Practical Thin Server Architecture with Dojo

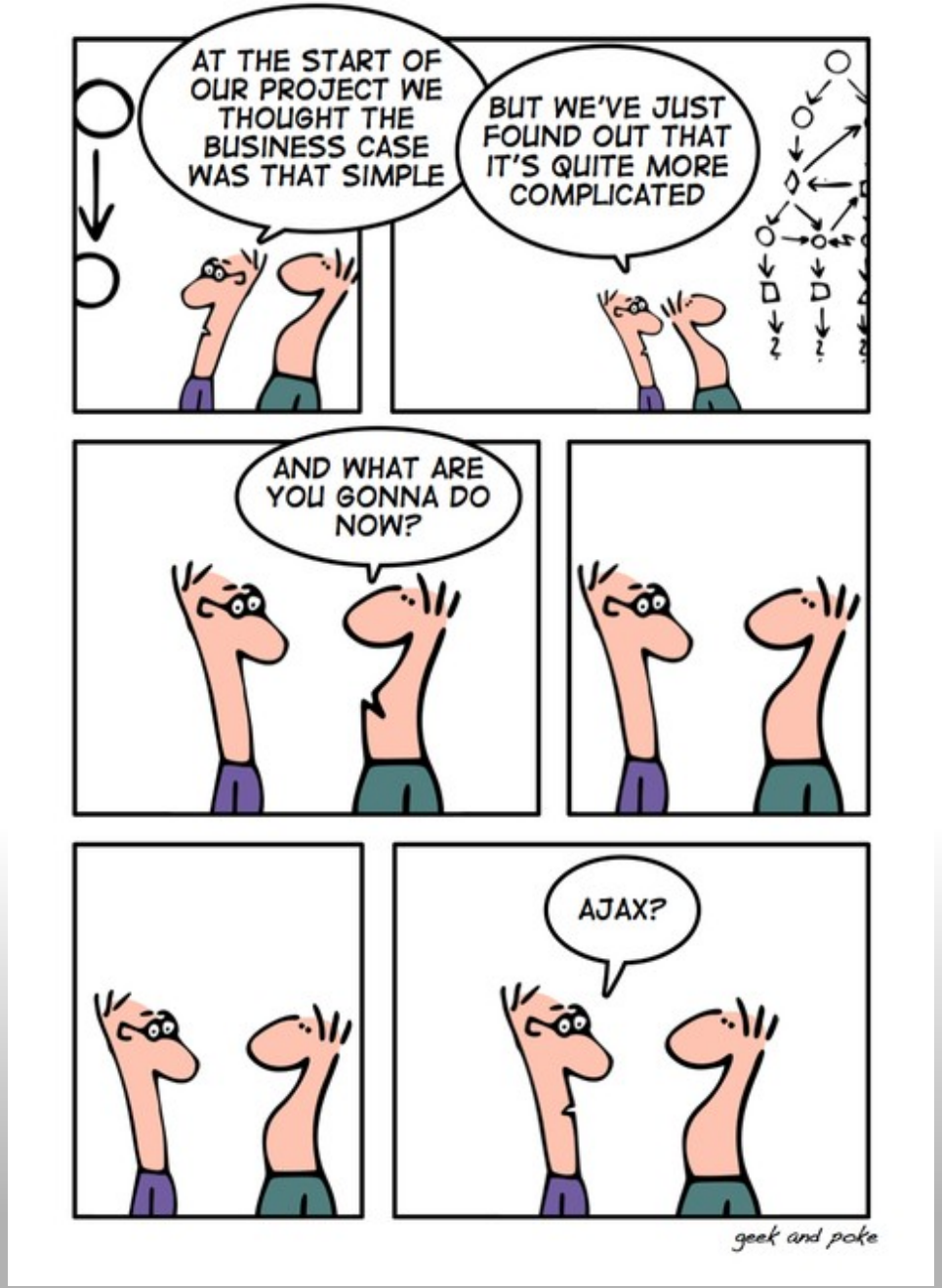
Peter Svensson, **Nethouse AB**
psvensson@gmail.com

But first..

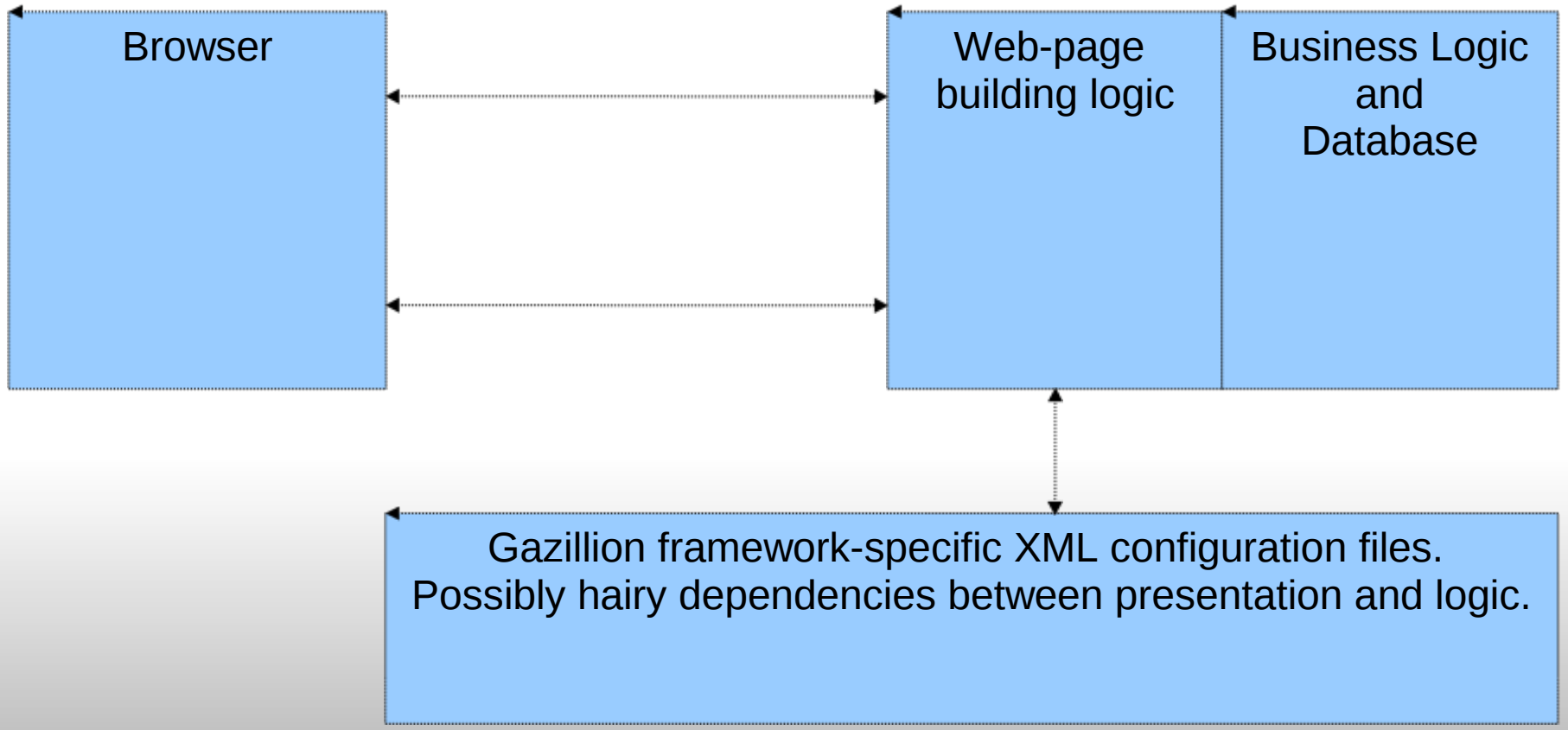
Some theory

What is this all about?

Reducing complexity



Server-Centric Frameworks

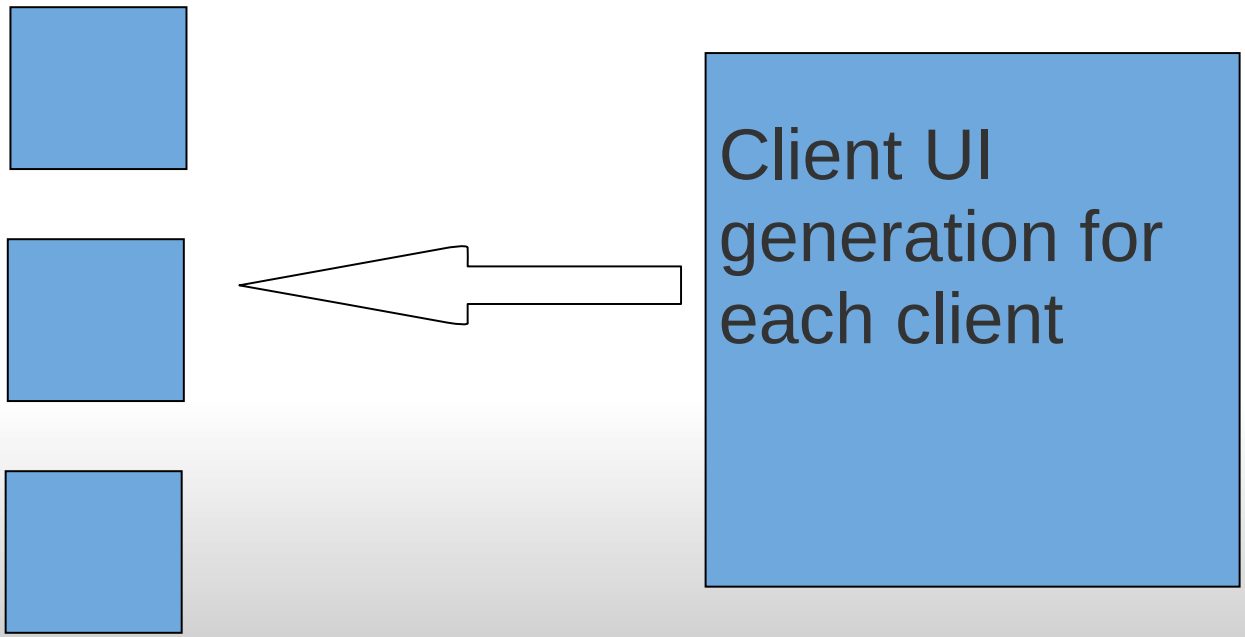


Problems with server-centric frameworks

- Poor distribution of processing
- High user response latency
- Difficult programming model
- Heavy state management on the servers
- Offline Difficulties
- Reduced opportunity for interoperability

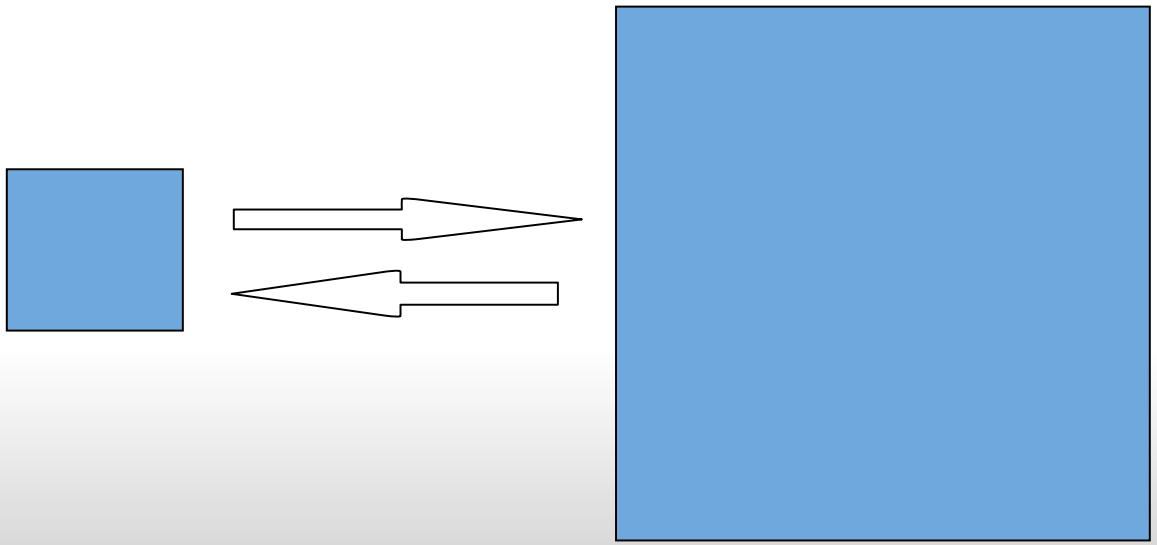
Poor distribution of processing

With a large number of clients, doing all the processing on the server is inefficient.



High user response latency

Traditional web applications are not responsive enough. High quality user interaction is very sensitive to latency, and very fast response is essential.



Difficult programming model

- Client interface is 'canned' in pieces on the server.
- JavaScript logic is hidden and/or mapped to server-side logic.
- C/S interaction is made into a faux single machine model.
- What you see is not what you coded.
- Additional complexity in mapping debugging.
- HTML needs massage to become server-side template.

Difficult programming model - contd.

- Risk of high coupling between view logic and business logic.
- Even if view and business logic is cleanly separated on server, the server is not the client.
- View logic / Interface need to undergo transformation when leaving server.

Heavy state management on the servers

- Client user interface state is maintained by the server.
- This requires a significant increase in resource utilization as server side sessions must be maintained with potentially large object structures within them.
- Usually these resources can’t be released until a session times out, which is often 30 minutes after a user actually leaves the web site.

Offline Difficulties

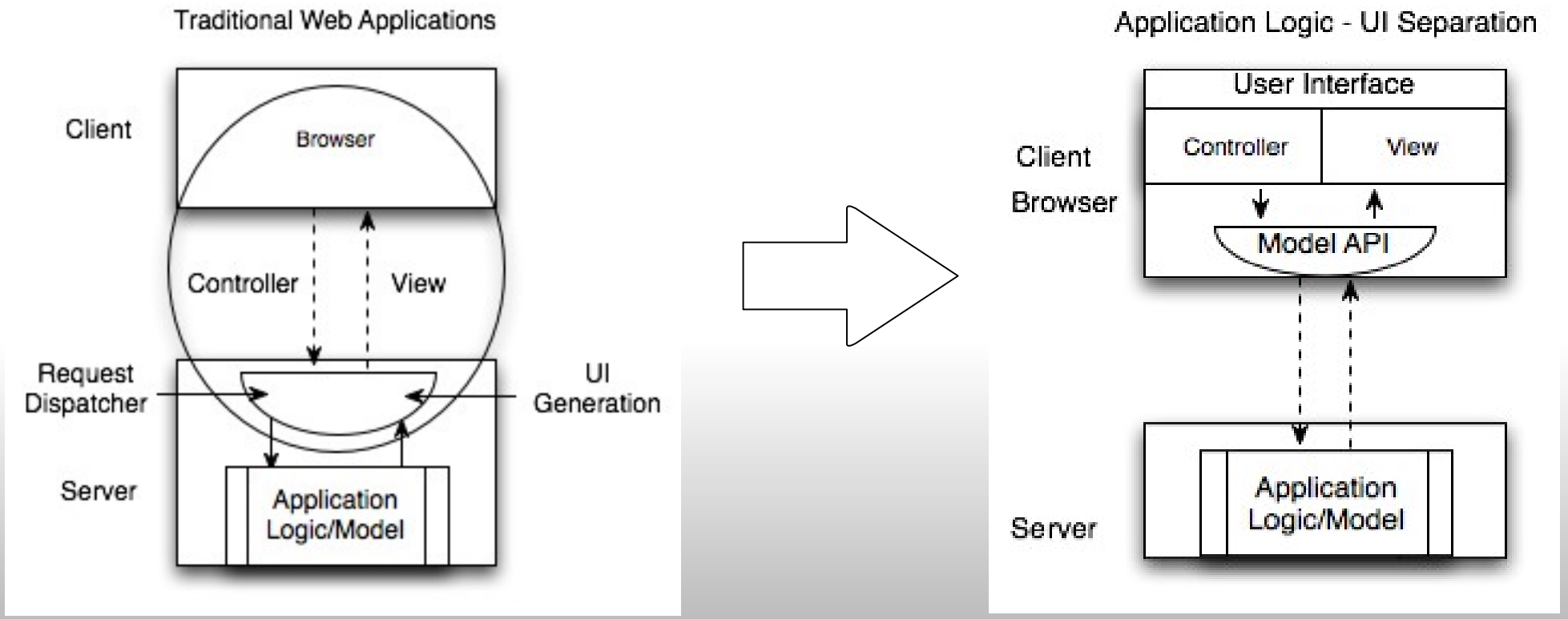
- Adding offline capabilities to a web application can be a tremendous project when user interface code is predominantly on the server.
- The user interface code must be ported to run on the client in offline situations.

Reduced opportunity for interoperability

- Client/server communication is hidden by framework.
- Server 'endpoints' is hard or impossible to isolate.
- Integration (SOA, vendor/customer system) projects must start from scratch.
- Risk of reinventing wheel.
- Changes in web application code might not match changes in integration code.

A modest proposal: Thin Server Architecture

Thin Server Architecture is about moving UI code from the server to the client, any way you like (more or less).



Similar ideas

SOFEA - Service Oriented Front-End Architecture;
http://www.theserverside.com/news/thread.tss?thread_id=47213

Mário Valente's blogs on the future of web applications;
http://mv.asterisco.pt/

Products based on the idea of 'thin servers';

- TIBCO GI
- SmartClient
- WaveMaker (uses Dojo! :)
- Appcelerator
- Probably lots of others..

Thin Server Architecture

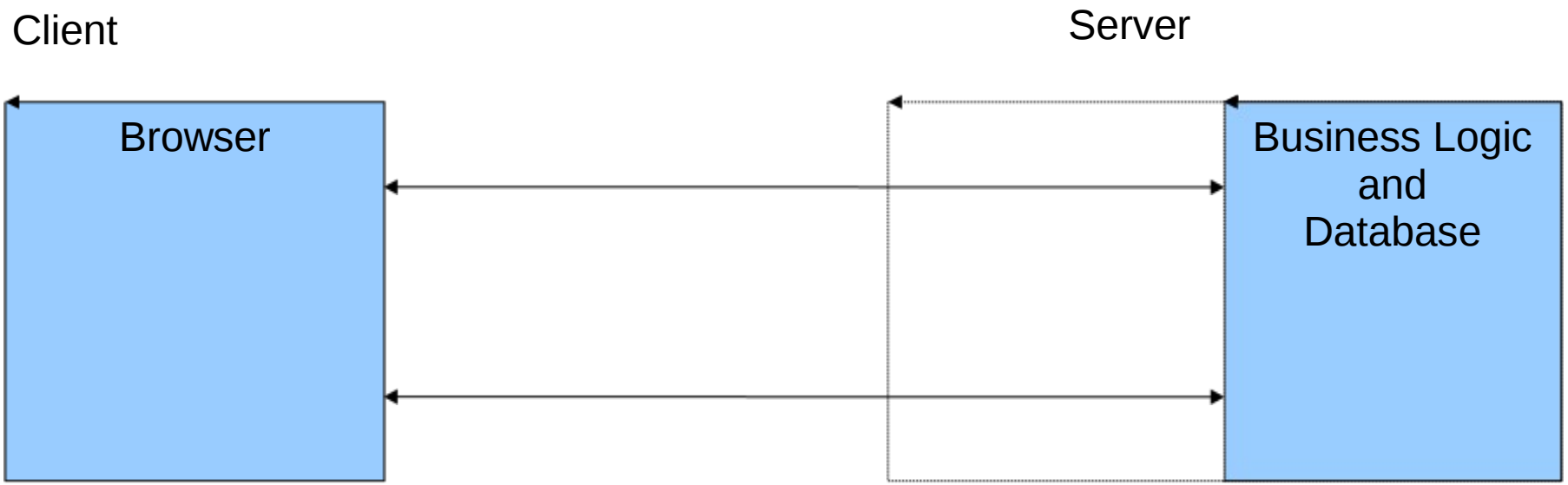
1. **All** functionality that possibly can be implemented on the client side should be implemented on the client side.

2. **All** communication with the application server should be using services (REST, SOAP, domain-specific).

3. No part of the client should be evoked, generated or templated from the server-side. This rules out in-line conditional HTML in JSP, ASP, PHP, et.c.

4. The logic in the server will **only** implement the following functionality; a) Security, b) Business logic, c) Proxying

Thin Server Architecture



Benefits of Thin Server Architecture

- Scalability
- Immediate user response
- Organized programming model
- Client side state management
- Offline applications
- Interoperability

Scalability

- The more clients that use an application, the more client machines that are available (d0h).
- Any presentation logic that can be moved to the client scales 1:1 with the number of clients.

Immediate User Response

- Client side code can immediately react to user input, rather than waiting for network transfers.
- If you need data on the server, you need data on the server. There's no way around that.
- At least we don't have to load THE WHOLE PAGE again, just to get an updated list of cheese.

Organized Programming Model

- Client and server are explicitly modeled and no 'mapping' is required.
- Neither side needs to be 'pretended' on the other side.
- Each side implement its own logic, completely separated from the other, leading to less risk of entanglement.
- When debugging, the messages and object being inspected are 'real'.
- Less complexity.
- Asynchronous development.

Client Side State Management

- Transient session state information on the client reduces the memory load on the server.
- The client decides what to show, cache and request from the server.
- The server manages security, not widget state.

Offline Applications

- Having a clean separation between client and server makes it simpler to implement offline modes.
- Certain toolkits have client-side data abstractions which can switch to offline stores transparently, when needed.

Interoperability

- Using structured data with minimal APIs for interaction, makes it easier to connect additional consumers and producers to interact with existing systems.
- By exposing explicit and secure data-interchange interfaces on the server, we get interoperability for free.

But what about GWT and stuff like that?

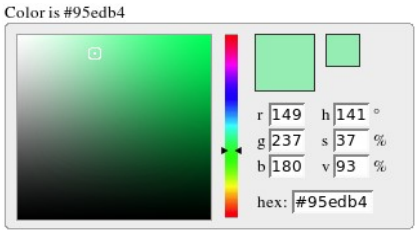
- The code generated does indeed follow separation of concerns, with a clearly defined client.
- However, the development environment is even “better” at hiding what goes on 'under the hood'.
- Debugging is convoluted, trying (and succeeding) in using a single Java abstraction to hide the dual layers of client and server.
- A wonderful but complex solution for a problem that is already solved more simply.

Psychology of not wanting to code on the client

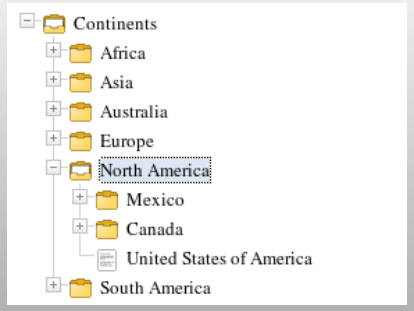
- The hard, steep climb of language mastery : Not again!
- The 'knowledge' gained anno 2003: JS sucks (== browser incompatibilities sucks)
- The world stopped at that date.
- Wanting to search for the keys where the light is, rather than were they were dropped.
- Wanting to make coding web apps less complex!

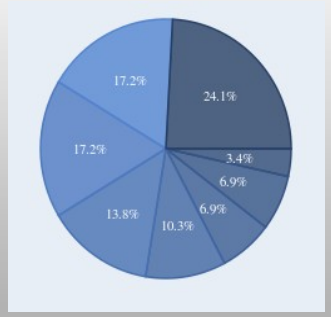
Some new things that has happened since 2003

Ajax toolkits provide consistent APIs and hide cross-browser differences for the following stuff;



- Browser events
- Styling
- DOM positioning and Box-models
- Ajax communication
- 2D graphics (Dojo wraps SVG, VML, Canvas and more)
- Pseudo class-based OO
- And more





Do what's best for each team (and the project)

- Don't force UI logic onto server-side developers.
- Don't force front-end developers to get entwined in server internals

What does thin-server architecture mean for the team?

- **The Designers:** Normal files makes for simpler mockups and simpler changes. Just ordinary HTML/CSS/Images
- **The Front-end developers:** Normal files makes for simpler mockups and simpler changes. Just ordinary HTML/JS
- **The Back-end developers:** No UI logic makes a simpler server. Can focus on _only_ business logic/database stuff
- **More Front-End goodness**: Mock servers/services: Creating the contract and developing asynchronously

Scenario 1

We need to change the styling of the web site.

Now: Does not affect back-end developers.

Before: The HTML and CSS needed to be 'templatized' according to the mad doctor scheme du jour.

Scenario 2

We need to add a tab that shows the same data as in the list tab, but as a bar chart - oh, and you must be able to switch dates dynamically.

Now: Does not affect back-end developers. Charts are rendered to the whim of the user with cached data.

Before: Each date change generates a new page reload. Server is bogged down by the generation of gazillion static images .

Scenario 3

We need to scrap a lot of the code that generates data for the list of car models.

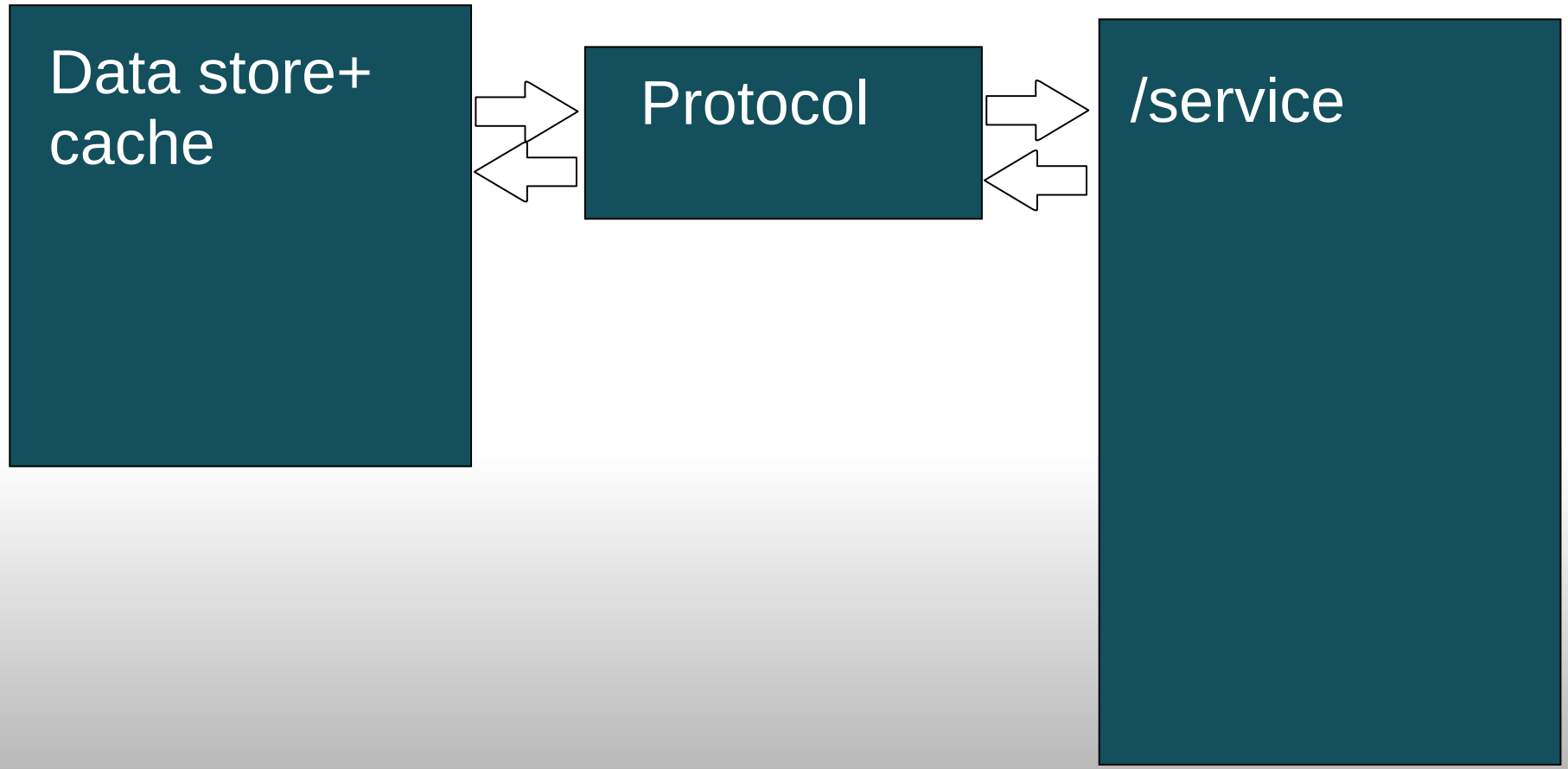
Now: Will not affect Designers or Front-End developers. The web endpoint API is conserved.

Before: Since the old base template that we used to generate the hand-coded rules which drives the XSLT engine which generates the PHP code which contains most of the HTML and CSS for the final page (half are in Barry's module), we need you to split the page into three-line stripes with separate but identical CSS files for each, with all images inlined as binary XML.

Protocol ensures separation of concerns

Client (Browser)

Server



Assertions

- Client logic and markup is less complex implemented on the client, without any magic.
- Client logic amount to between 25-50% of development and maintenance costs (for a user-driven application. My estimates only!).
- Moving client logic to the client reduce complexity on **both** client and server.
- There exists a slight chance that these reductions in complexity might lead to lower development costs..

Enough theory already

Let's see some code

Dojo Example (warming up)

```
<html>
   <head>
      <link href="http://o.aolcdn.com/dojo/1.2.0/dojo/resources/dojo.css" rel="stylesheet" type="text/css"/>
      <link href="http://o.aolcdn.com/dojo/1.2.0/dijit/themes/dijit.css" rel="stylesheet" type="text/css"/>
      <link href="http://o.aolcdn.com/dojo/1.2.0/dijit/themes/tundra/tundra.css" rel="stylesheet" type="text/css" /
>
      <title>Dojo Test</title>
      <script type="text/javascript">
        djConfig= {      parseOnLoad: true  }
      </script>
      <script src="http://o.aolcdn.com/dojo/1.2.0/dojo/dojo.xd.js"  type="text/javascript"></script>
      <script type="text/javascript">
         dojo.require("dojo.parser");
         dojo.require("dijit.form.Button");
         dojo.require("dijit.form.SimpleTextarea");
      </script>
   </head>
  <body class="tundra">
   <button id="mybutton" dojoType="dijit.form.Button" onClick='dijit.byId("q1").attr("value", "foo")'>Click
Me</button>
   <textarea id="q1" dojoType="dijit.form.SimpleTextarea" name="ta1" rows=5 cols=50 ></textarea>
  </body>
</html>
```

Dojo loader

The dojo loader looks where dojo.js is loaded from, and use that as a base url to look for requirements.

Given **http://o.aolcdn.com/dojo/1.2.0/dojo/dojo.xd.js**

The line dojo.require**("dijit.form.button");**

Tries to load **../dijit/form/button.js**

Which becomes **http://o.aolcdn.com/dojo/1.2.0/dijit/form/button.js**

Dojo Custom Widgets

- Use a folder hierarchy to leverage the loader
- Use client-side templates for widgets
- Create nested widgets

Example:

```
dojo-1.2.0                    --- root folder for Dojo
    dojo
    dijit
    dojox
    mycorp                    --- your custom folder
        scaffold
            main.js
            templates
                main.html
```

Steps to load a Dojo widget

1. The Dojo parser parses the page (or part of the page)
2. The parser finds elements with the 'dojoType="x.y.z" ' property
3. Is the Dojo class x.y.z loaded already (by dojo.require, perhaps)?
4. If no, resolve where from, and load it.
5. Create a new instance of class x.y.z
6. Find and populate template (if any) for x.y.z
7. Call magic functions (as defined) for lifecycle management of x.y.z

A custom widget class

```
dojo.provide("scaffold.main");
//...dojo.requires...

dojo.declare("scaffold.main", [  dijit._Widget, dijit._Templated ],
{
  templatePath          : dojo.moduleUrl("scaffold","templates/main.html"),
  widgetsInTemplate     : true,
  content               : {}, // name - url pairs for menu building
  name                  : "",

  postCreate: function()
  {
    console.log("postCreate for scaffold.main called");
      ...
  }
...
});
```

A custom widget template (main.html)

```
<div>
  <div dojoAttachPoint="mymenu" class="scaffold_menu">
      ${name}
  </div>
  <div dojoType="dijit.layout.StackContainer"
dojoAttachPoint="mystack" class="scaffold_main">
  </div>
</div>
```

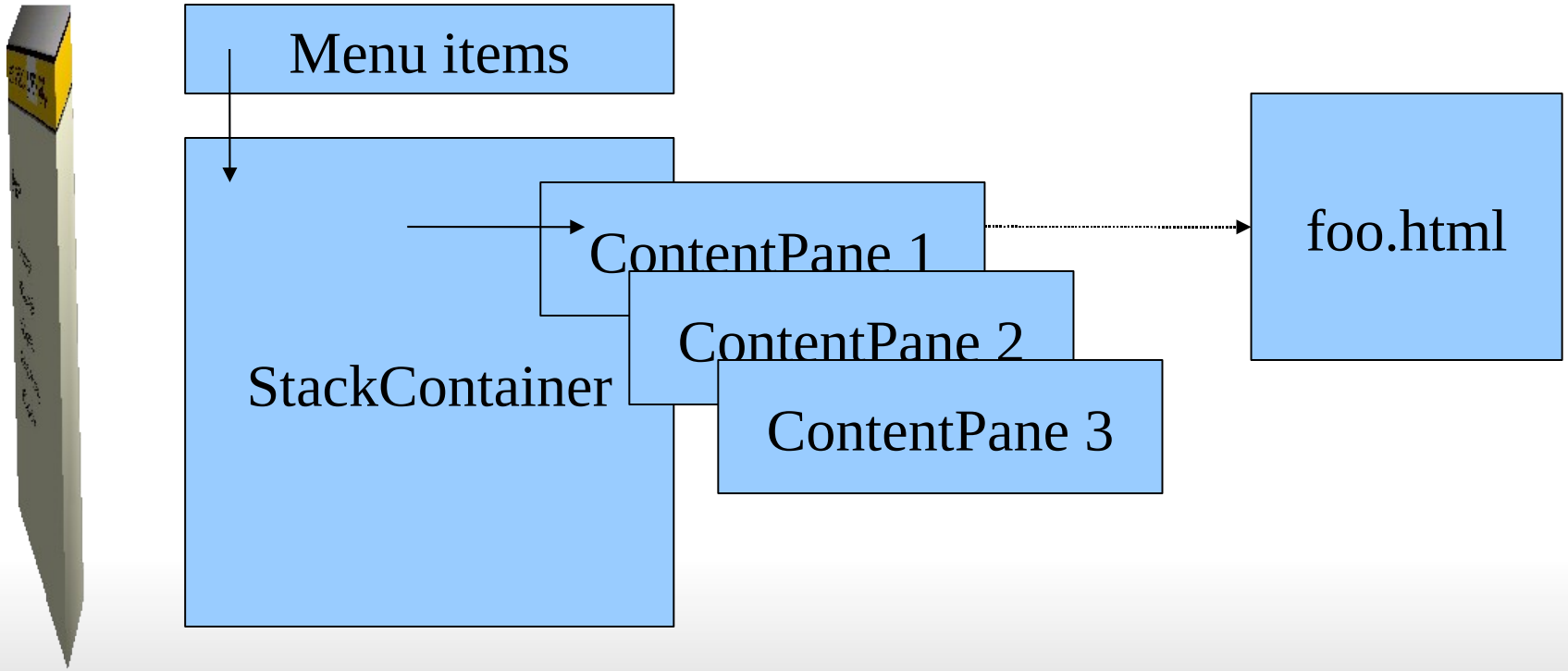
An 'attachPoint="x" ' element or widget gets mapped to a 'this.x' property in the Widget class. ${y} inserts the value of 'this.y' .

Using the custom widget

```
    ...
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("mycorp.scaffold.main");
        </script>
    </head>
<body class="tundra">
    <div dojoType="mycorp.scaffold.main" name="foobar"
    content="{'news': 'scaffold/content/news.html', 'products':
    'scaffold/content/products.html', 'partners':
    'scaffold/content/partners.html', 'people' :
    'scaffold/content/people.html'}">
    </div>
</body>
</html>
```

What the widget might look like

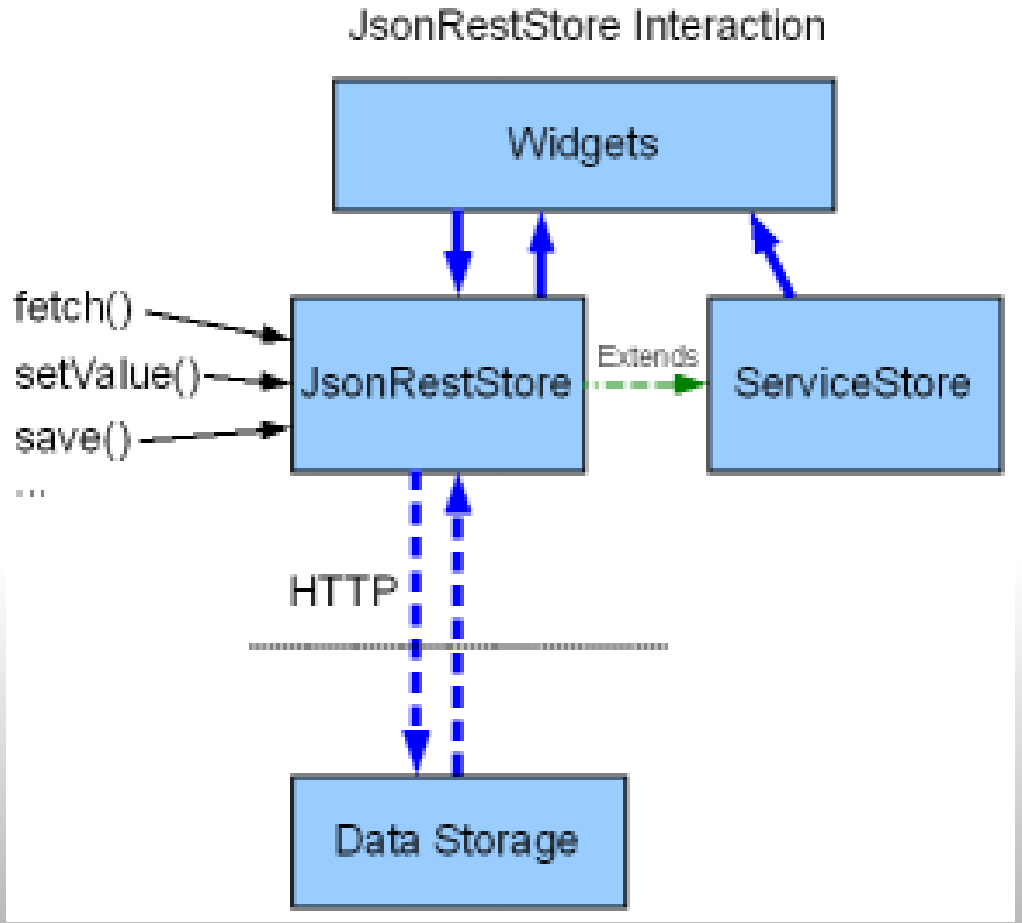
Widget analysis



No servers were harmed during this exercise

Dojo.data

Client-side data model that separates consumers of data (widgets, for example) from the producers (server, services, files, offline)



Dojo.data stores

- AndOrReadStore - Based on ItemFileReadStore, but support complex queries.
- AndOrWriteStore - Based on ItemFileWriteStore as above
- AtomReadStore - RO store for Atom XML based feeds
- CouchDbRestStore - Based on JsonRestStore for r/w couchDb operations.
- CsvStore - A simple store that read CSV-files
- FileStore - A RO store for filesystem traversal and file read.
- FlickrStore - A simple store for explicit Flickr feeds without wildcards.
- FlickrRestStore - Based on FlickStore using Flickr's REST services instead.
- GoogleSearchStore - RO store for Google web searches.
- GoogleFeedStore - RO store that uses Google's RSS/Atom feed services.
- HtmlStore - RO store that reads hierarchical DOM nodes as data.
- HtmlTableStore - Based on HtmlStore but specifically made for tables.
- JsonPathStore - Takes an arbitrary JS object and uses as store data.

Dojo.data stores contd.

- JsonRestStore - RW store using REST access to RESTful services.
- KeyValueStore - Read store for key-value JS objects
- OpmlStore - Read store for the OPML XML data format
- PersevereStore - Read and Write store for the server-side JavaScript data repository system Persevere – based on JsonRestStore.
- PicasaStore - Read store for Picasa web albums
- QueyReadStore - Read store which can pass queries to the server for generating data, which can then be queried differently in the client.
- S3Store - RW store for Amazon S3 storage service.
- ServiceStore - RW store that uses dojox.rpc clients to access server, using either urls or SMD files.
- SnapLogicStore - RO for the SnapLogic open source data integreation framework.
- WikipediaStore - Read access to wikipedia searches
- XmlStore - Read and Write store for XML data.

Read, Write, Identity, Notification APIs

A dojo.data store implements one or more of the following APIs:

- **Read** - Reading items, getting values of attributes of items.
- **Write** - Manages writing to in-memory storage. Can be extended for customized writebacks to services.
- **Identity** - For stores whose items have unique identifiers
- **Notification** - Defines a number of callbacks for store events.

Read API

- getValue(**item**)
- getValues
- getAttributes
- hasAttribute
- containsValue
- isItem
- isItemLoaded
- loadItem
- **fetch // supports queries, return 'item'**
- getLabel
- getLabelAttributes

Write API

- newItem
- deleteItem
- setValue
- setValues
- unsetAttribute
- save
- revert **// return to state of previous save**
- isDirty

Identity

- getIdentity(**item**)
- getIdentityAttributes
- fetchItemByIdentity **// can cause remote access**

Notification

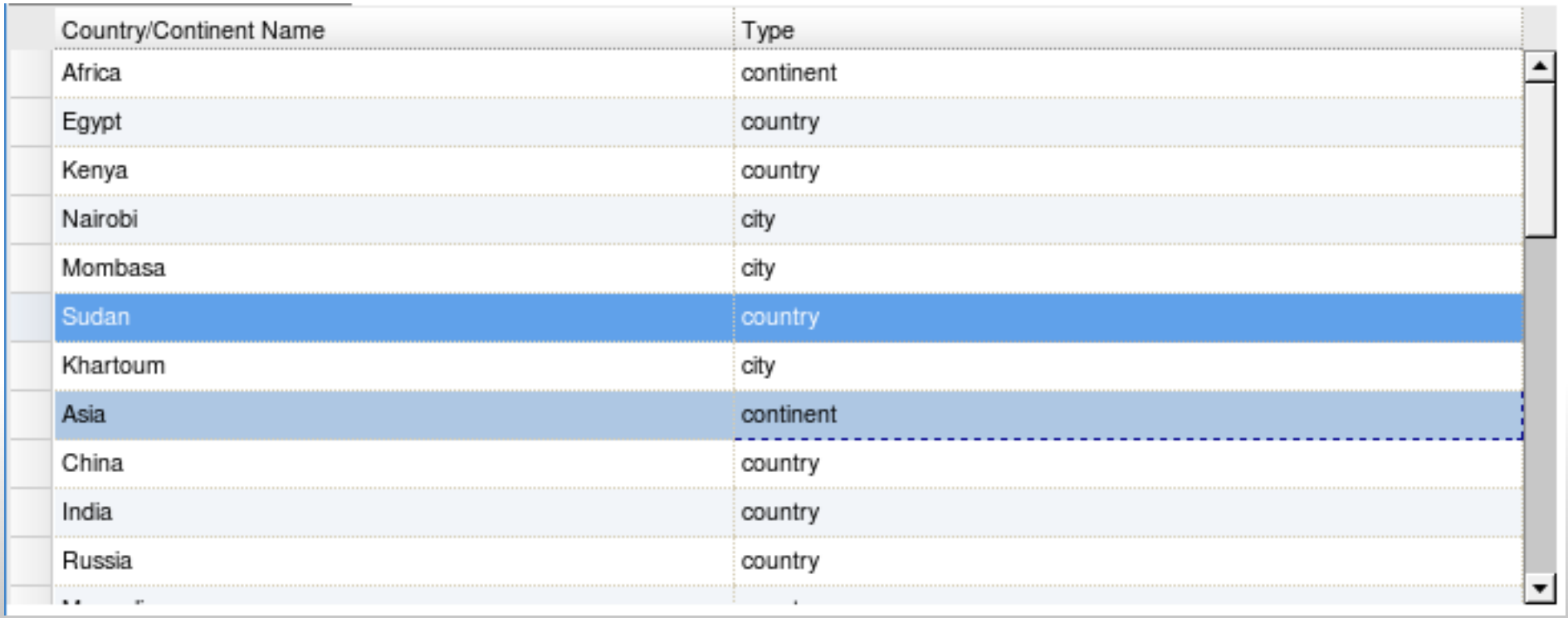
- onSet
- onNew
- onDelete

Example: Widget which reads from a FileItemWriteStore

```
<span dojoType="dojo.data.ItemFileWriteStore"
    jsId="jsonStore"
    url="../../../dijit/tests/_data/countries.json">
</span>
<table dojoType="dojox.grid.DataGrid" jsid="grid" id="grid"
        store="jsonStore" query="{ name: '*' }" rowsPerPage="20" >
    <thead>
        <tr>
            <th field="name" width="300px">Country</th>
            <th field="type" width="auto">Type</th>
        </tr>
    </thead>
</table>
```

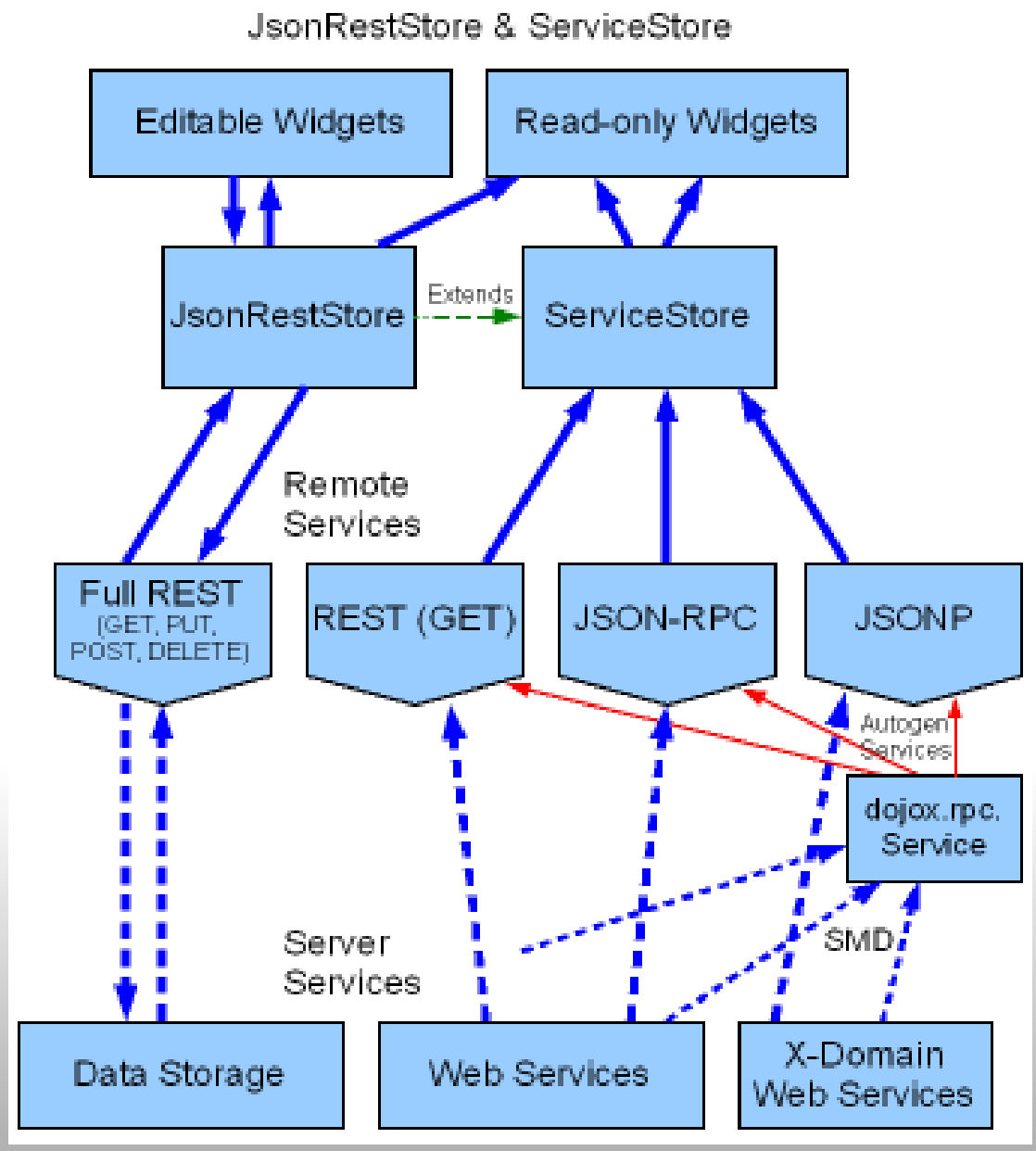
FileItemWriteStore contd.

First a store is defined. Then the widget is connected to the store, optionally using a specific query, like query="{ name: '*' }".



Country/Continent Name	Type
Africa	continent
Egypt	country
Kenya	country
Nairobi	city
Mombasa	city
Sudan	country
Khartoum	city
Asia	continent
China	country
India	country
Russia	country

JsonRestStore: The full monty



Example: JsonRestStore

```
var rs = new dojox.data.JsonRestStore({target:"/recipeStore"});
```

OR

```
var s = dojox.rpc.Service("/mySMD");
var rs = new dojox.data.JsonRestStore({service:s.myRestService});
```

THEN

```
var item = rs.getValue(item,"foo");
rs.setValue(item,"foo","bar");
rs.save();
```

Pluggable services: SMD

Service Mapping Description: 'WSDL done right'

http://groups.google.com/group/json-schema/web/service-mapping-description-proposal

```
{
   "transport": "POST",
   "envelope": "URL"
   "target": "/service/",
   "additionalParameters": true,
   "parameters": [
      {
         "name":"outputType":,
         "default": "json"
      }
   ],
   "services": {
      "foo": {
         "transport": "GET",
         "target": "executeFoo.php",
         "parameters": [
           {"name":"paramOne", "type": "string"},
           {"name":"paramTwo", "type": "integer", "default": 5},
           {"name":"paramThree", "type": "integer", "optional": true}
         ]
      }
   }
}
```

SMD Service Types

- REST
  - JSONP
  - Plain GET
  - Full Read/Write (POST, PUT, DELETE)
- RPC
  - JSON-RPC
- Different URL forms
  - /path/id
  - ?name=value

Dojo includes SMDs for major services

- Wikipedia
- Yahoo
- Google
- Geonames.org

JSON-Schema

"JSON Schema provides a contract for what JSON data is required for a given application and how it can be modified, much like what XML Schema provides for XML. JSON Schema is intended to provide validation, documentation, and interaction control of JSON data."
http://www.json.com/category/json-schema/

Example object:

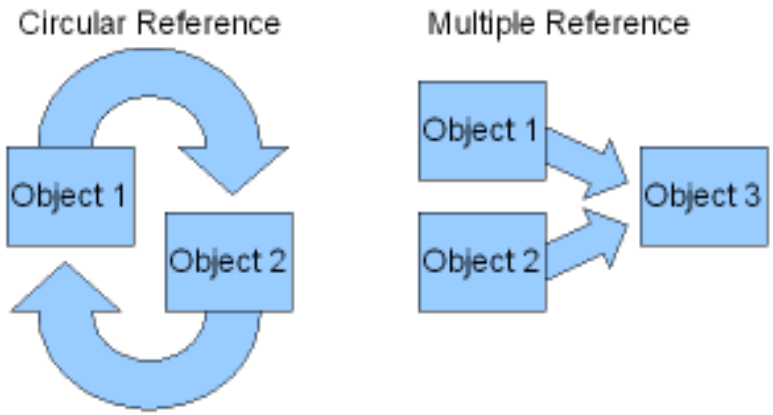
```
{
  "name" : "John Doe",
  "born" : "",
  "gender" : "male",
  "address" :
   {"street":"123 S Main St",
    "city":"Springfield",
    "state":"CA"}
}
```

JSON-Schema contd.

```
{
"description":"A person",
 "type":"object",
 "properties": {
    "name": {"type":"string"},
    "born" : {"type":["integer","string"], allow for a numeric year, or a full date
          "minimum":1900, min/max for when a numberic value is used
          "maximum":2010,
          "format":"date-time", format when a string value is used
          "optional":true}
    ],
    "gender" : {"type":"string",
                "options":[
                  {"value:"male","label":"Guy"},
                  {"value":"female","label":"Gal"}]},
    "address" : {"type":"object",
                 "properties":{
                   "street":{"type":"string"},
                   "city":{"type":"string"},
                   "state":{"type":"string"}
                  }
                }
  }
}
```

JSON-Schema contd.

- JSON Referencing
  - Cyclic/Circular
  - Multiple references
  - Cross-Message references
  - Cross-Site references



http://www.sitepen.com/blog/2008/06/17/json-referencing-in-dojo/

What does TSA imply

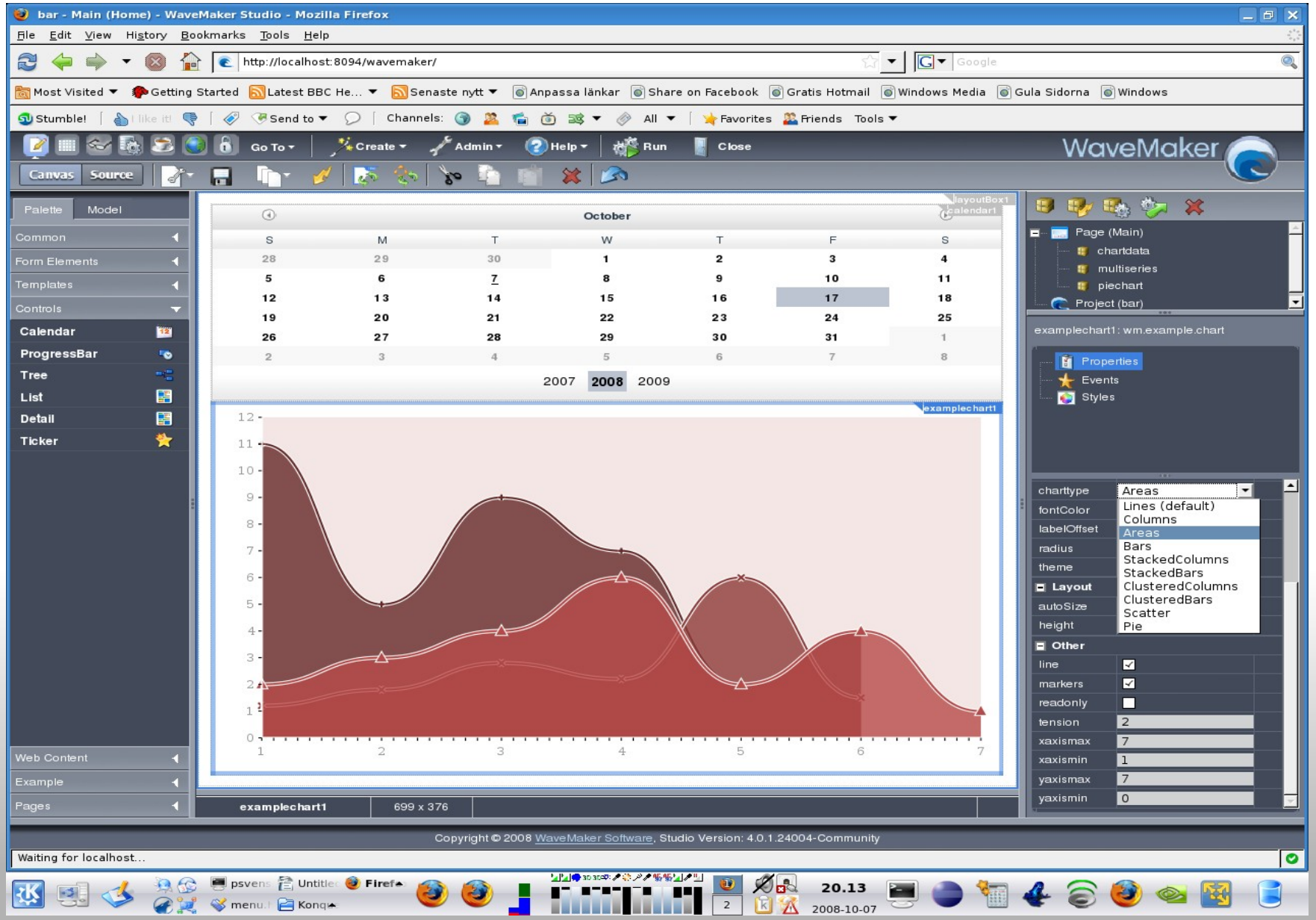
- Single-page applications
- This means the same URL during all states
- “Impossible” to bookmark
- An application as a collection of hyperlinks -> Web 1.0 == Not TSA.
- SEO issues
- Google will save us :) (Flash spidering etc.)

TSA best fit (right now)

- Public sites with login + personal info
- Intranet apps with no public exposure
- Applications that depend on Ajax/JS functionality anyway for their core functionality.
- Public apps that can live with no bookmarking of internal application states.

Helpful tools?

WaveMaker 4



Why engage in open source projects?

- You get to meet great people
- You reach interesting conclusions
- You get to speak on conferences
- You get asked to write a book.