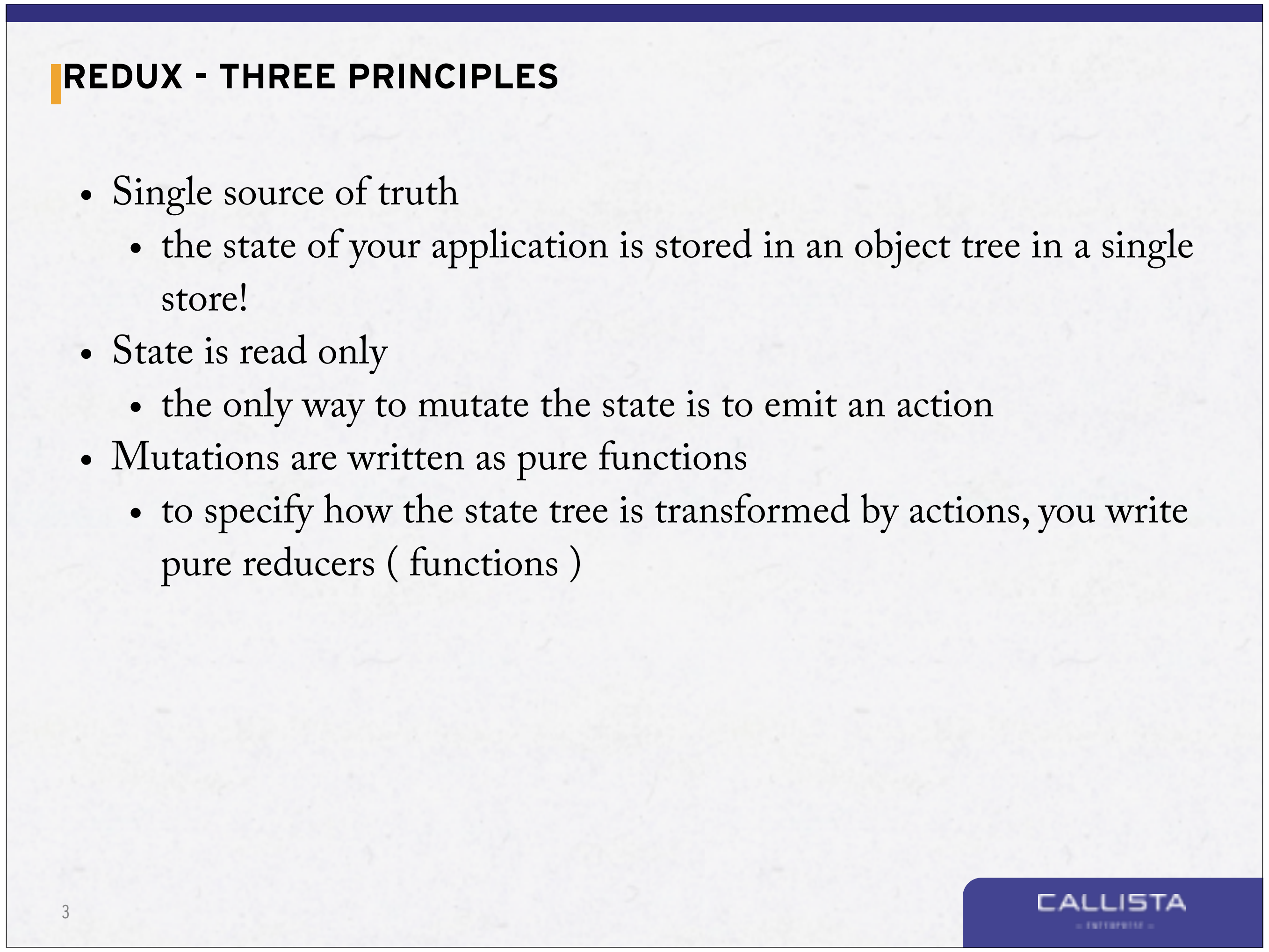
REDUX INTRO

STEPHEN WHITE

| CALLISTAENTERPRISE.SE

The Components …

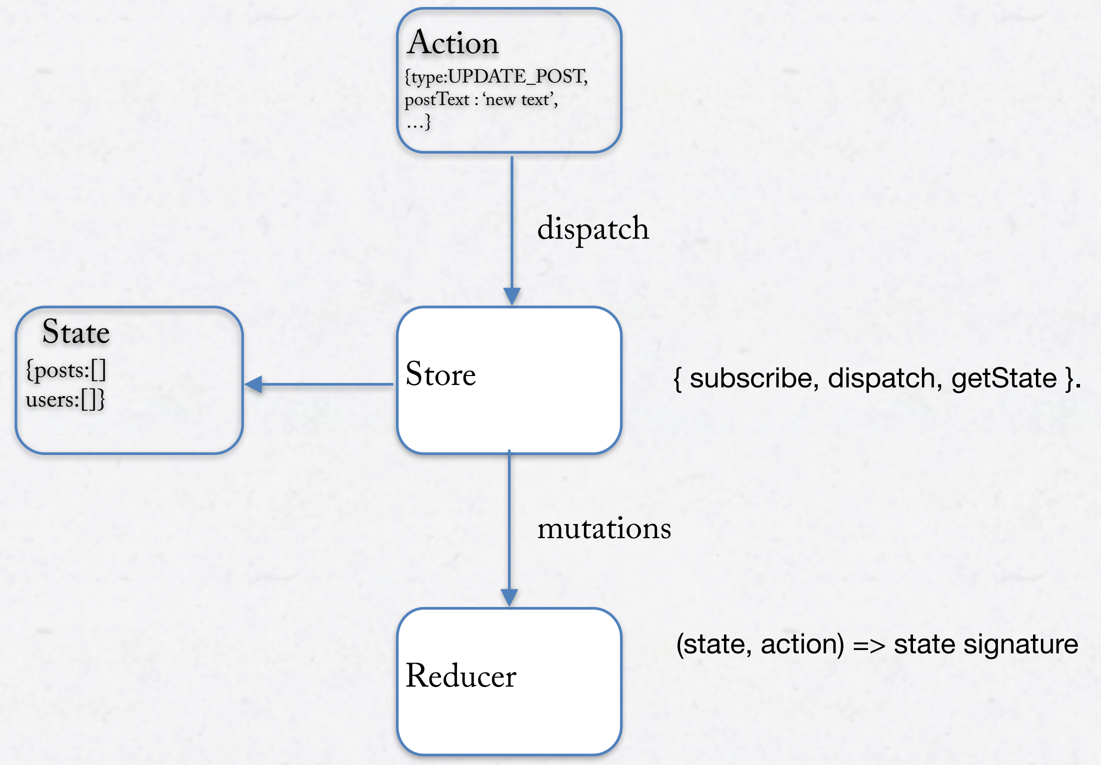


REDUX - THREE PRINCIPLES

- Single source of truth
  - the state of your application is stored in an object tree in a single store!
- State is read only
  - the only way to mutate the state is to emit an action
- Mutations are written as pure functions
  - to specify how the state tree is transformed by actions, you write pure reducers (functions)

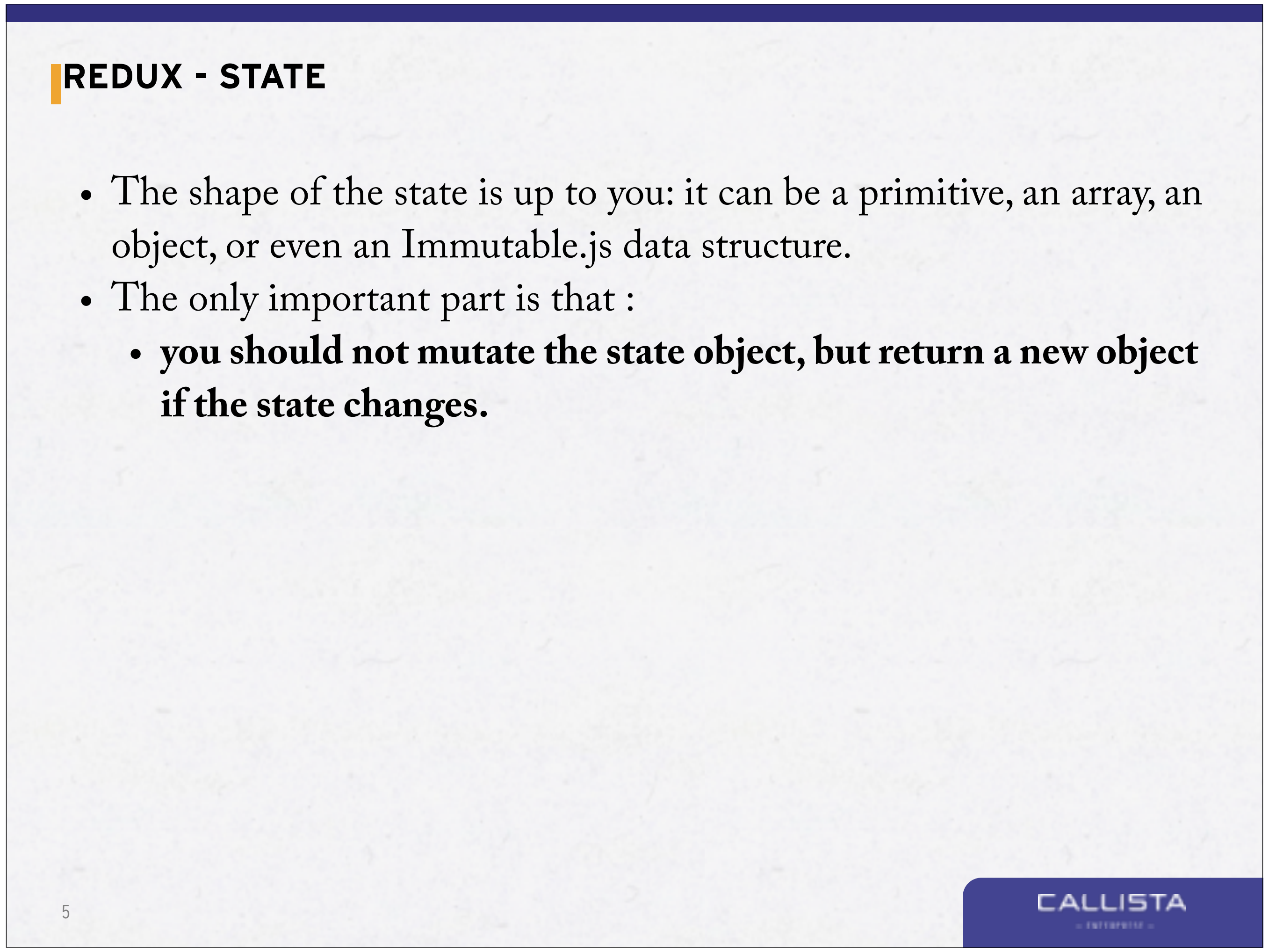
State :
The state from the server can be serialized and hydrated into the client with no extra coding effort. It is easier to debug an application when there is a single state tree. You can also persist your app's state in development for a faster development cycle. And with a single state tree, you get previously difficult functionality like Undo/Redo for free.
Actions :
views or the network callbacks never write directly to the state, but instead express the intent to mutate.
Because all mutations are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for. Actions are just plain objects, so they can be logged, serialized, stored, and later replayed for debugging or testing purposes.
Reducers :
pure functions that can be grouped by domain based on the state tree.

REDUX COMPONENTS



Store :
Create a Redux store holding the state of your app.
Its API is { subscribe, dispatch, getState }.
let store = createStore(counter);

Reducer :
pure function with (state, action) => state signature.
It describes how an action transforms the state into the next state.



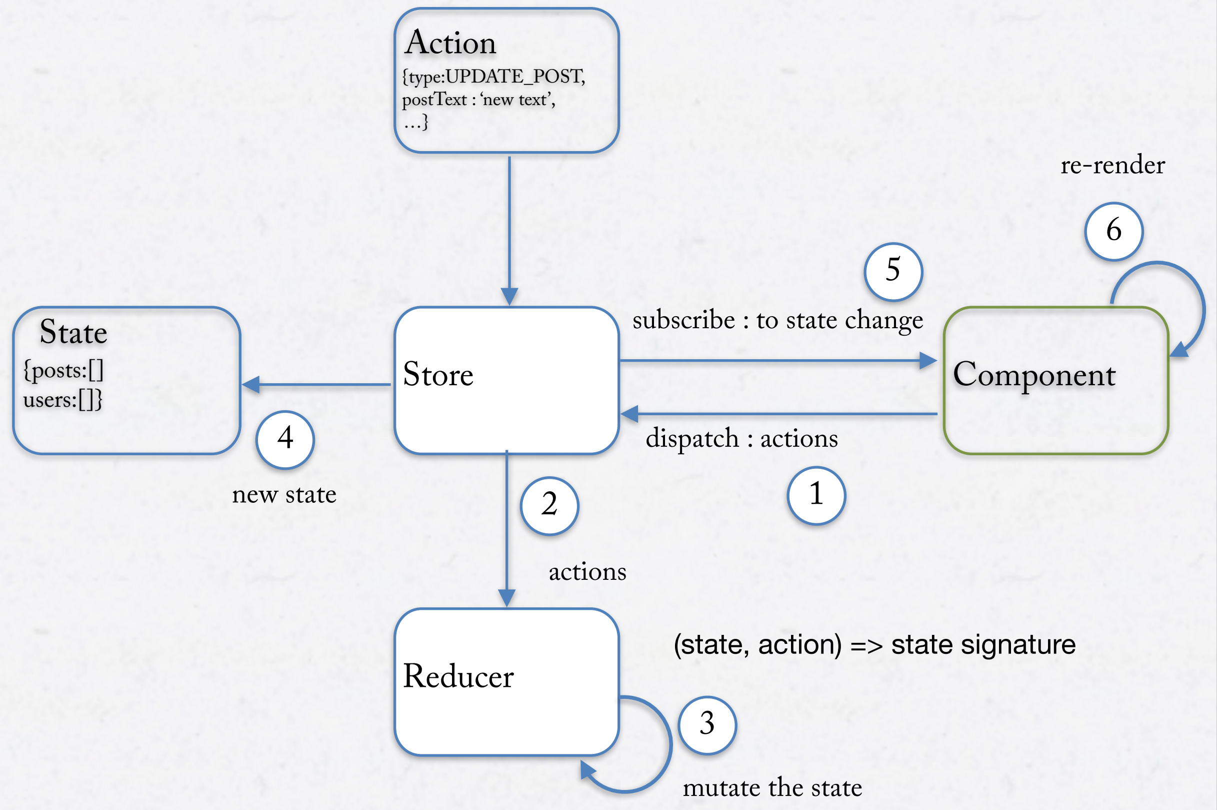
REDUX - STATE

- The shape of the state is up to you: it can be a primitive, an array, an object, or even an Immutable.js data structure.
- The only important part is that :
    - **you should not mutate the state object, but return a new object if the state changes.**

CALLISTA

Within the client we have the notion of remote and client actions
The remote actions will be picked up by the middleware and sent to the server
The client actions will just be handled by the clients redux store and reducers

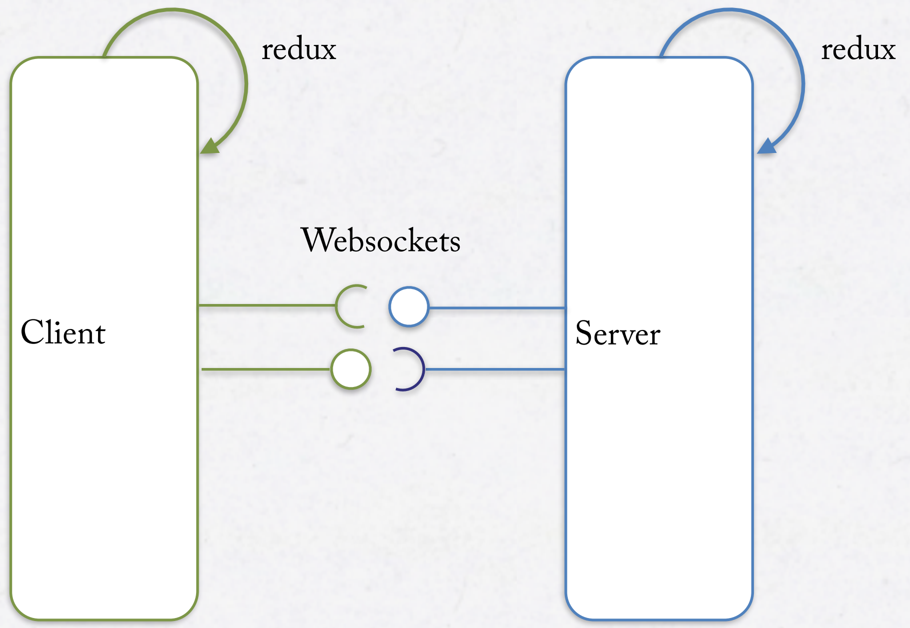
REDUX FLOW



1. dispatch an action, change of text, button click etc ..
2. store passes the action to the reducer
3. reducer mutates the state
4. the state gets replaced with the newly mutated state
5. the store informs listeners of a state change
6. on setState, the component will re-render

BLOGGS APP

- Demo - a look at the bloggs app

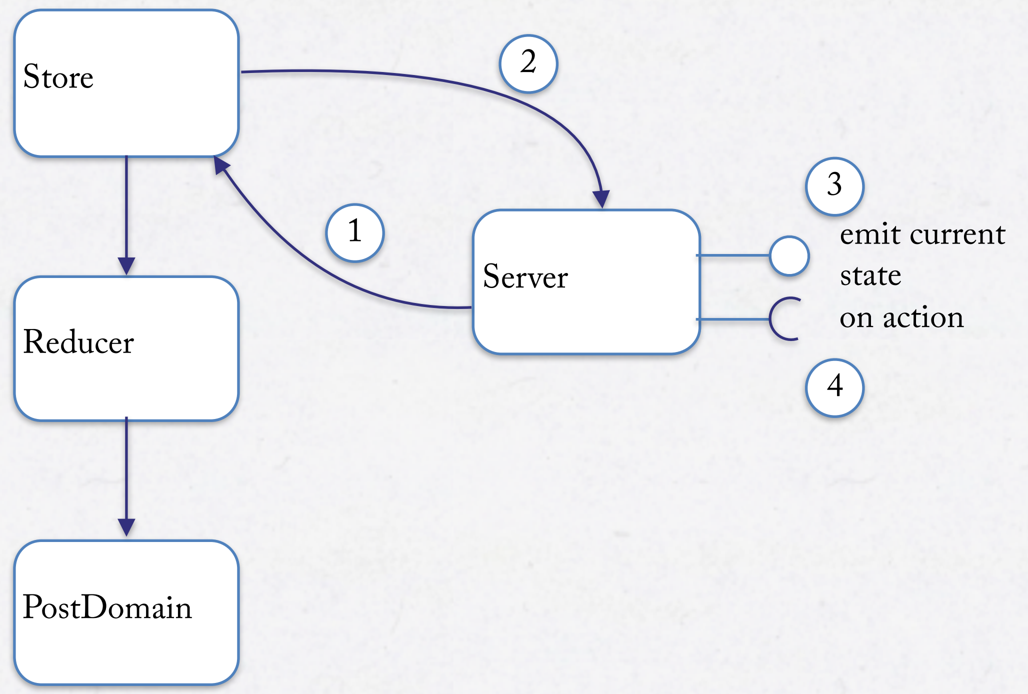


ooh web sockets

BLOGGS APP

The Server …

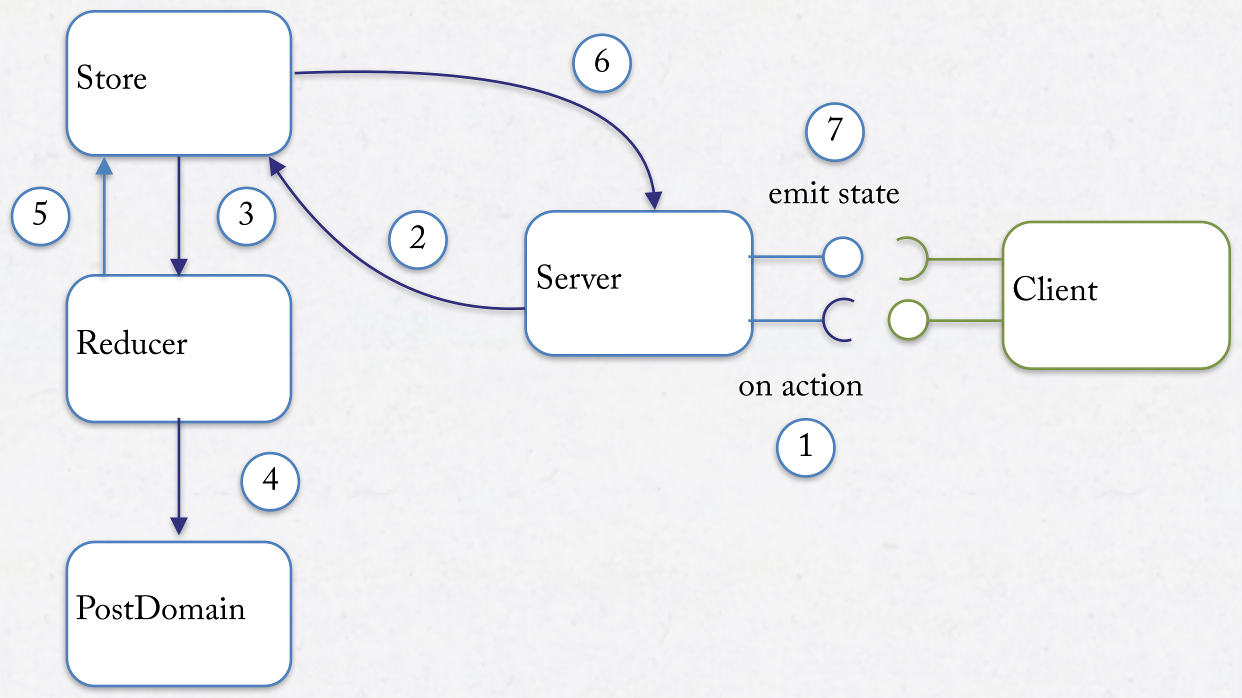
BLOGGS APP - SERVER ON START UP



As the server starts up it goes through the following steps
1. creates a store with a reducer
2. io subscribes to state changes on the server
    1. emits these state changes to the clients
3. io, on connection
    1. emits the current state to the clients
    2. on action, dispatch the client action to the store

The server basically has one in and out chanel :
1. reception of actions which it will dispatch to the store
2. emitting of state updates to the clients.

BLOGGS APP - SERVER ON ACTION



On Reception of an action the server behaves like this :

1. a client sends an action to the server
2. the server hands the action to the redux store
3. the store calls the reducer and the reducer executes the logic related to the action.
4. Post domain updates the state
5. the store updates its state based on the return value of the reducer
6. the store executes the listener function subscribed by the server.
7. the server emits a state event
8. all connected clients - inlcuding the one that initiated the original action - receive the new state

BLOGGS APP - SERVER TESTING

- We can test this without the client
- In isolation and with
  - Mocha - testing framework
  - Chai - assertions
- Demo …

Demo :
so if we take the flow we’ve just gone through we can easily simulate this without having to start a heavy server …

BLOGGS APP

Server Testing…

BLOGGS APP

Client Routing…

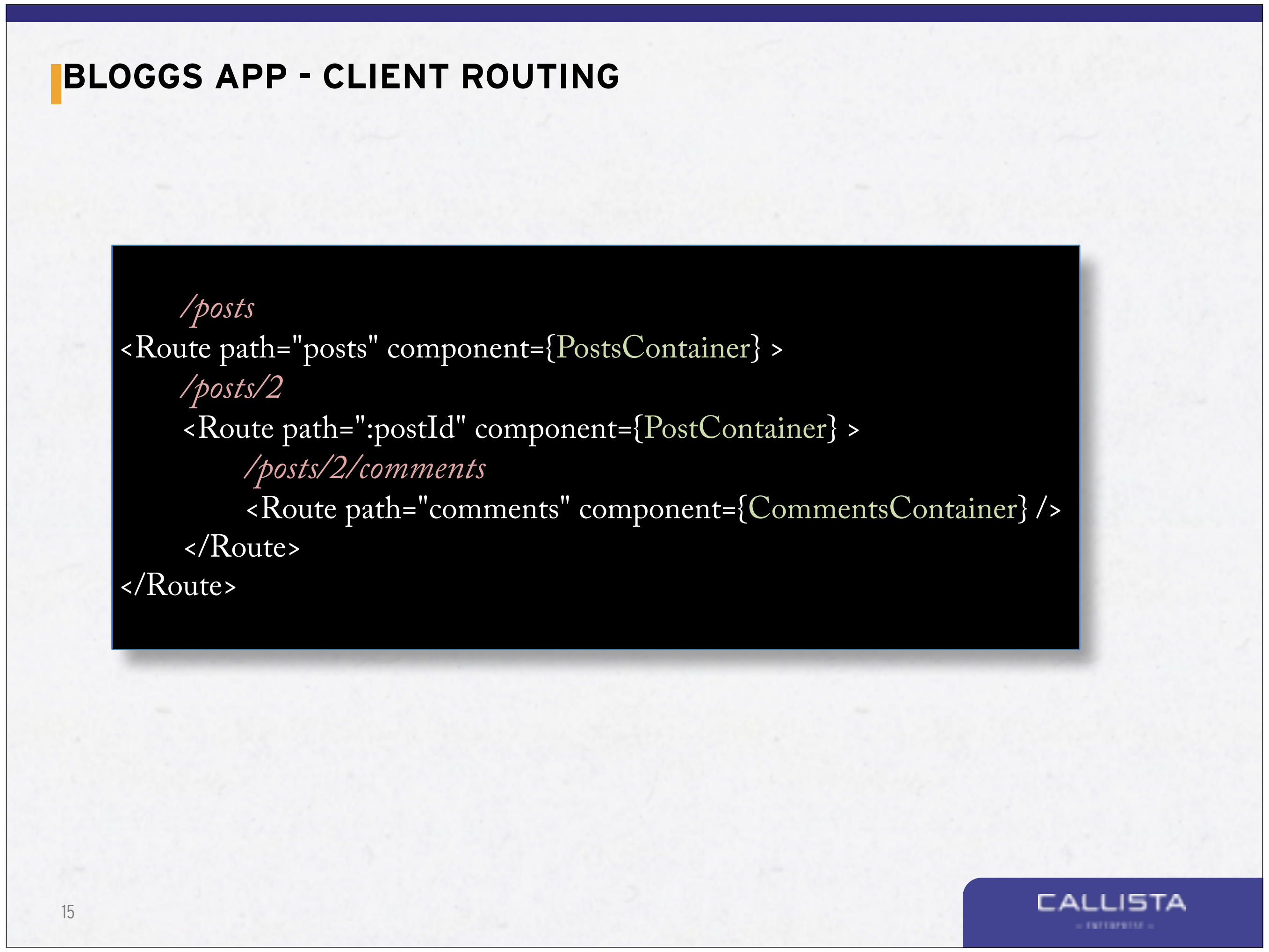
BLOGGS APP - CLIENT ROUTING

```
<Router>
  <Route path="/" component={App}>
    <IndexRoute component={PostsContainer} />
    <Route path="posts" component={PostsContainer} >
      <Route path=":postId" component={PostContainer} >
        <Route path="comments" component={CommentsContainer} />
      </Route>
    </Route>
    <Route path="users" component={UsersContainer} >
      <Route path="new" component={UserContainer} />
    </Route>
  </Route>
</Router>
```

The router contains a hierarchy of routes split into :
Posts
Users

The posts route has a sub route for post and then for comments :

The users route just has one sub route for creating a new user.



BLOGGS APP - CLIENT ROUTING

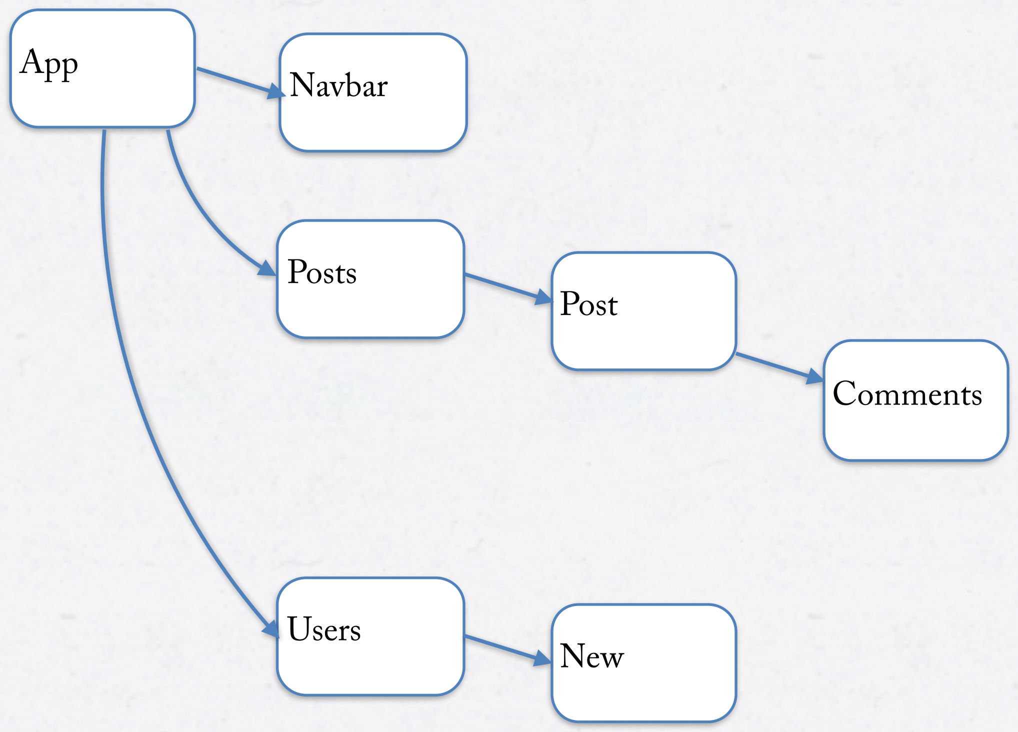
```
    /posts
<Route path="posts" component={PostsContainer} >
    /posts/2
    <Route path=":postId" component={PostContainer} >
        /posts/2/comments
        <Route path="comments" component={CommentsContainer} />
    </Route>
</Route>
```

Above you can see the how the routes are displayed in the urls

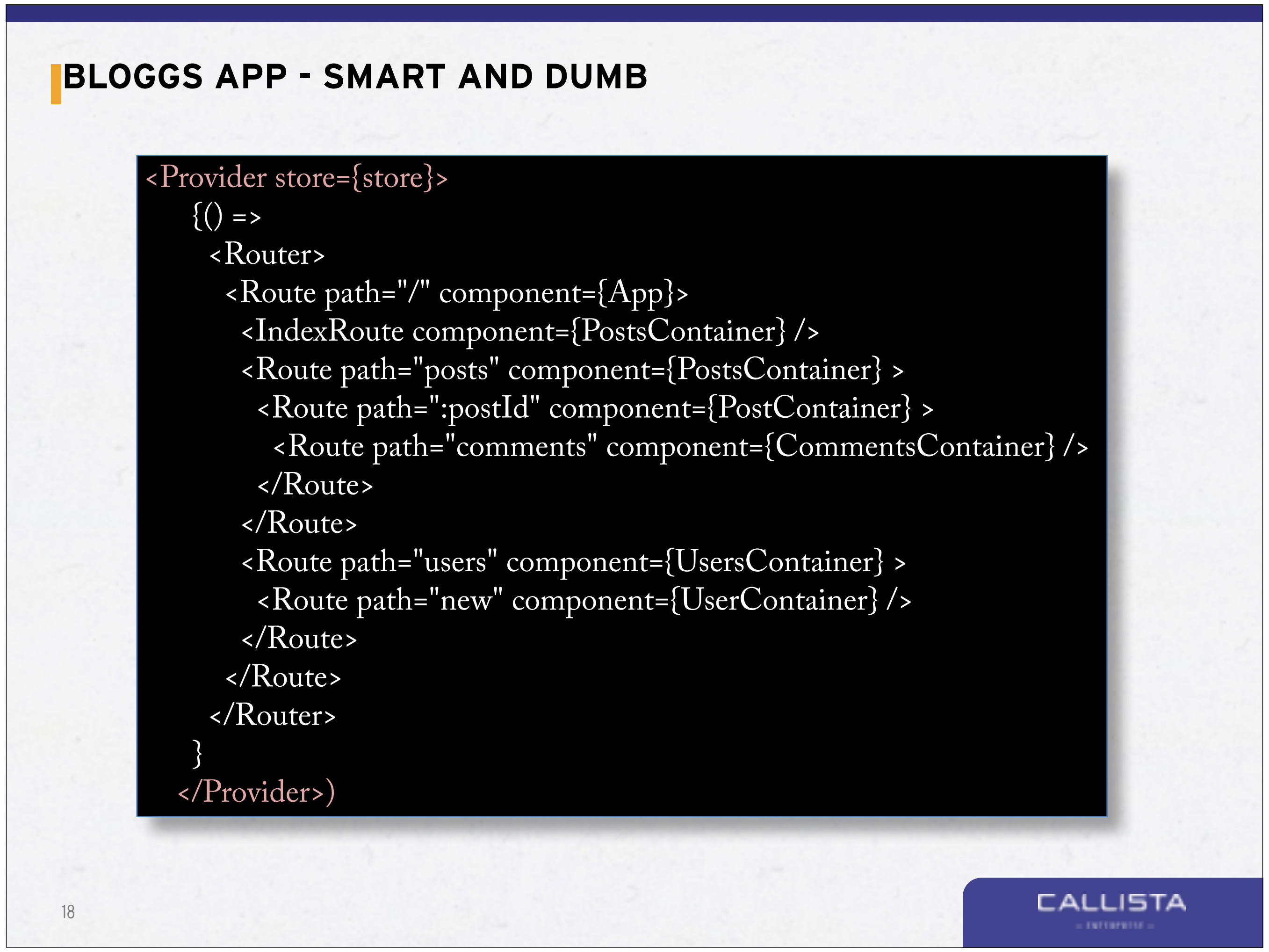
BLOGGS APP - SMART AND DUMB

	"Smart" Components	"Dumb" Components
Location	Top level, route handlers	Middle and leaf components
Aware of Redux	Yes	No
To read data	Subscribe to Redux state	Read data from props
To change data	Dispatch Redux actions	Invoke callbacks from props

BLOGGS APP - SMART AND DUMB



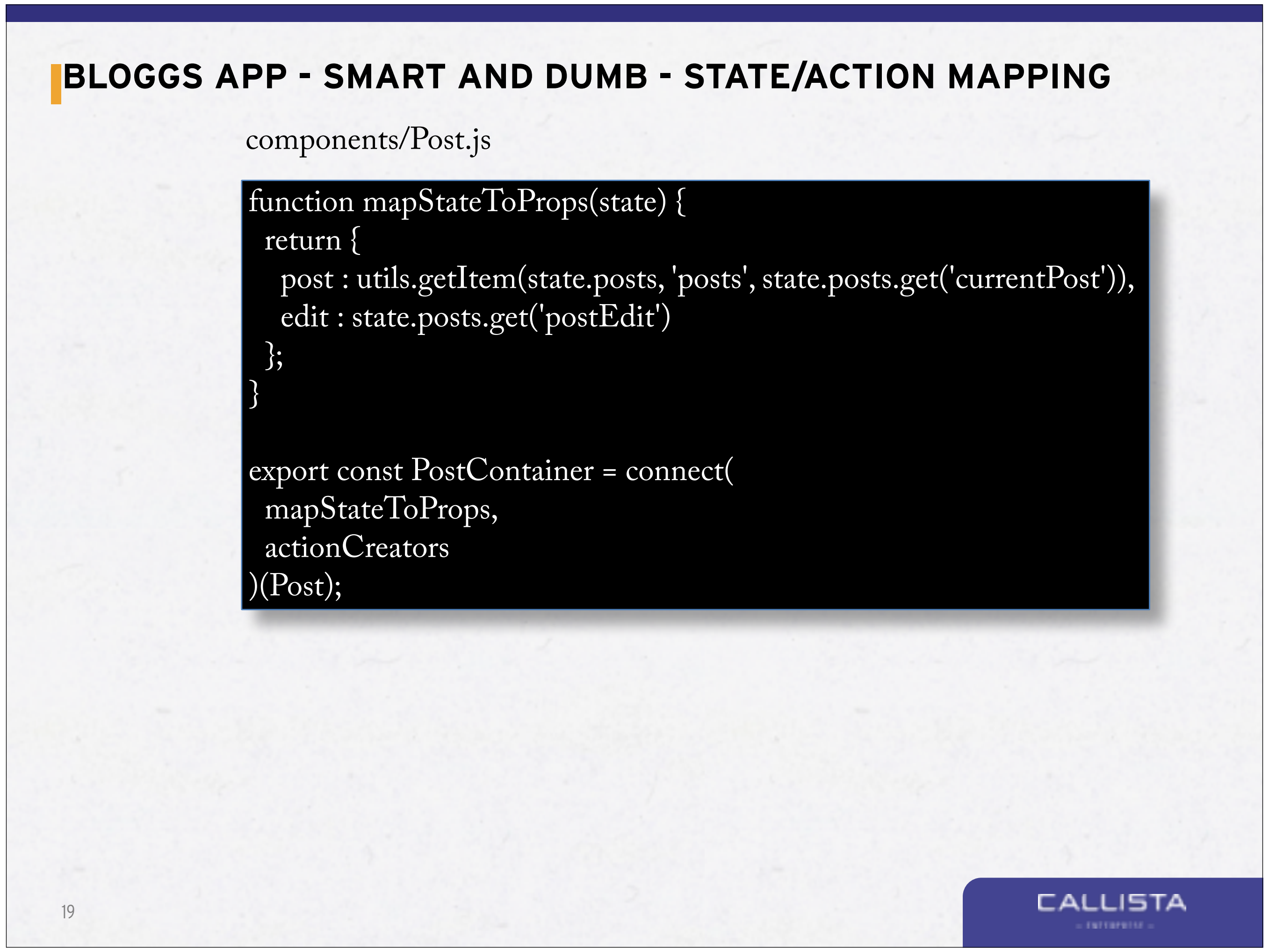
Each route should be responsible for it’s piece of state.
Redux likes to distinguish between smart and dumb components.
In our case all components are attached to a Route or as of 0.13 a route handler.
As a result they’re connected to the redux store with some very nasty code ….



BLOGGS APP - SMART AND DUMB

```
<Provider store={store}>
   {() =>
    <Router>
      <Route path="/" component={App}>
        <IndexRoute component={PostsContainer} />
        <Route path="posts" component={PostsContainer} >
          <Route path=":postId" component={PostContainer} >
            <Route path="comments" component={CommentsContainer} />
          </Route>
        </Route>
        <Route path="users" component={UsersContainer} >
          <Route path="new" component={UserContainer} />
        </Route>
      </Route>
    </Router>
   }
  </Provider>)
```

1. We need to make the router redux aware by wrapping it a Provide component
2. Making sure the store is passed in as an argument.



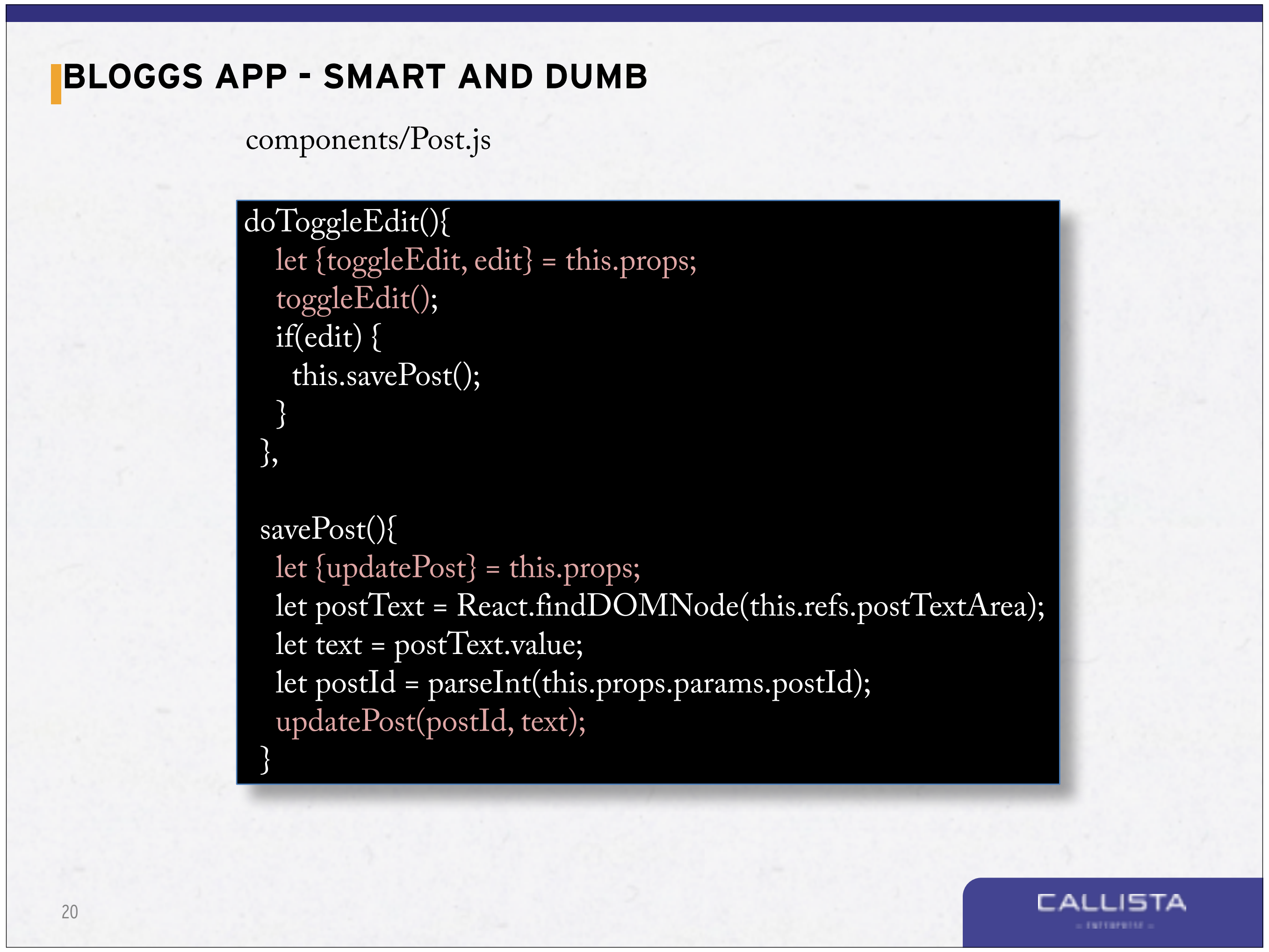
BLOGGS APP - SMART AND DUMB - STATE/ACTION MAPPING

components/Post.js

```
function mapStateToProps(state) {
  return {
    post : utils.getItem(state.posts, 'posts', state.posts.get('currentPost')),
    edit : state.posts.get('postEdit')
  };
}

export const PostContainer = connect(
  mapStateToProps,
  actionCreators
)(Post);
```

1. Then we need to connect the state to properties in the components
2. Each component will be concerned with a certain slice of the state which we express in the connect clojure
3. We then create a Container component which wraps the component in the redux state and actions …



BLOGGS APP - SMART AND DUMB

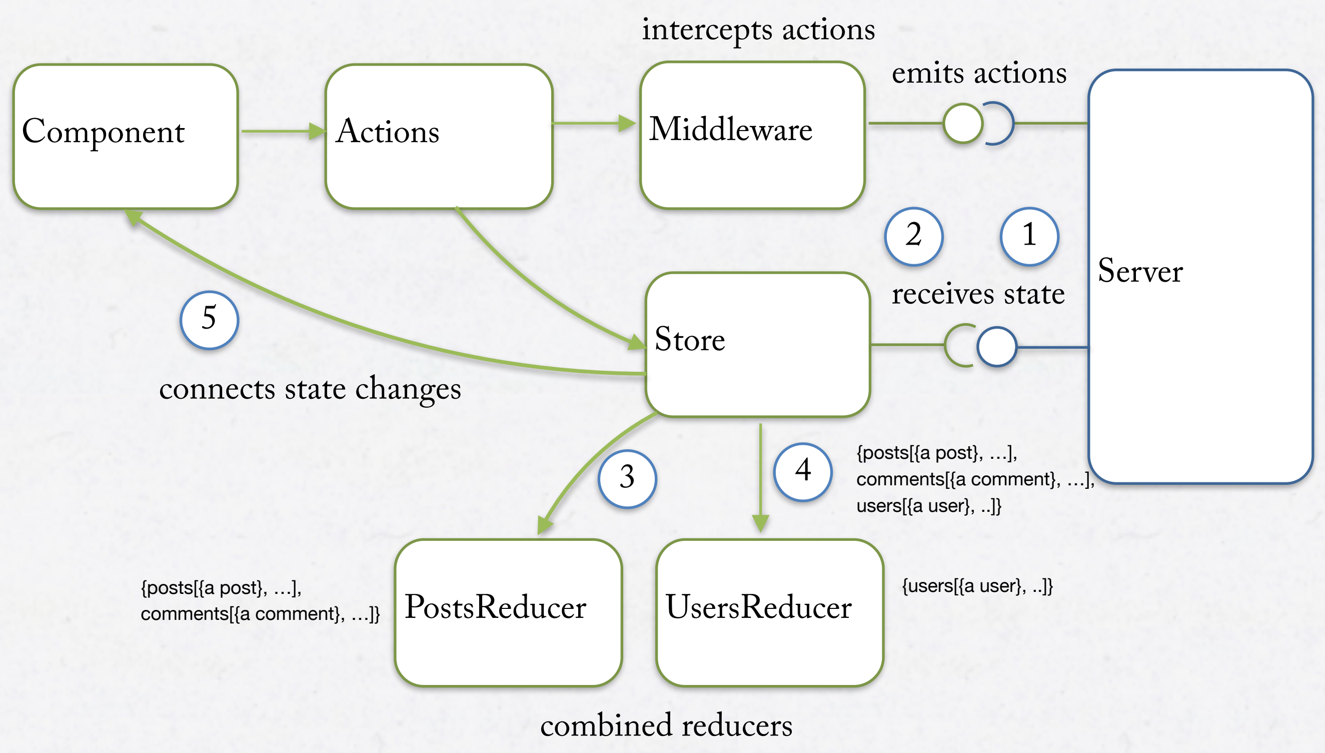
components/Post.js

```
doToggleEdit(){
    let {toggleEdit, edit} = this.props;
    toggleEdit();
    if(edit) {
      this.savePost();
    }
  },

  savePost(){
    let {updatePost} = this.props;
    let postText = React.findDOMNode(this.refs.postTextArea);
    let text = postText.value;
    let postId = parseInt(this.props.params.postId);
    updatePost(postId, text);
  }
```

1. the actions are now on the props hash
2. we can call them directly
3. they will fire an action through the store and into the reducers

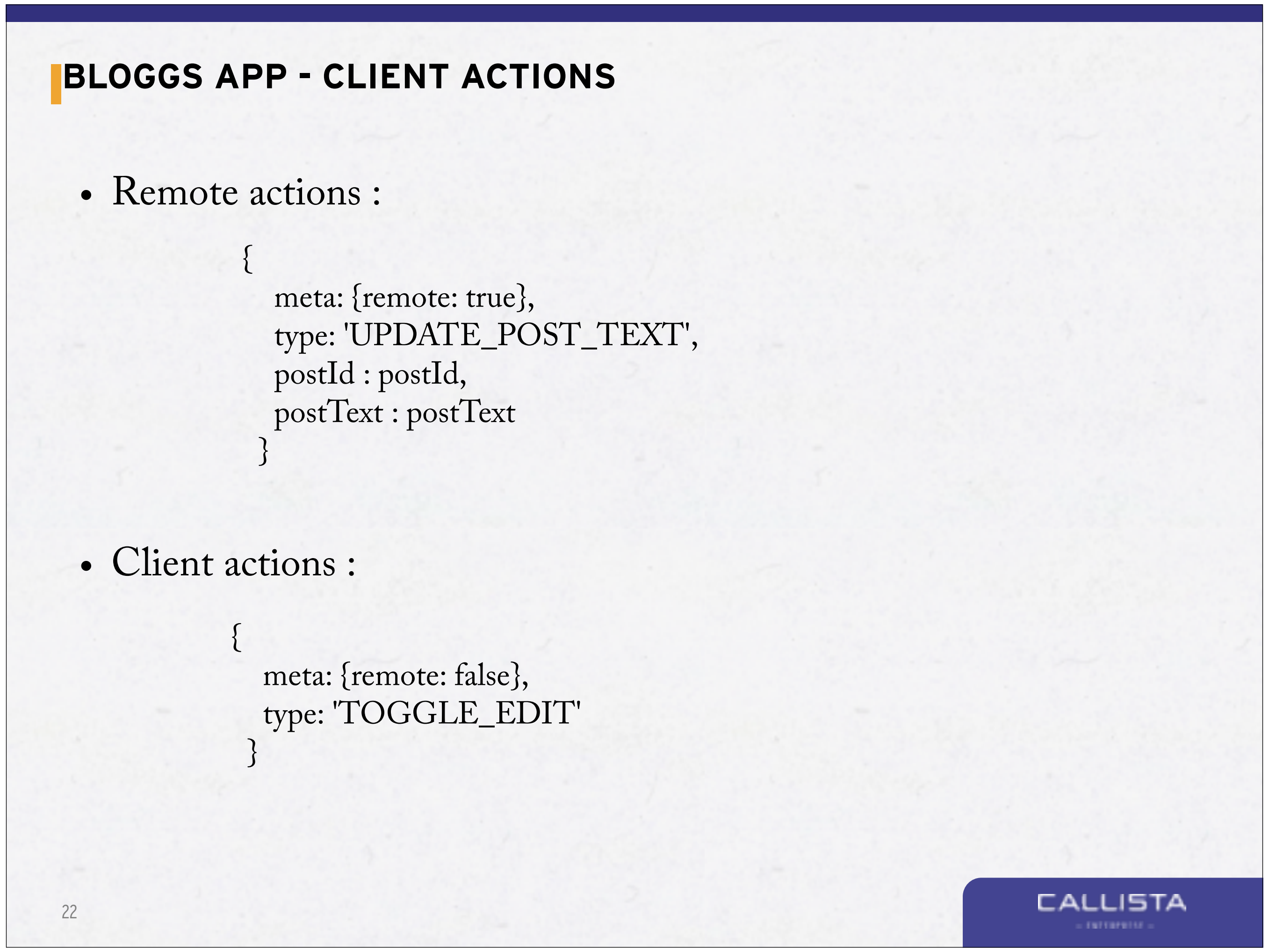
BLOGGS APP - CLIENT, RECEIVE STATE



Client also has a redux store and a number of reducers …
So, when the client starts up a connection is made to the backend server.
This enables us to receive state changes.

1. On reception of the state
2. a SET_DATA action is passed to the reducers
    1. each reducer will extract the data it needs for it’s actions
3. UsersReducer - user [] extracts it’s state
4. PostsReducer - posts[], comments[] extracts it’s state
5. when the state changes the connected react components will receive a setState call and update their state and .. re-render.

Demo this on screen



BLOGGS APP - CLIENT ACTIONS

- Remote actions :

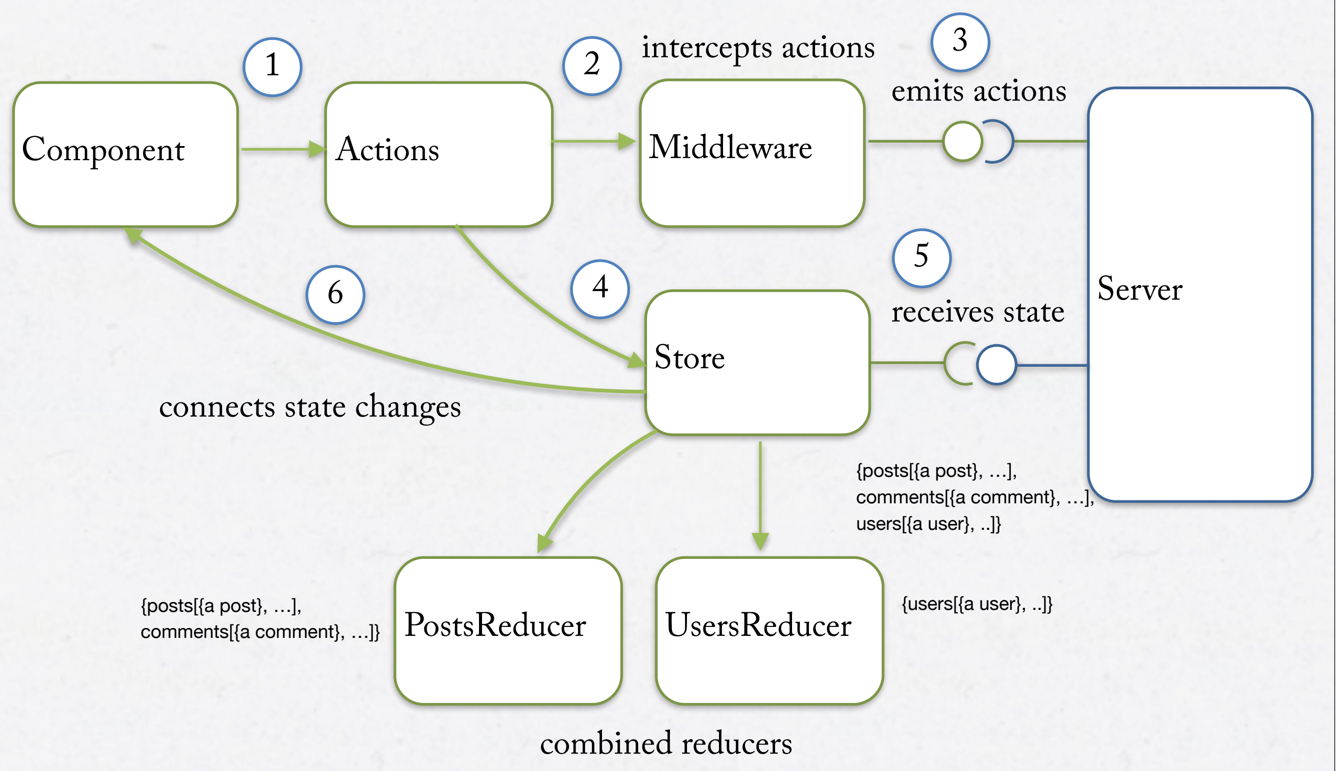
```
{
   meta: {remote: true},
   type: 'UPDATE_POST_TEXT',
   postId : postId,
   postText : postText
  }
```

- Client actions :

```
{
   meta: {remote: false},
   type: 'TOGGLE_EDIT'
  }
```

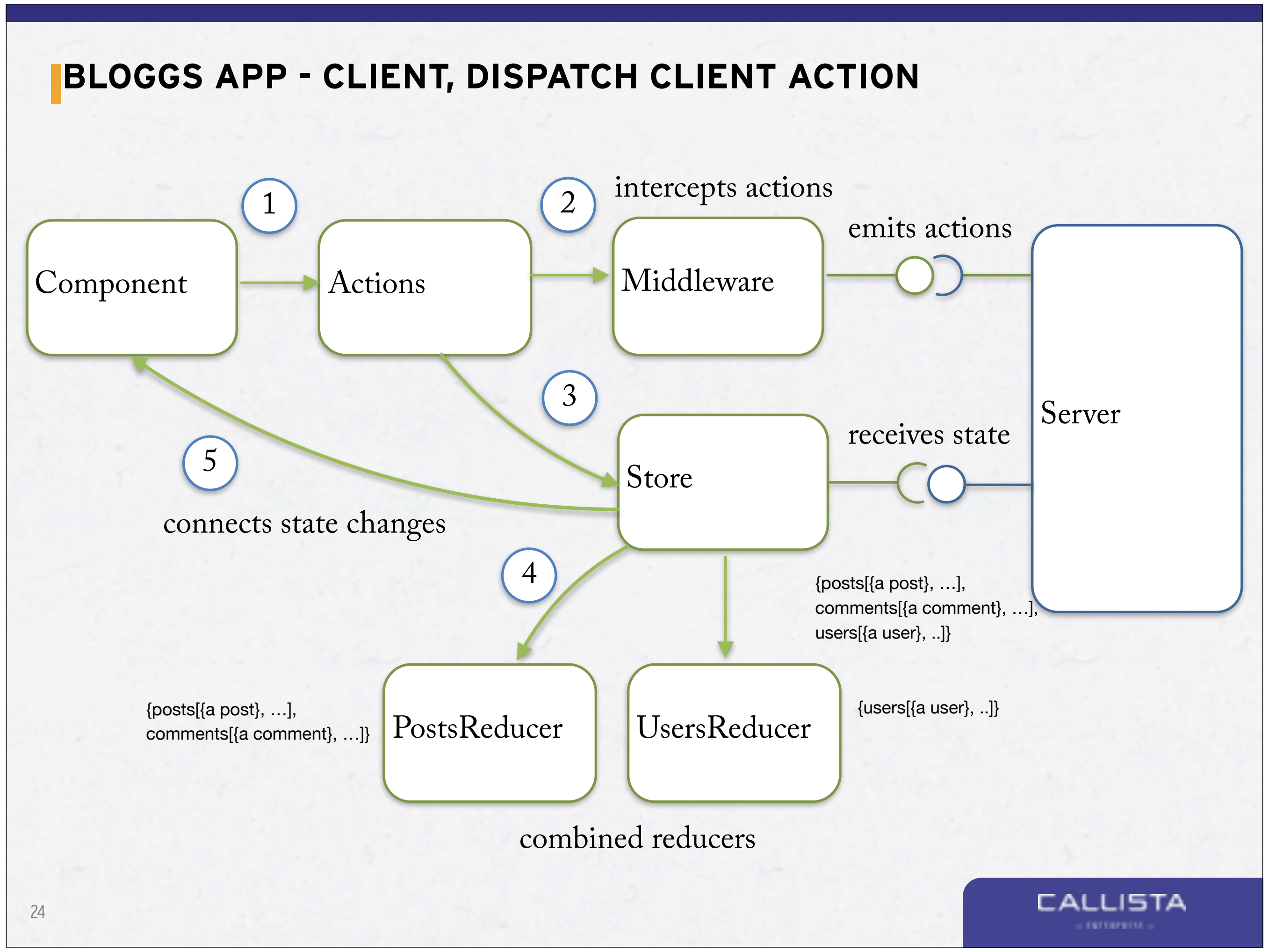
Within the client we have the notion of remote and client actions
The remote actions will be picked up by the middleware and sent to the server
The client actions will just be handled by the clients redux store and reducers

BLOGGS APP - CLIENT, DISPATCH REMOTE ACTION



1. update a post action is fired from the post component
2. the middleware intercepts this action, determines it’s a remote action
3. middleware sends the action to the server
4. The client store takes the action and passes it through it’s reducers, although it just passes through.2
5. the client receives an update
6. the components receive a new state and re-render

Demo this in action …



BLOGGS APP - CLIENT, DISPATCH CLIENT ACTION

1. user clicks edit post
2. the middleware intercepts this action, but does nothing as its remote prop is false
3. the store takes the action and passes it onto its reducers
4. the reducers handle the TOGGLE_EDIT action and update the client state
5. the components receive a new state and re-render

Demo this in action …

REDUX

Reflections…

REDUX - REFLECTIONS

- At first it seems like **a lot of code** that is an easy one liner in Ember or Angular
- Although, the concept of **unidirectional state brings traceability** and the ability to **clearly reason** about the state of your application.
- As you application grows this will become even more important for **maintainability**.
- Easy to work with and extremely **testable** (as was seen in the server implementation)
- I'v used this with socket.io and it works like a **dream**! especially the **interchange of client remote actions to the server**.
- Would be great to test this with a rest api, assume the resolution of a promise would then trigger another action with a new state
- **Immutable state**! it's the way to go!

REDUX - CONCLUSIONS

- Testing!
- Webpack
- Hot loading
- ES6/7
- Developer Experience!