MyFurnitureSPA_Steps

My-Furniture SPA Process

1. Extract project skeleton and examine files
2. Setuip NPM project (package.json) + libraries
3. Analyze the HTML example templates & determine what templates should be created
-- Go through existing html templates to get the idea how your SPA should look
-- Look for a header/footer which will be present in all pages
-- Look for the main container of the page where the main content will be created
-- Create a "Sample" folder to store all sample HTML
4. Use npm init -y to create package-json
5. Use npm i --save page lit-html (to install the libraries)
6. If tests are present (install playwright, mocha and chai as well)
7. Configure routing using page and placeholder modules (a js file for each view)
-- views folder (all view modules) !catalog and dashboard are usually the same views (check, tho)
-- src folder (app.js + api.js + data.js)
8. Implement requests (Api.js + Data.js)
9. Implement views

APP JS
1. Import page
2. Routing
3. For each view, create an exportable async viewHandler function which to import in app.js
-- each view handler can then access the context from page (app.js)
-- for each view handler try a console.log to see if routing works
4. page.start() to start routing

Router tests
1. Create an index.html
2. Import app js as module script
View Controller a.k.a (handler) (module) role:
- fetch data
- interpolate templates
- handle user input
- return content / render content
-- a universal rendering function can be used (no import in each view needed)

Api js - universal requests
1. Make sure it's located in the Api folder, a subfolder of src
2. Create a universal query module
3. Create a utility function that assembles the necessary options
4. Create four main CRUD functions: get, put, post, delete

-- Export them
-- Import them in app.js
-- Set them to Window.api to test them
-- After server is on (see 5.) in the console test a GET request -> await api.get(url)
5. Start server using terminal
-- use "cd ..." until reaching directory containing server
-- node server.js
6. Create Login, Register & Logout in the api.js
-- at line 1 create "export const settings = {host:''}
-- set host in the app.js
-- use settings.host in the url part of login and register
-- Login & Register: set sessionStorage (email, authToken, userID)
-- Logout: get request, no body, remove items from storage
7. Create data.js which will import api and be imported by all view controllers
-- Good practice if api changes
-- data.js in api folder
-- Import everything from api.js
  -- export only logout,register and login for later use
  -- change api import path in app.js to the newly available data.js
-- Set host
8. Create application-specific requests in data.js
-- these are usually requests connected to each view
-- examples:

- Create Furniture (POST): http://localhost:3030/data/catalog
- All Furniture (GET): http://localhost:3030/data/catalog
- Furniture Details (GET): http://localhost:3030/data/catalog/:id
- Update Furniture (PUT): http://localhost:3030/data/catalog/:id
- Delete Furniture (DELETE): http://localhost:3030/data/catalog/:id
- My Furniture (GET): http://localhost:3030/data/catalog?where=_ownerId%3D%22{userId}%22

9. Create views (viewControllers)
-- Create a static template (w/o data or requests at first)
-- Implement requests for each view
-- Add parameters to the template (use data from the requests)
-- Add event listeners if needed
10. Copy homepage template to the index.html and remove "dynamic" html leaving a container in which html will be rendered
-- initialize script (app.js) in index.html if removed by copying
-- remove . before / in paths for scripts/css

11. Dashboard
-- Copy html elements corresponding to this view
-- Import lit-html
-- Create lit html template function
!NB Create middleware function in app.js
-- route -> middleware -> view controllers

-- middleware inherits Context from route so selectors can be used by render globally
-- Import the needed request function from data.js
12. Await the request function in the main function of the page
-- Store the data retrieved from the request
-- Utilize the data within the template
-- Create a subtemplate if mapping of the data is needed
-- If there's a link (a href), use routing e.g. /details/${item._id}
13. Modify index.html links (navigation) so they work as routing is intended to
-- Logout is an action, so javascript:void (0) and later - an eventListener

14. Create Register view
-- Go to Sample html of each and take whatever html elements the main container has
-- Import lit-html
-- Create html template
-- Render a static page to make sure it all works
-- Add eventListener to form using @submit=${onSubmit}, make sure template gets async handler as a parameter; preventDefault!
-- Use FormData to get all fields
-- Validate fields
-- Import register function from api/data
-- Await register(email,password) -> function handles requests and errors and sessionStorage
-- If async handler is not in the main function, insert it so context is open
-- Use context to redirect - ctx.page.redirect('/')
-- Test alerts, registration and redirect
15. Fancy Validation with css style changes
-- Template will now receive 3 boolean parameters - invalidEmail, invalidPass, invalidRe
-- If parameters are False, add additional class
-- Additional class should look like -> ${'form-control' + (invalidEmail? ' is-invalid' : '')} // note that there's a space before the class
-- In the validation of all fields, execute render, adding to its parameters the validation itself ->
ctx.render(registerTemplate(onSubmit,email=='' || password=='' || repass==''))
-- '' -> will be either true or false and will execute function correctly
-- For password validation, call render and mark both password fields as true for invalid ->
ctx.render(registerTemplate(onSubmit,false,true,true))
16. Fancy error message
-- Set errorMsg as param of template
-- Set a div that will only appear if there is an errorMsg -> ${errorMsg? html `
${errorMsg} ` : ''}
-- Instead of alerting in onSubmit function, set alert message in registerTemplate render as the first param - errorMsg

17. Create Login view
-- Import html and login function (from api/data.js)
-- Copy necessary html elements from sample and use them in a template
-- Event Listener on form (make sure function is called in template aswell.)
-- Prevent default, formData, extract fields, validate, trim just in case
-- await login function, redirect using ctx.page.redirect

18. Tackle navigation in app.js
-- Check for userId from sessionStorage
-- If not null -> display user nav, if null -> display guest
-- Call SetUserNav before page.start() so application has an initial menu view
-- Set as part of context so it's available to login & register
-- Rename middleware function to decorateContext
  -- add ctx.setUserNav = setUserNav (available via context)
-- apply ctx.setUserNav in login and register
19. Create Logout --> in app.js
-- Import logout froom api/data
-- Select logoutBt and addEventListener -> async
-- Call logout function
-- Redirect, SetUserNav

20. Details view
-- Import getItemById in details.js (to aquire item via ctx.params)
-- Import lit-html
-- Create template, function receives Item as param
-- Get template divs from sample htmls
-- Populate template with item specifics
-- Edit links
  -- if "Edit" -> use /edit/${item._id}
  -- if "Delete" -> javascript:void(0) and eventListener later on
-- Get item -> getItemById(ctx.params.id)
-- Use render with template using the item to call template with
-- Note: if images don't load, check relative paths for ./

21. Delete functionality for a single item
-- Add event listener + create async function
-- In template add isOwner as param and use ${isOwner?} to add condition for visibility of Edit/Delete buttons
-- When calling ctx.render with template, add "sessionStorage.getItem('userId') == item._ownerId" to check isOwner
  -- Make sure to add funciton as parameter to template and render
-- Import delete function from data.js
-- Create a confirm dialog to check if confirmed -> redirect if confirmed

22. Edit functionality -> edit.js
-- Import html, editRecord, getItemById
-- Get html sample elements and paste them into a template
-- Get id from ctx.params.id
-- Get item using await getItemById
-- Create eventListener for form (@submit) -> add it to template params (and to template call in render)
-- FormData -> get all fields (use get for each field or use reduce)
-- Make sure to exclude optional fields in validation
-- Populate form fields using .value= ${item....} -> for each property
-- Await editRecord to send request with edited data
-- Redirect

23. Craete functionality -> create.js
-- Import createRecord and html from libraries
-- Copy/Paste from Edit functionality for relevant logic
  -- All logic is valid except for item id (getting from params and finding item by its Id)
  -- Rename template and create it using sample html
-- Add form event listener
-- Template only takes onSubmit form event listener

24. My Furniture view -> myFurniture.js
-- Such vies look a lot like catalog/dashboard (reuse code)
-- Import html and the specific request function (getMyFurniture)
-- Create template that takes data, copy html elements from sample
-- If a template is repeated across several views, it can be created separately and imported
  -- create a Common folder in views for such templates (e.g. Item template)
  -- move such a template from dashboard to common, make it exportable
-- Import item template (both in dashboard and in current view)
-- In current view use array.map to render each item via the imported Item template