

Project Phase 2

What we built

Frontend

[Login and Register](#)

[Form Selection Page](#)

[Form Submission page](#)

[Response Submission Page](#)

Backend

[Express Server](#)

[Server Tests](#)

[Docker](#)

[CICD](#)

[How is this different from what you originally proposed?](#)

Design

[Frontend](#)

[Backend](#)

Technical Highlights

[CICD, Anujan](#)

[NodeJS and Async Behaviour, Jacob](#)

[Answer Schema, James](#)

[Docker, Johnny](#)

[React Libraries, Kevin](#)

[Redux, Lina](#)

[Functional Programming, Ryan](#)

Process

Going Forward: Phase 3

[What we will be build](#)

Final Demo

[Anujan](#)

[Jacob](#)

[James](#)

[Johnny](#)

[Kevin](#)

[Lina](#)

[Ryan](#)

What we built

For P2 we took the tasks derived from our use cases and split them among the team, producing powerful frontend and backend features that facilitated our learning goals outlined in P1.

Frontend

On the frontend we created a number of react pages to create an easy experience for the form fillers to execute the expected use cases.

Login and Register

On the front end we built Login and Register pages that interacts with the back end properly. A JSON web token that is produced by the back end is stored in the localStorage of the browser, which then gets attached to every request the user sends. The user of the system is the form filler who can then interact with a number of pages.

Form Selection Page

We added a Form Selection page, from which the user (Form Filler) can either start a new form, or display a recently edited form response. This allows users the option to complete forms at their own pace, and edit their responses after if needed.

Form Submission page

We added a base form submission page from which we allow users to submit forms (in xml format only) to be used. We have allowed the user to enter an id for the form. This allows users to refer to forms later on when using the form selector. Eventually we would add the ability for only certain users to be accessing this page and to update/make new versions of existing forms.

Response Submission Page

We had the majority of the work for our Response Submission Page built after P1. In P2 the Response Submission Page was further styled by adopting Material-UI components. On top of the styling, the form submission functionality is enhanced over HTML form by using Form Control. We added additional functionality to the Response Submission Page, implementing the logic to deselect other checkboxes in checklist options when a certain checkbox is selected. This is required by the form in certain cases; for example when a checkbox labeled “Not Applicable” or “Other” is checked, the form specifies that all other checkboxes are to be deselected.

We also began implementing the logic relating answers to different questions in the form, as defined in the XML. For example, some questions have sub-questions, which should be disabled unless the relevant choice in the parent question is selected. If that specific choice is not selected, then its sub-questions should be disabled. We implemented this using a React/Material-UI

“Collapse” component and keeping track of the selections in local state, collapsing/un-collapsing the sub-questions as choices are selected/unselected. Another rule we were able to implement is “selectionDeselectsSiblings”. This is a property of some checkbox choices defined in the form XML. It requires that when the checkbox is checked, all other checkbox choices in that question should be unselected.

Backend

On the backend we expanded upon our work in P1 and continued to build the express server, the server tests, and the docker infrastructure.

Express Server

Using NodeJS with express framework we implemented our planned API with a configurable MongoDB instance. This aspect was delivered almost exactly as planned, all of our API endpoints are fully functioning and integrated with database operations. We also implemented all planned non-database operations such as validation of completed forms. The only change from what we originally proposed is the structure submitting answers. Instead of submitting fully completed forms we’ve structured form filling into smaller API calls to post individual answers. An artifact for this change can be seen in this github pull request.

<https://github.com/csc302-fall-2019/proj-Team4/pull/20/>

Within this request you can see the added functionality for answering individual questions here

<https://github.com/csc302-fall-2019/proj-Team4/blob/557e51d1b8a66eb8c2f2b78e3504d036adb130c4/server/routes/response.js#L160>

and the corresponding mocha tests here

<https://github.com/csc302-fall-2019/proj-Team4/blob/557e51d1b8a66eb8c2f2b78e3504d036adb130c4/server/tests/suites/response.test.js#L331>

The implementation of the express server was a significant step in progressing towards our goal as this backbone allowed to frontend team communicates with a working system when implementing the respective ui components. The updated API spec for this implementation is viewable in our P2 artifacts folder under openapi.yaml, where you can view all the endpoints we implemented and our database schema.

Server Tests

Using mocha we’ve implemented various suites of tests for our XML parser, SDCForm management, and SDCFormResponse/SDCAnswer operations. The tests are thorough and cover the backend implications of all use cases outlined in P1. Again the only difference from our originally proposed structure is additional tests for the finer grained SDCAnswer submission.

Docker

Using docker we've containerized our web app to be deployable. The container can be used both for building and deploying. Through docker we've managed to overcome the difference in our development environment and standardize our development. Our project rely on database Our final goal is to be able to deploy containers to production (Heroku) through CI/CD(Travis).

CICD

Along with the dockerization of our app we have built a deployment pipeline using travis ci to build and run tests on all commits. Until we got the dockerized version running we set travis to auto deploy the app on all successful merges to dev. We set up a heroku dyno to host our application with a dedicated mongo db instance for production use.

How is this different from what you originally proposed?

We originally built the backend to cover all the use cases, from the Form Manager's ability to add forms in xml format to the Family Doctor's ability to view it as a persistent link. However, as our front end members are for the most part new to redux, we scaled back on what the front end displays.

We decided to focus on the FormFiller and FormManager use cases for phase 2. In doing so, we created views to create, fill out and edit form responses, as well as adding to the forms themselves by adding an xml file.

Some of the things we decided were beyond our scope for this phase include the persistent link view page, the query page and component testing on front end parts.

Design

Frontend

The frontend is built with React components (with some pre-built components borrowed from Material UI). The entire form is rendered as a component, and it has sub-components for each “section” of the form, which in turn has sub-components for questions, sub-questions, inputs, etc.

We use react redux to centralize the state of the application. In doing so, we are able to extract only what is required at different components. For example, when you go deeper into the types of questions, we do not pass the entire state of the application, because we only need the nodes at that point. Thereby, the props that are passed into Questions, for example, is different to what is passed into Section, as they contain only what is necessary to render their individual components.

Backend

The overall design of our backend centers around the basic workflow of adding forms, creating and editing responses, submitting answers, and retrieving persistent links to responses.

Forms are uploaded to the system, parsed, and maintained in the database as an SDCForm with a version. The versioning allows updated forms to be uploaded and given priority in future retrievals, but keep past responses form dependencies intact.

SDCFormResponses are then the primary means for aggregating the answers to a form. New blank responses can be created in the database, or an existing response can be retrieved.

SDCAnswers are then submitted, and associated with an SDCFormResponse by its ID in the database. By separating SDCAnswers and SDCFormResponses we can efficiently create and overwrite answers in our database by leveraging basic indexing in our key-value store.

When a form response is complete, we simply generate an SCDPersistentLink and associate it with the SDCFormResponse. In the database this is simply a string with the referenceID, which we provide to users for an easy to use URL to retrieve the filled out form. E.g. “ourapp.com/hskxjh” vs “ourapp.com/api/persistent?responseID=abcdefghijklmnop”

When a persistent response is requested, our database schema lets us easily aggregate SDCForm, SDCFormResponse, and SDCAnswers together and deliver a static representation of a completed for response and the subset of answers that were submitted.

We designed the response answers to be their own documents because we wanted to think ahead about how we might query our database. When the SDCAnswers are more easily indexable it will

be easier to leverage database querying techniques to structure queries about the submitted answers as defined in our use cases.. We will explore this aspect of the design in P3.

Technical Highlights

CICD, Anujan

One of the challenges related to the CICD was being able to figure out small errors in configuration between the local builds and deployments. Having to check both the code the team made and the config decisions behind it and translating them to Travis and Heroku was challenging to implement. For example we had .env files to store environment variables, by having those it was easy to develop but since we didn't commit them and were different configs for different users it was difficult to find a way to add these files to the repo to be picked up by CICD. After a lot of research we decided that we would have travis create these files in a script and fill them in with variables that were okay to be public (such as the url) and database related configs would be set in heroku as env variables and having a dummy file for the server. A lesson that I would take away is to consider the deployment when creating the application to begin with so that it would be a painless process to translate locally working configs with deployment configs.

NodeJS and Async Behaviour, Jacob

While working on the backend code, one thing that I got a better understanding of was how nodejs' event loop and async code works. At the beginning, I ran into interesting situations. For instance, calling a function first then using its returned value in the next piece of code led to serious problems due to how node handles async functions and the rest of the code in the main thread. The value would come as null even though I knew very well that the function would never return null under any circumstances. It turned out that the two lines were running in parallel and the second line of code was trying to use the value way before the function call returns. In most programming languages, this would have been totally a good approach as the thread would block until the function returns then use the value in the next line. This intrigued me and I looked more into nodejs event loop and its async behavior. Nodejs event loop runs a single thread but there is a library called libuv which handles asynchronous functions in a separate thread pool. Therefore, the code handled by libuv is executed outside the event loop and in parallel. For my particular problem, the solution was to return a promise in the function that I wanted to call first! That way, the event loop would wait for the promise to resolve then execute the next piece of code depending on whether the promise was actually resolved or rejected. This challenge was a very important learning point.

Answer Schema, James

One of the challenges of the backend was implementing the database schema for our proposed question structure. Rather than creating multiple question types for fields/choices we instead have one answer type SDCFormAnswer with either a SDCFieldAnswer/SDCChoiceAnswer. The SDCChoiceAnswer itself then can have an SDCFieldAnswer for "please specify" types of choices.

This decision was conceptually simple but proved challenging to implement, as many operations on form answers required numerous checks such as `field != null`, `choices[i].length > 0`, `choices[i].field != null`, and required careful design of helpers to fill these attributes. While I feel this design is powerful for generalizing the answers, it required a thorough and challenging implementation.

Docker, Johnny

One interesting thing I learned while writing up the Dockerfiles happened when I was writing the `docker-compose.yml` file. For our `docker-compose` file, we have two containers that are connected through a bridge network. One container is running our web app that serves both the front and backend. The other container is the datastore where we are hooking up our web app. The server portion of the code relies on the connection of MongoDB meaning if we don't have a connection established with Mongo instance, our server will not run. The issue I've faced was caused by this and how containers work.

When the two containers stood up, our web app container wasn't able to connect the mongo instance running on the other container. To debug this process, I've tried connecting to the mongo running on the container from my local machine(host) and no issues were found. After a lot of debugging the issue happened to be a timing issue of the two instances. Even though both containers stood up at the same time, the services within didn't stand up in sync. The mongo daemon wasn't running when our server code initiated the connection to db. To resolve this issue, the web app container was modified to wait for the Mongo to come up and initiate the connection. An improvement that could have been done is for our server code to be decoupled from the database initialization thus not being affected by the status of database instance.

React Libraries, Kevin

Even though I had previous exposure to React in the context of React Native, using React for web development was a huge challenge. I had to learn things like using React Router to route between pages, and using Material-UI not only for styling, but also for form controls. These are the things that do not have exact counterparts in the mobile development using RN, but are specific to React on web environment. On top of that, I had an ambition to learn Redux, hoping it would solve the problems of managing states and props; it was sometimes unclear where to place them and sometimes even the correct placements required the props to be passed down multiple levels below. While Redux did solve that specific problem by having global state store, I realized that it came with a cost of increased complexity of the whole app.

Redux, Lina

This being my first foray into front end programming, react redux is a tool that induces many headaches. In creating the Form Selection page, I was given insight into how the store works, and in dispatching actions to obtain the recently accessed list, among others, I began learning why the store was essential.

One particularly interesting design decision was when I had to decide how the form page knew which form was chosen. Initially, I assumed react's state was not something that redux used, since

it had its props. So all the googling in relation to passing a state on seemed to be of no use. So I decided I would just load the form, because it would be stored in the props and thus accessible in the next round. When I conferred with my team members, we actually found out that you *can* pass a state, and it's just hidden in the props! That was pretty interesting for me, since I'm using react in CSC309 right now, and the differences of storage in the two are quite fascinating.

Functional Programming, Ryan

We were able to make use of an interesting application of functional programming; closures. The form is structured as a tree of questions in the XML, and so during rendering it is constructed as a tree of React components. However, during the parsing of the XML, the tree is flattened; the nodes of the tree are returned as a single list. The parent/child information (i.e. edges of the tree) is preserved within the nodes of the list; each node has a unique ID, and each node stores the IDs of its children. The tree must be reconstructed from this list, and the major problem is that when any question in the form is rendered, it needs to be able to find all of its children so that they can in turn be rendered as sub-questions.

The first step was to create a mapping of parent IDs to child IDs, so that when any question is rendered as a component, the code should be able to find the IDs of all of its children (all questions which are sub-questions of that question). After retrieving the ID, the actual child object can then be found in the list returned from the parser. We then created a closure in the context of this mapping and the list, which takes a parent ID, uses the mapping to retrieve all childIDs, and then searches the list to create a new list of all the child objects, and returns this list of children. This closure is passed to every question as it's rendered, which then passes it to all of its sub-questions as they're rendered, and so on. This greatly simplifies the code for rendering the form.

Process

Our process was very similar to what we outlined in P1. We have been meeting three times a week during or after class/tutorial. This has been sufficient and we have not required planning meetings outside of this. These in-person meetings have been effective as a standup meeting, where we could discuss issues from ongoing tasks. We then take our tasks and are individually productive, using our Slack workspace for our team where we can regularly update each other on our progress and inform the team of problems that arise. The one issue that arises from this system, is that since team members are working on these tasks in their own time, we are rarely dedicating time to the project at the same time. Because of this task completion can become staggered and members can be left waiting on each others work portions / github reviews. This is simply a consequence of being busy uoft students and is difficult to workaround without dedicating communal work hours.

We continued to work with the Kanban process, with tickets and tasks tracked via a private Trello board. This system has worked well for overall tasks. Shortly after P1 we realized we were not adequately tracking our use cases. After discussions with the instructor and TA we retro-actively completed these and it highlighted an issue with our trello management. When filling out the use cases and the associated tasks, it became a bit of a matching game trying to determine what tasks we had on the board went with which use cases. We have since gone and tried to improve the 1-1 relationship between use case tasks and trello tasks, in the future this process could have been improved by taking the time earlier to design the use case tasks and associated trello cards.

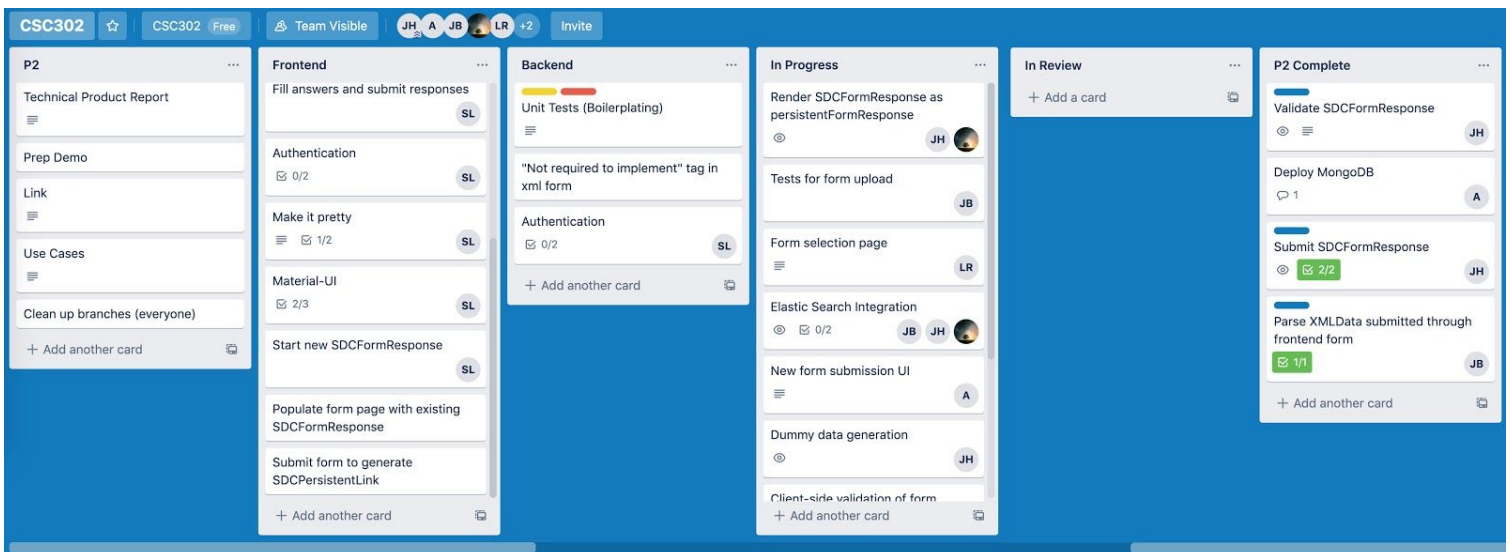


Figure 1. Trello board (full size in phase2/artifacts)

We still have two main branches on GitHub, master and dev. Master contains the most recent production-ready build. Team members work locally and push their commits to their own branch

on GitHub. They will then submit a pull request from their branch to dev, and assign team members to review their code. Once reviewers have approved the changes, the team member is free to merge. This system has worked well, the only downside so far is we do not have a dedicated branch manager, and each member is responsible for cleaning up after themselves. This has resulted in periods of numerous stale or extraneous branches. We have simply been addressing the bloat in meetings but in the future a dedicated repo manager may have been beneficial.

















<input type="checkbox"/>	 4 Open ✓ 21 Closed	Author ▾	Labels ▾	Projects ▾	Milestones ▾	Reviews ▾	Assignee ▾	Sort ▾
<input type="checkbox"/>	 Use Cases ✗ #24 by hooksjam was closed 2 hours ago							
<input type="checkbox"/>	 Form Page Styling using Material-UI and Authentication for Back End ✓ #23 by leesan70 was merged 2 days ago • Approved							
<input type="checkbox"/>	 Response backend ✓ #20 by hooksjam was merged 14 days ago • Approved							 3
<input type="checkbox"/>	 moved XMLformData changes to a new branch ✓ #19 by makuerj was merged 12 days ago • Approved							 4
<input type="checkbox"/>	 Merging P1 requirements ✓ #18 by wesgur was merged 24 days ago							
<input type="checkbox"/>	 P1 wrapup ✓ #17 by wesgur was merged 24 days ago • Approved							 3
<input type="checkbox"/>	 display checklists correctly ✓ #16 by ryanapilado was merged 25 days ago • Approved							 5
<input type="checkbox"/>	 Fix CI Tests Make Import Case Sensitive ✓ #14 by AnujanM was merged 25 days ago • Approved							
<input type="checkbox"/>	 Fix dev failing. Add xml property that determines whether question is radio or checkli... ✓ #13 by hooksjam was merged 25 days ago							 5

Figure 2. Pull request history (full size in phase2/artifacts)

Going Forward: Phase 3

What we will be build

Upon completion of P2, we had about 90% of our use cases supported in software. The only remaining tasks we haven't completed for our planned use cases are the answer querying and remaining form management frontend features.

So for P3 we will finish the UI in the frontend to manage SCDForms by adding delete functionality. We are currently looking into integrating elastic search into our backend for P3 and will also create the UI in the frontend to query our form answers.

Component testing is another area we want to focus on in P3, as it ties in to some of our learning goals. In particular, we would ideally test the form component, as it is the most complex. With the time restraints for P3, however, we will be testing parts of the Form component, such as individual question types.

We will also use P3 as an opportunity to polish our app overall and work out any remaining kinks.

Final Demo

For the final demo we will talk about how we worked as a team and the challenges of organizing such a large project. We have each chosen a portion of the software and will talk about the respective design/implementation challenges of that portion. We decided that each of use would outline what we plan on talking about below.

Anujan

For the demo, I will be presenting on the CICD for our application from code to PR to merge/test and then deployments on Heroku.

Jacob

For our demo, I will talk about how our APIs upload XML forms to the server to be converted to SDCForms. I will also talk about how our app achieves text search with elastic search.

James

For the demo I will go over the API spec and specifically discuss how we implemented the response and answer storage in the backend.

Johnny

I can go over how our overall project builds and the containerized version of our web app. Show step by step of our build and explain what happens in each step.

Kevin

I will describe authentication work done for back end and front end and how form submission works using material-ui form control.

Lina

I'll be discussing how the form selection page works, in the actions required to obtain form information from the database and add it to the props. Additionally, I may also speak about how we did component testing.

Ryan

I'll describe how the form is rendered once the front-end has received the output from the parser, and describe the implementation of the logic which controls how it responds to user input (deselection, disabling, etc.).

With this we expect to give a unique demo of our experience with this project, and provide insight into the successes and challenges of implementing the structured data capture application.