**MERN Stack Front to Back**

**MongoDB Setup**

1. Go to **cloud.mongodb.com** and sign up for an account to use **MongoDB Atlas** cloud database services
2. Click on the **CONTEXT** dropdown button next to the Atlas logo and press the **New Project** button to create a new MongoDB project
3. Create a database by clicking on the **Build a Database** button
4. Select a database type in the **Path Selector** menu
   1. Use the **Shared** free **Cluster** database for learning and experimenting
   2. Choose AWS as the cloud provider and select the closest region
   3. Click on the **Cluster Name** tab and provide a name for the database
   4. Click on the **Create Cluster** button
5. Choose an authentication method and add a database user
   1. **\***The first user will have permission to read and write any data in the project
   2. Select the **Username and Password** option to add a new user with a username and password
      1. **\*\***Creds for course: **Admin/20notches**
   3. Select the **My Local Environment** option in the **Connect From** menu to allow IP address **Whitelisting**
      1. Only whitelisted IP addresses may access the database
   4. Add IP addresses to the whitelist to permit them access
      1. Click the **Add My Current IP Address** button to add the current IP you’re using
6. Once the database is created, press the **CONNECT** button (inside the database’s panel) to access the **Connection String** for the database
   1. Select **Connect your application** as the connection method
   2. Choose **Node.js** as the driver at the desired version
   3. Click on the **Browse Collections** button to view the database’s collections (data tables)

**Express Setup**

1. To start an Express project:
   1. Create a folder and name it the name of the project (i.e. “devconnector”)
   2. Type **npm init** in a bash window inside the project folder
      1. Follow the npm prompts and provide the necessary information
         1. Use **server.js** as the file name of the **entry point**
      2. This will create a **package.json** file for the project containing the settings entered in the init prompts
   3. Use **yarn add** to add the following packages:
      1. **express** – used to instantiate an express server
      2. **express-validator** – used to validate data in express
      3. **bcryptjs** – used for password encryption
      4. **config** – used to create global variables
      5. **gravatar** – used to access online (GitHub) user avatars
      6. **jsonwebtoken** – used for auth token validation
      7. **mongoose** – A document relational mapping framework for MongoDB
      8. **request** – Allows an express server to make HTTP requests to another API
   4. Use the command **yarn add -D** to add the following packages to the project’s **devDependencies**:
      1. **nodemon** – Used to refresh the express server with latest code changes upon save
      2. **concurrently** – Allows for multiple dev servers (i.e. express and react servers) to run at the same time with a single command
   5. Create a **server.js** file as the main entry point of the express server
   6. To setup an active express server in the server.js file:
      1. Require the ‘express’ library and assign it to a const
      2. Call **express()** and assign the returned object to an **app** constant
      3. Create a **PORT** constant and assign it to **process.env.PORT** OR a desired port number
      4. Call **app.listen()** to initiate the express server and listen for incoming requests
         1. Pass in the PORT variable as the first parameter
         2. Pass in a function to execute (i.e. console.log()) once the express server has started as the second parameter
      5. Ex.:

**const express = require('express');**

**const app = express();**

**const PORT = process.env.PORT || 5000**

**app.listen(PORT, () => console.log('Server started on port ${PORT}'));**

1. Call the command **node [entry\_point\_file],** where entry\_point\_file is the name of the server file (i.e. server.js), in command line to invoke the express server and have it listen for incoming requests
   1. The .js extension at the end of the file name is optional
2. In a dev environment, use the command **nodemon [entry\_point\_file]** to invoke a nodemon server that will update the server’s code every time a file has changed
3. In the **package.json** file, edit the **“scripts”** property adding:
   1. A **“start”** script that invokes the express server in a production environment
      1. Use the command line command node [entry\_point\_file], in quotes, for production
   2. A **“server”** script that invokes the express server in a development environment
      1. Use the command nodemon [entry\_point\_file], in quotes, for development

**Connecting to MongoDB with Mongoose**

1. Use the **config** npm package to create global values that can be used though out the express application:
   1. In the root folder of the project, create a **config** folder
   2. In the config folder, create a new file called **default.json**
      1. This will store all the default values for the app
      2. Create a new JSON object and add a property called **“mongoURI”**
         1. Set this property equal to the connection string of the MongoDB used for the application
         2. If using an Atlas connection string, replace **<password>** with the actual password of the connecting user
   3. Create another file named **db.js** to establish the MongoDB connection:
      1. Require **mongoose** – a **Document Relational Mapper** used to connect to a MongoDB database and create schemas
      2. Require **config** to retrieve configuration values
      3. Declare a const called **db** and use the config.**get()** function to retrieve values from the default.json file
         1. Use this function to get the mongoURI value from the default.json file
      4. Use the **mongoose.connect()** function to instantiate a connection to the database
         1. This method is asynchronous
         2. Pass in db as the first parameter
         3. See code for detailed connection notes

**Route Files With Express Router**

1. All express routes can be placed inside of the server.js file, but this becomes long and messy
2. Break the express routes down into multiple files
   1. See code for examples
3. Use the **app.use()** function, where app is an instance of express(), to assign parent route paths to a route file
   1. Pass in the parent route path as the first parameter (optional)
   2. Pass in the required file, using require(), as the second parameter
      1. **\*\***When referring to the parent route inside a route file, start the **path** string with ‘**/**’ then add additional child routes
         1. This will start the route path right behind the path passed into the app.use() method stated above

**Using API Routes & JWT Authentication**

1. Create a folder called **Models** to house all the database schema models for the server
   1. See code for detailed examples
2. The JSON **body-parser** library is now absorbed into **express**
   1. Call the statement: **app.use(express.json({ extended: false }))** to enable body-parser on an express server to receive the JSON payload of each server request
      1. The **extended** option allows a dev to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true)
3. When making an API request in Postman to an express server:
   1. Set the **Content-Type** Header value to **application/json**
   2. Send JSON data {with curly braces} through the Body pane
4. Use the **express-validator** library as middleware to an express server request
   1. https://express-validator.github.io/docs/check-api/#checkfield-message
   2. Pass in validator methods (i.e. **check**, **body**, **header**, etc.) as the second parameter, **middleware**, to an **express.Router().Route()** function for the request object to be validated
      1. \*Validation functions may be passed in as an array OR separate parameters
   3. Validation results are returned in the **validationResult()** function imported from the express-validator library
5. Use the **bcrypt** library to encrypt passwords before saving them to the database
6. Use the **jsonwebtoken** (**JWT**) library to authenticate users and return a session-reusable web token to a user
   1. [https://jwt.io/](https://jwt.io/#debugger-io)
   2. JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object
      1. This information can be verified and trusted because it is digitally signed
         1. JWTs can be signed using a secret with HMAC algorithm or RSA or ECDSA public/private key pairs
   3. JWT is most often used to transmit authorization information between server and client
   4. JWT mints a **token** that is passed between the client and server for each authorized request
      1. Once a user has logged in, each subsequent request will include the JWT token allowing the user to access routes and resources he/she has permission to view
      2. The token can easily be **verified** to ensure that no client has tampered with it
   5. JWT has 3 parts separated by periods (.):
      1. **Header** – The JWT header typically consists of 2 parts:
         1. **alg** – Defines the signing algorithm: HMAC, SHA256, or RSA
         2. **typ** – The type of token (always **“JWT”**)
      2. **Payload** – Contains **claims** which are statements about an entity (typically, the user) and additional metadata about the JWT itself
         1. There are 3 types of claims:
            1. **Registered Claims** – A set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject/user ID), **aud** (audience/Calling API domain) and [more](https://www.rfc-editor.org/rfc/rfc7519#section-4.1)
            2. **Public Claims** – A value that contains a Collision-Resistant Name

In order to prevent collisions, any new Claim Name should either be registered in the IANA "JSON Web Token Claims" registry or be a Public Name

The definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Claim Name

* + - * 1. **Private Claims** – A producer and consumer of a JWT MAY agree to use Claim Names that are Private Names: names that are not Registered Claim Names

Unlike Public Claims Names, Private Claim Names are subject to collision and should be used with caution

* + 1. **Signature** – The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is
       1. The signature is created by taking the encoded header, the encoded payload, a (user defined) secret, and **sign**ing them all with the algorithm specified in the header

1. Use bcryptjs's **hash()** function to encrypt any password with 184 bit encryption
   1. Pass in any password as the first parameter
   2. Pass in a **salt** (phrase that will distinguish this hash from any other) as the second parameter
2. Use bcyptjs's **compare()** function to poll if a plain text password is equal to its hashed value
3. An **Access Token** is given in response to an access token request (typically an authentication request)
   1. This token is used to access APIs and various backends
   2. It is typically issued by an authorization server or a third party vendor (server)
   3. An access token should have a short lifetime and should only be accepted while still valid
   4. Access tokens can be revoked, typically during user logout
4. A **Refresh Token** allows a user to gain an access token without signing in using their username/password
   1. This token should be saved to an application user collection in a database and verified upon auth requests
   2. A web service can request access tokens on behalf of the user without the user being present
   3. A refresh token should have a longer lifetime than an access token
   4. **\***The refresh token should be stored in a database for security purposes
5. Use the **jsonwebtoken.sign()** method to produce a "JSON Web Token"
   1. Pass a JSON object as payload data into the first parameter
   2. This JSON data typically contains info about the user and can contain "claims" i.e. "iss" (issuer), "exp" (expiration time), "iat" (issued at), etc: https://www.rfc-editor.org/rfc/rfc7519#section-4.1
   3. Pass in a **secretOrPrivateKey** as the second parameter
      1. This can be a secret for HMAC algorithms or the PEM encoded private key for RSA and ECDSA
      2. In case of a private key with passphrase an object { key, passphrase } can be used
   4. Pass in a JSON object for JWT **options** (JWT header/claims) OR an array containing options and a **callback** function for async processing
      1. These options are defined in the JWT docs: https://www.npmjs.com/package/jsonwebtoken
      2. The default "algorithm" option is "HMAC SHA256"
      3. The default "typ" option is "JWT"
      4. The "expiresIn" option can be a number (interpreted in seconds) OR a string describing a time span vercel/ms, i.e. 60, "2 days", "10h", "7d"
   5. \*\*If claims are passed in the payload, they CANNOT be passed in the options parameter
   6. \*\*The header can be customized via the options.header object
   7. \*\*Generated JWTs will include an iat (issued at) claim by default unless noTimestamp is specified
   8. \*\*The secretOrPrivateKey parameter can be read from an external file i.e. fs.readFileSync('private.key')
   9. \*\*It is common practice to pass in a version number to the refresh token's payload object so that the exact same refresh token will not be created twice
6. \*The default Authorization Header type for JWT is the **Bearer Token**, where an **Authorization** HTTP header is added to the request and its value is the word "Bearer" plus an empty space plus the JWT value
7. When making requests against a JWT enabled server, use the **x-auth-token** in the request header and assign it to the JWT key generated for the application
8. Use the **jsonwebtoken.verify()** method to decode the payload of a JWT
   1. Pass in the JWT to decode as the first parameter
   2. Pass in the **secretOrPublicKey** as the second parameter
      1. This should be a secret for HMAC, OR the PEM encoded public key for RSA and ECDSA algorithms
   3. Pass in optional "options" as the third parameter OR an array containing options and a callback for async processing as the third parameter; options available here: https://www.npmjs.com/package/jsonwebtoken
   4. This function returns an object which is the decoded contents of the JWT's payload
   5. **\*\***Claims data is returned in the payload
   6. \*\*Errors such as "TokenExpiredError" or "JsonWebTokenError" are implicitly thrown from this method if the header/payload/signature of the JWT are in error

**More on API Routes**

1. Use the **type** and **ref** schema property attributes, while creating a Mongoose model, to create a foreign key to (reference) another document (object) defined in another Mongoose model
   1. The **type** attribute should be assigned to **mongoose.Schema.Types.ObjectId**
   2. Add a **ref** attribute to this property and assign it to the name of the MongoDB **model** being referenced (i.e. ‘user’)
   3. Ex.: **const ProfileSchema = new mongoose.Schema({**

**user: {**

**type: mongoose.Schema.Types.ObjectId,**

**ref: 'user'**

**},**

1. When performing a query operation on a model for a referenced (joined) model object, pass the referenced property's name in the query and assign it to the ID of the referenced document (object)
   1. Ex.: **const profile = await Profile.findOne({user: req.user.id});**
2. Use Mongoose's **Query.populate()** method to specify fields that will be populated by the referenced document
   1. This changes the referenced property from just an ID to and object containing all fields specified
   2. Ex: **const profile = await Profile.findOne({user: req.user.id}).populate('user', ['name', 'avatar']);**
3. Use the **findOneAndUpdate**() method to find the first doc that fits given filter and update it or insert a new one (upsert)
   1. Pass in a JSON object containing the **filter criteria** as the first parameter
   2. Pass in a JSON object containing the fields to update as the second parameter
      1. **\*\***Use MongoDB's **$set** operator as a key and assign it to a whole object to update vs setting each doc field one-at-a-time
         1. **\*\***If the field does not exist, $set will add a new field with the specified value, provided that the new field does not violate a type constraint
   3. Pass in a JSON object containing "options" as the third parameter
      1. Set the **new** option to **true** to return the document AFTER its been updated vs un-updated values
      2. Set the **upsert** option to **true** to insert the doc if no doc fitting the criteria is found
         1. An upsert combines the filter and update parameters to save all given fields to the doc
         2. Set the **rawResult** option to true in order to return an object containing both the updated/new doc along with a **lastErrorObject** which contains metadata about the update/new doc
            1. Use the **updatedExisting** field of this object to determine whether the doc was updated or inserted
   4. \*This method is "Atomic" meaning the doc will not change between finding it and updating it (unless the doc is upserted)
4. Use the **Request.params** object, inside a router method, to access the **URL Parameters** (values in the route that are prefixed with a colon [**:**]) passed into the request
   1. These URL parameters should be passed into the **path** parameter of a Router.[verb]() method, where [verb] is get, post, put, etc.
   2. URL parameter names should follow RESTful API standards
5. When logging errors or trouble shooting a route, use the statement **error.kind === 'ObjectId'** to poll if the kind of error is a malformed **MongoDB Object ID**
6. Use Mongoose's **Model.findOneAndDelete()** method to find a document that matches a given criteria and remove it
   1. Pass in a JSON object as the **condition** or filter criteria as the first parameter
   2. Pass in a JSON object as **options** to delete the doc as the second argument
      1. Use the **sort** option to set the sort order to deleting the first doc if multiple docs are found
7. **\*\***Sub-Documents (JSON objects) inside arrays of other documents are automatically assigned IDs by MongoDB
8. Saving a top-level document also saves the state of all sub-documents (collections/arrays) as well
9. When content is being removed/deleted from a database, even in sub-document(s) it is good practice to declare an express **Router.delete()** method instead of a PUT request
10. When deleting sub-documents inside a sub-document array (since the sub-document is NOT a Mongoose Model object):
    1. Use **Array.map()** to map the sub-documents to an array of IDs
    2. Use **Array.indexOf()** to find the index of the sub-document ID passed to the request
    3. Use **Array.splice()** to remove the sub-document at given index
    4. Ex.: **const profile = await Profile.findOne({user: req.user.id});**

**const removeIndex = profile.experience**

**.map(itm => itm.id)**

**.indexOf((req.params.exp\_id));**

**profile.experience.splice(removeIndex, 1);**

**Post API Routes**

1. Sub-documents inside an array of a Mongoose model can also use the **type** and **ref** property attributes to form a foreign key relationship with another Mongoose model
2. Use Mongoose’s **Query.select()** method on a Mongoose model’s Query object to specify which document fields to include or exclude in a query (also known as the query “**projection**”)
   1. Pass in strings containing the names of fields to select (or exclude) as parameters
   2. **\*\***Placing a minus sign, "**-**", in front of a field name instructs Mongoose to exclude the proceeding field
3. Instantiate a new model object using a declared mongoose Model to create a new collection object that can be saved to MongoDB
   1. Ex.: **const** **user = new User({**

**name,**

**email,**

**avatar,**

**});**

**\***Where User is a declared Mongoose model

1. Use the **save()** method on an instantiated Mongoose model to insert or update the model object in the database
   1. **\***This methods persists to all collections and sub-documents of the model object
2. Use the **sort()** method on the result of a Mongoose model’s Query object to return a sorted list of documents from a MongoDB collection
   1. Pass in JSON object containing properties with:
      1. Its keys equal to names in the query’s MongoDB collection
      2. Its values equal to:
         1. **asc**, **ascending**,or **1** for ascending order
         2. **desc**, **descending**, or **-1** for descending order
      3. Assigning multiple field names to this object will cause the query to sort by the first field name first then within all results of the query that contain that field’s same value, sort by the proceeding field recursively
   2. Ex.: **const posts = await Post.find().sort({date: -1, username: asc});**
3. To get the string value of a Mongoose model’s ObjectID (either the model’s ID or a reference ID to another model) use the **toString()** method of that ObjectID
4. Use Mongoose’s **[Model].findByIdAndDelete()** method, where [Model] is a declared Mongoose model, to delete a single document by ID from the database
5. When assigning a reference object to a Mongoose model object, simply assign the referenced object’s ID to the reference property of the instantiated model object

**Getting Started with React & the Frontend**

1. Use **npx** instead of npm to run applications such as **create-react-app** without installing them globally on the local machine
2. In command line, use the command **yarn create react-app <app\_name>** where <app\_name> is the name of the new app to be created, to setup the most modern version of a React app (as of April, 2023)
3. **\*\***Use the syntax: **yarn --cwd <dir> <cmd>** to run a command, <cmd>, in yarn (including “scripts” defined in a package.json file) against a directory, <dir>, other than the current command line’s directory
4. Use the **concurrently** npm library to run multiple servers simultaneously
   1. Type the keyword **concurrently** at the beginning of a script
      1. Then enclose each command to run simultaneously in double quotes afterward
         1. **\***The double quotes must be escaped using **\** in a package.json file
   2. Ex.: **"scripts": {**

**"dev": "concurrently \"yarn server\" \"yarn client\""**

**}**

1. Use the package.json’s **“proxy”** field to specify a host server (including local machine) and port number that all URLs (without a host name specified) in the application route through
   1. \*\*This **ONLY** works in development, in production the routes will resolve to the host machine running the application
2. If using the package.json’s “proxy” field does NOT work in later versions of create-react-app, setup the **http-proxy-middleware** npm library as an alternative:
   1. Add the **http-proxy-middleware** library to the React application using yarn or npm
   2. Inside the **src** folder of the React project, add a new file named: **setupProxy.js**
   3. Use the **require** keyword to import the **createProxyMiddleware** package
   4. Use **module.exports** to export a function that:
      1. Call the **app.use()** method to bind middleware to the create-react-app’s dev server
         1. Pass in the **route** that is to be proxied as the first parameter (i.e. /api)
         2. Pass in the **createProxyMiddleware()** method as the second parameter
            1. Pass in a JSON object as the **context** to this method to configure the proxy

Set a **target** key equal to the desired proxy URL (i.e. ‘http://localhost:5000

Set a **changeOrigin** key equal to **true**

* 1. Ex.: **const { createProxyMiddleware } = require('http-proxy-middleware');**

**module.exports = function(app) {**

**app.use(**

**'/api',**

**createProxyMiddleware({**

**target: 'http://localhost:5000',**

**changeOrigin: true,**

**})**

**);**

**};**

**Thunk Overview**

1. The word **thunk** is a programming term that means: **“a piece of code that does some delayed work”**
2. Redux specific “**thunks**” are a pattern of writing functions with logic inside that can interact with a Redux store’s **dispatch()** and **setState()** methods
3. A file must import the **redux-thunk** middleware to use thunks
4. **\***Thunks are a standard approach for writing async logic in Redux apps
   1. They are commonly used for a variety of tasks and contain both synchronous and asynchronous logic
5. A **thunk** function is a function that accepts two arguments:
   1. The Redux store’s **dispatch()** method as the first parameter
   2. The Redux store’s **getState()** method as the second parameter
   3. This function may contain any arbitrary sync or async logic and can call dispatch() or getState() at any time
6. **\*\***Thunk functions are NOT directly called by application code; instead, they are passed to Redux’s store**.**dispatch() function
7. A **Thunk Action Creator** is a function that may have some arguments returns a new thunk function
   1. The thunk typically closes over any arguments passed to the action creator, so they can be used in the logic
8. Thunks allow side effect logic from being bound to any specific Redux store instance and keeps them reusable
   1. Use thunks to move complex logic outside of components
9. **\*\***Thunks can dispatch multiple actions in a row or over time
10. Thunks are "one-shot" functions, with no sense of a lifecycle:
    1. They cannot see other dispatched actions, so they should not generally be used for initializing persistent connections like websockets
    2. Thunks can't respond to other actions
11. Thunks are best used for complex synchronous logic, and simple to moderate async logic such as making a standard AJAX request and dispatching actions based on the request results
12. Dispatching thunk functions requires that the **redux-thunk middleware** has been added to the Redux store as part of its configuration
    1. Thunks can dispatch Redux actions or even other thunks
13. Thunk also supports dependency injection into all actions using its **extraArgument** attribute:
    1. const store = configureStore({

reducer: rootReducer,

middleware: getDefaultMiddleware =>

getDefaultMiddleware({

thunk: {

**extraArgument**: { serviceApi }

}

})

})

* 1. **\***There can only be one extra argument value; If multiple dependencies are needed, pass in an object containing them
     1. The thunk function will then receive that extra value, as an object, as its third parameter

1. A **Selector Function** is any function that accepts the Redux store state (or part of the state) as an argument and returns data based on that state
   1. **\***By convention, a selector function should start with the word **select**, combined with a description of the value being selected (i.e. **selectTodoById**)
2. The **useSelector()** react hook accepts a selector function and passes the entire Redux state into it
3. Use a thunks **getState()** parameter to access the current root Redux state value
   1. This can be useful for running conditional logic based on the current state
   2. It is common to use selector functions when reading state inside of thunks, rather than accessing nested state fields directly
4. It's preferable to put as much logic as possible in reducers, but it's fine for thunks to also have additional logic inside as well
5. **\***State is synchronously updated:
   1. As soon as any reducer processes and action, a call to getState(), even after an action has just been dispatched, gets the most updated state
6. Thunks may contain async logic as well as side effects (such as updating localStorage) using promise chaining and/or async/await syntax
7. **\*\***When making async requests inside a thunk, it is standard to dispatch actions BEFORE and AFTER the request is fulfilled/rejected
   1. The first request should dispatch a “pending” action and could be used to trigger any kind of loading graphic or icon to appear (i.e. using a state enum marked as “in progress”
   2. Either an async response or error should dispatch accompanying actions
   3. **\*\***It’s best to use the first and second parameters of the .then() function to handle responses and errors to ensure only errors related to the request itself are handled
   4. In async/await syntax, put a try/catch block around a single ajax request, use catch to deal with a rejection, then AFTER the necessary data is retrieved, dispatch a success action outside of the try/catch block
      1. This ensures that only network errors related to the async request are handled and not network errors resulting from the dispatch of the success action, which should be handled somewhere else
8. Thunks can also be repurposed to data querying repositories:
   1. Pass in a selector function to a thunk
   2. Use the selector function to retrieve the desired data from the result of getState() passed to the action
   3. Return desired state data back to the component requesting

**Using Redux**

1. **Redux** is a publish and subscribe, “pub-sub”, action-based data storage and retrieval framework that is commonly used with React; it allows for data (state) to selectively be acquired and distributed throughout components across an entire React application
   1. It is comprised of 3 attributes:
      1. **Store** – An immutable JSON object that stores the entire state of the React application
      2. **Action** – A Redux event that is fired by a component to get, update, or delete state data against the store
      3. **Reducer** – A function that subscribes to one or more Redux actions, processes the properties of the action, and manipulates a “**slice**” of the store based on the specific action triggered
         1. Reducers ONLY have access to their individual slice of the Redux state
         2. **\*\***Redux reducers must NOT contain side effects
2. Install the **redux**, **react-redux**, and **redux-thunk** npm packages to the React app
3. Dev Install the **redux-devtools-extension** npm package to the app to allow for inspection and debugging of react on the client
4. **\*\*\***Redux is a centralized queuing system:
   1. All of the reducers in an app are combined into one **Root Reducer** which is held within the Redux store
   2. Every action is dispatched by the Redux store
   3. All reducers subscribe the same set of actions
      1. **\*\*\***Each action type that a reducer responds to MUST match the exact value of the action type that the store dispatches
   4. An app’s **state** is universal and is represented by the store
   5. Each react component MUST **connect** to redux using the react-redux module
      1. Once connected, the component has access to Redux’s state (store) and actions with the use of the **mapStateToProps()** and **mapDispatchToProps()** methods
      2. Components can then access app state and dispatch Redux actions using its passed in **props** argument\*--+9
5. Create a folder named **actions**, under src, to hold the thunks and actions dispatched by the application
   1. Create a file inside the actions folder named **types.js** to hold all the action types
      1. For each **action type** to be dispatched, export a const named the action type and set it equal to a string of the exact same name
         1. This allows for action type’s names to be changed throughout the course of the application without changing the exported const
         2. **\*\***By convention, an action type should be uppercase, and each word separated by an underscore (i.e. **‘SET\_ALERT’**)
   2. Create a new file in this folder for every thunk action creator that creates a new action that is to be dispatched by the app
      1. Import the action types (created in the types.js file above) that are to be dispatched by this thunk
      2. Export a thunk (function) that receives all the specific parameters relevant to the action being invoked and assign it to a function that receives:
         1. A **dispatch** method as the first parameter which allows the thunk to dispatch any action defined in the Redux store
            1. The dispatch() method MUST accept an object containing:

The action’s **type** (defined in the type.js file above)

The action's **payload** which is arbitrary data specific to the action being dispatched

* + - 1. The **getState** method as the second parameter which grants access to the specified **slice** of **state** (added to the **rootReducer** via the **combineReducers()** function in the **index.js** file of the **reducers** folder)
    1. Use the getState() method to perform state-centric logic based on the action(s) to be dispatched
    2. Use the dispatch() method to dispatch actions based on the parameters received by the thunk and the state logic performed within
    3. Ex:

**import {v4} from 'uuid';  
import {*SET\_ALERT*, *REMOVE\_ALERT*} from './types';  
  
export const setAlert = (msg, alertType, timeout = 5000) => (dispatch, getState) => {  
 const state = getState();  
 *console*.log('State: ', state);  
 const id = v4();  
 dispatch({  
 type: *SET\_ALERT*,  
 payload: {msg, alertType, id}  
 });  
  
 setTimeout(() => dispatch({ type: *REMOVE\_ALERT*, payload: id}), timeout);  
}**

1. Create a **reducers** folder inside src to hold all the reducers for the application
   1. Create an **index.js** file to import all reducer files, then combine them and export them as one **Root Reducer**
      1. Import **{combineReducers}** from redux
      2. Import each reducer (declared in the reducers folder)
      3. Call the **combineReducers()** method and export the results as the default object
         1. Pass in a JSON object, containing each reducer, as its first parameter
         2. **\*\***This function maps each reducer to its file name in reducers folder under the src folder
            1. **\*\*\***If the reducer name does NOT match the file name, an error is thrown
         3. **\*\*\***By convention: the action’s file name MUST also match the reducer’s file name in order for Redux to link the two
   2. Create a **reducer** file for each reducer that is to handle action(s) dispatched by the store:
      1. Import the **action types** the reducer is to handle (defined in the **src/actions/types.js** file)
      2. Define a const that represents the reducer’s initial state
      3. Export a default function that represents the reducer being defined containing:
         1. A **state** object as its first parameter which contains the current **slice** of store state that the reducer reacts to
            1. **\*\***The reducer being defined now has access to the slice of state data that the reducer handles (respective to the name of the reducer passed into the combineReducers() method)
            2. Assign this to the reducer’s initialState const by default
         2. An **action** object as its second parameter which contains the dispatched action’s **type** and **payload** as its properties
            1. Extract the type and payload properties into separate variables using javascript object destructing
         3. Within this function define a **switch()** statement predicated to the **type** of action passed into the **action** object
            1. **\*\*\***This **type** passed to the switch() statement MUST exactly match the value of the action **type** being dispatched in order for Redux to process it
            2. Create a **case** statement for each **action type** (defined in the src/actions/types.js file) that the reducer reacts to

Within each reducer’s case statement, process the action’s **payload** data to manipulate the store’s data

Return the updated **state** object that results from processing the action’s payload

**\*\***This returned state object then UPDATES the slice of data processed by the reducer, thus updating the application’s overall store state in real time

* + - * 1. It is convenient to create a **default** case which returns the same **state** passed into the reducer, thus performing a no-op by the reducer

In this case, the Redux store state is NOT changed and no components are re-rendered

1. Create a file named **store.js** inside the /scr folder of the react app to hold the store
   1. Import the **createStore** and **applyMiddleware** libraries from **redux** package
   2. Import **{compseWithDevTools}** library from **redux-devtools-extension** package
   3. Import **thunk** form **redux-thunk** package
   4. Import **rootReducer** from the index file of the **reducers** folder
   5. Set a const named **initialState**, which holds the initial state of the app, to an empty JSON object
   6. Set a const named **middleware**, which holds an array of redux middleware frameworks, to an array
      1. Pass **thunk** middleware into this array
   7. Define a const named **store** and assign it to the result of the **createStore()** method
      1. Pass in the rootReducer as its first parameter
      2. Pass in the initialState as its second parameter
      3. Pass in all middleware as its third parameter
         1. Call the **compseWithDevTools()** method in this parameter to allow client-side redux debugging
            1. Pass in the **applyMiddleware()** function

Destruct the middleware const using the rest operator, passing the array of middleware into this method

* 1. Export the **store** const as the default parameter

1. In the **App.js** file:
   1. Import **{Provider}** from the **react-redux** package
      1. This object connects redux to a react app as redux also works independently from react
   2. Import the **store** from the store.js file
   3. Wrap the entire application (all JSX tags inside the **App** function) with the **<Provider>** tag imported from react-redux
      1. Pass in the store to the **store** attribute of this tag
2. Create a new reducer for every set of actions to be managed by Redux:
   1. Create a new file in the **reducers** folder and name it after the state and set of actions it subscribes to (i.e. **alert.js**)
      1. Import the necessary action type variables from the types.js file inside the actions folder
      2. Declare a const named **initialState** and assign it to an empty array
      3. Export a default function that accepts:
         1. The reducer’s **state** as its first parameter
            1. Give this parameter a default value of the initialState const
         2. The **action** it is subscribing to as its second parameter
            1. A reducer’s action MUST contain a **type** property
            2. The action should also contain a **payload** property if it has data that needs to be processed
      4. Inside the default function of the reducer:
         1. Declare a switch() statement to evaluate the action’s type property
            1. Define a case statement for every action type the reducer is to process

Each case should predicate on the name of the action’s type (imported from the types.js file)

After the case has finished processing its action’s type, return and array containing:

Use the spread operator (…) to return all the values of the **state** parameter passed into the function

Use commas to delineate each property key and value of the state that has been altered by the reducer inside this case

* + - * 1. **\***By convention, the default case of this switch statement SHOULD return the reducer function’s state parameter, indicating no action has taken place and the state has not changed
  1. In the root reducer file (the **index.js** file inside the reducers folder), import the new reducer file and pass it in as a property to the JSON object passed into the combineReducers() method

**\*\*\***The **state** parameter passed into **mapStateToProps** represents the Redux store’s **state** property:

1. **\*\***Each reducer’s state is held in a property of this variable named after (having a key of) the dispatched action’s file name
2. **\*\***To access a specific reducer’s state, assign **state.<action\_file\_name>** to a property in this method