**Delivery**

**FNZ - On boarding – Reference Document**

|  |  |
| --- | --- |
| Prepared For: |  |
| Date: | 27th October 2016 |
| Reference: | v0.1 (draft) |

TABLE OF CONTENTS

[1. Engagement Overview 4](#_Toc465364409)

[2. Domain Overview 4](#_Toc465364410)

[3. Business overview 4](#_Toc465364411)

[4. D2C Portal Overview 4](#_Toc465364412)

[5. Advisor Portal Overview 4](#_Toc465364413)

[6. FNZ Execution Model 4](#_Toc465364414)

[7. Associate Roles and responsibilities 4](#_Toc465364415)

[a. Analysis 4](#_Toc465364416)

[b. Coding 4](#_Toc465364417)

[c. Code Review 4](#_Toc465364418)

[d. Unit Testing 4](#_Toc465364419)

[e. Delivery 4](#_Toc465364420)

[8. Teams 4](#_Toc465364421)

[9. Machine Setup 4](#_Toc465364422)

[10. Softwares and tools used 4](#_Toc465364423)

[a. Accurev 4](#_Toc465364424)

[b. Confluence 4](#_Toc465364425)

[c. JIRA 4](#_Toc465364426)

[d. Code Collaborator 4](#_Toc465364427)

[e. Resharper 4](#_Toc465364428)

[11. Developer Specific 4](#_Toc465364429)

[a. MVC to TSX 4](#_Toc465364430)

[b. React Lifecycle , FLUX 4](#_Toc465364431)

[c. Run Custom Tool, Creating Updating JSON 4](#_Toc465364432)

[d. JSXElement, Render, State, Props, Lightbox 4](#_Toc465364433)

[e. URL.ts, Routing, Context 4](#_Toc465364434)

[f. Action, Store, Static Store 4](#_Toc465364435)

[g. Events in tsx 4](#_Toc465364436)

[h. Wizard Stepper 4](#_Toc465364437)

[i. DataApplicationTarget, URL, Layout, Route 4](#_Toc465364438)

[j. Shared folders, Helper Files, Resources 4](#_Toc465364439)

[k. Naming Convention 4](#_Toc465364440)

[l. Error Handling 4](#_Toc465364441)

[m. Exception Handling 4](#_Toc465364442)

[n. Hi Chart 4](#_Toc465364443)

[o. Controller(1 Get, 1Post, 1Put 1 Delete) , Adapter, ContextType 4](#_Toc465364444)

[p. MVC vs WebAPI calls (HTTPGet, HTTPPost, HTTPPut(Not allowed)) 4](#_Toc465364445)

[q. Multiple Service calls in single controller, error handling, common error enumeration 4](#_Toc465364446)

[r. Model, Model Mapping, Error Dictionary [Enum under resources vs error message mapping] 4](#_Toc465364447)

[s. Effective LINQ usage, Aggregare, filter, sort, avg, for loops 4](#_Toc465364448)

[t. SOLID - Single responsibility and Dependency Injection 4](#_Toc465364449)

[u. Resharper - Xtest, Effective usage of Resharper 4](#_Toc465364450)

[v. Service Reference Update, Xtest 4](#_Toc465364451)

[w. Basic Database Queries 5](#_Toc465364452)

[12. Multi Language support 5](#_Toc465364453)

[13. DevOps process 5](#_Toc465364454)

# Engagement Overview

# Domain Overview

# Business overview

# D2C Portal Overview

This portal allows the end user to investment on various products. These products are grouped under below accounts

* Investment Account (ISA)
* Junior Investment Account (JISA)
* General Investment Account (GIA)

Screenshots of Vanguard application can be referred in below attachment.

# Advisor Portal Overview

This portal allows the advisors to manage investments of their clients. AVIVA applications is built to serve both Advisors and customers

# FNZ Execution Model

# Associate Roles and responsibilities

# Analysis

# Coding

# Code Review

# Unit Testing

# Delivery

# Teams

There are 4 projects in FNZ engagement.

1. Vanguard

Vanguard is a D2C portal developed for FNZ’s VANGUARD client.

As of **15-Nov-2016**, Most development is complete. Few enhancements and defects fixing are ongoing.

3 Member team.

1. AVIVA

AVIVA is a Advisor portal developed for FNZ’s AVIVA client.

As of **15-Nov-2016**, development is in progress. Work is assignment id done via JIRA.

6 Member team.

1. Reorient

Reorient is a D2C portal developed for FNZ’s Reorient client.

As of **15-Nov-2016**, development is in progress. Work is assignment id done via JIRA.

8 Member team

1. BBVA

BBVA is a D2C portal developed for FNZ’s BBVA client.

As of **15-Nov-2016**, development is in progress. Work is assignment id done via JIRA.

6 Member team

# Machine Setup

Every individual will be provided with a VM by FNZ. Associates can login using below URL

<https://somewhereelse.fnz.co.uk/collab>

* User credentials will be provided by FNZ
* Login process will automatically install Juniper client required for remote login.
* Once logged in, the associate will see the remote desktop.
* Ensure to install below softwares.
  + Microsoft Visual Studio Professional 2013

# Navigate [\\tpdad1\Developer Tools\](file://tpdad1/Developer%20Tools/) from VM machine

# Copy Visual Studio Pro 2013 VLM to your local path

# Open the folder “Visual Studio Pro 2013 VLM” from your local path

# Navigate to “SW\_DVD5\_Visual\_Studio\_Pro\_2013\_English\_-2\_MLF\_X19-27145”

# Run the “vs\_professional” setup

# Go with the default options and complete the setup

# Navigate back to “Visual Studio Pro 2013 VLM”

# Run the “VS2013.5” setup

# Select the package path to “..\Visual Studio Pro 2013 VLM\Update 5\packages\vsupdate\_KB2829760\Preparation.exe”

# Complete the setup with default options

* + SQL Server Management Studio

# Install software from [\\tpdad1\Developer Tools\](file://tpdad1/Developer%20Tools/) in VM machine

* + TYPESCRIPT Installation

# Navigate [\\tpdad1\Developer Tools\](file://tpdad1/Developer%20Tools/) from VM machine

# Open the folder “Type Script for Visual Studio”

# Run the “TypeScript\_Dev12” setup

# Go with the default options and complete the setup

* + Resharper.2016.2

# Navigate to [\\tpdad1\Developer Tools\Resharper.2016.2](file://tpdad1/Developer%20Tools/Resharper.2016.2)

# Install Resharper

# Get licence from project manger and update your VM

* + Accurev

# Navigate [\\tpdad1\Developer Tools\](file://tpdad1/Developer%20Tools/) from VM machine

# Open the folder “Type Script for Visual Studio”

# During installation, provide Host name – pescm1 and Port – 5080

# Complete the installation

# Open Accurev -> use Accurev username and password to login(see attached mail) -> change the password once you are in

# Click on View Streams -> Select SRC in the list -> click OK

# Ctrl +F ->

# VANGUARD - Search for “SRC\_UI\_Vanguard\_Dev”

# AVIVA - Search for “SRC\_UI\_One\_21.0\_Dev”

# REORIENT - Search for “SRC\_UI\_Reorient\_Dev”

# BBVA - Search for “SRC\_UI\_BBVA\_Dev”

# In the search results, right click on the searched name, and then -> select “New workspace”

# Point your code directory to “C:/Source/” and click OK

# Click on Finish. (Don’t click Next button)

# Code will get downloaded in the local path

* + Code Collaborator

# Navigate [\\tpdad1\Developer Tools\](file://tpdad1/Developer%20Tools/) from VM machine

# Locate code collaborator, and install

# Server UTL : <http://pecod1>

# Provide userid and password given to you

* The steps to install, configure and use are documented in below attached reference document.



* + JIRA

# Ensure to able to navigate to <https://projects.fnz.com/jira/secure/Dashboard.jspa> and login

# Check if able to view the respective project items

* + Confluence

# Ensure to able to navigate to <https://portal.fnz.com/confluence> and login

# Check if able to view the respective project items

# Software’s and tools used

* 1. VISUAL STUDIO 2013
  2. SQL Server Management Studio
  3. TYPESCRIPT
  4. Accurev

Accurev is the source code control tool used in FNZ development environment.

# Confluence

Confluence is used as a document repository system. And it can be access using below link.

<https://portal.fnz.com/confluence>

For Reorient, confluence is used to track the queries. Steps are taken to track the queries using Confluence for all the projects.

Confluence is not used by Cognizant for Vanguard project (as of 20 Nov 2016)

# JIRA

JIRA is a project management planning and tracking tool. JIRA is accessed using below URL

<https://projects.fnz.com/jira/secure/Dashboard.jspa>

Work will be assigned in JIRA as Story or enhancement or defects.

Developer must log the hours in JIRA to respective work item. This may vary from project to project.

# Code Collaborator

Code Collaborator is a tool used to do the peer code reviews.

URL : <http://pecode1>

# Resharper

Resharper.2016.2 is used in FNZ environment. It is installed from below location.

[\\tpdad1\Developer Tools\Resharper.2016.2](file://tpdad1/Developer%20Tools/Resharper.2016.2)

# Code quality analysis

On-the-fly code quality analysis is available in C#, VB.NET, XAML, ASP.NET, JavaScript, TypeScript, CSS, HTML, and XML. ReSharper will tell you right away if your code contains errors or can be improved.

# Code editing helpers

Multiple code editing helpers are available, such as extended IntelliSense, hundreds of instant code transformations, auto-importing namespaces, rearranging code and displaying documentation.

# Code generation

You don't have to write properties, overloads, implementations, and comparers by hand: use code generation actions to handle boilerplate code faster.

# Eliminate errors and code smells

Instant fixes help eliminate errors and code smells. Not only does ReSharper warn you when there are problems in your code but it provides quick-fixes to solve them automatically.

# Safely change your code base

Apply solution-wide refactorings to safely change your code base. Whether you need to revitalize legacy code or put your project structure in order, you can lean on ReSharper.

# Compliance to coding standards

Use code formatting and cleanup to get rid of unused code and ensure compliance to coding standards.

# Unit Testing

ReSharper provides a unit test runner using xTest. Use Resharper to unit test the code before checking in.

# XTest Plugin for Resharper

Install Xtest plugin for Resharper.

# Continuous Integration

Team City is used for Continuous integration. Below URL can be used to track the CI status.

<http://pebuildmbc/overview.html>

The developer, when he checks in a code, must ensure that

1. Before Check-in -> All xtest cases passes using Visual Studio – Resharper in developer machine.
2. After Check-in -> Login to <http://pebuildmbc/overview.html> and ensure that CI picks up the changes, and the status is green.

# Developer Specific

# 

# What is React

# React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, giving a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional data binding.

# FLUX Architecture

Flux is the application architecture that Facebook uses for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework, and you can start using Flux immediately without a lot of new code.

Flux applications have three major parts: the dispatcher, the stores, and the views (React components). These should not be confused with Model-View-Controller. Controllers do exist in a Flux application, but they are controller-views — views often found at the top of the hierarchy that retrieve data from the stores and pass this data down to their children. Additionally, action creators — dispatcher helper methods — are used to support a semantic API that describes all changes that are possible in the application. It can be useful to think of them as a fourth part of the Flux update cycle.

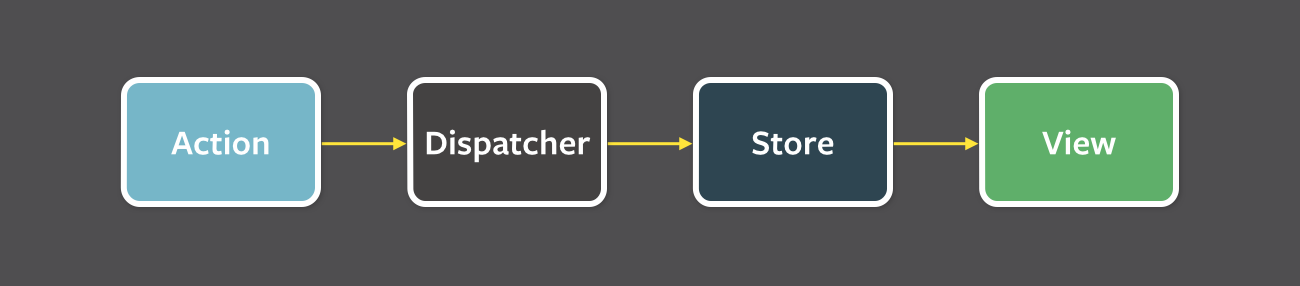
Flux eschews MVC in favor of a unidirectional data flow. When a user interacts with a React view, the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected. This works especially well with React's declarative programming style, which allows the store to send updates without specifying how to transition views between states.

We originally set out to deal correctly with derived data: for example, we wanted to show an unread count for message threads while another view showed a list of threads, with the unread ones highlighted. This was difficult to handle with MVC — marking a single thread as read would update the thread model, and then also need to update the unread count model. These dependencies and cascading updates often occur in a large MVC application, leading to a tangled weave of data flow and unpredictable results.

Control is inverted with stores: the stores accept updates and reconcile them as appropriate, rather than depending on something external to update its data in a consistent way. Nothing outside the store has any insight into how it manages the data for its domain, helping to keep a clear separation of concerns. Stores have no direct setter methods like setAsRead(), but instead have only a single way of getting new data into their self-contained world — the callback they register with the dispatcher.

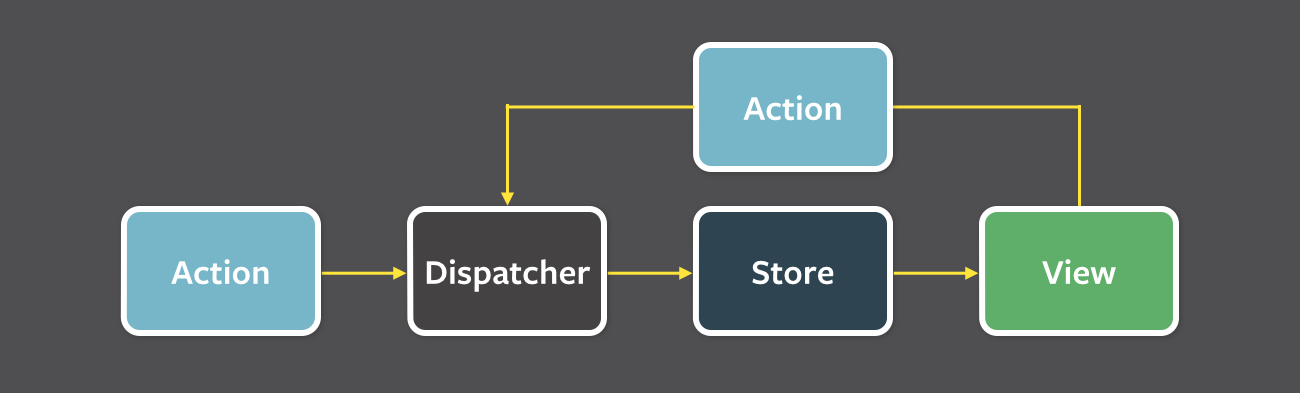
## Structure and Data Flow

Data in a Flux application flows in a single direction:

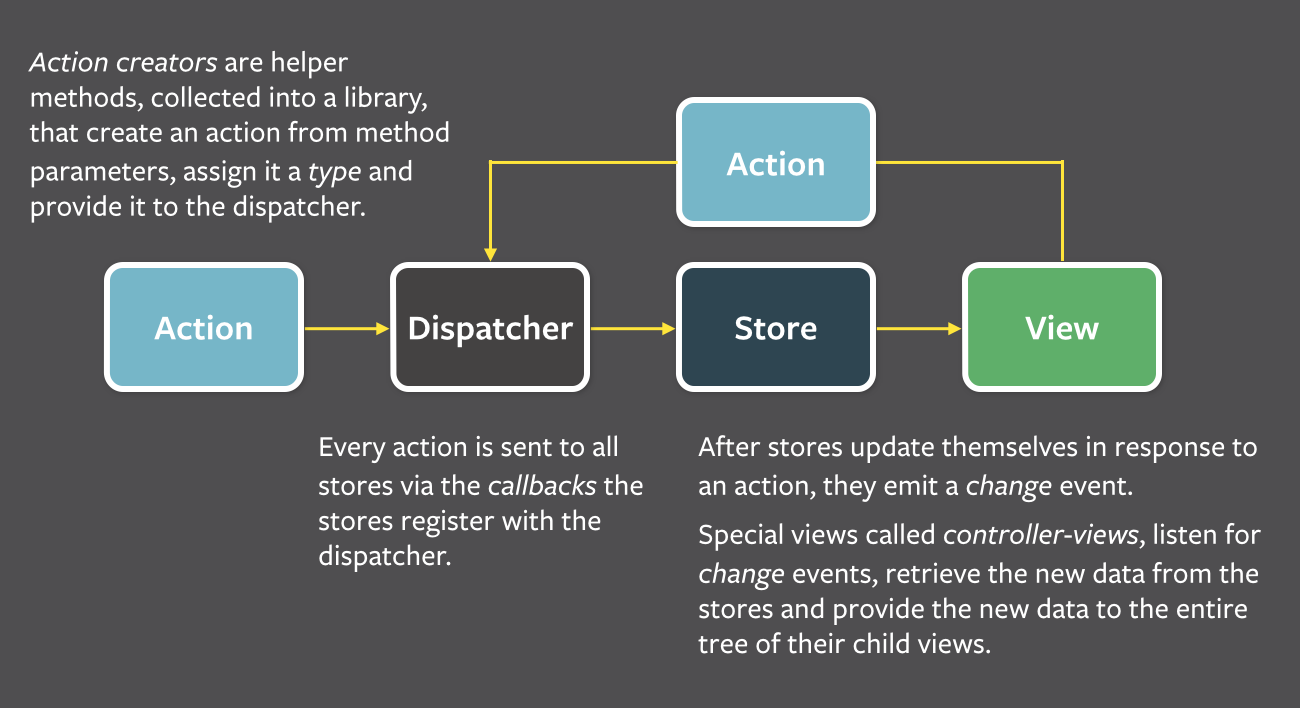


A unidirectional data flow is central to the Flux pattern, and the above diagram should be **the primary mental model for the Flux programmer**. The dispatcher, stores and views are independent nodes with distinct inputs and outputs. The actions are simple objects containing the new data and an identifying *type* property.

The views may cause a new action to be propagated through the system in response to user interactions:



All data flows through the dispatcher as a central hub. Actions are provided to the dispatcher in an *action creator* method, and most often originate from user interactions with the views. The dispatcher then invokes the callbacks that the stores have registered with it, dispatching actions to all stores. Within their registered callbacks, stores respond to whichever actions are relevant to the state they maintain. The stores then emit a *change* event to alert the controller-views that a change to the data layer has occurred. Controller-views listen for these events and retrieve data from the stores in an event handler. The controller-views call their ownsetState() method, causing a re-rendering of themselves and all of their descendants in the component tree.



This structure allows us to reason easily about our application in a way that is reminiscent offunctional reactive programming, or more specifically data-flow programming or flow-based programming, where data flows through the application in a single direction — there are no two-way bindings. Application state is maintained only in the stores, allowing the different parts of the application to remain highly decoupled. Where dependencies do occur between stores, they are kept in a strict hierarchy, with synchronous updates managed by the dispatcher.

We found that two-way data bindings led to cascading updates, where changing one object led to another object changing, which could also trigger more updates. As applications grew, these cascading updates made it very difficult to predict what would change as the result of one user interaction. When updates can only change data within a single round, the system as a whole becomes more predictable.

Let's look at the various parts of Flux up close. A good place to start is the dispatcher.

### A Single Dispatcher

The dispatcher is the central hub that manages all data flow in a Flux application. It is essentially a registry of callbacks into the stores and has no real intelligence of its own — it is a simple mechanism for distributing the actions to the stores. Each store registers itself and provides a callback. When an action creator provides the dispatcher with a new action, all stores in the application receive the action via the callbacks in the registry.

As an application grows, the dispatcher becomes more vital, as it can be used to manage dependencies between the stores by invoking the registered callbacks in a specific order. Stores can declaratively wait for other stores to finish updating, and then update themselves accordingly.

The same dispatcher that Facebook uses in production is available through [npm](https://www.npmjs.com/package/flux" \t "_blank), [Bower](http://bower.io/), or[GitHub](https://github.com/facebook/flux).

### Stores

Stores contain the application state and logic. Their role is somewhat similar to a model in a traditional MVC, but they manage the state of many objects — they do not represent a single record of data like ORM models do. Nor are they the same as Backbone's collections. More than simply managing a collection of ORM-style objects, stores manage the application state for a particular **domain** within the application.

For example, Facebook's [Lookback Video Editor](https://facebook.com/lookback/edit) utilized a TimeStore that kept track of the playback time position and the playback state. On the other hand, the same application's ImageStore kept track of a collection of images. The TodoStore in our [TodoMVC example](https://github.com/facebook/flux/tree/master/examples/flux-todomvc/" \t "_blank) is similar in that it manages a collection of to-do items. A store exhibits characteristics of both a collection of models and a singleton model of a logical domain.

As mentioned above, a store registers itself with the dispatcher and provides it with a callback. This callback receives the action as a parameter. Within the store's registered callback, a switch statement based on the action's type is used to interpret the action and to provide the proper hooks into the store's internal methods. This allows an action to result in an update to the state of the store, via the dispatcher. After the stores are updated, they broadcast an event declaring that their state has changed, so the views may query the new state and update themselves.

### Views and Controller-Views

React provides the kind of composable and freely re-renderable views we need for the view layer. Close to the top of the nested view hierarchy, a special kind of view listens for events that are broadcast by the stores that it depends on. We call this a controller-view, as it provides the glue code to get the data from the stores and to pass this data down the chain of its descendants. We might have one of these controller-views governing any significant section of the page.

When it receives the event from the store, it first requests the new data it needs via the stores' public getter methods. It then calls its own setState() or forceUpdate() methods, causing itsrender() method and the render() method of all its descendants to run.

We often pass the entire state of the store down the chain of views in a single object, allowing different descendants to use what they need. In addition to keeping the controller-like behavior at the top of the hierarchy, and thus keeping our descendant views as functionally pure as possible, passing down the entire state of the store in a single object also has the effect of reducing the number of props we need to manage.

Occasionally we may need to add additional controller-views deeper in the hierarchy to keep components simple. This might help us to better encapsulate a section of the hierarchy related to a specific data domain. Be aware, however, that controller-views deeper in the hierarchy can violate the singular flow of data by introducing a new, potentially conflicting entry point for the data flow. In making the decision of whether to add a deep controller-view, balance the gain of simpler components against the complexity of multiple data updates flowing into the hierarchy at different points. These multiple data updates can lead to odd effects, with React's render method getting invoked repeatedly by updates from different controller-views, potentially increasing the difficulty of debugging.

### Actions

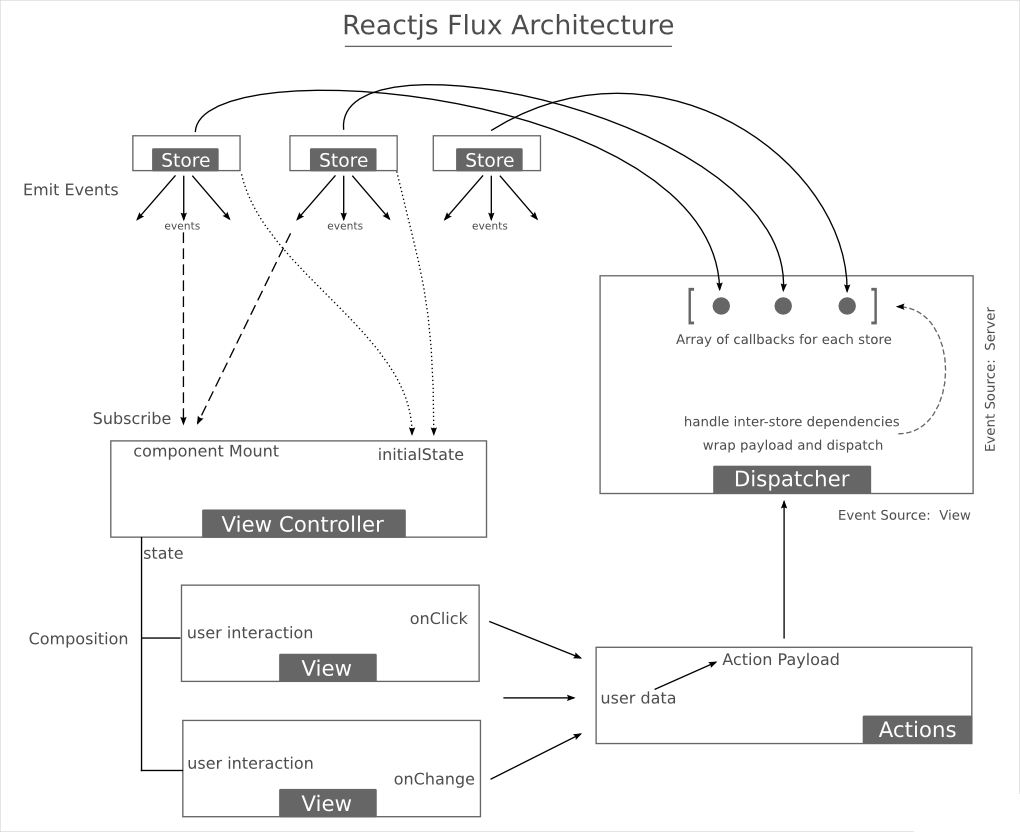
The dispatcher exposes a method that allows us to trigger a dispatch to the stores, and to include a payload of data, which we call an action. The action's creation may be wrapped into a semantic helper method which sends the action to the dispatcher. For example, we may want to change the text of a to-do item in a to-do list application. We would create an action with a function signature like updateText(todoId, newText) in our TodoActions module. This method may be invoked from within our views' event handlers, so we can call it in response to a user interaction. This action creator method also adds a type to the action, so that when the action is interpreted in the store, it can respond appropriately. In our example, this type might be named something like TODO\_UPDATE\_TEXT.

Actions may also come from other places, such as the server. This happens, for example, during data initialization. It may also happen when the server returns an error code or when the server has updates to provide to the application.

### What About that Dispatcher?

As mentioned earlier, the dispatcher is also able to manage dependencies between stores. This functionality is available through the waitFor() method within the Dispatcher class. We did not need to use this method within the extremely simple [TodoMVC application](https://github.com/facebook/flux/tree/master/examples/flux-todomvc/" \t "_blank), but it becomes vital in a larger, more complex application.

Within the TodoStore's registered callback we could explicitly wait for any dependencies to first update before moving



# React - Detail

**Introducing JSX**

React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, but there are some pros that comes with it.

* JSX is faster because it performs optimization while compiling code to JavaScript.
* It is also type-safe and most of the errors can be caught during compilation.
* JSX makes it easier and faster to write templates if you are familiar with HTML.

Refer for details : <https://facebook.github.io/react/docs/introducing-jsx.html> and

<https://www.tutorialspoint.com/reactjs/reactjs_jsx.htm>

**Rendering Elements**

Elements are the smallest building blocks of React apps. An element describes what you want to see on the screen. Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

Refer for details: <https://facebook.github.io/react/docs/rendering-elements.html>

**Components and Props**

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen

Refer for details : <https://facebook.github.io/react/docs/components-and-props.html>

**Component Lifecycle Methods**

* componentWillMount is executed before rendering, on both server and client side.
* componentDidMount is executed after first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution like setTimeoutor setInterval. We are using it to update the state so we can trigger the other lifecycle methods.
* componentWillReceiveProps is invoked as soon as the props are updated before another render is called. We triggered it fromsetNewNumber when we updated the state.
* shouldComponentUpdate should return true or false value. This will determine if component will be updated or not. This is set to true by default. If you are sure that component doesn't need to render afterstate or props are updated, you can return false value.
* componentWillUpdate is called just before rendering.
* componentDidUpdate is called just after rendering.
* componentWillUnmount is called after the component is unmounted from the dom. We are unmounting our component in main.js.

**Reference:** <https://www.tutorialspoint.com/reactjs/reactjs_component_life_cycle.htm> and

<https://facebook.github.io/react/docs/state-and-lifecycle.html>

**State**

State is the place where the data comes from. You should always try to make your state as simple as possible and minimize number of stateful components. If you have, for example, ten components that need data from the state, you should create one container component that will keep the state for all of them.

**Props**

The main difference between state and props is that props are immutable. This is why container component should define state that can be updated and changed, while the child components should only pass data from the state using props.

Properties validation is useful way to force correct usage of your components. This will help you during development to avoid future bugs and problems once your app become larger. It also makes code more readable since you can see how each component should be used.

**Validating Props**

In this example we are creating App component with all the props that we need. App.propTypes is used for props validation. If some of the props aren't using correct type that we assigned, we will get console warning. After we specified validation patterns, we are setting App.defaultProps.

**Handling Events**

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences: React events are named using camelCase, rather than lowercase. With JSX you pass a function as the event handler, rather than a string.

Refer for details : <https://facebook.github.io/react/docs/handling-events.html>

**Conditional Rendering**

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Refer for Details: <https://facebook.github.io/react/docs/conditional-rendering.html>

# MVC to TSX

# Run Custom Tool, Creating Updating JSON

# JSXElement, Render, State, Props, Lightbox

# URL.ts, Routing, Context

# Action, Store, Static Store

# Events in tsx

# Wizard Stepper

# Application Context

# DataApplicationTarget, URL, Layout, Route

# Shared folders, Helper Files, Resources

# Naming Convention

# Error Handling

# Exception Handling

# Hi Chart

# Controller(1 Get, 1Post, 1Put 1 Delete) , Adapter, ContextType

# MVC vs WebAPI calls (HTTPGet, HTTPPost, HTTPPut(Not allowed))

# Multiple Service calls in single controller, error handling, common error enumeration

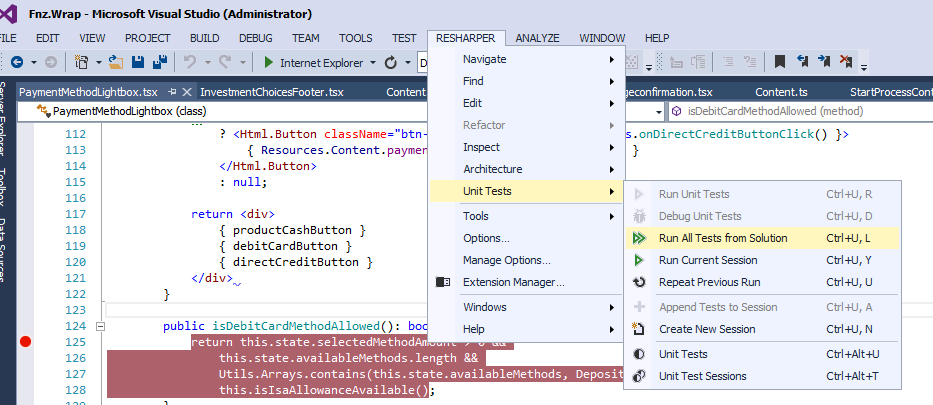
# Model, Model Mapping, Error Dictionary [Enum under resources vs error message mapping]

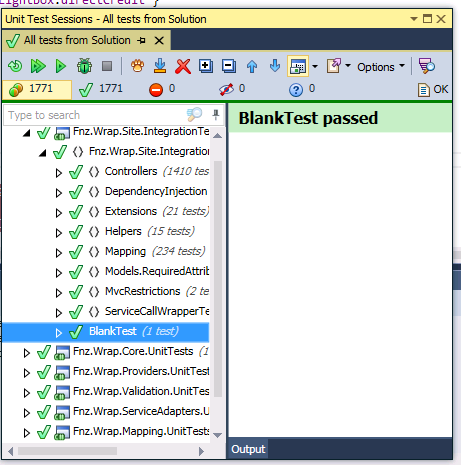
# Effective LINQ usage, Aggregare, filter, sort, avg, for loops

# SOLID - Single responsibility and Dependency Injection

# Resharper – Xtest usage

Run unit test under Resharper menu. Ensure XTest plugin is installed.





All the tests should have passed before promoting the code.

# Quick guide to update service reference

**Prerequisites**:

         Notepad++ (or another application that can run find and replace for all files in the folder)

         Old version of service API, e.g. 21.4.0.40

-          check the Fnz.Distribution.Service.Api in packages.config in ServiceAdapter project

         New version of service API, e.g. 21.4.0.42

-          someone from service will provide this to you

         ServiceExceptions.xml file with service metadata for **the version of new API** (for 21.4.0.42 in our case)

-          someone from service team can generate this for you or you can download it here: <https://te85service.fnzc.co.uk/api/distribution/v3/help/exceptions/xml> (keep in mind that you need it for the correct version of API)

**Steps:**

1.       Close the solution

2.       Open Notepad++

**a.**       Open Find in Files

                                                                i.      Find what: Old version of API , e.g. **21.4.0.40**

                                                               ii.      Replace with: New version of API, e.g. **21.4.0.42**

                                                             iii.      Filters:                 **\*.csproj;\*.vbproj;packages.config;\*.targets**

                                                             iv.      Directory: choose folder structure for …\**Websites\FNZWrap.Web**

v.    Try Find All (there should be 48 occurrences for Vanguard but this can ofc. differs for different projects)

                                                            vi.      Replace in files

**b.**      \*\*\* ONLY in the case you are changing the major or minor version (first to groups of numbers), e.g. from 20.1.1.1 to 21.1.5.8 or 20.1.1.1 to 20.2.5.8 \*\*\*

i.      Repeat the steps in 2. a. but replace the last 2 groups of numbers with 0 for both old and new version, e.g. find and replace 20.1.0.0 with 21.1.0.0 or 20.1.0.0 with 20.2.0.0

3.       Open the solution

4.       Navigate to Fnz.Wrap.ServiceAdapters/ServiceClients

a.       Replace the ServiceExceptions.xml with new one

b.      Run Custom Tool on ServiceClients.tt (right click)

5.       **Rebuild the solution** and fix the errors caused by changing the API

6.       **Run Unit tests** and fix the errors

7.       Run the site (just login page to make sure it is loaded correctly)

8.       Done

# Basic Database Queries

# Multi Language support

TypeScript Language Resource Files Builder



Working with the Multi-Language framework on the Reactjs and ASP.NETMVC



# Coding Guidelines

# DevOps process