The fog of SPA

Seven hard-won lessons for developing SPAs at scale

Single Page Web Applications
http://manning.com/mikowski

The fog of war

What "**the fog of war**" means is: war is so complex it's beyond the ability of the human mind to comprehend all the variables. Our judgment, our understanding, are not adequate. **And we kill people unnecessarily.**

— Robert Strange McNamara
"The Fog of War," 2003

The fog of SPA

What "**the fog of SPA**" means is: we can unwittlingly make SPA development so complex it's beyond the ability of the human mind to comprehend all the variables. Our judgment, our understanding, are not adequate. **And we kill projects unnecessarily.**

— me

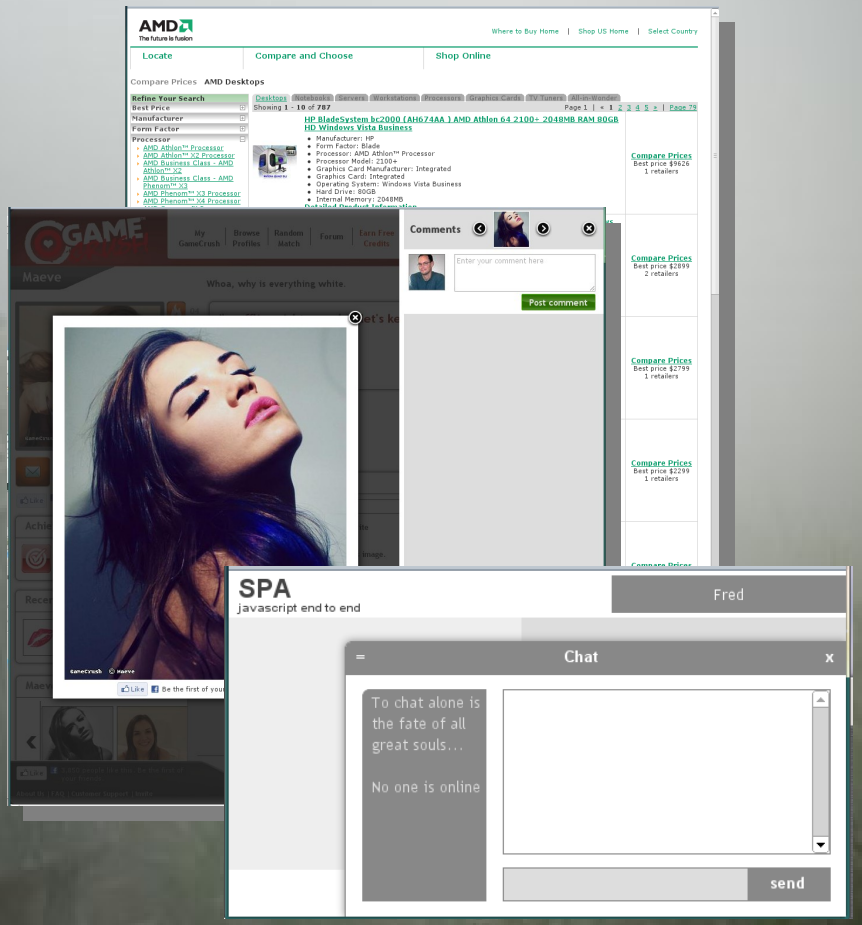
My goal today

break through the fog

- Minimize complexity
- Maximize effectiveness

A bit about SPAs

- SPAs are web applications that don't reload during a user session
- Increasingly popular, as users are now expecting desktop-like behavior
- Flash games, Java office suites, Javascript calculators have been around for a long time
- We are considering **Native** JavaScript SPAs here

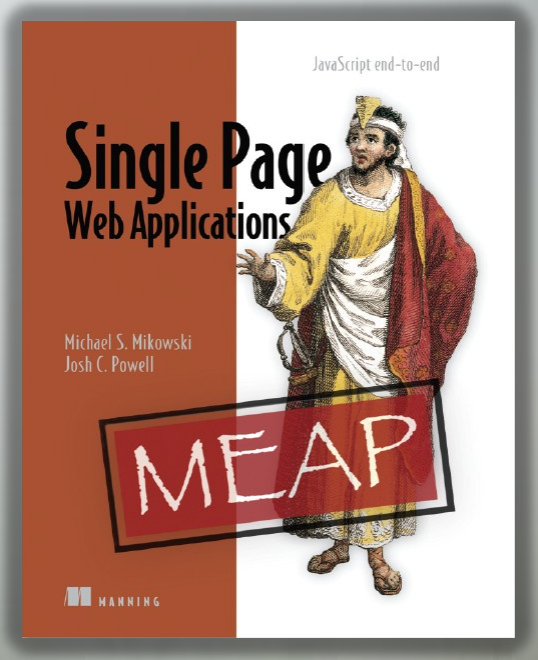


A bit more about SPAs

- Business logic: **Server** → **Browser**
- JS coding at a scale an order of magnitude greater than a traditional websites (100k lines)
- One SPA may require many developers
- **Conventions and discipline previously reserved for server-side development becomes a must for working at this scale**

A bit about me

- **Single Page Web Applications** – JavaScript end-to-end
- **UI Architect** at SnapLogic, team of 5
- Developer on **5 production SPAs** since 2006, Architect on all but one
- Previous back-end development manager on **HP+HA mod_perl clusters** (~2B web transactions per week)
- First SPA: European and US AMD "**wheretobuy.amd.com**," rel. 2007



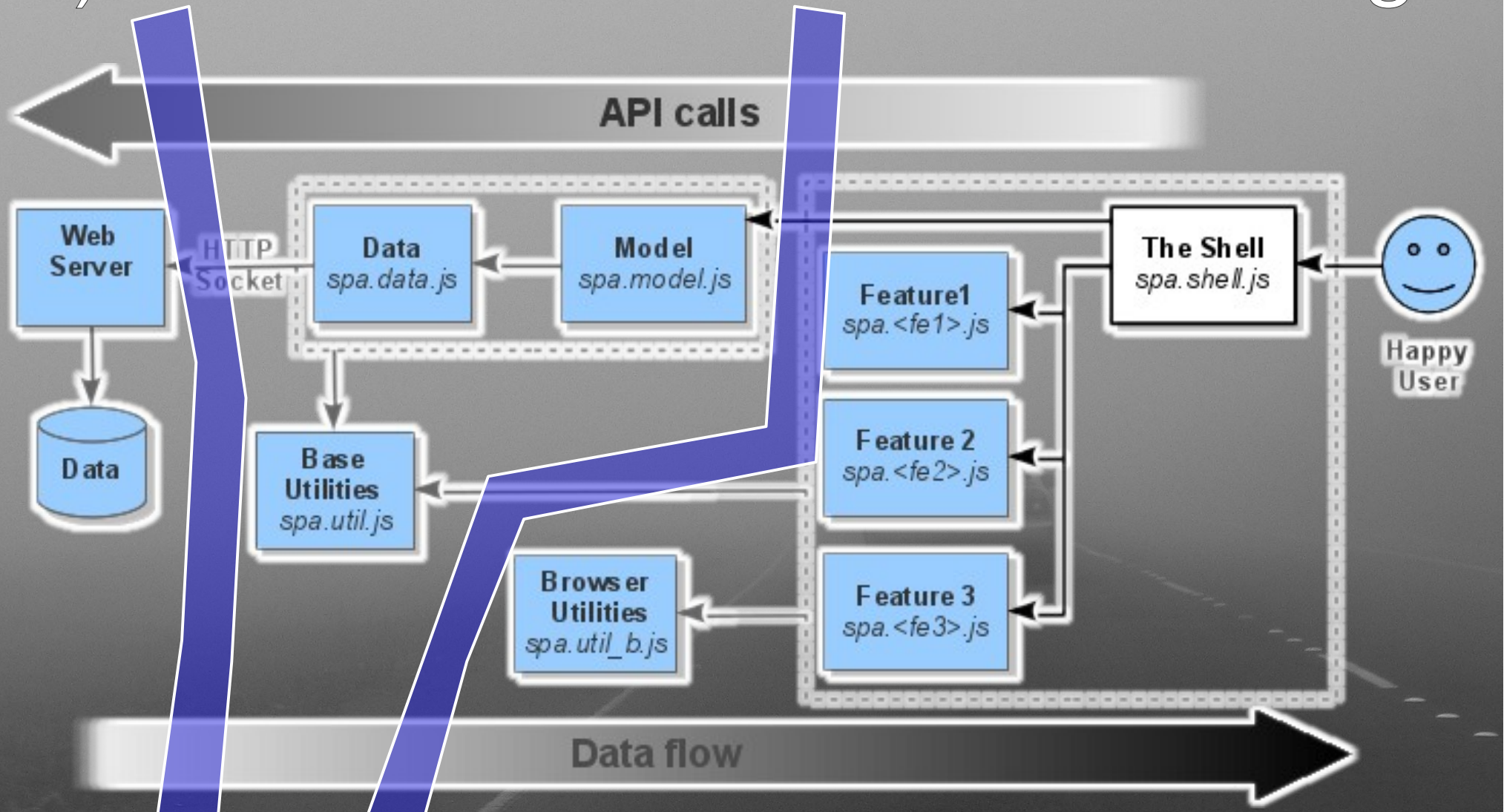
http://manning.com/mikowski

Also see LinkedIn

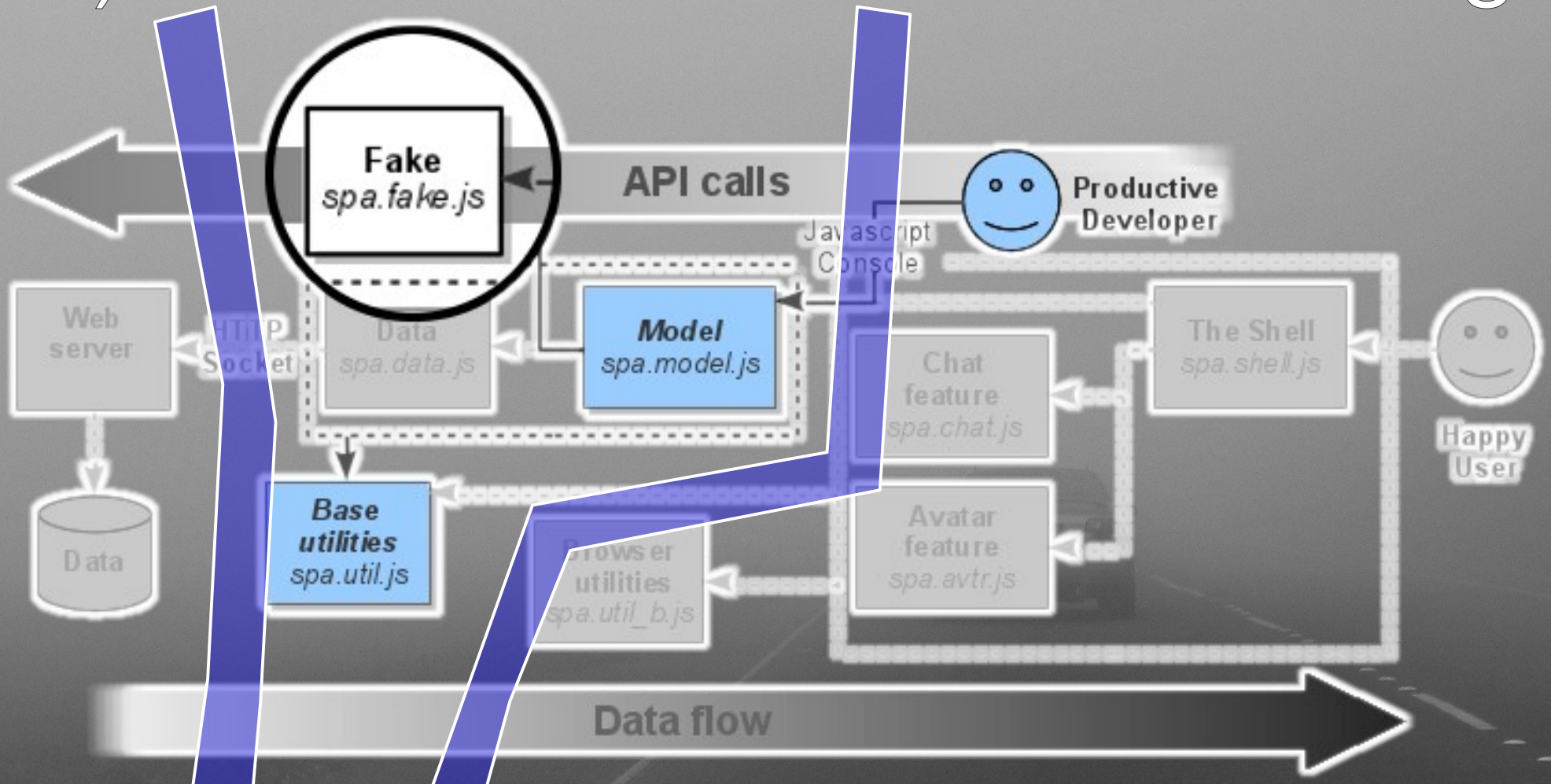
Seven lessons of SPA dev

1) Architect for workflow and testing
2) Design third-party style modules
3) Start at the front
4) Plan for many SPAs
5) Use a common language
6) Test the client back-end
7) Avoid shiny objects

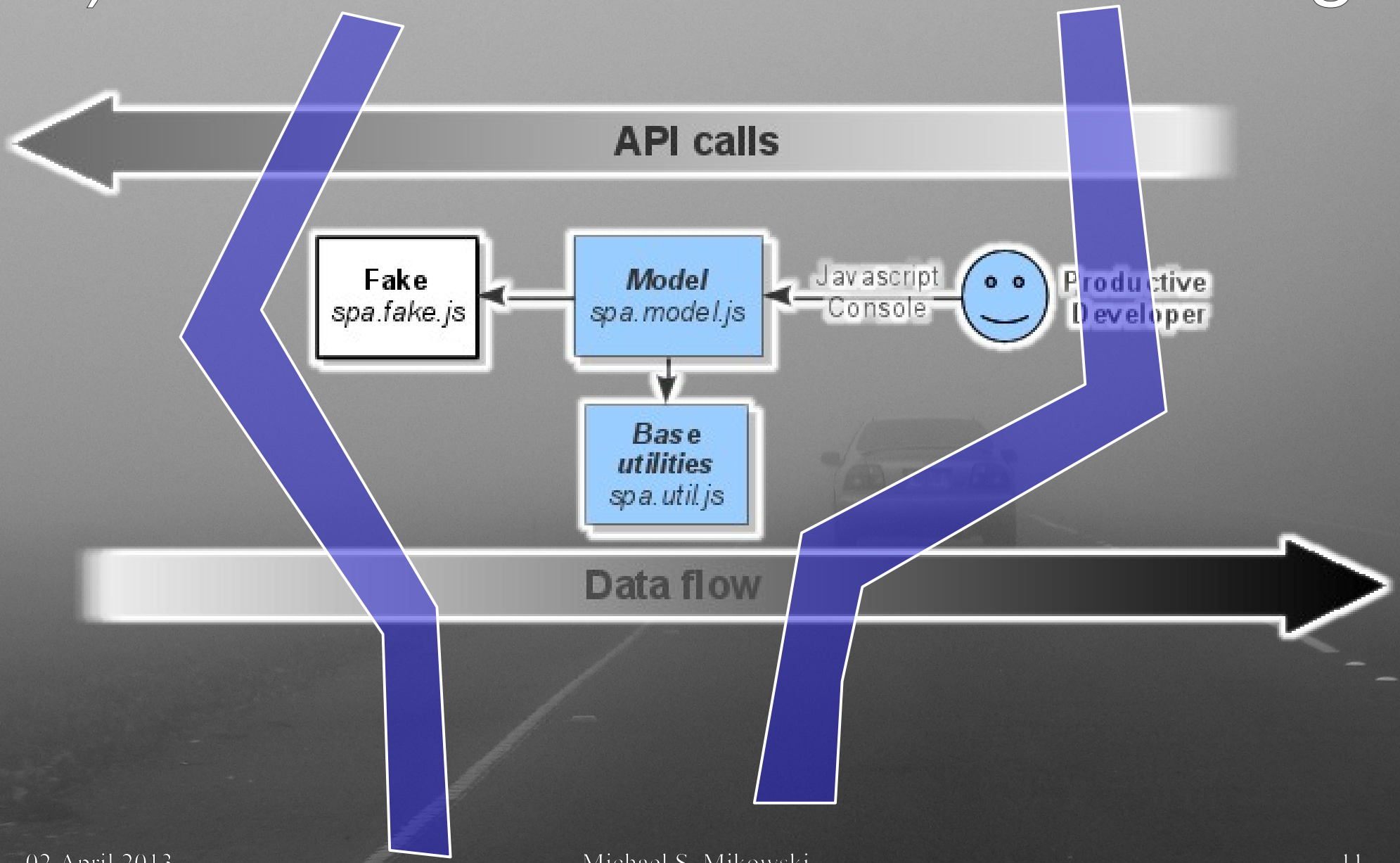
1) Architect for workflow + testing



1) Architect for workflow + testing



1) Architect for workflow + testing



2) Design third-party style modules

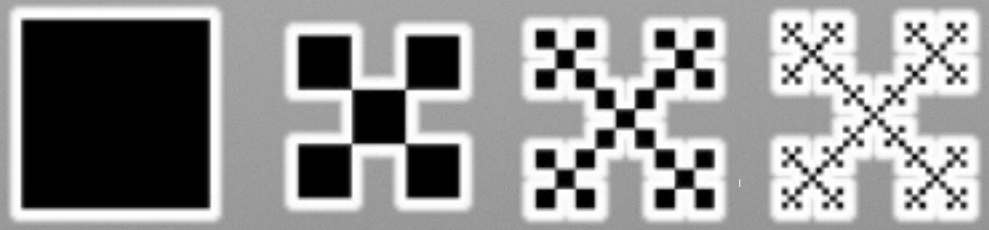
- **Add high-quality features at low cost**
- Examples:
  - **comments** (Disquss or Livefyre)
  - **advertising** (DoubleClick or ValueClick)
  - **analytics** (Google or Overture)
  - **sharing** (AddThis or ShareThis)
  - **social services** (Facebook “Like” or Google “+1”)

2) Design third-party style modules

- How popular are third-party modules?
- Techcrunch – a snapshot
  - **At least 16** web services
  - **5** analytic services alone
  - A whopping **53** script tags

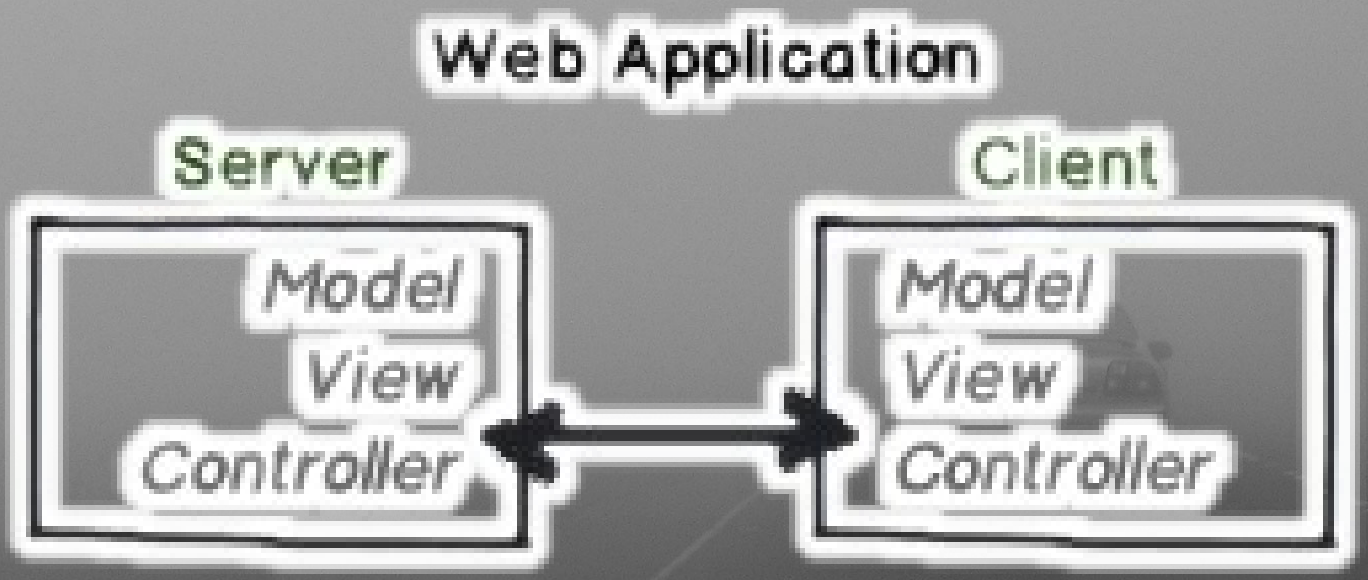
2) Design third-party style modules

Fractal MVC



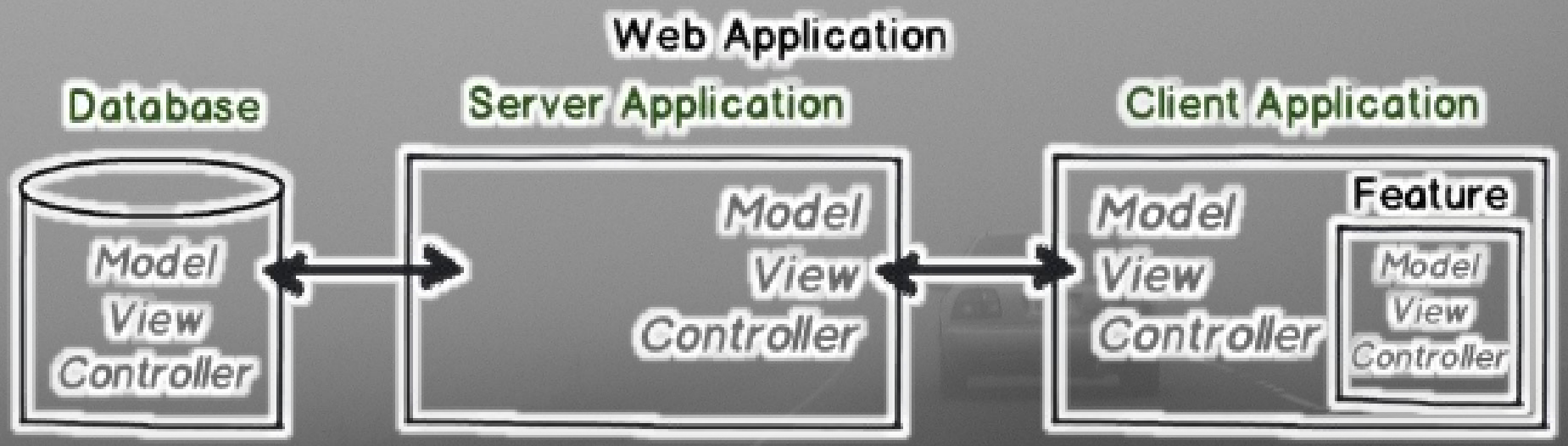
2) Design third-party style modules

Fractal MVC



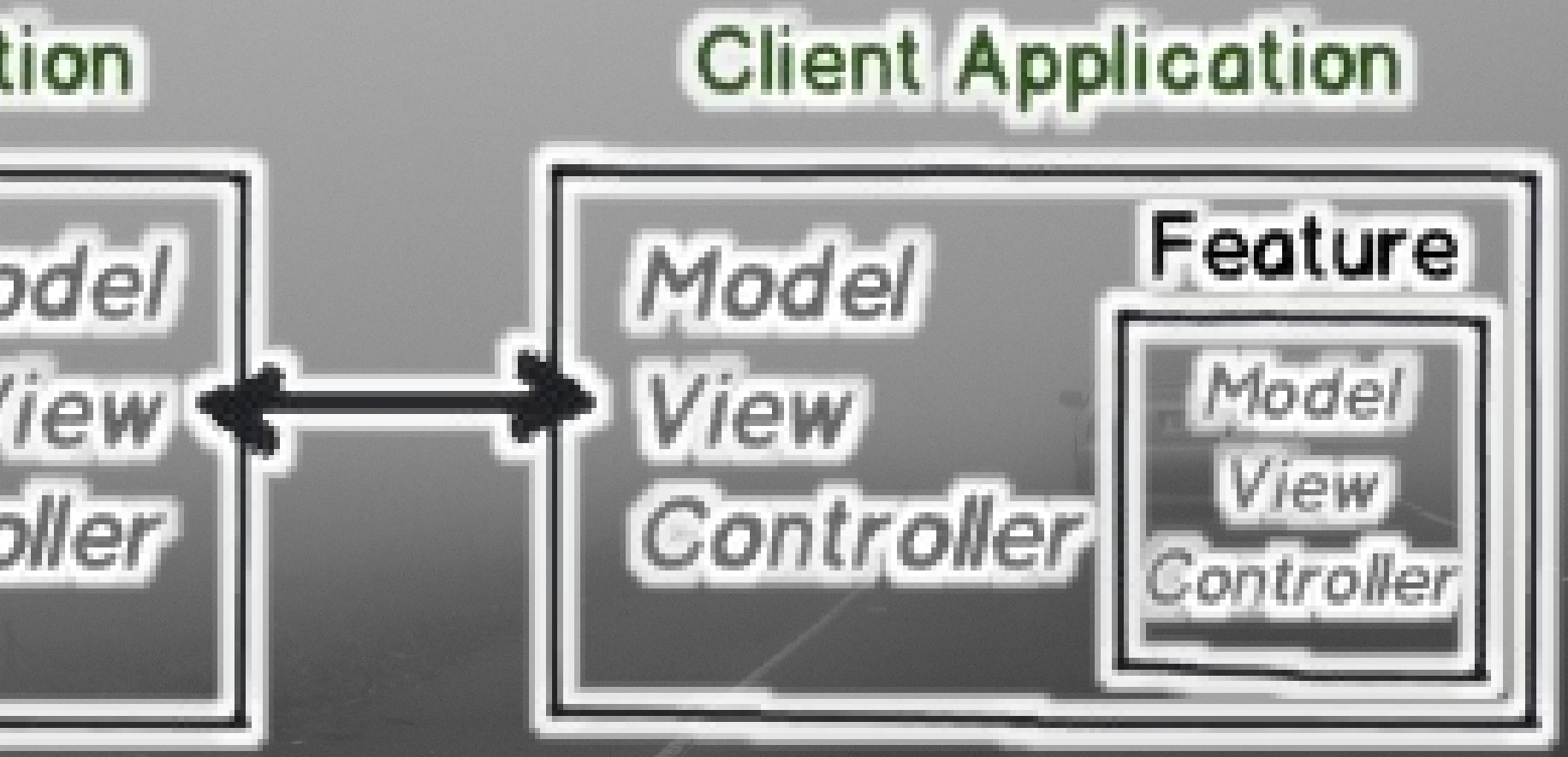
2) Design third-party style modules

Fractal MVC



2) Design third-party style modules

Fractal MVC

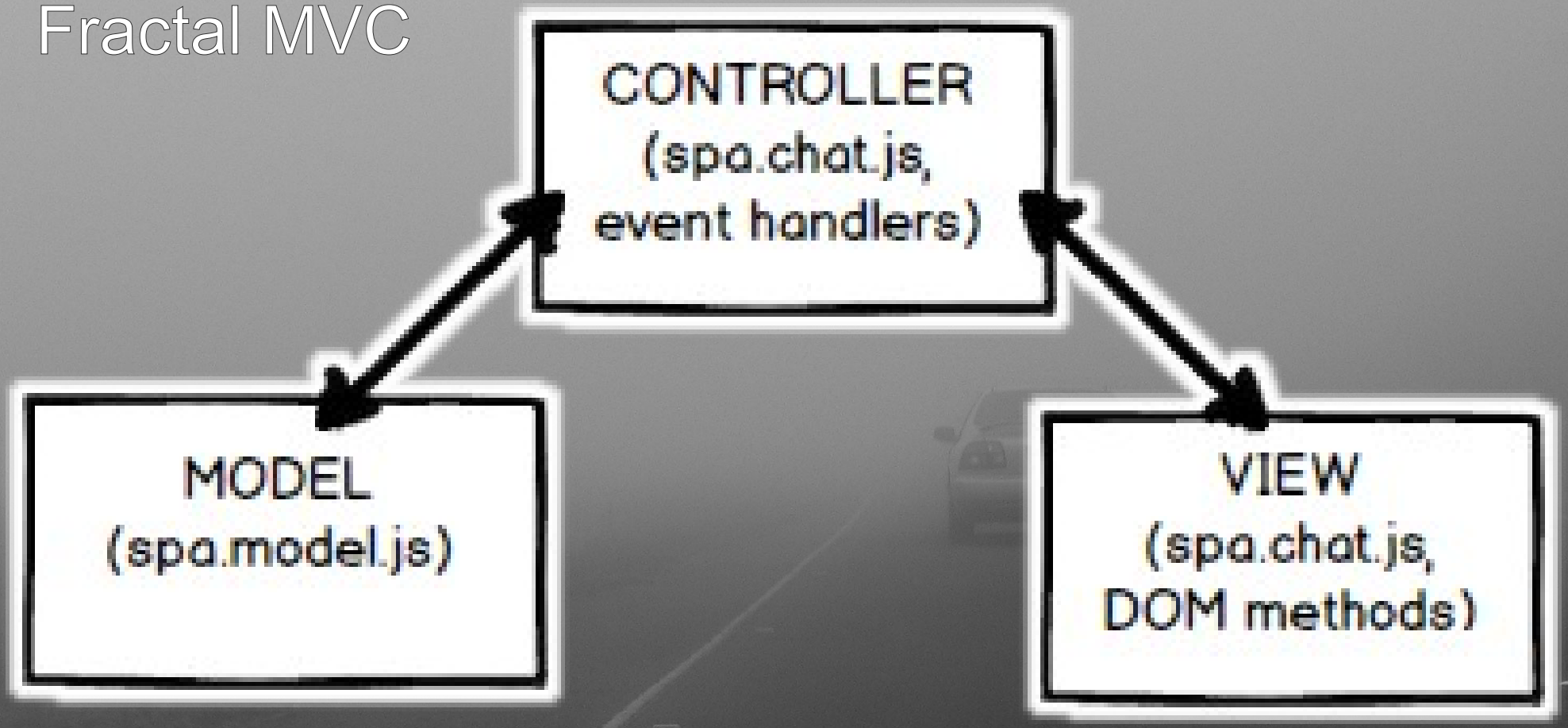


2) Design third-party style modules

- A feature module provides a **well-defined and scoped** capability to the applications
- No cross talk between modules
- **Benefit**: Third-party and your modules are interchangable
- **Benefit:** Reusable across projects
- **Benefit:** Focus on core and use third-party modules, replace as resources allow
- **Benefit:** A great way to divide work

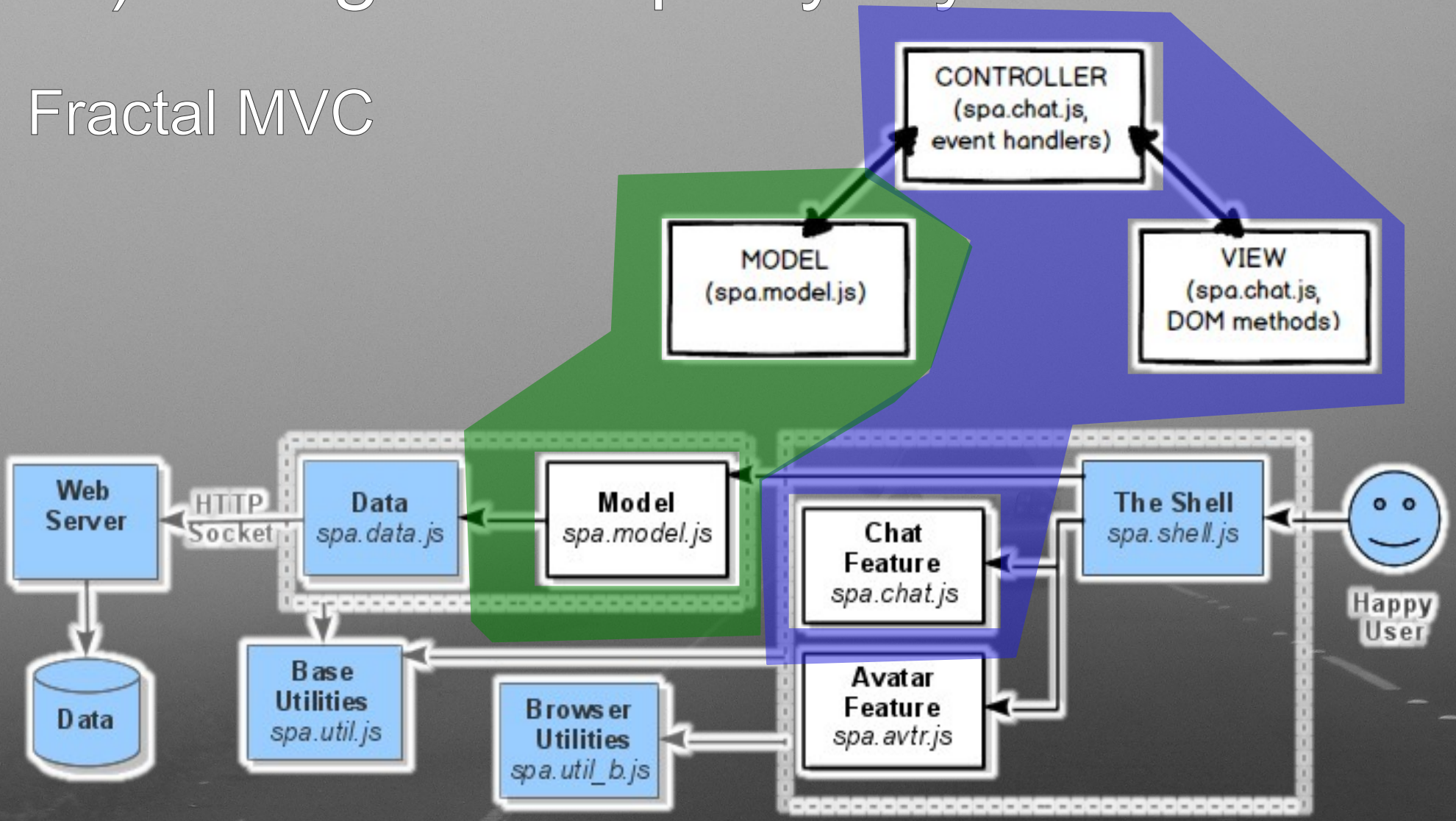
2) Design third-party style modules

Fractal MVC



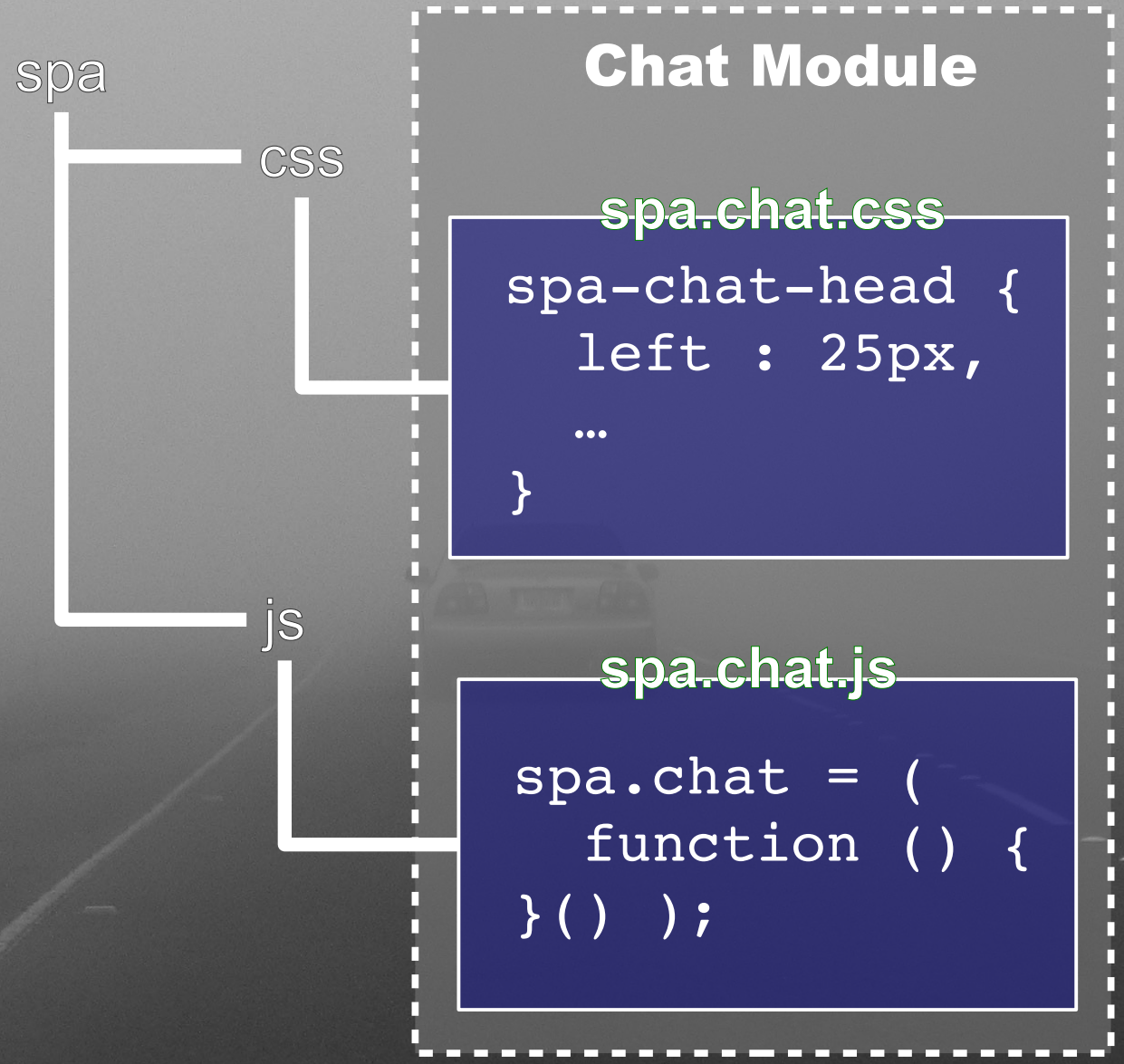
2) Design third-party style modules

Fractal MVC



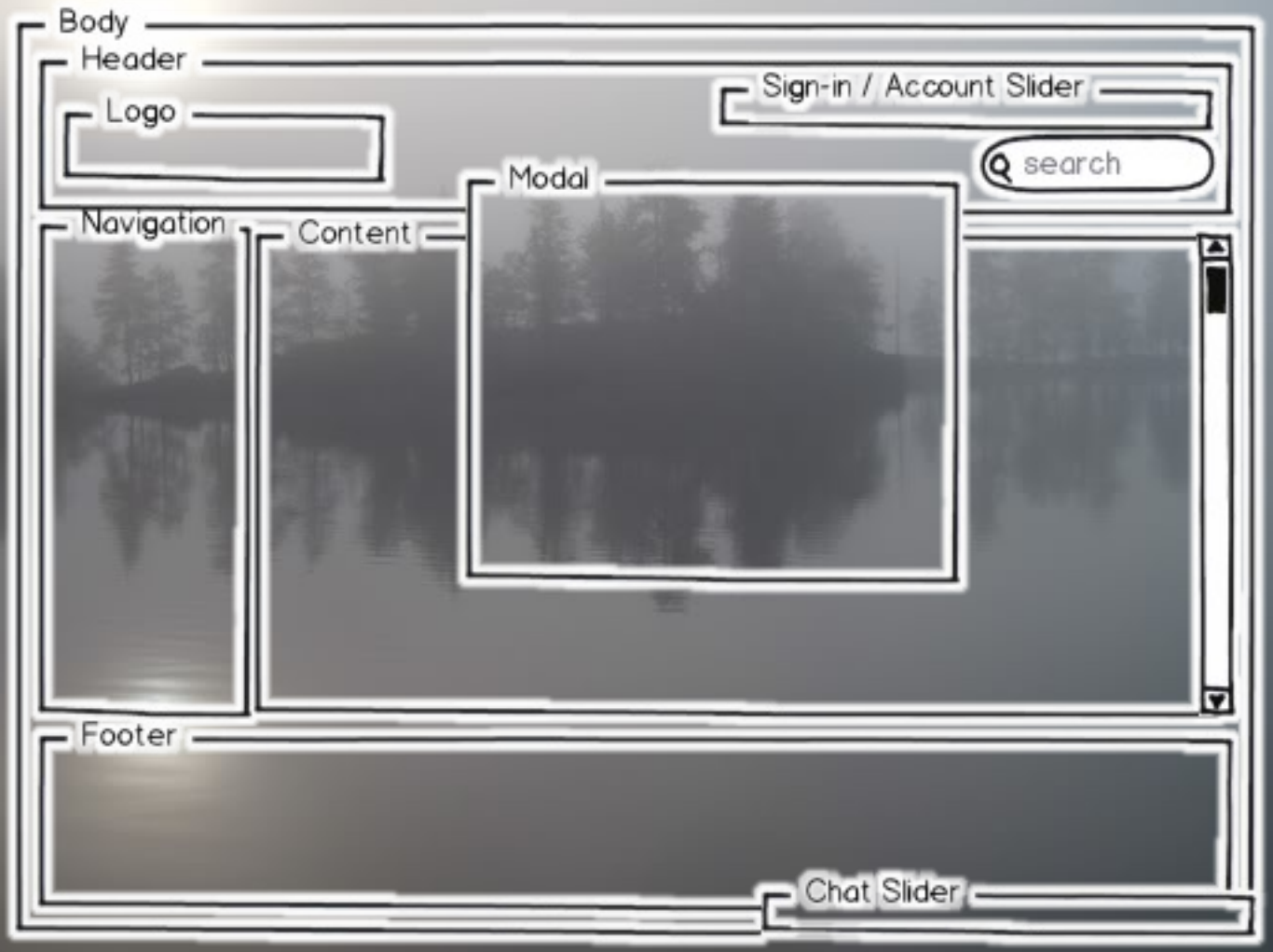
2) Design third-party style modules

- Namespace
  - JavaScript
  - CSS
  - Files
- Our spa should 'use' only two global variables 1) our root namespace, and 2) our common namespace

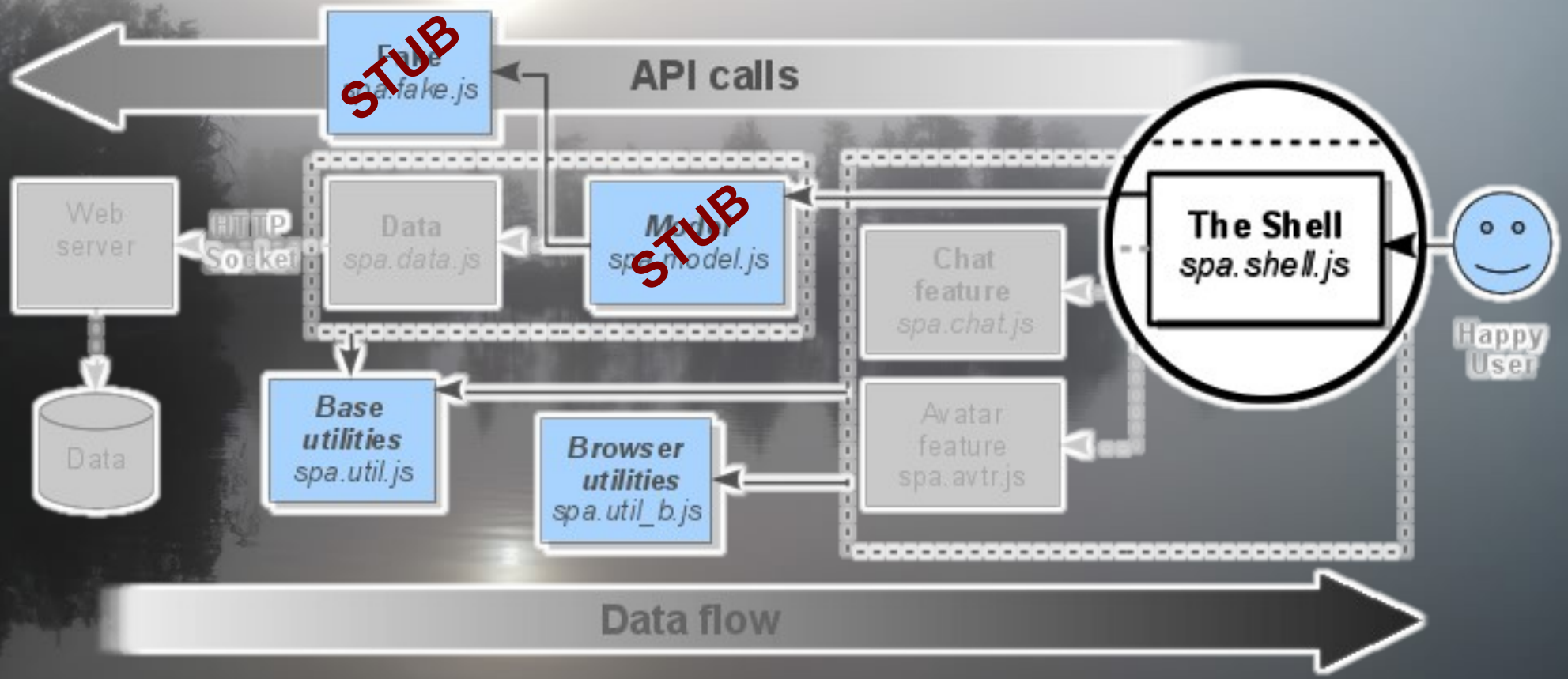


3) Start at the front

Design layout first



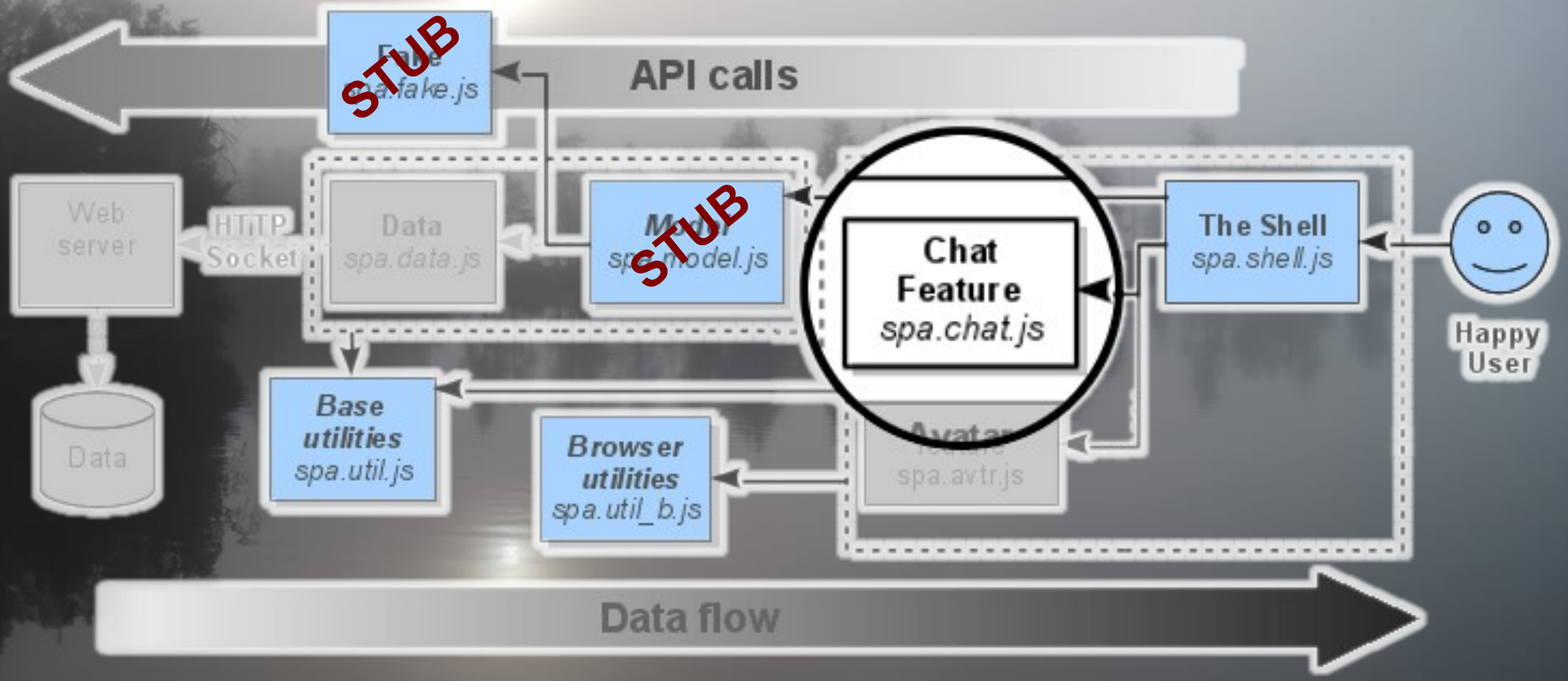
3) Start at the front



3) Start at the front

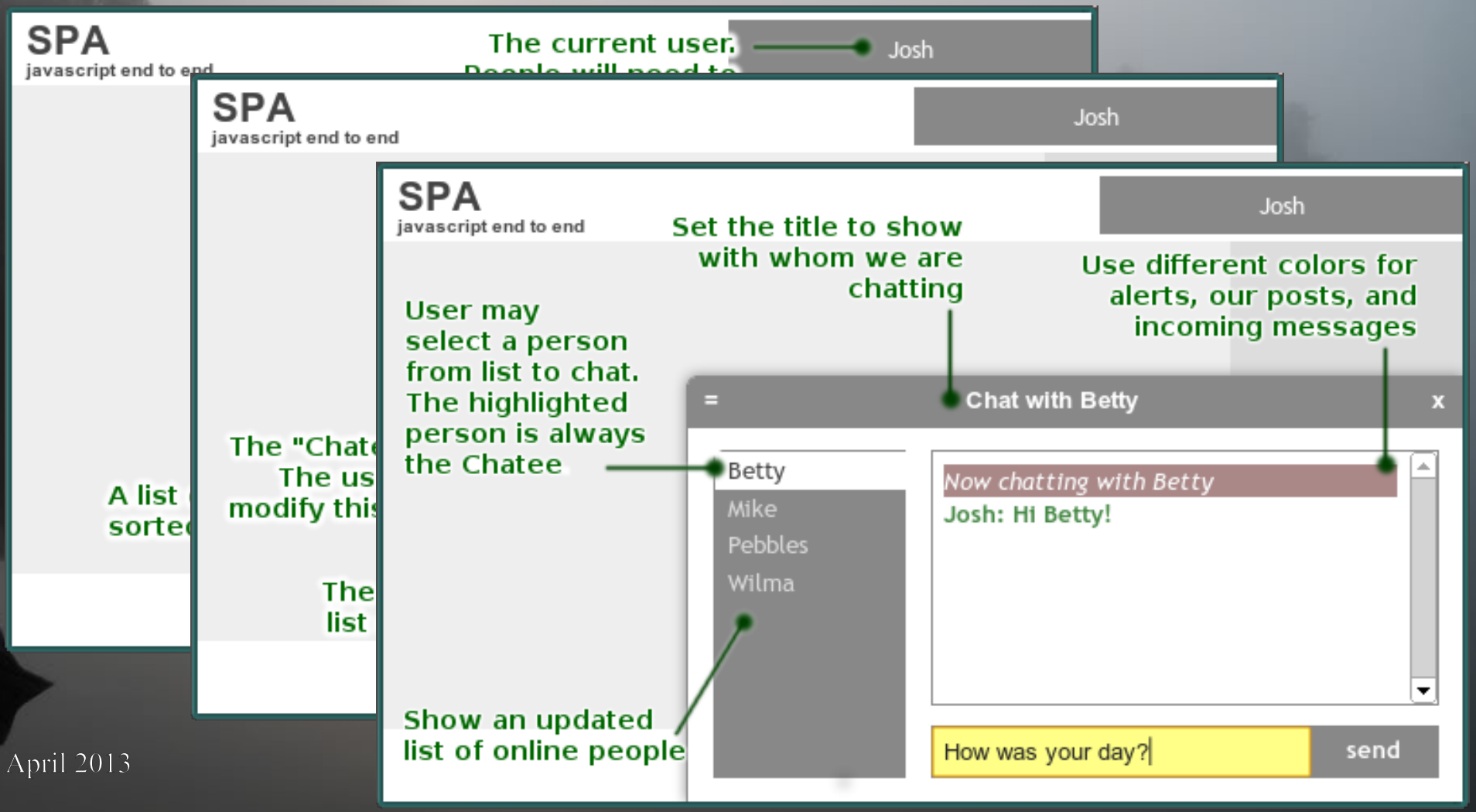
- Now start the Shell:
- “Master controller”
- Coordinates feature modules
- Handles browse-wide interfaces like...
  - URL anchor (“hash fragment”)
  - Feature containers
  - Cookies

3) Start at the front



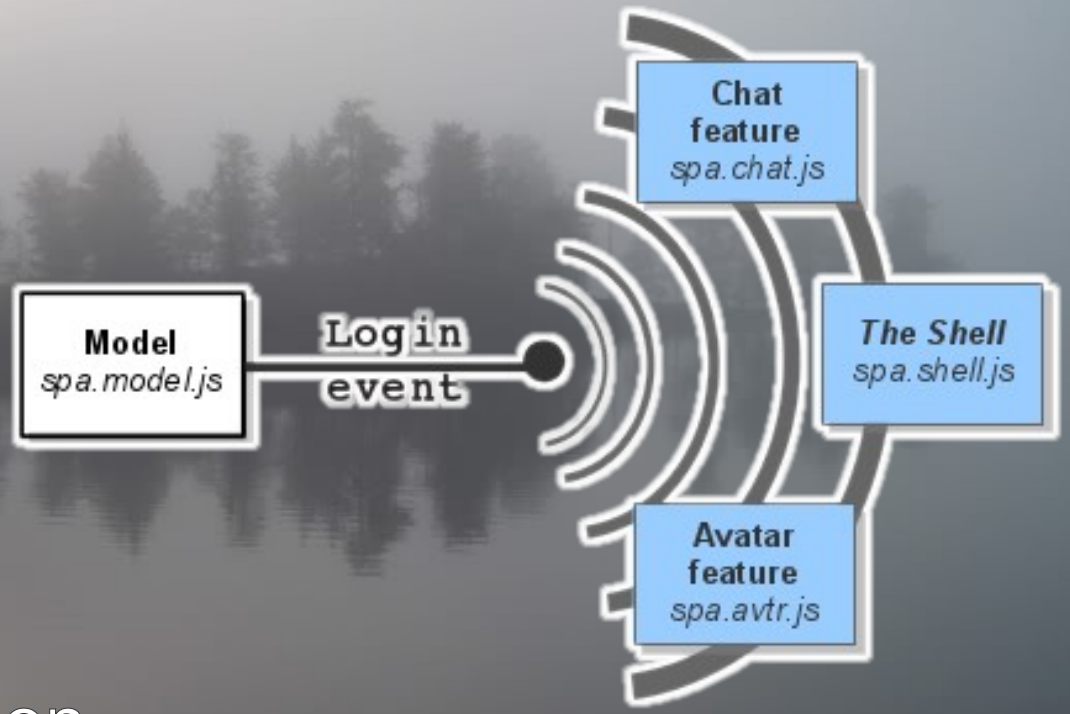
3) Start at the front

Now design your **feature module**



3) Start at the front

- Write the API first
- No cross-talk between modules
- Use events to minimize callbacks
- Use promises to minimize callbacks, dependencies
- Use argument checks on all major APIs, prefer named arguments



3) Start at the front

- API Doc Format
- Overview
- Methods
  - Synchronous calls
  - Minimize callbacks
- Events
  - Asynchronous data
  - Often results from external input
  - Include data with events

API

4) Plan for many SPAs

- Most “web application” sites are actually a collection of a 2 or more SPAs
- Example: AMD, one SPA per computer type
- Decide where to split the capabilities
- Plan on common capabilities and look and feel per site

4) Plan for many SPAs

- spa
  - css
    - common
    - admin
    - designer
    - commerce
  - js
    - common
    - admin
    - designer
    - commerce
  - img
    - common...

We have selected 'sp' as our “common” namespace

Include common libraries before the SPA specific libraries

- Standard utilities (sp.util)
- Browser utilities (sp.util_b)
- Argument checking (sp.arg)
- Templating (sp.dust)

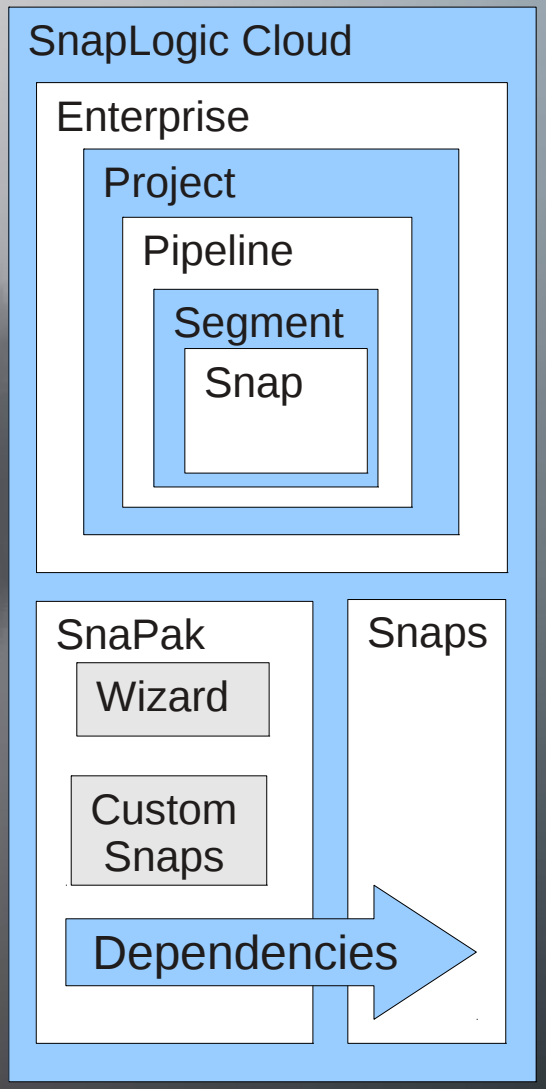
5) Use a common language

- Q. Why is Google developing Dart?
- A. The overhead of Java/JavaScript development is a serious drag on iterations

Multiple languages drain creativity and productivity

5) Use a common language

- Start with product design
- Define terms as early as possible
- It is best if marketing uses the same terms as product planners, developers, and sysops
- Write definitions for each
- Avoids miscommunication throughout the product
- As important as code standards



5) Use a common lanuguage

- Keep the languages **simple and few**
- Use **native** JavaScript and CSS
  - Use of a JS complier and CSS compiler are two additional steps that can go wrong
  - When you debug your application, you see CSS and JavaScript – so you need to know these really well
  - The best way to know CSS and JavaScript really well is to use them everyday
  - **We have found the tradeoffs aren't worth it**
- Do we want to test the application immediately or work through a 2-step compile each time?

5) Use a common language

- Name your variables **to indicate type**, or …
  - Use TypeScript?
  - Use Closure Compiler?
  - Use GWT?
  - Use Dart?
- **All of these** try to solve type discovery problems
- **Most variables never change type**; most are mistakes
- Polymorphic interfaces are almost always a bad idea
- If your function returns a string, always return a string. ***Throw an exception on error – don't return false!***

5) Use a common language

DataType	Indicator	Examples	Notes
Boolean	sw, is, has, do	is_used	***true*** or ***false***
String	name, text, title, type, key, string	user_name	Type and key indicate enum
Integer	int, count, i,j,k, index, length	list_length	Indicates intent; only Firefox uses type-inference
Number	num,n,ratio	scale_ratio	Always signed double fp
Regex	regex	regex_match	Technically an object
Array	list	user_list	Ordered list
Hash (Mapt)	map	user_map	Technically an object
Object	(no indicator)	house_boat	Traditional object with methods
JQuery Obj	$	$tabs	Technically an object
Function	<verb>noun	make_dog	First class artifact
unknown	data	http_data	Unknown or polymorphic

5) Use a common languange

- Compare meaning in conventions ...

```
var make_house
  = curry_build_item({
    item_type : 'house'
  });
```

5) Use a common languange

Versus no convention:

```
// 'creator' is an object constructor we get by
// calling 'maker'. The first positional argmuent
// of 'maker' must be a string, and it directs
// the type of object constructor to be returned.
// 'maker' uses a closure to remember the type
// of object the returned function is to
// meant to create.
//
var creator = maker( 'house' );
```

Updates are a nightmare!

5) Use a common language

- **Share a common IDE**
  - We have found an IDE that is great as a result of a 4-day study
  - Sharing it reduces language barriers
  - Allows us to **share settings** in our GIT repository
  - Hosts a nice test framwork
  - Spend time evaluating the best IDE for your use
- **Use a code review tool**
  - A great way to keep team members in sync
  - Prevent everyone from getting too far “out there”

6) Test the client backend

- Remember an SPA is a **client** not just the UI
- **JSLint** is invaluable in spotting common, stupid errors. Make it part of your commit hook
- **All public APIs** should use common argument checking and named arguments
- The Model and Fake data provide an excellent foundation for **TDD** using known data
- **Regression tests!**

7) Avoid shiny objects

- **Dangling beads** and accidents
- **Candidate example** (avoiding requirements)
- The moral: Minimize tool kits and languages
  - Add toolkits only after **careful consideration.** Can we do this with our existing tools?
  - Can we test without numerous build steps or multiple layers of abstraction?
  - **avoid** unmanagable complexity
- Backbone event example
- Simplify, simplify, simplify.

Seven lessons of SPA dev

1) Architect for workflow and testing
2) Design third-party style modules
3) Start at the front
4) Plan many SPAs
5) Use a common language
6) Test the client back-end
7) Avoid shiny objects

The fog of SPA

Seven hard-won lessons for developing SPAs at scale

Single Page Web Applications
http://manning.com/mikowski