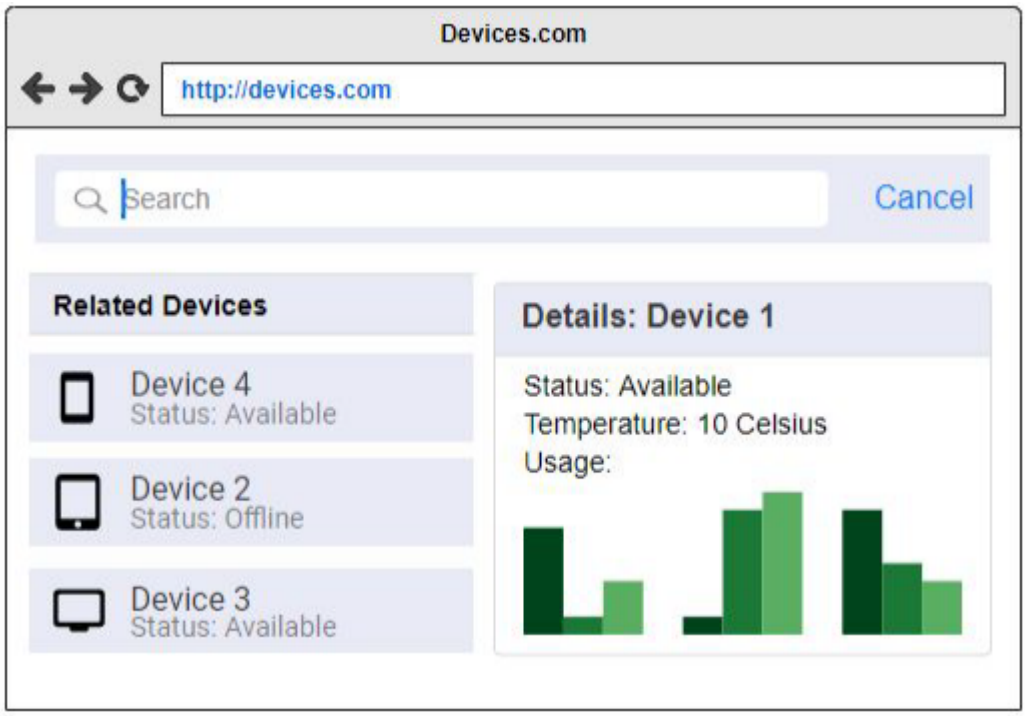
1VALET Coding Exercise

Cavan MacPhail



Function Requirements

1. There would need to be List<Devices> filtered from the database based on the UserID which is currently logged in. This would populate the related devices. The default sort would be based on usage of the devices. (Can be handled in the SQL query to pull the data).
2. The UI would require the ability to have a mode for selected, similar to a radio button set. This would cause only a single selection at a time.
3. The selected device would populate the right device display. This requires the Device to have a few derived properties from a based class Device.
    a. Device Name -> Detail: {string DeviceName}
    b. Status -> Status: {enum [On, Off]}
    c. Mode -> Mode: {enum [Enabled, Disabled]}
    d. Usage -> This can be tracked based on the mode and status. Timestamps and usage calculation can then be derived.
4. Other properties of the Devices can be unique to an inherited class {Thermostat, Camera, Intercom, etc..}
5. The search bar at the top would be used to Query the different devices attached to a user account.

API Endpoints

User Registation/Authentication

UsersController

GET: api/Users
Task<ActionResult<IEnumerable<User>>> GetUsers()
    Get a list of all the users in the database.

GET: api/Users/{id}
Task<ActionResult<User>> GetUser(int id)
    Get the user defined by the user_id.

POST: api/Users/{user}
Task<ActionResult<User>> RegisterUser(User user)
    Register a user to the database using the provided json body to build a user entity.

POST: api/Users/Authenticate
Task<IActionResult> Authenticate([FromBody] AuthenticateModel model)
    Authenticate the current user.

Device CRUD

GET: api/Devices
Task<ActionResult<IEnumerable<Device>>> GetDevices()
    Get all the devices in the database

GET: api/Devices/User/{id}
Task<ActionResult<IEnumerable<Device>>> GetDevices(int id)
    Get all devices assigned to the user_id

GET: api/Devices/{id}
Task<ActionResult<Device>> GetDevice(int id)
    Get the device with {id}, this would be used for the selected state.

PUT: api/Devices/{id}
    Update all properties of the device in the database with device_id

POST: api/Devices
Task<ActionResult<Device>> PostDevice(Device device)
    Add a new device to the database.

DELETE: api/Devices/{id}
Task<IActionResult> DeleteDevice(int id)
Delete the device with device_id in the table.

PATCH: api/Devices/{id}/Mode
Task<IActionResult> UpdateDeviceMode(int id, [FromBody] string mode)
Update the Mode of the device with device_id

PATCH: api/Devices/{id}/Status
Task<IActionResult> UpdateDeviceMode(int id, [FromBody] string status)
Update the Status of the device with device_id

DB Tables

Table - Users

user_Id (PK)	Username	Password	First_Name	Last_Name	Email
int/Guid	string	string	string	string	string

Table - Devices

device_id (PK)	User_id (FK)	name	status	mode
int/Guid	int/Guid	string	string	string

Questions:

1. I have used both styles before, and I feel that they both serve good purposes. I do prefer the first style as I feel it has more personal ownership of the task.
2. Overall I prefer Style 1, but I do see the merit in style 2 as it puts you in the shoes of a customer.
3. For tests I would:
    a. Test device return values - status, modes, Display Icons. I would generally look to check likely, and unlikely results however I can fit in the possible unlikely result.
    b. I would test the endpoint being called for a full set of details. Looking to pass in the correct and incorrect data for routing. Just to be sure things fail and pass when they should.
    c. I would add time logging into my tests beginning to end of complex calls or queries. Knowing where the test and intern function is taking time can help for better optomiziona.