1. Look at package.json
    a. “main” tells us where to look next: server.js
    b. “dependencies” tells us what to expect
2. Look at server.js
    a. I see a few things ‘out of the ordinary’:
        i. var db = require("./models");
            1. I make a mental note to look at /models next
        ii. require("./routes/api-routes.js")(app);
            1. After models, I will take a look at this
3. Look in /models
    a. index.js is boilerplate Sequelize
    b. todo.js
        i. I see that it is exporting a Todo model that is defining a ‘Todos’ table with two columns
            1. text
            2. complete
4. Now I want to see where this model is being used. Based on what I saw in server.js, I know it will be used in api-routes.js. Andale!
    a. Well, what do you know? We require here ./models which imports my Todo model
    b. I see that these are my routes and there are four HTTP request methods, each listening to /api/todos
        i. app.get uses .findAll to return all records in our database
        ii. app.post creates a new record in our database
        iii. app.delete destroys a record based on its primary key id
        iv. app.put updates a record
    c. But where are these used? I haven’t looked in the ‘public’ directory yet. To the Batmobile!
5. In public
    a. index.html
        i. Because we’re working with POST, DELETE and PUT requests, I know there’s a Form somewhere in here. So I scroll until I find it.
        ii. Yup. There it is. But it doesn’t tell me much.
        iii. But! I see there’s a JavaScript file imported at the bottom of our HTML body. Away we go!
    b. view.js
        i. It’s the motherlode of JQuery! This is where the magic is happening.
        ii. I can now map JQuery .on() events to their functions

---

1. From the client-side http://localhost:8080/
    a. I see a form with placeholder text and a button to ‘Add Todo’

b. index.html
    i. I find the form and see that it has an id ‘todo-form’ and the <input> has a class of ‘new-item’
    ii. I also see that we are loading a JavaScript file, view.js
c. view.js
    i. I search for ‘new-item’ to see when and where it is used in the script
        1. It’s getting passed to $newItemInput
            a. The $ tells me this is a JQuery variable
    ii. I then search for ‘todo-form’
        1. I see that we have a JQuery listener for our “submit” button that will take the contents of our form and call the insertTodo function
    iii. I then search for insertTodo
        1. I see that the function is creating an object, todo, with property/value pairs and then sending a $.post request to /api/todos
    iv. Before I go look at that, I see that our $.post request is being passed another argument. Is it a function? Is it a variable? I search to find out
        1. getTodos is a callback
        2. After $.post posts to the database, it then calls getTodos which makes a $.get request of the api and calls initializeRows()
    v. I want to know what initializeRows() does, so I search for it
        1. I see that it is emptying out the $todoContainer that I remember seeing above
        2. I see that we are declaring an empty array, then iterating over the todos we grabbed with our $.get request and prepending them to $todoContainer
        3. I also see createNewRow and I want to understand how that works, so…
    vi. I search for createNewRow and see
        1. We are passing it an argument, ‘todo’, which I surmise is our database data
        2. I see that, using JQuery, we are creating a new HTML element, $newInputRow, that has a list item with a text input and two buttons, delete and complete
        3. I see that we are then using JQuery .find() to find the delete button and then using JQuery .data() to assign it an id that corresponds with our database id
        4. I also see that we have the ability to edit our todo items and that the we are using .find() and .css() to set the display to none
        5. …
        6. I see there’s a conditional for the complete, which I know is a database column from our insertTodo() function
        7. I’m curious about how all of this works, so I go back to my browser

        a. And enter a todo ‘Learn Sequelize’
        b. I get the results I expect, but I want to look under the hood, so I open up the Inspector
    8. A quick scan shows me that there is more functionality here, but I don’t want it to distract me from my initial quest, the HTTP requests
2. From the server side
    a. I know the HTTP requests are routes, so I open routes/api-routes.js
    b. First, I see we are requiring ./models and passing it to a variable db
    c. Then I see four routes
        i. get
        ii. post
        iii. delete
        iv. put
    d. I’m interested in .get and .post specifically
        i. app.get
            1. The server is listening for a get request, so when it hears that request made from view.js (via index.html) it calls db.Todo.findAll({})
            2. I know this is a Sequelize method, so I switch over to ./models to see what is happening
            3. ./models
                a. There are two files
                    i. index.js
                    ii. todo.js
                b. I know that index.js is boilerplate Sequelize, so I open...
                c. todo.js
                    i. I see we are creating a Sequelize model, Todo, with two columns
                        1. text
                        2. complete
                    ii. This lines up with what I discovered on index.html and view.js
            4. Back over at api-routes.js
                a. I see that when a get request is made to the server, it will return everything in the database as JSON
        ii. app.post
            1. The server is listening for a POST request. When it hears one, it calls our Sequelize model via db.Todo and uses the create() method to post a new record to our database inserting text and complete into the columns we saw in ./models/todo.js
            2. I also recall from ./models/todo.js that when a new POST is created, the default value of complete is false. I can surmise that

when the complete button is clicked, the POST request value will then be true
3. Cool beans.

3. Back to the client-side
    a. I have those two buttons
        i. checkmark
        ii. x
    b. view.js
        i. I see
            1. “button.delete”
            2. “button.complete”
        ii. Each is associated with a function
            1. deleteTodo
            2. toggleComplete
        iii. I search for deleteTodo
            1. First thing I see is event.stopPropagation();
            2. I forget exactly what that does, so I RTFM
                a. https://api.jquery.com/event.stoppropagation/
            3. JQuery docs suck, so I Google it
                a. https://www.w3schools.com/jquery/event_stoppropagation.asp
                b. This stops our button from triggering events that it is nested inside (bubbling)
            4. The rest is straightforward, I see that we are grabbing the id from the button and passing it to an .ajax call using the DELETE method
            5. After it deletes the record associated with the id, it calls getTodos
        iv. Next I search toggleComplete
            1. I see eventPropagation again, check
            2. Next we declare a variable, todo and we assign it the data from the parent element, which is a list item
            3. Then we assign todo.complete its opposite value
                a. If complete is true before we click the button
                    i. Then its new value is false
                b. If complete is false before we click the button
                    i. Then its new value is true
            4. The we call updateTodo and pass it our todo item
        v. Next I search updateTodo
            1. I see that it is an .ajax PUT method putting the data passed to it to /api/todos
            2. After it PUTs the data, it calls getTodos
    c. There’s some additional functionality I want to inspect, so back to the browser
        i. When I click on the todo item, I can edit it

1. I also notice that my buttons disappear

ii. Back in view.js I see
    1. $(document).on("click", ".todo-item", editTodo);
    2. $(document).on("keyup", ".todo-item", finishEdit);
    3. $(document).on("blur", ".todo-item", cancelEdit);

iii. I search for editTodo
    1. We load the data associated with this item in a variable, currentTodo
    2. Then we hide it's children (the buttons)
    3. We then load the text from data into the field
    4. And we show the CSS that we hid earlier with display: none
    5. We set its state to focus()
        a. I forget exactly how that works, so I RTFM
        b. https://www.w3schools.com/jquery/event_focus.asp

iv. Next I search for finishEdit
    1. It's very similar to editTodo
    2. The conditional statement is listening for a keyEvent
        a. This one is 13, which maps to the Enter key
            i. I know this because I Googled it
                1. https://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes
    3. If Enter is pressed, then we update our database
    4. We then remove focus using .blur()
        a. RTFM https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_event_blur_alert
    5. We then call updateTodo, which PUTs the edit in our database

v. Lastly, I search for cancelEdit
    1. This is a combination of editTodo and finishEdit
    2. If I start editing my todo and click outside the field before pressing Enter, the original text will be restored