Reactive, unidirectional data

without a required client/server technology

David-John Burrowes

Talk Structure

1. What are patterns?
2. How are they implemented?

Part 1
What are the patterns?

Reactive

- Problem
    - How does client know when data has changed on the server?
    - App writer must ask for data, cache it, manage it

Reactive

- Definition: Changes to the data on the server are swiftly communicated to the client
  - App developer doesn’t need to do any special work. It “just happens”

Reactive

- When data is created, updated or deleted (CRUD), server queues a nugget of data:
  - CREATED
  - Object: FOO-1
  - Type: Widget

Reactive

- Library on client long-poll’s the server
- Nuggets delivered
  - Library compares with data already managing
  - If it cares, it asks for the created/updated objects

Unidirectional

- Problem
  - Data changes on client and server
    - Where is the truth?
    - How to deal with implementation disagreements?

Unidirectional

- Server is the source of truth
- Client versions of server data immutable
- To change the server data, client asks server to change

Unidirectional

- Easy to reason about the data
- Makes code cleaner

Common Types (and API's)

- Problem
  - Client and server views of data types can diverge
  - URL's given precedence over data
  - Custom code to validating data

Common Types (and API's)

- Data types defined in neutral format
  - Check out http://json-schema.org/

Common Types (and API’s)

- Schemas define
  - Properties
    - Data types
    - Constraints (max, min, length, etc)
- We add
  - Supertypes
  - Operations on the types
  - Locations of types in the URL-space

Common Types (and API’s)

- Single source of (data type) truth for all consumers (clients and server)
- Code generated on every build
- Changes always validated (tests or build)

Common Types (and API’s)

- Client and server can’t get out of sync
- Can enforce type’s constraints at the library level
- Server can generically validate all input from client

Common Types (and API’s)

- Client testing is easier
- API Migration handled automatically

Part 2
How are they implemented?

Demo application

- Available at
  - https://github.com/delphix/dxData

Reactivity: App developer

- App developer does no special work to make sure data is up to date

Reactivity: Server

- Change to db conveyed to notification system
- Notification system queues changes for clients

Reactivity: Client

- Client polls for notifications
    - Is the object in the cache?
        - If so, it is being used, so we should update it
    - Is there a collection of the object type?
        - If so, we want the new object
    - Otherwise, we ignore the notification

Unidirectional

- getServerModel() vs normal Backbone.
extend()
- .set() throws an exception
- $$update() to update

Common Types (and API’s)

- Look at a buzz.json
  - Properties
  - Super type
- Look at user.json
  - Has an operation

Common Types (and API's)

- python parseSchemas.py
  - Generates python classes
- dxCoreData
  - Dynamically creates Backbone-based objects at runtime

Common Types (and API’s)

- App developer only works with objects and their operations
  - URL’s and HTTP method’s not visible or relevant

Other topics

- Mock server
- Details of dxCoreData
- Filtering

Contributors

- Aaron Garvey
- Abdullah Mourad
- Brett Lazarus
- Chris Patten
- Chris Siden
- Eyal Kaspi
- Dan Kimmel
- David Burrowes
- Eric Schrock
- Henrik Mattesson
- Henry Rodrick
- Kenneth Lim
- Prateek Sharma
- You?

Resources

- David-John Burrowes
  - david.burrowes@delphix.com
- Repository
  - https://github.com/delphix/dxData
- json-schema
  - http://json-schema.org/

Thank you