Modern Application Development - II

Review of MAD-I

- What is an app (at least in our context):
  - An application or program used for interacting with a computing system
  - Allow the user to perform some tasks useful to them
- Components:
  - Backend: Store data, processing logic, relation between data elements etc.
  - Frontend: User-facing views, abstract for machine interaction
  - Naturally implies a client-server or request-response type of architecture
- Why Web:
  - Close to universal platform with clear client-server architecture
  - Low barrier to entry - trivial to implement simple pages and interaction
  - High degree of flexibility - possible to implement highly complex systems

Review of the Web Application Development Model

- **Presentation** - HTML for semantic content, CSS for styling
- **Logic** - Backend logic highly flexible - we used Python with Flask
- ***Application architecture***:
  - Model - View - Controller. Good compromise between understandability and flexibility
- ***System architecture***:
  - REST principles + sessions - how to build stateful applications over stateless protocols
  - **APIs**: separate data from view
  - RESTful? APIs - useful for basic understanding, but not strict adherence to REST
- Others:
  - Security, Validation, Logins and RBAC, database and frontend choices etc.

Moving Forward

- Advanced Frontend Development
  - Exploring JavaScript and how to use it
  - JavaScript, APIs, Markup - the JAMStack
  - VueJS as a candidate frontend framework
- Other topics of interest
  - Asynchronous messaging, Email
  - Mobile / Standalone apps, PWA/SPA
  - Performance measurement, benchmarking, optimization
  - Alternatives to REST
  - etc.

JavaScript

- A little history
- Implications of origins
- Writing and Running code
  - Terminal
  - DOM
- References

Origins

- Originally created in 1995 as scripting language for Netscape Navigator
- Intended as “glue” language
  - Stick modules from other languages together
  - Not really meant for much code
- Primarily meant to assist “applets” in Java - hence JavaScript
  - Trademark issues, name changes, …
- Issues:
  - Slow
  - Limited capability

Power

- Glue was useful, but…
- Then Google Maps, Google Suggest etc. (~ 2005)
  - Pan around map, zoom in/out seamlessly - fluid user interface:
  - Load only what is needed!
- Described and Named Ajax - Garrett 2005
  - Asynchronous Javascript and XML
- Allowed true “web applications” that behave like desktop applications
- Evolved considerably since then
  - Much more on this approach moving forward

Standardization

- Move beyond Netscape needed
- ECMA (European Computer Manufacturers Association) - standard 262
- Subsequently called ECMAScript
  - Avoid trademark issues with Java
- In practice:
  - Language standard called ECMAScript (versions)
  - Implementation and use: JavaScript
- Significant changes in ES6 - 2015
  - Yearly releases since then
  - “Feature readiness” oriented approach

What version to use?

- ES6 has most features of modern languages (modules, scoping, class etc.)
- Some older browsers may not support all of this
- Possible approaches:
  - Ignore old browsers and ask user to upgrade
  - Package browser along with application - useful for limited cases, like VSCode (Electron apps)
  - Polyfills: include libraries that emulate newer functionality for older browsers
  - Compilers: BabelJS - convert new code to older compatible versions
- Backend
  - Node.js, Deno: directly use JS as a scripting language like Python

Implications of JS Origins

- Ease of use given priority over performance (to start with)
- Highly tolerant of errors - fail silently
  - Debugging difficult!
  - Strict mode: “use strict”;
- Ambiguous syntax variants
  - Automatic semicolon insertion
  - Object literals vs Code blocks {}
  - Function: statement or expression? Impacts parsing
- Limited IO support: errors “logged” to “console”
- Closely integrates with presentation layer: DOM APIs
- Asynchronous processing and the Event Loop - very powerful!

Using JS

- Not originally meant for direct scripting
  - Usually not run from command line like Python for instance
- Need HTML file to load the JS as a script
  - Requires browser to serve the files
  - Links and script tags etc.
  - *May* not directly work when loaded as a file
- NodeJS allows execution from command line

Examples here will be shown on Replit for the most part

DOM

- Document Object Model
  - Structure of the document shown on the browser
- DOM can be manipulated through JS APIs
- One of the most powerful aspects of JavaScript
- **Input**: clicks, textboxes, mouseover, …
- **Output**: text, colours/styles, drawing, ...

References

- JavaScript for impatient programmers - exploringjs.com
  - detailed reference material, focused on language, not frontend or GUI - very up to date
- Mozilla Developer Network
  - several examples and compatibility hints
- Learn JavaScript Online - interactive tutorial

Utilities

- BabelJS - Compiler converts new JS to older compatible forms
- JS Console - interactive console to try out code
- Replit - complete application development

JavaScript Syntax

- Program structure, Comments
- Identifiers, Statements, Expressions
- Data Types and Variables
  - Strings and Templates
- Control Flow
- Functions

Basic Frontend Usage

- Frontend JavaScript: must be invoked from HTML page
  - In context of a “Document”
  - will not execute if loaded directly
- Scripting language - no compilation step
- Loosely structured - no specific header, body etc.

Identifiers - the words of the language

Reserved words:

await break case catch class const continue debugger default delete do else export extends finally for function if import in instanceof let new return static super switch this throw try typeof var void while with yield

Literals (values)

true false null

Others to avoid

enum implements package protected interface private public Infinity NaN undefined async

Statements and Expressions

- Statement:
  - Piece of code that can be executed

```
if ( … ) {
    // do something
}
```

Standalone operation or side effects

- Expression
  - Piece of code that can be executed to obtain a value to be returned

```
x = 10;
“Hello world”
```

Anywhere you need a “value” - function argument, math expression etc.

Data Types

- Primitive data types: built into the language
  - undefined, null, boolean, number, string (+ bigint, symbol)
- Objects:
  - Compound pieces of data
- Functions
  - Can be handled like objects
  - Objects can have functions: methods
  - Functions can have objects???

Strings

- Source code is expected to be in Unicode
  - most engines expect UTF-16 encoding
- String functions like length can give surprising results on non-ASCII words
- Can have variables in other languages!
  - But best avoided… keep readability in mind

Non-Values

- undefined
  - Usually implies not initialized
  - Default unknown state
- null
  - Explicitly set to a non-value

Very similar and may be used interchangeably in most places -

Keep context in mind when using for clarity of code

Operators and Comparisons

- Addition, subtraction etc.
  - Numbers, Strings
- Coercion
  - Convert to similar type where operation is defined
  - Can lead to problems - needs care
- Comparison:
  - Loose equality: == tries to coerce
  - Strict equality: === no coercion
- Important for iteration, conditions

Variables and Scoping

- Any non-reserved identifier can be used as a “placeholder” or “variable”
- Scope:
  - Should the variable be visible everywhere in all scripts or only in a specific area?
  - Namespaces and limiting scope
- `let`, `const` are used for declaring variables
  - Unlike Python, variables MUST be declared
  - Unlike C, their type need NOT be declared
  - `var` was originally used for declaring variables, but has function level scope - avoid

let and const

- `const` : declares an immutable object
  - Value cannot be changed once assigned
  - But only within scope
- `let`: variable that can be updated
  - index variable in for loops
  - general variables

Control Flow

- Conditional execution:
  - if , else
- Iteration
  - for , while
- Change in flow
  - break, continue
- Choice
  - switch

Functions

- Reusable block of code
- Can take parameters or arguments and perform computation
- Functions are themselves objects that can be assigned!

Function Notation

Regular declaration

```
function add(x, y)
{
  return x + y;
}
// Statement
```

Named variable

```
let add =
function(x, y) {
  return x + y;
}
// Expression
```

Arrow function

```
let add =
(x, y) => x + y;


// Expression
```

Anonymous functions and IIFEs

```
let x = function () { return “hello”} // Anonymous bound


(function () { return “hello” }())    // Declare and invoke
```

Why? Older JS versions did not have good scoping rules.

- Avoid IIFEs in modern code - poor readability

DOM API

- User Interaction
- Examples

Interaction

- console.log is very limited
  - variants for error logging etc.
  - But mostly useful only for limited form of debugging - not production use
- But JS was designed for document manipulation!
- Inputs from DOM: mouse, text, clicks
- Outputs to DOM: manipulation of text, colours etc.