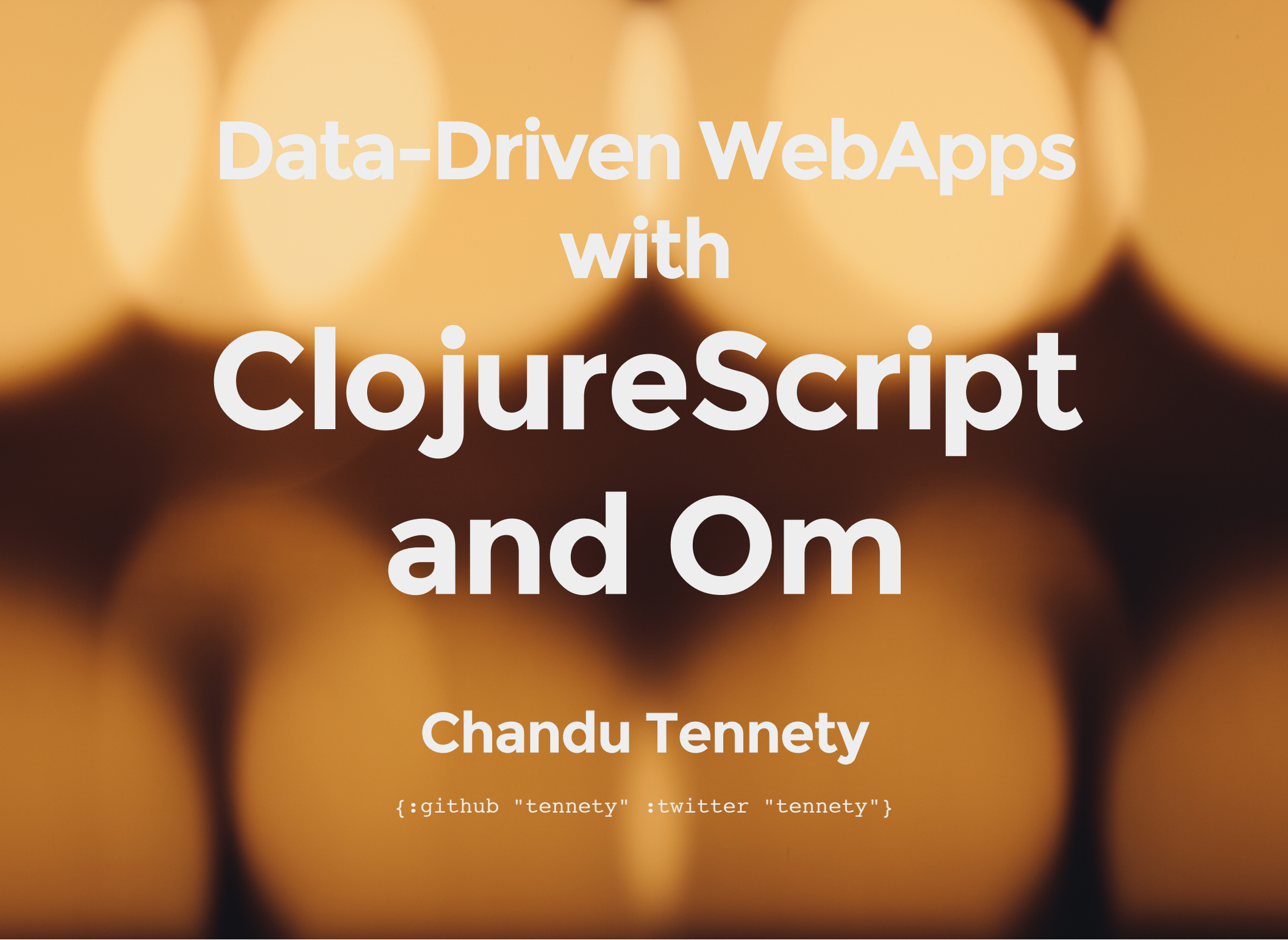
Data-Driven WebApps
with
ClojureScript
and Om
Chandu Tennety
{:github "tennety" :twitter "tennety"}

What is ClojureScript?

Compiler for Clojure to JS

Benefits over vanilla JS

- Module system
- Persistent data structures
- Laziness
- Destructuring and method arity dispatch
- Polymorphism with protocols and multimethods
- Macros

Interop with JavaScript

- Property access

```
(.. js/d3 -event -targe
```

- Method access

```
(.select js/d3 ".active
```

- New object creation

```
(History.)
```

Reader literals

- `#js`
- `clj->js`
- `js->clj`

Makes use of Google's Closure tools

Installation and Setup

Grab the project

- Install Leiningen
- https://github.com/tennety/slides

Start the REPL

- `cd slides`
- `lein deps`
- `lein repl`
- `(run)`
- `(browser-repl)`

Chestnut

`git checkout intro`

At the bottom of `src/cljs/slides/core.cljs`, type:

```
(swap! app-state assoc :text "LambdaConf 2015")
```

Then save the file.

You could also type it out in the browser-repl and hit Enter.

Data-Driven Applications

Application state is global

UI is divided into components

Each component is a function over a cursor into the application state

Each component can be responsible for its own state

What We'll Build

Demo

React and Om

What happened in the intro?

Try it again!

- Application state changes
- Related component is notified
- Rerender is scheduled

Breaking the UI into components

Components in our app:

- Slide
  - Background image
  - Caption
- Previous slide
- Next slide

Exercise

Let's start simple

Where's the data?

For one slide:

- Image URL
- Caption

```
(defonce app-state (atom {:slide {}}))
```

```
(def slide-img
  {:title "A Bird's Eye View of ClojureScript"
   :bg "/images/gull.jpg"})
```

```
(swap! app-state assoc :slide slide-img)
```

What should the DOM look like?

For one slide:

```
<section class="slide">
  <img src={bg} class="bg">
  <div class="slide-content">
    <h1>{title}</h1>
  </div>
</section>
```

Build the function that would:

- Take the `slide-img` map
- Return the markup shown

```
(dom/section #js {:className "slide"}
  (dom/img #js {:className "bg" :src (:bg model)}
  (dom/div #js {:className "slide-content"}
     (dom/h1 nil (:title model))))
```

Pull out the component

- `om/build` takes a component function and value

Code Review

- `git reset --hard`
- `git checkout first-slide`

Let's build 2 slides!

Where's the data?

```
(defonce app-state (atom {:slides []}))
```

```
(def slide-imgs
  [{:title "A Bird's Eye View of ClojureScript"
    :bg "/images/gull.jpg"}
   {:title "Using Om Components"
    :bg "/images/gull2.jpg"}])
```

```
(swap! app-state assoc :slides slide-imgs)
```

What should the DOM look like?

- Root always renders a single node
- `om/build-all` takes a component function and a sequence of values
- `apply`-ing a `dom/div` function to the result returns a div with child nodes

```
(apply dom/div #js {:className "content"}
   (om/build-all slide (:slides model)))
```

Code Review

- `git reset --hard`
- `git checkout multiple`

Events and Transactions

What's missing?

What if the app state knew which slide to display?

Exercise

Where's the data?

- Introduce an `:index` property
- Always render the one slide at that index

```
(defonce app-state (atom {:slides [] :index 0}))
```

```
(dom/div #js {:className "content"}
   (om/build slide ((:slides model) (:index mode
```

What happens when you `swap!` the index?

Code Review

- `git reset --hard`
- `git checkout index`

Let's create the nav components

What should the DOM look like?

```
<section class="content">
  <a href="" class="control banner previous"></a
  <div class="slide">
    <img src={bg} class="bg">
    <div class="slide-content banner">
      <h1>{title></h1>
    </div>
  </div>
  <a href="" class="control banner next"></a>
</section>
```

Adding an `:onClick` handler

- `git reset --hard`
- `git checkout empty-handler`

Adding an `:onClick` handler

```
(defn next-slide [e model f]
  (.preventDefault e)
  (js/alert (f (:index @model))))
```

```
(dom/a #js {:className "control banner previous"
            :href ""
            :onClick #(next-slide % model dec)})
```

```
(dom/a #js {:className "control banner next"
            :href ""
            :onClick #(next-slide % model inc)})
```

What happens when you click the links?

Code Review

- `git reset --hard`
- `git checkout working-handler`


Break

Client-side Routing

What's missing?

The Secretary Library

- Routes are functions
- `defroute` macro maps URL fragment to a set of params
- `defroute` also creates named route functions to construct the URL string
- `dispatch!` calls the appropriate route based on the fragment

Google Closure's History API

- `EventType.NAVIGATE` represents a navigation event
- The `token` property represents the URL fragment, can be get and set
- `dispatch!` calls the appropriate route based on the fragment

Exercise

Let's create a route!

- `git reset --hard`
- `git checkout empty-route`

Try this in the REPL:

```
(slide-path {:index 5})
```

```
(swap! app-state assoc :index 3)
```

```
(next-slide-path app-state dec)
```

Try implementing the `slide-path` route!

Code Review

- `git reset --hard`
- `git checkout working-route`

Interactivity

What's missing?

Understanding component local state

Who controls where the data changes?

Whose rendering is affected while the data is changing?

`render` **vs** `render-state`

- `render-state` receives the state of the backing Om component
- ...which can be used to change what is rendered
- `render` is static with respect to local state

Initial state

The default state a component renders with, e.g. default selection in a select box

`om/set-state!`

Used to update the local state of a component, e.g the value the user types in a search field

Exercise

Changing local state

- `git reset --hard`
- `git checkout empty-state`

Try implementing the `set-editing-state` function! Check out how the event handlers on the input field are implemented.

Code Review

- `git reset --hard`
- `git checkout working-state`


Thank You!
Backgrounds thanks to: Pexels and Pixabay