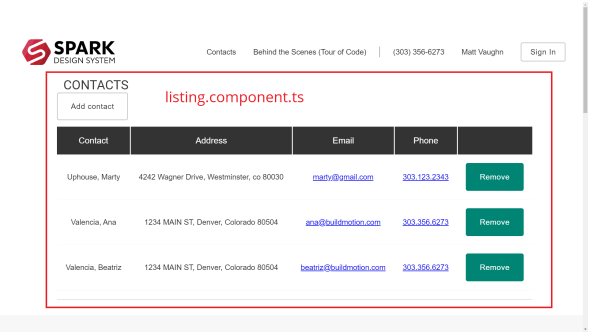
The Landing component is wrapped within the layout component. This allows for other templates, for example the header and footer, to be combined with these other single-component application modules. In our reference application the *LandingComponent* contains two cards.



Landing Component for Contacts

List Component

If a user clicks on the view contacts routable link, the *ListComponent* will be lazy-loaded by the application module. The *ListComponent* shows a list of contacts using the table from the Spark Design System.



Notice that the *ListComponent* does not contain any HTTP or Web API logic. The component will initiate a request for data however it is not responsible for actually making that request for data. Basically, the *ListComponent* is ***asking*** for data only - it is not responsible for or should know how to actually retrieve the data. This is really an example of the Inversion of Control (IoC) pattern. The component will react to the `contacts$` Observable when contact data is published to that data stream by the UI service.

> Note that there are no `Observable` subscriptions or unsubscribe logic within the component. This
> is handled in the template using the `async` pipe.

```
import { Component, OnInit } from '@angular/core';
import { ComponentBase } from '@angular-architecture/foundation';
import { LoggingService, Severity } from '@angular-architecture/logging';
import { Router } from '@angular/router';
import { Observable } from 'rxjs';
import { ContactListUIService } from './contact-list-ui.service';
// tslint:disable-next-line:nx-enforce-module-boundaries
import { Contact } from '@angular-architecture/reference/domain/common';

@Component({
  selector: 'valencia-list',
  templateUrl: './list.component.html',
  styleUrls: ['./list.component.scss'],
})
export class ListComponent extends ComponentBase implements OnInit {
  contacts$: Observable<Contact[]> = this.uiService.contacts$;
  removeContactSpinner$: Observable<boolean> =
this.uiService.removeContactSpinner$;
  showSpinner$: Observable<boolean> = this.uiService.showSpinner$;

  constructor(
    private uiService: ContactListUIService,
    loggingService: LoggingService,
```

```
23      router: Router) {
24      super('ListComponent', loggingService, router);
25    }
26
27    ngOnInit(): void {
28      this.uiService.retrieveContacts();
29    }
30
31    addContact() {
32      this.routeTo('contacts/new-contact');
33    }
34
35    removeContact(contact: Contact) {
36      this.uiService.removeContact(contact);
37    }
38  }
```

List Template

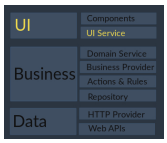
Notice that the component receives data stream events from the UI service and reacts to them accordingly within the template.The template takes advantage of the `async` pipe to subscribe and unsubscribe to the target `Observable` items:

- `showSpinner$`: A boolean observable used to show or hide the spinner control.
- `contacts$`: An Observable of `Contact` items.

```
 1 <div class="container">
 2   <div class="qc-contact-list--default-item">
 3     <ng-container *ngIf="showSpinner$ | async as showSpinner; else
contactListTempla\
 4 te">
 5         <sprk-modal [isVisible]="showSpinner" title="Retrieving contacts..."
modalTyp\
 6 e="wait" idString="modal-wait-1">
 7           <p class="sprk-c-Modal__content">
 8             Please wait while we retrieve your contacts.
 9           </p>
10         </sprk-modal>
11     </ng-container>
12     <ng-container *ngIf="removeContactSpinner$ | async as
removeContactSpinner">
13         <sprk-modal [isVisible]="removeContactSpinner" title="Removing
contact..." m\
14 odalType="wait" idString="modal-wait-1">
15           <p class="sprk-c-Modal__content">
16             Please wait while we remove the selected contact.
17           </p>
18         </sprk-modal>
19     </ng-container>
20     <ng-template #contactListTemplate>
21       <ng-container *ngIf="contacts$ | async as contacts; else
nodataTemplate">
```

```
22          <div class="qc-contact-list--default-item ">
23            <h1>Contacts</h1>
24          </div>
25          <div class="qc-contact-list--add-button">
26            <button (click)="addContact()" variant="secondary"
idString="button-second\
27 ary" sprkButton>
28              Add contact
29            </button>
30          </div>
31          <!-- CONTACT(S) TABLE -->
32          <sprk-table idString="table-1">
33          </sprk-table>
34          <sprk-divider idString="divider-1"></sprk-divider>
35        </ng-container>
36        <ng-template #nodataTemplate>
37          <!-- DISPLAY NO DATA MESSAGE TO USER -->
38          <sprk-alert alertType="fail" idString="alert-fail-1"
analyticsString="object\
39 .action.event">
40            The application attempted to retrieve your contacts - however, it
was not \
41 successful.
42          </sprk-alert>
43        </ng-template>
44      </ng-template>
45    </div>
46 </div>
```

12.4 UI Service



UI Service

The magic that makes this all happen is the UI service. The UI service is playing the part of the presenter (P) in the MVP or model view presenter pattern. The UI service is playing the role of an adapter or mediator between the UI component and the domain service of the application. It has the responsibility to manage the data operations for the component. Not to implement the data operations but to handle the response from the domain service layer of the application. Doing this keeps data operations out of the UI layer where it has no responsibility. It keeps the UI layer clean to provide data display and data capture functionality only.

UI Service Responsibilities

The UI service also provides State Management and also coordinates UI Logic for display elements for the specific view. In the code below noticed that there is a `contacts` array as well as a `showSpinner$` Observable. Notice that the observables are managed by a set of private RxJS Subjects along with a public `readonly` Observable exposed to the component.

> The domain service is just an `@Injectable` service that is added to the feature domain library. The domain service encapsulates the business logic for a specific feature domain of the application. It includes the validation of inputs, business rule evaluation, and interacting with the application[s Web API.

Handles Data Operations

The domain service is injected into the constructor of the UI service. The UI service initiates a data operation and subscribes to that operation to handle the response or the error from that operation call. When I use the

domain service, I like to handle the response, handle any errors, and also finish the observable request. This style or mechanism allows me to perform different behaviors for the UI and handle errors to provide notifications or information to the user. I use this pattern in all of the UI services for consistency and maintainability of UI logic.

> Note that an observable will return either a successful response or an error. Therefore it is a good practice to handle both options for a call that returns an observable. Not handling the error response is optimistic-programming - there are always unexpected errors in real life and real applications. The *third* member of the observable response is always called (regardless of success or error), so I usually implement this as a finish of the request for the Observable data.

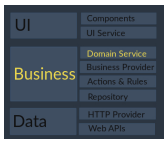
I use the pointer-functions to private methods within the UI service. This groups related code closer together and keeps the subscription code clean. The encapsulation of handling the success or error response separately is a good convention.

```
  1 import { Injectable } from '@angular/core';
  2 import { ServiceBase } from '@angular-architecture/foundation';
  3 import { LoggingService, Severity } from '@angular-architecture/logging';
  4 import { ContactsService } from '@angular-
architecture/reference/domain/contacts-ser\
  5 vice';
  6 import { Observable, BehaviorSubject } from 'rxjs';
  7 // tslint:disable-next-line:nx-enforce-module-boundaries
  8 import { Contact } from '@angular-architecture/reference/domain/common';
  9 import { ApiResponse } from '@angular-architecture/common';
 10
 11 @Injectable({
 12   providedIn: 'root',
 13 })
 14 export class ContactListUIService extends ServiceBase {
 15   private contactsSubject: BehaviorSubject<Contact[]> = new
BehaviorSubject<Contact[\
 16 ]>(null);
 17   private removeContactSpinnerSubject: BehaviorSubject<boolean> = new
BehaviorSubjec\
 18 t<boolean>(false);
 19   private showSpinnerSubject: BehaviorSubject<boolean> = new
BehaviorSubject<boolean\
 20 >(false);
 21
 22   public readonly contacts$: Observable<Contact[]> =
this.contactsSubject.asObservab\
 23 le();
 24   public readonly removeContactSpinner$: Observable<boolean> =
```

```
this.removeContactSpi\
 25 nnerSubject.asObservable();
 26   public readonly showSpinner$: Observable<boolean> =
this.showSpinnerSubject.asObse\
 27 rvable();
 28
 29   contacts: Contact[];
 30
 31   constructor(
 32     private contactsService: ContactsService,
 33     loggingService: LoggingService) {
 34     super('ContactListUIService', loggingService);
 35   }
 36
 37   removeContact(contact: Contact) {
 38     this.removeContactSpinnerSubject.next(true);
 39
 40     this.loggingService.log(`${this.serviceName}.removeContact`,
Severity.Informatio\
 41 n, `Preparing to remove contact.`);
 42     this.contactsService
 43       .removeContact<boolean>(contact)
 44       .subscribe(
 45         response => this.handleRemoveContactResponse(response),
 46         error => this.handleRemoveContactError(error),
 47         () => this.finishRemoveContactRequest()
 48       );
 49   }
 50
 51   retrieveContacts() {
 52     this.showSpinnerSubject.next(true);
 53
 54     this.contactsService
 55       .retrieveContacts<Contact[]>()
 56       .subscribe(
 57         response => this.handleRetrieveContactsResponse(response),
 58         error => this.handleRetrieveContactsError(error),
 59         () => this.finishRetrieveContactsRequest()
 60       );
 61   }
 62
 63   private handleRemoveContactResponse(response: ApiResponse<boolean>): void
{
 64     this.loggingService.log(this.serviceName, Severity.Information,
`Preparing to pr\
 65 ocess API response for [remove contact]`);
 66     if (response) {
 67       if (response.isSuccess) {
 68         this.loggingService.log(this.serviceName, Severity.Information,
`Preparing t\
 69 o process [successful] API response`);
 70         this.retrieveContacts();
 71       } else {
 72         this.loggingService.log(this.serviceName, Severity.Information,
`Preparing t\
 73 o process [unsuccessful] API response`);
 74       }
 75     }
 76   }
 77
 78   private handleRemoveContactError(error: any): void {
```

```
 79      this.handleError(error);
 80    }
 81
 82    private finishRemoveContactRequest(): void {
 83      this.removeContactSpinnerSubject.next(false);
 84      this.loggingService.log(this.serviceName, Severity.Information,
`Finished proces\
 85 sing request to remove contact..`);
 86    }
 87
 88    private handleRetrieveContactsResponse(response: ApiResponse<Contact[]>):
void {
 89      this.loggingService.log(this.serviceName, Severity.Information,
`Preparing to pr\
 90 ocess API response for [retrieving contacts]`);
 91      if (response) {
 92        if (response.isSuccess) {
 93          this.loggingService.log(this.serviceName, Severity.Information,
`Preparing t\
 94 o process [successful] API response`);
 95          this.contacts = response.data;
 96          this.sortContacts(this.contacts);
 97          this.contactsSubject.next(this.contacts);
 98        } else {
 99          this.loggingService.log(this.serviceName, Severity.Information,
`Preparing t\
100 o process [unsuccessful] API response`);
101        }
102      }
103    }
104
105    private sortContacts(contacts: Contact[]) {
106      if (contacts && contacts.length > 0) {
107        const sorted = contacts.sort((o1, o2) => {
108          if (o1.lastName > o2.lastName) {
109            return 1;
110          }
111
112          if (o1.lastName < o2.lastName) {
113            return -1;
114          }
115
116          return 0;
117        });
118
119        this.contacts = sorted;
120      }
121    }
122
123    private handleRetrieveContactsError(error: any): void {
124      this.showSpinnerSubject.next(false);
125      this.handleError(error);
126    }
127
128    private finishRetrieveContactsRequest(): void {
129      this.showSpinnerSubject.next(false);
130    }
131  }
```

12.5 Domain Service



Domain Service

> See Create Domain Library Projects for more information on setting up domain service library projects.

A domain service for the application is typically implemented as a library project most of the time. A domain service library is implemented for a specific feature vertical of the application. In our example, we have a *contacts* domain library service. The domain library will contain an exported service. This service provides an API for consumers of the domain service to interact with. All of the internal business logic, input validations, business rule implementations, and HTTP calls will be encapsulated within this specific domain service library and is not exposed to the consumers of the domain library.

> A library of this sort can also be used by multiple UI consumers. Therefore if there are other component UI services that require the specific domain service, it can be injected into the constructor of that UI service. This is another good example of code reuse and sharing.

The domain service library really isn't anything without an `@Injectable` service. The service provides an entry point or an API for the consumers. It does not have the responsibility for business logic or other details of the domainLibrary. It will delegate this responsibility to a distinct business logic layer within this domain library. In our reference application, we use another `@Injectable` service, BusinessProviderService, to coordinate all of the business logic for the specific domain library.

Service Context

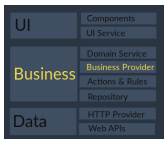
Notice that the domain library service provides a `ServiceContext` to the `BusinessProviderService`. The `ServiceContext` provides a mechanism to capture information during the execution flow of the specific operation. Some of this information may be useful messages for the user or for the application's centralized logging repository.

> The ability to capture messages and/or information during the flow of a data operation
> is very important. The `ServiceContext` is a mechanism used by backend or enterprise
> applications to allow tracking between the different *layers* of the application. TypeScript's
> object-oriented features and Angular's dependency injection features make this
> possible for Angular web applications.

```
 1 import { Observable } from 'rxjs';
 2
 3 import { Inject, Injectable } from '@angular/core';
 4 import { ApiResponse } from '@angular-architecture/common';
 5 import { ServiceBase } from '@angular-architecture/foundation';
 6 import { LoggingService } from '@angular-architecture/logging';
 7 // tslint:disable-next-line: nx-enforce-module-boundaries
 8 import { Contact, ContactDto } from '@angular-
architecture/reference/domain/common';
 9
10 import { BusinessProviderService } from './business/business-
provider.service';
11
12 @Injectable({
13   providedIn: 'root',
14 })
15 export class ContactsService extends ServiceBase {
16   constructor(
17     @Inject(BusinessProviderService) private businessProvider:
BusinessProviderServi\
18 ce,
19     loggingService: LoggingService) {
20     super('ContactsService', loggingService);
21     this.businessProvider.serviceContext = this.serviceContext;
22   }
23
24   add<T>(contact: ContactDto): Observable<ApiResponse<T>> {
25     return this.businessProvider.add<T>(contact);
26   }
27
28   removeContact<T>(contact: Contact) {
```

```
29      return this.businessProvider.removeContact<T>(contact);
30    }
31
32    retrieveContacts<T>(): Observable<ApiResponse<T>> {
33      return this.businessProvider.retrieveContacts<T>();
34    }
35  }
```

12.6 Business Logic



Business Provider

The `BusinessProviderService` implements a set of business actions based on the definition of the `IBusinessProviderService` interface. The business logic layer provides a single `BusinessProviderService` along with as many actions required to implement the business logic units. This is a unique place in the application where an instance of something is created during the runtime of the application. The business actions are discrete *units of work* that run within a specific context of inputs.

Responses are Observable and Reactive

The methods of the `BusinessProviderService` will return an `Observable` response - this keeps the application reactive. Please note that an HTTP service as well as a configuration service is injected into the constructor of the business provider. This information is also made available to the business action by passing in a reference of the business provider into the `Do(this)` method of the action. This keeps the signatures of the business actions clean from any infrastructure or cross-cutting concern items. I definitely appreciate clean and simple constructors when it comes to testing the business actions. There is less set up for tests.

```
 1 import { Injectable, Inject } from '@angular/core';
 2 import { HttpContactsRepositoryService } from './http-contacts-
repository.service';
 3 import { ConfigurationService } from '@angular-architecture/configuration';
 4 import { LoggingService } from '@angular-architecture/logging';
 5 import { ServiceBase } from '@angular-architecture/foundation';
 6 import { ApiResponse } from '@angular-architecture/common';
 7 import { Observable } from 'rxjs';
 8 import { IBusinessProviderService } from './i-business-provider.service';
 9 import { AddContactAction } from './actions/add-contact.action';
10 // tslint:disable-next-line: nx-enforce-module-boundaries
11 import { Contact, ContactDto } from '@angular-
architecture/reference/domain/common';
12 import { RetrieveContactsAction } from './actions/retrieve-contacts.action';
```

```
13 import { RemoveContactAction } from './actions/remove-contact.action';
14
15 @Injectable({
16   providedIn: 'root',
17 })
18 export class BusinessProviderService extends ServiceBase implements
IBusinessProvide\
19 rService {
20   constructor(
21     @Inject(HttpContactsRepositoryService) public apiService:
HttpContactsRepository\
22 Service,
23     public configService: ConfigurationService,
24     loggingService: LoggingService
25   ) {
26     super('BusinessProviderService', loggingService);
27   }
28
29   add<T>(contact: ContactDto): Observable<ApiResponse<T>> {
30     const action = new AddContactAction<T>(contact);
31     action.Do(this);
32     return action.response;
33   }
34
35   removeContact<T>(contact: Contact): Observable<ApiResponse<T>> {
36     const action = new RemoveContactAction<T>(contact);
37     action.Do(this);
38     return action.response;
39   }
40
41   retrieveContacts<T>(): Observable<ApiResponse<T>> {
42     const action = new RetrieveContactsAction<T>();
43     action.Do(this);
44     return action.response;
45   }
46 }
```

12.7 Business Action

Business Actions

A business action is a unit of work that has a distinct single responsibility. Business actions use the @angular-architecture/actions library package. A business action participates in a template method design pattern. In this pattern there are pre and post operations that are always executed within the pipeline of an action. If the business rules or validation rules of an action do not contain any errors, the action is performed. performing an action will most likely be implemented as a data operation to the web API. one of the pre-processing operations of an action is validation. When a business action is executed it will validate any defined business rules or input validation rules using the rules engine library project.

Good artists copy, Great artists steal

The implementation details of the business action framework is based on the classic Template Method design pattern. I studied how it was implemented in different frameworks (i.e., Microsoft ASP.NET Page-lifecycle). In 2005, the source
code of the ASP.NET page-lifecycle was available. I was fascinated by the simplicity of the pattern - and how you could use it to create simple or very complex web pages: consistently and reliable.

> If the *pipeline* or *template method pattern* sounds familiar, it is because it is used
> in many popular UI frameworks since the late 1990's (Java Server Pages (JSP),
> Microsoft's ASP.NET, and now Angular Components). The Angular component provides a set of

hooks (events) and interfaces (OnInit and others) to hook into or implement.

The business action library/package leverages the pattern borrowed from my analysis and study of the implementation details of webpage lifecycle code. I discovered the name of the design pattern later - it has existed for many years. I have the 1994 Design Patterns book published by Addison-Wesley. If you understand the core fundamentals of a design pattern, you will begin to see its use in every day life but also how it may simplify technical implementations.

Simple Business Action

A simple business action only requires the implementation of the `performAction()` method. The `preValidateAction()` method is completely optional - but is the location to add business rules and validation for inputs. By using a very structured and conventional approach to business logic implementation allows for consistency. The business logic is the heart of the application. It deserves the attention as a first-class citizen within the application code base.

```
 1 import { BusinessActionBase } from './business-action-base';
 2 import { Severity } from '@angular-architecture/logging';
 3
 4 /**
 5  * Use this action to perform business logic with validation and business
rules.
 6  */
 7 export class RetrieveContactsAction<T> extends BusinessActionBase<T> {
 8   constructor() {
 9     super('RetrieveContactsAction');
10   }
11
12   /**
13    * Use the [preValidateAction] to add any business or validation rules
that
14    * are required to pass in order to perform the action.
15    *
16    * Use the [ValidationContext] item of the action to add rules. The
ValidationCont\
17 ext
18    * uses a Fluent API to allow for chained rules to be configured.
19    */
20   preValidateAction() {}
21
22   /**
23    * Use the [performAction] operation to execute the target of the action's
busines\
```

```
24 s logic. This
25    * will only run if the rules and validations are successful.
26    */
27   performAction() {
28     this.loggingService.log(this.actionName, Severity.Information,
`Preparing to cal\
29 l API to complete action.`);
30     this.response = this.businessProvider.apiService.retrieveContacts<T>();
31   }
32 }
```

Action with Rules

The business action below demonstrates how to use the rules engine to validate the input before making a web API call. Any rule violations will be added to the `ServiceContext` which will allow these messages to be used as notifications to the user. Most of this functionality is handled in the *base* classes of the specific business action. The business action makes use of TypeScript's object-oriented capabilities to provide common behaviors for all actions.

The rule engine contains a set of predefined rules that can be used together to create specific composite rules. The rules engine allows you to create custom rules based on a simple or composite rule Base Class. All rules are executed consistently which allows them to be maintainable and testable for an application. *Some of the rules were removed for brevity.*

```
  1 import { BusinessActionBase } from './business-action-base';
  2 import { StringIsNotNullEmptyRange, IsNotNullOrUndefined } from '@angular-
architectu\
  3 re/rules-engine';
  4 import { Severity } from '@angular-architecture/logging';
  5 // tslint:disable-next-line:nx-enforce-module-boundaries
  6 import { ContactDto, EmailAddressFormatIsValidRule } from '@angular-
architecture/ref\
  7 erence/domain/common';
  8
  9 /**
 10  * Use this action to perform business logic with validation and business
rules.
 11  */
 12 export class AddContactAction<T> extends BusinessActionBase<T> {
 13   constructor(private contact: ContactDto) {
 14     super('AddContactAction');
 15   }
 16
 17   /**
 18    * Use the [preValidateAction] to add any business or validation rules
that
 19    * are required to pass in order to perform the action.
```

```
 20     *
 21     * Use the [ValidationContext] item of the action to add rules. The
ValidationCont\
 22 ext
 23     * uses a Fluent API to allow for chained rules to be configured.
 24     */
 25   preValidateAction() {
 26     this.validationContext.addRule(
 27       new IsNotNullOrUndefined('ContactDtoIsValid', 'The contact
information is not \
 28 valid.', this.contact, this.showRuleMessages)
 29     );
 30
 31     if (this.contact) {
 32       this.validationContext
 33         .addRule(
 34           new StringIsNotNullEmptyRange(
 35             'Address1IsValid',
 36             'The address information 1 is required. Cannot be greater than
60 charac\
 37 ters.',
 38             this.contact.address1,
 39             3,
 40             60,
 41             this.showRuleMessages
 42           )
 43         )
 44         .addRule(
 45           new StringIsNotNullEmptyRange(
 46             'Address2IsValid',
 47             'The address 2 information is required. Cannot be greater than
60 charac\
 48 ters.',
 49             this.contact.address2,
 50             0,
 51             60,
 52             this.showRuleMessages
 53           )
 54         )
 55         .addRule(
 56           new StringIsNotNullEmptyRange(
 57             'EmailAddressIsValid',
 58             'The email is required. Cannot be greater than 80 characters.',
 59             this.contact.emailAddress,
 60             5,
 61             80,
 62             this.showRuleMessages
 63           )
 64         )
 65         .addRule(
 66           new EmailAddressFormatIsValidRule(
 67             'EmailAddressFormatIsValid',
 68             'The email address format is not valid.',
 69             this.contact.emailAddress,
 70             this.showRuleMessages
 71           )
 72         )
 73         .addRule(
 74           new StringIsNotNullEmptyRange(
 75             'PostalCodeIsValid',
 76             'The postal code value is required. Cannot be greater than 25
```

```
characters\
 77 .',
 78               this.contact.postalCode,
 79               5,
 80               25,
 81               this.showRuleMessages
 82             )
 83           )
 84           .addRule(
 85             new StringIsNotNullEmptyRange(
 86               'StateIsValid',
 87               'The state value is required. Cannot be greater than 45
characters.',
 88               this.contact.state,
 89               2,
 90               45,
 91               this.showRuleMessages
 92             )
 93           );
 94       }
 95     }
 96
 97     /**
 98      * Use the [performAction] operation to execute the target of the
action's busines\
 99 s logic. This
100      * will only run if the rules and validations are successful.
101      */
102     performAction() {
103       this.loggingService.log(this.actionName, Severity.Information,
`Preparing to cal\
104 l API to complete action.`);
105       this.response = this.businessProvider.apiService.addContact<T>
(this.contact);
106     }
107 }
```

Unit Testing Business Actions

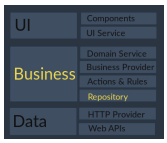
Business actions do not contain any UI logic or UI concerns. They are much easier to test and allow you to use a more traditional unit testing strategy. The following example runs the specification tests targeting the `AddContactAction` business action. If the code and the structure (architecture/organization) allow for easier testing there will be more tests. More tests increase and quantify the quality of your application.

> The actual specification test file is in the Github repository https://github.com/buildmotion/contacts-reference-app

```
 1 yarn run v1.22.1
 2 $ ng test --project=reference-domain-contacts-service --test-file=add-
contact.action\
```

```
 3 .spec.ts
 4  PASS  libs/reference/domain/contacts-service/src/lib/business/actions/add-
contact.a\
 5 ction.spec.ts
 6   VerifyAccountsAction
 7     √ should create an instance (35ms)
 8     √ should contain validation error for MIN LENGTH first name (18ms)
 9     √ should contain validation error for MIN LENGTH last name (10ms)
10     √ should contain validation error for MIN LENGTH company (9ms)
11     √ should contain validation error for MIN LENGTH phone (10ms)
12     √ should contain validation error for MIN LENGTH email address (9ms)
13     √ should contain validation error for MIN LENGTH postal code (10ms)
14     √ should contain validation error for MIN LENGTH state (13ms)
15     √ should contain validation error for MIN LENGTH city (9ms)
16     √ should contain validation error for MAX LENGTH first name (11ms)
17     √ should contain validation error for MAX LENGTH last name (8ms)
18     √ should contain validation error for MAX LENGTH company (12ms)
19     √ should contain validation error for MAX LENGTH email address (15ms)
20     √ should contain validation error for MAX LENGTH phone (10ms)
21     √ should contain validation error for MAX LENGTH city (10ms)
22     √ should contain validation error for MAX LENGTH state (14ms)
23     √ should contain validation error for MAX LENGTH postal code (8ms)
24     √ should contain validation error for MAX LENGTH address 2 (8ms)
25     √ should contain validation error for MAX LENGTH address 1 (12ms)
26     √ should contain validation error for MIN LENGTH address 1 (11ms)
27     √ should contain validation error for UNDEFINED address 1 (11ms)
28     √ should contain valid VALIDATION CONTEXT (9ms)
29     √ should contain email address format validation rule (12ms)
30
31 Test Suites: 1 passed, 1 total
32 Tests:       23 passed, 23 total
33 Snapshots:   0 total
34 Time:        54.297s
35 Ran all test suites matching /add-contact.action.spec.ts/i.
36 Done in 67.11s.
```

12.8 HTTP Repository



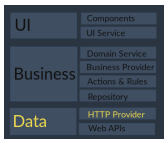
Business Repository

The business action will perform the action when there are no rule violations. Most of the time it will use the HTTP service from the business provider to make a web API call. This HTTP service is implemented as a repository pattern. The implementation details of the repository do not include any specifics about anything related to HTTP or the Angular HttpClient. The signature of the HTTP service or repository only includes the required input to construct the date of operation.

```
 1 import { Injectable } from '@angular/core';
 2 import { ServiceBase } from '@angular-architecture/foundation';
 3 import { ConfigurationService } from '@angular-architecture/configuration';
 4 import { HttpService, HttpRequestMethod } from '@angular-architecture/http-
service';
 5 import { LoggingService, Severity } from '@angular-architecture/logging';
 6 import { Observable } from 'rxjs';
 7 import { IHttpContactsRepositoryService } from './i-http-contacts-
repository.service\
 8 ';
 9 import { ContactDto } from '@angular-architecture/reference/domain/common';
10
11 @Injectable({
12   providedIn: 'root',
13 })
14 export class HttpContactsRepositoryService extends ServiceBase implements
IHttpConta\
15 ctsRepositoryService {
16   constructor(
17     private httpService: HttpService,
18     private configService: ConfigurationService,
19     loggingService: LoggingService) {
20     super('HttpSearchRepositoryService', loggingService);
21   }
22
23   retrieveContacts<T>(): Observable<any> {
24     const requestURL = `https://yts930ocx2.execute-api.us-west-
1.amazonaws.com/dev/c\
25 ontacts`;
26     const options = this.httpService.createOptions(
27       HttpRequestMethod.get,
28       this.httpService.createHeader(),
29       requestURL,
```

```
30        null,
31        false);
32      return this.httpService.execute(options);
33    }
34  }
```

12.9 HTTP Client



HTTP Client

The HTTP service is a utility @Injectable service that uses the Angular HttpClient class. It allows for the construction of the API options, header information, and any required body for the request. It is the only place in the entire application where API calls are made.

> Note: There is no HttpClient usage in the UI layer of the application. This keeps the components simple and lightweight. They are also easier to test and maintain.

```
 1 import { Injectable } from '@angular/core';
 2 import { HttpRequestMethod } from './http-request-methods.enum';
 3 import { HttpHeaders, HttpClient, HttpResponse, HttpErrorResponse } from
'@angular/c\
 4 ommon/http';
 5 import { HttpRequestOptions } from './http-request-options';
 6 import { Observable, throwError } from 'rxjs';
 7
 8 @Injectable({ providedIn: 'root' })
 9 export class HttpService {
10   constructor(private httpClient: HttpClient) {}
11
12   /**
13    * Use to create [options] for the API request.
14    * @param method Use to indicate the HttpRequest verb to target.
15    * @param headers Use to provide any [HttpHeaders] with the request.
16    * @param url Use to indicate the target URL for the API request.
17    * @param body Use to provide a JSON object with the payload for the
request.
18    * @param withCredentials Use to indicate if request will include
credentials (coo\
19 kies).
20    */
21   createOptions(
22     method: HttpRequestMethod,
23     headers: HttpHeaders,
24     url: string,
25     body: any,
26     withCredentials: boolean = true): HttpRequestOptions {
27
```

```
28      let options: HttpRequestOptions;
29      options = new HttpRequestOptions();
30      options.requestMethod = method;
31      options.headers = headers;
32      options.requestUrl = url;
33      options.body = body;
34      options.withCredentials = withCredentials;
35      return options;
36    }
37
38    /**
39     * Use to create a new [HttpHeaders] object for the HTTP/API request.
40     */
41    createHeader(): HttpHeaders {
42      let headers = new HttpHeaders();
43      headers = headers.set('Content-Type', 'application/json');
44      return headers;
45    }
46
47    /**
48     * Use to execute an HTTP request using the specified options in the
[HttpRequestO\
49  ptions].
50     * @param requestOptions
51     */
52    execute<T>(requestOptions: HttpRequestOptions):
Observable<HttpResponse<T>> {
53      console.log(`Preparing to perform request to:
${requestOptions.requestUrl}`);
54      return this.httpClient.request<T>(
55        requestOptions.requestMethod.toString(),
56        requestOptions.requestUrl, {
57        body: requestOptions.body,
58        headers: requestOptions.headers,
59        reportProgress: requestOptions.reportProgress,
60        observe: requestOptions.observe,
61        params: requestOptions.params,
62        responseType: requestOptions.responseType,
63        withCredentials: requestOptions.withCredentials,
64      });
65    }
66  }
```

13 Analysis and Design

13.1 Do You Know *Who* is Using Your App?

13.2 Actors - Not Just for Hollywood

13.3 *What* Does Each Actor Want To Do?

- Action (verb)
- What are the characteristics of this action?

13.4 *Why* is 42?

- What is the purpose?
- What is the outcome?
- What are the benefits?
- Does anyone else benefit from the action?
- Do you need to track information about the action?

13.5 *When* Do Things Happen?

- What is the frequency of the action?
- What are the required inputs to perform the action?
- Is it performed on-demand?
- Does it need to be scheduled?
- Does it run based on an external event?

13.6 *Where* Do Things Happen?