ROCA RESOURCE-ORIENTED CLIENT ARCHITECTURE

A collection of simple recommendations for decent Web application frontends

Introduction

A Web application's architecture is heavily influenced by the design decisions, both implicit and explicit, that have been made by framework developers. Sometimes these decisions are consciously accepted as being in line with the intended overall system architecture. More often, though, they are accepted simply because developers assume they embody the state of the art of development practices.

ROCA is an attempt to define a set of recommendations — independent of any particular framework, programming language, or tooling — that embodies the principles of what we consider to be good web application architecture. Its purpose is to serve as a reference, one that can be implemented as-is or be compared to other approaches to highlight diverging design decisions.

ROCA splits into two parts: The server-side and the client-side architecture. The server-side consists of RESTful backends, serving human-readable content as well as services for machine-to-machine communication, either public or internal. The client-side focuses on a sustainable and maintainable usage of JavaScript and CSS, based on the principle of Progressive Enhancement. This technique is pursued by nearly every basic web technology, e.g. HTML or HTTP. Client and server are largely independent from, yet complement each other.

Server-side

1. The server application adheres to REST principles, i.e. it exposes a set of resources that are meaningful to a user sitting in front of a browser, each resource has its own URI, all of the information necessary for handling a request is contained within the request itself, HTTP methods are used in line with their definition, and the resource state is maintained by the server (stateless communication).
#REST

2. All application logic resides on the server.

#APPLICATION-LOGIC

3. The client interacts with the server through RESTful HTTP requests.

#HTTP

4. A user must be able to link to a specific piece of information, e.g. by copying the address from the browser's address bar and pasting it into an e-mail, creating a bookmark, or using any of the fancier ways to share URIs.

#LINK

5. It must be possible to use the server's logic through user agents other than a browser, e.g. a command-line client such as curl or wget.

#NON-BROWSER

6. Resources have additional representations in other formats, e.g. JSON and/or XML.

#SHOULD-FORMATS

7. All authenticated communication relies on HTTP Basic or Digest Authentication, typically combined with SSL, possibly with client certificates. Alternatively, because of the limits of browser-native authentication (e.g. no logout, no styling), form-based authentication in conjunction with cookies can be used. If cookies are used, they should include all of the state needed for the server to process them, and another authentication mechanism should be supported for non-browser access.

#AUTH

8. Cookies may not be used for purposes other than authentication, user tracking or security purposes like CSRF protection.

#COOKIES

9. There may not be any session state beyond what's needed for simple algorithmic validation of authentication information.

#SESSION

10. The browser controls like the back, forward and refresh buttons must work as expected. I.e. the back button should take the users where they expect to be taken to (the last meaningful resource they worked with). A browser refresh should not cause a re-rendering of the login or home page instead of the page the user was looking at, or a (to the user) unexpected question about wanting to submit the same data again (when the user doesn't recall submitting any data, indicating a mis-use of the POST verb).

#BROWSER-CONTROLS

Client-side

1. The server returns structured semantic HTML markup that is independent of layout information and client behavior.

#POSH

2. It must be possible to access each page's information and functionality by using accessibility tools like screen readers.

#ACCESSIBILITY

3. CSS is used for formatting and layout. This is done following the principles of progressive enhancement, e.g. to allow a browser not capable of CSS3 features still to use a CSS3-based site.

#IDIOMATIC-CSS

4. In line with the principles of progressive enhancement, JavaScript is used unobtrusively and the application remains usable (albeit with a decrease in usability and convenience) if JavaScript is disabled.

#UNOBTRUSIVE-JAVASCRIPT

5. The same functionality must not be implemented redundantly on both the client (JavaScript) and the server. Thus, due to the application logic requirement, application logic must not reside on the client-side.

#NO-DUPLICATION

6. The server code may not "know" the HTML structures the client code generates (beyond CSS) or vice versa. Exceptions are some well defined HTML structures the server generates to initialize the client functionality above.

#KNOW-STRUCTURE

7. All JavaScript code and CSS code must be static, and must not be dynamically generated by the server in a form specific to the resource requested. (Note that this does not prohibit the use of preprocessors like CoffeeScript or LESS, as the respective code is usually pre-compiled as part of the release process.)

#STATIC-ASSETS

8. Any dynamic routing or URI state modification triggered by JavaScript on the client side should use the HTML5 History API.

#HISTORYAPI