Testing environment/rules

- [ ] Due to the limited time remaining, this will not be a traditional test.
- [ ] This is a continuation of the Issue Tracker project that we have been building on for this semester.
- [ ] This assignment serves as your **"Final Project Grade"** and will be weighted as **2.0 exam grades.**
- [ ] Review the rubric that follows and ensure that you complete all parts.

How to submit your code

Submit your code using the issue tracker repositories that you created earlier.

- [ ] **issue-tracker-react** for the frontend
- [ ] **awd1111-issue-tracker** for the backend API
- [ ] ***Do not create new repositories for this assignment!***

Ensure that all of the following have been done for both repositories:

- [ ] Make the repository **"private"**
- [ ] Add a **README** file
- [ ] Add a **.gitignore** file, using the **"Node"** template **(Don't forget this!)**
- [ ] Add a **LICENSE** file, using the **"MIT License"**
- [ ] Add **evangudmestad** as a **Collaborator/Admin**

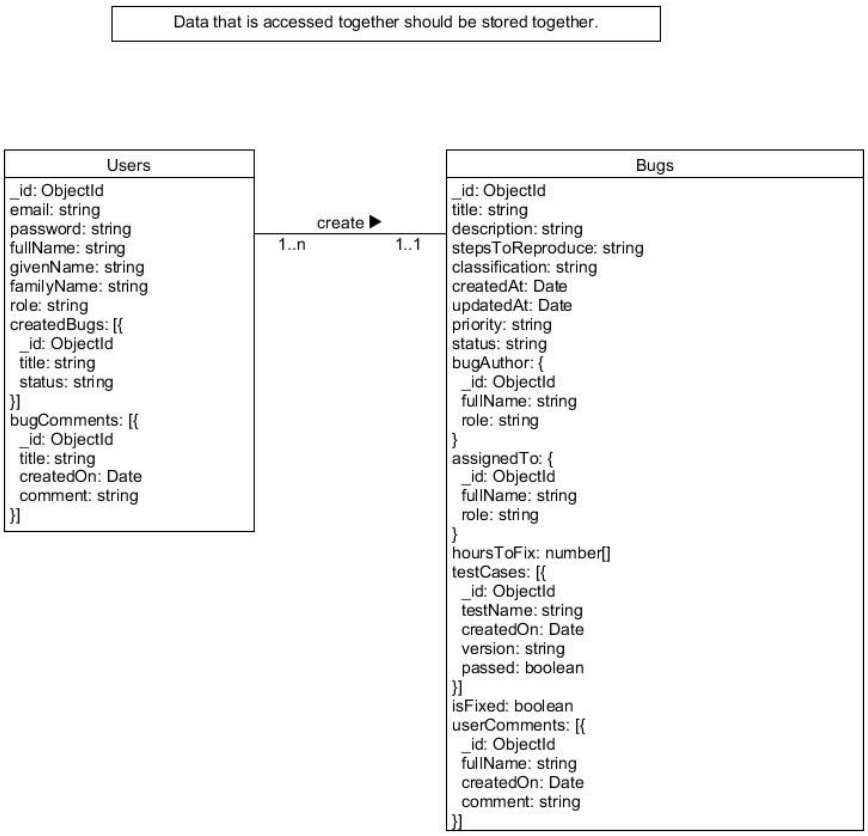
Read through the rubrics for each phase of the project

You will find an updated rubric for each phase of the project on the next page.

Review the instructions and complete the required tasks.

Phase 1 - Database Design - 14pts

Use UMLET to diagram your final database schema/ERD. (Lab 02-02 we created a draft of the database schema, similar to the image below. Do not use this image as it is outdated and for example only.) This schema should be highlighted in your video presentation.



Phase 2 - Web API (48pts)

You worked on the backend Web API for the first half of this course. This was built over a series of lab assignments.

In this part I want you to review the code that you wrote for the backend, and fix any outstanding issues or bugs.

Routes (24 routes, 48pts)

Your backend must implement all of the following routes.

(2pts for each fully completed and fixed route, see following pages for more instructions)

Users

- ☐ POST /api/user/register — Register a new account
- ☐ POST /api/user/login — Log into an existing account
- ☐ GET /api/user/list — Search for users in database
- ☐ GET /api/user/me — View your own profile
- ☐ GET /api/user/:userId — View any user's profile
- ☐ PUT /api/user/me — Update your own profile

- [ ] PUT /api/user/:userId Update any user's profile
- [ ] DELETE /api/user/:userId Permanently delete a user from the database

Bugs

- [ ] GET /api/bug/list Search for bugs in database
- [ ] GET /api/bug/:bugId View a particular bug
- [ ] PUT /api/bug/new Report a new bug
- [ ] PUT /api/bug/:bugId Update a bug (only title, description, and steps to reproduce!)
- [ ] PUT /api/bug/:bugId/classify Classify a bug as "Unclassified", "Approved", "Unapproved" or "Duplicate"
- [ ] PUT /api/bug/:bugId/assign Assign a bug to a user (by userId)
- [ ] PUT /api/bug/:bugId/close Close and Reopen bugs

Comments

- [ ] GET /api/bug/:bugId/comment/list View comments for a single bug
- [ ] GET /api/bug/:bugId/comment/:commentId View a single comment
- [ ] PUT /api/bug/:bugId/comment/new Post a new comment

Test Cases

- [ ] GET /api/bug/:bugId/test/list View test cases for a single bug
- [ ] GET /api/bug/:bugId/test/:testId View a single test case
- [ ] PUT /api/bug/:bugId/test/new Add a new test case (title and body only!)
- [ ] PUT /api/bug/:bugId/test/:testId Update an existing test case (title and body only!)
- [ ] PUT /api/bug/:bugId/test/:testId/execute Execute a test case (record who executed and whether it passed/failed)
- [ ] DELETE /api/bug/:bugId/test/:testId Delete a test case

Status Codes

Ensure that the correct status codes are sent from all routes

- [ ] **Send status code 404 when a route or entity is not found.**

  - [ ] Ensure the **404** handler in **server.js** contains the code below.
  A previous variation of this code sent the errors as plain text, instead of JSON.
  Ensure that your code is updated to send a JSON response.

```
app.use((req, res, next) => {
  debugError('Page not found.');
  res.status(404).json({ error: 'Page not found.' });
});
```

  - [ ] The **validId()** middleware helps with this as well, in sending a 404 error if a route parameter is not a valid ObjectId.

```
router.get('/:bugId', validId('bugId'), (req, res, next) => { ... });
```

- [ ] Within the route remember to check that the requested document exists and send a 404 error if the document is not found.

```
const bug = await findBugById(bugId);
if (!bug) return res.status(404).json({ error: 'Bug not found.' });
```

- [ ] **Send status code 401 when the route requires authentication, but the user is not authenticated. (req.auth is falsy)**
  - [ ] The **isLoggedIn()** or **hasPermission()** middleware functions implement this logic. *Install only the relevant one, not both.*

```
router.get('/me', isLoggedIn(), ...);
router.put('/me', isLoggedIn(), ...);
router.get('/list', hasPermission('viewUsers'), ...);
router.get('/:userId', hasPermission('viewUsers'), ...);
router.put('/:userId', hasPermission('manageUsers'), ...);
```

- [ ] **Send a status code 403 when the user is authenticated, but does not have the required permissions.**
  - [ ] Use the **hasPermission()** middleware function for most of these cases.
  - [ ] When the permissions required are based on ownership, what fields are being modified, or what action is being taken, use **if statements** to check **req.auth.permissions** for the permission.

Status Codes (cont.)

- [ ] **Send status code 400 when the request body (for POST and PUT) does not satisfy the appropriate Joi schema.**
  - [ ] All POST and PUT routes must have a Joi schema which defines what fields and datatypes are allowed and/or required.
  - [ ] Ensure that all of your shemas reflect the final project requirements.
  - [ ] **No authorship information is allowed in these schemas,** *it must be provided via **req.auth** instead.*
  - [ ] **No date/timestamps are allowed in these schemas,** *use the current server time instead.*
  - [ ] Use the **validBody()** middleware to implement this logic.

```
router.put('/me', isLoggedIn(), validBody(updateMeSchema), ...);
router.put('/:userId', hasPermission('manageUsers'),
validBody(updateUserSchema), ...);
```

- [ ] **Send status code 500 when there is an unhandled error.**
  - [ ] Ensure the **500** handler in **server.js** contains the code below. *A previous variation of this code sent the errors as plain text, instead of JSON. Ensure that your code is updated to send a JSON response.*

```
app.use((err, req, res, next) => {
  debugError(err);
  res.status(err.status || 500).json({ error: err.message });
});
```

- [ ] Wrap all routes in a try-catch block as shown below:

```
router.get('/me', isLoggedIn(), async (req, res, next) => {
  try {
    ...
  } catch (err) {
    next(err);
  }
});
```

- [ ] Or use the **asyncCatch** middleware that was provided earlier.
(Do not refactor your code, if you have used the previous method up until now.)

```
router.get('/me', isLoggedIn(), asyncCatch(async (req, res) => {
  ...
}));
```

- [ ] The **asyncCatch** middleware is also available as an npm package: [express-async-catch]

Phase 3 - Authentication and Authorization

Permissions Table - (5pts)

Update the table below with the required permission for each route. This should be updated to match the permissions implemented in your live application. This table should be demonstrated on your video presentation

Users	**Permission**
`POST /api/user/register`	*no authentication needed*
`POST /api/user/login`	*no authentication needed*
`GET /api/user/list`	
`GET /api/user/me`	*isLoggedIn()*
`GET /api/user/:userId`	
`PUT /api/user/me`	*isLoggedIn()*
`PUT /api/user/:userId`	
`DELETE /api/user/:userId`	
Bugs	**Permission**
`GET /api/bug/list`	

`GET /api/bug/:bugId`	
`PUT /api/bug/new`	
`PUT /api/bug/:bugId`	
`PUT /api/bug/:bugId/classify`	
`PUT /api/bug/:bugId/assign`	
`PUT /api/bug/:bugId/close`	
Comments	**Permission**
`GET /api/bug/:bugId/comment/list`	
`GET /api/bug/:bugId/comment/:commentId`	
`PUT /api/bug/:bugId/comment/new`	
Test Cases	**Permission**
`GET /api/bug/:bugId/test/list`	
`GET /api/bug/:bugId/test/:testId`	
`PUT /api/bug/:bugId/test/new`	
`PUT /api/bug/:bugId/test/:testId`	
`DELETE /api/bug/:bugId/test/:testId`	

Middleware (1pt each - 21pts)

Using the table above, ensure that all routes have the correct middleware installed.

- ☐ For only the routes listed above as **isLoggedIn(),** this must be the **isLoggedIn** middleware.
- ☐ For all other routes, this must be the **hasPermission** middleware.

Login & Register (5pts)

Both the login and register have some special requirements in terms of authentication.

- ☐ They do not require the user to be logged in.
- ☐ They must generate a new JWT token and store it in a cookie.
- ☐ For the login route, the payload must be populated with a permissions table that combines the permissions of all their assigned roles.

Update User (10pts)

The routes to update your own account and other users accounts have some special considerations as well:

PUT /api/user/me

- ☐ Only allows you to update your own account.
- ☐ Use **req.auth.userId** to determine who you are logged in as.
- ☐ **Do not allow users to change their own role unless they have the correct permission to do so.**

PUT /api/user/:userId

- ☐ Allows you to update all user accounts. (Including your own.)
- ☐ Use **req.auth.userId** to determine who you are logged in as.
- ☐ Use **req.userId** to determine who is being edited.
- ☐ **If you are updating yourself:**
  - ☐ **You need to issue a new token for yourself, with updated information and permissions.**
  - ☐ **And this token must be returned in the body of the request.**
- ☐ **If you are not updating yourself:**
  - ☐ **DO NOT ISSUE A NEW TOKEN!**
  - ☐ ***If you issue a new token for other users, this creates a CRITICAL security problem, in that it allows users to impersonate any other user.***

Phase 4 - React Frontend

We have built a frontend application with React, which connects the API that we built previously.

In this part I want you to review the code that you wrote for the frontend, and fix the outstanding issues.

Dependencies

Ensure that you have all of the following dependencies installed on the frontend.

- ☐ **axios**
- ☐ **bootstrap**
- ☐ **Lodash (optional)**
- ☐ **moment(optinoal)**
- ☐ **node-sass(optional)**
- ☐ **(Any)Icon set**
- ☐ **react-router-dom**
- ☐ **react-toastify**

Use all of these dependencies in the project.

Components

Your frontend must include all of the following components (additional components are allowed.)

- ☐ **Navbar**
- ☐ **Footer**
- ☐ **NotFound**
- ☐ **LoginForm**
- ☐ **RegisterForm**
- ☐ **BugList**
- ☐ **BugListItem**
- ☐ **BugEditor**
- ☐ **ReportBug**

- ☐ **UserList**
- ☐ **UserListItem**
- ☐ **UserEditor**

index.js (2pts)

- ☐ Wrap **<App />** in a **<BrowserRouter>**
- ☐ Do not import any bootstrap stylesheets in **index.js**

App.js (8pts)

- ☐ Create your own **App.scss** file to style and theme the application.
- ☐ Import **App.scss** on **line 1.**
- ☐ Do not import any bootstrap stylesheets in **App.js**
- ☐ Import both your chosen bootswatch theme and bootstrap inside **App.scss**

```
// replace [theme] with the name of your selected theme
// remember to keep the theme name lowercase
@import "~bootswatch/dist/[theme]/variables";
@import "~bootstrap/scss/bootstrap";
@import "~bootswatch/dist/[theme]/bootswatch";
```

- ☐ Use a state variable named **auth,** to store the current login info. (Must exclude the password.)
  - ☐ **userId, email, payload, token**
- ☐ Use **localStorage** to persist the auth token.
- ☐ Add a **useEffect()** hook which reloads the auth token from **localStorage** when the browser is reloaded.
- ☐ Use the **<Routes>** and **<Route>** elements to implement all of the routes listed below:

/	redirects to /login
/login	displays a login page (use <LoginForm />)
/register	displays a registration page (use <RegisterForm />)
/bug/list	displays the list of bugs and the search interface (use <BugList />)
/bug/report	allows users to report a new bug (use <ReportBug />)
/bug/:bugId	displays an editor for a single bug (use <BugEditor />)
/user/list	displays the list of bugs and the search interface (use <UserList />)
/user/me	displays an editor for your own profile (use <UserEditor />)
/user/:userId	displays an editor for a single user (use <UserEditor />)
*****	displays a not found page (use the <NotFound/> component for this)

- ☐ Include a **<ToastContainer>** to display toast messages.
- ☐ Include the **<Navbar>** component at top of all pages.
- ☐ Include the **<Footer>** component at bottom of all pages.
- ☐ Wrap the **<Routes>** component within a <main> element.
- ☐ Use **CSS Flexbox** to keep the footer at the bottom of the viewport. (MDN Sticky Footers)

Navbar.js (5pts)

- ☐ Use the **<NavLink>** component for the links in the navbar.
- ☐ Style the Navbar component using all of the appropriate bootstrap classes.
- ☐ The navbar must be horizontal on desktop devices.
- ☐ The navbar must be collapsed vertically on mobile devices.
- ☐ Change which links are visible based on whether the user is logged in or logged out.

Footer.js (5pts)

- ☐ Include copyright information and your name.
- ☐ Remember to use © for the copyright symbol.

LoginForm.js (5pts)

- ☐ Provides an email/password login form for the user.
- ☐ Use **axios** to send a **POST** request to the **/api/user/login** route to login the user.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ Use **.catch()** to display all validation errors returned from Joi.
- ☐ Use frontend validation logic to prevent the request, if invalid data is present.
- ☐ If login is successful, redirect the user to **"/bug/list"** or **"/user/me"** (if they do not have a role)

RegisterForm.js (5pts)

- ☐ Provides a registration form for the user.
- ☐ This registration form must include all of the following fields:
  - ☐ Email
  - ☐ Confirm Email
  - ☐ Password
  - ☐ Confirm Password
  - ☐ Given Name
  - ☐ Family Name
  - ☐ Full Name (must be a separate field!)
- ☐ Use **axios** to send a **POST** request to the **/api/user/register** route to register the user.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ Use **.catch()** to display all validation errors returned from Joi.
- ☐ Use frontend validation logic to prevent the request, if invalid data is present.
- ☐ If register is successful, redirect the user to **"/user/me"**

BugList.js (8pts)

- ☐ Provides an interface to view the bug list.
- ☐ Also provides an interface to search the bug list. (See Lab 06-04 and Lab 04-04 for details.)
- ☐ Use **axios** to send a **GET** request to the **/api/bug/list** route to get the bug list.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ When the user clicks on a bug, redirect them to **"/bug/:bugId"** where **:bugId** is the bug's ID.

BugListItem.js (2pts)

- ☐ Renders a single bug in the list.
- ☐ Use a bootstrap card to style this item.
- ☐ Use a prop to receive the bug data.

BugEditor.js (10pts)

- ☐ Provides an editor for a single bug.
- ☐ Use the **useParams()** hook to read the bugId from the URL.
- ☐ Use **axios** to send a **GET** request to the **/api/bug/:bugId** route to get a bug.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/:bugId** route to update a bug.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/:bugId/classify** route to classify a bug.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/:bugId/assign** route to assign a bug.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/:bugId/close** route to close/open a bug.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ Use **.catch()** to display all validation errors returned from Joi.
- ☐ Use frontend validation logic to prevent the request, if invalid data is present.
- ☐ This component must include all of the following forms:
  - ☐ Edit Bug Details (title, text, and steps to reproduce only!)
  - ☐ Classify Bug
  - ☐ Assign Bug
  - ☐ Open/Close Bug
  - ☐ Post Comment
- ☐ When the user submits any of these forms, do not redirect them to a different page.

ReportBug.js (10pts)

- ☐ Provides an interface to report a new bug.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/new** route to create a new bug.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ Use **.catch()** to display all validation errors returned from Joi.
- ☐ Use frontend validation logic to prevent the request, if invalid data is present.
- ☐ When the bug is successfully created, redirect the user to **"/bug/:bugId"** where **:bugId** is the bug's ID.

UserList.js (8pts)

- ☐ Provides an interface to view the user list.
- ☐ Also provides an interface to search the user list. (See Lab 06-04 and Lab 04-04 for details.)
- ☐ Use **axios** to send a **GET** request to the **/api/user/list** route to get the user list.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ When the user clicks on a user, redirect them to **"/user/:userId"** where **:userId** is the user's ID.

UserListItem.js (2pts)

- ☐ Renders a single user in the list.
- ☐ Use a bootstrap card to style this item.
- ☐ Use a prop to receive the user data.

UserEditor.js (10pts)

- ☐ Provides an editor for a single user.
- ☐ Use the **useParams()** hook to read the userId from the URL.
- ☐ Use **axios** to send a **GET** request to the **/api/user/:userId** route to get a user.
- ☐ Use **axios** to send a **PUT** request to the **/api/bug/:userId** route to update a user.
- ☐ Use **.catch()** to handle and display the error message returned by the server.
- ☐ Use **.catch()** to display all validation errors returned from Joi.
- ☐ Use frontend validation logic to prevent the request, if invalid data is present.
- ☐ When the user submits any forms on this page, do not redirect them to a different page.

Additional Notes:

- ☐ The user should be able to update/reset their password from this page.
- ☐ The user is not required to change their password, if the user leaves the password field blank, then send the password to the backend as **undefined,** and ensure that the backend does not update the password.
- ☐ For changing the password include both **Password** and **Confirm Password,** so that the user can't accidentally lock themselves out of their account.
- ☐ If the user is editing themselves, then you need to get the updated token from the server response, and update the **auth** state variable in **App.js**

Theming (6pts)

See directions in **Lab 06-04** and **lecture notes here,** to customize your app's theme.

- ☐ Create a color palette using **color.adobe.com** or similar site**,** and use it in your app.
- ☐ Use at least two fonts from **Google Fonts**
- ☐ Use **react-icons** to add icons to your navigation links and buttons.

Adding Custom Functionality (10pts)

Throughout all of the labs, we have implemented various features. I'd like you to implement at least 1 new feature that was not covered in the class. These features should require additional research, code experimentation (outside of the project), and troubleshooting.

Items to consider implementing include but are not limited to:

- Research popular NPM packages and utilize a new package that was not introduced in the course
- Login with Passport.js with gmail credentials
- Email or SMS notifications
- CRUD functionality on another "thing".
    - We work with users and bugs. Implement another Collection or SubDocument to work with
- Bring in another free API
    - Facebook
    - Amazon
    - Twitter/X
    - Ebay