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.

Modern Application Development - II

JavaScript Collections

- Arrays
- Synchronous Iteration
- Multi-dimensional
- Maps, Sets, …
- Destructuring
- Generators

Basic Arrays

- Collection of objects of any type
  - Can even be mixed type (numbers, strings, objects, functions…)
- Element access
- Length
- Holes
- Iteration

Iteration

- Go over all elements in a collection
- Concepts:
  - **Iterable**: an object whose contents can be accessed sequentially
  - **Iterator**: pointer to the next element
- Iterable objects:
  - Array
  - String
  - Map
  - Set
  - Browser DOM - tree structure
- Objects: object.keys(), object.entries() - helper functions

Iterations and Transformations

- Functions that take functions as input
- `map, filter, find`
  - Apply a callback function over each element of array
- Elements of functional programming: create a transformation chain

Callback: important concept - function passed in to another function, to be called back for some purpose

Other Collections

- Maps: proper dictionaries instead of objects
- WeakMaps
- Sets

More advanced topics - use only if needed

Destructuring

- Simple syntax to split an array into multiple variables
- Easier to pass and collect arguments etc.
- Also possible for objects

Generators

- Functions that `yield` values one at a time
- Computed iterables
- Dynamically generate iterators
- Advanced topic - skip for now

Modularity

- Modules
- Objects
  - Prototype based inheritance

Modules

- Collect related functions, objects, values together
- “export” values for use by other scripts
- “import” values from other scripts, packages

Ways of implementing

- script - direct include script inside browser
- CommonJS - introduced for server side modules
  - synchronous load: server blocks till module loaded
- AMD - asynchronous module definition
  - browser side modules

ECMAScript 6 and above:

- ES6 Modules
  - Both servers and browsers
  - Asynchronous load

npm

- Node Package Manager
- Node:
  - command line interface for JS
  - Mainly used for backend code, can also be used for testing
- npm can also be used to package modules for frontend
  - “Bundle” managers - webpack, rollup etc.

Objects

- Everything is an object …
- Object literals
  - Assign values to named parameters in object
- Object methods
  - Assign functions that can be called on object
- Special variable `this`
- Function methods
  - call(), apply(), bind()
- Object.keys(), values(), entries()
  - use as dictionary
  - iterators

Prototype based inheritance

- Object can have a “prototype”
- Automatically get properties of parent
- Single inheritance track

Class

- Better syntax - still prototype based inheritance
- constructor must explicitly call super()
- Multiple inheritance or Mixins
    - Complex to implement - out of scope here

Asynchrony

- Asynchronous calls
- Asynchronous Iteration
- Basic ideas of Promises
  - async/await

Function calls

- Function is like a “branch”
  - but must save present state so we can return
- Call stack:
  - Keep track of chain of functions called up to now
  - Pop back up out of stack

- main() on stack - current - calls f()
- f() goes on stack - calls g()
- g() goes on stack - calls h()
- h() goes on stack - executes
- return from h -> pop into g
- return from g -> pop into f
- return from f -> pop into main

Call Stack

Explanatory video:

https://vimeo.com/96425312

Visualizing the call stack:

http://latentflip.com/loupe/

Event Loop and Task Queue

- Task Queue: store next task to execute
  - Tasks are pushed into queue by events (clicks, input, network etc.)
- Event loop:
  - Wait for call stack to become empty
  - Pop task out of queue and push it onto stack, start executing
- Run-to-completion
  - Guarantee from JavaScript runtime
  - Each task will run to completion before next task is invoked

Blocking the browser

https://exploringjs.com/impatient-js/ch_async-js.html#how-to-avoid-blocking-the-javascript-process

Why callbacks?

- Long running code
  - Will block execution till it finishes!
- Push long running code into a separate “thread” or “task”
  - Let main code proceed
  - Call back when completed

Example: reading files - synchronous

```
const fs = require('fs')

try {
 const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
 console.log(data)
} catch (err) {
 console.error(err)
}
```

Example: reading files - asynchronous

```
const fs = require('fs')

fs.readFile('/Users/joe/test.txt', 'utf8' , (err, data) => {
 if (err) {
   console.error(err)
   return
 }
 console.log(data)
})
```

Asynchronous Code

- Very powerful - allows JS to have high performance even though it is single threaded
- Can be difficult to comprehend
  - Focus on using async libraries and functions before writing your own
- Promises, async function calls etc.
  - Important and useful concepts
  - Deferred for now

JSON

JSON

- Object notation - for serialization, communication
- Notation is frozen
  - Means even problem cases will remain (trailing “,” etc could be useful but will not be used)
- Usage through JSON API

JSON API

- Global namespace object `JSON`
- Main methods:
  - `JSON.stringify()`
  - `JSON.parse()`

Frontend implementation

What is the frontend?

- User-facing part of app
  - User Interface (UI) and User Experience (UX)

What is the frontend?

- User-facing part of app
  - User Interface (UI) and User Experience (UX)
- Requirements
  - Avoid complex logic - application logic should be in backend
  - No data storage
  - Work with stateless nature of HTTP

What is the frontend?

- User-facing part of app
  - User Interface (UI) and User Experience (UX)
- Requirements
  - Avoid complex logic - application logic should be in backend
  - No data storage
  - Work with stateless nature of HTTP
- Desirable
  - Aesthetically pleasing
  - Responsive - no lag / latency
  - Adaptive - different screens

Programming Styles

- Imperative: sequence of actions to achieve final result
  - Draw boxes for navigation, main text, fill in text, wait for clicks etc.
  - Functions for each step, composition of functions
- Declarative: specify desired result
  - Compiler / Interpreter knows how to achieve result
  - Function integration automated

Credit: Flutter documentation “Start thinking declaratively”

State?

- Internal details of the system: memory
- Reproducibility
  - Given a “system state”, the system should always respond the same way to input
- Complexity
  - Any non-trivial application needs internal state

System State

- Complete database of amazon.in, flipkart.com
  - Stocks of available items, prices, logged in/registered users etc.
- All news articles ever published on toi.com, thehindu.com, bbc.com
- All students, courses, marks, certificates etc. for NPTEL

Typically huge, but comprehensive

- Completely independent of the user interface / frontend!

Application State

Application:

- System as seen by an individual user / session
- Includes interactivity, session management

Examples:

- Shopping cart, user preferences, theme
- Followed news items, recommendations
- Dashboard displays

UI State (Ephemeral State)

UI

- Part of application actually seen / interacted with
- Ephemeral - “lasting for a very short time” (term used by Flutter)

Examples:

- Loading icons
- Currently selected tab in multi-tab document / page

Application and UI management

- HTTP is stateless
- How to convey state between client and server?
  - Client maintains state - sends requests to server for specific items
  - Server maintains state - only specific requests allowed to client

Example: Tic-Tac-Toe

- What to display on screen?
- Who determines the display?
- How should user input be collected and processed?

Vue

Declarative Rendering

- Reactivity
- Directives
- Event handling
- Class and Style binding
- Lists and Loops

Reactivity

- Auto-update in response to changes in data
- Binding between model (data) and view (display)

Focus on **What** instead of **How**

Why?

- User interaction is fundamentally reactive
  - Respond to changes in input
- Change in one parameter may need multiple updates on screen
  - User logs in
  - Update navigation: add logged in links
  - Update displayed list of courses
  - Change colours based on theme

How?

- Server tracks *state*
  - user logged in?
  - date/time?
- Server responds with complete HTML based on present state
  - Capture different layouts, visibility in views
  - Render appropriately for each user
  - Fully server-side
- Client-JS
  - Login controller retrieves user model
  - Client side JS goes through each element and updates as needed (jQuery, vanilla JS)

Vue - directives

- v-bind: one way binding - update variable, reflects on display
  - v-model: two way binding - inputs, checkboxes etc: form data
- v-on: event binding

Class binding

- Dynamically modify class of an element
- Special support for bind - object
- Multiple classes attached based on key-values
    - If value for a given key is true, that key is applied as a class or style

Conditional rendering

- `v-if="argument"`
  - what follows is shown when argument evaluates to true
  - JS based - changes DOM
- `v-show="param"`
  - only if the show parameter evaluates to true
  - always rendered and present in DOM - only CSS display parameter changed

Looping

- Iterate over any JS iterable
  - Array, Object, String etc.
- Usage similar to Jinja {{}} templates
- Examples:
  - `v-for="item in items"` // items is an array
  - `v-for="value in obj"` // obj is an object
  - `v-for="(value, name) in obj"` // name -> key: “key” has another meaning in v-for
  - `v-for="(value, name, index) in obj"` // index -> numerical index

Loop Keys

- Looping “creates” new DOM elements
  - Vue needs some way to keep track of elements if updating needed
  - Virtual DOM “diffing” used to find what changes to reflect on screen
  - Vue uses heuristics to see what can be minimized
- For simple updates, simple heuristics are sufficient
  - For more complex updates, may not be easy to track what has “changed” or what is “new”
- Provide a “key”
  - Key must be unique for each loop element
  - Updating item with same key will automatically update only relevant items
- Rule of thumb: provide a key where possible - index, ID, …

ViewModel

Model / View

- Model:
  - The “data” of the application
  - Usually stored on server in database
- View:
  - Displayed to end user (or to non-human consumer)
  - Rendering of data

Problem: sometimes *view* needs more info than needed for *model*, or perhaps there is derived information

Example

- Form with username, password, repeat_password
  - Should repeat_password be in model? Where should it be stored? How should it be used?
- Page with top comments, top posts
  - Should top_comments, top_posts be in model?
  - Should they be derived from other data?
  - What does the displayed information correspond to?

Example credits: https://www.c-sharpcorner.com/UploadFile/abhikumarvatsa/what-is-model-and-viewmodel-in-mvc-pattern/

ViewModel

- Yet another “Pattern”
- Create model constructs with additional data / derived data
- “bind” to view
  - Auto update view on change in data
  - (possibly) auto update data when changed in view
- Why?
  - cleaner code

Vue - ViewModel - Vue Instance

“Although not strictly associated with the MVVM pattern, Vue’s design was partly inspired by it.”

- Vue documentation

- Data binding

“data” of Vue instance is the “instance data”

- Update to data will be reflected in the view
- Update in the view *can* change the data

MVC vs MVVM

- NOT either-MVC-or-MVVM
- Controller:
  - Convey actions to model
  - Call appropriate view based on inputs and model
- ViewModel:
  - Create framework for data binding
  - Can still use controllers to invoke actions

Computed Properties

- Often need to work with *derived* data
  - Take original data and modify it according to some function / logic
  - Examples: Boolean on-off on styles depending on values in data; navigation links depending on history
- Each time the source data changes, update the derived data

Computed properties

- Auto-update
- *Cached* based on their *reactive dependencies*

Computed “setter” also possible - see documentation for example

Watcher

- Explicitly look for changes
- Can be used for imperative code
- For more complex logic than just updating a property

When possible, use computed property instead of watcher

- more *declarative*
- caching

Components

Reuse

- DRY principle: Don’t Repeat Yourself
- Examples:
  - News items on IITM front page
  - “People also bought” items on Amazon
- Same structure, formatting, repeated

Refactor

- Change code without changing functionality
- Mainly for readability and maintainability - not functionality

Vue Component Structure

- Properties:
  - passed down from parent - customize each instance
- Data:
  - individual data of the present instance
  - Also it’s own watchers, computed properties etc.
- Template:
  - how to render
  - render functions possible - see docs
  - *Slots*

Templates

- {{}} format - similar to Jinja
- Safety features:
  - will not interpolate text into tags - why?
  - errors on unclosed divs etc
- More complex render functions possible
  - JSX: mix JS + HTML - similar to React

Slot

- Main element of text:
  - use like regular tag
- Properties can be defined in tag

Reactivity

How does reactivity work?

- Need to track when data is
  - accessed
  - modified
- Add methods to objects
  - Everything in JS is an object!

Object.defineProperty()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

Examples from: https://blog.logrocket.com/your-guide-to-reactivity-in-vue-js/

```
const data = {
    count: 10
};

const newData = {

}

Object.defineProperty(newData, 'count', {
  get() { return data.count; },
  set(newValue) { data.count = newValue; },
});

console.log(newData.count); // 10

newData.count = 20;
```

```
const data = {
    count: 10
};

const newData = {

}

function track(){
  console.log("Prop accessed")
}
function trigger(){
  console.log("Prop modified")
}
Object.defineProperty(newData, 'count', {
  get() {track();return data.count; },
  set(newValue) { data.count = newValue;trigger(); },
});

console.log(newData.count);
// Prop accessed
// 10

newData.count = 20;
// Prop modified
```