Tacit

A Modular System for Serving Applications Online

Willem van Heemstra

Tacit

A Modular System for Serving Applications Online

Willem van Heemstra

Contents

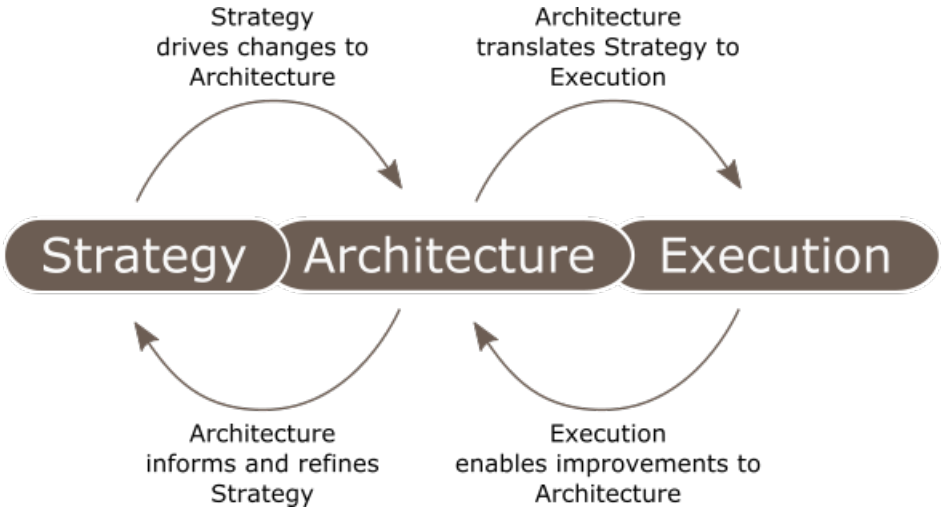
Chapter 1

The 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

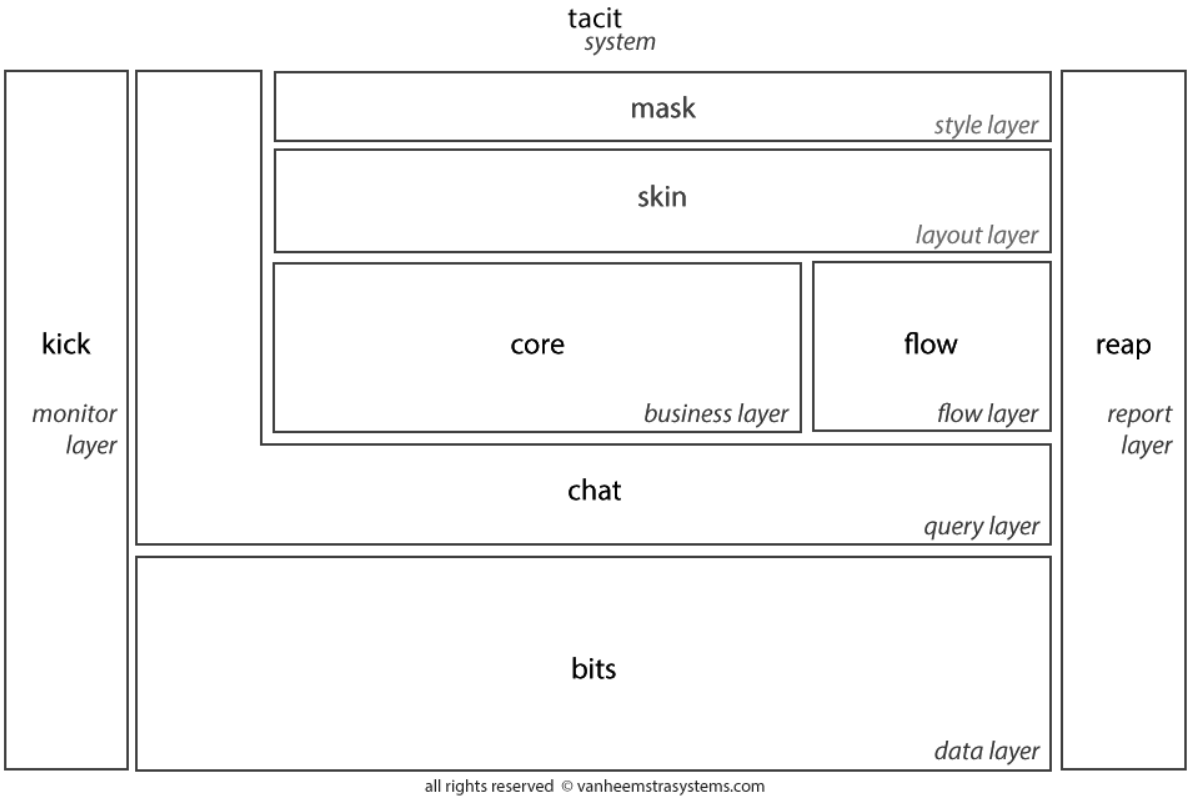
Architecture translates Strategy to Execution. Architecture informs and refines Strategy.

System

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

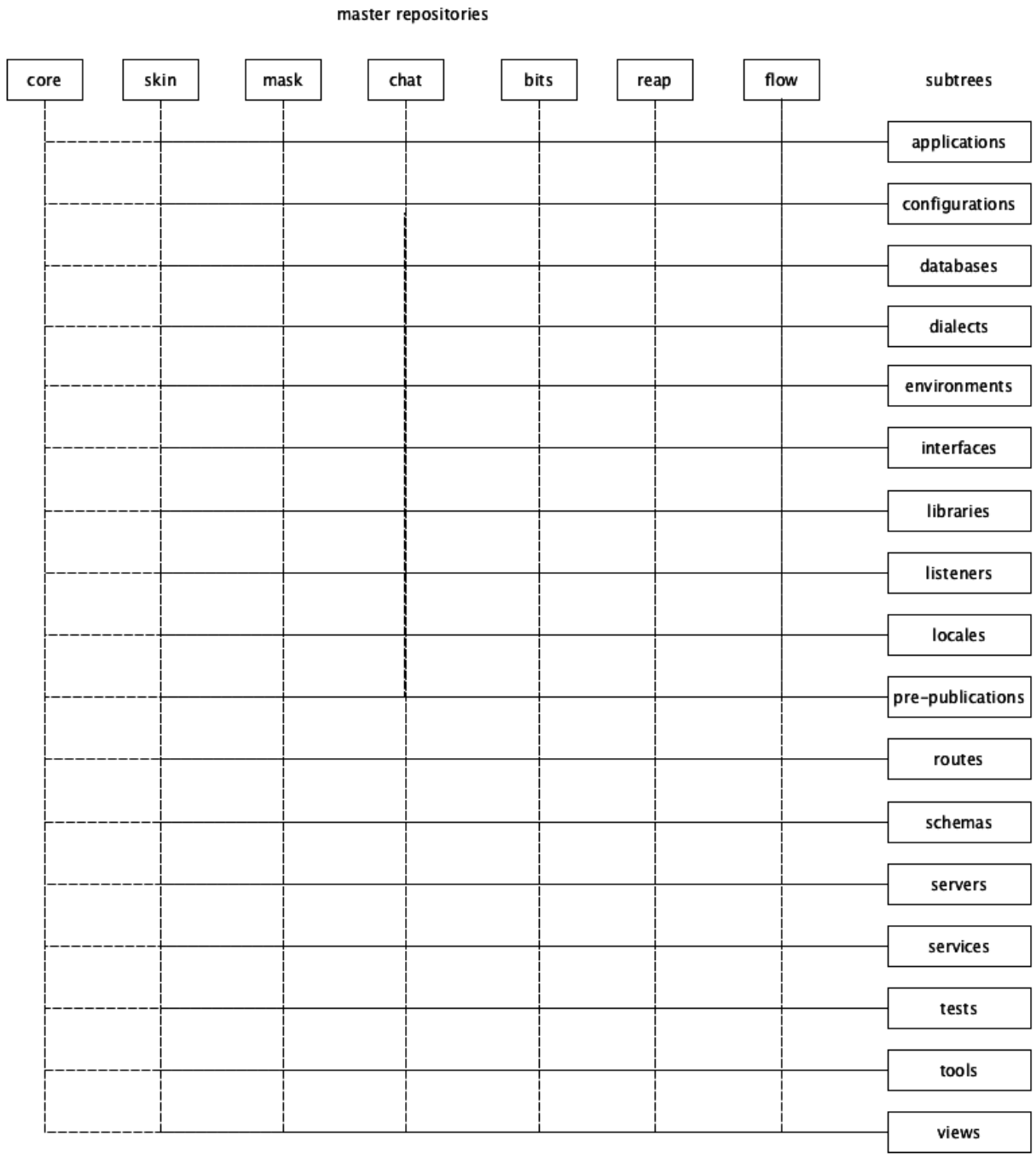
- core: business layer
- skin: layout layer
- mask: style layer
- chat: query layer
- bits: data layer
- kick: monitor layer
- reap: report layer
- flow: flow layer

Their relation to one another is depicted below:



System Overview

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

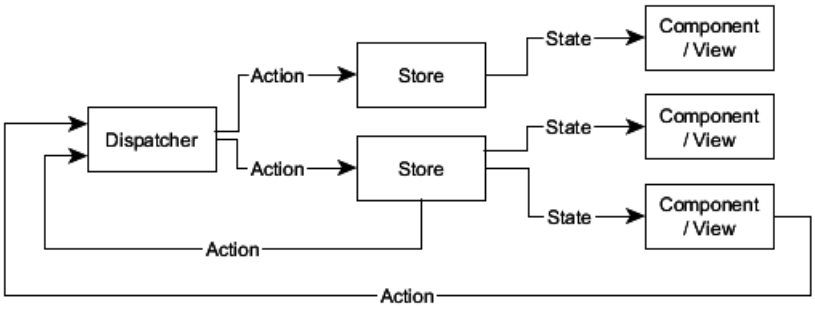


Repository Hierarchy

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.

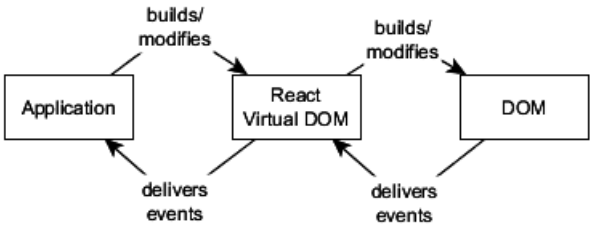
A *store* contains the business logic. It listens to actions, when it receives one, it does something based on the action and updates its state appropriately.

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)

Unit Testing

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

Chapter 4

Execution

Execution enables improvements to Architecture.