Tacit

A Modular System for Serving Applications Online

Willem van Heemstra

Tacit

A Modular System for Serving Applications Online

Willem van Heemstra

Contents

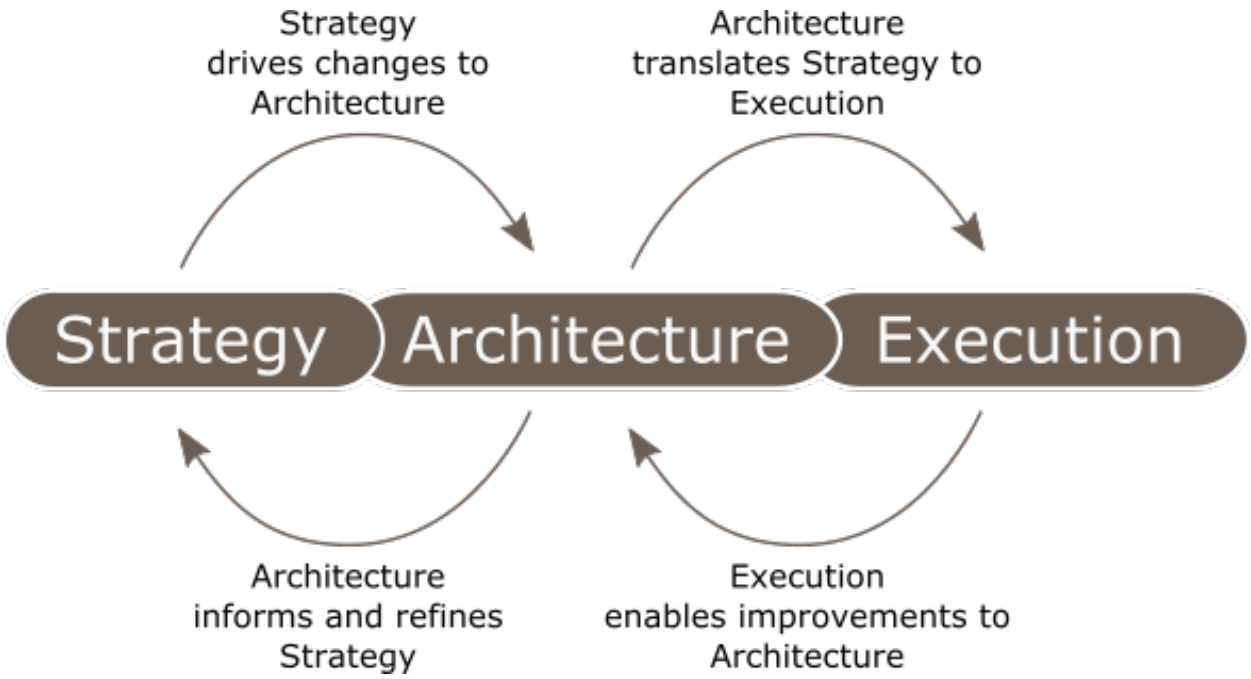
Chapter 1

Overview

The Tacit enterprise consists of the following viewpoints:

- strategy: drives changes to architecture
- architecture: translates strategy to execution, informs and refines strategy
- execution: enables improvements to architecture

Their relation to one another is depicted below:



The following chapters address each of these individually.

Chapter 2

Strategy

Strategy drives changes to Architecture.

It was a **bold** and *italic* night!

Suddenly, a shot rang out!

The cabin started to rumble…

Chapter 3

Architecture

Architecture translates Strategy to Execution. Architecture informs and refines Strategy.

Requirements

... more

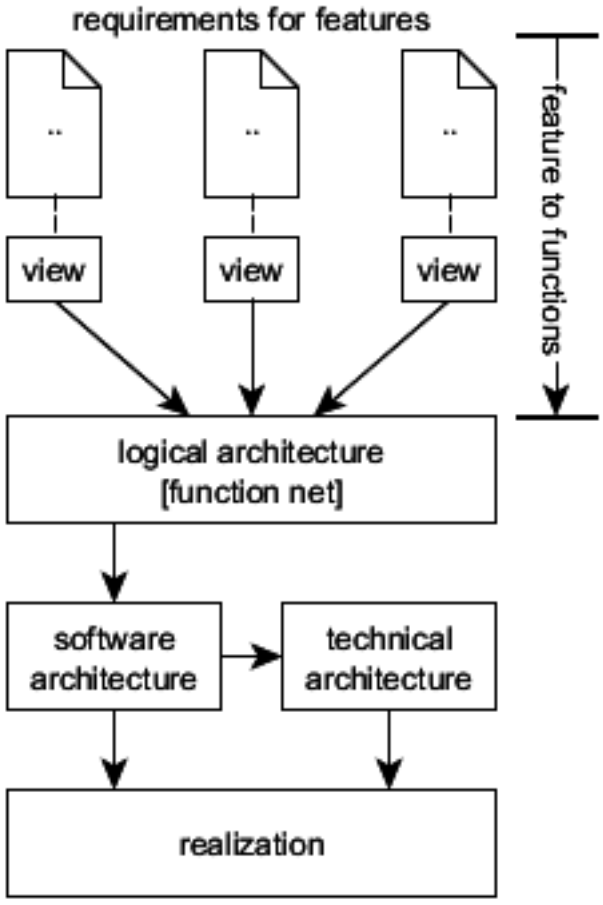
Requirements describe what customers and other stakeholders want from a product or service. Consider using Dynamic Object-Oriented Requirements System (DOORS) for requirement management.

Features

... more

Functions

To translate features to functions the aid of view based function net modeling is applied.



View-Based Function Net Modeling

source: http://www.sse-tubs.de/publications/GHKKR07OMER.pdf

Logical Architecture

... more

Function Net(s)

... more

SysML

SysML block diagrams can be used by modeling functions of an system as blocks. These blocks can be hierarchical decomposed into subblocks that define the internal structure. Blocks can be connected to each other via directed connectors that represent a communication relationship. The connector

can be used across the block hierarchy but also ports can be used to describe a well-defined interface. To increase the reuse, blocks can optionally have a type that allows the multiple instantiation of a single block within a diagram.

Software Architecture

… more

Technical Architecture

… more

Functional Programming with JavaScript

see also https://www.youtube.com/watch?v=BMUiFMZr7vk The technical implementation embraces functional programming as the approach of coding.

Functional programming allowes for ‘composition’ of code, hence it adopts the concept of components at code level. Functions in JavaScript are values. A variable can be assigned a function.

e.g.

var triple = function(x) { return 3 * x }

Higher order functions are functions that can be put inside other functions. - Code will be easier to debug (as the logic is contained inside a small function). - Less code will be required (as code is reused through composition).

Hence, code will be better maintainable and more scalable.

e.g.

var animals = [{ name: ‘Snoopy’, species: ‘dog’}, { name: ‘Roger’, species: ‘rabbit’}, { name: ‘Kermit’, species: ‘frog’}];

alert(‘Animals: ‘ + JSON.stringify(animals));

Task: Filter for frogs.

Traditional approach:

```
var frogs = [];
for (var i = 0; i < animals.length; i++) {
  if(animals[i].species === 'frog')
    frogs.push(animals[i])
}

alert('Frogs: ' + JSON.stringify(frogs));

var names = [];
for (var i = 0; i < animals.length; i++) {
  names.push(animals[i].name)
}

alert('All Names: ' + JSON.stringify(names));
```

Functional approach:

```
var frogs = animals.filter(function(animal) {
  return animal.species === 'frog'
});

alert('Frogs: ' + JSON.stringify(frogs));

var names = animals.map(function(animal) {
  return animal.name
});

alert('All Names: ' + JSON.stringify(names));
```

Here ‘filter’ - a higher order function that is part of javascript’s array object - loops through each element of the animals array and passes the key-value pairs into its callback function. The callback will return either true or false (depending of the value of species being ‘frog’ or not). When it is done, it will retun the new filtered array, frogs.

And ‘map’ - a higher order function that is part of javascript’s array object - transforms each element of the animal array. Here it returns the animal’s name only.

We can make it more functional:

```
var isFrog = function(animal) {
  return animal.species === 'frog'
};

var frogs = animals.filter(isFrog);

alert('Frogs: ' + JSON.stringify(frogs));

var names = animals.map((animal) => animal.name); // Here we use the ECMAScript \
6 arrow syntax, where the value after the arrow is implicitly returned.

alert('All Names: ' + JSON.stringify(names));
```

The JavaScript core language features are defined in a standard called ECMA-262. The language defined in this standard is called **ECMAScript**, of which the JavaScript in the browser and Node.js environments are a superset. While browsers and Node.js may add more capabilities through additional objects and methods, the core of the language remains as defined in ECMAScript. For the new features in ECMAScript 6, see http://es6-features.org/

And extend it with showing animals other than frogs (by using the higher order function 'reject', which comes with the underscore library):

```
var nonFrogs = animals.reject(isFrog);

alert('nonFrogs: ' + JSON.stringify(nonfrogs));
```

Realization

... more

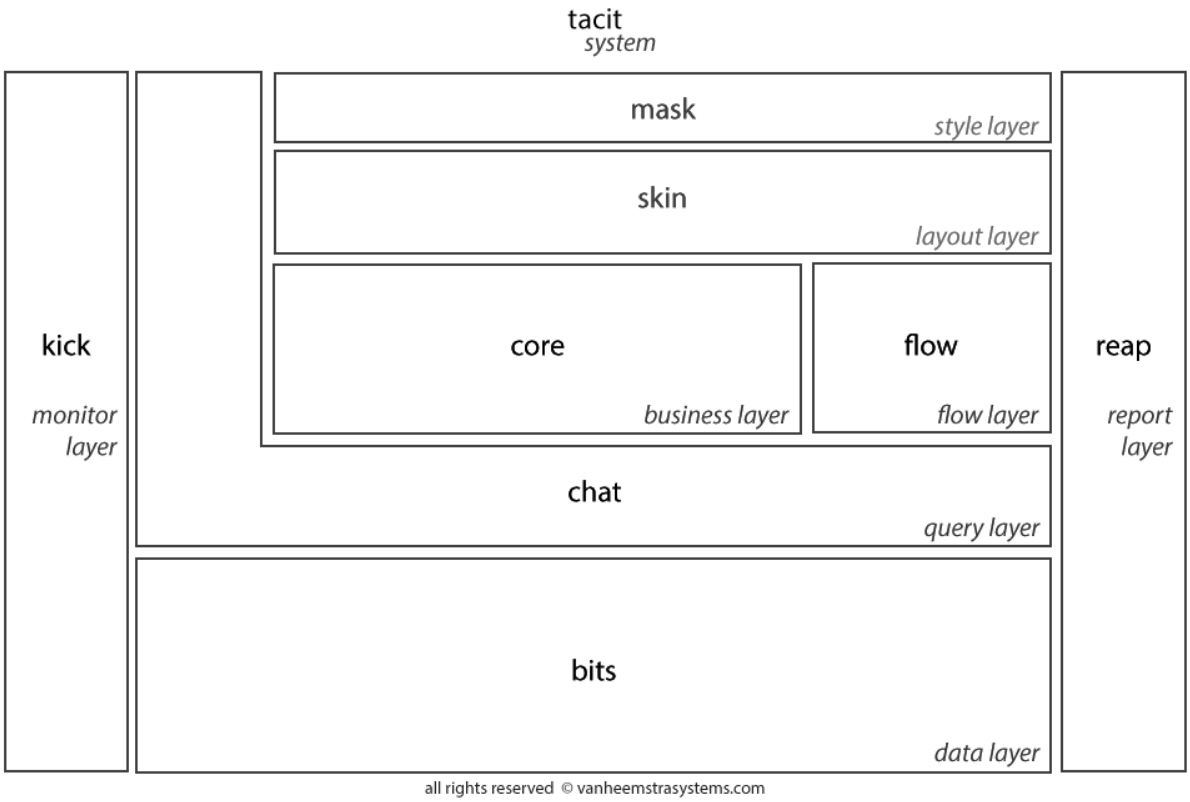
System

The Tacit system consists of the following layers (listed here in no particular order):

- core: business layer
- skin: layout layer
- mask: style layer
- chat: query layer
- bits: data layer
- kick: monitor layer
- reap: report layer

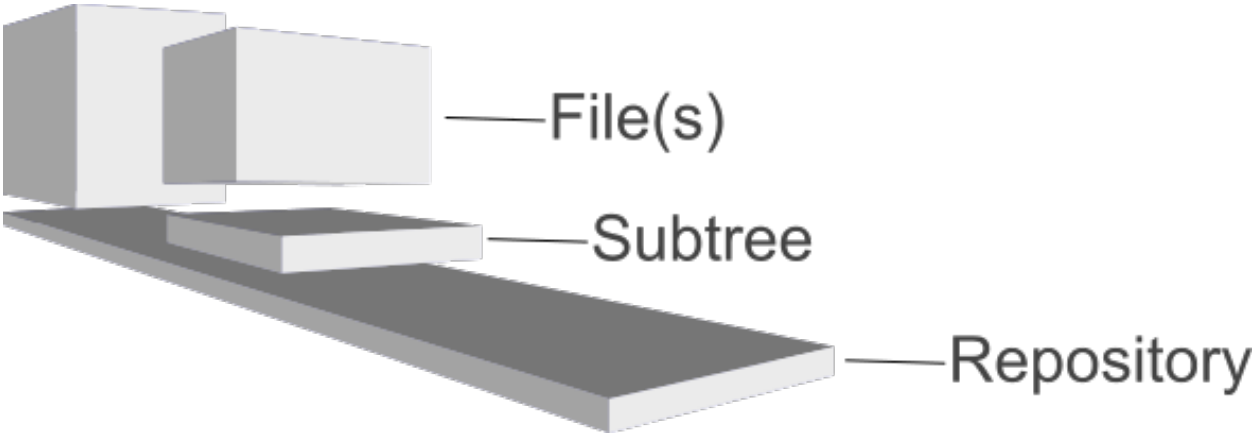
- flow: flow layer

Their relation to one another is depicted below:



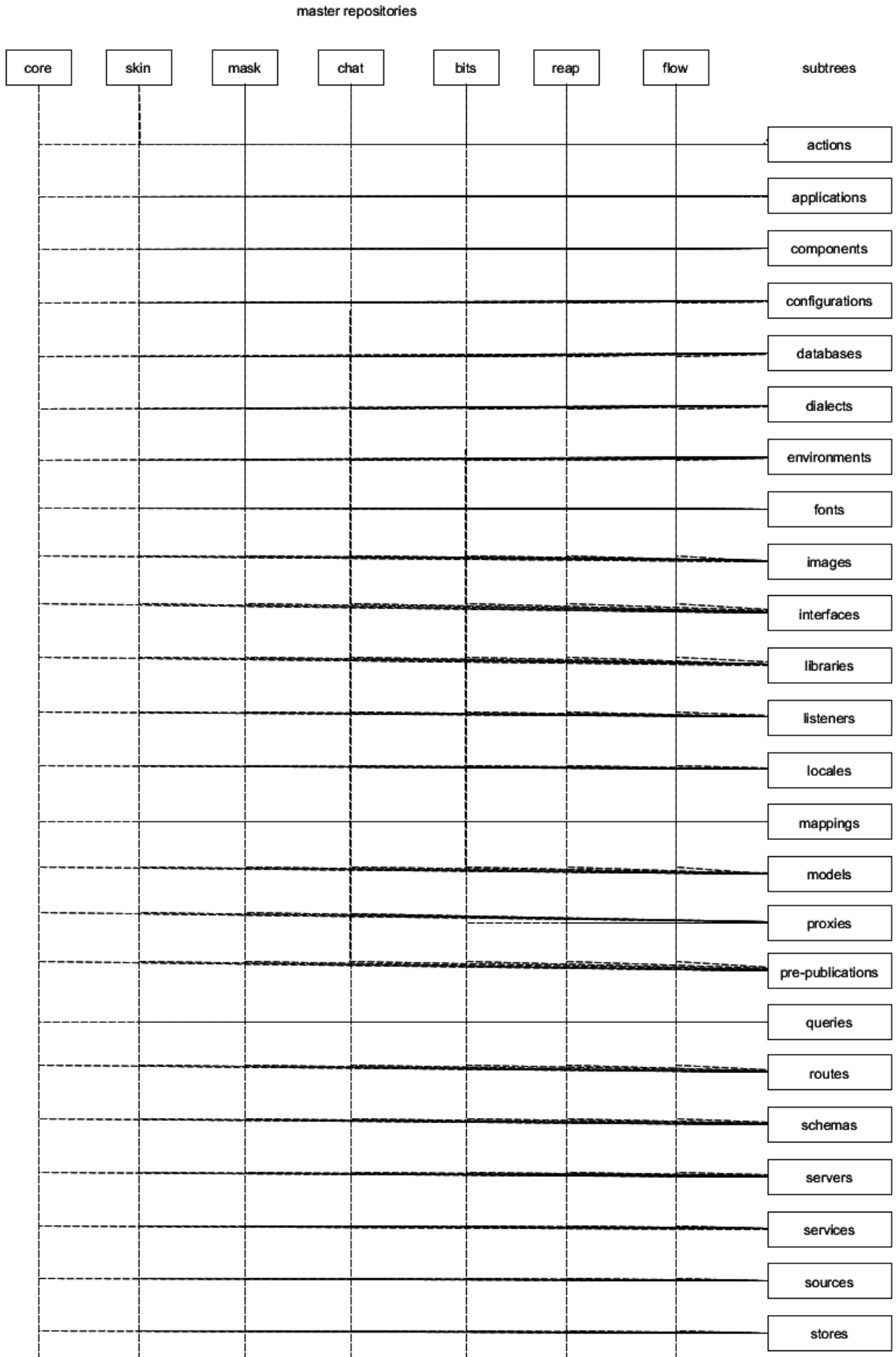
System Overview

Modularity as a pre-requisite for maintainability and scalability is obtained through the stacking of subtrees onto repositories. Each subtree is re-used by several repositories in order to prevent duplication of code.



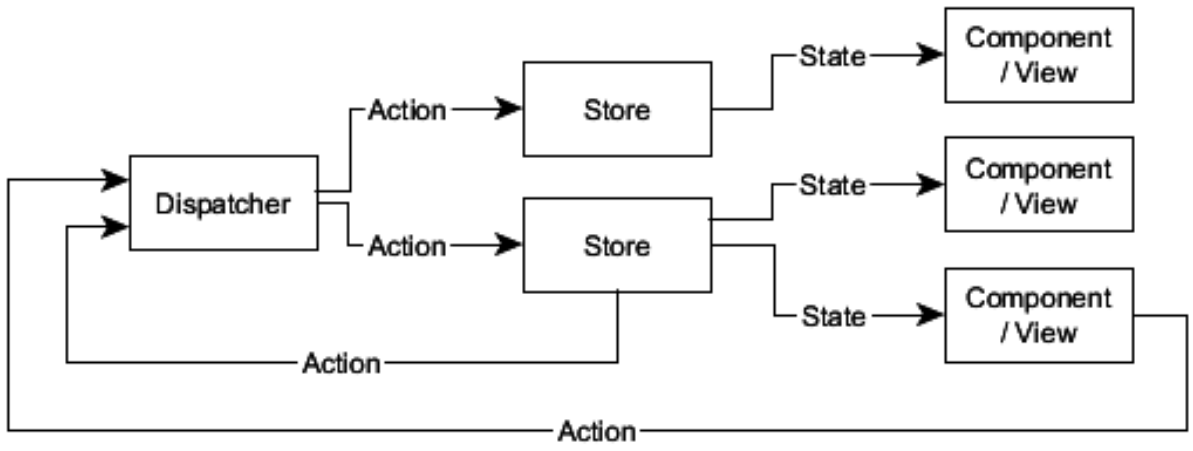
Modularity

On Github, the master repositories share centrally managed code kept as subtrees as follows:


master repositories
core
skin
mask
chat
bits
reap
flow
subtrees
actions
applications
components
configurations
databases
dialects
environments
fonts
images
interfaces
libraries
listeners
locales
mappings
models
proxies
pre-publications
queries
routes
schemas
servers
services
sources
stores

Application

Each application is an instance of a collection of code, following the Flux pattern (https://facebook.github.io/flux/doc
This pattern if prefered over the Model-View-Controller (MVC) pattern at this level of the overall architecture, because Flux has a much more defined structure of events (their results being actions).



Flux Pattern

An *action* is effectively a result of an event, that changes the system.

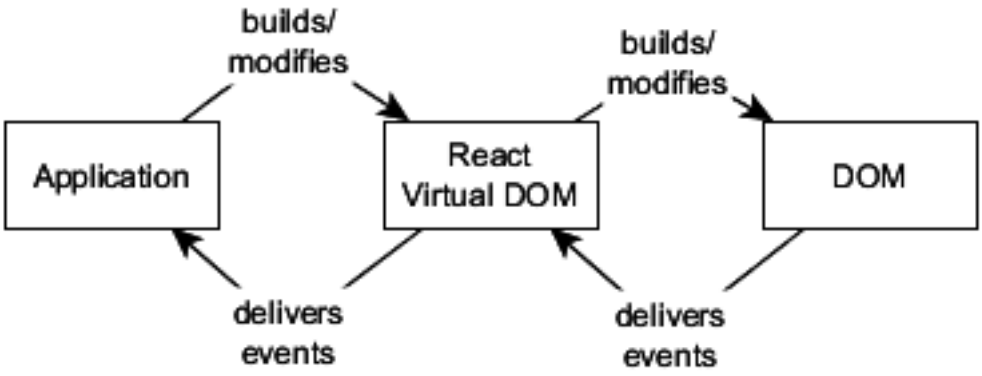
A *store* contains the business logic. It listens to actions, when it receives one, it does something based on the action and updates its state appropriately.

A *component* is a view. The view has a set of properties (passed in values) and/or state (the state is obtained from the store's state). For a given set of properties and state, you always get the same layout. The components listen for updates to the state in the stores and update appropriately.

We also have a *dispatcher*. The dispatcher dispatches actions to interested stores. Only one action can be processed at any one time. If a new action comes in, then the dispatcher queues it.

Actions are always **synchronous** – if changes would happen due to external stimuli then these will be new actions. For example, this prevents actions from blocking other actions whilst waiting for a response from the server.

The Component (i.e. View) in the Flux pattern uses the React pattern (https://en.wikipedia.org/wiki/React_-(JavaScript_library)).



React Pattern

React provides the component structure, it has defined ways of tracking state and properties, and the re-rendering on state change gives much automation. Since it encourages the separation of immutable properties, a whole class of inadvertent errors is eliminated.

There's also many advantages with debugging – we have a flag that lets us watch all the actions going through the system, so its much easier to track what events are taking place and the data passed with them. This combined with the fact that actions have limited scope, helps with debugging the data flows.

The entire point of React is to do **updates** efficiently (by using components in a *virtual* DOM, which is synchronized with the DOM of the views). React models its state as a state property of the component.

(see also http://blog.reverberate.org/2014/02/react-demystified.html)

UMD (Universal Module Definition)

 [1]

This repository formalizes the design and implementation of the Universal Module Definition (UMD) API for JavaScript modules. These are modules which are capable of working everywhere, be it in the client, on the server or elsewhere.

It is the intention to make all modules adhere to the UMD standard, for maximum compatibility.

Unit Testing

- Components/Views are tested by setting up their state and/or properties and ensuring the correct elements are displayed.

[1] https://travis-ci.org/umdjs/umd

- Stores are tested by setting an initial state, sending an action, and checking the resultant state.

(see also https://blog.mozilla.org/standard8/2015/02/09/firefox-hello-desktop-behind-the-scenes-flux-and-react/)

Design

Human Psychology

Basing the design of the application on human psychology secures suitable interaction with its user interface. (see also the course 'Build Persuasive Products' at https://www.udemy.com/designing-engaging-user-experiences/).

User Interface

The application is designed along the following design guidelines:

- Card Pattern (supported by Twitter's Bootstrap version 4, see http://getbootstrap.com), as highlighted here at http://blog.juntoo.co/card-ui-next-big-interaction-paradigm/

One of the most important aspects of cards is that they are incredibly malleable. They can be tweaked and interacted with in a multitude of ways. Just think about what you can do with an actual physical card. A physical card can be:

stacked expanded folded turned over spread out etc.

Digital cards can do all that, and more. Inside a card we can embed:

video gifs coupons photos payments music etc.

The possibilities are endless…

Going forward, the majority of mobile ui designs will be based on the card ui paradigm. The next logical step is marketing professionals and ad agencies starting to embrace cards. The larger platforms are already embracing it (look at Twitter Cards or Pinterest Promoted Pins).

The card ui is set to be the next creative canvas for online content and will consequently also be the next big ad unit.

- Flat Design (more specific Google's Material Design, see https://www.google.com/design/spec/material-design/)

… more to follow

Chapter 4

Execution

Execution enables improvements to Architecture.