T3A2 School Yearbook App

Wenxuan Pan, Dieter Schmid, Alicia Han

Website Description

A Digital School Yearbook

Makes a permanent record of all students in attendance for a year level that has graduated a school.

- Displays Image of the Student
- Contact Information on the student
- Final Thoughts

Features/Functionality

- The admin can add students to the app
- The students can update their details
- The admin can upload photos and display them for students
- The students can fill out a questionnaire regarding their final thoughts
- The student can provide contact details
- Admin can invite students to the app via a unique code
- The app will allow admin and student logins

Target Audience

The Application is targeted at all school stakeholders this includes :

- Students of the graduating year level
- Other students at the school
- Alumni members
- Parents of the students
- Teachers and other staff

Tech Stack

This application uses the MERN technology stack:

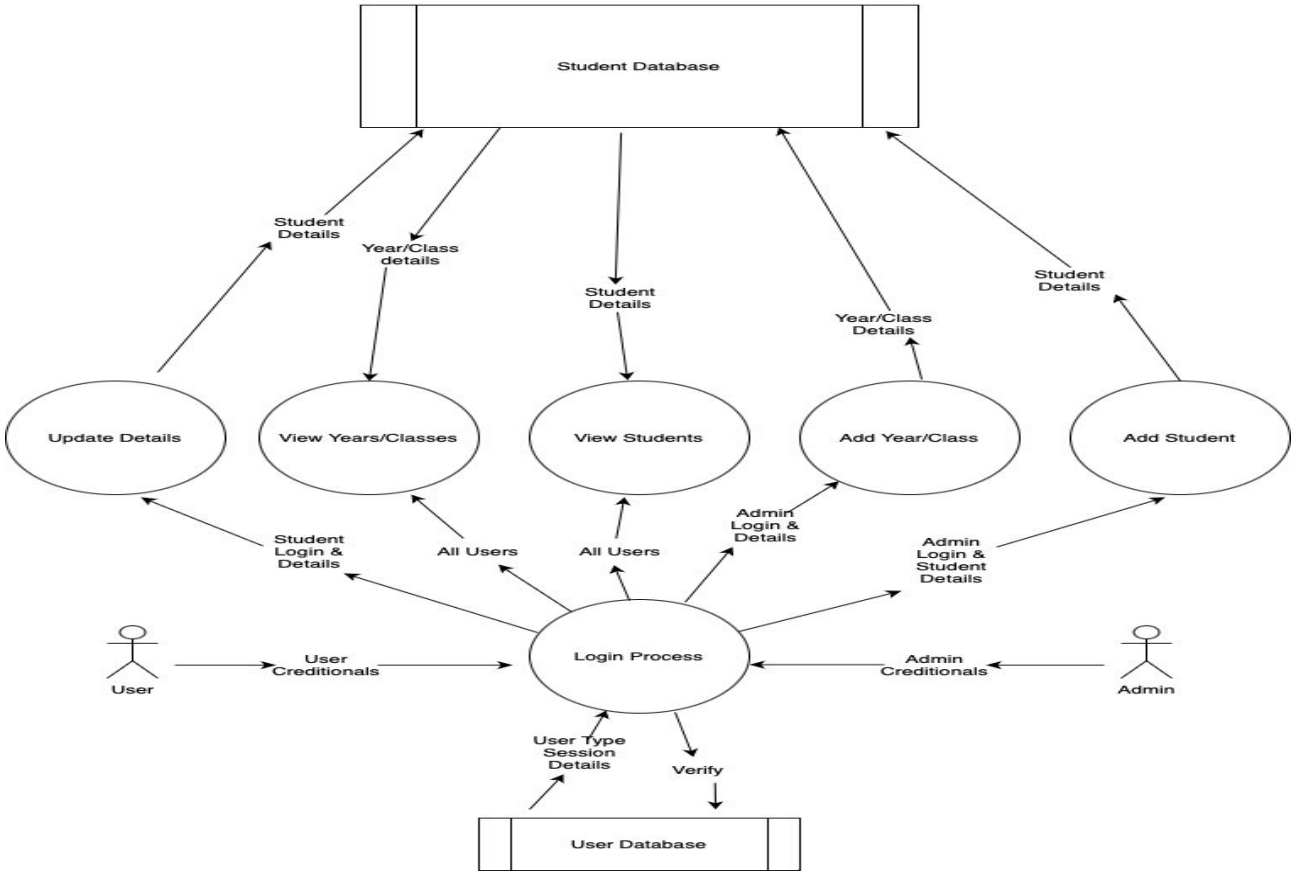
- MongoDB
- Express.js server
- Node.js
- React front end.

Deployment:

- Front end: Vercel
- Back end: Render

Dataflow Diagram

Dataflow Diagram



User Stories

User Stories

Split into three categories:

1. All users (students and admin)
2. Admin only
3. Students only

Throughout the planning process, we found that we needed to add and modify ideas in order for the application to work better. Hence, user stories from the beginning of the planning phase were updated to reflect these changes.

Initial User Stories

All users:

1. As a user, I would like to be able to add a student profile into different classes/cohorts and year levels, so that student data is organised into the correct categories.
2. As a user, I would like to be able to login and view a list of students in each class and year level/cohort at my school.
3. As a user, I would like to be able to click into individual students and see their relevant information, so that I can stay in touch with classmates and connect with acquaintances.
4. As a user, I would like to have a navigation bar so that I can easily access different parts of the website.

Initial User Stories

Admin only:

1. As an admin, I want to be able to remove any profile from the database to ensure data integrity.

Students only:

1. As a student, I would like to be able to update my information, so that others can get in touch with me via my current contact details if needed, however I would like to ensure that only admin and myself can manipulate my account, for better security.
2. As a student, I want to be able to delete any information on my account, or my entire profile, from the database so that I have control over my information.

Revised User Stories with Updated Features

All users:

1. As a user, I would like to be able to login and view a list of classes in my school, and the students in each class.
2. As a user, I would like to be able to click into individual students and see their relevant information, so that I can stay in touch with my classmates and connect with acquaintances.
3. As a user, I would like to have a navigation bar so that I can easily navigate around the application.

Revised User Stories with Updated Features

Admin only:

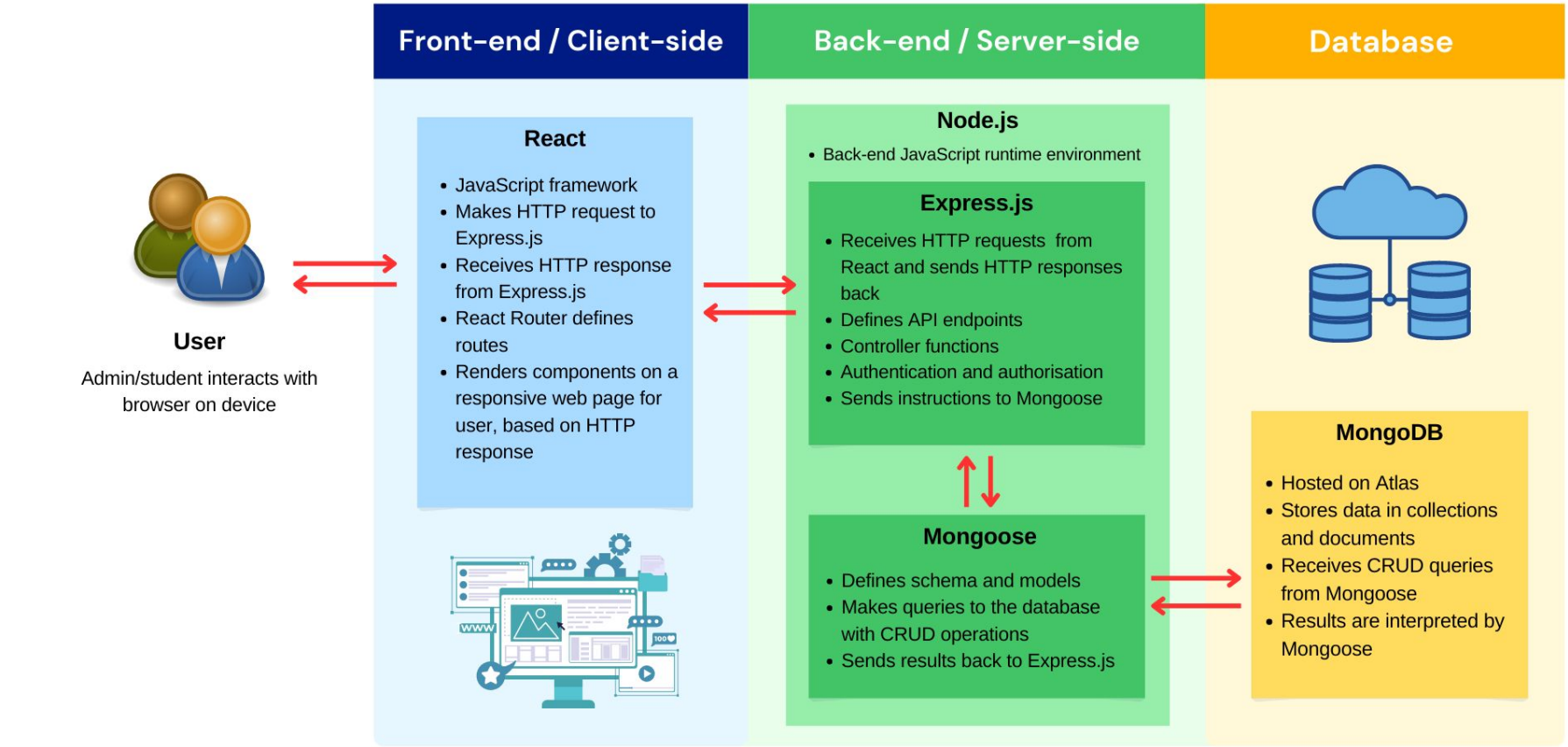
1. As an admin, I would like to see a list of all yearbooks and classes at my school, and be able to modify or delete them as appropriate, in case of error or duplication.
2. As an admin, I would like to be able to input a class of students along with their email and photos, and modify and delete student data as needed in case of error or duplication.
3. As an admin, I would like to be able to have a unique code to give to students, so they can utilise it to sign up and create their own accounts linked to the student database, this will ensure that only students who are actually in the class will be able to register to that class.

Revised User Stories with Updated Features

Students only:

1. As a student, I would like to have a unique code from my school admin, to be able to register as an account that is linked to a yearbook in the database.
2. As a student, I would like to be able to add, modify and delete any of my data from my account, and ensure that only the admin or myself can manipulate my account, for better security.

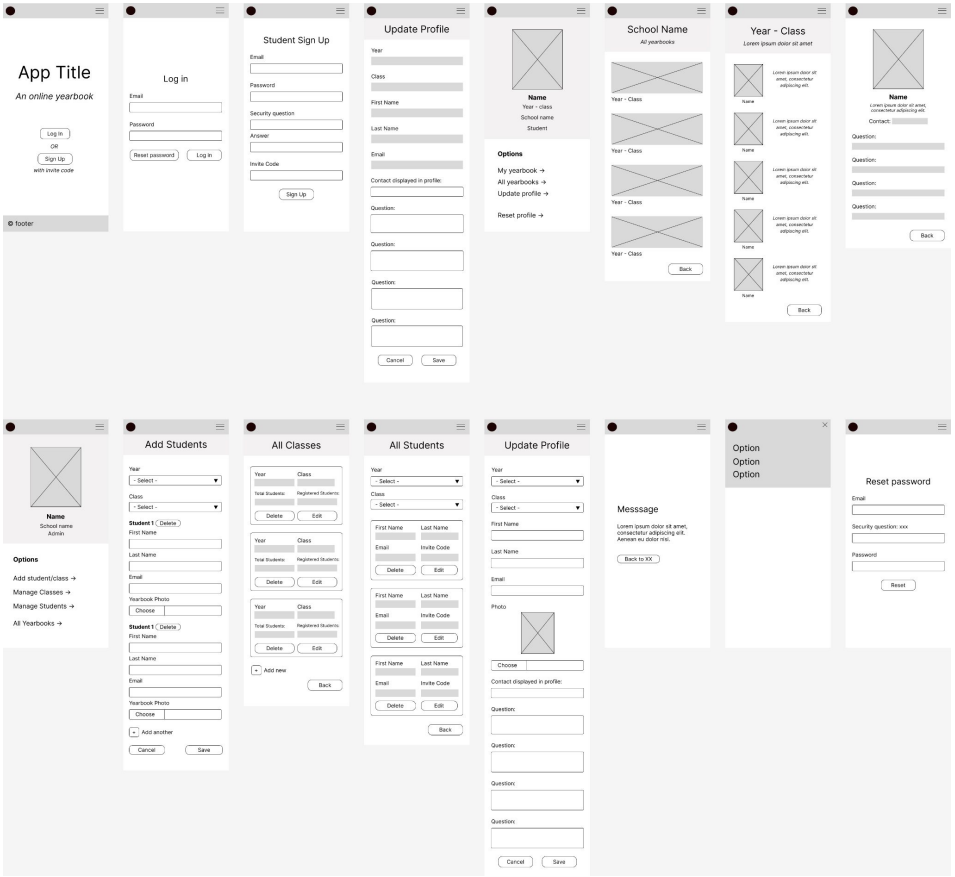
Application Architecture Diagram


Front-end / Client-side
Back-end / Server-side
Database
User
Admin/student interacts with browser on device
React
JavaScript framework
Makes HTTP request to Express.js
Receives HTTP response from Express.js
React Router defines routes
Renders components on a responsive web page for user, based on HTTP response
Node.js
Back-end JavaScript runtime environment
Express.js
Receives HTTP requests from React and sends HTTP responses back
Defines API endpoints
Controller functions
Authentication and authorisation
Sends instructions to Mongoose
Mongoose
Defines schema and models
Makes queries to the database with CRUD operations
Sends results back to Express.js
MongoDB
Hosted on Atlas
Stores data in collections and documents
Receives CRUD queries from Mongoose
Results are interpreted by Mongoose

Wireframes

Mobile

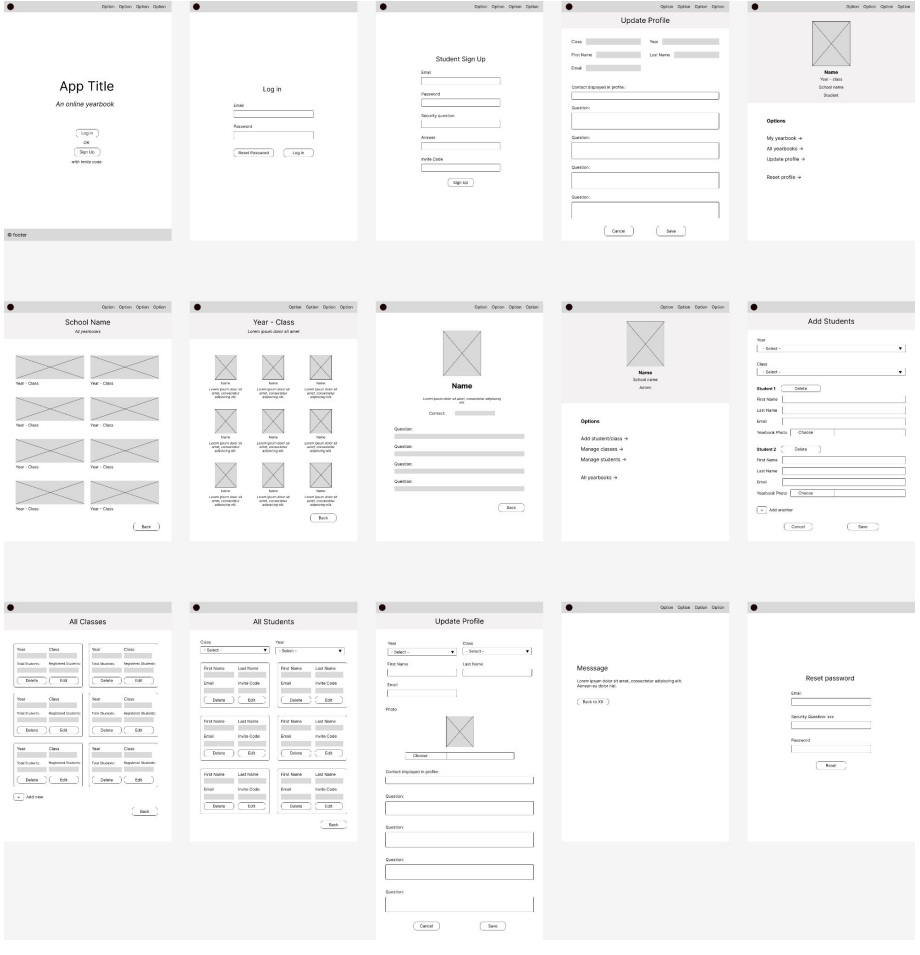
[View in Figma]



*Note (2 Sep): Reset password not implemented in production stage due to time constraints

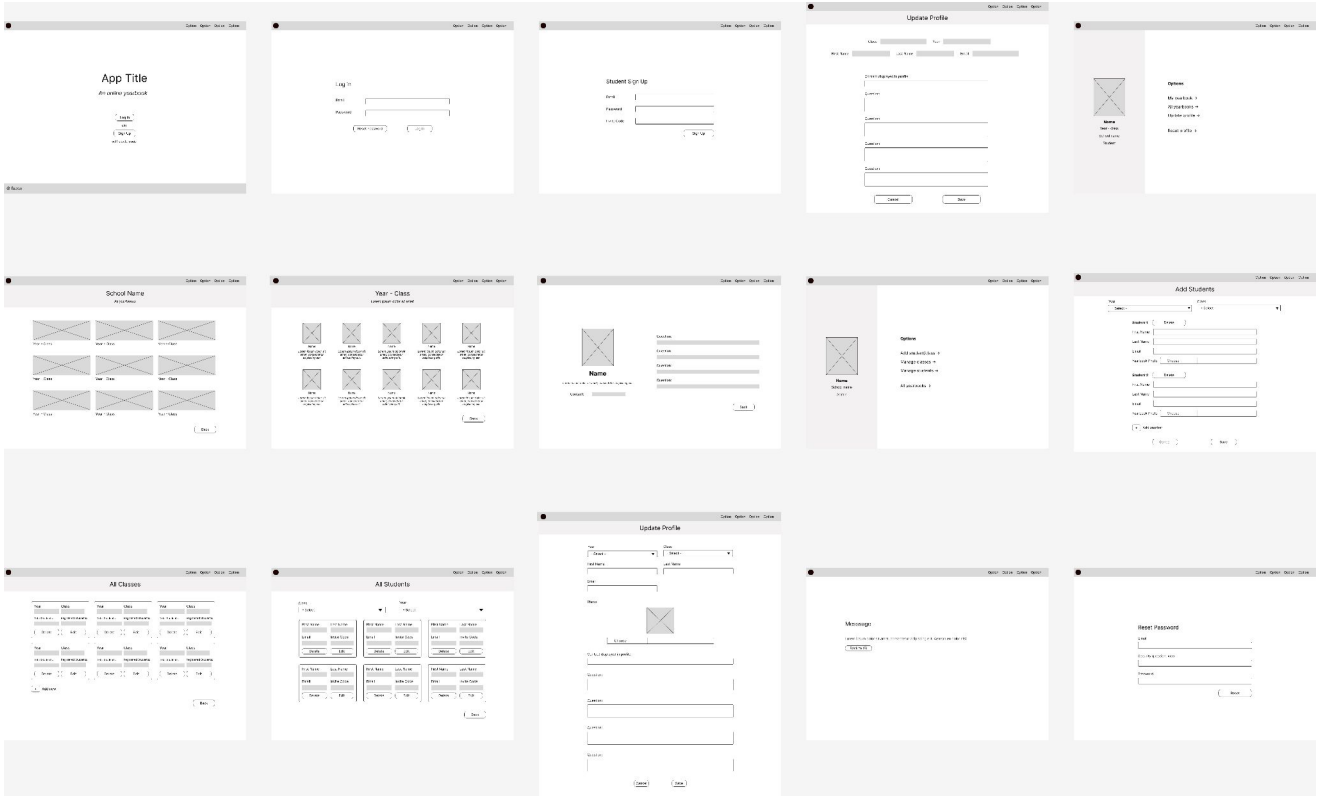
Tablet

[View in Figma]



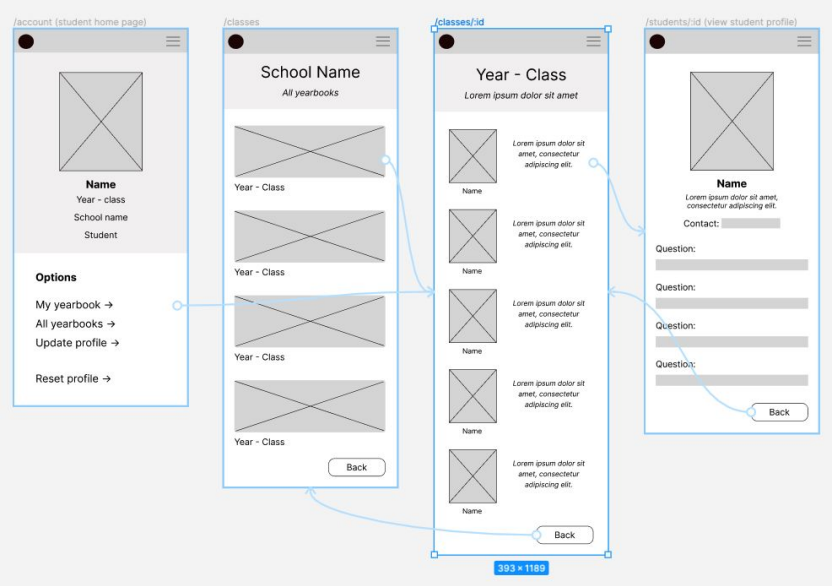
Desktop

[View in Figma]



Relationships between screens

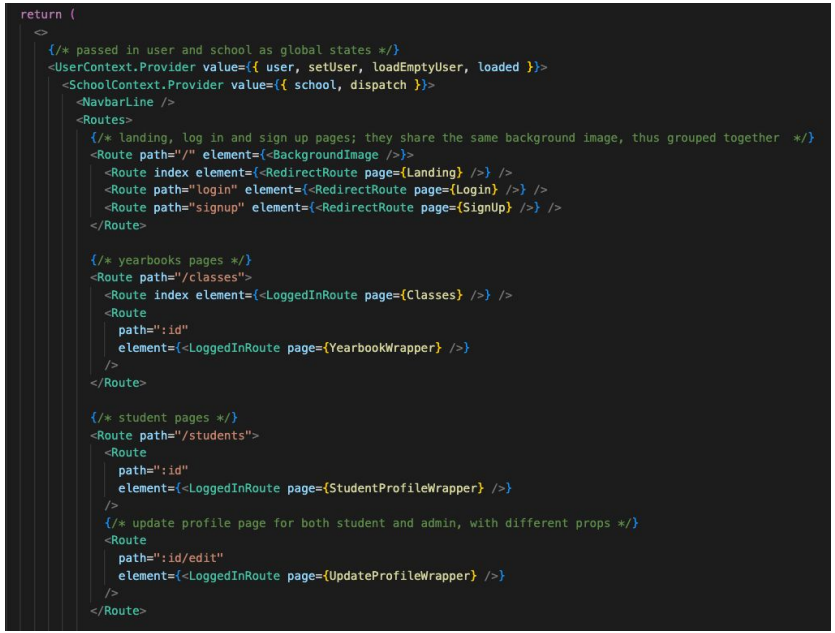
- [Student flow - Figma]
- [Admin flow - Figma]



Code Examples: Front End

App component

- React router is used to enable client side routing
- Context API is used to allow components lower down the tree to easily access shared state without having to pass props through multiple levels
- Reducer pattern is applied to manage complex state (i.e. school - classes, years, students) logic in a maintainable and predictable way

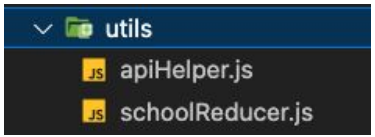


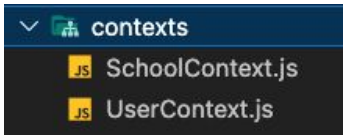
```
return (
  <>
    {/* passed in user and school as global states */}
    <UserContext.Provider value={{ user, setUser, loadEmptyUser, loaded }}>
      <SchoolContext.Provider value={{ school, dispatch }}>
        <NavbarLine />
        <Routes>
          {/* landing, log in and sign up pages; they share the same background image, thus grouped together  */}
          <Route path="/" element={<BackgroundImage />}>
            <Route index element={<RedirectRoute page={Landing} />} />
            <Route path="login" element={<RedirectRoute page={Login} />} />
            <Route path="signup" element={<RedirectRoute page={SignUp} />} />
          </Route>

          {/* yearbooks pages */}
          <Route path="/classes">
            <Route index element={<LoggedInRoute page={Classes} />} />
            <Route
              path=":id"
              element={<LoggedInRoute page={YearbookWrapper} />}
            />
          </Route>

          {/* student pages */}
          <Route path="/students">
            <Route
              path=":id"
              element={<LoggedInRoute page={StudentProfileWrapper} />}
            />
            {/* update profile page for both student and admin, with different props */}
            <Route
              path=":id/edit"
              element={<LoggedInRoute page={UpdateProfileWrapper} />}
            />
          </Route>
```

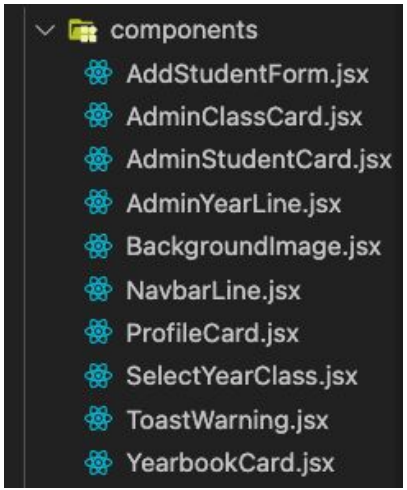
```
          {/* student pages */}
          <Route path="/students">
            <Route
              path=":id"
              element={<LoggedInRoute page={StudentProfileWrapper} />}
            />
            {/* update profile page for both student and admin, with different props */}
            <Route
              path=":id/edit"
              element={<LoggedInRoute page={UpdateProfileWrapper} />}
            />
          </Route>

          {/* account pages */}
          <Route path="/account">
            <Route index element={<LoggedInRoute page={Account} />} />
            <Route
              path="classes"
              element={<AdminRoute page={ManageClasses} />}
            />
            <Route
              path="classes/new"
              element={<AdminRoute page={AddClass} />}
            />
            <Route
              path="students"
              element={<AdminRoute page={ManageStudents} />}
            />
            <Route
              path="students/new"
              element={<AdminRoute page={AddStudents} />}
            />
          </Route>
          <Route
            path="/user-error"
            element={<RedirectMessage type={"userError"} />}
          />
          <Route path="*" element={<RedirectMessage type={"notFound"} />} />
        </Routes>
      </SchoolContext.Provider>
    </UserContext.Provider>
  </>
```

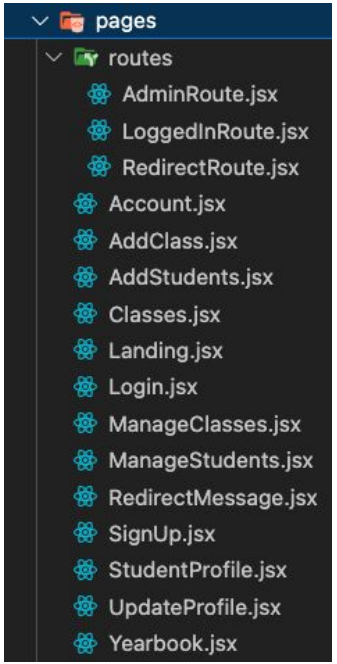
Components/modules

Code is divided into different components and utilities that are responsible for one single purpose or a specific task, to meet the DRY principle and single responsibility principle, and to ensure maintainability, reusability and testability









Conditional Rendering

- isAdmin and isLoggedIn statuses are used in route wrappers to protect and redirect confidential pages
- isAdmin variable is used to conditionally render UI elements for students and admins

```
const AdminRoute = ({ page: Page }) => {
  const { user, loaded } = useContext(UserContext)
  if (!loaded) {
    return <p>Loading...</p>
  }

  if (!user.isAdmin) {
    if (!user.isLoggedIn) {
      return <Navigate to="/" replace />
    }
    return <Navigate to="/account" replace />
  }
  return <Page />
}
```

```
// set navbar text and direct links for different views
const views = {
  default: [
    { text: "Log In", navLink: "/login" },
    { text: "Sign Up", navLink: "/signup" }
  ],
  student: isLoggedIn &&
    !isAdmin && [
      { text: `${user.name} - Student`, navLink: "/account" },
      { text: "My yearbook", navLink: `/classes/${studentClass._id}` },
      { text: "All yearbooks", navLink: "/classes" },
      { text: "Update profile", navLink: `/students/${student._id}/edit` }
    ],
  admin: isLoggedIn &&
    isAdmin && [
      { text: `${user.name} - Admin`, navLink: "/account" },
      { text: "Students", navLink: "/account/students" },
      { text: "Classes", navLink: "/account/classes" },
      { text: "Yearbooks", navLink: "/classes" }
    ]
}

// find out current view based on logged in and admin statuses
const currentView = isLoggedIn
  ? isAdmin
    ? views.admin
    : views.student
  : views.default
// nav options based on current view
const viewEl = currentView.map((option) => (
  <Nav.Link as={Link} to={option.navLink} key={option.text}>
    {option.text}
  </Nav.Link>
))
```

Error Handling

- Try-catch blocks are used to handle possible errors during API calls
- React-toastify library is used to display error message on the client's end

```
async function handleSubmit(e) {
  e.preventDefault()
  try {
    const { token, user: registeredUser } = await apiPost("/signup", content)
    const { __v, role, ...filteredUser } = registeredUser
    const newUser = {
      token,
      isLoggedIn: true,
      isAdmin: role === "admin" ? true : false,
      ...filteredUser
    }
    setUser(newUser)
    localStorage.setItem("user", JSON.stringify(newUser))
    nav("/account")
  } catch (e) {
    console.error(e)
    toast.warn("Sign up failed. Please check your credentials and try again")
  }
}
```

Example component - Add Class (admin)

- React-bootstrap is used for styling UI elements
- useState is used to handle the input state specific to this component
- handleSubmit and handleInputChange functions are used to encapsulate the workflows of form actions
- Dispatch method from the SchoolContext is used to update the global state with the new class (and year)
- Error message will be displayed when API call fails

```
const AddClass = () => {
  const nav = useNavigate()
  const [content, setContent] = useState({ name: "", year: "" })
  const { school, dispatch } = useContext(SchoolContext)
  const { user } = useContext(UserContext)

  function handleInputChange(changed) {
    setContent((prev) => ({ ...prev, ...changed }))
  }

  async function handleSubmit(e) {
    e.preventDefault()

    try {
      // check if year exists
      const yearExists = school.years.find((y) => y.name === content.year)
      if (!yearExists) {
        // if not exists, create new year
        const newYear = await apiPost(
          "/years",
          { name: content.year },
          user.token
        )
        // add new year to school state
        dispatch({ type: "add_year", year: newYear })
      }
      const createdClass = await apiPost("/classes", content, user.token)
      dispatch({ type: "add_class", class: createdClass })

      nav("/account/classes")
    } catch (e) {
      // if POST request failed, show toast message
      console.error(e)
      toast.warn("Class creation failed. Please check input and try again")
    }
  }
```

```
return (
  <Container className="mt-4">
    <Row>
      <h1 className="fs-2">Add Class</h1>
    </Row>
    <Form onSubmit={(e) => handleSubmit(e)}>
      <Row xs={1} md={2}>
        <Form.Group controlId="formYear" as={Col}>
          <Form.Label>Year</Form.Label>
          <Form.Control
            value={content.year}
            onChange={(e) => handleInputChange({ year: e.target.value })}
            type="number"
            min="1960"
            max="2099"
          />
        </Form.Group>
        <Form.Group controlId="formClass" as={Col}>
          <Form.Label>Class</Form.Label>
          <Form.Control
            value={content.name}
            onChange={(e) => handleInputChange({ name: e.target.value })}
            type="text"
            maxLength="100"
          />
        </Form.Group>
      </Row>
      <Row className="text-center mt-4">
        <Col>
          <Button as={Link} to={"/account"}>
            Cancel
          </Button>
        </Col>
        <Col>
          <Button type="submit">Save</Button>
        </Col>
      </Row>
    </Form>
    <ToastWarning />
  </Container>
)
```

Test: Log in component

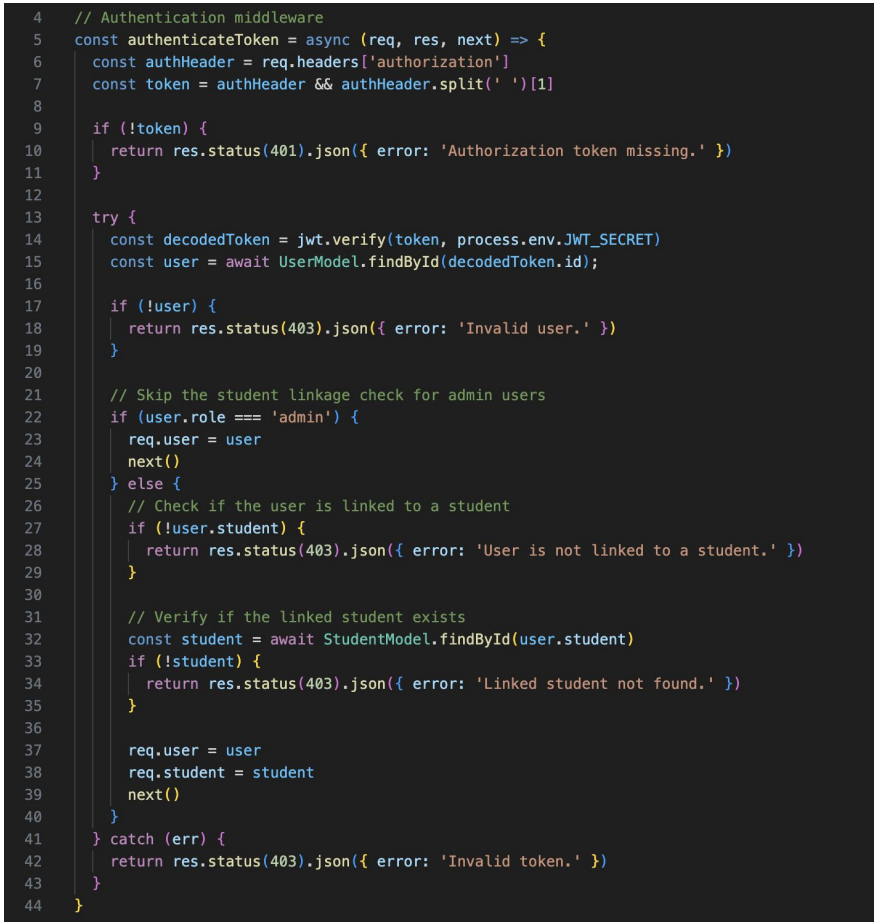
- Vitest and react testing library are used for automated testing
- userEvent is used to simulate actual browser interaction
- `screen.getByRole` is used as the preferred method to query elements in the accessibility tree
- Mocked data is used to pass into the Context Provider

```
describe("Login component", () => {
  let mockedSetUser = vi.fn()
  beforeEach(() => {
    render(
      <BrowserRouter>
        <UserContext.Provider
          value={{
            setUser: mockedSetUser
          }}
        >
          <Login />
        </UserContext.Provider>
      </BrowserRouter>
    )
  })

  it("renders without crashing", () => {
    expect(screen.getByRole("heading", { level: 1 })).toBeInTheDocument()
    expect(screen.getByRole("heading", { level: 1 })).toHaveTextContent(
      "Log In"
    )
  })

  it("accepts user input and changes value", async () => {
    expect(screen.getByLabelText("Email address")).toBeInTheDocument()
    expect(screen.getByLabelText("Password")).toBeInTheDocument()

    await userEvent.type(
      screen.getByLabelText("Email address"),
      "test@example.com"
    )
    await userEvent.type(screen.getByLabelText("Password"), "password")

    expect(screen.getByLabelText("Email address")).toHaveValue(
      "test@example.com"
    )
    expect(screen.getByLabelText("Password")).toHaveValue("password")
  })
```

Code Examples: Back End

Authentication

- Middleware for route protection
- Checks if the incoming request has an appropriate JWT token
- Error handling for non-existent user, student or user that isn't linked to a student in the database



```
// Authentication middleware
const authenticateToken = async (req, res, next) => {
  const authHeader = req.headers['authorization']
  const token = authHeader && authHeader.split(' ')[1]

  if (!token) {
    return res.status(401).json({ error: 'Authorization token missing.' })
  }

  try {
    const decodedToken = jwt.verify(token, process.env.JWT_SECRET)
    const user = await UserModel.findById(decodedToken.id);

    if (!user) {
      return res.status(403).json({ error: 'Invalid user.' })
    }

    // Skip the student linkage check for admin users
    if (user.role === 'admin') {
      req.user = user
      next()
    } else {
      // Check if the user is linked to a student
      if (!user.student) {
        return res.status(403).json({ error: 'User is not linked to a student.' })
      }

      // Verify if the linked student exists
      const student = await StudentModel.findById(user.student)
      if (!student) {
        return res.status(403).json({ error: 'Linked student not found.' })
      }

      req.user = user
      req.student = student
      next()
    }
  } catch (err) {
    return res.status(403).json({ error: 'Invalid token.' })
  }
}
```

Authorization

- Middleware for route protection
- One for only admin access
- One for admin OR linked student access (e.g. updating student details)

```
46  // Authorization middleware for admin
47  const authorizeAdmin = (req, res, next) => {
48    if (req.user && req.user.role === 'admin') {
49      next()
50    } else {
51      return res.status(403).json({ error: 'Unauthorized.' })
52    }
53  }
54
55  // Authorization middleware for admin or linked student
56  const authorizeAdminOrLinkedStudent = (req, res, next) => {
57    if (req.user.role === 'admin' || req.user.student.equals(req.student._id)) {
58      next()
59    } else {
60      return res.status(403).json({ error: 'Unauthorized.' })
61    }
62  }
```

Route: Register

- POST ‘/signup’
- Student will have unique code (the student ID that is created when the admin adds a student to the database) that will link the new account with an existing student profile
- The registered email must also match the email in the student profile (that was input by the admin)
- Password is encrypted
- JWT token is generated

```
// Register an account as a student
router.post('/signup', async (req, res) => {
    try{
        const { name, email, role, studentId } = req.body // studentId has been emailed to user as registration code

        // Password encryption
        let password = req.body.password
        const salt = await bcrypt.genSalt(10)
        password = await bcrypt.hash(password, salt)

        // Find the student by ID
        const student = await StudentModel.findById(studentId)

        if (!student) {
            return res.status(400).json({ error: 'Invalid registration code.' })
        }

        // Check if the provided email matches the student's email
        if (email !== student.email) {
            return res.status(400).json({ error: 'Email does not match.' })
        }

        const newUser = await UserModel.create({ name, email, role, password, student: student._id })

        // return user without password
    const returnedUser = await UserModel.findById(newUser._id)
        // Generate a JWT token for the new user
    const token = jwt.sign({ id: newUser._id }, process.env.JWT_SECRET, {
            expiresIn: process.env.JWT_EXPIRE,
      })

      res.status(201).json({ token, message: `Registration successful, welcome ${name}!`, user: returnedUser })
    }
    catch (err) {
        res.status(500).send({ error: err.message })
    }
})
```

Route: Login

- POST ‘/login’
- Checks that both email and password have been entered
- JWT token is returned if email and password are both correct

```
// Login route
router.post('/login', async (req, res) => {
    const { email, password } = req.body

    // Checking that both email and password are entered
    if (!email || !password) {
        return res.status(400).json({ error: 'Email and password are required.' })
    }

    try {
        const user = await UserModel.findOne({ email }).select('+password')

        // Check the password if user is found
        if (user) {
            const isPasswordMatch = bcrypt.compare(password, user.password)
            // Return jwt token if password is correct
            if (isPasswordMatch) {
                const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {
                    expiresIn: process.env.JWT_EXPIRE
                })
                const returnedUser = await UserModel.findOne({ email })
                res.status(200).json({ token, message: 'Login successful!', user: returnedUser })
            } else {
                res.status(400).send('Invalid credentials.')
            }
        } else {
                res.status(404).send({ error: 'User not found.' })
        }
    }
    catch (err) {
        res.status(500).send({ error: err.message })
    }
})
```

Route:
Creating a student

- POST ‘/students’
- Route can only be accessed with valid admin token
- Class input needs to be a valid (to be chosen from a drop down menu on the front end)

```
// Create a student
// Admin access only
router.post('/', authenticateToken, authorizeAdmin, async (req, res) => {
  try {
    const { firstName, lastName, email, class: classId, photo } = req.body

    // Check if class exists
    const selectedClass = await ClassModel.findById(classId)
    if (!selectedClass) {
      return res.status(404).json({ error: 'Class not found' })
    }

    const newStudent = await StudentModel.create({
      firstName,
      lastName,
      email,
      class: selectedClass._id,
      photo,
      questionOne: '',
      questionTwo: '',
      questionThree: '',
      questionFour: '',
      contactDetails: '',
      quote: ''
    })

    res.status(201).send(newStudent)

  } catch (error) {
    res.status(500).send({ error: error.message })
  }
})
```

Test: Get Students

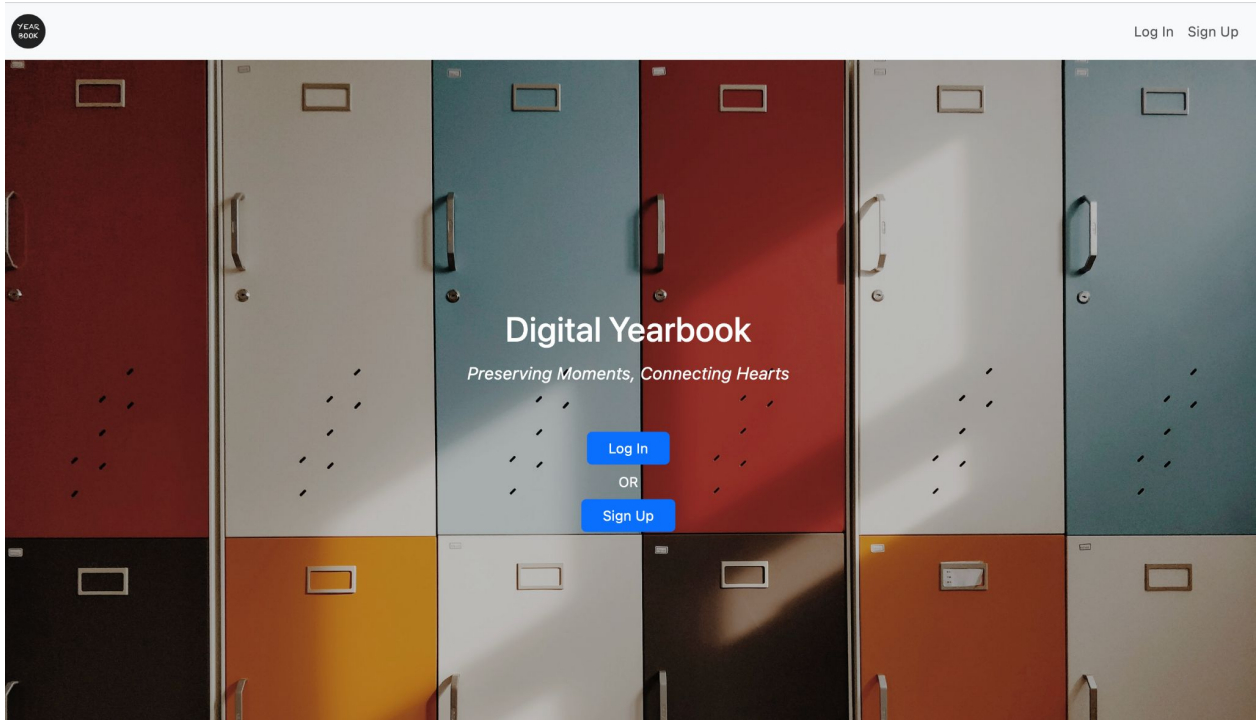
- Sends bearer token with the test request
- Checks that status code is 200 and that JSON is returned
- Checks that an ID of 24 characters exists for each student
- Checks that each student has necessary fields returned

```
// Testing get all students
describe ('GET /students', () => {
  let res

  beforeEach(async () => {
    res = await request(app).get('/students').set('Authorization', `bearer ${token}`)
  })

  test ('Returns JSON', async () => {
    expect(res.status).toBe(200)
    expect(res.header['content-type']).toMatch('json')
  })

  test('Each student has an "_id" of 24 characters', () => {
    res.body.forEach(el => {
      expect(el._id).toBeDefined()
      expect(el._id).toHaveLength(24)
    })
  })

  test('Each student has a first name, last name, class and email', () => {
    res.body.forEach(el => {
      expect(el.firstName).toBeDefined()
      expect(el.lastName).toBeDefined()
      expect(el.class).toBeDefined()
      expect(el.email).toBeDefined()
    })
  })
})
```

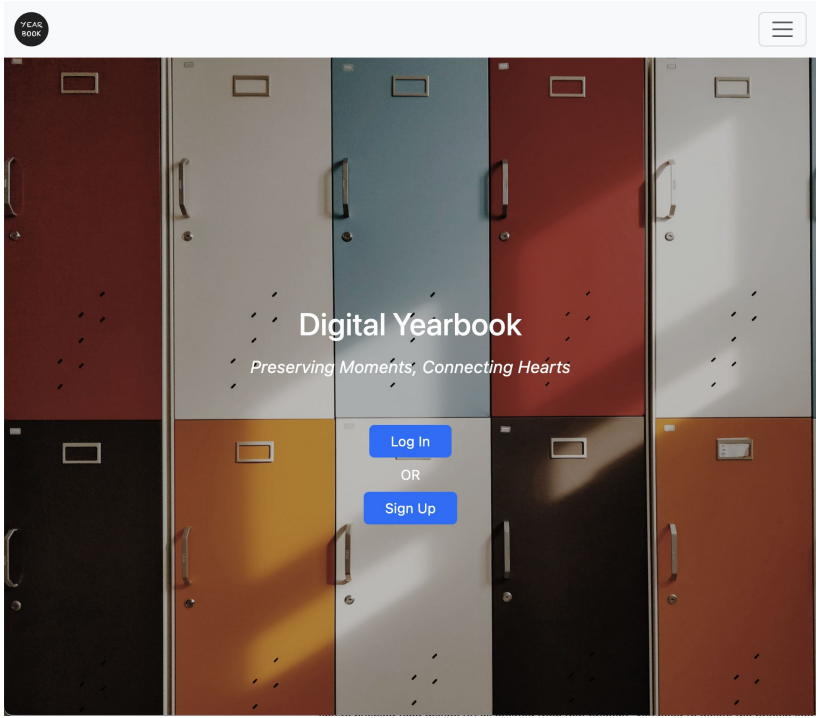
Deployed App

https://student-yearbook.vercel.app/

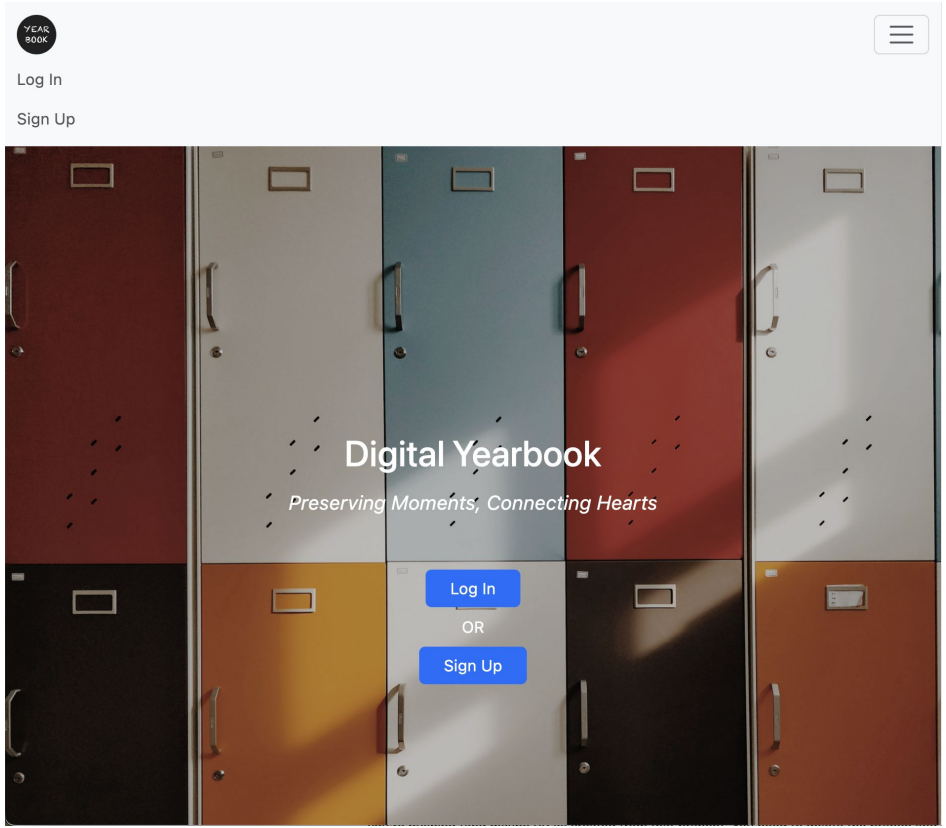
Home Page - Desktop



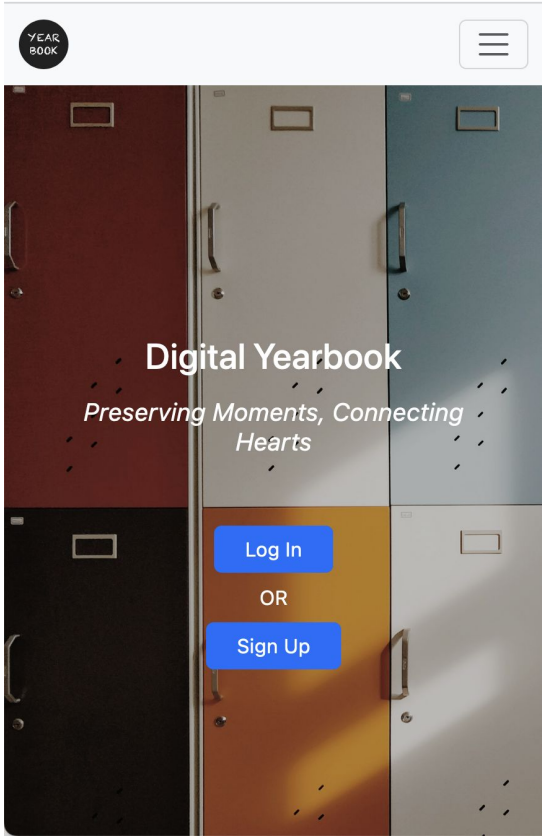
Home Page - Tablet



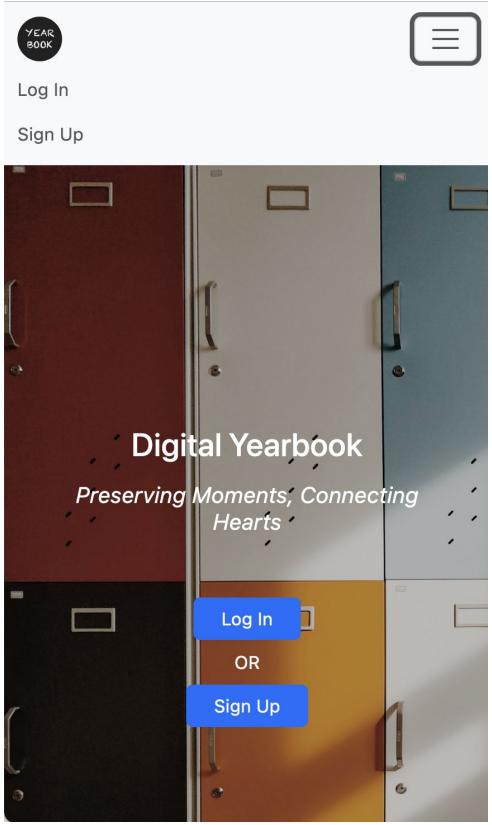
Expanded Navigation Bar:



Home Page - Mobile

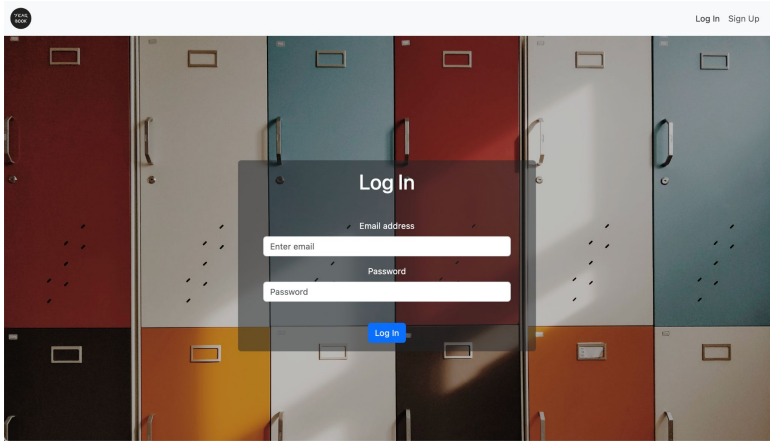


Expanded Navigation Bar:

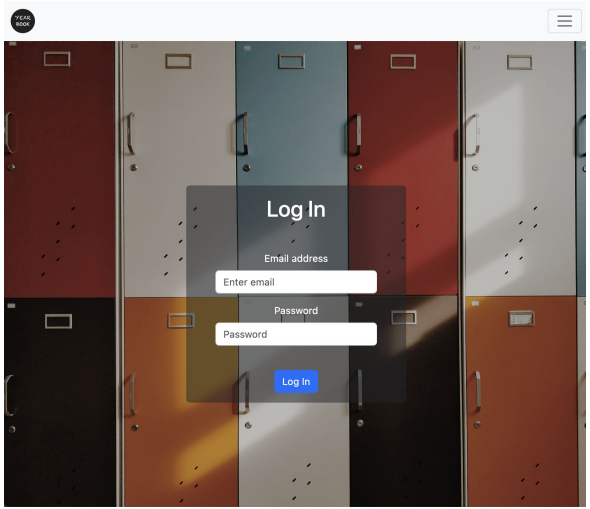


/login

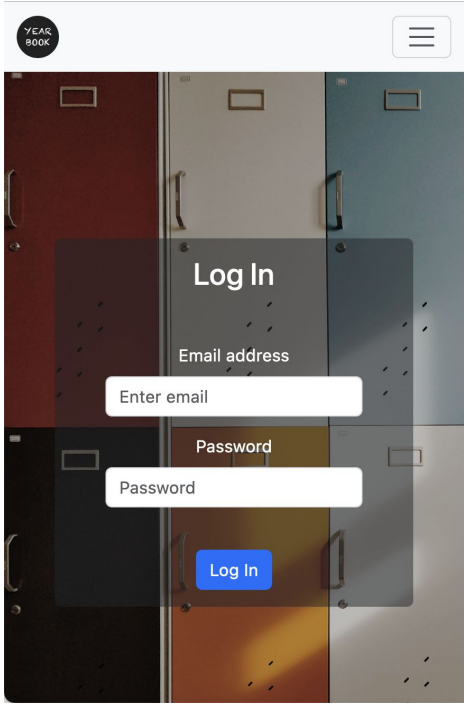
Desktop



Tablet

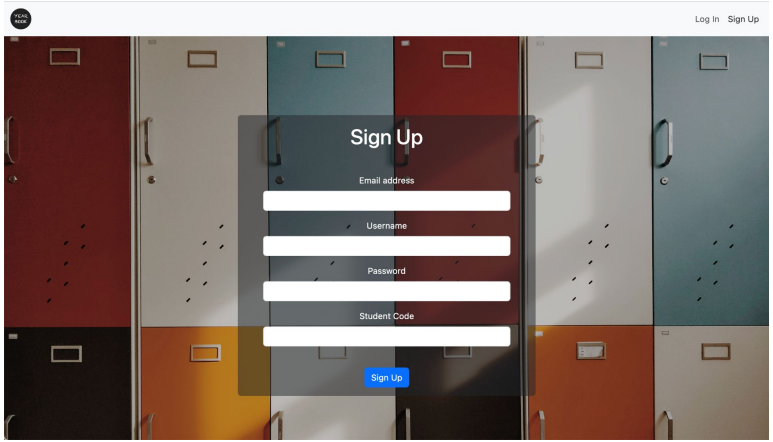


Mobile

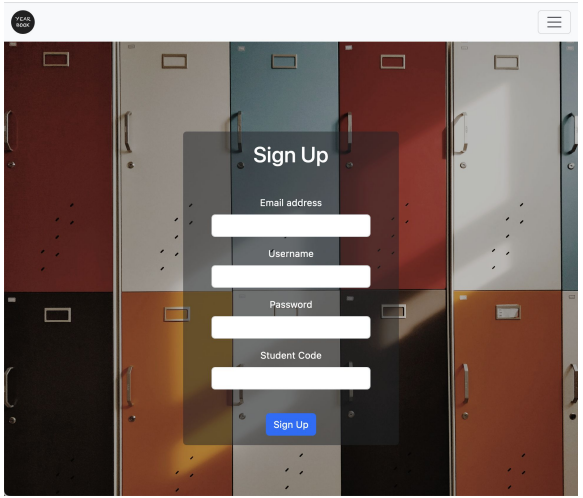


/signup

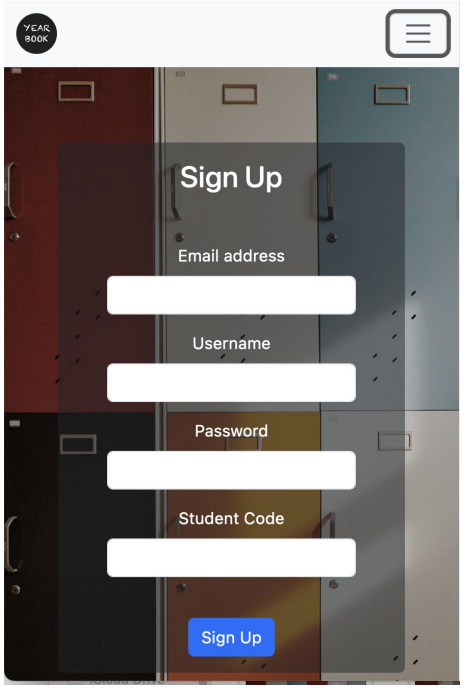
Desktop



Tablet



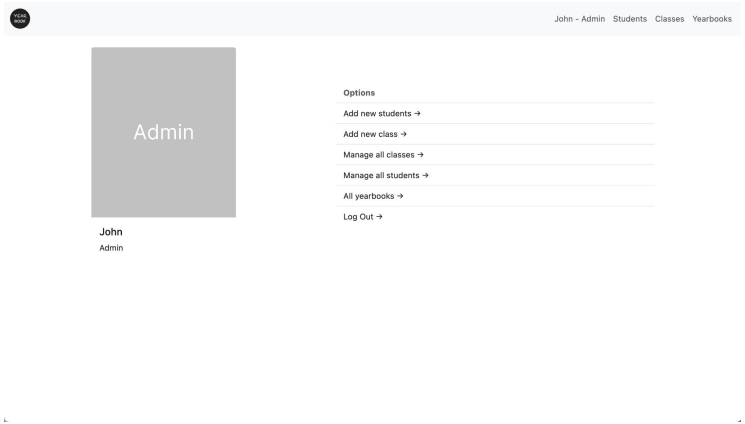
Mobile



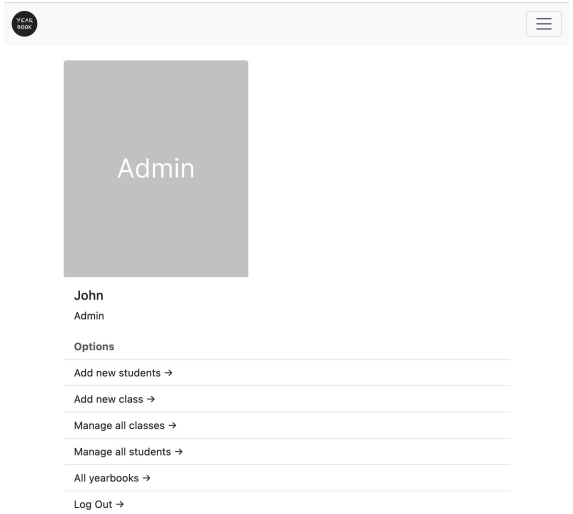
Student can sign up using student code provided by the admin

/account (admin)

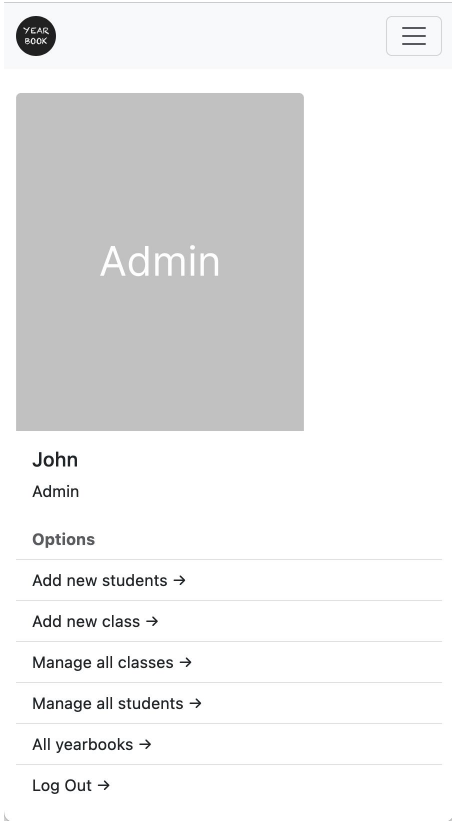
Desktop



Tablet

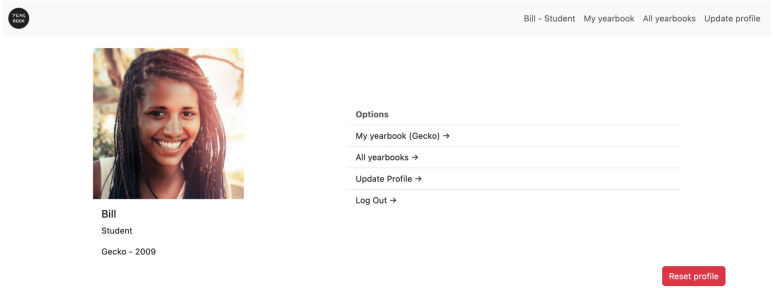


Mobile

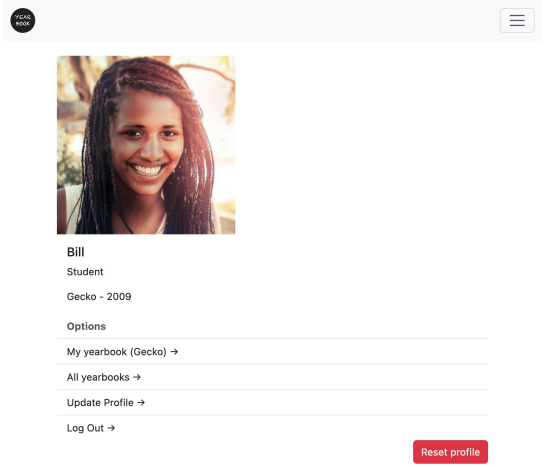


/account (student)

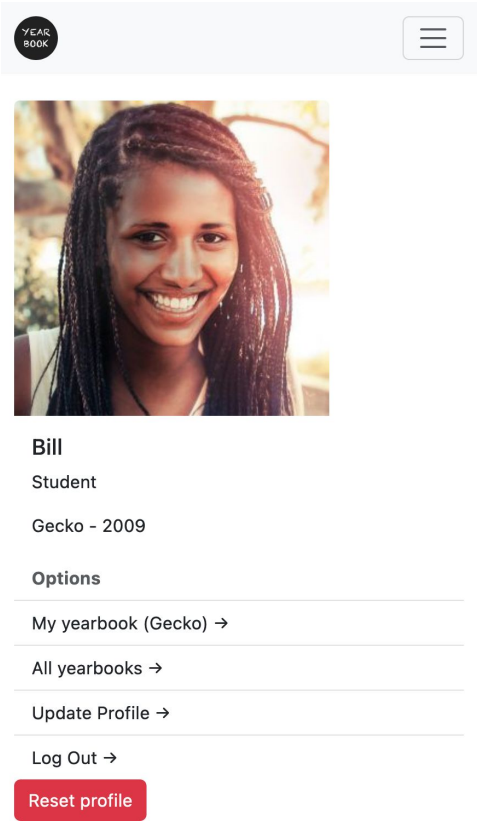
Desktop



Tablet



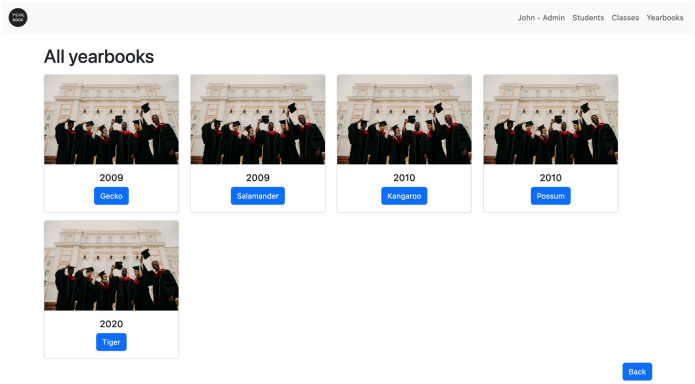
Mobile



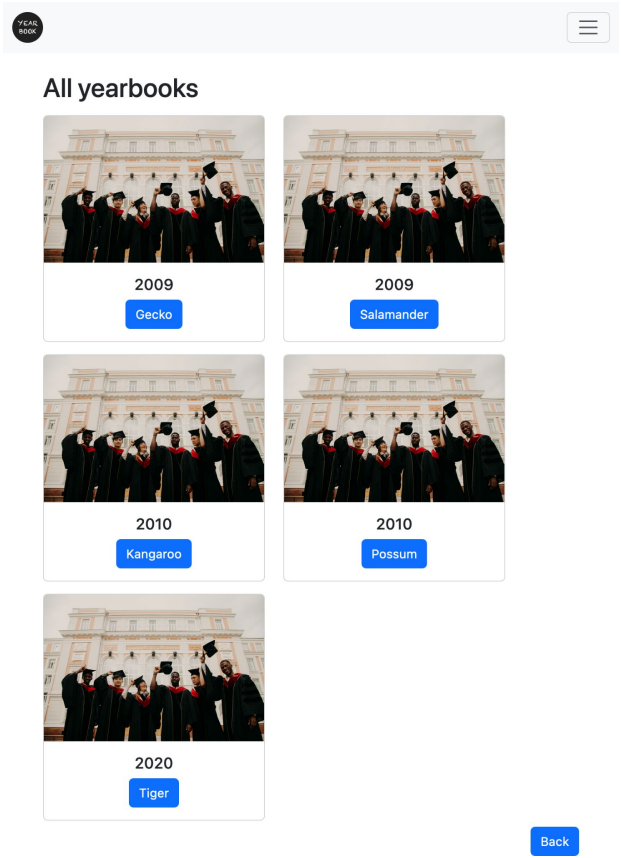
By resetting profile, all information of the linked student will be cleared except for names, email, class and photo link

/classes

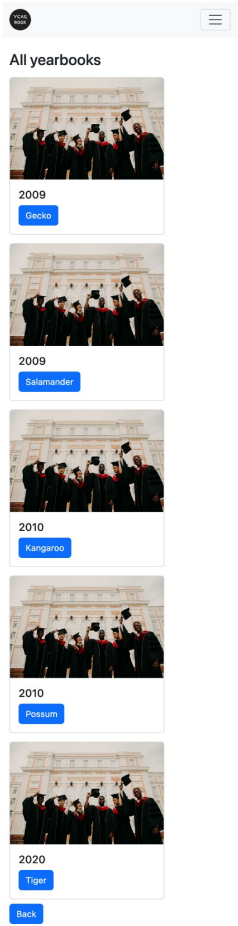
Desktop



Tablet

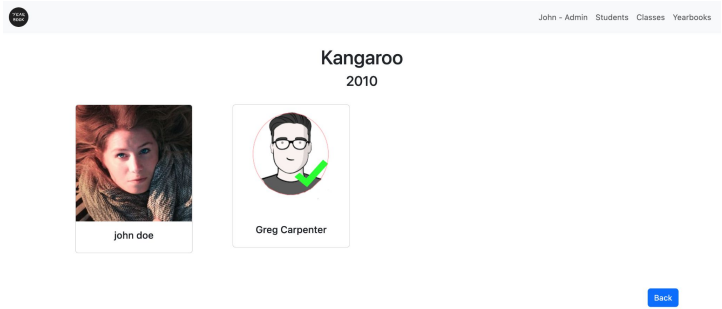


Mobile

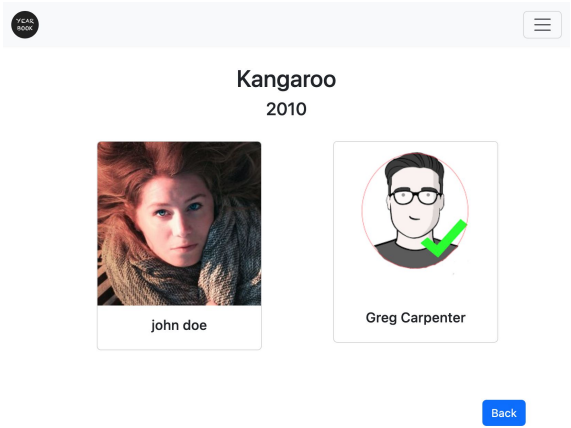


/classes/:classId

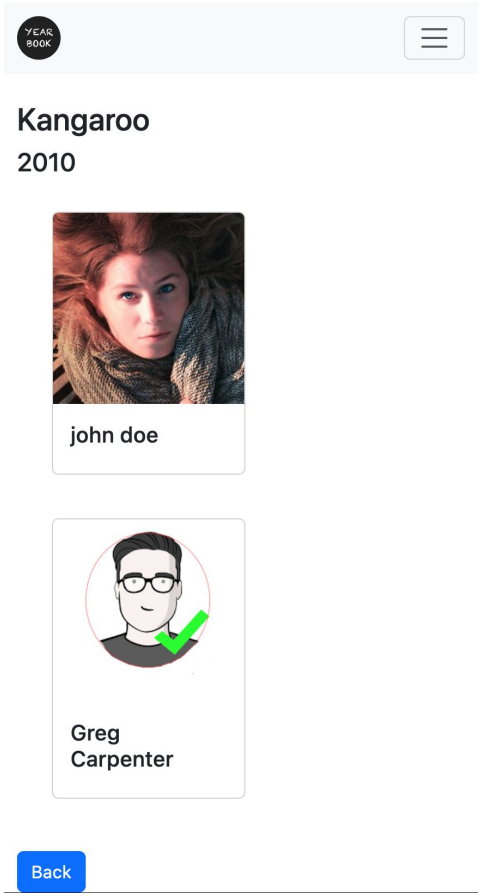
Desktop



Tablet



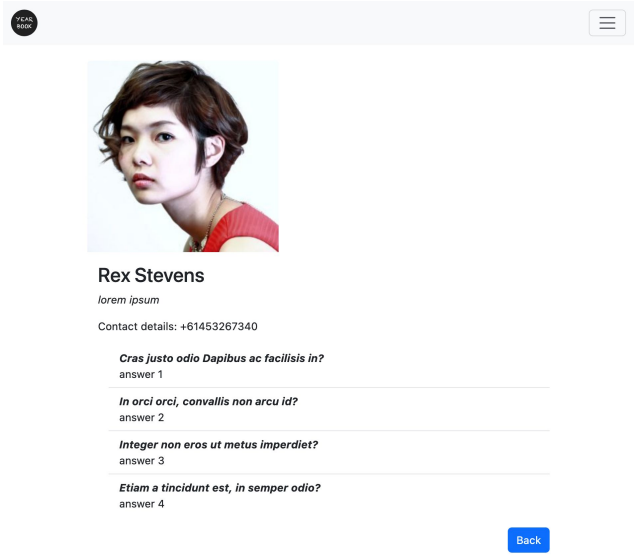
Mobile



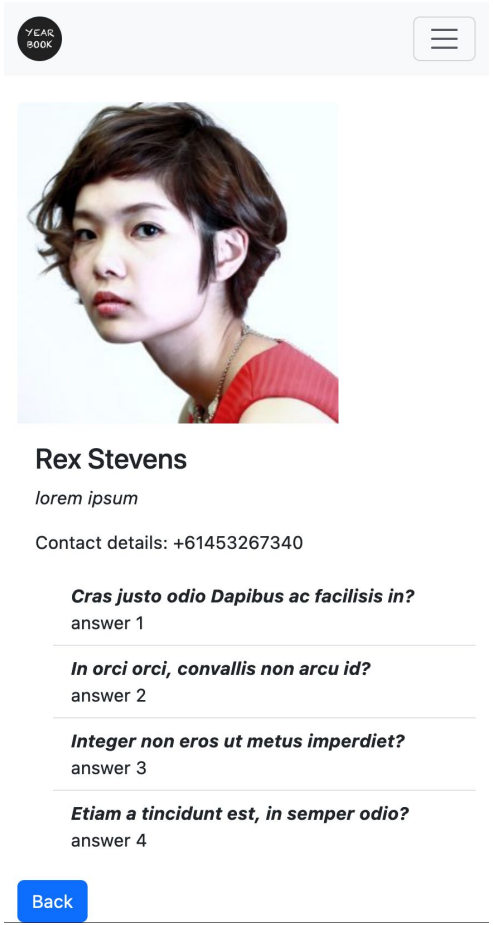
/students/:studentId

Desktop

Tablet

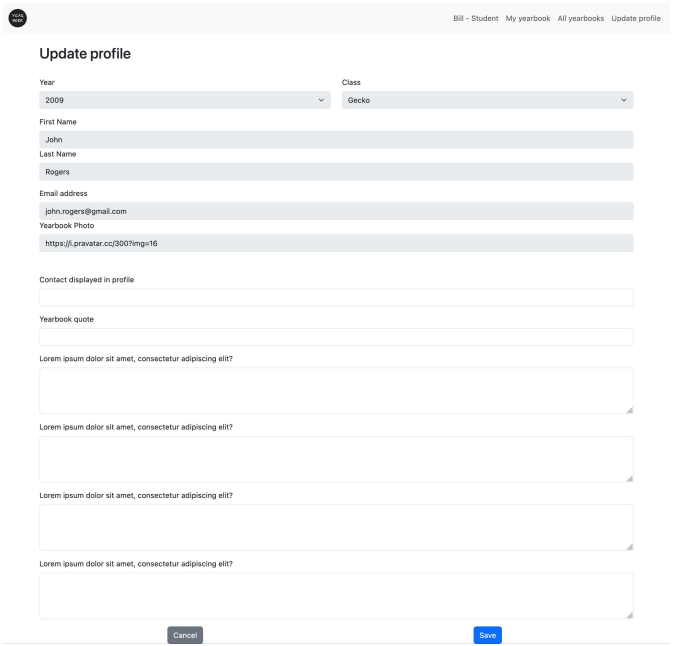


Mobile



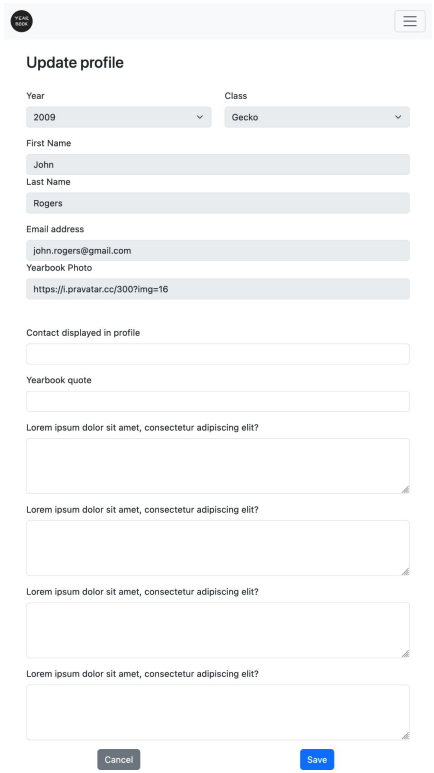
/students/:studentId/edit (student)

Desktop

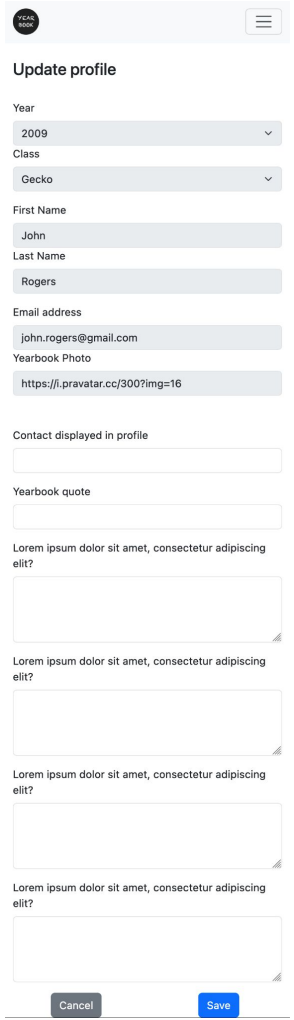


Student can't update critical information including year-class, name, email and photo link

Tablet

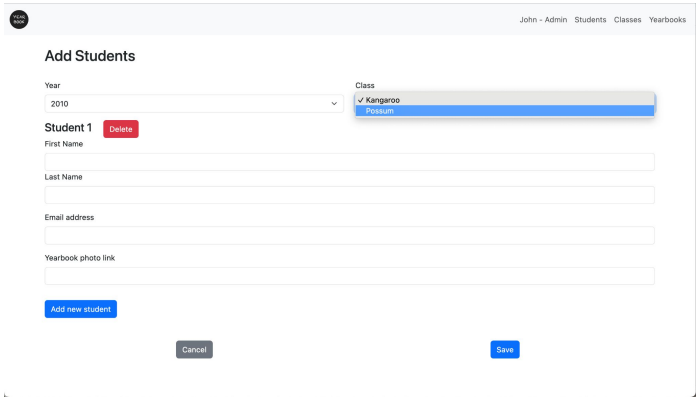


Mobile

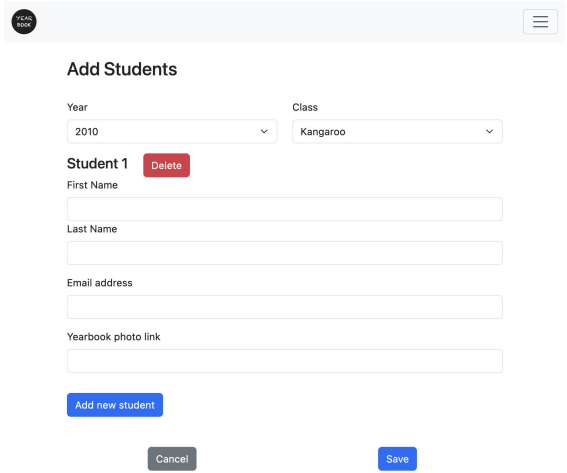


/account/students/new (admin)

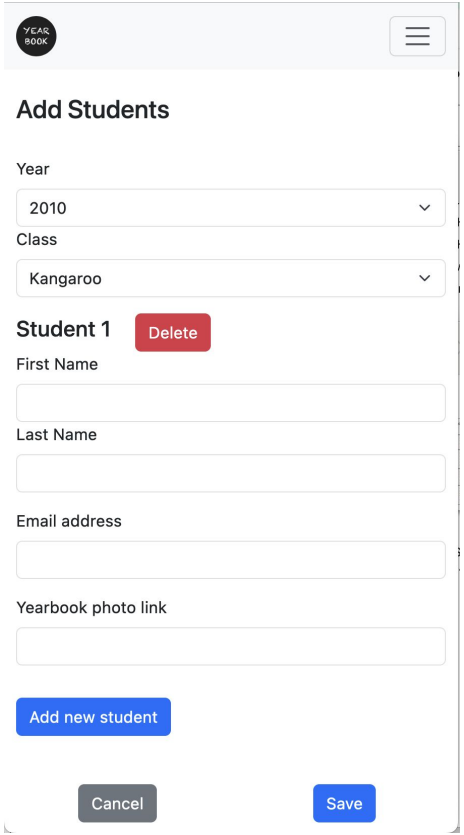
Desktop



Tablet

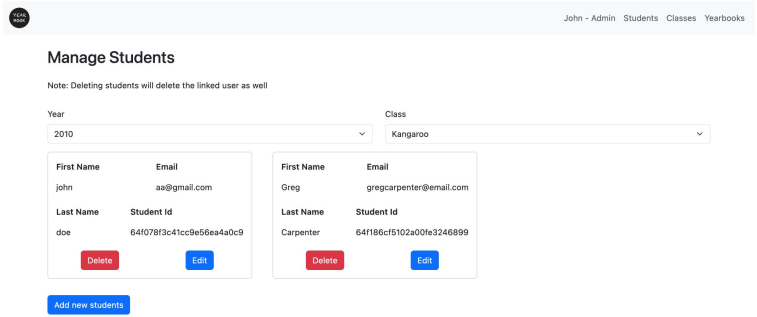


Mobile

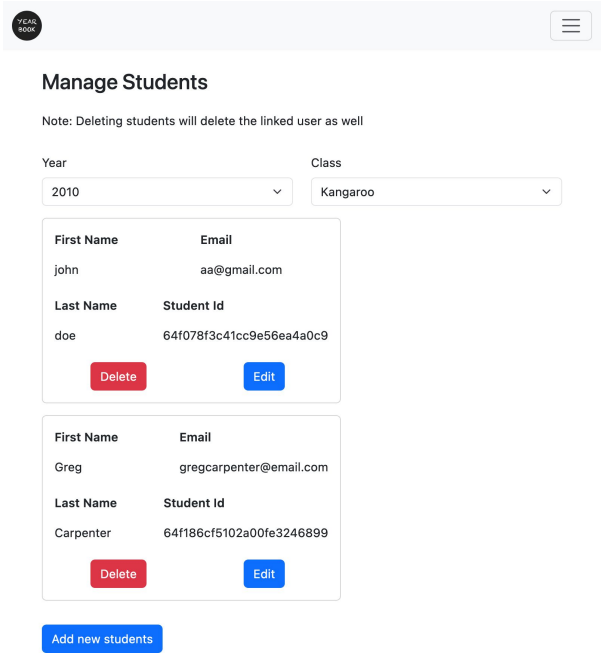


/account/students (admin)

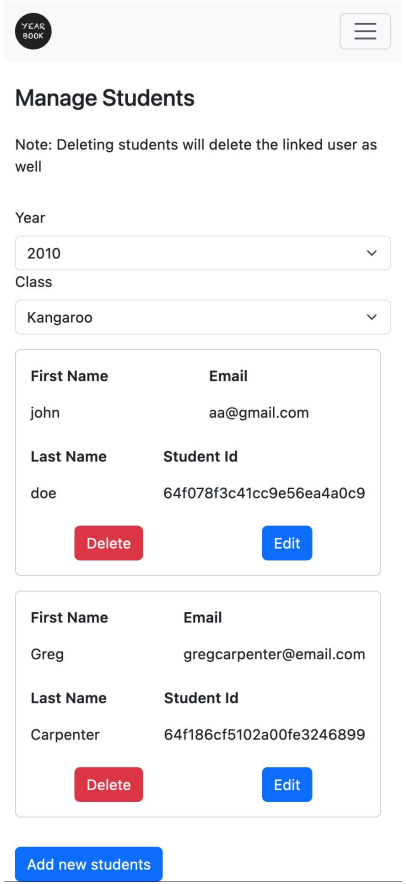
Desktop



Tablet



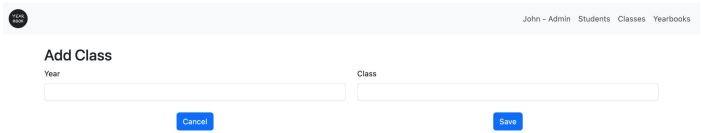
Mobile



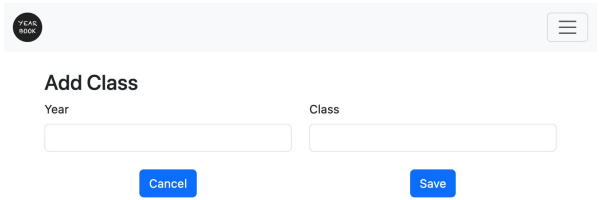
After students are created in the database (from last slide), admin can go to this link to view generated student id. Admin can then send these details to students for them to register in the system (assuming this is done outside the app)

/account/classes/new (admin)

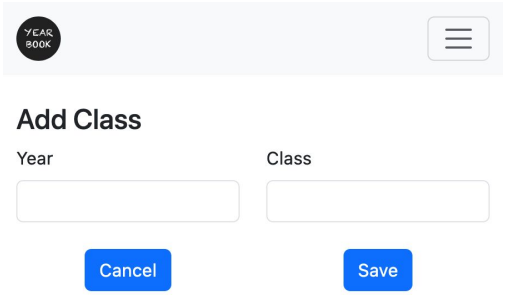
Desktop



Tablet

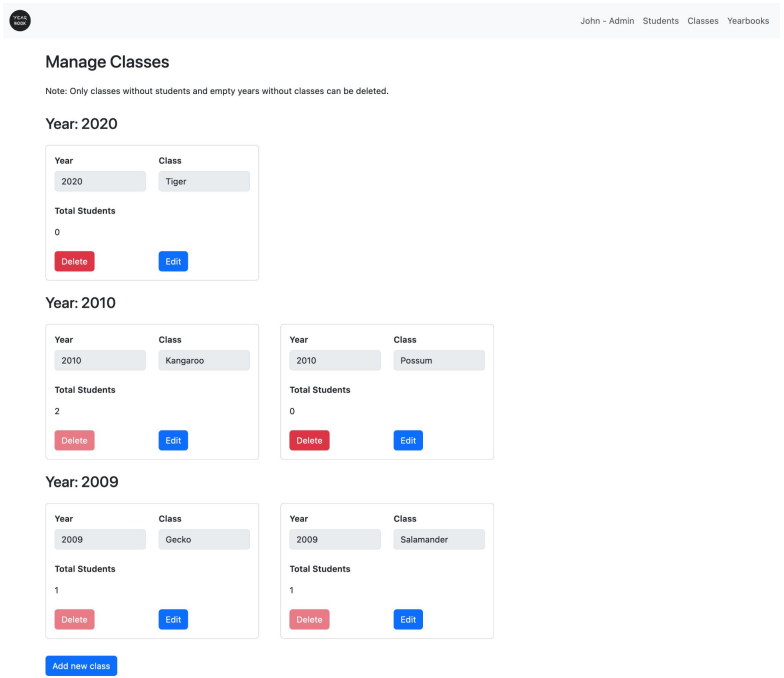


Mobile

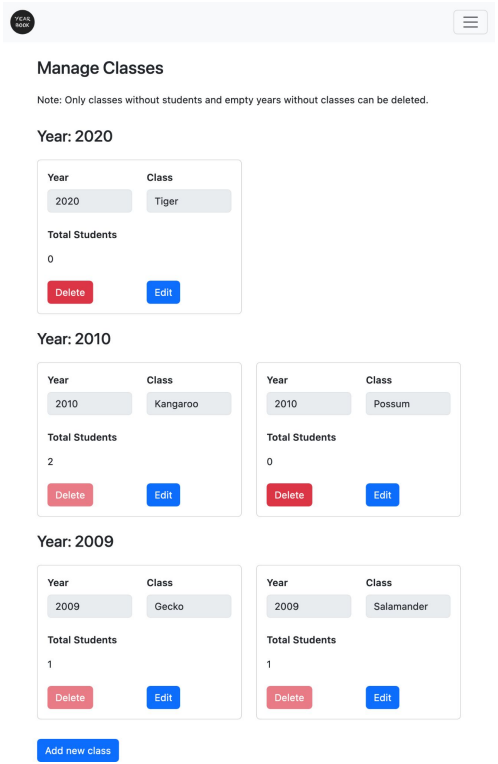


/account/classes (admin)

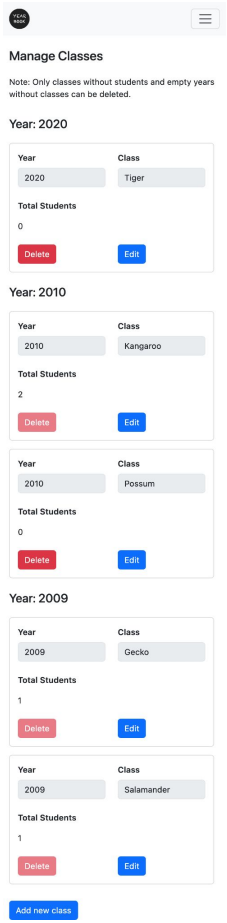
Desktop



Tablet

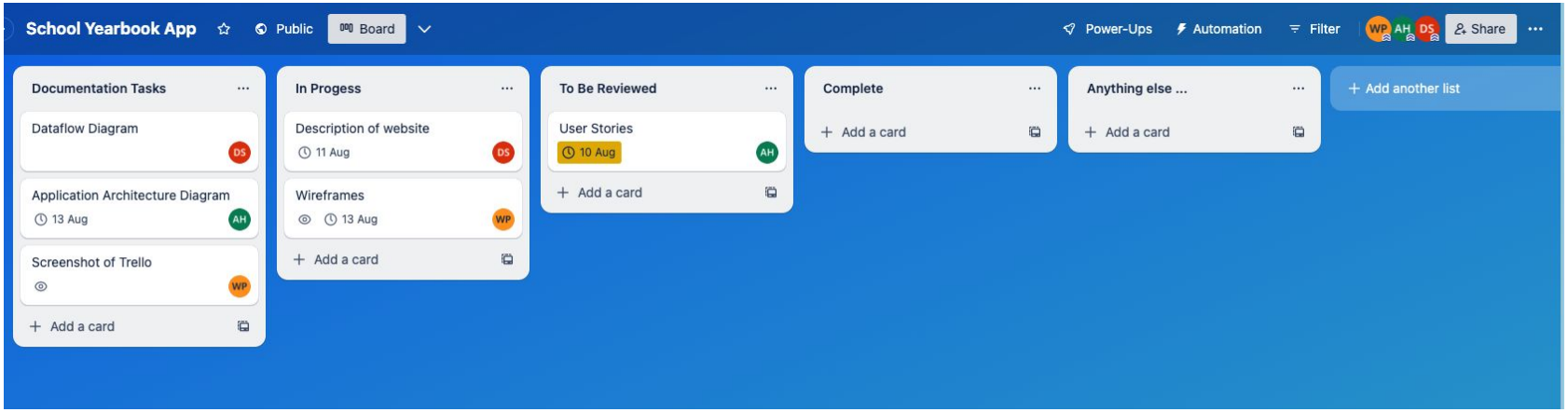


Mobile

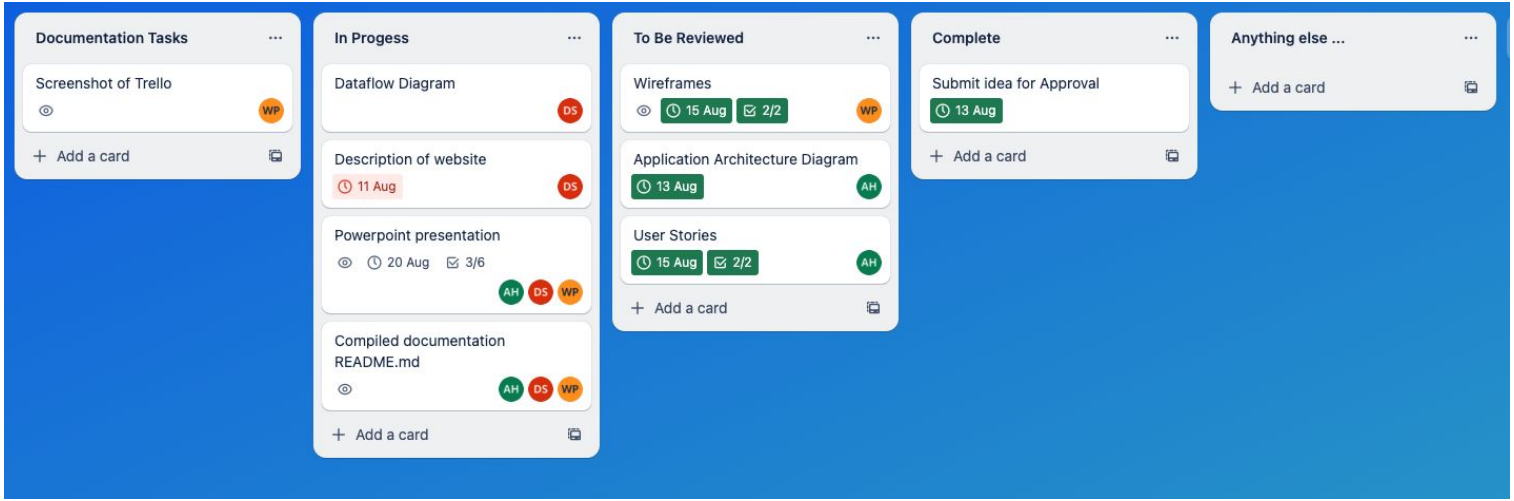


Trello board

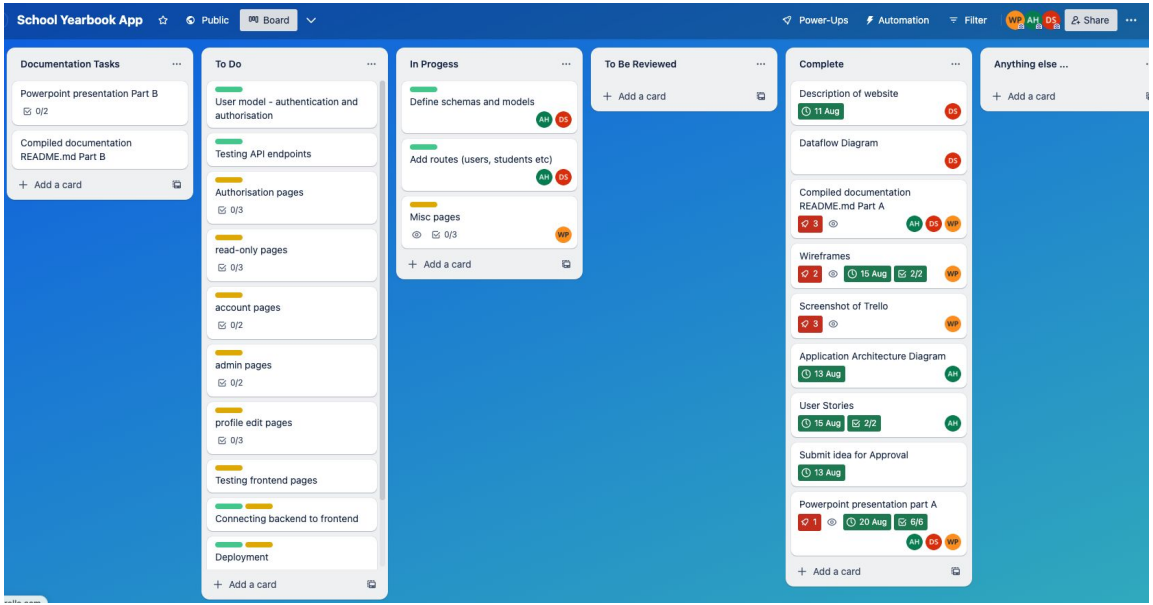
9 August: project start



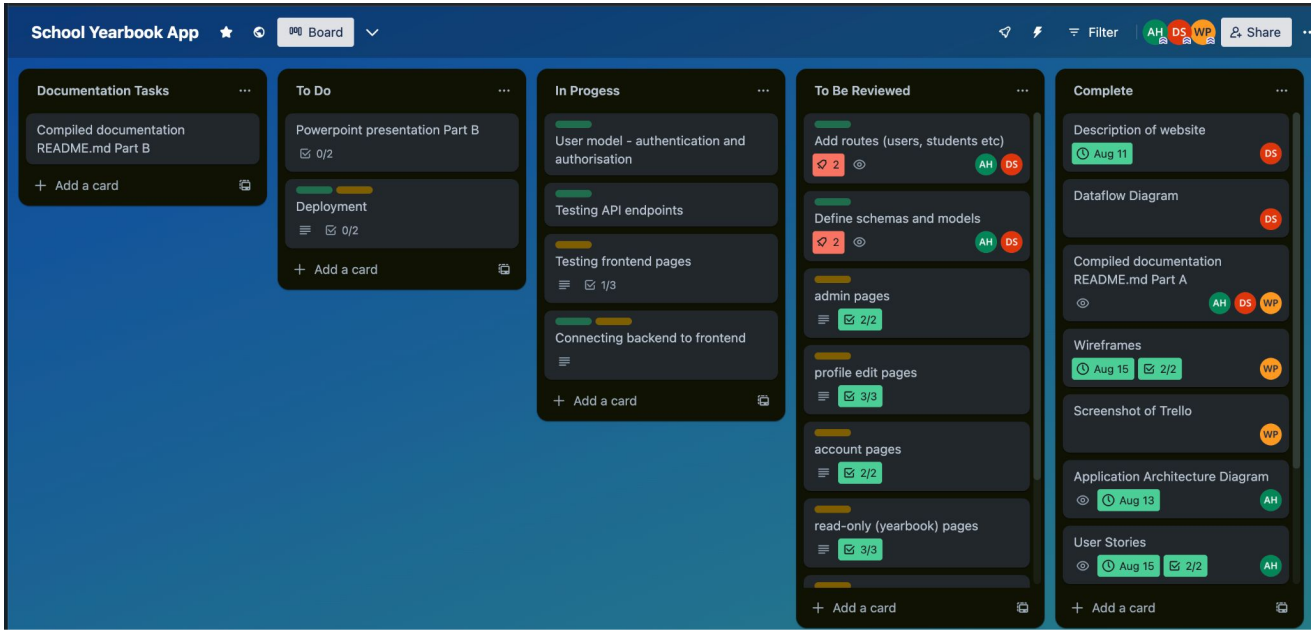
19 August: near-end of part A



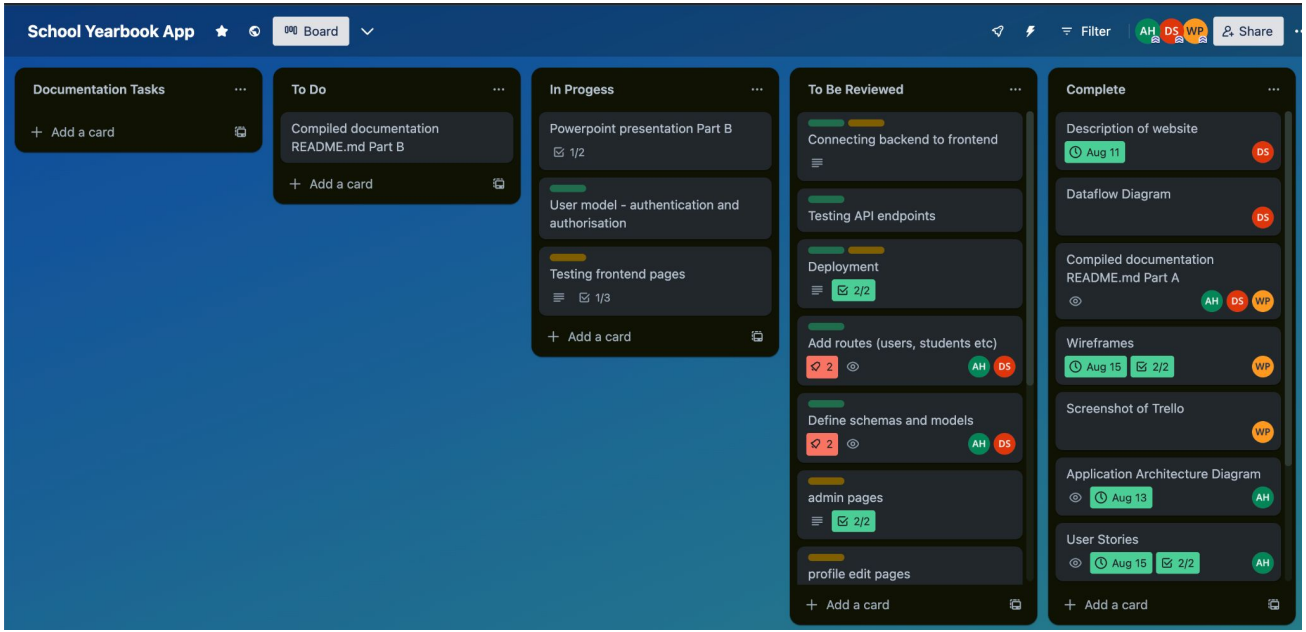
21 Aug: Start of Part B



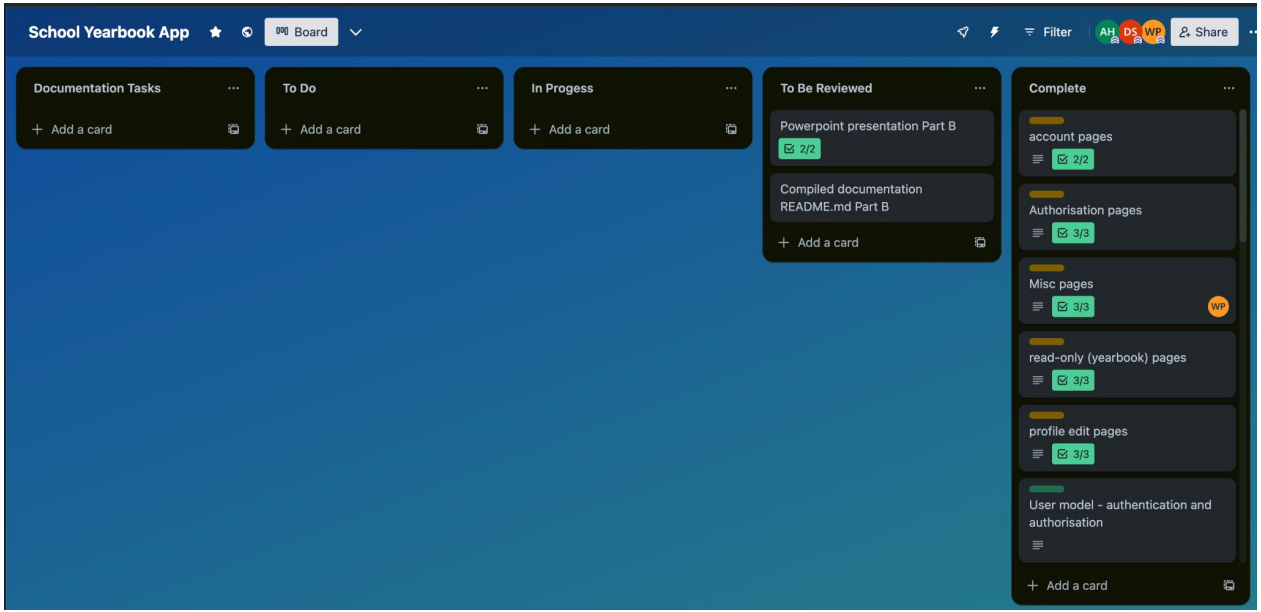
30 August


School Yearbook App
Board
Filter
Share
Documentation Tasks
Compiled documentation README.md Part B
Add a card
To Do
Powerpoint presentation Part B
0/2
Deployment
0/2
Add a card
In Progess
User model - authentication and authorisation
Testing API endpoints
Testing frontend pages
1/3
Connecting backend to frontend
Add a card
To Be Reviewed
Add routes (users, students etc)
2
Define schemas and models
2
admin pages
2/2
profile edit pages
3/3
account pages
2/2
read-only (yearbook) pages
3/3
Add a card
Complete
Description of website
Aug 11
Dataflow Diagram
Compiled documentation README.md Part A
Wireframes
Aug 15
2/2
Screenshot of Trello
Application Architecture Diagram
Aug 13
User Stories
Aug 15
2/2
Add a card

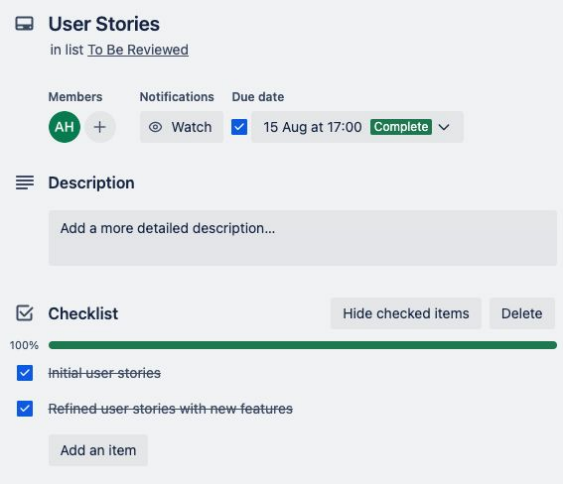
02 September

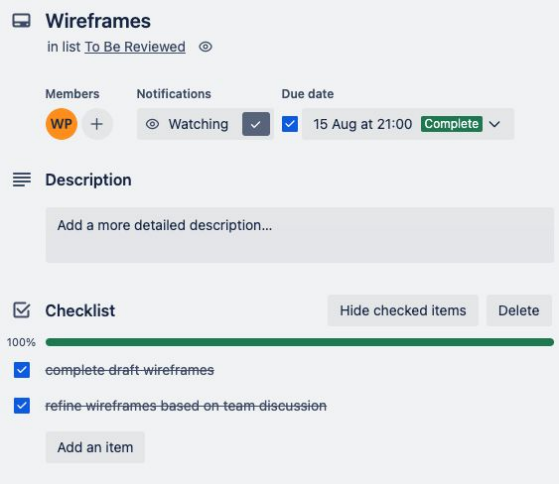


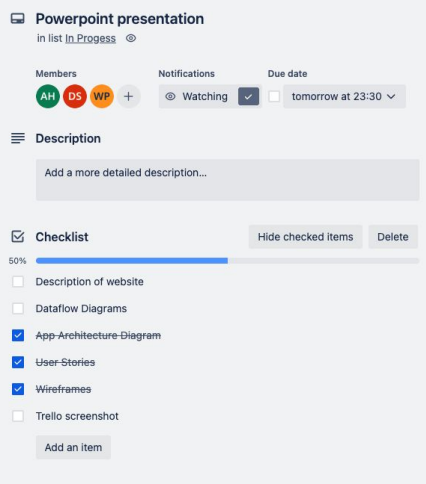
03 September - Completion



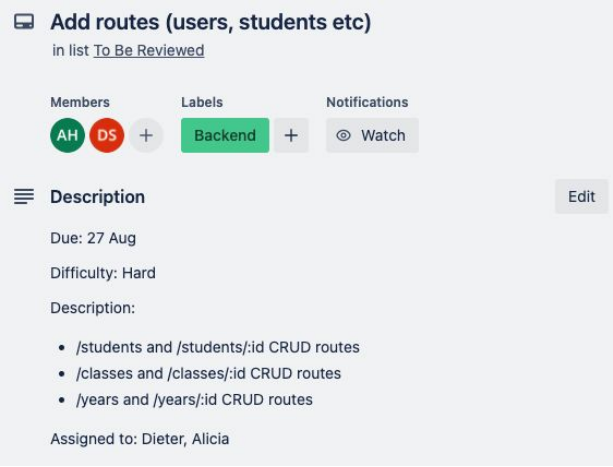
Subtasks (Part A)

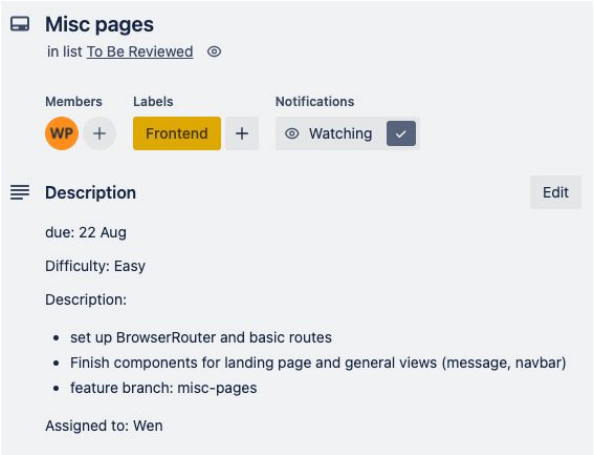






Subtasks (Part B)





More in Trello