Logic Room Presents

Think!
Like a UI Architect

5 Critical Lessons for Framework-agnostic JavaScript UI App Design and Structure

Big Picture Ideas to Help You Create Better Architecture
with Any UI Framework (inc. React, Angular or Vue)

by Pete Heard

1. Introduction

At Logic Room, we teach engineers how to use powerful framework-agnostic techniques to radically improve their UI testing and architecture skills.

This guide will outline our philosophy and give you 5 clear concepts you can begin to use immediately in your projects to help you ***think like a UI architect***.

1.1. About You

But before we go any further, we need to establish some perspective; I want you to imagine that in front of you there are two doors…

Door number 1, and door number 2.

Door number 1 says ‘Frameworks’.

Behind this door lies pain.

It’s a place where FRAMEWORKS dictate the rules of engagement as you design your apps.

It’s a place where FRAMEWORKS are in control of your decisions as you build out your architecture.

It’s a place where FRAMEWORKS force their opinions on how you do everything and leave you in control of nothing.

This door opens easily.

Door number 2 says ‘Architecture’.

Behind this door is control.

The CONTROL to make your own rulebook on how you structure your app.

The CONTROL to make your own decisions on what your architecture looks like.

The CONTROL to use your ***own*** opinions and not frameworks'.

Door number 2 is hard to open, but once you do, your code will be liberated.

Logic Room helps engineers open door number 2 because this is the door where you can use the rules of **architecture**, not the rules of *frameworks*. If you chose door number 1 then unfortunately you won't get much from this guide.

But… if you are ready to open door 2 with us and learn about **architecture** then welcome!

1.2. About Me

I am the founder of Logic Room; I have been helping engineers like you learn how to test and architect UI apps using battle tested techniques (which we have refined through our work as a JavaScript consultancy) for many years.

I created the things I am about to teach you because I became so frustrated with how hard it was to build, test and scale JavaScript UI apps. I got fed up with having to change my approach to building new UI apps every time I had to learn a new framework.

So, I decided to do something about it by developing the techniques that you are about to

learn.

All of the things you learn with us can be used in any project; you will get timeless skills that will serve you for the rest of your career.

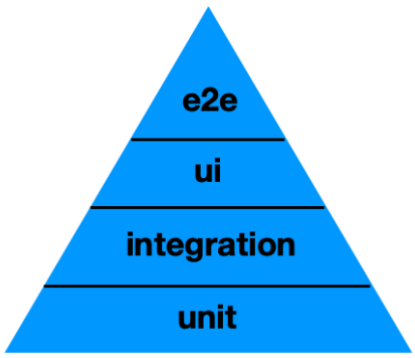
2. Lesson 1: Testing

It's easy to debate with a fellow engineer about the best way to do something and spend a long time going backwards and forwards discussing it. Often, I realise after much to-ing and fro-ing that we both may approach the same problem from different angles.

I don't want to do that in this guide, so in this section I will explain the most important thing you can do to build great UI architecture. You will learn to create architecture that is testable ***in the first place***!

2.1. Testing Triangle

Before we get going let's have a look at something famous from the topic of test automation. Many engineers will turn to the 'testing triangle' when establishing the correct strategy to test a UI app;



The testing triangle breaks the testing problem down into (roughly) 4 types of test.

The general idea is that there are more of the types of test from the bottom and less of the types of test from the top.

It's a handy tool to visualise and have a clear 'rule of thumb' to guide you as to what type of test and how many of them you should write for your app.

However, I have a problem with the testing triangle because it **breeds obscurity** and here's why…

When we use the testing triangle it tells us that we are going to need **more** unit tests than any other type of *test*. We know this because unit tests are at the bottom - which is *wider*.

As we build up our unit tests, they can only test individual 'sections' of our new code - since they are so low level. They work fine but over time they don't tell us how to test larger parts.

So, as we build out more complex features, we begin reaching into our more expansive

‘integration tests‘ which are further up the triangle. These tests allow us to ‘stick’ larger parts of our architecture together so that they are ‘integrated’.

When we have really hard stuff to test which includes lots of sections and lots of features, then we use end-2-end tests. These end-2-end tests normally drive the rendered markup of our app (the output HTML). They do this by physically manipulating the browser and parsing whatever the user sees.

Then, for the edges of our app and for things that only really affect the top-most layer (our markup and UI) we use a UI test.

The triangle works well, but has a limitation. It’s just a model and the problem with a model like the triangle is that it is often lacking something (as all models are). In this case the triangle does not account for what happens over ‘time’.

Because, over time as your application gets more complex you are not going to be able to use simple ‘unit tests’ to give you much confidence that your app is fully working.

Sure, you will be happy writing them and they give you a good feeling of *doing your job and writing tests*.

However, as your architecture gets more and more complex and your features and modules become more and more convoluted, your unit tests will live in isolation, buried with little context as to what they are doing. They will fail to give you *confidence* that your new code ‘works’.

You will run them and they may well ‘pass’ but often they can produce false positives, telling you your whole app is working, when in fact only the low level parts are working.

So, what happens is that you will naturally want to leverage more and more end-2-end tests to actually test these complicated features in your app.

You will have no choice but to use these because you need to assert that your highly complex application features work and as your code base grows, you will end up with a triangle that begins to become bloated at the top and excessively reliant on end-2-end tests.

2.2. E2E Test Inversion

Your triangle will become an inverse of what it is meant to be. It will become more and more overloaded with end-2-end tests and will heavily rely upon testing the outer loop of your app; creating friction between your tests and the implementation.

At this point you will end up with a test suite that will become top-heavy and we all know what happens to things which are top-heavy….

The result is bad for engineers because end-2-end tests are not as simple or as easy to write as unit tests; they are cumbersome and brittle to maintain.

Most engineers don't want to be lumbered with complexity unless they need to be and unfortunately that's exactly what end-2-end tests are. They ***are*** the most complex type of test; to test specification of a system.

The question is, can we get the benefit of end-2-end tests without actually writing end-2-end tests?

The answer is YES.

The solution is to break away from the triangle and take a different perspective. You can consider that your code should allow testing with the following two principles:

- Firstly, aim always to write tests using the simplest type of test; the humble unit test (e.g. Jest, Mocha, Jasmine)
- Secondly, instead of your unit tests checking low level code as you write them, they will be written to actually test the specification of your app

In other words, our priority should not be to have lots of unit tests which simply tell us that **parts** of our app are working, but we should have lots of unit tests that tell us that the **specification** of our app is working.

The secret to doing this is to lay out our apps in a slightly different way in order to make sure our code is implicitly testable.

At Logic Room we have a phrase for this: a "testable architecture"

2.3. Testable UI Architecture

In order to set up our code so that we can use simple tests to test the specification, we are going to need to focus on the 'right' architecture.

When we teach people about UI architecture we always start with testing because if your code is testable, it will force a better architecture. In fact, the two are deeply connected. You make one, then you get the other!

So, if we want a good UI architecture then we need a system to make this happen. We are going to need rules, we are going to need principles and we are going to need naming and concepts that ***mean*** something.

It's only through the power of standardising the way we think and discuss our architecture, that we will learn to build ***better*** architecture. There are many ways we can do this.

But at Logic Room our way is to use something called the "Fast-test architecture" …

The Fast-test architecture is based on the idea that all good structure starts with the simple and most foundational rule; that the framework is NOT in charge!

The reason we do this is because when you want to test something that uses any type of framework, normally you end up testing the framework ***at the same time*** without even realising.

In fact, this happens in all languages and all platforms; for example here's a quick list of things I have seen happen myself…

A team wanted to test a Ruby app but ended up testing Rails itself.

Another team I worked with wanted to test a C# web API, but ended up testing the Web API itself.

Many students who come to Logic Room are still stuck in the mode of wanting to test JavaScript apps, and actually testing whatever UI framework is hot at that particular time!

We can do better!

The simplest way to test your code is to do just that; test YOUR code.

What this means is that you must isolate YOUR code independent of the framework and then simply test ***it***, instead of testing each detail of the ***framework*** too!

There is actually a name for this approach, it's called the humble object pattern.

2.4. The Humble Object Pattern

The humble object pattern says:

"We extract all the logic from the hard-to-test component into a component that is testable via synchronous tests. This component implements a service interface consisting of methods that expose all the logic of the untestable component; the only difference is that they are

accessible via … method calls"

When this quote says 'component' it does not mean UI component. It is talking more generically of any type of module/component within an app.

So, the rough translation of this for our UI app is that we are actually going to keep all our business logic and presentation logic in an entirely separate architecture that is callable by the UI framework. We do this through the power of abstraction…

In our 12-Week UI Architecture Academy we show students how to attain testable code using UI architecture which is abstracted away from the framework using something called: the 'black box'.

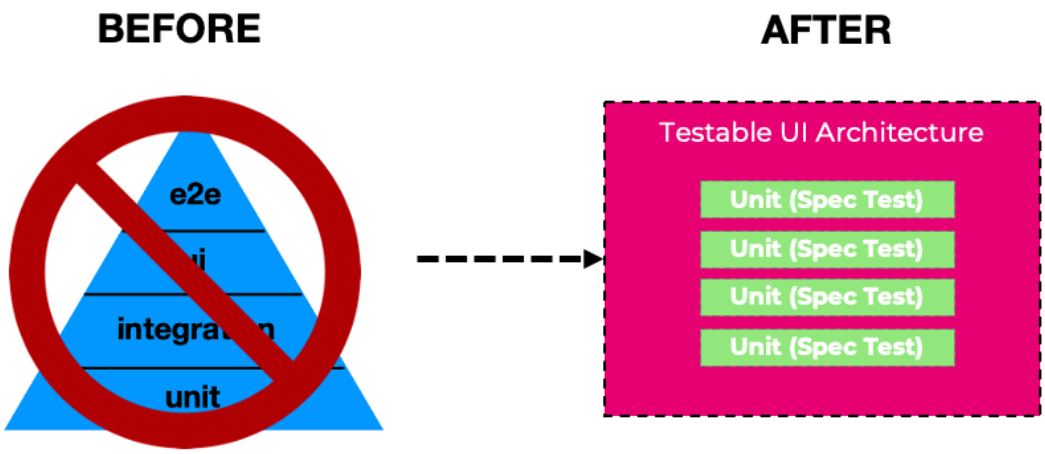
This is a descriptive noun that helps us have an architectural understanding of applying this principle of the humble object. We will cover the black box next but first up the conclusion to lesson 1.

2.5. Lesson 1: Conclusion

Whilst useful, the testing triangle can make us too reliant on end-2-end tests. We can re-wire our approach by focusing instead on a testable architecture. This is easiest using the humble object pattern.

Logic Rooms implementation of this is the Fast-test architecture which comes with rules and principles to help us achieve testable code which isn't coupled to the UI framework.

By leveraging the Fast-test architecture and creating a testable UI architecture, you can use simple unit tests to test specifications of your system which will make your tests and code easier to understand and scale.



3. Lesson 2: Black Box

Mental models are explanations of thought processes that explains how something works. They can be useful when it comes to explaining software architecture because they convey a lot of context about whatever it is, we want to communicate.

In this regard we use the term 'black box' to describe a mental model to which we ascribe to a clear boundary between two types of code in our app.

The first type of code is the presentation logic and the business logic that we write. This could include all of the data transfer objects, the business objects, the entities, the functions, the transformations and the view abstractions (including ViewModels). It's normally the code which is under our complete control and does not necessarily need to be put into the markup or the component within our UI framework.

The second type of code is the code that directly serves the view layer of our app and lives in the component of the UI framework.

This is everything that is concerned with markup, buttons, display logic, validation and browser events, etc. Normally we are not under complete control of this code, as different UI frameworks will force us to implement this code in different ways.

So our black box contains all of the ***first type*** of code from above, let's look at an example and see why this distinction and invisible boundary is so important:

3.1. The Problem with UI Frameworks (React Example)

Consider the following UI component;

```
1   // Framework Code + Our Code
2   function CarComponent() {
3     const [currentSpeed, setSpeed] = useState(0)
4
5     useEffect(() => {
6       setSpeed(api.get('api/current'))
7       if(currentSpeed < speedLimit) {
8         setSpeed(motor.increaseSpeed())
9       }
10    })
11
12    return (<div>You are driving {currendSpeed}</div>)
13  }
```

Although this example is in React; code that looks like this is quite common in many UI apps regardless of which UI framework is in use. This code is problematic because it mixes both types of code into a single block. For example, it has mixed something which should live in the black box **api.get** (line 6) and has connected it to something which should only live in the component **setSpeed** (also line 6).

You may have already guessed the problem with this code; it is twofold;

1. Our code **api.get** is not easily testable without using end-2-end tests or UI tests because we have buried it inside a framework component. We will struggle to intercept or check what this line is doing without actually invoking testing at the component level.
2. It's hard to know where the responsibility of the code which does things like **app.get** begins and ends, compared to the UI framework code **setSpeed.**

Our brains are not very good when we throw too many types of code (presentation/business logic and UI framework code) like this 'all in' together. They struggle to deal with code of different categories that is running different abstractions all at the same time.

3.2. Abstracting the Framework

This is where the black box comes in. The black box says we should provide a clear boundary with certain types of files that hold code that does certain things behind an invisible wall. This wall logically appears when we follow certain architectural rules.

I won't go into these rules here, but broadly speaking they seek to create what we know as a cornerstone architecture, which lets us create 'categories' of files.

These files have overarching concerns and give us the rough guidance of what should live in them. In other words, it's sort of like a 'recipe' to tell us what code lives in which file!

By doing this, we now naturally solve the problems that we got above (testability and comprehension) because we will automatically begin creating this abstracted code that lives in our black box and within the right file too!

We end up with the right mental model to begin placing the right code in the right location! This simple approach begins to yield powerful benefits.

Let's have a look at an example:

```
1   // Framework Code
2   function CarComponent() {
3     const [currentSpeed, setSpeed] = useState(0)
4
5     useEffect(() => {
6       setSpeed(new Car().accelerate())
7     })
8
9     return (<div>You are driving {currendSpeed}</div>)
10  }
11
12  // Our Code (Black Box)
13  class Car {
14    accelerate () {
15      const currendSpeed = api.get('api/current');
16      if(currendSpeed < speedLimit) {
17        return motor.increaseSpeed()
18      }
19    }
20  }
```

Between lines 2 to 9 you can see where the original component code lives. However, you will see now it's much more 'minimal'. What I mean by this is we have removed much of the code that was buried in it.

We have removed this code into a separate class called Car, line 13 to 19. In this respect, it's this code that is **in** the black box. We could say that we have isolated Car into its own independent system which has abstracted the presentation and business concerns that were originally in the component.

Setting up our code like this will give us clear advantages;

- The ability to re-use code from the Car class.
- Easier understanding of what the Car is meant to do.
- Simpler testing of the code in the Car class (the black box).

We call this concept the separation of concerns. This separation also helps us with something called symmetry.

3.3. Black Box Symmetry

In our 12-Week UI Architecture Academy when we get engineers to abstract their code like this, we begin from the fundamentals. We begin from a place of testing. The first few weeks in the course are actually about testing since this is the highest priority for a UI architect.

What students see is that because of this abstraction they can now connect their component to the black box in the same way as they connect the tests.

In other words, their UI architecture is 'symmetrical'. This takes a lot of effort away from

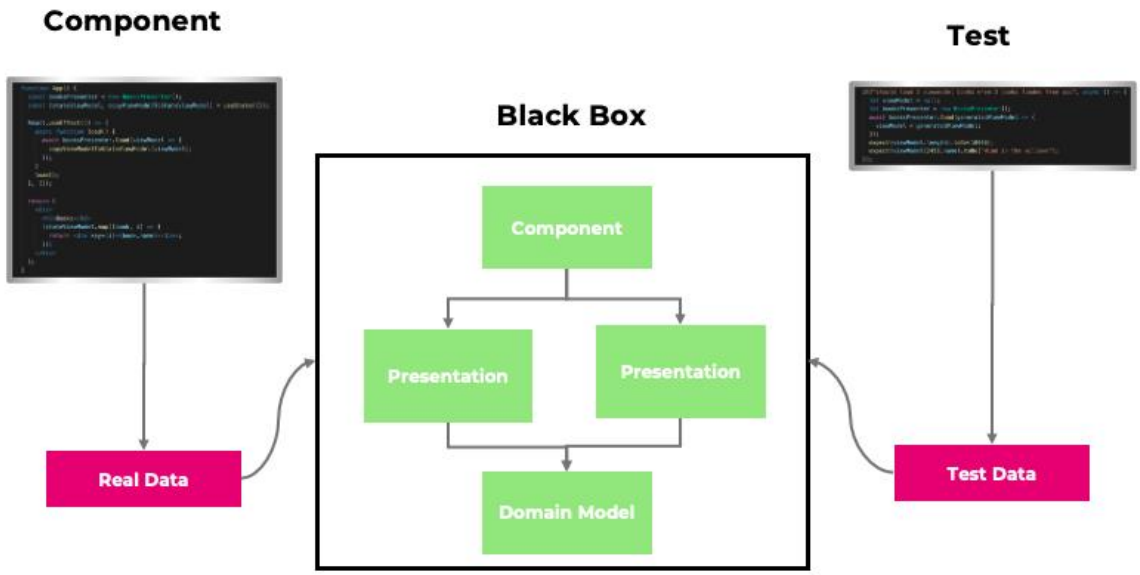
them because now instead of having to think about testing and UI architecture separately, they simply solve one and the other falls into place!

3.4. Lesson 2: Conclusion

UI framework components often suck in too much code. Without realising it we will end up with UI concerns, presentation concerns and business logic concerns, all put into framework files or tools that surround the framework.

The black box is a mental model to begin segregating our business and presentation logic behind an invisible barrier. If you design this barrier right you will achieve symmetry, this means that your component and your tests can invoke the black box in the same way!

This makes your code cleaner, less coupled, easier to understand and critically… easier to test!



4. Lesson 3: Flat Presentation

I remember a few years ago with a client, one of their engineers was getting frustrated that the management kept 'changing' their mind. The app we were working on had a repeater control on a screen which was displaying records from multiple API calls.

The managements indecision stemmed from the sales team. They were responsible for steering the product. However, they couldn't quite decide what selection of these records needed to be shown on this particular screen.

I sat down with the engineer and went through their frustration with them. It turns out that this particular control had been a point of contention for some time. This particular engineer had stressed a number of times that the business should 'make up their mind'.

When I asked why this mattered so much the engineer showed me the code and the architecture they had chosen. It was quite easy to see why this engineer was so adamant that the decision be final. What his team had done was to hard-wire the markup in the UI layer directly to data that came out of the API.

This meant that every time the business wanted something to be changed, the markup would need to be changed too. Because of this, when the business then changed its mind, the markup had to be changed too!

The problem was less to do with the actual change (after all, it *could* be changed) and more to do with the direct coupling in their chosen architecture. The engineer could not support these changes because every time a change was requested, the view would invariably break (through bugs caused by incorrect code updates).

This is when I realised something important; the changing requirements were not the true source of the engineers upset, but instead the breaking UI was!

This is important because if there is one thing I have learned about business, it's that they **need** to change. In any competitive market, a company will use innovation to move past their competitors in a sort of 'biologically inspired' race to the top.

So, as engineers we should never be annoyed by things that need to change, because if we are, then we cannot truly support the businesses we work for – the very businesses that put money in our bank accounts!

In fact, the very reason our job exists, is 'because' businesses change. They need to develop all this wonderful technology including the web apps we are employed to create if they want to survive.

As it turns out, the reason that the engineer could not change the code in this story was because his UI architecture was not right…

4.1. Tell Don't Ask

Upon inspection with the engineer, I found a commonly occurring situation which I have seen many times over in my career.

What this engineer and his team had done was to embed too much information about their business problem (including the entities which came from the API) into the view itself.

When I looked at the code on this project this is what I saw…

```
{ entity.type === "router" <h1>{entity.title}</h1>          }
{ entity.type === "node"   <h1>{entity.mainNodeTitle}</h1> }
```

Upon first inspection this code 'looks' innocuous.

We have a simple entity which exposes an **entity.type** in our markup. The rendering logic then switches on this type and returns a sub block of html, which is either the **entity.title** or the **entity.mainNodeTitle**.

However, this code is problematic because it is trying to be ***too*** clever.

Can you see why?

The problem is that the markup is analysing the entity type and then making a decision on what it should show between the <h1> using the very ***same*** entity.

This creates a problem because whilst it looks simple, we have subtly introduced far too much information into the markup about this entity! What we have done is create a view with markup which needs to **interrogate** the underlying entity to work out what parts of that very same entity to show.

Let me say this now, this is the WRONG way around!

Your view should never need to figure out what it needs to do like this. It should not need to take this complex logic which has nothing to do with the actual display and twist around it in order to show information. Instead it should be ***told*** what to do!

4.2. The Rule of Manageable Code

Manageable code doesn't really happen because of brilliant design ideas. We can learn things like the black box and how to make our code testable, but really all of these high-level things come down to small individual habits that are applied at the low level consistently.

One of the most important low-level consistent things you can apply when building views is the rule of **tell don't ask.** By always thinking about this as you develop your views, you will consistently develop views that look… dumb!

4.3. Making Dumb Views

Dumb views are views that get told what to do. In our example, we want to tell the view: 'here, show this and nothing more'.

If we implemented the code from earlier as a dumb view that gets told what to do, then we would most likely end up with something like this…

```
1   const viewModel = {title:null}
2
3   if(entity.type === "router") {
4     viewModel.title = entity.title
5   } else if (entity.type === "node") {
6     viewModel.title = entity.mainNodeTitle
7   }
8
9   {<h1>{viewModel.title}</h1>}
```

In this example, between the lines 1 and 7, we are using some setup code which is going to do all the nasty stuff we **don't** want in our view.

This code is going to extract through interrogation the right information. It will then create another very simple data structure which contains nothing more than what our view *needs*.

It turns out our view doesn't actually need anything, it just needs to be told "here is the .**title** – now get on and render it!". This code is a pure example of **tell don't ask** in action.

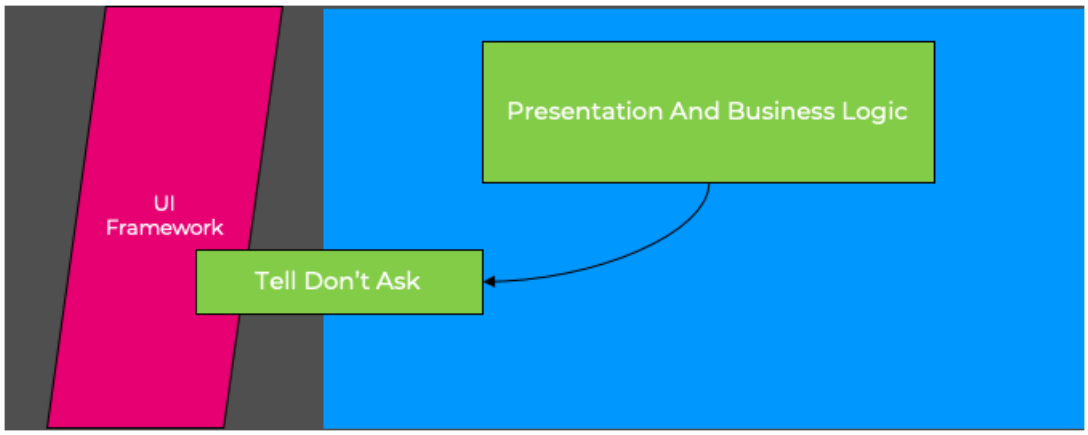
By implementing code like this we will massively reduce the brittleness of our view whilst at the same time radically improving its testability too. Because now, we do not need to invoke the UI framework to see what data the view is rendering. We could simply abstract the ViewModel code and put this into the black box!

4.4. Lesson 3: Conclusion

The biggest problem with markup and views is making them too clever. We see the subtle danger of being in a position where our views need to interrogate objects on the fly to work out how to render properly.

This is the wrong way around! Avoid this by blocking complex business objects from being rendered by your UI framework markup.

Instead, flatten all objects that need to be displayed into flat data structures that expose just enough for your markup to work. These should allow your view markup to be 'dumb' and mean it can be told what to do without needing to ask!


Presentation And Business Logic
UI
Framework
Tell Don't Ask

5. Lesson 4: Reactivity

JavaScript runs in a single thread meaning it must rely primarily upon event driven paradigms to operate. However, the systems which host JavaScript and its single threaded runtime, are not themselves single threaded. For example, the OS' that supports Node.js and a normal browser can do more than one thing at a time!

From our single threaded context, we *can* call threaded resources, but if we were to wait for them to complete, our app could 'lock-up' whilst it's waiting.

To avoid this situation, we use callbacks. Callbacks are functions that we pass to external boundaries and say *'here - call this when you are finished'.*

We say JavaScript is a naturally 'event driven' language - it configures as a series of events which are called back at some point during the future. But this natural aspiration of the language causes two architectural problems we need to solve; ***timing*** and ***encapsulation***.

5.1. Timing

The first problem an event driven architecture will present to us is that we are using callbacks to execute blocks of code at some point in the future. The problem is, we never know exactly 'when' this code will be called.

This is a problem because when we defer execution of code, we are not only handing over the process but also handing over the *expectation* the process has about the state of our app. In a normal UI app this could mean our processes accesses the state at the wrong time, or in the wrong order which will create odd bugs which are hard to diagnose.

We need more predictability so we do this by creating some sort of wrapper around how we pass our callbacks; this will help us control timing.

One of the primary ways to do this is by using things like promises and async and await, however these are slightly limited because these types of callbacks will only ever invoke a callback once. What we need is a system which can manage and run our callback code with perpetuity; and we do this with the observer pattern.

5.1.1. Observer Pattern

The observer pattern is a simple pattern which lets you do two things:

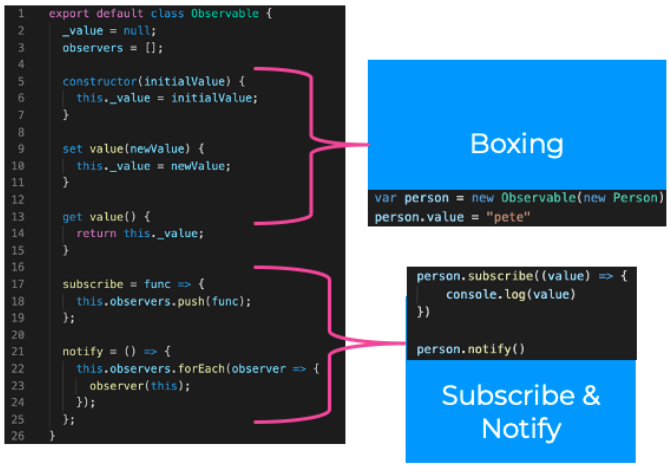
- Firstly, it lets you **box** some underlying 'state' into a container object so that you can access it later on.
- Secondly, it lets you manage a list of callback methods and have them automatically called at the right time.

It creates a wrapper around any variables you want to store as state in your app so that it

can intercept calls to them. If you think back to our original premise (callbacks getting called at the wrong time), this makes sense.

Because what happens is that when anything changes the underlying values, the observer knows, and if it knows, then it can then *act*!

This means it can make the decision to invoke any callback functions which have been asked to be called **when** the data changes. This secondary mechanism is called **subscribe and notify**.



The most interesting thing I can tell you about the observer pattern is that it's used EVERYWHERE in the JavaScript world. You see it implemented in various ways in UI frameworks, in state containers, in real time libraries, etc.

For this reason in our 12-Week UI Architecture Academy, we get students to write and implement at least one, in a real-world assignment. It's our belief that when you understand the pattern deeply and have practiced it, then you will be able to identify it in your exposure to frameworks in the future; making you a better UI architect!

5.1.2. Observable Frameworks

Being able to write and understand the observer pattern is essential knowledge. However, we don't recommend you do actually roll-your-own in your production apps. From our experience we have found that there are features that need to be built on top of the basic pattern that will take a long time to implement.

For this reason I suggest you use an observable framework which will give you a lot of features out of the box.

In our training and for the majority of our commercial engagements we use Mobx which comes with many of the features you will need for complex application development. It's advertised as a simple scalable functional reactive programming tool.

It builds on top of the simple observable pattern to give you more advanced features, such as creating data streams which are going to help us with the second problem, we see in callback aspirated apps; encapsulation.

5.2. Encapsulation

The encapsulation problem builds on top of the timing problem. It says that even when we control the scope and timing of our data and callbacks, we can still get shared state problems and unwanted side-effects.

Engineers building UI apps often worry about these side-effects because even though we all aspire to build stateless web apps, it's not practical to do because normally for performance reasons we do need to merge data inside our Single Page App.

So almost all UI apps will need to store state somewhere but we need to make sure we properly encapsulate the flow of data through our app; we do this using a reactive architecture…

5.3. Reactive UI Architecture

The concept of a reactive architecture helps us control our encapsulation by letting us connect a stream of functions together. Then we are able to pass data between them in a controlled manner.

This flow of data is one-way, meaning that we can begin with some sort of state and then resolve to some sort of transformation of this state later down the line in our app; this will be *predictable*.

So in order to implement a reactive UI architecture we should implement it so that it will operate on immutable data structures. This means that at every layer in our app, we completely rebuild data structures instead of modifying them.

We can then use these immutable data structure streams to represent different conceptual

models in our app. The reason for doing this is simple; it will enable a more scalable UI architecture because now we have an architecture which has clear and ***idempotent*** data models at each layer.

5.4. React, Redux and Tooling – A Warning

React and tools like Redux have a deep connection to the observable pattern and reactivity. Redux itself exposes the observer pattern if you look at its core API.

Many engineers rely upon React and/or Redux to solve many of the issues we have addressed here; including having side-effect free, state enabled architecture.

But I do have a quick warning for you:

Using tools 'can' create a problem because sometimes these tools 'overstep the mark'.

For example in our time consulting with many teams we have seen container solutions using React and Redux become too complex. They often have an excessive reliance on these tools to solve architectural and data model design issues.

What I mean by this is that there often isn't a clear separation inside these containers about what each type of data structure is for. We mentioned earlier about the importance of data streams with immutable models at each state to represent concepts within our domain model.

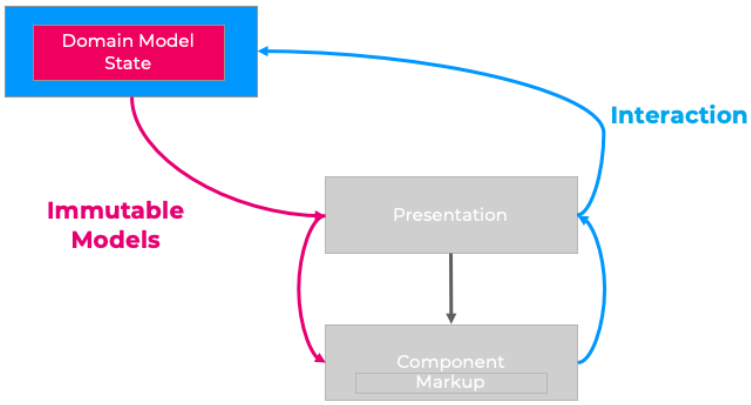
The problem with Redux or just using state from React, is that it forces you to throw in too many concepts side by side. We often see presentation data structures alongside business logic structures. Even Redux's creator Dan Abramov warned on this issue.

So…we are not saying don't use Redux and React for state if you must but we are saying that there are certain things to avoid and if you want to take our training, we can teach you on this.

Lesson 4: Conclusion

A JavaScript UI event-driven app will naturally present timing and encapsulation issues which will manifest as inconsistent callbacks, odd race conditions with your data and unwanted side effects which will leave you feeling frustrated.

To tackle this, we implement the observer pattern, with immutable data models and a reactive stream which will help you build a one-way predictable state and execution flow through your app!



6. Lesson 5: Dependencies

The final thing you should try and understand if you want to think like a UI architect is how you manage dependencies in your app.

Here's a quick bit of code that we are already familiar with but has been extended to show more complexity.

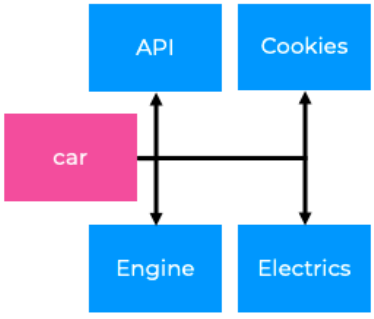
```
1  class Car {
2    accelerate () {
3      const currendSpeed = api.get('api/current');
4      cookies.store('speed', currentSpeed)
5      if(currendSpeed < speedLimit) {
6        petrolengine.start()
7        electrics.accelerate()
8      }
9    }
10 }
```

This code has many dependencies on external objects. Specifically we can see it is dependent on **api**, **cookies**, **petrolengine** and **electrics** (lines 3,4,6 and 7).

The fact that this code has so many dependencies means it is tightly coupled. This tight coupling presents two problems….

- The first is that if we ever change the interface to those objects (the properties and methods on them) then we immediately break the Car
- The second is that it's almost impossible to test this code without physically overwriting these objects and mocking the data they deal with

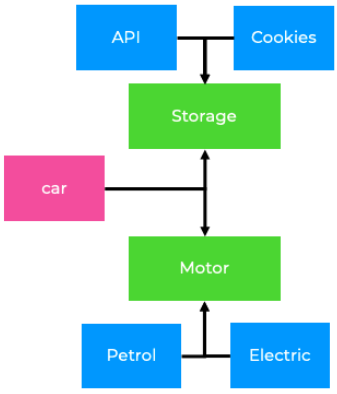
We call code like this brittle because it is accessing too many things.



In order to reduce the complexity of this code we need to use proxy objects which expose a 'higher' interface. What we do is plug the Car and the original dependencies into these proxy objects. This technique is called dependency inversion (DI).

6.1. Dependency Inversion

Applying dependency inversion to the current code could easily mean we halve the number of dependencies by using two proxy objects.



What we have done is to remove the low-level information in **api**, **cookies**, **petrolengine** and **electrics** and put this code into the two files (Storage and Motor in green). Now, when we do this, we have done something very magical which isn't always obvious at first sight.

If you think back to our original premise above, we discovered that the first problem with coupled code was that changing the interface (properties and methods) of the child dependencies (**api**, **cookies**, **petrolengine** and **electrics**) would break the Car.

However, by wrapping these dependencies in another file those dependencies can no longer break the car.

They can only break inside the file they are declared. The reason for this is simple; those low-level objects return information to the interface (properties and methods) of the Motor and the Storage file NOT the Car file itself.

By moving these low-level dependencies into these files, we have actually inverted the dependencies because now they are FORCED to return through an interface we control, not the one that they control (as they did when we called them directly). This is what Dependency Inversion means!

And that is a powerful concept to be aware of because inverting dependencies like this will not only make your code easier to understand but also be easier to test. Why? Because it means that YOU are in control of the interfaces in your architecture, not the dependencies that you need to call!

Applying DI means the car class is now so much simpler because of this inversion.

```
class Car {
  accelerate () {
    const currendSpeed = Storage.getCurrentSpeed()
    if(currendSpeed < speedLimit) {
      return Motor.increaseSpeed()
    }
  }
}
```

This code example demonstrates at least half the amount of coupling that we had in the first example!

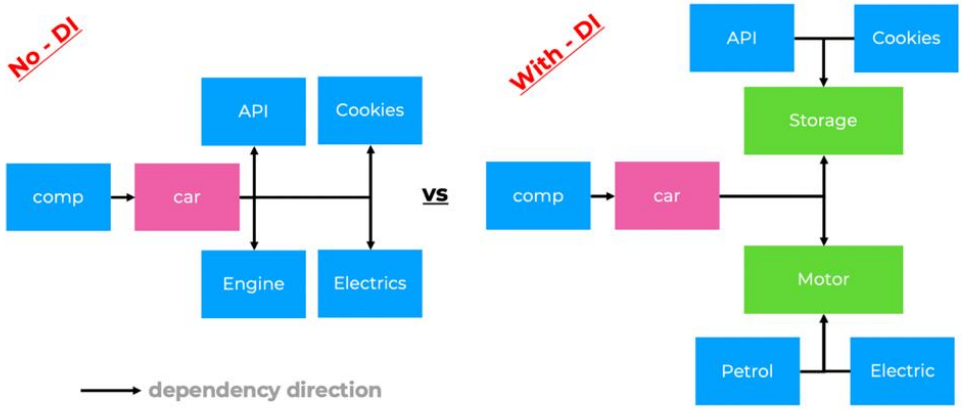
6.2. Lesson 5: Conclusion

As we grow our UI app, we are going to need to connect many dependencies. The problem is that when we connect too many dependencies our architecture becomes brittle.

So, to battle this problem we need to begin putting our dependencies behind interfaces we control. This is as simple as wrapping up contextually similar dependencies into proxy objects.

By separating control like this we automatically turn the dependencies the other way around. Now those lower dependencies are dependent on the interfaces we control, not on the ones that they declare.

This makes our code easier to test and scale!



7. Conclusion

In this guide we have covered 5 critical insights that you should try and fully master if you want a robust UI architecture.

In the first lesson we learned that testing is the highest priority for UI architecture, and that good code is implicitly **testable**.

In the second lesson we learned that the **black box** is a useful mental model to help us segregate our presentation logic and business logic away from our UI components. The black box helps us test our code because of this separation.

In the third lesson we saw how **reactivity** will help us protect against timing and encapsulation issues which create unwanted side effects and odd race conditions in our app.

In the fourth lesson we saw how we should have dumb views that are **told what to do.** They should not be having to work out what to do based on deeply nested business data structures.

And finally, in the fifth lesson we saw how our **dependencies** should be managed using the principle of dependency inversion (DI). Doing this will make our code much simpler, easier to test and less brittle!

If you have enjoyed this guide, why not consider taking your training further with us by enrolling on the 12-Week UI Architecture Academy!

In the academy we teach you breakthrough techniques to build, architect and test UI apps in any framework.

Head over to https://www.logicroom.co/ for more info.