FIFA Ultimate Team at REST

Dr. Harold Chaput, Technical Director, EA Canada

Acknowledgements

- Server Dev Team
  - Andrew Tjew
  - Chris Brown
  - Mohammed Raihan
  - Mark Obsniuk
- Web Dev Team
  - Neale Genereux
  - Andrey Soubbotin

Overview

* FIFA Ultimate Team
* What is REST?
* REST Benefits and Features
* Migrating FUT to REST
* Benefits of a RESTful FUT
* REST beyond FUT
* Advice for becoming RESTful

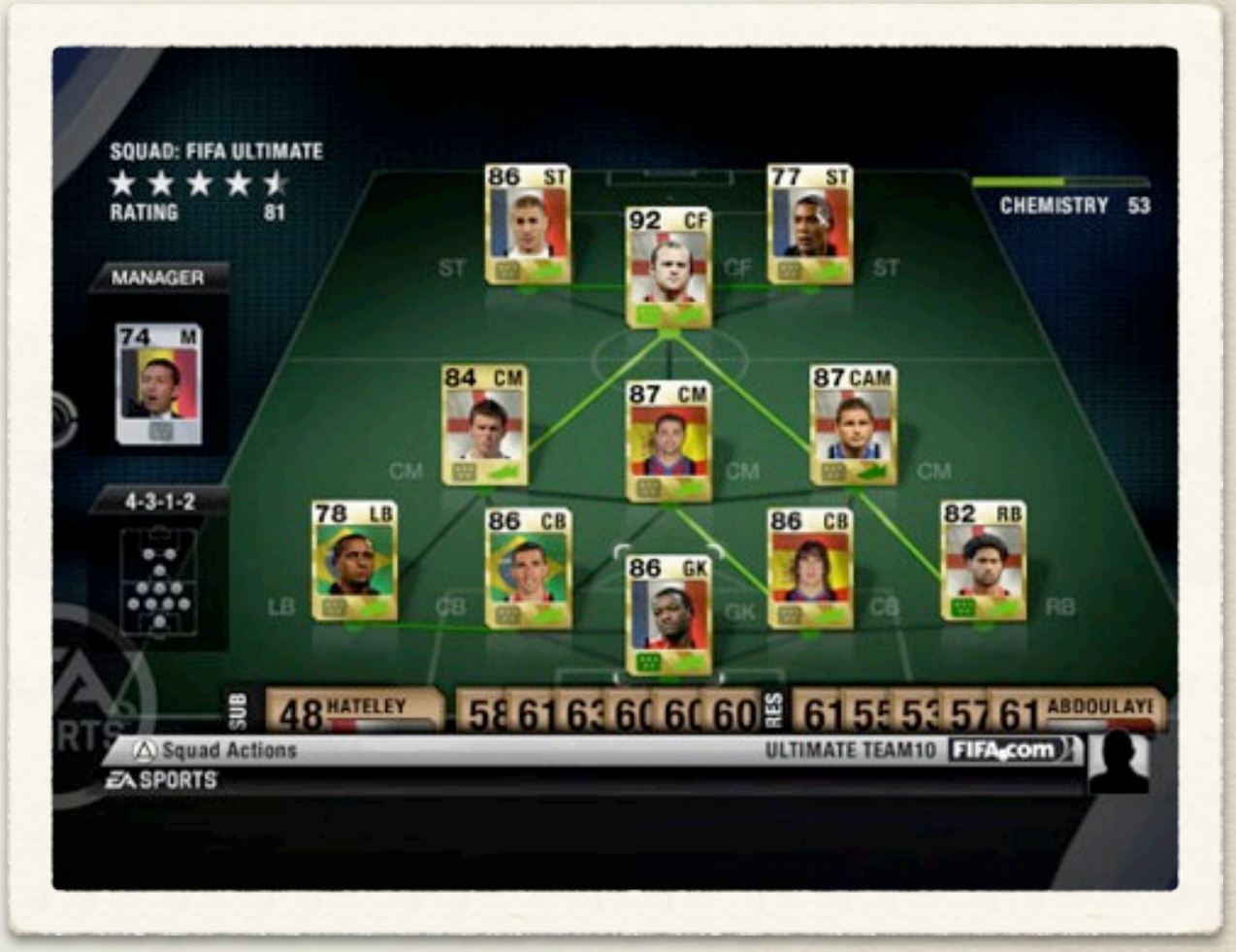
FIFA Ultimate Team

An Unexpected Game

- Break some new ground, alternative to a licensed title
- Card trading game mode in Champions League 07
- Expand the feature set, put it online, sell as add-on
- New product idea: prepared to break even

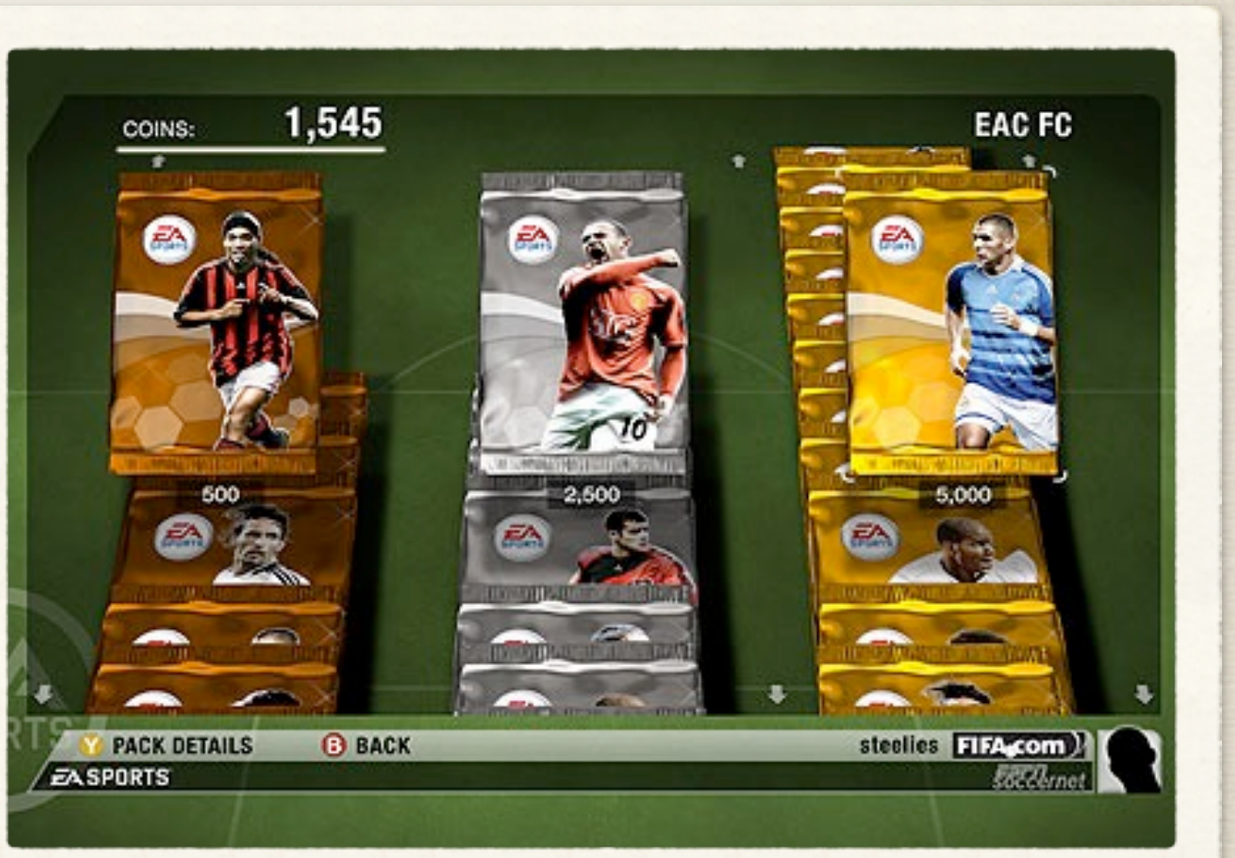
Collect, Trade and Play

- Purchase packs of players, contracts, power-ups
- Trade with other players
- Build your team
  - Team chemistry
- Play your team against another player's team
- Win coins



An Unexpected Success

- Turned a very good profit
  - More than the licensed product would have
- Made more money w/ MTX than selling the mode
- Followed up w/ FUT2 and continued success



Unexpected Problems

- FUT11 servers were shaky at launch
  - Server bottlenecks, connected to FIFA servers
- Game logic on client
  - Hard to update post-launch
- UI info on the server ("glow")
- Followed console model of server per title
  - No year-over-year support

A New Client

* FUT Web introduced in April 2010
* FUT servers used proprietary format, not HTTP
* Took advantage of an opportunity to start over
* ...with REST

What is REST?

The dirty details

What is REST?

- REST stands for “REpresentation State Transfer”
  - REST is style of software architecture
  - REST is intended for online services

- REST first defined by Roy Fielding
  - “Architectural Styles and the Design of Network-based Software Architectures” (2000)
  - Fielding is the principle author of HTTP 1.0 and 1.1
  - Created for “distributed hypermedia” systems
  - Applications and benefits extend beyond WWW

REST is a Style, like OOP

- OOP is a *style* of software architecture
  - REST is a convention, not a syntax
- OOP can be done in many languages
  - Many protocols can be RESTful
- OOP has several variants and flavors
  - REST is also underdetermined
- OOP is open to interpretation
  - There are many ways to be RESTful
- OOP won't solve all your problems, introduces new ones
  - REST is not a complete solution
- Follow OOP, and gain benefits
  - ...and so it is with REST

REST Constraints

- Client/Server (separation of concerns)
- Uniform Interface w/ Hyperlinks
- Stateless
- Cacheable
- Layered
- *Code on Demand (optional)*

Client and Server

Client

User Interface
Rendering
Current Page
Device Security

?

Server

!

Database
File Access
Load Balancing
Fraud Detection

Session State

Application State

Separation of Concerns

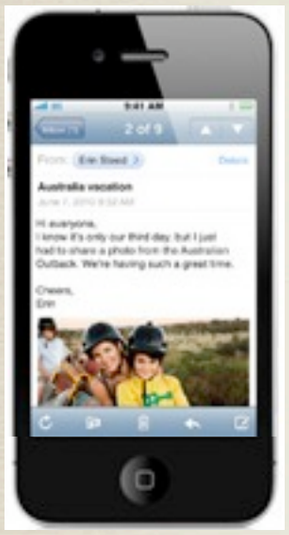
Services and Resources

Client

Resources

Services

Server



?

Session State

Message

Folder

Mail

Contact

Mailing List

Address Book

Appointment

Meeting Room

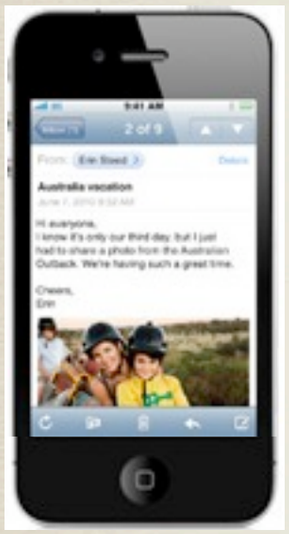
Calendar

Application State

Resource Representation

Changing Application State

Client



Session State

Resources

Services

Message

Folder

Mail

Contact

Mailing List

Address Book

Appointment

Meeting Room

Calendar

Server

Application State

Representational State Transfer

Statelessness

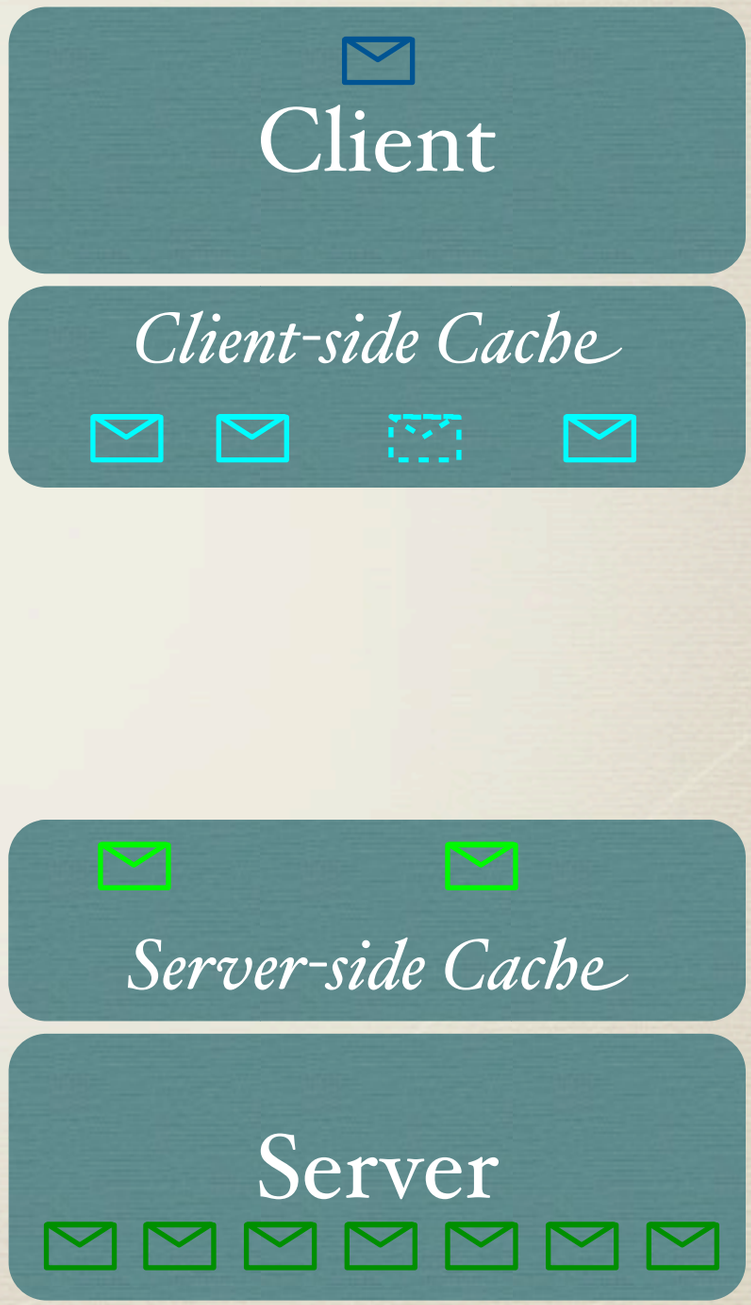
Stateful	Stateless
`move(direction, speed)`	`set_position(x, y)`
`find_user("Bob")` `send_msg("Hello!")`	`send_msg("Bob", "Hello!")`

`u1=owner(i1)`
`u2=owner(i2)`
`assign(i1, u2)`
`assign(i2, u1)`
	`u1=owner(i1)`
	`u2=owner(i2)`
	`assign(i1, u1, u2)`
	`assign(i2, u2, u1)`

All required state information is in the request.

Cacheability

- Representations are cacheable
  - Counters extra traffic caused by statelessness
- Representations include expiration info
- Representations can contain references to resources



Uniform Interface

* Resources must be uniformly identified
  * The same ID results in the same resource

* API consists of:
  * Resource representations
  * Resource identifiers
  * Requests and responses

Layered

Filter

Authentication

Cache

Router

Service A | Env 1 | Service B

Service C | Env 2 | Service D

REST Constraints

* Client/Server (separation of concerns)
* Uniform Interface w/ Hyperlinks
* Stateless
* Cacheable
* Layered

* If it has all these properties, it is RESTful.

Why REST?

What's in it for me?

REST Benefits

- Simplicity
  - Issues stay where they belong
  - Information is localized
  - Developers know what to build, can work in parallel
- Performance
  - Caching decreases response time, reduces DB hits
  - Can distribute across multiple servers
- Scalability
  - Services interact through references only
  - Can easily introduce new hardware as required
- Visibility
  - Clients know what resources are available
  - Clients are informed of the new state

REST Benefits

- Portability
  - All clients use the uniform interface
  - New clients can be added post hoc
  - Prepare us for a multi-platform online world
- Reliability
  - Transactions continued on new hardware if overloaded or crashed
  - Layered routers can direct traffic as needed

REST Best Practices

* HTTP
  * Conceptually compatible
  * Use four verbs: GET, PUT, POST, DELETE
* XML / JSON
  * Structured, easy to construct & parse, allows refs
* Java / C# / Ruby / Python
  * Can be deployed on arbitrary hardware
  * Built for reliability (not speed)
* REST vs. SOAP
  * SOAP *can be* RESTful (and C can be OO)
  * SOAP is more than you need for REST

RESTful Examples

- Twitter
- Facebook
- Picasa
- YouTube
- Flickr
- Google
- OpenSocial
- JIRA
- Gowalla
- Amazon
- Spore
- ...hundreds more

Migrating FUT to REST

Getting from here to there

Step 1: A RESTful API

FUT Console Servers

FUT REST API

Step 2: RESTful Proxy

FUT Console Servers

FUT REST API

FUT Application Server

Step 3: Componentization

FUT Console Servers

Warehouse	Store
Trading	Tournaments

FUT REST API

FUT Application Server

Step 4: Application Servers

FUT Console Servers

FUT REST API

FUT Application Server

Warehouse

Store

Trading

Tournaments

Step 5: All Clients on REST

FUT REST API

FUT Application Server

Warehouse

Store

Trading

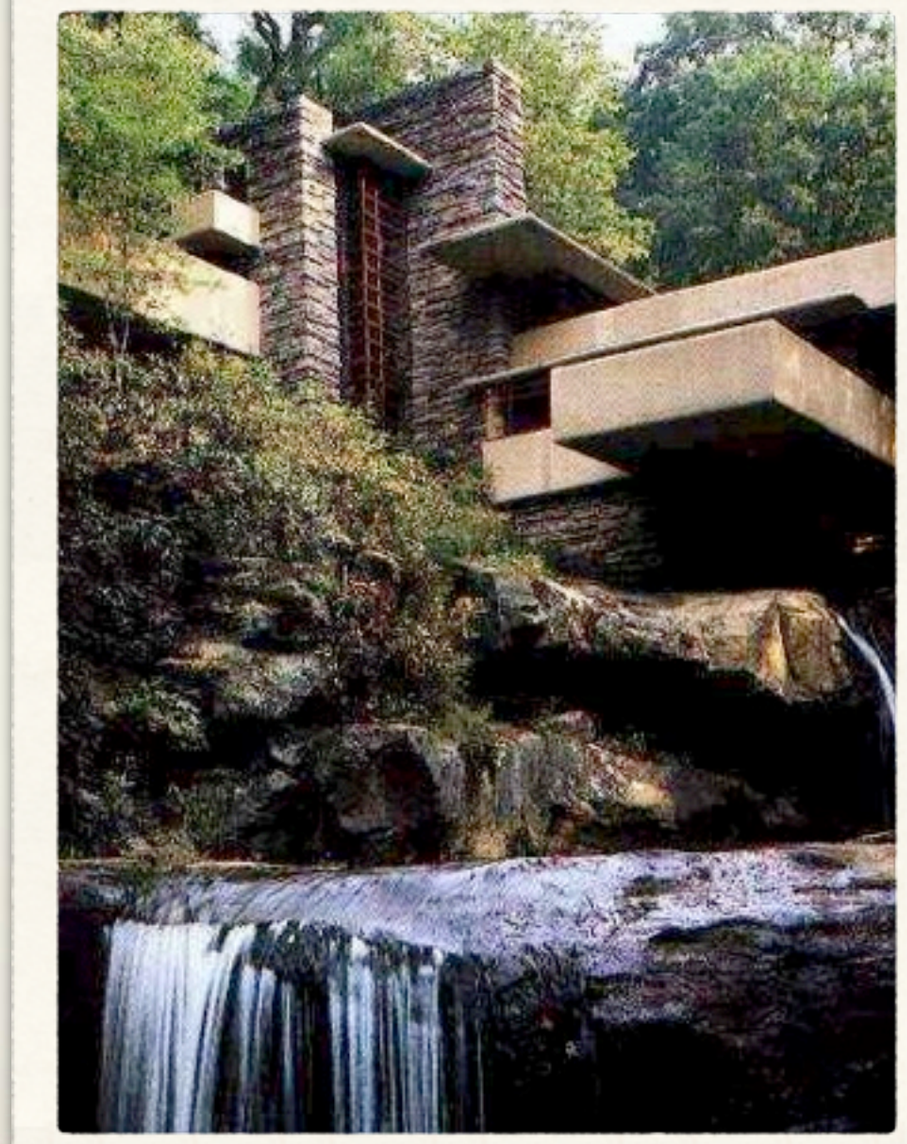
Tournaments

How REST Benefits FUT

The Big Wins

REST for Good Design

* REST can guide good design decisions
* Example: transactions as resources
* Enforces stateless transactions
* Removed serious transaction problems and increased robustness



Modular Services

- FUT built from components
- Scaling components independently based on use
- Updating implementation one step at a time

Services are Shared

- FUT components used by other products
- FUT uses other shared components
- Components managed independently

Enforces Organization

- Each service has a well-defined function
- Dramatically fewer server bugs
- Problems are isolated and easy to find
- Clients better understand how to implement (web client took four months)

Promotes Live Updating

- Client gets state from server
- Easy to update functionality
  - Even for data we didn't expect to update: item types

Support New Clients

- No client assumptions in the server
- New clients can be supported quickly
- No need for server team support

REST beyond FUT

What else can we do?

Games as Products

Games as Services

Games as Services

Advice for Getting Rest

Lessons Learned

Think Resources

- Define your services in terms of resources
- Determine how they will be identified, represented, manipulated
- Organize them hierarchically
- Opposite of OOP
  - Not data hiding, data exposing

API Before Code

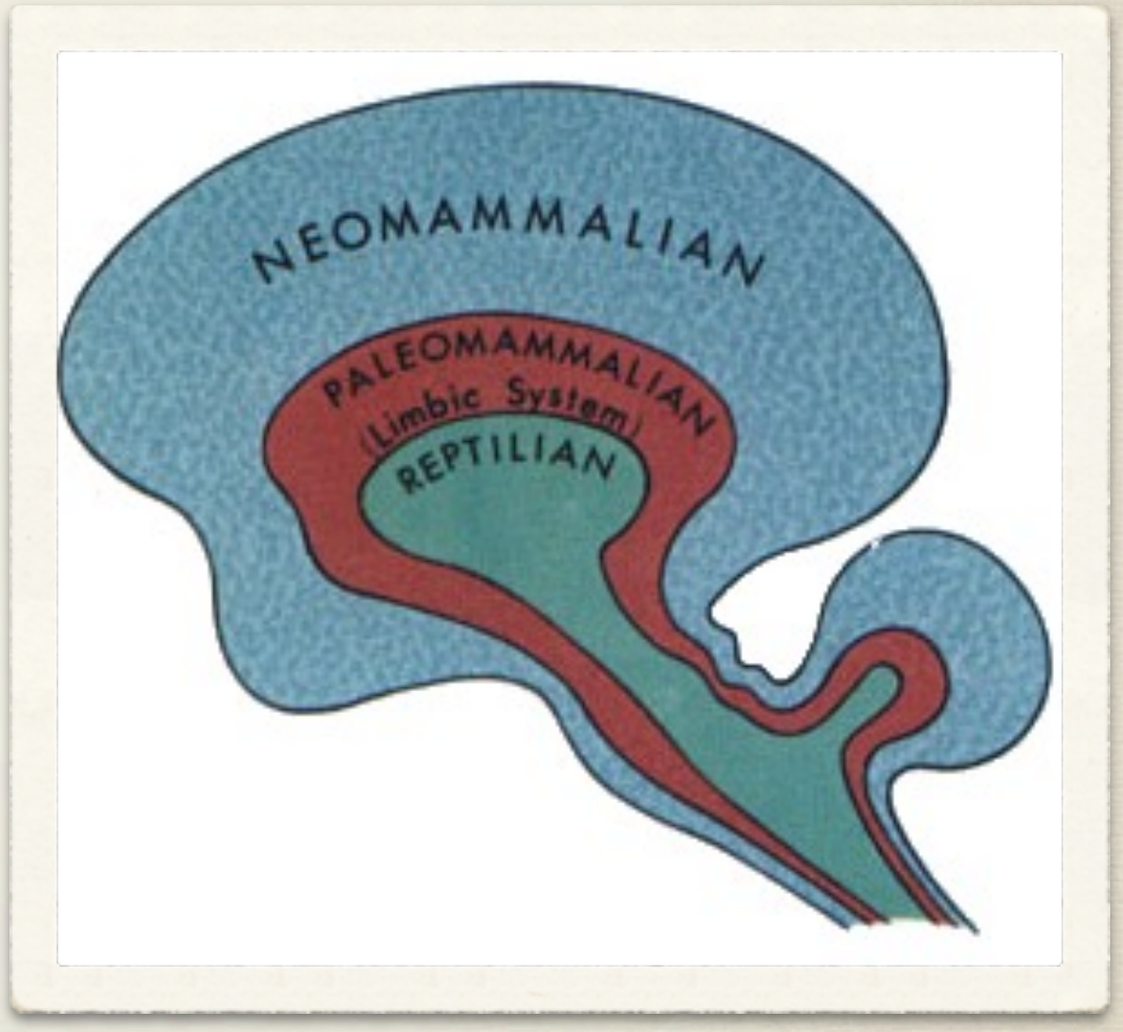
- Document your API *before* you write your server
- Get API vetted by client teams
- Development can start on client & server together
- Can test out API with working client & dummy data

Plan to Hand Off

- Build your service to be handed off to your ops team
- Don't assume you have any control of the server
- Clients can go though unexpected layers
- May not talk to the same server instance twice

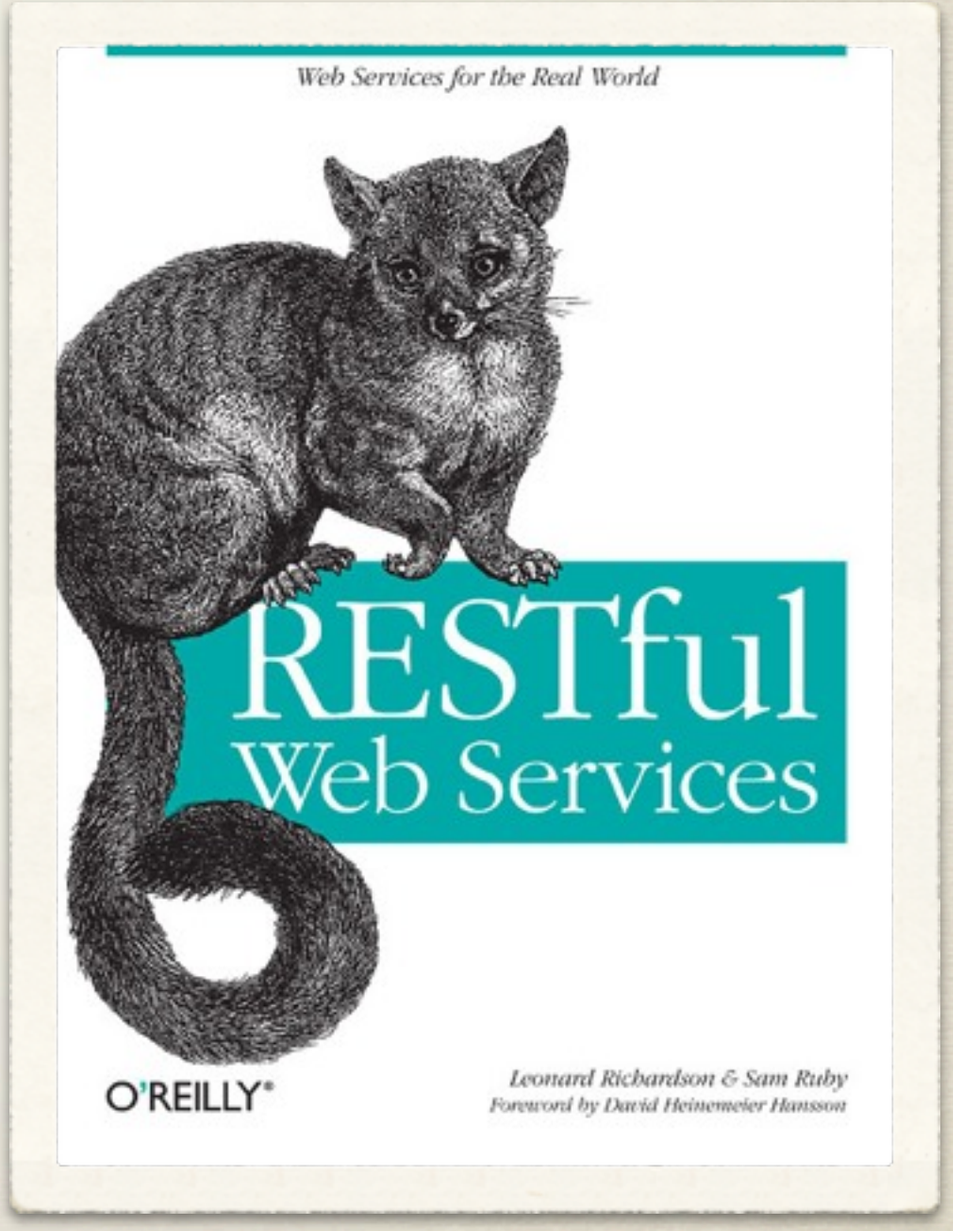
Version through API

* Expand existing APIs
  * Build clients to ignore extra information
* When API meaning changes, make new resource
* Sunset old resources when clients stop using them



Good Reference

* Richardson & Ruby
* Understandable explanations
* Several good examples
  * Transaction resource



The End

hchaput@ea.com