progressive enhancement for JS apps

Garann Means  @garannm
why this talk?

---

```
# behavior
'fieldState' based on cursor position
'fieldState' insert syntax, tags, and step into
'fieldState' delete this set of tags and user-deleted syntax; if state has parents/siblings, leave them in place (flagged as potential?)
'fieldState' ensure tag is closed, move out of
'fieldState' ?
'fieldState' ?
'fieldState' when start syntax recognized, look for subsequent tags that could match as end syntax (in the same block?)
'fieldState' given a block, hierarchical list of tags (and potentials?) it contains
'fieldState' enter twice, inserts new `<p>`
'fieldState' cleanup, move cross-block end tags to previous block

# cursor
'fieldState' based on cursor position, replace any existing content
```
a little case study

- tons and tons of JS
- real-time
- collaborative
- offline capabilities*
- fallback: a textarea
the fix: progressive enhancement
2003–2008

🚀 “let’s use JS for everything!”
2003-2008

“let’s use JS for everything!”

“what, people have JS turned off?”
2003-2008

“let’s use JS for everything!”

“what, people have JS turned off?”

“what, browser support is inconsistent?”
2003-2008

“let’s use JS for everything!”

“what, people have JS turned off?”

“what, browser support is inconsistent?”

“who cares, our customers are cutting edge”
2003-2008

“let’s use JS for everything!”

“what, people have JS turned off?”

“what, browser support is inconsistent?”

“who cares, our customers are cutting edge”

from yesterday: “if you’re seeing a broken experience, your environment is broken”
2003-2008

“let’s use JS for everything!”

“What, people have JS turned off?”

“What, browser support is inconsistent?”

“Who cares, our customers are cutting edge”

From yesterday: “If you’re seeing a broken experience, your environment is broken”

“Oh weird our business failed”
progressive enhancement:

🚀 deliver a usable baseline experience

🚀 offer an upgraded experience to more powerful browsers
vs. graceful degradation

🚀 code for the most powerful clients

🚀 problem? deliver a lesser experience
why’d we switch?

- degradation is harder to plan accurately
- degradation makes things even slower for less powerful clients
- hope for the best, plan for the worst
this is a best practice on the front-end
JS apps
JS apps

“oh, but this isn’t for people who don’t have JavaScript”
JS apps

“oh, but this isn’t for people who don’t have JavaScript”

“..or people returning to the site”
JS apps

“oh, but this isn’t for people who don’t have JavaScript”

“.or people returning to the site”

“.or people going through a tunnel on a train”
have we learned nothing?

🚀 obviously we want the best experience possible

🚀 this is the internet; “possible” changes

🚀 brittle expectations of our users means a worse experience

🚀 for everyone
what to do with that thinking?
a method

- build servers as if every client is text-only
- build clients as if the internet will disappear tomorrow
- expect people not to use the app the way you expected
some simple principles

- state is shared between client and server
- the client takes responsibility for state offline
- and attempts to sync when it comes online
- actions, by default, occur on both client and server
“computer, enhance!”
“computer, enhance!”

🚀 is JS available?

🚀 great, run the client-side app
“computer, enhance!”

🚀 is JS available?

🚀 great, run the client-side app

🚀 are we online?

🚀 great, send updates to the server
“computer, enhance!”

- is JS available?
  - great, run the client-side app
- are we online?
  - great, send updates to the server
- client and server aren’t in sync?
  - no problem, put them in sync
sharing state is more than just sharing state
this is a state

$( "#form-step-2" ).show();
(this is a shared state)

body.registration-step-2 #form-step-2 {
  display: block;
}

this one isn’t being shared

$( "#form-step-2" ) . show();
state runs the app

🚀 interface changes

🚀 fetching data

🚀 app logic updates the state

🚀 state changes trigger everything else
we can still do this

$( "#form-step-2" ) . show();
it just works differently

```javascript
$( "#form-step-2" )[
    registration_step == 2 ?
    "show" :
    "hide"
](());
```
protip: CSS is an observer by default
app.get("/register/:step?", function( req, res ) {
  var step = req.params.step || registration_step;
  res.render( "reg_step_" + step, registration_data );
});
question: where is registration_step set?
glad you asked!

- updates about state are separate from updates as a result of state
- data updates are chunked
- state updates are instant (if possible)
separate state updates
make syncing easier
let's say our user goes offline

> my_app.state.registration_step
1
then clicks submit

$( "#form-step-1" ).on( "submit", function()
  set_registration_step( 2 );
  return false;
});
it’s fine

> my_app.state.registration_step
2
function set_registration_step( step ) {
  my_app.state.registration_step = step;
  update_state( "registration_step_" + step );
  var data = get_data( step - 1 );
  save_data( ( step - 1 ), data );
  $( "form-step-" + step ).show();
}
and stored

function save_data( step, data ) {
  db.put( data, "registration_" + step,
    function( err, response ) {
      if ( err ) {
        show_error( "Data not saved!" );
      }
    });
}
note: plan your storage

🚀 small apps may be able to be stored entirely

🚀 only the current workflow in large apps may be able to be stored

🚀 the more changes happen offline, the less room for potentially used data
if you accept data, you have to hang onto it
how does the client know it’s offline?

- poll for server updates
- catch a socket error
- sending data via normal XHR
- wait for an error saving
hope for the best, plan for the worst

function update_state( state ) {
  state_cache.push( state )
  var req = $.ajax({
    type: "POST",
    url: "/update_state",
    data: state_cache
  }).done( function() {
    state_cache = [];
  }).fail( function( xhr, err ) {
    if ( err == "timeout" ) start_heartbeat();
  });
}
offline, the client is on its own
how do know you’re back?

try it

navigator.onLine

probably some sort of heartbeat

no matter your strategy
and then:

- send client’s current state
- follow a defined path from server’s current state to client’s
- or send specific steps from the client
- sync server data with client
tl;dr, offline is scary
so let’s talk actions

<input type="submit" value="Continue 🖐" />

<!-- NO NO NO NO NO NO NO NO NO NO NO NO NO NO NO NO NO NO NO
<div id="save-step-1">Continue 🖐</div> -->
actions in the interface

🚀 never trigger actions directly
🚀 actions are triggered by state changes
🚀 state changes update client and server
🚀 then the action itself occurs
this guy

$( "#form-step-2" ).show();
and this one

$( "#form-step-2" ).save();
saving (for example)

- fetches user-entered data from the interface
- stores it in an application variable
- adds it to the local data store
- notifies the server that a save occurred
- syncs data
then the server

- default: fetches user-entered data from the client
- enhanced: syncs with client DB
- stores it in an application variable
- adds it to the backend data store
- notifies any other connected clients that there are changes
same thing happens on both sides, why?
other connected clients & reloads

- fetches user-entered data from the server
- stores it in an application variable
- adds it to its local data store
- updates its interface
if JS disappears

🚀 we use HTML behaviors (links, buttons) to skip right to the server step

🚀 concurrent users receive immediately-expiring data
we need state reflected in:

- client and server variables, ofc
- the URL
- the CSS
tada: it’s progressive!

🚀 our server has a version of the app and can render its interface

🚀 client side can continue without the server, in case we go offline

🚀 real-time communication and multiple concurrent users follow the same pattern
our app is now

🚀 usable by multiple clients

🚀 in multiple locations

🚀 and easier to continue extending

🚀 by layering functionality, as with progressive enhancement
thanks!

@garannm  garann@gmail.com