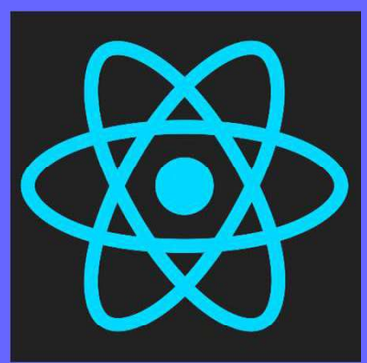
1

Introduction to Web Aspects



Frond End Applications Development

Brief history of Web Applications

- Initially: static HTML files only with HTML forms for input
- Common Gateway Interface (CGI)
- Certain URLs map to executable programs that generate web page
- Program exits after Web page complete
- Introduced the notion of stateless servers: each request independent, no state carried over from previous requests. (Made scale-out architectures easier)
- Perl typically used for writing CGI programs

First-generation web app frameworks

- Incorporate language runtime system directly into Web server
- **Templates**: mix code and HTML - HTML/CSS describes view
- Web-specific library packages:
  - URL handling
  - HTML generation
  - Sessions
  - Interfacing to databases

Examples: (PHP, ASP.net, Java servlets, JSP)

Second-generation frameworks

- **Model-view-controller**: stylized decomposition of applications
- Object-relational mapping (**ORM**): simplify the use of databases (make database tables and rows appear as classes and objects)
- Easier fetching of dynamic data

Examples: (JSF, Ruby on Rails, Django):

Third-generation frameworks

- JavaScript frameworks running in browser - More app-like web apps
  - Interactive, quick responding applications - Don't need server round-trip
- Frameworks not dependent on particular server-side capabilities
  - Node.js - Server side JavaScript
  - No-SQL database (e.g. MongoDB)
- Many of the concepts of previous generations carry forward
  - Model-view-controller
  - Templates - HTML/CSS view description

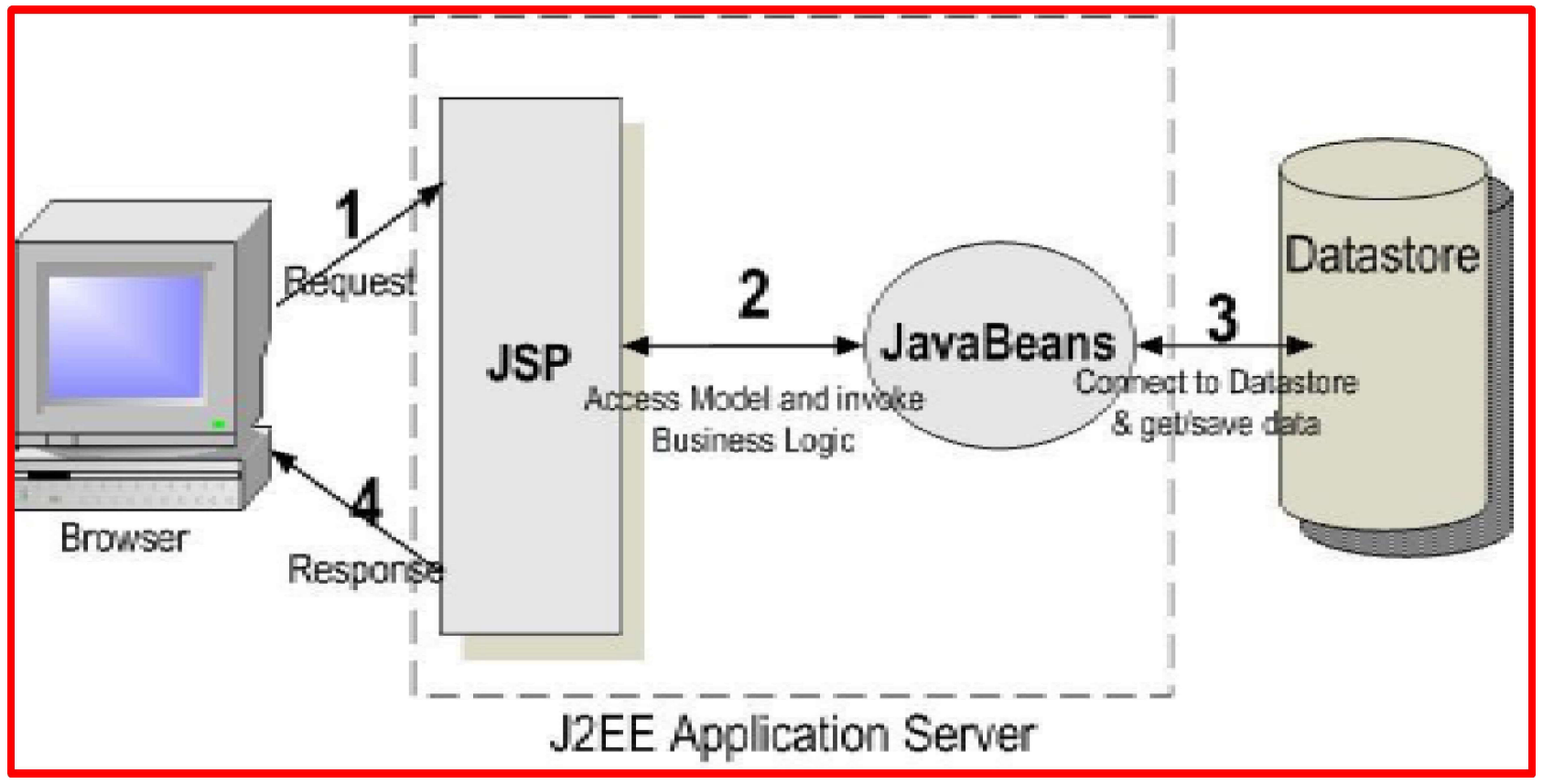
Example: Angular

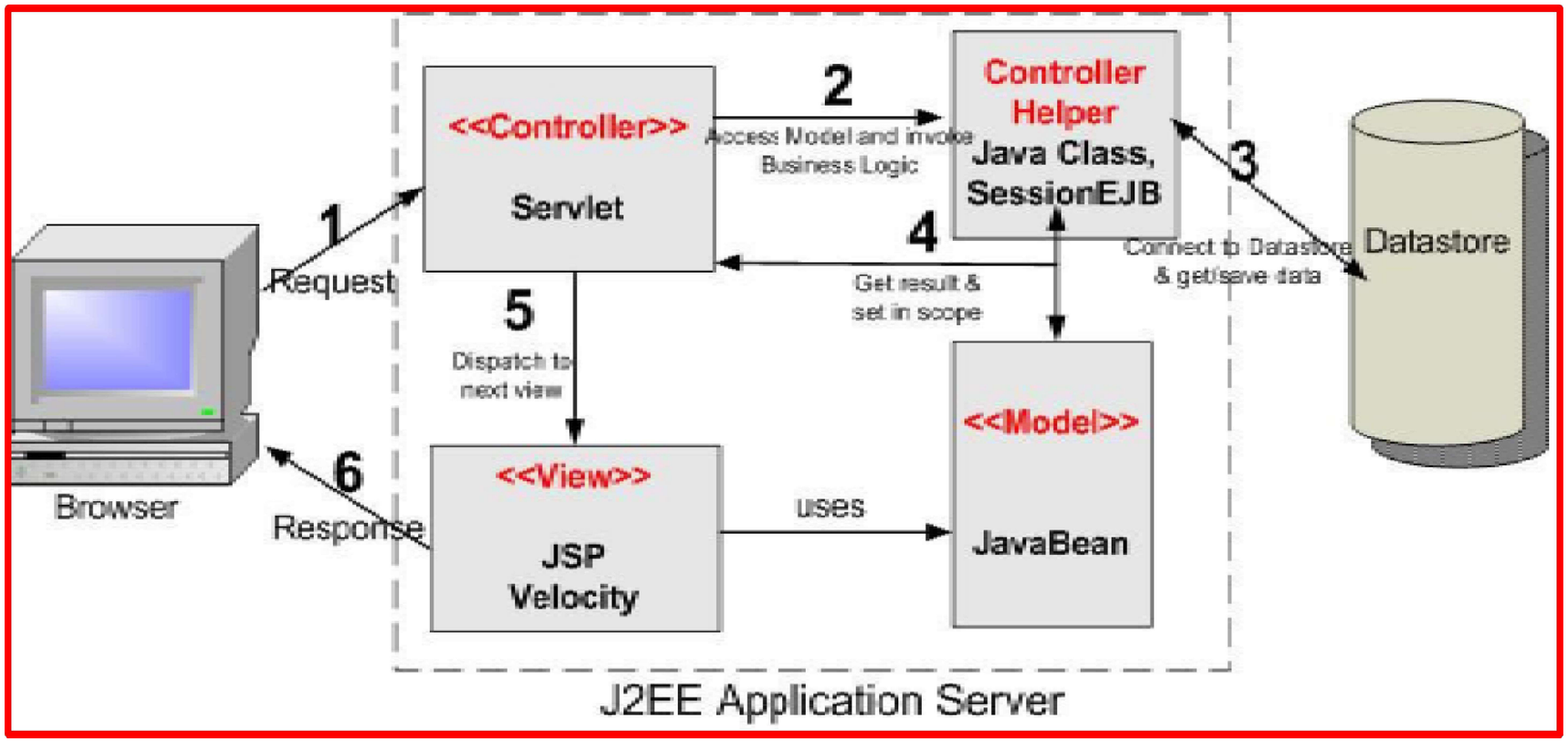
Model-View-Controller (MVC) Pattern

- **Model**: manages the application's data
  - JavaScript objects. Photo App: User names, pictures, comments, etc.
- **View**: what the web page looks like
  - HTML/CSS. Photo App: View Users, View photo with comments
- **Controller**: fetch models and control view, handle user interactions
  - JavaScript code. Photo App: DOM event handlers, web server communication

MVC pattern been around since the late 1970's

- Originally conceived in the Smalltalk project at Xerox PARC


1
Request
JSP
2
Access Model and invoke
Business Logic
JavaBeans
3
Connect to Datastore
& get/save data
Datastore
Browser
4
Response
J2EE Application Server


Controller
Helper
Java Class,
SessionEJB
2
Access Model and invoke
Business Logic
<<Controller>>
Servlet
1
Request
3
Connect to Datastore
& get/save data
Datastore
4
Get result &
set in scope
5
Dispatch to
next view
<<Model>>
JavaBean
Browser
6
Response
<<View>>
JSP
Velocity
uses
J2EE Application Server

View Generation

- Web App: Ultimately need to generate HTML and CSS
- **Templates** are commonly used technique. Basic ideas:
  - Write HTML document containing parts of the page that are always the same.
  - Add bits of code that generate the parts that are computed for each page.
  - The template is expanded by executing code snippets, substituting the results into the document.
- Benefits of templates (Compare with direct JavaScript to DOM programming)
  - Easy to visualize HTML structure
  - Easy to see how dynamic data fits in
  - Can do either on server or browser

AngularJS view template (HTML/CSS)

```
...
<body>
    <div class="greetings">
        Hello {{models.user.firstName}},
    </div>
    <div class="photocounts">
        You have {{models.photos.count}} photos to review.
    </div>
</body>
```

Angular has rich templating language (loops, conditions, subroutines, etc.). Later...

Controllers

- Third-generation: JavaScript running in browser Responsibilities:
- Connect models and views
  - Server communication: Fetch models, push updates
- Control view templates
  - Manage the view templates being shown
- Handle user interactions
  - Buttons, menus, and other interactive widgets

AngularJS controller (JavaScript function)

```
function userGreetingView ($scope, $modelService) {
    $scope.models = {};

    $scope.models.users = $modelService.fetch("users");
    $scope.models.photos = $modelService.fetch("photos");


    $scope.okPushed = function okPushed() {
        // Code for ok button pushing
    }
}
```

Angular creates $scope and calls controller function called when view is instantiated

Model Data

- All non-static information needed by the view templates or controllers
- Traditionally tied to application's database schema
  - Object Relational Mapping (ORM) - A model is a table row
- Web application's model data needs are specified by the view designers But need to be persisted by the database
- Conflict: Database Schemas don't like changing frequently but web application model data might (e.g. user will like this view better if we add … and lose ...)

Angular doesn't specify model data (JavaScript objs)

- Angular provides support for fetching data from a web server
  - REST APIs
  - JSON frequently used

On Server:

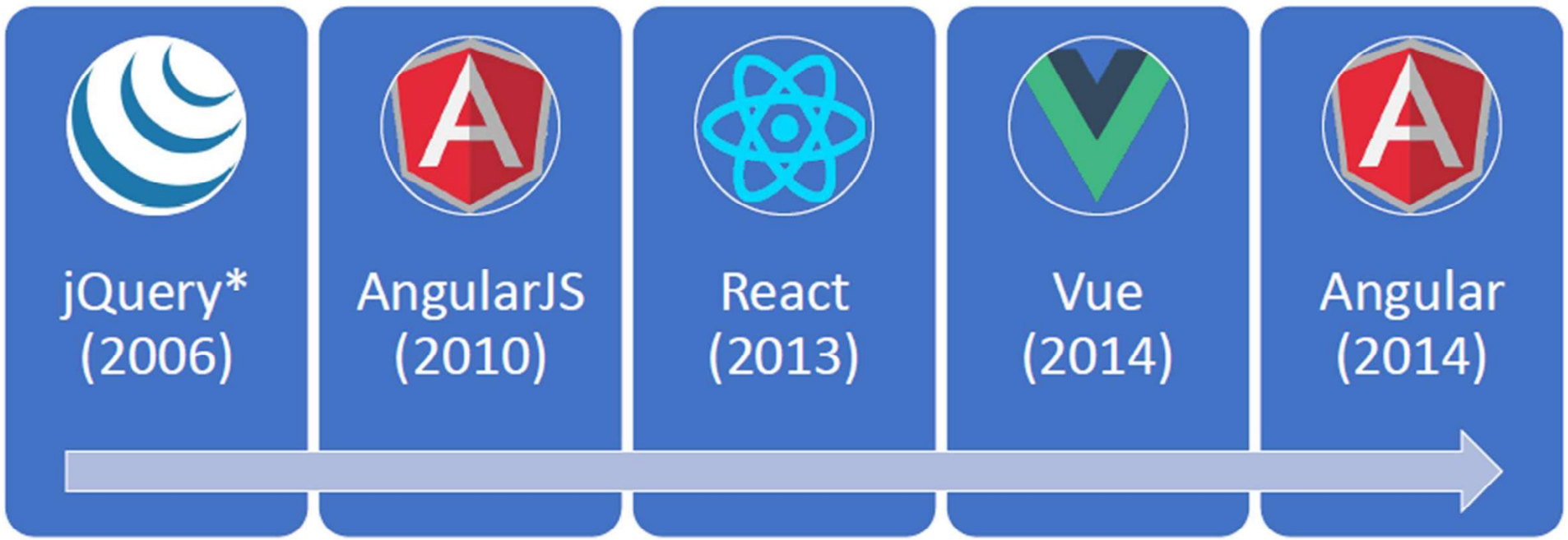
- Mongoose's Object Definition Language (ODL) has "models"

```
var userSchema = new Schema({
  firstName: String,
  lastName: String,
});
var User = mongoose.model('User', userSchema);
```

Fourth-generation frameworks

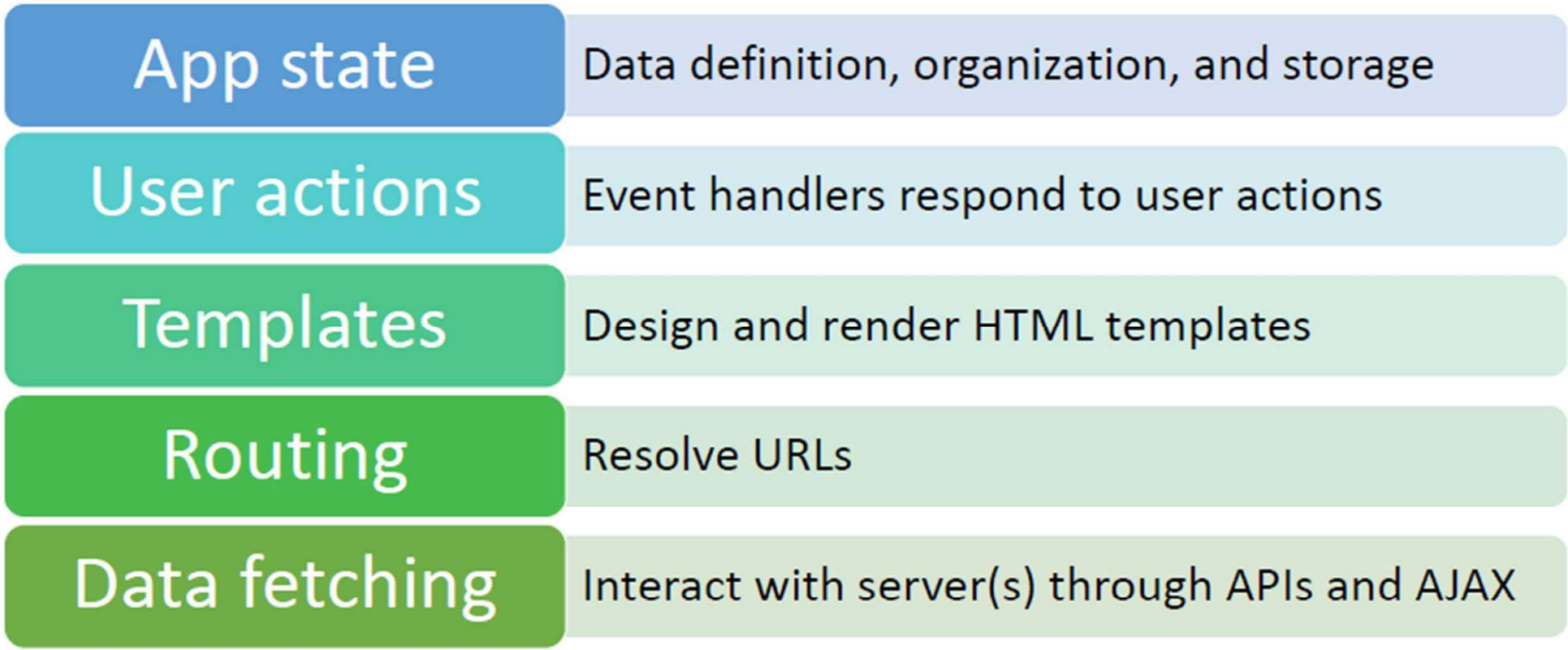
- Many of the concepts of previous generations carry forward
  - JavaScript in browser
  - Model-view-controllers
  - Templates
- Focus on JavaScript components rather than pages/HTML
  - Views apps as assembled reusable components rather than pages.
  - Software engineering focus: modular design, reusable components, testability, etc.
- Virtual DOM
  - Render view into DOM-like data structure (not real DOM)
  - Benefits: Performance, Server-side rendering, Native apps

Examples : Angular, React, OJET, Vue JS



* jQuery is more often considered a **library** than a **framework**

Common tasks in front-end development



Every Framework can be viewed as an attempt to say "The hardest part of writing a webapp is $X, so here's some code to make that Easier".

- **Knockout.js** is basically what you get when dev says, "The Harders part of writing a webapp is implementing 2-Way Binding", Which you can tell because that's basically 90% of what the Framework does with ease.
- **Similarly, Backbone** is the result of feeling like the hard parts are Fetching and Persisting Models to a REST API and Client side Routing; That's Basically all it Does with Ease

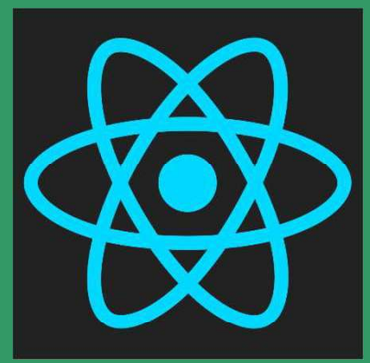
Cont …

- **Angular** is what you get if you think the biggest Problem with writing Webapps is that JavaScript isn't Java.
- **Ember** that's its not Ruby
- **React + Redux** is based on the idea that what's Really Hard with writing Webapps is Non-deterministic behavior and unclear Data Flow.

Everyone has their own ideas of what's hard to Solve.

Cont …

- **Why do Framework exist ?**
  - Keep state out of the DOM
  - Higher Level Abstractions
  - Code organization
- **Pros**
  - Common concepts that can be shared between apps and developers
  - Larger communities, Shared Knowledge, Documentation, Bug Fixes
  - Better App Structure through tools and guidelines
- **Cons**
  - Learning Curve
  - Minimum requirements for size
  - Setup and infrastructure

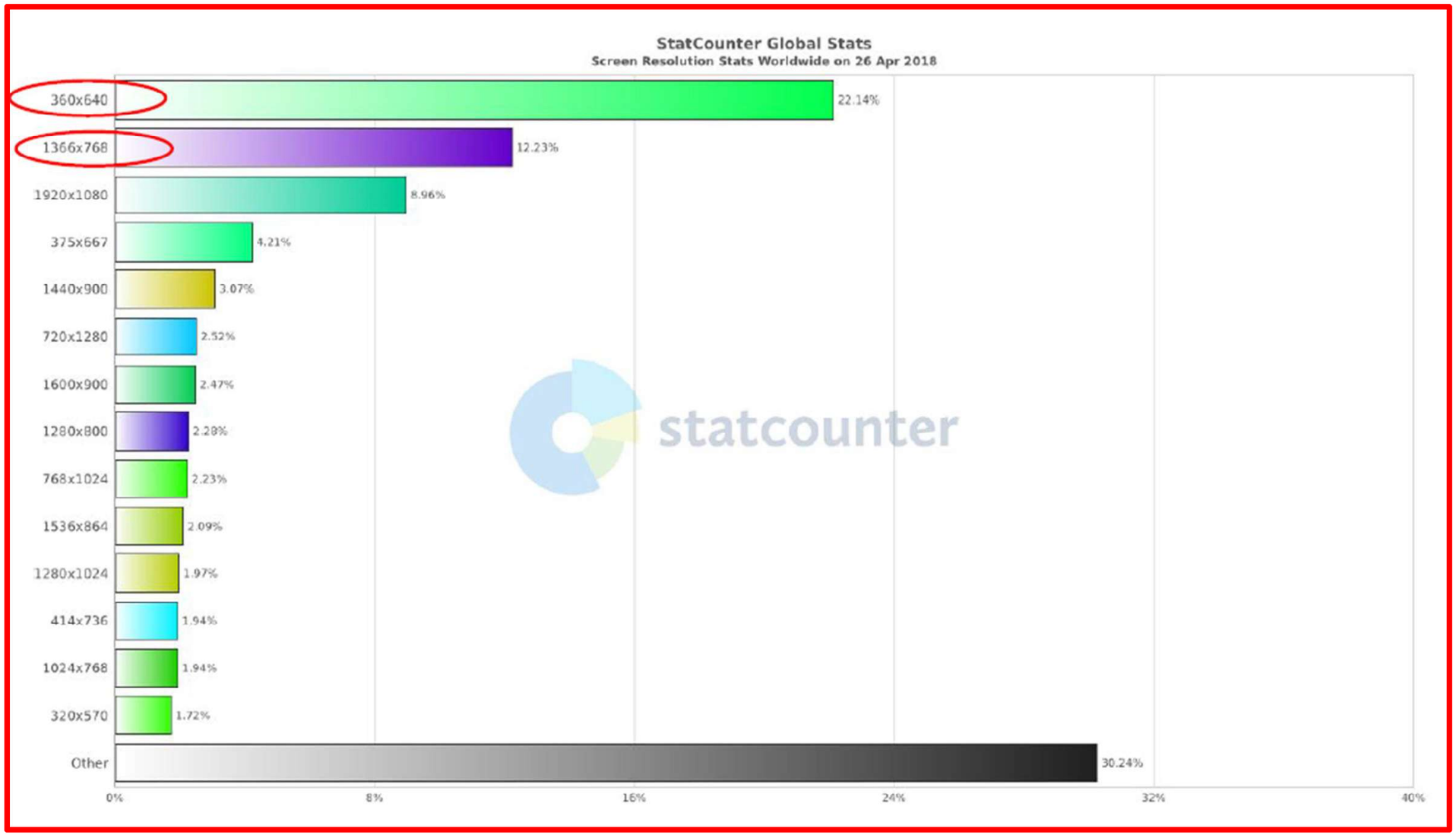


Reponsive Web Design

Web App Challenges: Screen real estate

- Do we need to build N versions of each web application?

Statistics

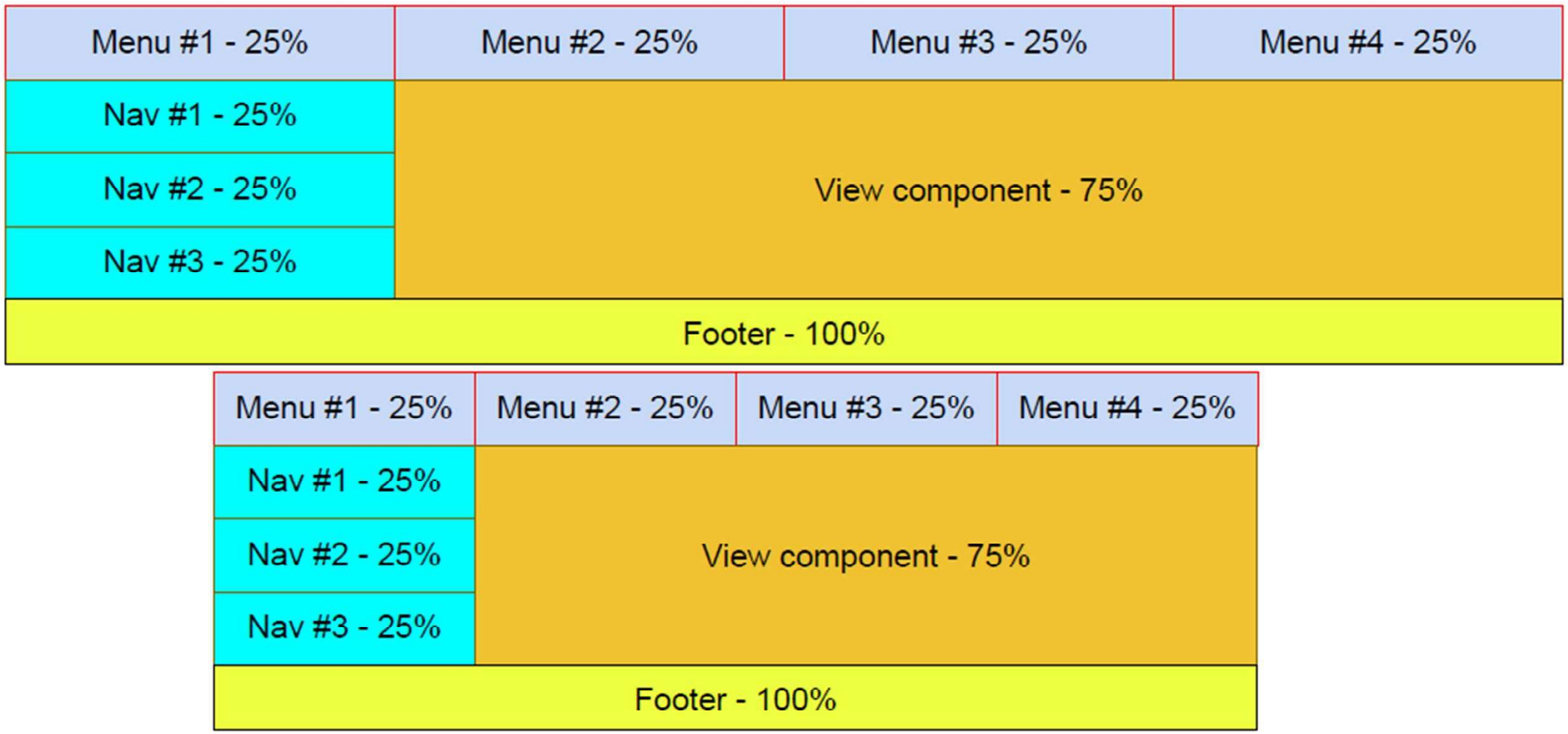


Responsive Web Design

- **Content is like Water !**
  - The Web app should flow into and fill whatever **Device** you have.
- **Possible With CSS Extensions:**
  - Add Grid Layout System with relative (Eg 50%) rather than absolute (Eg 50pt) Measures
    - Specify element packing into columns and rows
  - Add @media rules based on screen sizes
    - Switch layout based on Screen sizes
  - Made images support Relative Sizes
    - Autoscale image and videos to fit in screen region

```
img { width: 100%; height: auto; }
video { width: 100%; height: auto; }
```

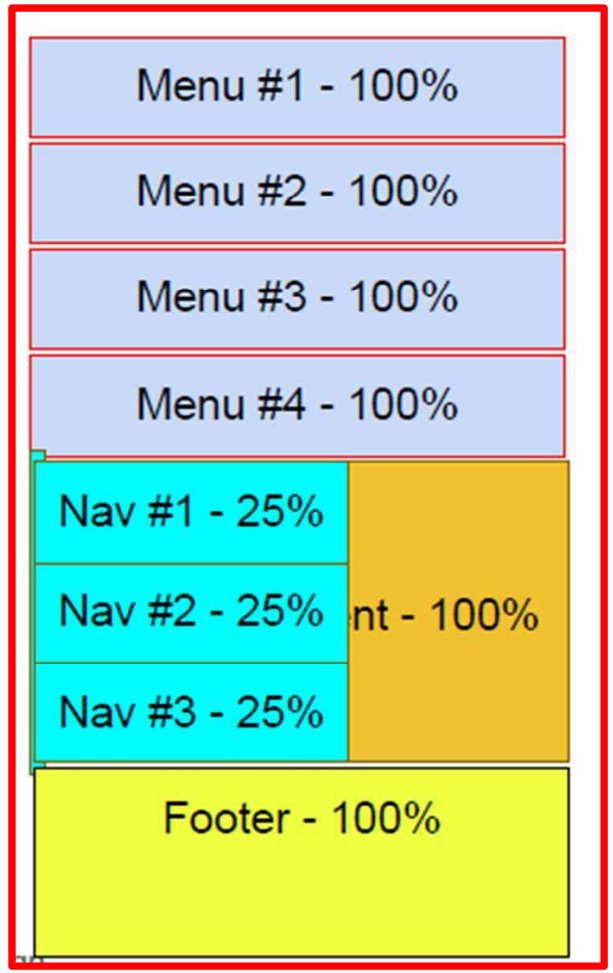
Example of Responsive Web Layout



CSS Breakpoints

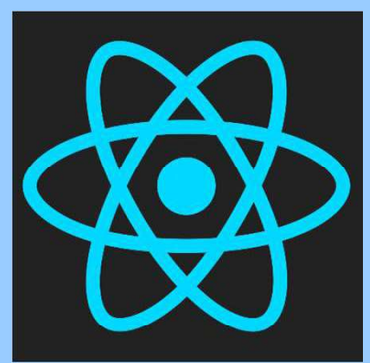
CSS Rules:

```
@media only screen and (min-width: 768px) {
     /* tablets and desktop layout */ }

@media only screen and (max-width: 767px) {
      /* phones */ }

@media only screen and (max-width: 767px)
     and (orientation: portrait) {
          /* portrait phones */ }
```



Responsive implementation

- Build components to operate at different screen sizes and densities
  - Use relative rather than absolute
  - Specify sizes in device independent units
- Use CSS breakpoints to control layout and functionality
  - Layout alternatives
  - App functionality conditional on available screen real estate
- Mobile first popular
  - Expand a good mobile design to use more real estate



Single Page Applications (SPA)

Web Apps and Browsers

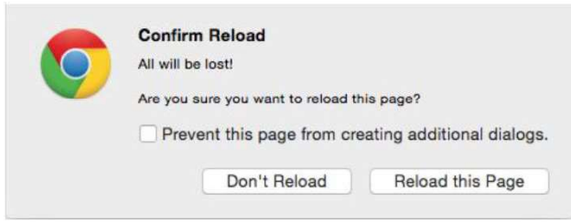
- Web Apps run in browsers (by definition)
- Users are use to browsing in browsers
  - Browser maintains a history of URLs visited
    - Back button - Go back in history to previous URL
    - Forward button - Go forward in history to next URL
  - Can move to a different page
    - Typing into location bar or forward/back buttons
    - Selecting a bookmarked URL
    - Page refresh operation
- Browser tosses the current JavaScript environment when navigating to a different page
  - Problematic for JavaScript frameworks: URL (and cookies) are the only information preserved

Problem with some web apps

- Initial: pages served from web server
  - Each page had a URL and app switched between pages served by web server
- Early JavaScript apps: Website a single page/URL with the JavaScript

  Problem: Restart web app on navigation (Can lose a lot of work!)

  window.onbeforeunload = function(e) { return 'All will be lost!'; }



- Users expect app in browser to do the right thing:
  - Navigate with forward and back buttons, browser page history
  - Navigate away and come back to the app
  - Bookmark a place in the app
  - Copy the URL from the location bar and share it with someone
  - Push the page refresh button on the browser

Changing URL without page refresh

- Can change hash fragment in URL without reload

  http://example.com

  http://example.com#fragment

  http://example.com?id=3535

  http://example.com?id=3535#fragment

- HTML5 give JavaScript control of page reload
  - Browser History API - window.history - Change URL without reloading page

Deep linking

- Concept: the URL should capture the web app's context so that directing the browser to the URL will result the app's execution to that context
  - Bookmarks
  - Sharing
- Context is defined by the user interface designer!

  Consider: Viewing information of entity and have an edit dialog open

  Should the link point to the entity view or to the entity & dialog?

  Does it matter if I'm bookmarking for self or sharing with others?

  How about navigating away and back or browser refresh?

Deep linking in Single Page Apps

Two approaches:

1. Maintain the app's context state in the URL
   + Works for browser navigation and refresh
   + User can copy URL from location bar
2. Provide a share button to generate deep linking URL
   + Allows user to explicitly fetch a URL based on need
   + Can keep URL in location bar pretty

Either way web app needs to be able to initialize self from deep linked URL

Ugly URLs

```
http://www.example.org/dirmod?sid=789AB8&type=gen&mod=Core+Pag
es&gid=A6CD4967199
```

versus

```
http://www.example.org/show/A6CD4967199
```

What is that ugly thing in the location bar above my beautiful web application?

```
https://www.flickr.com/photos/jarnasen/24593000826/in/explore-2016-01-26/
```

ReactJS support for SPA

- ReactJS has no opinion! Need 3rd party module.
- Example: React Router Version 5 https://v5.reactrouter.com/
  - Idea: Use URL to control conditional rendering
  - Newer version 6 is available using same concepts as v5 but slightly different syntax
- Various ways of encoding information in URL
  - In fragment part of the URL: `HashRouter`
  - Use HTML5 URL handler: `BrowserRouter`
- Import as a module:

```
import {HashRouter, Route, Link, Redirect} from 'react-router-dom';
```

Example React Router V5

```
<HashRouter>
   <div>
     ...
     <Route path="/states" component={States} />
     ...
     <Link to="/states">States</Link>
      ...
    </div>
 </HashRouter>
```

- JSX block controlled by URL enclosed in `HashRouter`
- `Route` will render the component if URL matches.
- Use `Link` component to generated hyperlink:

```
<a href="#/states">
  States
</a>
```

Passing parameters with React Router

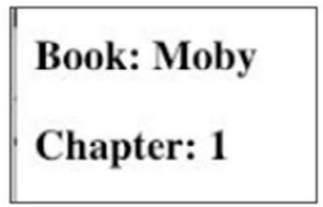
- Parameter passing in URL

```
<Route
    path="/Book/:book/ch/:chapter"
    component={BookChapterComponent}
/>
```

- Parameters put in prop.match of the component

```
function BookChapterComponent({ match }) {
  return ( <div>
             <h3>Book: {match.params.book}</h3>
             <h3>Chapter: {match.params.chapter}</h3>
           </div> );
}
```

```
<Link to="/Book/Moby/ch/1">
    Moby
</Link>
```



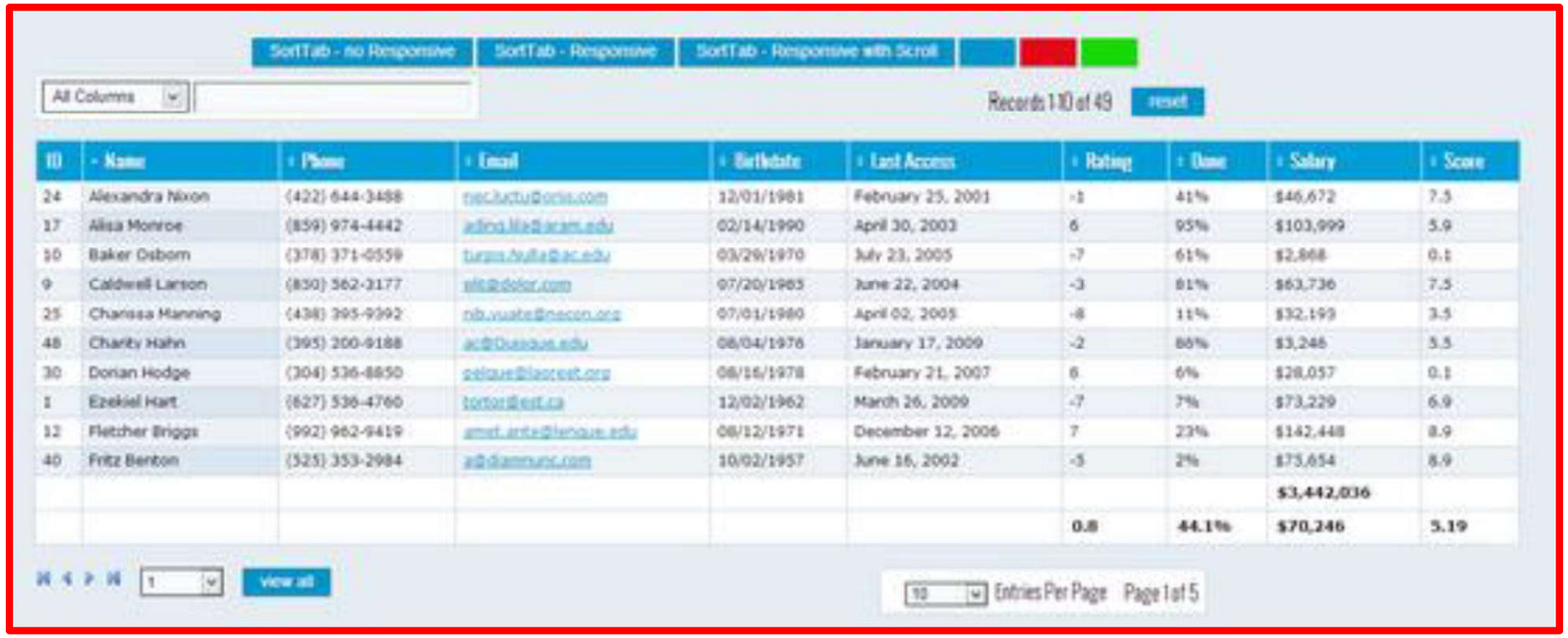
Route: component=, render=, children=

- `component={BookChapterComponent}`
  - Mounts components on match (unmounts on URL change)
  - Passes `match` object with: `params`, `url`, `history`
- `render={props => <BookChapterComponent book={props.match.params.book} chapter={props.match.params.chapter} />}`
  - Calls function with props having `match` object from above.
  - Doesn't mount/unmount component (does update it)
- `children=` - Like render= except is called regardless of the match
  - match will be `null` if URL doesn't match
  - Useful if you want to have something always render but only active on matching URL.

Multiple route matches have precedence order: component, render, children

`Switch` is useful with multiple Route - Renders the first matching one

Example



- What to keep in URL: table length, viewport in table, search box, sort column, etc.
- Is it different for bookmark or share? Nav away and back?

Example: Not everything goes in URL