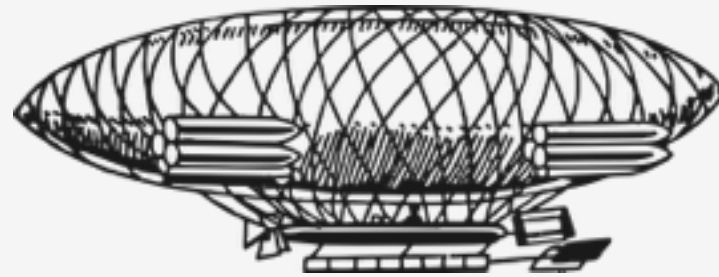



The *Post*-**MVC** Age



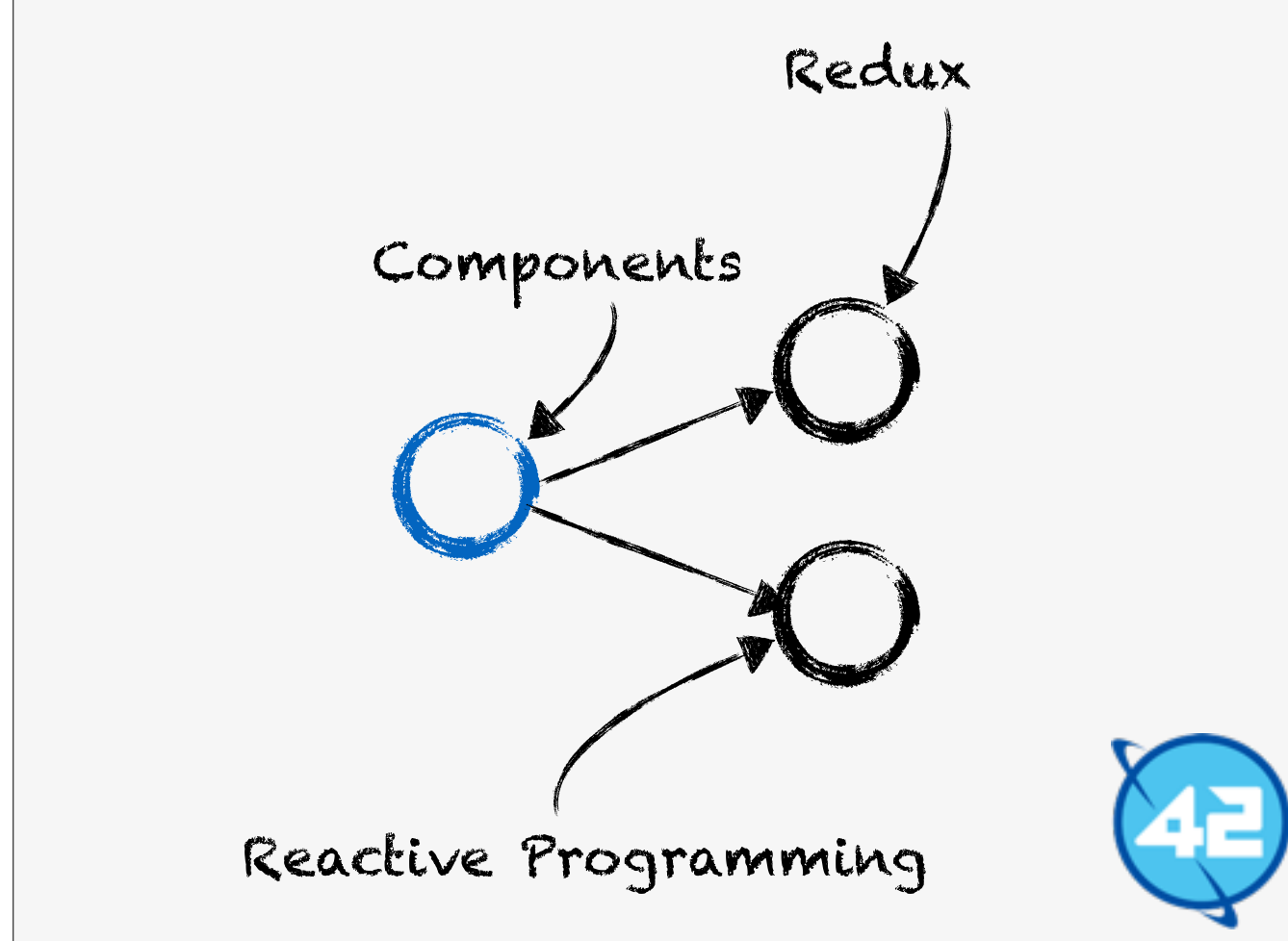
This presentation is called "The Post MVC Age" and in it I want to show you guys a new way of building applications using the "Component Architecture".

Maarten **Hus**

 *@MrHus*

Developer @ 

Hello my name is Maarten Hus and I'm a developer working at 42. 42 is a Dutch company and we make Java and Angular applications.



Here is the roadmap for today.

Then I want to talk about the Component, what they are, and how they can be used to create applications.

We will also discover that a "Component" based architecture has some downsides and how Redux and Reactive Programming are a possible solution for these downsides.

Case Study



Lets look at an actual component from Google's Polymer Library which creates a Google Map widget.



Here's an example of what the Component visually can produce.

So on this page you see four cities and a map which is centred on the city centre. Also there is a google map marker to indicate where this center is. You can drag and drop the marker and when you hover over the marker it shows the name of a local sports team.

Lets focus on the map in the top right the one which shows San Francisco, what is the code needed to render that map.

```
<google-map latitude="37.77493"
              longitude="-122.41942"
              fit-to-markers>
  <google-map-marker
    latitude="37.779"
    longitude="-122.3892"
    draggable="true"
    title="Go Giants!">
  </google-map-marker>
</google-map>
```

So here's the code: Polymer makes their Components look like regular HTML with custom tag names.

The Component takes some input from the outside world: the longitude and latitude. It is declarative because we can all guess what this Component will do.

```
var map = document.querySelector('google-map');  
  
map.addEventListener('google-map-click',  
  function(e) {  
    alert('The user clicked on the Map!');  
  }  
);
```

The Component also has clear output, take a look at this code:

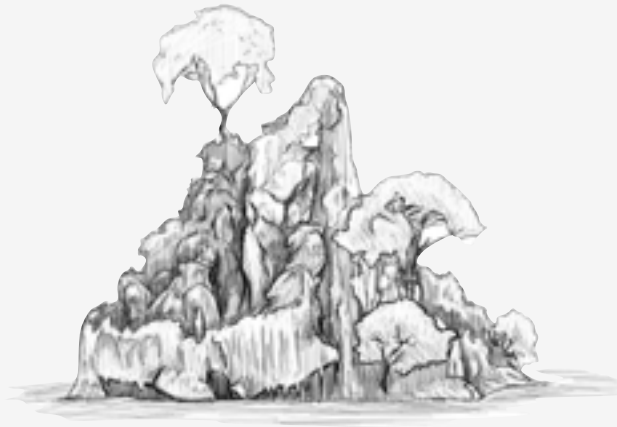
Here you can see the "output", we can do "something" when the marker is clicked, in this case alert a string.

Characteristics of Component



So what are some of the characteristics of Components.

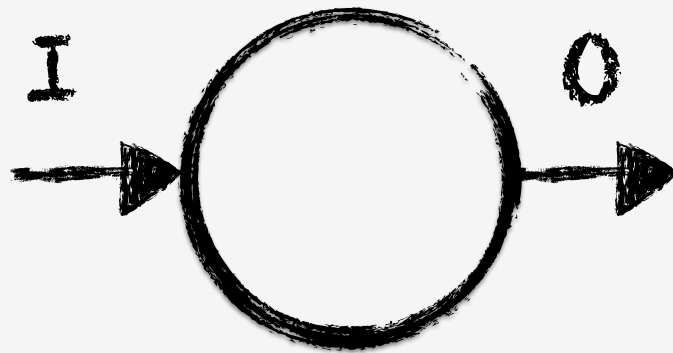
A Component *is* ***Isolated***



A Component is isolated they: two components of the same type do not interfere with each other.

Meaning can use two components of the same type on the same page. We saw this with the maps example. We put four map components side by side without any trouble.

A Component *has* clear *I/O*



A Component has very clear I/O all the inputs and outputs of a Component run through explicitly defined channels.

In other words you cannot get into the component without going through the proper channels.

The component cannot go to the outside world without going through the proper channel as well.

In the maps example the input were the coordinates. The outputs was the click event.

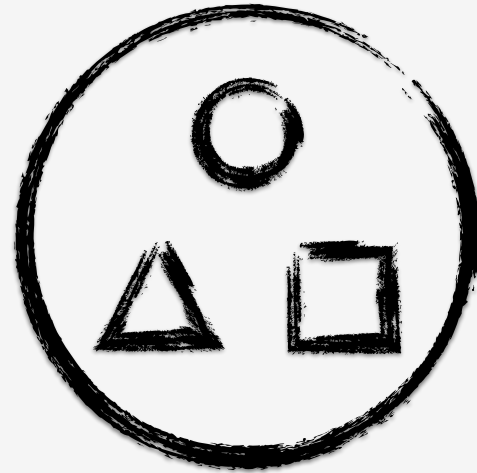
Components *are* *declarative*



A Component is declarative. Reading the usage of a Component should give you a good indication of what the Components represent.

For example the code example of the San Fransisco map was very declarative. I'm sure that everyone here by looking at the code had a pretty clear idea of what it was going to do.

Components *are* ***composable***



A Component is composable. You can use multiple Components and combine them together to form one larger Component. For instance a Button Component and a Label Component can together form the basis of a Counter Component.

Components *are easy to reason about*



Components are easy to reason about because of these four characteristics. Because they are isolated you know they don't affect the outside world. Because they have clear I/O semantics you know how to affect them. Because they are declarative they are easy to use. Because they are composable makes them easy to create bigger abstractions.

Conclusion



Components are awesome.

Building applications with Components



So Components are pretty powerful and we know we can make applications by using them using the Component Architecture. But how does such an application look like?

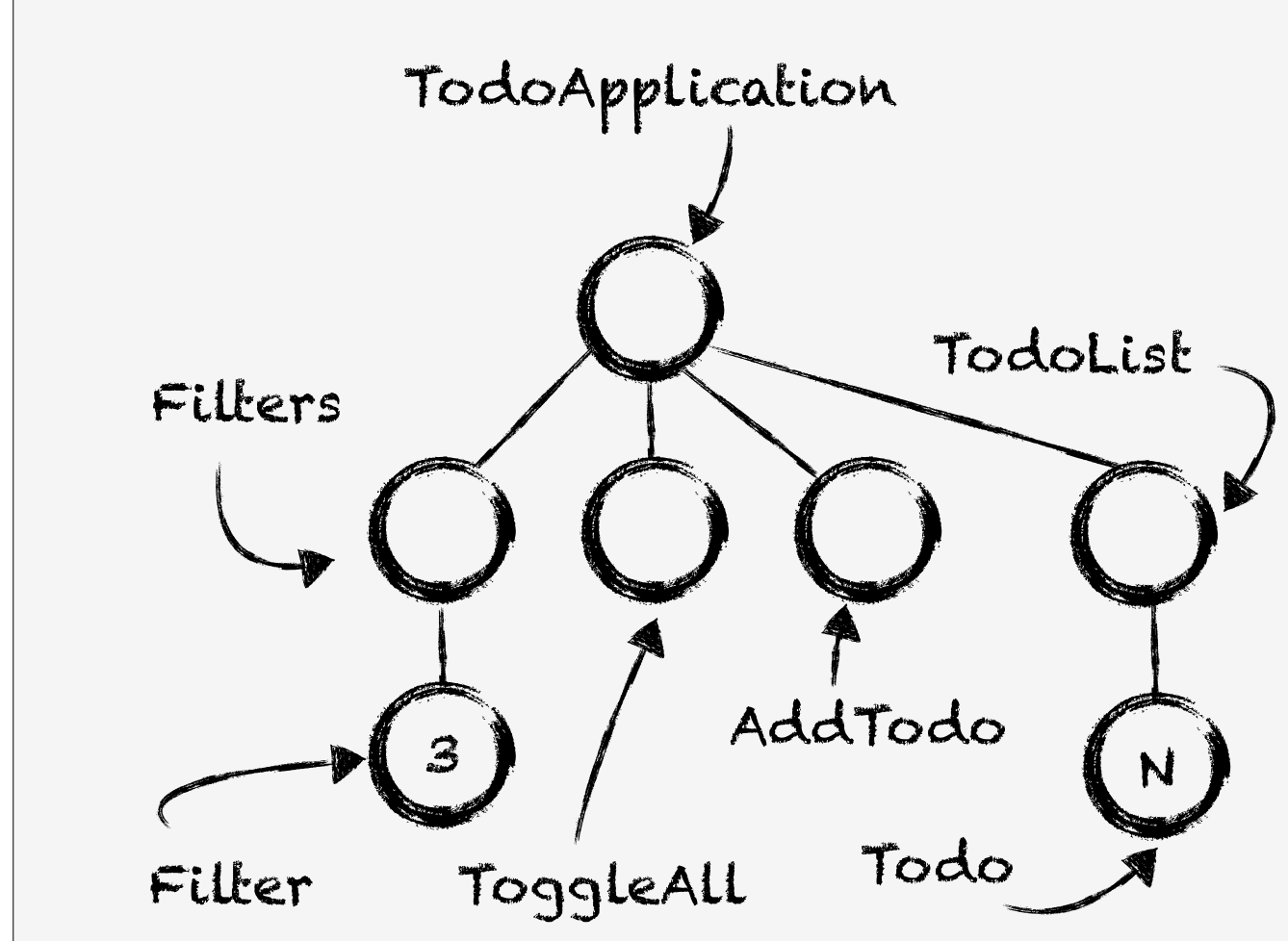


Lets take the universal TODO application as an example.

With it you can add todo's, cross them of your list, and filter them based on their complete status.

It is *built* from the
following ***Components***





The TodoApplication is the root Component. All other Components are descendant of this Component.

The AddTodo Component is responsible for adding Todo's behind the scenes it is an Input with a submit button.

The TodoList component renders the current todos, it does so by rendering each todo via the Todo Component using a loop. The Todo Component has a button to toggle the Complete status of the Todo, and a button to remove the todo completely.

The Filters component renders a list of 'filters' which the user can use to filter the todo's. There is a filter for showing all todos, a filter for showing all complete todos, and a filter for showing all active (non complete) todos.

Two Questions

1. Where does *the state* live?
2. How do components *communicate*?



There are two interesting questions I had when I first saw this "Component Architecture" tree.

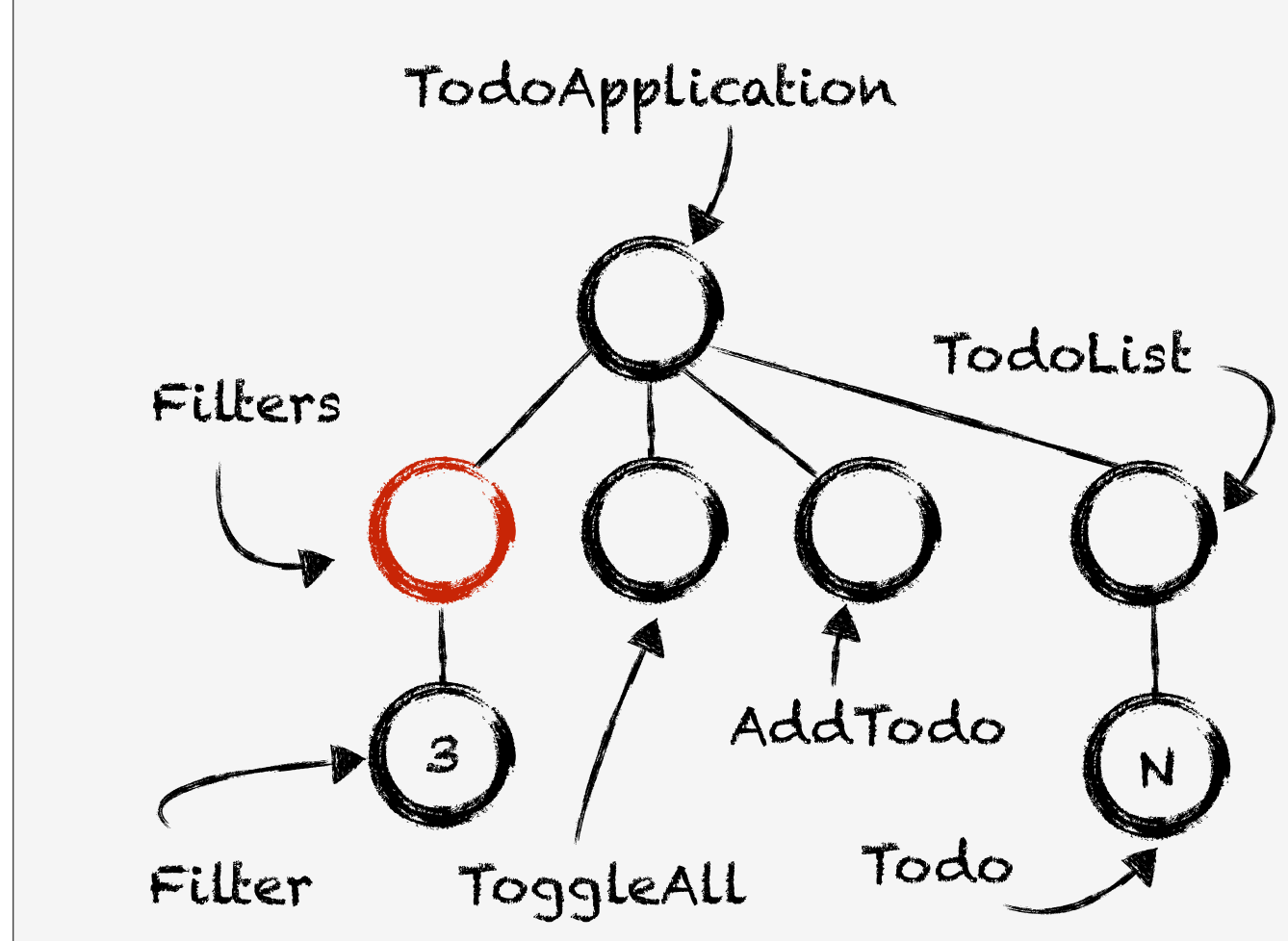
The first question is: "Where does the state live in this tree?"

The second: "How do Components communicate with each other?".

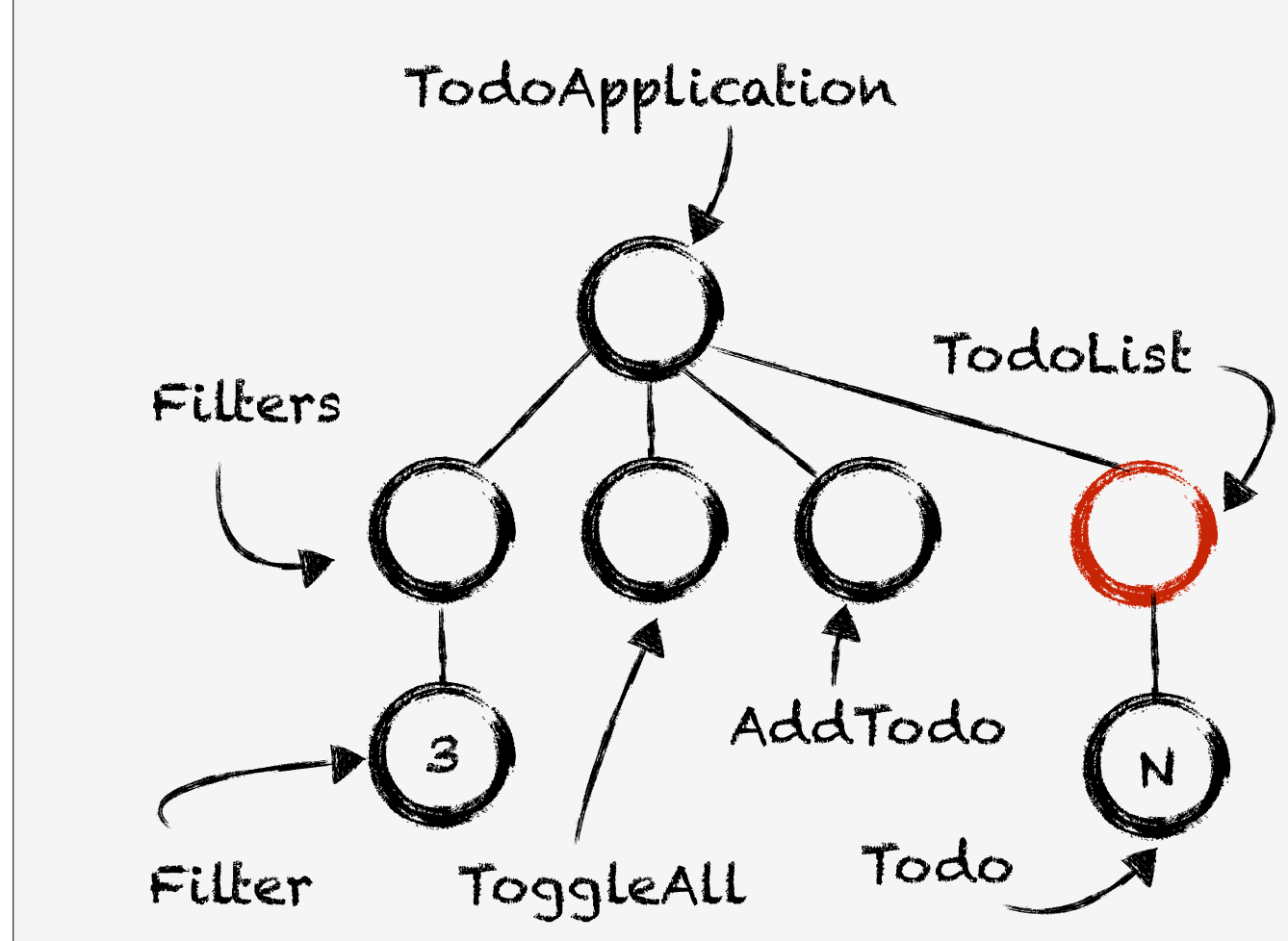
Where do we put the
current ***active filter***?



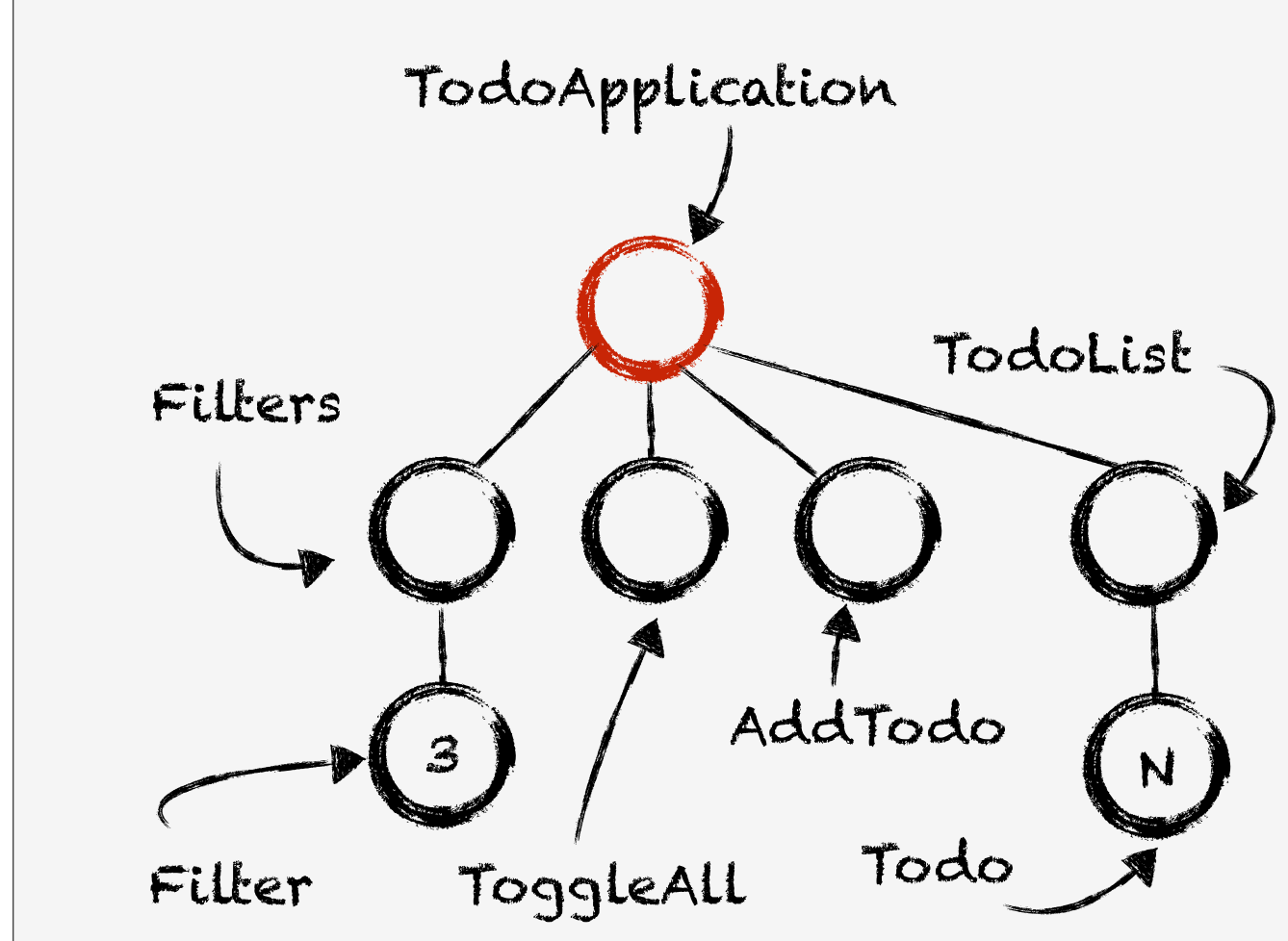
To make "Where does the state live?" more concrete lets ask ourselves this question: "Where do we put the current active filter?"



Do we keep it in the Filters Component since that is the closest to where it is manipulated.



Or do we put it in the TodoList Component where the filter is actually applied.



Do we put it in the root component the TodoApplication because it is the parent of both the Filters and TodoList Component.

The *answer* is not very *clear*



There are cases to be made for each of the three options. But the last option to put them into the TodoApplication Component makes it easier for us so we will go with that option.

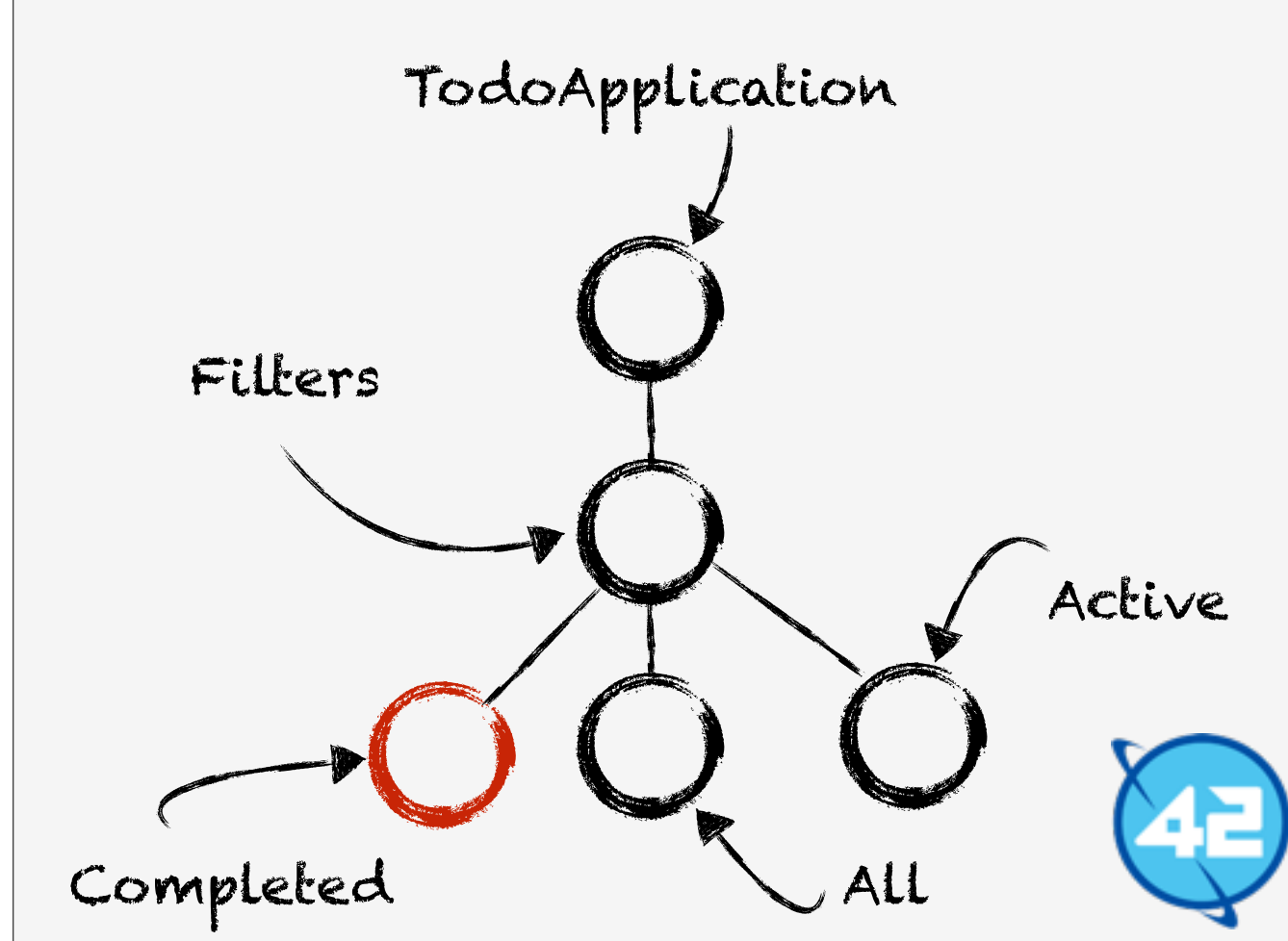
Later on we will see how using Redux can help us get to a completely different conclusion.

How *does* the active filter *change*?



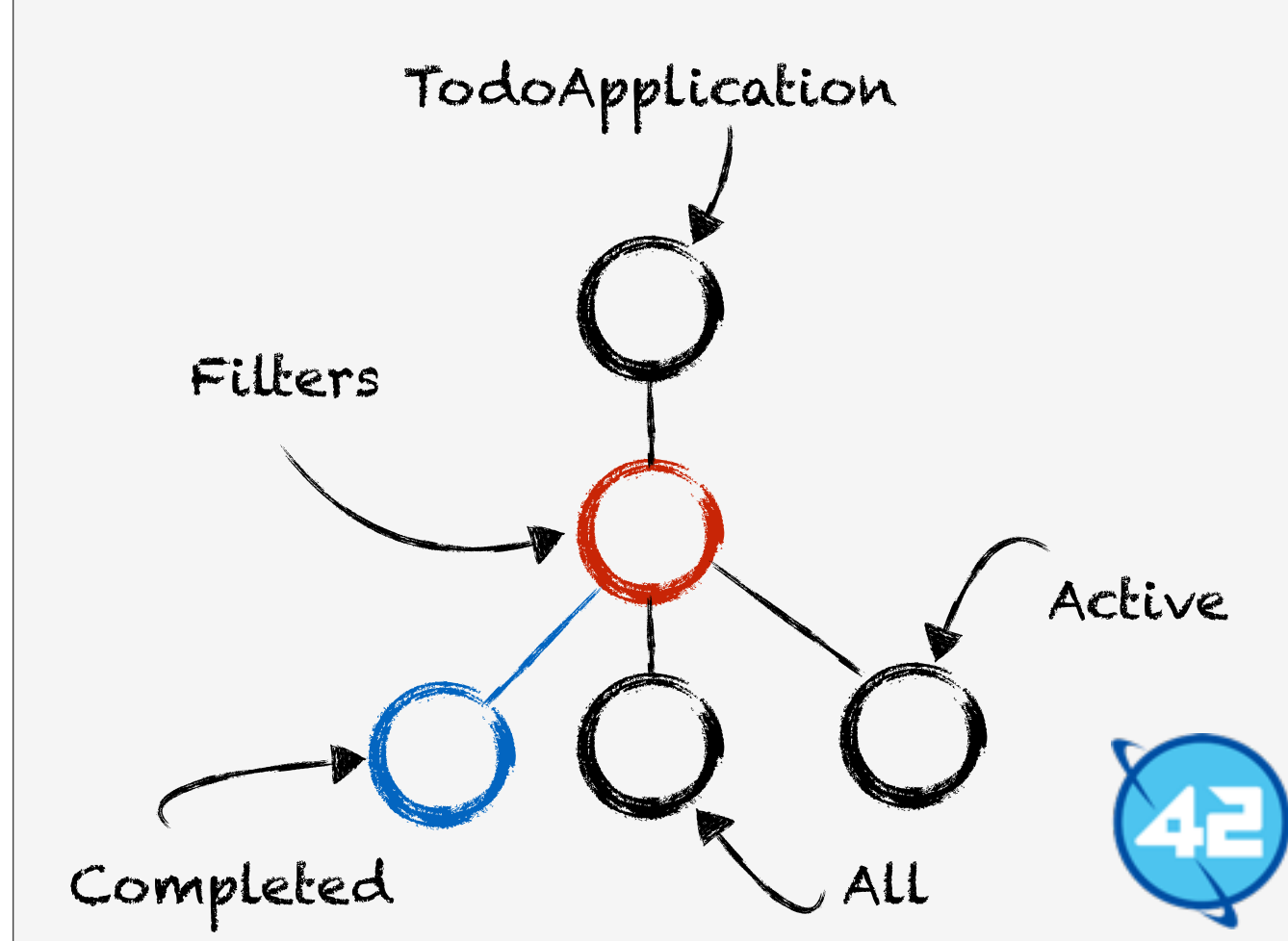
To make "How do components communicate?" more concrete lets ask ourselves this question:

"Where do we put the current active filter?"

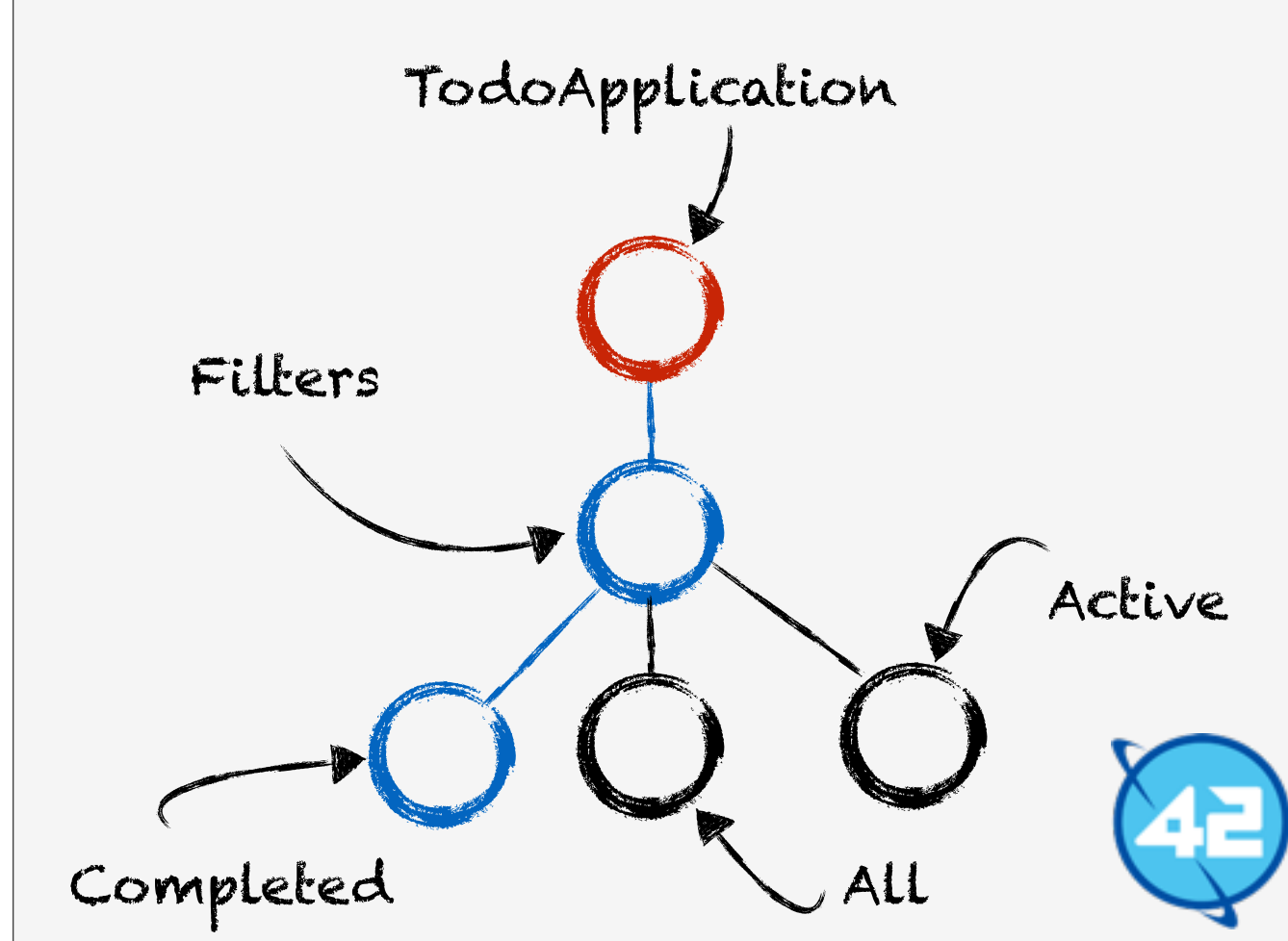


I've taken the tree and zoomed in on the Filters sub tree.

So lets look at what happens when the "Completed" filter is clicked.



Filters doesn't keep track of the active filters's state, TodoApplication does, so it needs to notify TodoApplication.



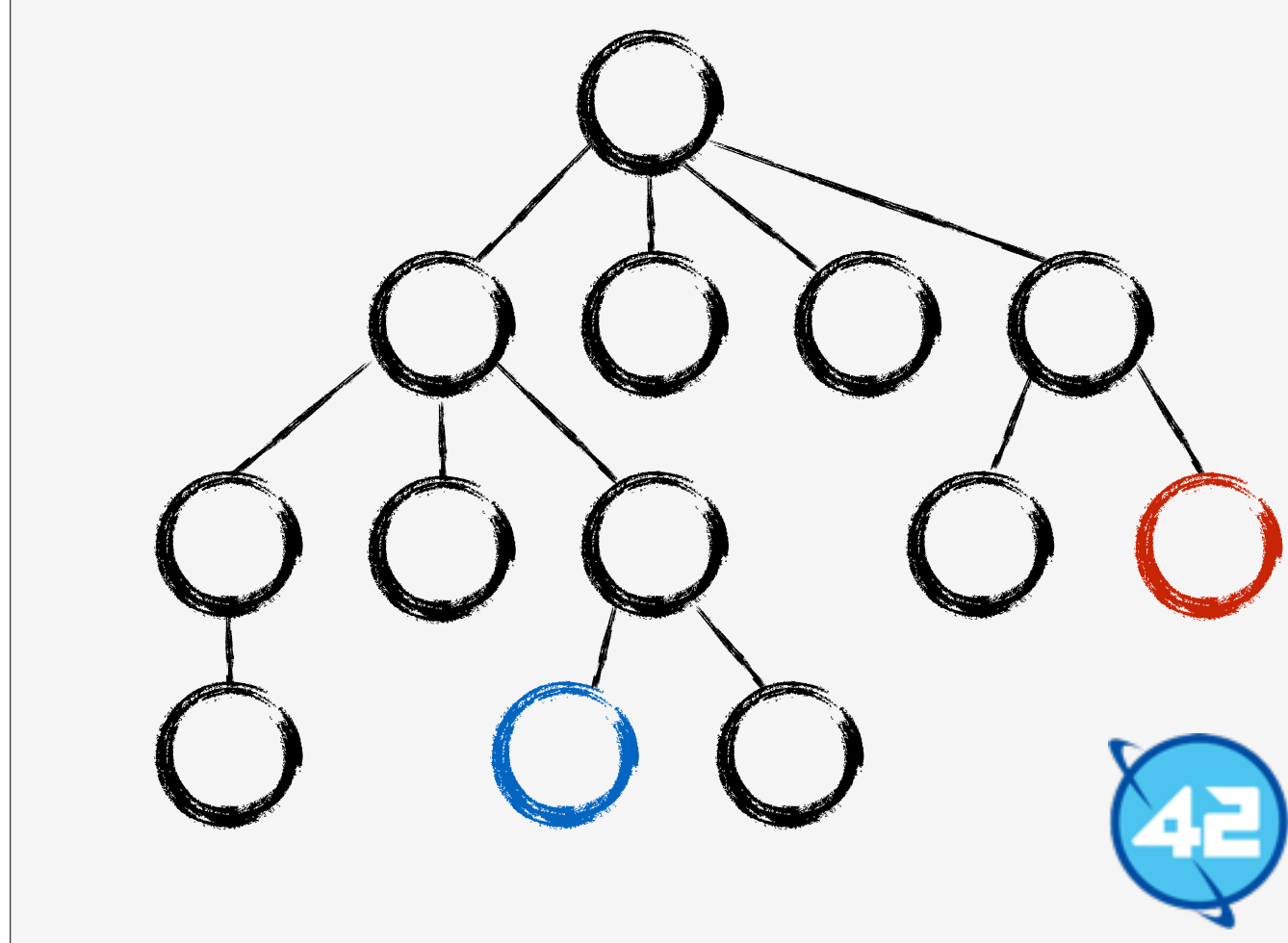
Finally it arrives and it can set the current active filter.

***Communication between
components is a *hassle*.***



The more distance there is between a event and the state that it needs to mutate the more hops up the tree must be made in order to get the event to where it belongs.

This causes the need for a lot of boiler plate code.



So imagine that we want to communicate from the blue component to the red component. We would first have to go all the way to the root component at the top and then move all the way to the right. This is a lot of boilerplate code.

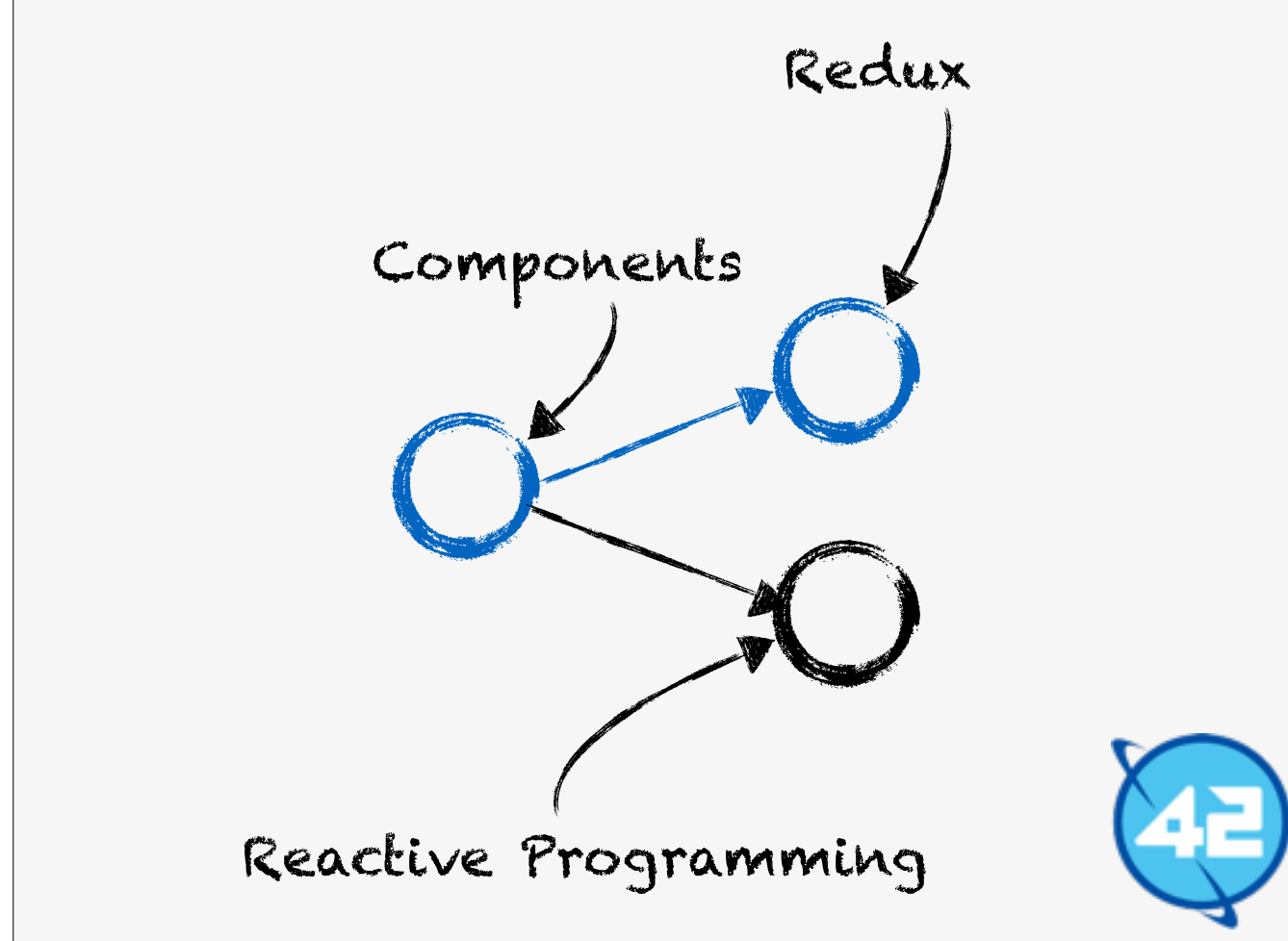
Two questions

1. Where does *the state* live?
2. How *do* components *communicate*?



These two questions highlight the inherent weaknesses of the "Component Architecture".

It is not always clear where the state of an application should be, and communication between component that are one hop 'removed' from each other is a hassle.



Which brings us to the final part of this talk. Covering Redux and Reactive Programming.

Redux is a way to mitigate the weaknesses we have uncovered.

Reactive Programming is a way to super charge asynchronous events.

Redux

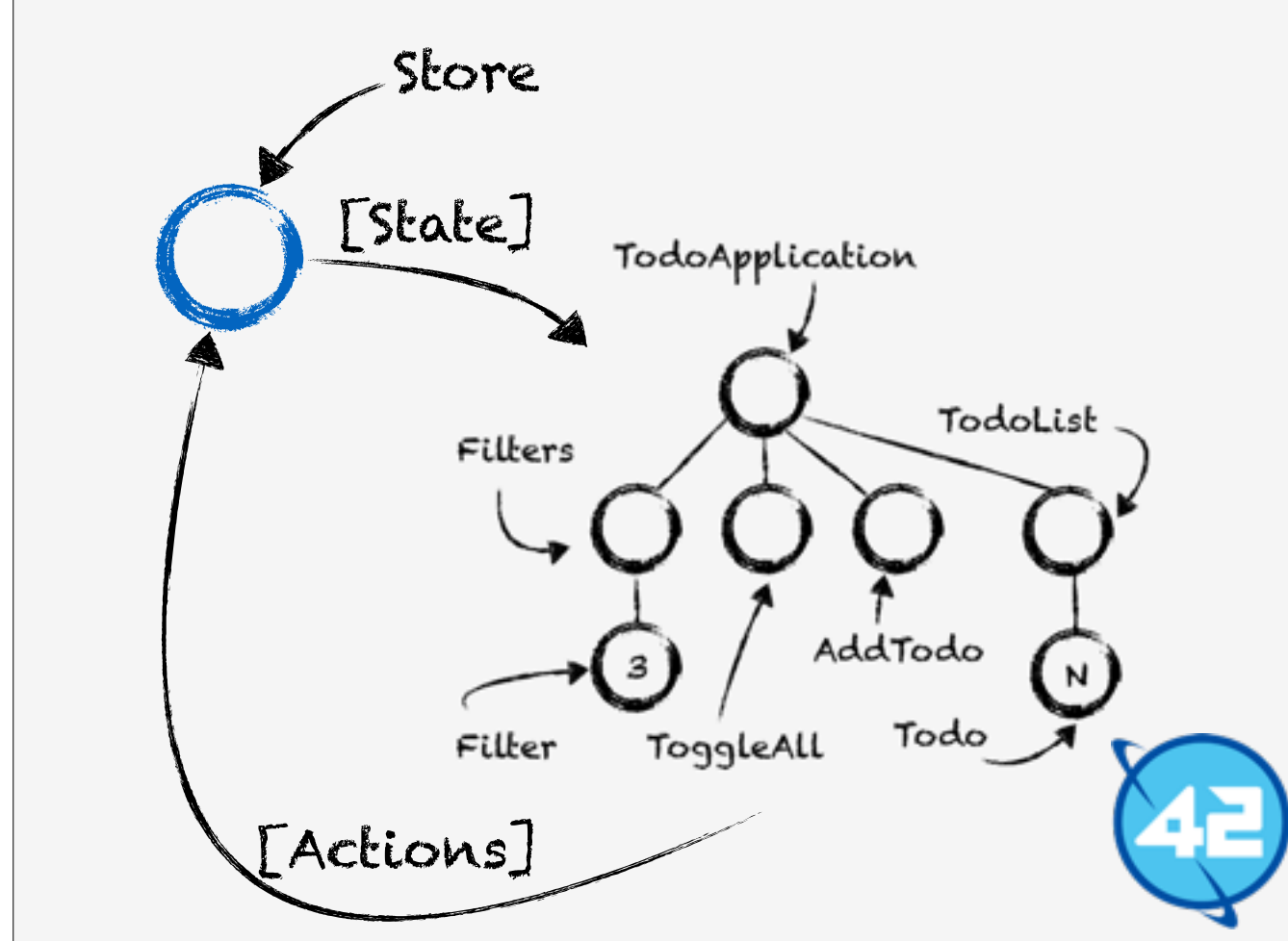


So first up is Redux, which was created by Dan Abramov and is the most popular implementation of Flux out there.

Flux* / *Redux provides a
unidirectional data flow



Redux is all about providing a unidirectional data flow. This means that all information in our big tree of Component flows from the top to bottom.

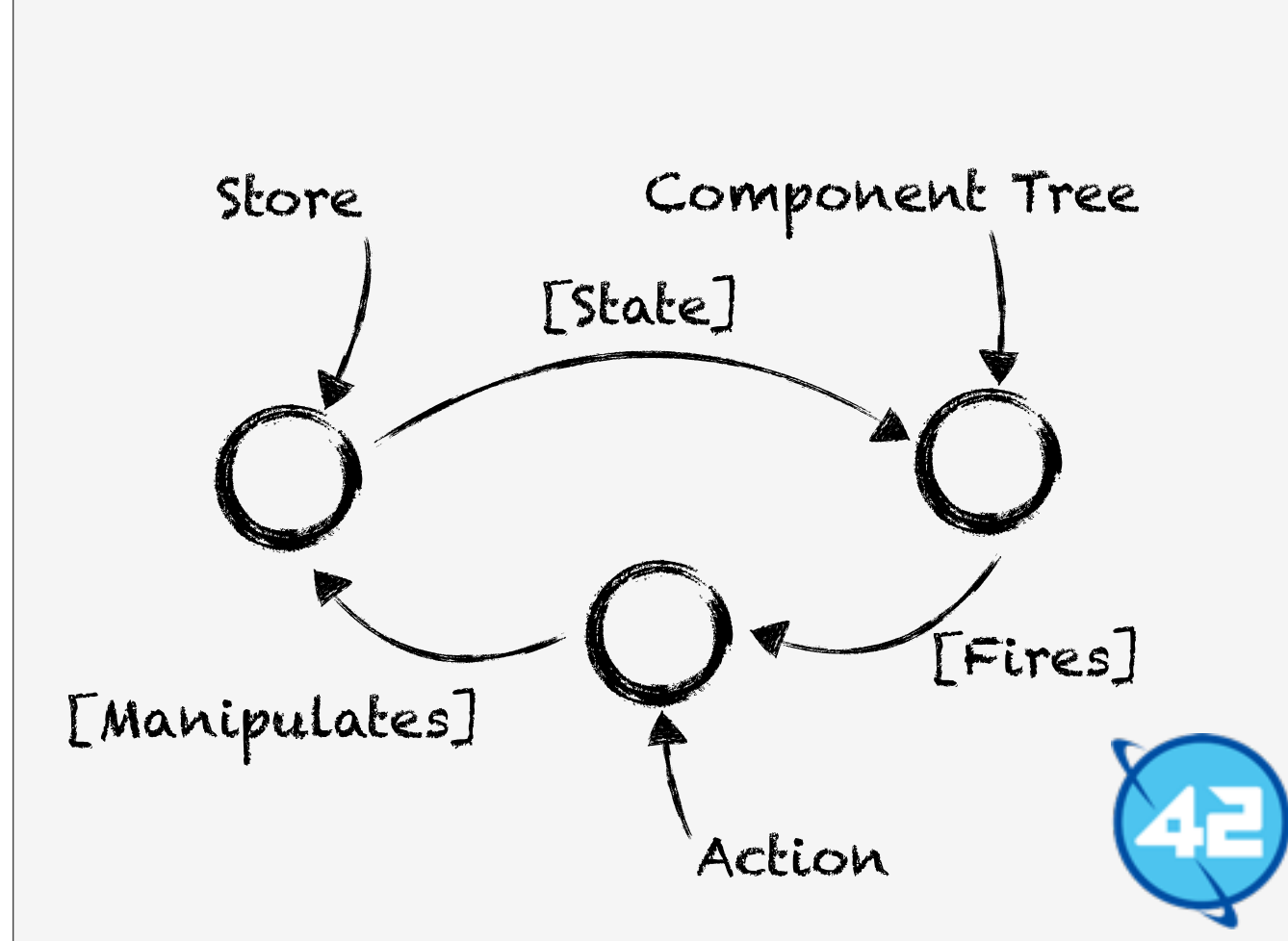


What Redux actually does it add this new concept called the "Store". In the store the entire state of the application is stored inside a single variable.

The various Components can then register themselves to receive updates for when the store has changed. When the store changes they can fetch the state and use that state to render themselves with.

So for example the **TodoList** Component would register itself and when the todos' change update itself to show the current Todo's.

So what manipulates the store? The answer is "actions" which any Component can fire towards the store.



You can clearly see by this diagram that the flow is unidirectional. State flows from the top of the tree to the bottom. Events always go from the bottom back to the top.

Two questions

1. Where does *the **state*** live?
2. How *do* components ***communicate***?



So if we take a look at our Two Questions, Redux provides us with two Answers:

Two answers

1. All ***state*** lives in the ***store***
2. All ***events*** route to the ***store***



The "state" will live in the store, the store will act as the single source of truth in the application.

The second answer to the question how do components communicate is via the "store". Component A sends an event to the store, the store state is then changed, and Component B receives the new state from the store. The store will act as a sort of middle man between Components.

Case Study



Let's look at some Redux code for a simple counter app.

The counter app is really simple click the plus button and the number increases click the minus button and the counter decreases.

```
// Defines all possible actions as const's.
const INCREMENT_COUNTER = "INCREMENT_COUNTER";
const DECREMENT_COUNTER = "DECREMENT_COUNTER";

/*
  Action creator functions,
  functions that create Redux actions
*/
function incrementCounter() {
  return {
    type: INCREMENT_COUNTER
  };
}

function decrementCounter() {
  return {
    type: DECREMENT_COUNTER
  };
}
```

Lets look at the actions first before we dive into the store. Actions in Redux are simply objects, with a 'type' property to identify them with. These objects act as payloads which are sent to the store to be processed.

Besides 'type' you can add any property you want. For example we could create an action called "SET_COUNTER" and provide a value to which we want to set the counter to in a "count" property.

```
// The initial count starts at zero.
const initialCount = 0;

// Create the store with the reducer and the initial state.
let store = Redux.createStore(counterApp, initialCount);

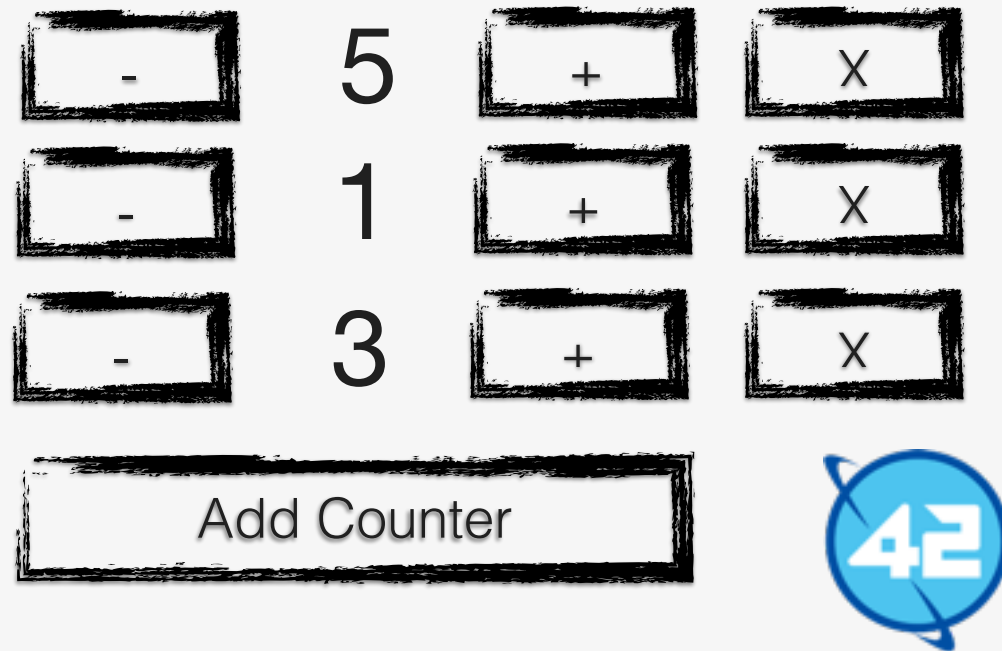
// The reducer for the counter application.
function counterApp(state, action) {
  switch(action.type) {
    case INCREMENT_COUNTER:
      return state + 1;
    case DECREMENT_COUNTER:
      return state - 1;
  }

  return state;
};
```

So here we see the creation of the store with an initial value of zero. The "counterApp" is what Redux called a Reducer, a Reducer takes the state and an action payload and performs the action on the state.

The Reducer is a pure function which means that it will never mutate the current state but always returns a new state instead.

Case Study



The counter app was really simple so how about a list of dynamic counters. The idea is simple clicking on the "Add Counter" button creates a new counter. The counters works the same as before only they have a "X" button to delete the counter.

```

// Initially there will be one counter
const initialState = {
  nextCounterId: 1,
  counters: { 1: { count: 42, counter: 1 } }
};

function removeCounter(counter) {
  return {
    type: REMOVE_COUNTER,
    counter: counter
  }
}

// This case is inside the Reducer
case REMOVE_COUNTER:
  var nextState = Object.assign({}, state); // Copy state
  delete nextState.counters[action.counter];
  return nextState;

```

The code is a little bit too big to fit into one slide so let's go through each action one at a time.

The first action is REMOVE_COUNTER to remove a counter from the list.

The state is stored into an object where counters is an object which will contain all counters, and nextCounterId is used to determine which id to use for the next counter.

The REMOVE_COUNTER action has a property counter which represents the id of the counter that needs to be removed.

Inside the reducer we find REMOVE_COUNTER it copies the current state via Object.assign because we cannot MUTATE the current state. The reason we cannot MUTATE the current state is because Redux expects the reducer to be pure.

Then REMOVE_COUNTER simply removes the counter by using 'delete' and it returns the next state.

```

// Initially there will be one counter
const initialState = {
  nextCounterId: 1,
  counters: { 1: { count: 42, counter: 1 } }
};

function createCounter() {
  return {
    type: CREATE_COUNTER
  };
}

// This case is inside the Reducer
case CREATE_COUNTER:
  var nextState = Object.assign({}, state); // Copy state
  nextState.nextCounterId = state.nextCounterId + 1;
  nextState.counters[state.nextCounterId] = {
    count: 0,
    counter: state.nextCounterId
  };
  return nextState;

```

The second action is CREATE_COUNTER to add a new counter to the list.

The createCounter produces an action with just the type as the payload.

CREATE_COUNTER increases the nextCounterId by one. Then it adds a new counter to the counters object with the 'nextCounterId' as the key, the counter itself starts with count: 0 and also knows it's 'id'.


```
// Initially there will be one counter
const initialState = {
  nextCounterId: 1,
  counters: { 1: { count: 42, counter: 1 } }
};

function incrementCounter(counter) {
  return {
    type: INCREMENT_COUNTER,
    counter: counter
  };
}

// This case is inside the Reducer
case INCREMENT_COUNTER:
  var nextState = Object.assign({}, state); // Copy state
  nextState.counters[action.counter].count += 1;
  return nextState;
```

The third action is INCREMENT_COUNTER.

The incrementCounter takes as an argument the counter to be increased.

Inside the reducer it copies the state again like the other actions also did. Then it gets the counter object by the provided id and increases the count by one.

```
// Initially there will be one counter
const initialState = {
  nextCounterId: 1,
  counters: { 1: { count: 42, counter: 1 } }
};

function decrementCounter(counter) {
  return {
    type: DECREMENT_COUNTER,
    counter: counter
  };
}

// This case is inside the Reducer
case DECREMENT_COUNTER:
  var nextState = Object.assign({}, state); // Copy state
  nextState.counters[action.counter].count -= 1;
  return nextState;
```

The final action DECREMENT_COUNTER is exactly the same as INCREMENT_COUNTER except it decrements the counter by one instead of increasing it.

Benefits of Redux

1. *Easier Universal* JavaScript



Besides the obvious benefits that Redux has for dealing with state and allowing easy communication between Components. There are two more benefits.

The first one is that Redux due to having a single store makes it easier to do Universal JavaScript. This means rendering the Application on a Node.js server and sending it to the browser, where it is made "alive" again. Normally this is a bit harder because the state of the application lives in a lot of different Components. With Redux you simply set the initial state correctly once in the store, and all components should then render properly.

UI = Tree(state)



In Redux your UI is just a function of the state applied to your component tree.

So you get your 'state' once from the database for a particular user and ram it through the system. You will only need to set the state once for the entire application which is rather easy.

$$UI = \sum \text{Component}(\text{state})$$



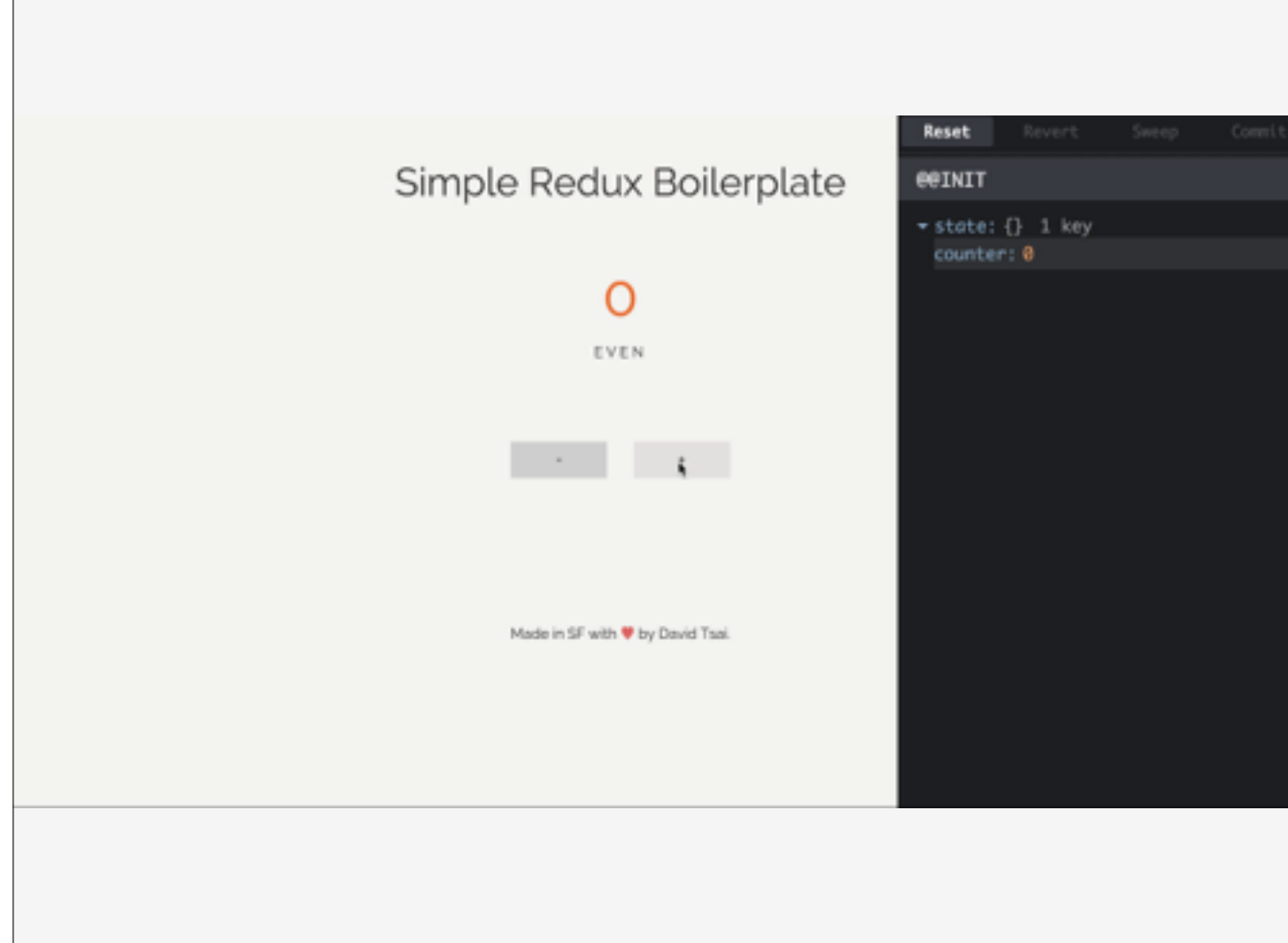
Contrast this to when your state is put in multiple different components. You will have to put the state correctly in all of them. This might not be a very difficult thing to do but it is a tedious chore. The more components you have which store some 'user' based state the more 'set' state code you will have to write.

Benefits of Redux

1. *Easier **Universal** JavaScript*
2. *Good **Developer Experience***



The second one is that Redux has a really good DX story.



The second one is that Redux has a really good DX, it has a lot of tools that make debugging Redux applications easier such as a Chrome Plugin that shows the state of the store. The plugin even allows you to travel through time by moving back and forward through actions.

Cons of Redux

1. There is a *learning* curve.



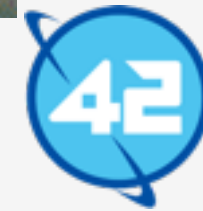
"We are not in Kansas anymore". What I mean by this is that Redux has a philosophical overhead. Immutable, Pure Functions, Reducers, Dispatchers, Thunks, Sagas. There is a lot to learn. Fortunately the concepts are not that difficult to learn. The pay-off for learning these how ever is immense. They give you a completely new look at how state can be managed.

Cons of Redux

1. There is a *Learning* curve.
2. *JavaScript* is not *Immutable*



The second con "JavaScript is not Immutable" has to do with the nature of JavaScript itself. Redux provides a great DX but those will only work if your reducers are pure functions which do not mutate the current state.



JavaScript is not immutable by default, so you have to be really careful not to mutate the state, a mistake is very easy to make, one tiny mutation and the whole things falls down like a house of cards.

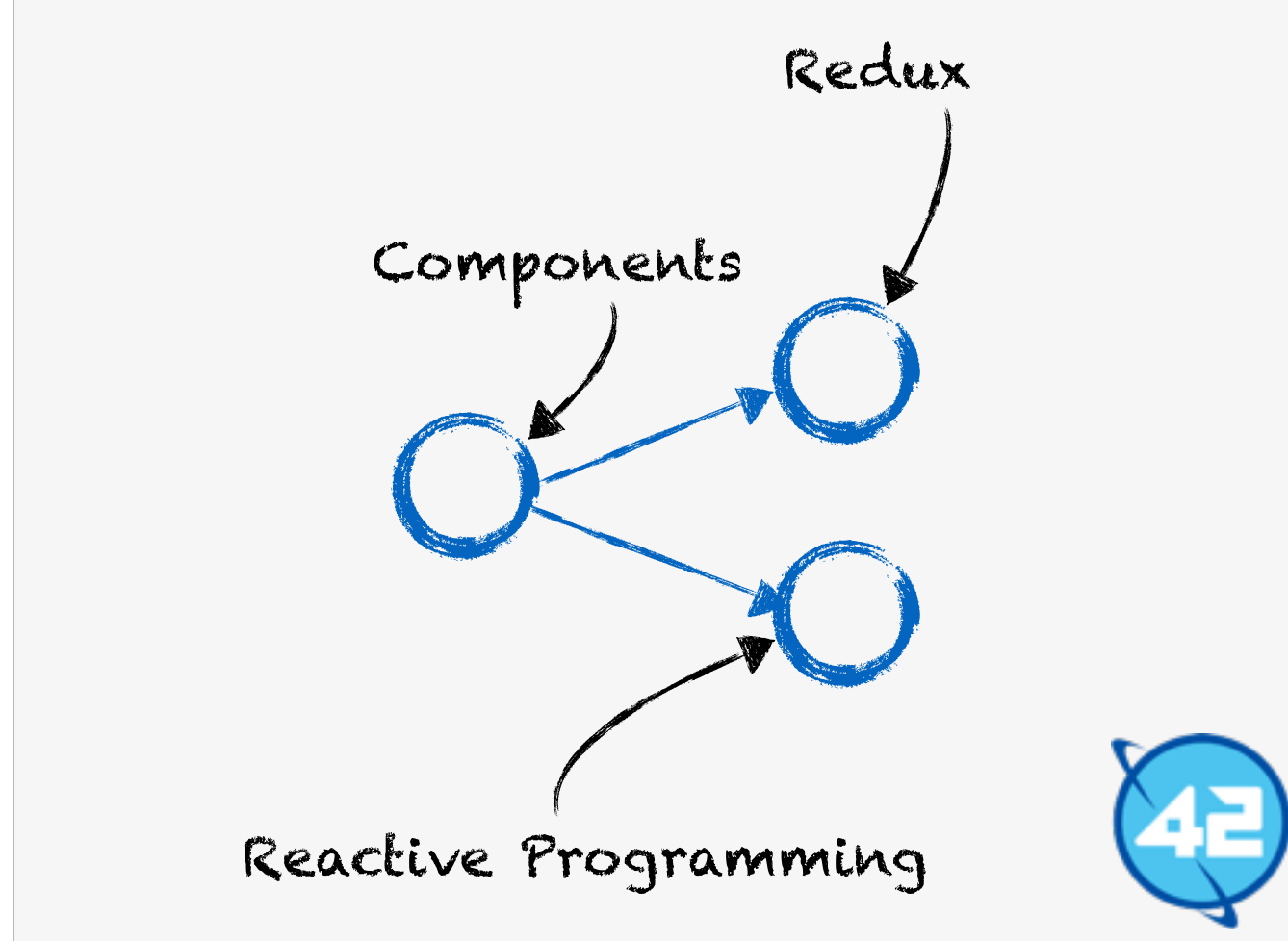
IMMUTABLE

```
var map = Immutable.Map({a: 1, b: 2});  
var list = Immutable.List.of(1, 2);
```

Luckily there are libraries like Immutable.js that provide immutable arrays and object to make this impossible to do but they do require one extra level of abstraction.

So in code this looks like this: The first line shows how to create an immutable 'map' which replaces object. The second line shows how to make an immutable list which replaces the array.

All operations performed on the List and Map always return new values and never manipulate the existing values.



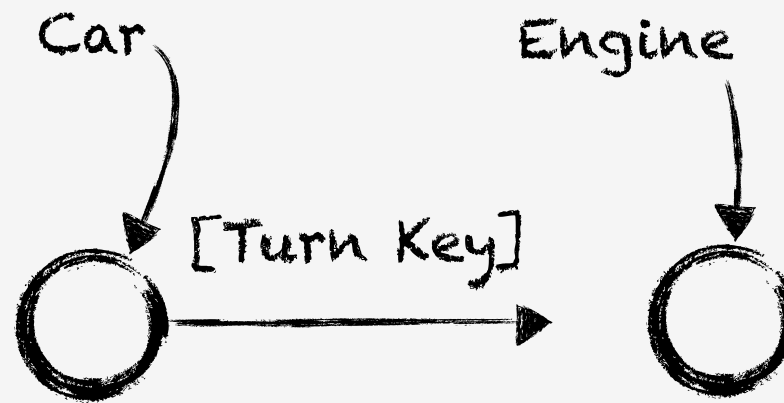
Now it is time for the final part of this talk. Reactive Programming is a way to supercharge asynchronous events.

Reactive Programming



Let's start by defining Reactive Programming first.

Passive Programming

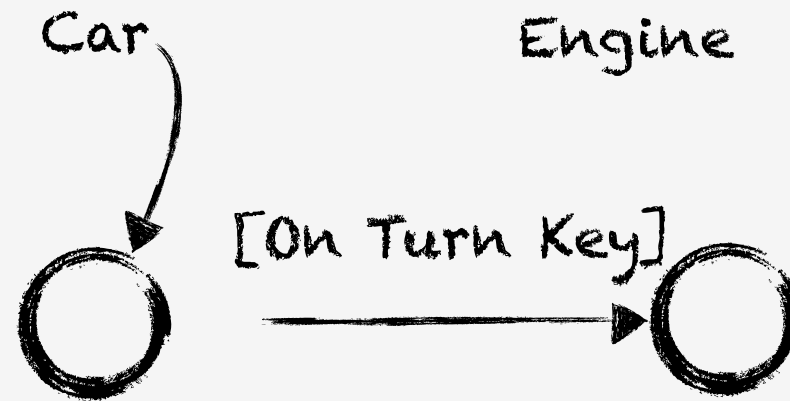


```
Car.prototype.turnKey = function() {  
  this.engine.fireUp();  
}
```

The best way to do it is to look at how we program "passively" what most of us are doing right now.

In Passive Programming when we have two components in this case the Car and the Engine. The Car calls methods in the Engine to tell the Engine when it should start.

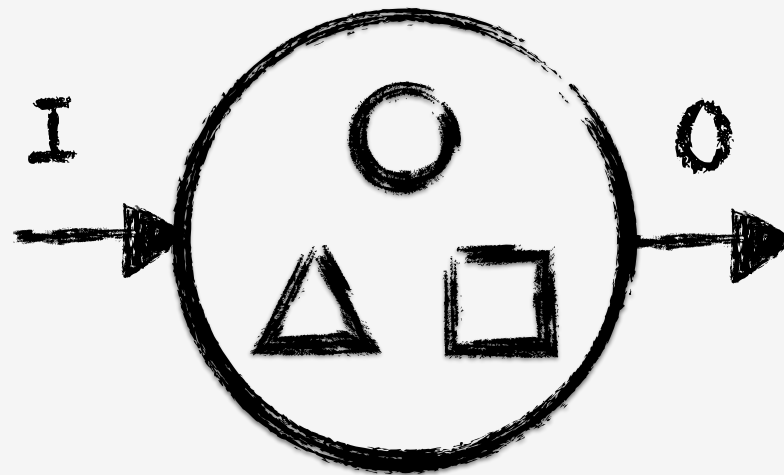
Reactive Programming



```
Engine.listenToKey = function(car) {  
  car.onKeyTurned(() => {  
    this.fireUp();  
  });  
}
```

In Reactive Programming this is turned upside down. The Engine knows when it needs to start up: when the "key is turned". The Engine is in control of its own destiny.

Reactive Components



This idea maps back very nicely with components being easy to reason about. Because all of the triggers that influence the component are located inside of the component.

RxJS



There's this library called RxJS which enables us to do Reactive Programming in JavaScript. RxJS is part of the ReactiveX family of frameworks, ReactiveX can be seen as a standard API which is ported to many languages such as Java, C#, Clojure, Swift etc etc

Two main *Concepts*

1. **Observable**: *something you can **listen** to.*
2. **Operations**: *manipulating observables.*



With Reactive Programming you listen to something called an Observable.

An Observable allows you to listen to events and react when such an event occurs.

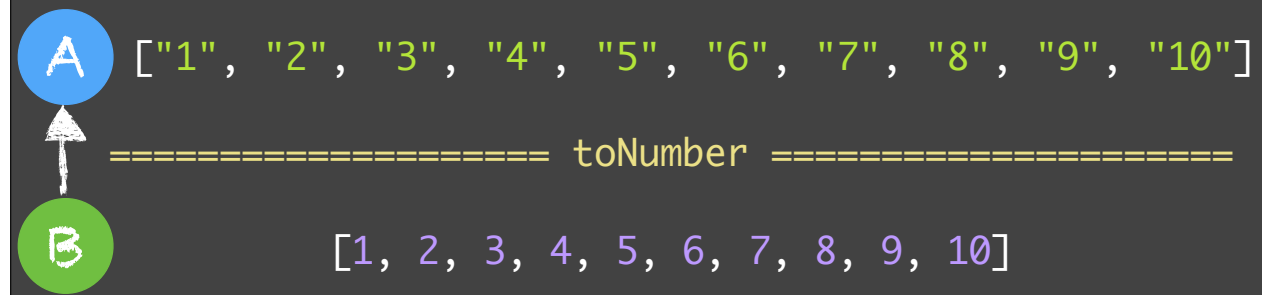
But you can also perform operations on such an event. Such as ignoring certain events, or transforming values that come out of an event.



```
A ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
```

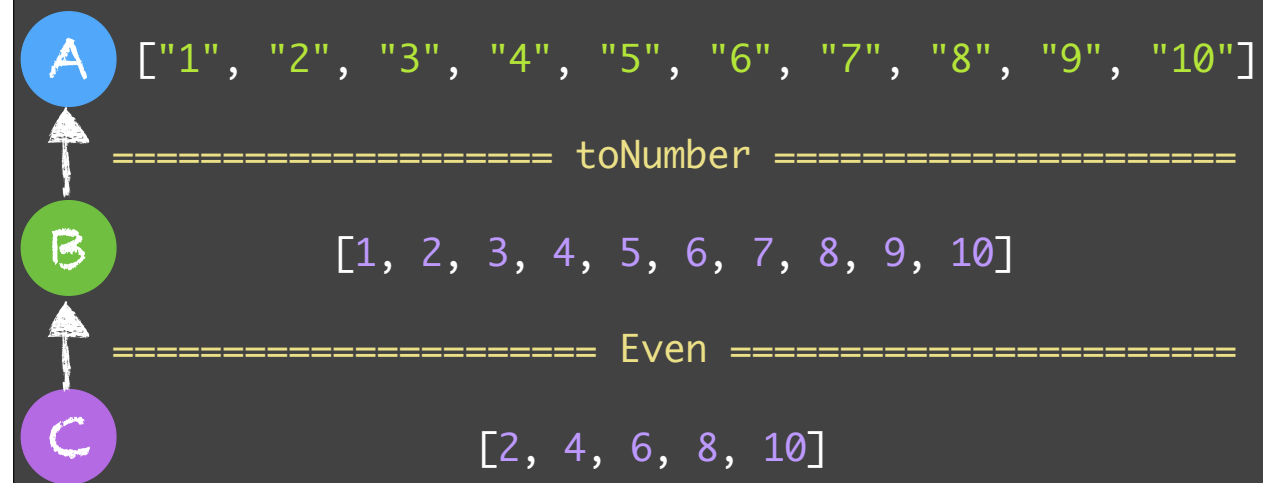
So imagine an Observable, sometimes also called a Stream, as an array. But an array which can have operations performed on it.

Here we have an Observable called A which is an array of string numbers.

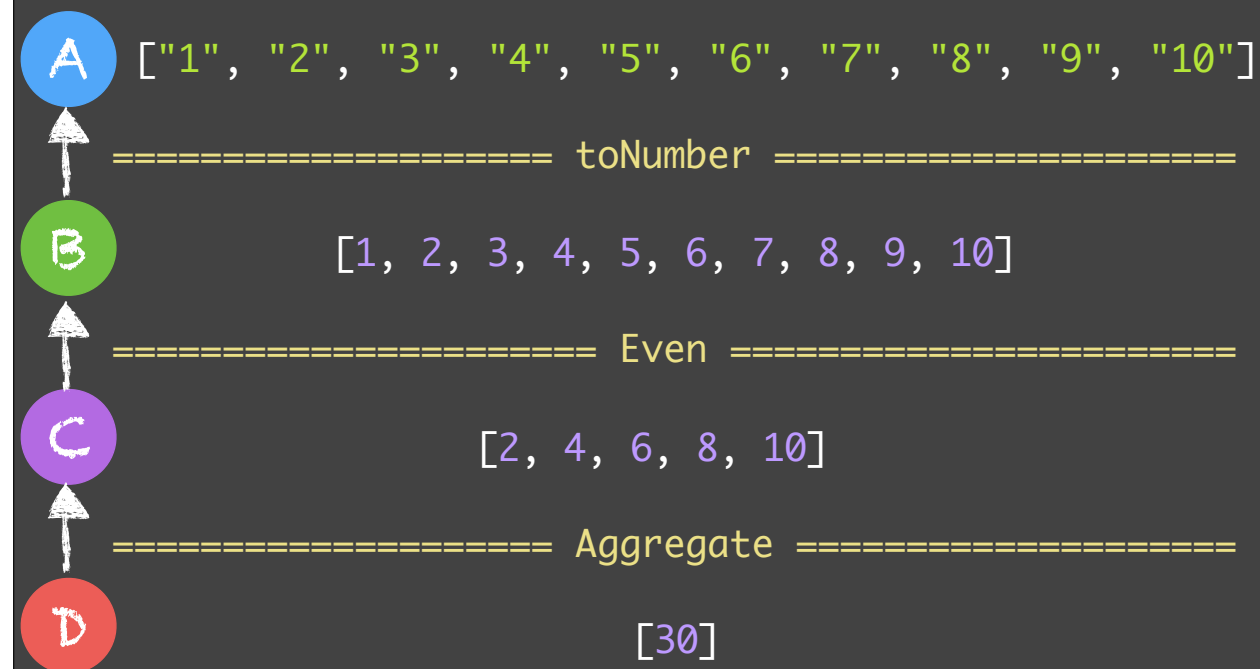


We could perform an operation 'toNumber' to turn the strings into numbers.

Now this means that each time the Observable A changes. Observable B also updates. This is because Observable B listens to Observable A.

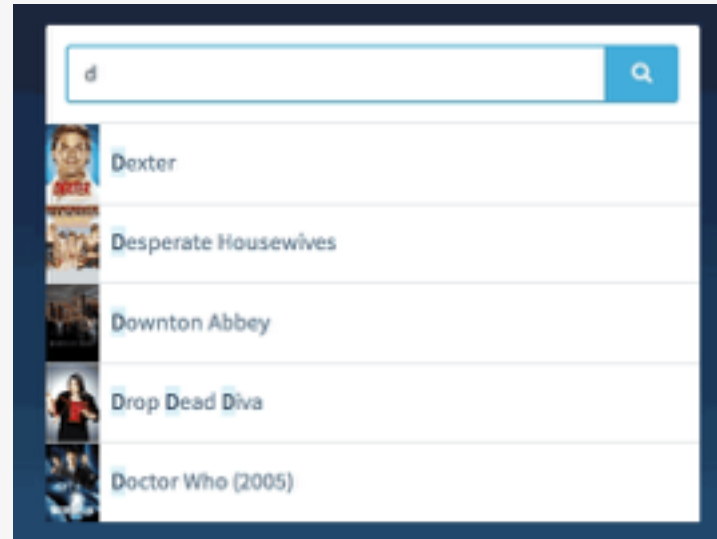


We can then create Observable C and perform an operation that only keeps the even numbers. Observable C listens to Observable B, which in turn listens to Observable A. Which means that if A changes C is updated automatically as well.



Finally we can create Observable D which sums up the array to a single value. Observable D listens to changes on Observable A through C and D. Which means that if a number is added to A, D is automatically recalculated.

Case Study



Lets look at Rx.JS in action by looking at an autocomplete Component.

Everyone knows what an autocomplete is you type in some letters in a search box it sends a request to the back-end, the back-end responds with a list of completions. The completions are shown to the user and when he clicks on them it automatically fills it in.

An autocomplete Component is more complex than you might realise on first glance how ever. Lets look at a naive example in Rx.JS.

```
var $input = $('#search-input');

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value) // Project the text from the input
  .map(search);            // Search does an 'ajax' request
  .subscribe(function(data) {
    // Update the ui here with the data
  }, function(error) {
    // Update the ui here saying what the error was.
  });
```

So what happens here is that we create an Observable from the '#search-input' 'keyup' events. When ever the 'keyup' event occurs we take the event's target value to get the value of the search-input at that moment.

Next we call 'search' with the text of the input field. 'search' is a function that does an Ajax request for us.

Next we have a subscribe for when the Ajax call succeeded or failed. When it succeeds we update the UI with the data that comes back from the server. When it fails we ignore the error complete, which is great error handling :P.

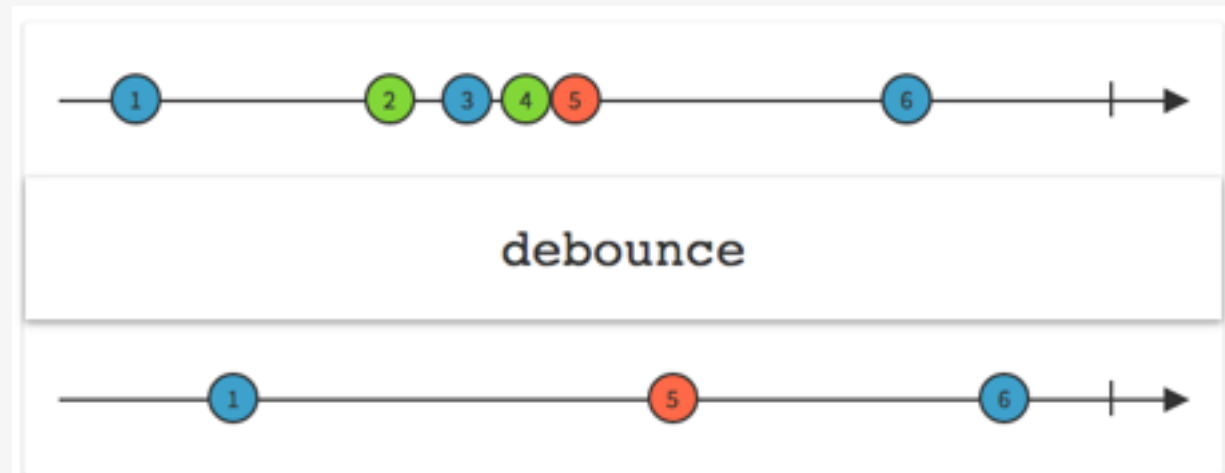
Pretty cool stuff, but there is something wrong with it.


```
[2016-06-14 10:00:01] Query: "I"  
[2016-06-14 10:00:02] Query: "I "  
[2016-06-14 10:00:03] Query: "I l"  
[2016-06-14 10:00:04] Query: "I lo"  
[2016-06-14 10:00:05] Query: "I lov"  
[2016-06-14 10:00:06] Query: "I love"  
[2016-06-14 10:00:07] Query: "I love "  
[2016-06-14 10:00:08] Query: "I love G"  
[2016-06-14 10:00:09] Query: "I love GO"  
[2016-06-14 10:00:10] Query: "I love GOT"  
[2016-06-14 10:00:11] Query: "I love GOTO"
```

So during testing a back-end programmer comes up to us and says that he's seeing this weird log. We are spamming the server too much.

What we want to do is to only send a request to the back-end when the user has stopped typing for a while.

Debounce



Rx.JS has this operator called `debounce` which waits for x number of milliseconds after the last event before it fires the event. This means that when the same event fires multiple times in a row only after x milliseconds will the last event fire.

Here's a so called 'marble' diagram showing what `debounce` does. This comes from the ReactiveX website and they have them for every operator.

```
var $input = $('#search-input');

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value) // Project the text from the input
  .debounce(500) // Wait for 500 milliseconds after the last event.
  .map(search) // Search does an 'ajax' request
  .subscribe(function(data) {
    // Update the ui here with the data
  }, function(error) {
    // Update the ui here saying what the error was.
  })
```

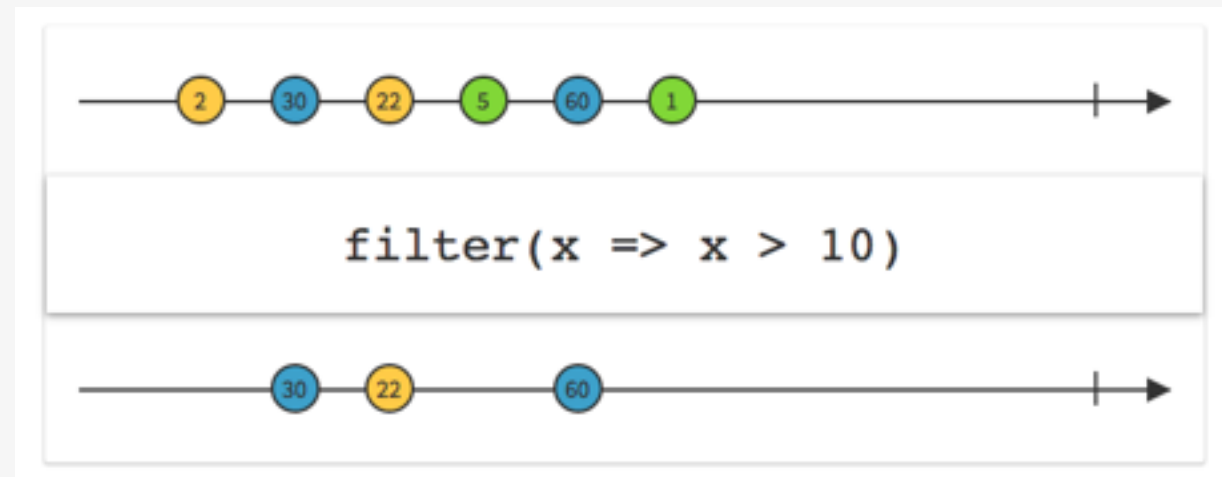
So we apply a debounce of 500 milliseconds before we search and the problem is solved.

```
Query: "G"      -> 100000 bad results
Query: "Go"     -> 10000 bad results
Query: "Got"    -> 1000 bad results
Query: "Goto"   -> 10 good results
```

Then we get another report from our back-end guy. He tells us that the API he is using doesn't really provide good results for less than three characters. So we only need to send queries that have at least three characters in it.

Luckily there is an operator for that called filter.

Filter



Filter takes a predicate function and applies that function to the stream of values. When the predicate function returns true the value is allowed in the stream.

```
var $input = $('#search-input');

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value) // Project the text from the input
  .filter(query => query.length > 3) // Three or more characters
  .debounce(500) // Wait for 500 milliseconds after the last event.
  .map(search) // Search does an 'ajax' request
  .subscribe(function(data) {
    // Update the ui here with the data
  }, function(error) {
    // Update the ui here saying what the error was.
  })
```

So we use filter and make sure that the query length is always bigger than three.

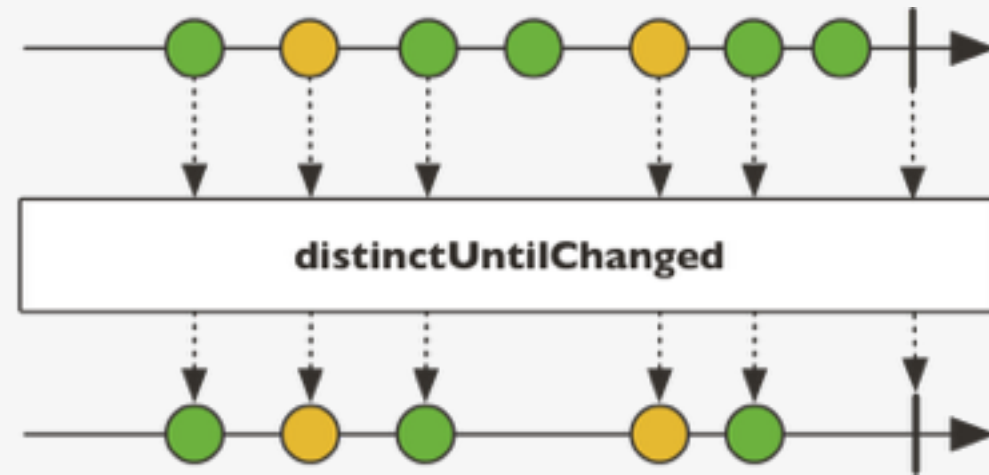
So we fixed yet another problem, but we are not done yet, our back-end man has found yet another strange log.

```
[2016-06-14 10:00:00] Query: "GOTO"  
[2016-06-14 10:05:00] Query: "GOTO"
```

Here we can see that we are firing the same 'query' twice. This happens when the user types in a query, waits for a while and starts typing again only to change his mind and backspace back to the original query.

Luckily there is an operator for that.

distinctUntilChanged



Filter takes a predicate function and applies that function to the stream of values. When the predicate function returns true the value is allowed in the stream.

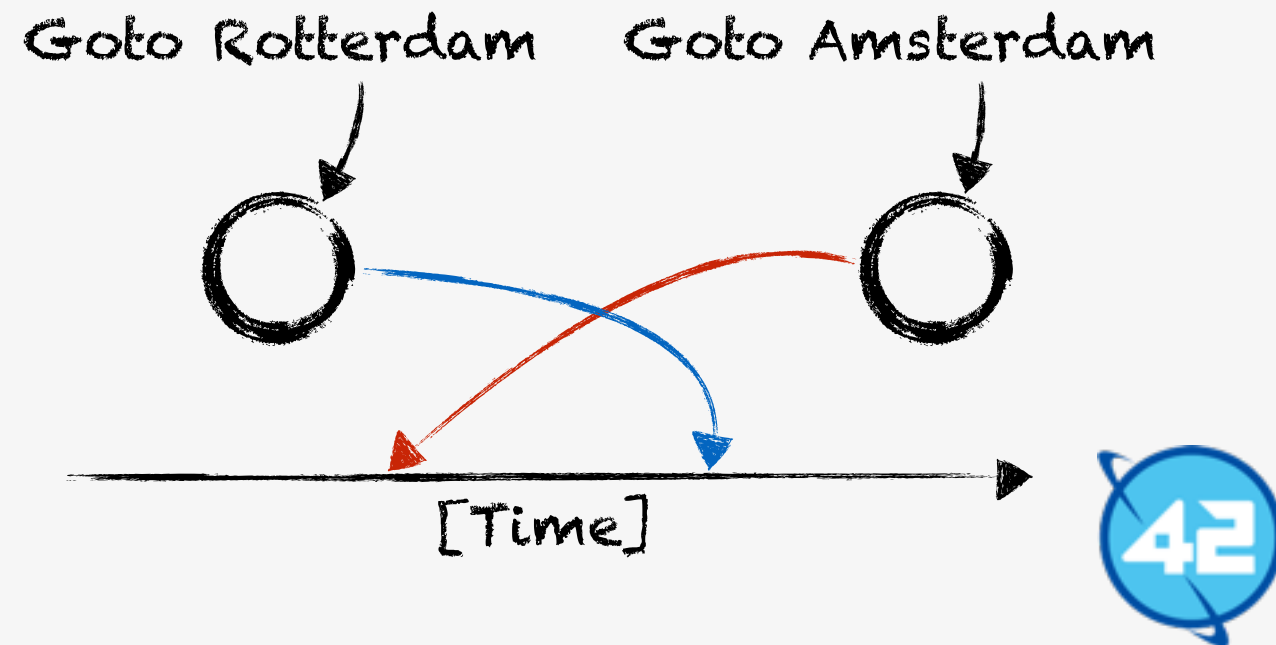

```
var $input = $('#search-input');

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value) // Project the text from the input
  .filter(query => query.length > 3) // Three or more characters
  .debounce(500) // Wait for 500 milliseconds after the last event.
  .distinctUntilChanged(); // Only if the query has changed
  .map(search) // Search does an 'ajax' request
  .subscribe(function(data) {
    // Update the ui here with the data
  }, function(error) {
    // Update the ui here saying what the error was.
  })
```

This operator is called 'distinctUntilChanged' it will suppress any duplicates queries.

Now we have one final difficult big problem to solve.

A big problem



What if the user types in "Goto Rotterdam" and waits, then realises that he meant to search for "Goto Amsterdam" instead. Now two Ajax calls are open, what happens when "Goto Amsterdam" finishes before "Goto Rotterdam".

The answer is that the user sees the wrong search results!

We actually only care about the search results for the last query that was made. All other results we don't care about, luckily there is an operator for that!

```
var $input = $('#search-input');

Rx.Observable.fromEvent($input, 'keyup')
  .map(e => e.target.value) // Project the text from the input
  .filter(query => query.length > 3) // Three or more characters
  .debounce(500) // Wait for 500 milliseconds after the last event.
  .distinctUntilChanged() // Only if the query has changed
  .flatMapLatest(search) // Use only the Resp from the last query.
  .subscribe(function(data) {
    // Update the ui here with the data
  }, function(error) {
    // Update the ui here saying what the error was.
  })
```

The operator is called 'flatMapLatest' it takes group of observables, in this case the Ajax requests, and turns them into a single observable by only taking the latest one.

Now we can put the thing in production and never talk about it again. So now for some observations about this process.

Benefits of RxJS

1. **Producer** and ***consumer*** are ***decoupled***



The operations that come with RxJS can be composed to achieve bigger things. There are a lot of operations that come with RxJS so there is a lot of power in the library.

Benefits of RxJS

1. **Producer** and ***consumer*** are ***decoupled***
2. *Operations* are a ***powerful*** building ***block***



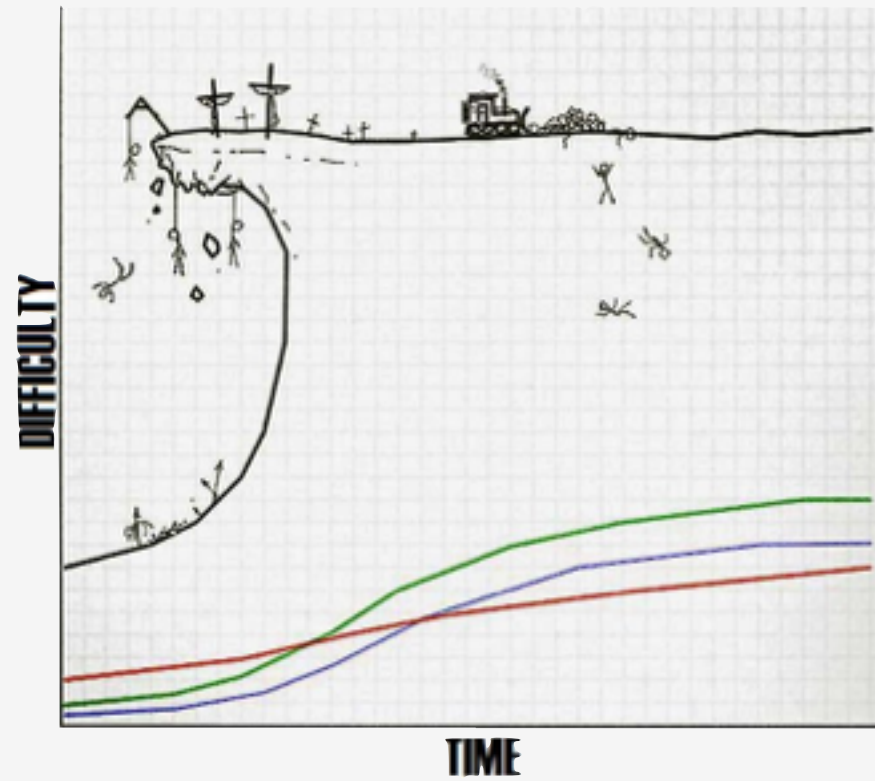
The operations that come with RxJS can be composed to achieve bigger things. There are a lot of operations that come with RxJS so there is a lot of power in the library.

Cons of RxJS

1. There is a *steep learning* curve



Like Redux before RxJS shares the same weakness as Redux. It takes a lot of getting used to to fully grok the RxJS philosophy and get into the 'stream' mindset. It does give you a better perspective once you begin to get it, some problems like the 'autocomplete' have surprisingly little code, so it is well worth it.



Like Redux before RxJS shares the same weakness as Redux. It takes a lot of getting used to to fully grok the RxJS philosophy and get into the 'stream' mindset. It does give you a better perspective once you begin to get it, some problems like the 'autocomplete' have surprisingly little code, so it is well worth it.

Cons of RxJS

1. There is a ***steep learning*** curve
2. The ***documentation*** is highly ***conceptual***.



The documentation of RxJS is very high level and conceptual. Reading some of the documentation is feels like reading a thesis. I wish they made it a little bit more practical sometimes.

"Returns an observable sequence that contains only distinct contiguous elements according to the keySelector and the comparer."



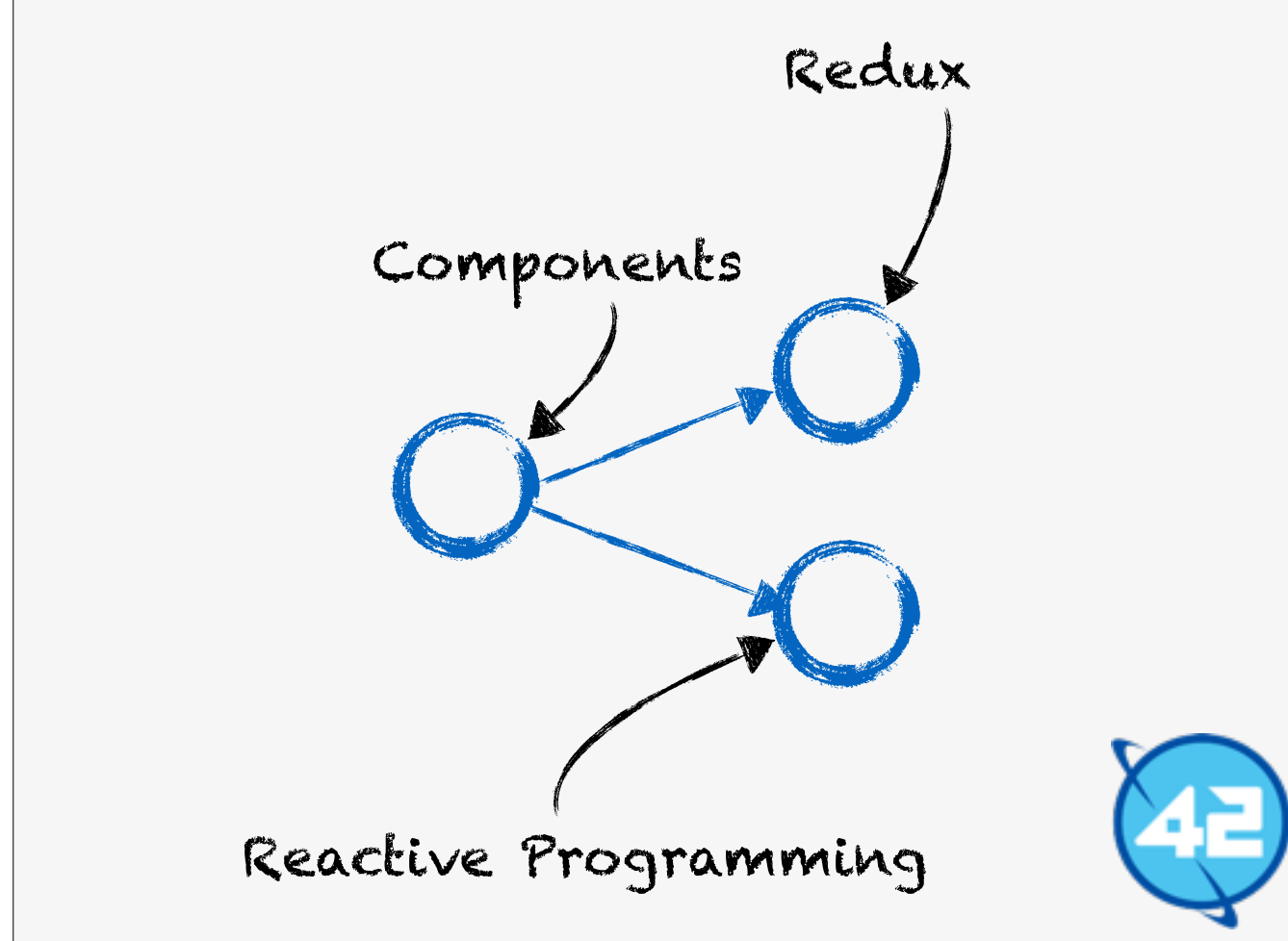
For example read this description. Can you guess which operator this is?

distinctUntilChanged

"Returns an observable sequence that contains only distinct contiguous elements according to the keySelector and the comparer."



It's distinctUntilChanged. Sometimes I really just want to see some practical examples instead of reading this prose. Hopefully in the future they will improve the documentation.



Now we have seen Redux and Reactive Programming through RxJS, and how they can help us write better Component based Applications. So we are at the end of this talk so it is time for some recommendations.

#1 If your app is really small
don't use ***Redux*** or ***RxJS***.



Both Redux and RxJS give overhead. If your application is really small I would not recommend using either of them.

#2 When the application starts
feeling too ***big*** start using
Redux.



When you application "feels" to big, and trust me you will know when you know, start using Redux. It is relatively simple to learn and your team will quickly catch up.

#3 When *events* require *coordination* use RxJS.



When events start becoming more and more complex and require more coordination use RxJS. This is the area where RxJS really shines. It does how ever require your team to swallow the RxJS pill.

Final Words

Keep a eye on [dontpanic.42.nl](#) for my blog post series on this topic.



I realise that this talk was a lot to take in. That's why I'm in the process of writing a series of blog posts about this topic on [dontpanic.42.nl](#). There I can dive into more detail to give you a clearer picture, plus give you links to some resources to help you guys in learning Redux and RxJS.



And if you want to talk to me, or play a Fallout based terminal game and win a great prize come to the 42 stand in the big hall.