Faculty of Mathematics and Natural Sciences

Alexander de Battista Kvamme, Halvor Einar Hagenes, Jens Andreas Thuestad, Sindre Rudrud Kristensen, Thomas Storli

INF219

Early pregnancy risk score – web tool

May 2021

Department of Informatics

Technical report

Researching information about pregnancy can be a hard and time-consuming task. There is a large amount of research on various topics relating to pregnancy and complications surrounding it. Because of this, tools have been developed to expose certain risk factors in a person's habits and medical history. This project endeavours to create a single, user-friendly resource that can calculate a wide variety of pregnancy-related complications, and convey that information in a useful manner.

The project has been designed from the ground up to be extendable and to support an arbitrary number of questions and complication calculations. This gives us the unique position to create a useful tool for private citizens and possibly also medical professionals. As this is supposed to be a tool for the general public, there has been a good amount of work put into localization, dynamic rendering, and general ease-of-use. Because of this, we believe that this tool has the potential to be helpful to a wide array of people.

Table of contents

Intro

Context, background information

During pregnancy, women are at risk of various complications affecting their individual health (gestational diabetes, pre-eclampsia) as well as fetal health (pre-term birth, miscarriage, fetal heart defects). There exist many websites that do a single risk calculation, but some are difficult to use and understand the results.

There are many factors to consider when it comes to pregnancy. Some factors are specific to a single complication, while other factors are related to several complications. However at times it can be difficult to know what to look for.
Below are some factors that can increase the risk for complications:

- pregnant at age 35 or older
- pregnant at a young age
- having an eating disorder like anorexia
- smoking cigarettes
- using illegal drugs
- drinking alcohol
- history of pregnancy loss or preterm birth
- carrying multiples (twins or triplets)
- diabetes
- cancer
- high blood pressure
- infections
- sexually transmitted diseases, including HIV
- kidney problems
- epilepsy
- anemia

For most people this does not tell them anything. A good web tool might help the person in question to decide if they need to consult with a physician. However there are not a lot of good online tools that do this risk calculation, and as far as we know there are not any websites that do more than one risk calculation.

Most pregnancies have little to no risk, but some women will experience complications when they are pregnant. With early detection and preventive care, they may reduce risk to both herself and the baby.

Our client had read and compiled information from over 30 sources about pregnancy-related complications and made a set of questions based on those. The user then receives a score based on the answers they give throughout the questionnaire. Although such a questionnaire might be helpful for medical professionals, it is not a particularly user-friendly method to assess themselves. This is where our project may have the potential to enhance the usability, reachability. It is also important that the user does not have to have any prior technical knowledge in these fields.

Our client wanted a user-friendly website and mobile application where users were able to assess themselves and receive the probability of receiving the different complications.

Some of the most common complications:

- high blood pressure
- gestational diabetes
- preeclampsia
- preterm labour
- a loss of pregnancy, or miscarriage

(Cafasso, n.d.)

Goal and Motivation

The goal of this project is to build a web tool for pregnant women to feed in their individual data and obtain the risk scores for various pregnancy complications.
We want to give women and healthcare workers a tool that helps them calculate risk, either at home before going to the doctor so that the women can ask questions about the risks and get help, or the doctor to do a quick risk assessment and then be able

to check more specifically on the different risks.

This can help women in the risk group and does not know it can check and ask questions and find help. This can lead to early risk detection which can lead to lowering the risk of complications and in the end lead to a better pregnancy period. In the end, this is important for all people that are involved in pregnancy and who are pregnant.

Another goal of this project is that the web site is easy to use and to understand. This is because the user, pregnant woman or healthcare worker, both should be able to use it and understand it. This is important because if a pregnant woman want to know the risk before going to the doctor she would need a tool that tells her but in easy terms.

Project Scope

So this is not a diagnostic tool, this should not be used by a private citizen as a set diagnosis. This should therefore only be used as a general guideline, and the results are not in any way absolute. After getting a risk calculation, the user should go see a doctor or a medical professional for help and further information.

This is not a tool to calculate the risk of getting a diagnosis that is not related to being pregnant. This means that if you have a heightened risk of getting gestational diabetes, does this not mean that the user has a heightened risk of getting diabetes, outside of pregnancy.

Report Structure

We are first going to talk about the project, Why it is important and relevant. We are also going to talk about how we plan to solve the project. After which we are going to go more in depth on the choices we made in the beginning when it comes to choosing the tools and frameworks that we used. After which we are going to be talking in depth of how we worked and how the different parts of the website works, we will be staring at the design stage and we will end on the result page.
Then we are going to go through the results of the page, and discuss the project.

In the end we are concluding.

Project Description

Client’s Goal and Requirements

Our client’s initial goal was to have a website that would calculate the risk of Gestational Diabetes Mellitus for pregnant women, based on their individual health status. If this was successful, and the timeframe allowed it, it was also desirable to include other complications that could occur during pregnancy.

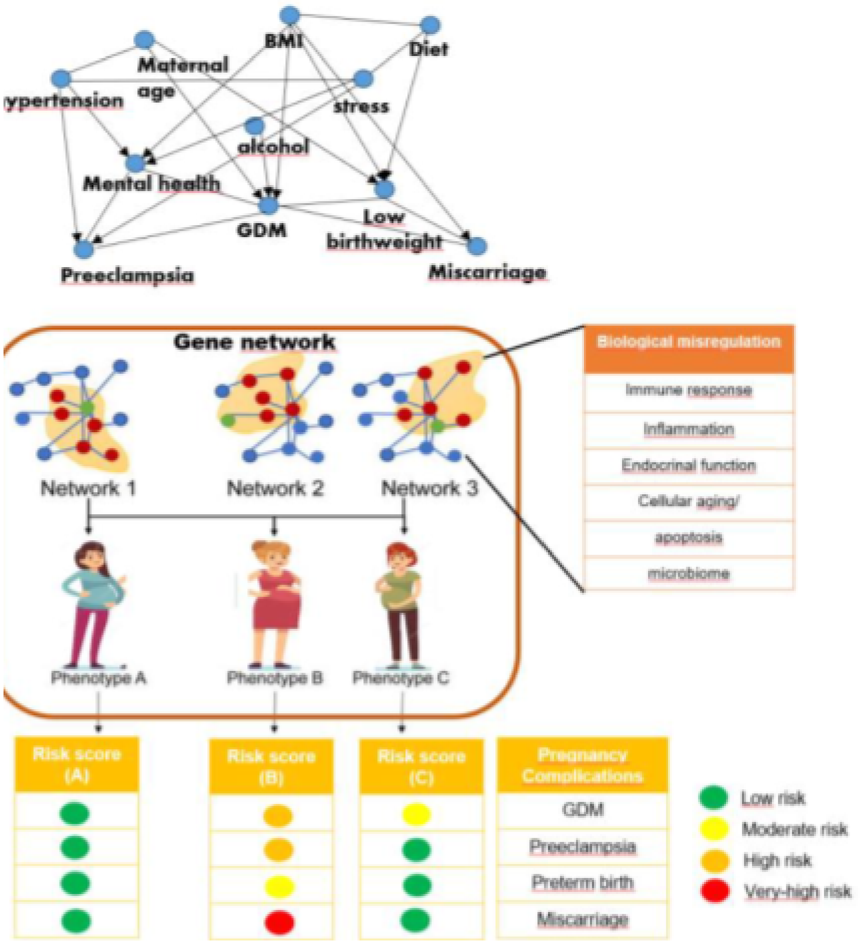
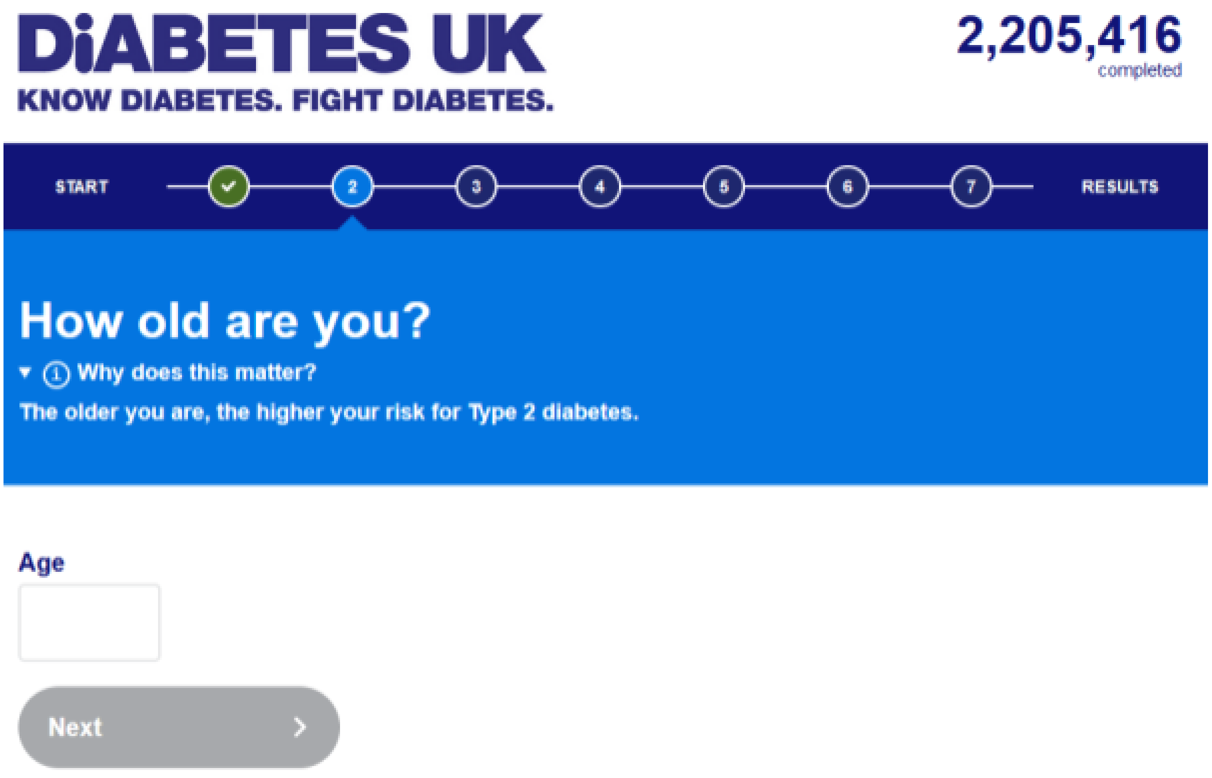


Illustration of biological misregulation and variables that can lead to miscarriage (gathered from the client’s presentation of the project)

Inspiration and Relevance

An example of a finished product was shown to us, and our client wanted something that had more or less the same core functionality and design. A frontpage with description of the tool, easy to navigate through the different questions, and a result page that displays the different results the user got with their respective risk score.


DiABETES UK
KNOW DIABETES. FIGHT DIABETES.

2,205,416 completed

How old are you?

Why does this matter?

The older you are, the higher your risk for Type 2 diabetes.

Age

Screenshot of the Diabetes UK test site from The British Diabetic Association

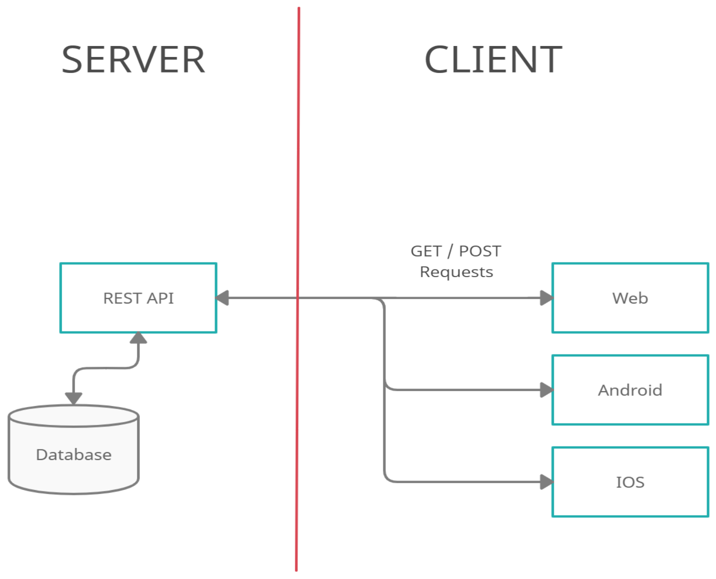
(Diabetes UK et al. n.d.)

The tool itself is no substitute for a real consultation in a clinic, and the calculations are only as good as the research that it is based upon. However, it can still be of significant use for someone that are planning to have a baby, or are wondering if they should consult with a physician. It could also be of use to hospitals and physicians, as they could use this tool to find out which areas that the patient could develop complications. We thought that the idea of this tool was brilliant, and wanted to contribute in any way possible to make it as good as possible. It had the potential to reduce potential complications, and even save lives!

Core Concept

The Early Pregnancy Risk project consists of two independent systems. One being the REST API backend written in Python, and the other being a multi-platform frontend developed using the React-Native framework. Both systems will be developed independently with no direct communication between them, only through HTTP.

As shown in the simple diagram below, the backend and frontend are only going to communicate through a single connection to the backend. No direct database calls



or algorithms will be needed; because of this we can handle all the processing on the backend, and only worry about rendering on the frontend-side. This abstraction will allow us to work on the projects independently without the need for constant dialogue between the two teams, and also giving us the ability to take advantage of different technologies.

As both projects are completely separate, it also means that they can be deployed independently, giving us the ability to scale the services as needed without wasting resources. The frontend will have a minimal amount of static assets and text, this data will be sent from the backend on demand, giving us the ability to change information on the website/apps instantly, without having to resort to redeployment or an update to the APKs.

The main philosophy throughout the development of the projects has been modularity, giving us - and future maintainers - the ability to quickly change or add code. Making the client behave in a dynamic way, and letting the backend take on more responsibility, is one of the key concepts we have followed during the development process.

Project Design

Approaches and Planning

While we were planning, one of the key goals was to develop a product that was easy to maintain and upgrade further down the line. This meant that while it would have been easier and faster to develop it all in one program like Vue.js to get a MVP, it would also mean that we would have to rewrite a lot of code if we wanted to add more to it. Considering that we had a lot of time to do this project, we decided that the best way would be to split it up with a separate backend and frontend. We decided on using Django for the backend. The Django framework offers a very scalable, responsive and secure backend, all built on Python. It is trusted and used by some of the most popular sites on the internet and has a lot of benefits to a service that is meant to be available 27/7.

Initially, we wanted to use the Vue framework for the front end. Vue offers a vast library, is very lightweight, and has a fast DOM, which we found met all of our requirements. After initially settling for Vue, we then asked ourselves if it would be desirable to the client to also have applications for iOS and Android. Seeing as this tool is something that would be used by practitioners at hospitals, our client expressed that this was something that she would want, but the main focus was still to get a functional website.
This either meant developing apps for 2 new platforms or by using the React Native Framework.

Tools:

- Git
- GitHub
- GitHub Desktop
- Figma

Development methods:

- Django
- React Native
- REST API

Programming environments:

- PyCharm
- Visual Studio Code
- WebStorm

Planning the project has mainly been done in GitHub, as this allowed us to work remotely in a rather efficient way. It also has tools such as milestones and issues. The latter has been used quite frequently, all from fixing bugs to new feature ideas.

During the development process, we have tried to include the client as much as possible for continuous evaluation. This allowed us to make sure that we were on the right track in regards to our client needs. It also meant that our client could be more involved in the decision making process.

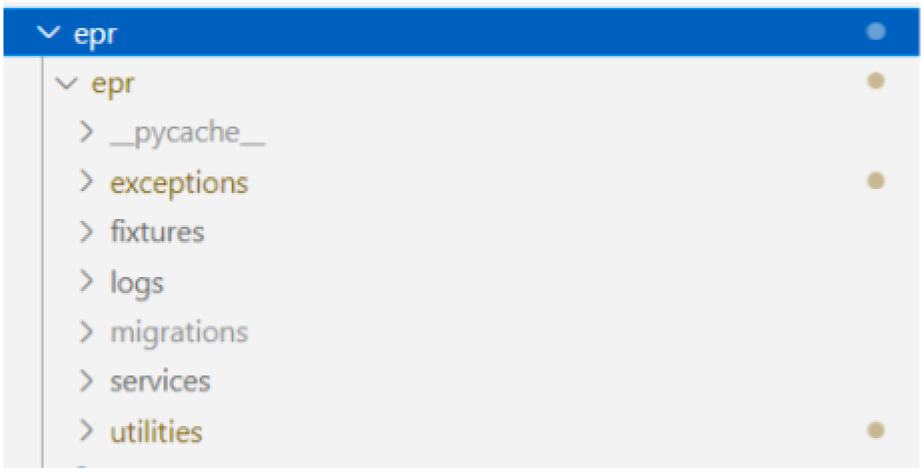
Backend Framework

The backend of this project is written in Python and uses the Django Web framework for the server and Django REST for handling API calls. Python is amongst the most popular programming languages today and allows for rapid development. Everyone on the project also had prior knowledge of the language, which meant that it would be easy for anyone to assist in the development of the backend if needed.

Ease of development, scalability and reliability were our top priorities when choosing a web framework, and Django's Web Framework, therefore, seemed like a natural fit. Flask, which is another web framework for Python, was also considered during the planning phase of the project. Flask only offers a very “bare bone” and minimal server out of the box and gives developers the power to manually extend which features are installed. Django, on the other hand, comes with a lot of ready-to-use features.

Django Web Framework offers seamless integration with a variety of database services (such as MySQL), which makes the development of the database easy. The database can easily be edited through a web-based admin panel, which means that our client can maintain the database even after we have left the project. It is also a widely used framework, with websites such as Instagram and Pinterest are using Django's Web Framework for their services (Stackshare, n.d.). Future maintainers are therefore likely to already know the framework, which will make maintenance much easier. Several people working on the project had also worked on the framework in the past, which meant that the development of the backend could start right away.

When designing a web server, it is important to structure the components of the project logically. This will improve maintainability for future developers, and make it easier for current developers to structure their thoughts. A lot of care was therefore put into the folder structure.



Folders are used to keep the webserver organized. All files that are related to exceptions, for example, are found in the “exceptions” folder.

Python’s virtual environments are used when developing the webserver. A virtual environment in Python is an isolated Python instance which we have full control over. This enables us to choose the exact Python version that is used, and which packages to install alongside the Python instance. This ensures developers that their application will work on all PCs that are using the same Python virtual environment.

Frontend Framework

The initial prototype of the webpage was designed in Figma. This is a great tool for creating a dummy site, to show all the different pages that we would need to implement. We then needed to verify that this was what the client actually wanted. While developing the website, we have had regular meetings with the client, to ensure that we developed within the required specifications. This meant that it was more desirable for us to incorporate an Agile development style. This allowed us to continually show the progression to our client with the early delivery of a working prototype. It also meant that continuous development would be rather flexible, and allowed us to evolve the product, instead of rewriting it all.

React Native offers a way to develop for the web, Windows, Android and iOS all with the same codebase. There are of course platform differences that one has to account for, but the main codebase remains the same. There are also some restrictions though compared to Vue or React. One can only use native components for the applications, which at the moment is rather limited. For us, this meant making a decision to exclude some functionality from the web to mobile applications, like graph drawing.

Early Pregnancy Risk

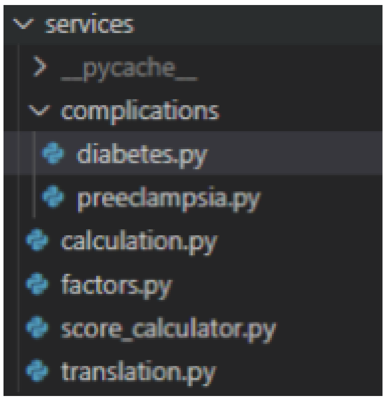
Backend

Python

A big part of the project's backend includes the python code that calculates the risk of each complication. There are many different complications that could occur during pregnancy with many different factors that have an impact on the results for each of them. Therefore, there was no easy way of making one single calculation model to handle all the risks.

We started by making a calculation class for calculating the risk of complications. By using this class, we could make a calculation object based on the user input received from the frontend. Since the scores for each individual complication are calculated differently based on the user's answers, we started by making a function that only calculates the risk of gestational diabetes.

The function calculating the risk score for gestational diabetes runs through all the relevant questions from the dictionary containing the user's answers and adds points based on the answer. These points form a risk score which is used to estimate the severity of the complication. The risk score and severity will then be returned to the calculation object and further sent to the frontend displayed as a result for the user.



```
services
  __pycache__
  complications
    diabetes.py
    preeclampsia.py
  calculation.py
  factors.py
  score_calculator.py
  translation.py
```

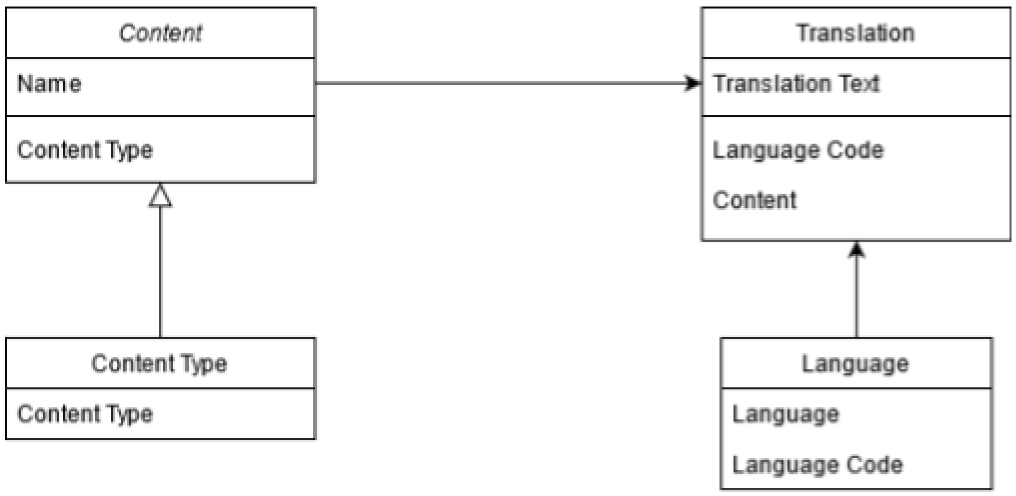
The calculation.py file gathers calculation results from each of the complication files.

From this point on, we implemented calculations for multiple complications by adding additional files like the one covering gestational diabetes. The calculation function in each of these files is written based on the calculation “recipes” given by the client. Since this has to be done manually there is currently no easy way of adding new complications without writing a new python function.

When a client connects to our Django web server, it connects to the domain associated with the server followed by a path corresponding to a method on the server. As the project grew, we realized that we would need a lot more endpoints than just the aforementioned “calculate” endpoint.

The “calculate” endpoint has minimal interaction with the server’s database and calculates the risk scores based on hard-coded criteria. Most of the other endpoints, however, require that something is queried for and fetched from the database. Django allows us to make database models in Python, which are automatically converted to SQL tables and objects. This makes the conversion from SQL objects to Python objects (and vice versa) more or less seamless for the developer, and greatly reduces complexity and mental clutter.

Our database is designed in such a way that most objects are part of a table called “content”.Content is an object type that is meant to be easily translated. Different content objects are associated with different content types. A factor, for example, is a content object associated with the factor content type. This allows us to easily associate several translations to content. Querying for content objects associated with a specific language is by extension made very easy.



This is the diagram representation of how translations are modeled in the database. If you want to query for all English factors, for example, you simply query for “Translation objects” that have “en” as associated language code, and “factor” as the content’s content type.

When a client wants to fetch all of the factors associated with a language, it needs to connect to the “factors/<lang_code>” endpoint. The “lang_code” string is then passed as an argument to a function that queries for all factors with that specific language code.

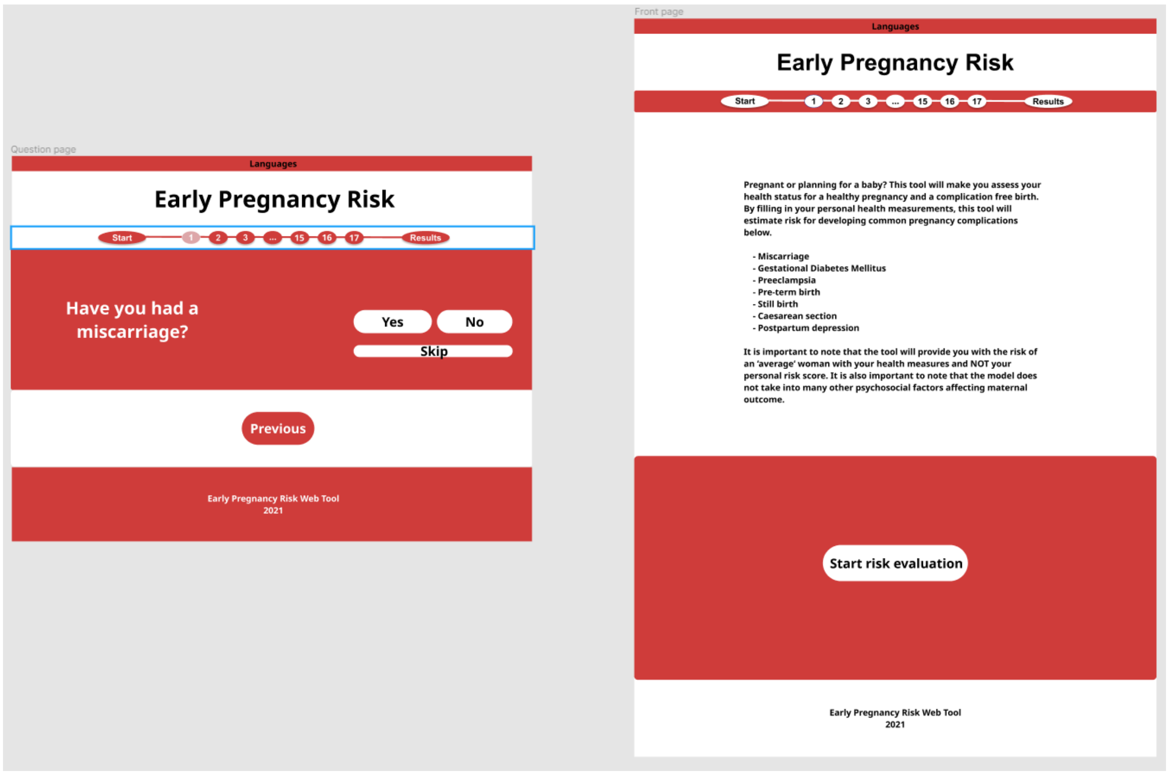
In order to properly handle various exceptions and errors that might happen in the backend, we created our own error handling decorator. In Python a decorator is a function that is “wrapped around” the function it is decorating. Our decorator, called “exception_handler_request”, wraps the function it’s decorating in a try/except block. This means that exceptions thrown anywhere below it automatically get caught very high up. Our code therefore adheres to the “throw low, catch high” principle. The decorator then constructs a response based on the contents of the exception. Exceptions that are not explicitly thrown by us are handled in a different way than exceptions that are, as they might expose sensitive information about the server.

Exceptions should always be logged, as they are an indication of something bad happening in the code. We therefore decided to implement logging through Python’s own logging library. Logging is also done in the decorator, since all exceptions are eventually caught and handled there. Exceptions that we explicitly threw are logged with a “warning” severity, since they are indicating that something we knew could go wrong went wrong (i.e. received incomplete/bad data from a client). These types of exceptions should be looked at, but are most likely not very severe. All other exceptions, however, are logged with “error” severity, which is the greatest severity. These types of exceptions should be looked at immediately, since they are an indication of something unexpected going wrong in the code. IP addresses of clients are not logged by default, but all requests (including the IP addresses) that lead to exceptions are logged and stored on a separate file.

Frontend

On the frontend project we decided to use the framework React, the community project React Native, and the Expo CLI. These tools/frameworks work together to give us a multi platform codebase (also known as cross platform), meaning that we only have to write the code once, and then we can compile the source code into some select platforms, such as Android, IOS, and Web. The intriguing part about using React and React Native, is that we do not have to worry about writing native code, we can - to a certain extent - only think about the web project. There are some instances where we need to use OS specific styling, but most of the time we do not have to worry about any platforms except for the web. The biggest downside of this way of working is that if we want to have a multi platform system, it drastically limits the amount of prebuilt libraries and components we can use. Any library we decide to use must have been developed with multiplatform support in mind, and some of the bigger component libraries, such as React Native Elements, add too much overhead for a project like this.

With this in mind, we set out to create the first version of the website, which meant to come up with a prototype of the website. Here we used Figma to create and collaborate on a website design that we were happy with. We wanted to keep the front page of the site as simple as possible and as easy to use as possible.



Pregnant or planning for a baby? This tool will make you assess your health status for a healthy pregnancy and a complication free birth. By filling in your personal health measurements, this tool will estimate risk for developing common pregnancy complications below.

- Miscarriage
- Gestational Diabetes Mellitus
- Preeclampsia
- Pre-term birth
- Still birth
- Caesarean section
- Postpartum depression

It is important to note that the tool will provide you with the risk of an ‘average‘ woman with your health measures and NOT your personal risk score. It is also important to note that the model does not take into many other psychosocial factors affecting maternal outcome.

This meant to have limited options and keep the user experience as streamlined as possible. The client wanted to have a progress bar of some kind, so we had to find a way to implement this on the site as simply as possible and not make the site cluttered.
We also needed a prototype for the question page, it needed to include the progress bar, a skip button, the answer options and the question. We included a previous button in the prototype that was included in the final product.

After we had a good idea of how we wanted the site to look, we presented it to the client and got an OK. This leads us into the development stage, where we ended up changing the look of the website some, this was because we found some limitations with React Native and the fact that we had to plan for both a website and for apps.

Lazy loading and code splitting

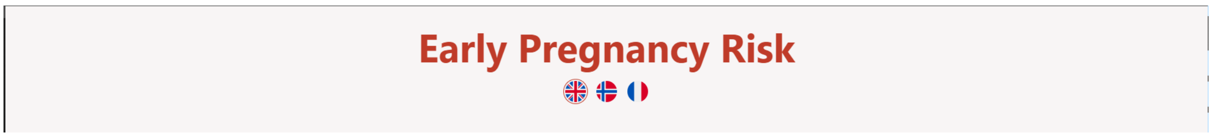
An important factor for any web project is the loading time of your site, especially if you want to reach a lot of people. One way to minimize the perceived loading time of your page, is to load certain parts of it at a later date, or just before you expect the user to need it. This is called lazy loading. In order to achieve this you have to split up your page in smaller chunks. In this project we use Loadable Components, it allows us to specify when and what we want to lazy load. In the code snippet below, we have defined the Front Page, the Result Page, and the Form Page as “loadable”, which means they are ready to be lazy loaded.

```
import loadable from "@loadable/component";
const Form = loadable(() => import("./components/Form"));
const Results = loadable(() => import("./components/Results"));
const FrontPage = loadable(() =>
import("./components/FrontPage"));
```

Since we are using a framework instead of plain html, css, and javascript, there will be a lot of extra, arguably unnecessary, information transmitted to the user. Because of this, users with a slow internet connection will see a white screen until the main chunk of data has loaded and unzipped, limiting the waiting time until the user sees something on the page is a good motivation for lazy loading, which is exactly why we do it. In our case we remove as much information as possible from the main chunk, and load it in after the fact. Because of this, we display a loading indicator to the user, while the other chunks are transmitted, as well as any REST api requests we are sending.

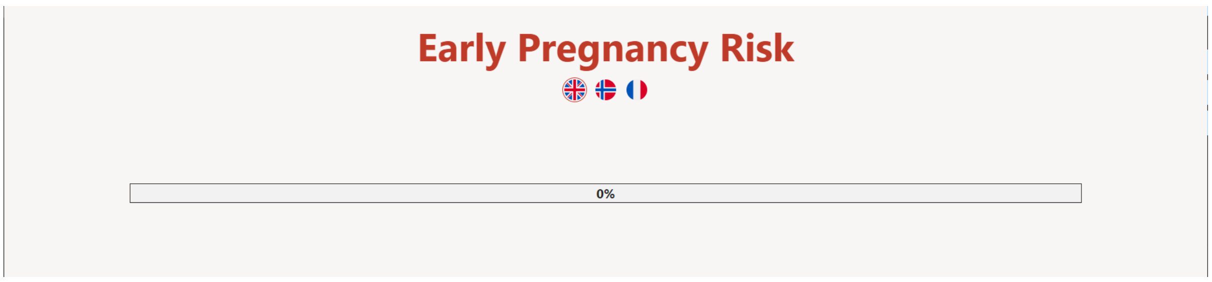
```
  // Preload components
  useEffect(() => {
    fadeTo(1, 1000, fadeGlobal);

    (async () => {
      FrontPage.preload();
      Form.preload();
      Results.preload();
    })();
  }, []);
```

In React there is something called a useEffect “hook”, this hook will not execute until the main chunk has been downloaded. This means that we can use a useEffect hook as the start of our loading procedure. In our case, it is implemented like shown in the code snippet below.


Early Pregnancy Risk

We started with making the front-page of the site to get the look of the page right. We wanted a header that contains the “Logo” of the page, translation options and a progress bar. We ended up putting the progres bar on the form page to have the front page uncluttered.

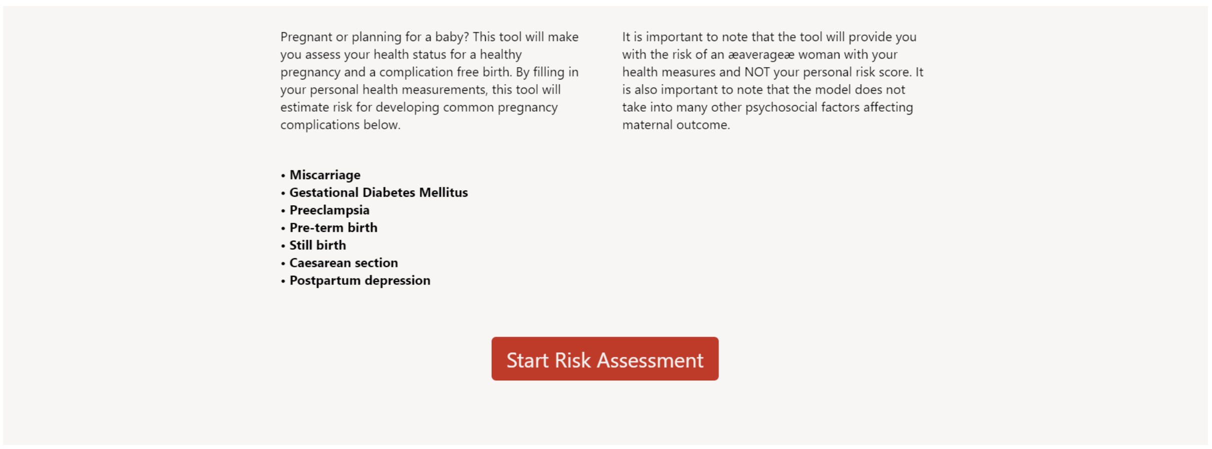
The translation bar is it’s own component, the country flags are hardcoded in. but easy enough to add more. The flags are `TouchableOpacity` and they contain the


Early Pregnancy Risk

0%

language code that the site sends to the server to get the translations for the site. The progress bar is an `Animated.View` That gets filled dynamically by how many factors that you have answered.

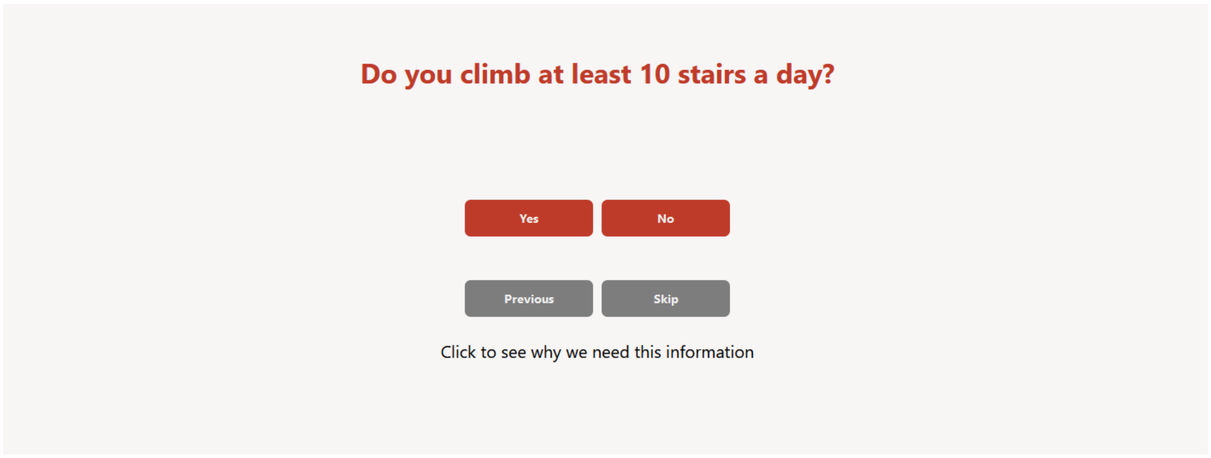
The rest of the page is just a single component that holds the overview of what the page does and how it can be used. This is dynamically changed in the translation since all of the text gets retrieved from the back end.


Pregnant or planning for a baby? This tool will make you assess your health status for a healthy pregnancy and a complication free birth. By filling in your personal health measurements, this tool will estimate risk for developing common pregnancy complications below.

It is important to note that the tool will provide you with the risk of an æaverageæ woman with your health measures and NOT your personal risk score. It is also important to note that the model does not take into many other psychosocial factors affecting maternal outcome.

- **Miscarriage**
- **Gestational Diabetes Mellitus**
- **Preeclampsia**
- **Pre-term birth**
- **Still birth**
- **Caesarean section**
- **Postpartum depression**

This is so that we can have the whole of the page translated to the specific languages.


Do you climb at least 10 stairs a day?
Yes
No
Previous
Skip
Click to see why we need this information

When the user wants to start the risk assessment we load in the form component and deliver the factors from the App.JS to the component. The factors are in a JSON format that contains the input type, the question, if it is skippable and the factor name. Since we have multiple different input types we need to distinguish the different types, and then load the required input type. We do this by checking the input type and then returning the specific input.

```
function renderInput(type) {
    switch (type) {
      case "integer":
        return (
          <IntInput
            value={factorInteger}
            setValue={(v) => {
              setFactorInteger(v);
            }}
            completed={(b) => setIsSubmitting(b)}
            maxDigits={factors[nr].maxdigits}
            unit={factors[nr].unit}
          />
        );
        case "multiple":
          return (
            <MultipleInput
              value={factorMultiple}
              setValue={(v) => {
                setFactorMultiple(v);
              }}
              completed={(b) => setIsSubmitting(b)}
              maxDigits={factors[nr].maxdigits}
              unit={factors[nr].unit}
            />
          );

      case "boolean":
        return (
          <BooleanInput
            setValue={(v) => setFactorBoolean(v)}
            completed={() => setIsSubmitting(true)}
          />
        );

      case default:
        break;
    }
  }
```

Since the client also wanted some questions to be skippable we needed to implement a skip button as well, but only for the questions that are skippable.

```
{factors[nr].skippable ? (
            <SkipInput
              setSkipped={() => {
                setSkipped(true);
                setTotalSkipped((v) => v + 1);
              }}
              completed={() => setIsSubmitting(true)}
            />
          ) : null}
```

We did this by a true/false gate, since it will do the same thing every time.

To load the whole question with answers we use a mic of true/false gates and basic list manipulation.

```
return (
    <View style={styles(width).container}>
      <View style={styles(width).progressBarContainer}>
        <Progressbar progress={nr} total={factors.length} />
      </View>
      <View style={styles(width).questionContainer}>
        <Text
style={styles(width).question}>{factors[nr].question}</Text>
      </View>
      <View style={styles(width).buttonContainer}>
        {renderInput(factors[nr].answertype)}
      </View>
      <View style={styles(width).staticButtonContainer}>
        {nr > 0 ? (
          <BackInput
            setInput={() => {
              setInputController(["back", null]);
              setIsSubmitting(true);
            }}
          />
        ) : null}
        {factors[nr].skippable ? (
          <SkipInput
            setInput={() => {
              setInputController(["skip", null]);
              setIsSubmitting(true);
              setTotalSkipped((s) => s + 1);
            }}
          />
        ) : null}
```

We found that this is the easiest especially when it comes to handling answers and handling the back button. since we then have the index of the item and then do not need to find the item in the list and can just use the index.

The client also wanted a way for each question to have a source listed and the relevant risks to be displayed. We decided that each question would get an overlay that contains these. Here we used the React nativ Modal component, this is because we then did not have to worry about the z-index of the components. The overlay component receives the factor, lang code and a boolean statement if it is visible or not. After opening the overlay we sent the factor name and the lang_code to the refrencelist. And it is here that the references get retrieved from the back end.

```
(async function () {
 const response = await getReferences(factor_name, lang_code);
 if (response == null) {
   close();
 } else {
   setReferences(response);
   setIsLoading(false);
 }
})();
```

This leads to a more responsive site, that does not always load something from the backend. Or have to load a lot from the back-end in the beginning.
We want to display it in a scrollview This is because some factors have a lot of sources and some have just a few. As well as it makes it easier to display on mobile devices.

```
centeredView: {
 flex: 1,
 justifyContent: "center",
 alignItems: "center",
 marginTop: Platform.OS == "ios" ? 20 : 0,
},
modalView: {
 justifyContent: "center",
 margin: 35,
 backgroundColor: "white",
 borderRadius: 20,
 paddingHorizontal: isTablet ? "2%" : "3%",
 paddingVertical: isTablet ? 20 : 0,
 ...Platform.select({
   web: {
     maxWidth: Platform.OS == "web" ? "60%" : "20%",
     maxHeight: Platform.OS == "web" ? "60%" : "20%",
   },
   default: {
     paddingVertical: 100,
   },
 }),
```

We fix the view of the devices in the CSS, here we see that we check what kind of device we are on and then change the CSS accordingly

This leads into how the page collects the answers and stores them before sending them to the backend.

The answer is set as a state, this is to make it easier to manipulate. We first want to copy the current data state and create a new one. This is just to keep the answers separate if something happens. After that we have to check if the question is skipped. and if so we want to remove that factor from the data list. This is so that the back end does not take that factor into its calculation. By skipping a question the calculation is not as accurate as it could be, but the user gets a message about this on the result page.

After we have checked if it is skipped we have to check what kind of input it contains. This is just a precaution, since the integer input can be received as a float and we want a Number. After adding the answers to the copy, we set the copy as the current answers data state.

After all of this is done we also have to check if we have to add subfactors. This is because some of the questions are just relevant if you have been pregnant before. And if you have not been pregnant before these are questions that are unnecessary.

```
  useEffect(() => {
    if (!isSubmitting) return;

    if (inputController[0] == "category") {
      addSubFactors();
    } else if (inputController[0] == "back") {
      setNr((n) => (n - 1 > 0 ? n - 1 : 0));
      setInputController([null, null]);
      setIsSubmitting(false);
      return;
    } else if (inputController[0] == "skip") {
      let tData = data;
      delete tData[factors[nr].factor];
      setData(tData);
    } else {
      let tData = data;
      tData[factors[nr].factor] = inputController[1];
      setTotalSkipped((s) => {
        const index = s.indexOf(factors[nr].factor);
        if (index > -1) {
          s.splice(index, 1);
        }
        return s;
      });
      setData(tData);
      addSubFactors();
    }
    setInputController([null, null]);
    setIsSubmitting(false);
    setNr(nr + 1);
  }, [isSubmitting]);

  useEffect(() => {
    if (factors != null && nr >= factors.length) {
      changePage(data, totalSkipped);
    }
  }, [nr]);

  if (factors == null) {
    return (
      <View style={styles(width).loading}>
        <Loading />
      </View>
    );
  } else if (nr >= factors.length) return null;
```

The way we add subfactors is first we check if it contains subfactors, and if it does we have to check if the requirements are met to add subfactors. If they are, we split the factors list at the point we currently are, this is that the relevant questions come after the required answer. After splitting we just add the subfactors to the factors list and then concat the two lists together again.

```
function addSubFactors() {
    if (factors[nr].subfactors != null) {
      let shouldAdd = false;
      if (factors[nr].answertype === "integer") {
        shouldAdd = checkRequirement(
          factors[nr].requirement,
          inputController[1],
          "integer"
        );
      } else if (factors[nr].answertype === "multiple") {
        shouldAdd = checkRequirement(
          factors[nr].requirement,
          inputController[1],
          "integer"
        );
      } else if (factors[nr].answertype === "boolean") {
        shouldAdd = checkRequirement(
          factors[nr].requirement,
          inputController[1],
          "boolean"
        );
      } else if (factors[nr].answertype === "category") {
        shouldAdd = true;
      }

      if (shouldAdd) {
        //remove null values first
        let subfactors = factors[nr].subfactors.filter(
          (el) => el != null && !factors.includes(el)
        );
        if (factors[nr].answertype !== "category") {
          setTotalFactors((v) => v + subfactors.length);
        }
        let left = factors.slice(0, nr + 1);
        let right = factors.slice(nr + 1);
        left.concat(subfactors);
        left.concat(right);
        let temp = left.concat(subfactors, right);
        setFactors(temp);
      }
    }
  }
```

After all of the factors have been answered and collected we send the answers to the backend in a JSON format. JSON is the easiest way for the backend to read and manipulate the data.

We then get back a JSON file that contains:

```
payload: [
 {
   complication: "TEST Gestational Diabetes Mellitus",
   severity: 0,
   severity_str: "Very Low",
   risk_str:
     "5% of pregnancies with similar health status are affected by
this complication",
   risk_score: 53,
 },
```

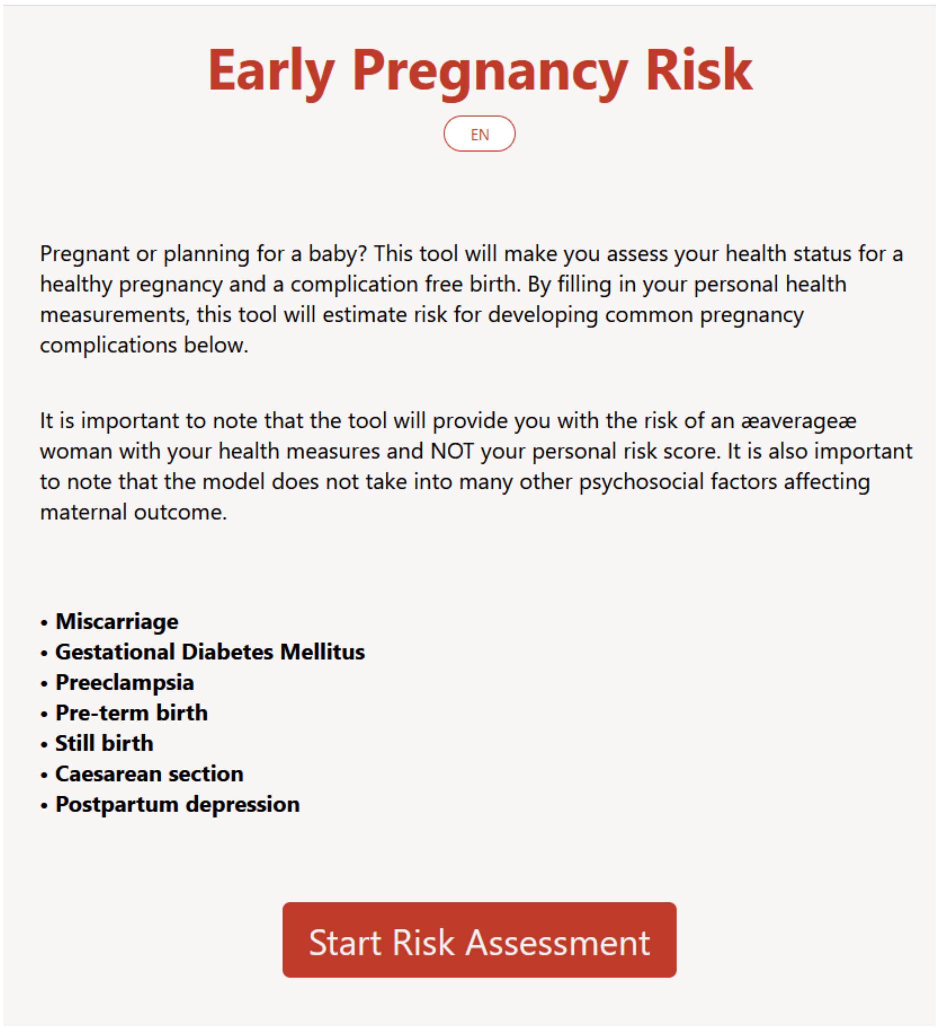
Here we have the name of the risk, the severity and the risk score. We read this response and we then again produse a list over the risks that have been calculated and then we produce an animated bar that shows the severity of the risk score.

In the end we have an overview of all of the risk calculations that have been done, and display the results for the user. (Go to page 34 for picture of the result display)

Results

Webtool

In our webtool the user has the opportunity to get multiple risk calculations, all at once. At the same time they will get a statistic of how many other women that have the same risk score have had the complication.
As some of our questions are dependent on the outcome of the previous question, we needed to make a lot of the site dynamic. If someone who is completing the questionnaire says they have never been pregnant before, for example, we do not want to ask more questions related to earlier pregnancies.


Early Pregnancy Risk

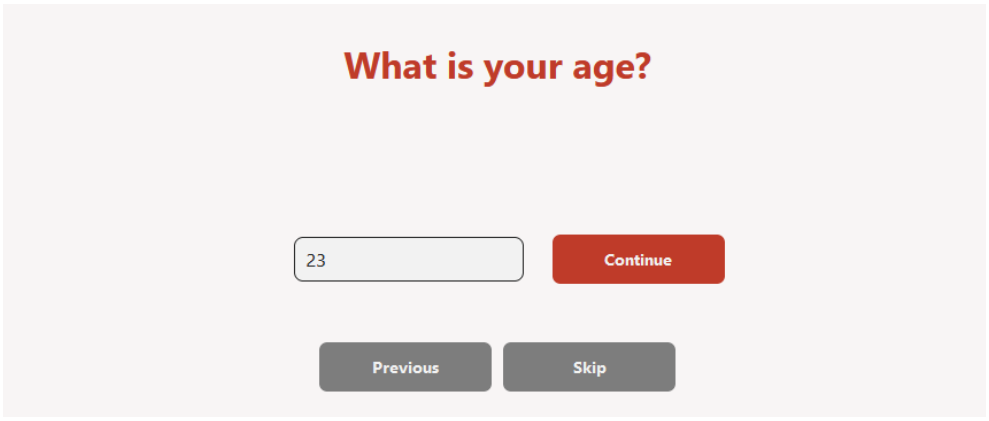
EN

Pregnant or planning for a baby? This tool will make you assess your health status for a healthy pregnancy and a complication free birth. By filling in your personal health measurements, this tool will estimate risk for developing common pregnancy complications below.

It is important to note that the tool will provide you with the risk of an æaverageæ woman with your health measures and NOT your personal risk score. It is also important to note that the model does not take into many other psychosocial factors affecting maternal outcome.

- **Miscarriage**
- **Gestational Diabetes Mellitus**
- **Preeclampsia**
- **Pre-term birth**
- **Still birth**
- **Caesarean section**
- **Postpartum depression**

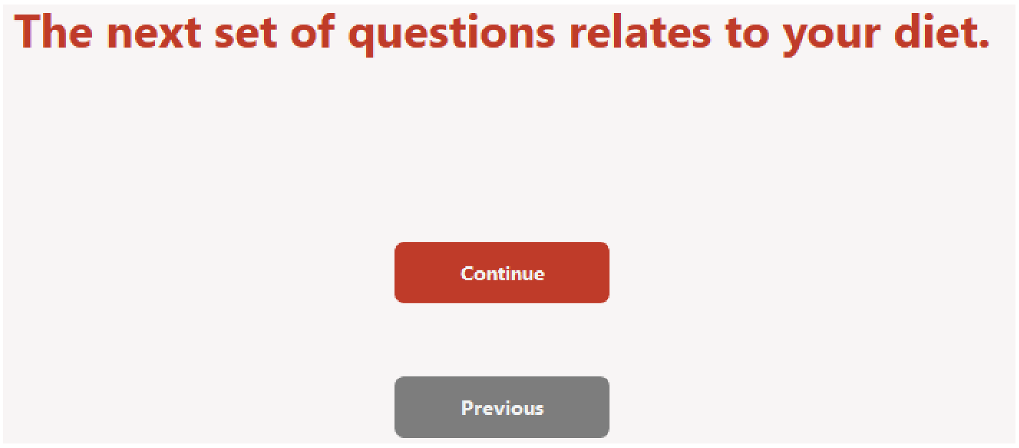
Start Risk Assessment


What is your age?

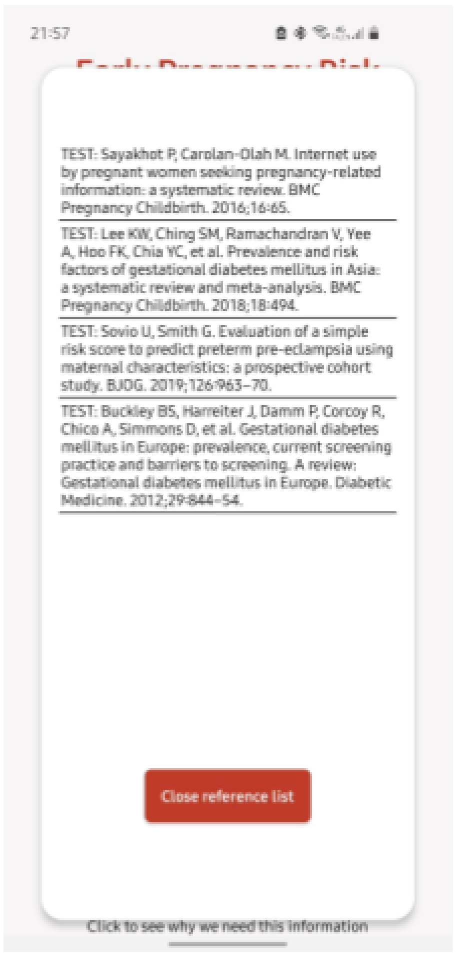
23

Continue

Previous Skip



The next set of questions relates to your diet.



TEST: Sayakhot P, Carolan-Olah M. Internet use by pregnant women seeking pregnancy-related information: a systematic review. BMC Pregnancy Childbirth. 2016;16:65.

TEST: Lee KW, Ching SM, Ramachandran V, Yee A, Hoo FK, Chia YC, et al. Prevalence and risk factors of gestational diabetes mellitus in Asia: a systematic review and meta-analysis. BMC Pregnancy Childbirth. 2018;18:494.

TEST: Sovio U, Smith G. Evaluation of a simple risk score to predict preterm pre-eclampsia using maternal characteristics: a prospective cohort study. BJOG. 2019;126:963–70.

TEST: Buckley BS, Harreiter J, Damm P, Corcoy R, Chico A, Simmons D, et al. Gestational diabetes mellitus in Europe: prevalence, current screening practice and barriers to screening. A review: Gestational diabetes mellitus in Europe. Diabetic Medicine. 2012;29:844–54.

Android application displaying the popup for details about why this question is relevant

Gestational Diabetes Mellitus

Very high risk

92.5 % of pregnancies with similar health status are affected by this complication.

Preeclampsia

High risk

60 % of pregnancies with similar health status are affected by this complication.

Pre-term Birth

Low risk

7.0 % of pregnancies with similar health status are affected by this complication.

Miscarriage

Increased risk

48.0 % of pregnancies with similar health status are affected by this complication.

Still Birth

Increased risk

35.0 % of pregnancies with similar health status are affected by this complication.

Postpartum Depression

Increased risk

40 % of pregnancies with similar health status are affected by this complication.

Cesarean Section

Low risk

0 % of pregnancies with similar health status are affected by this complication.

Placental Abruption

Low risk

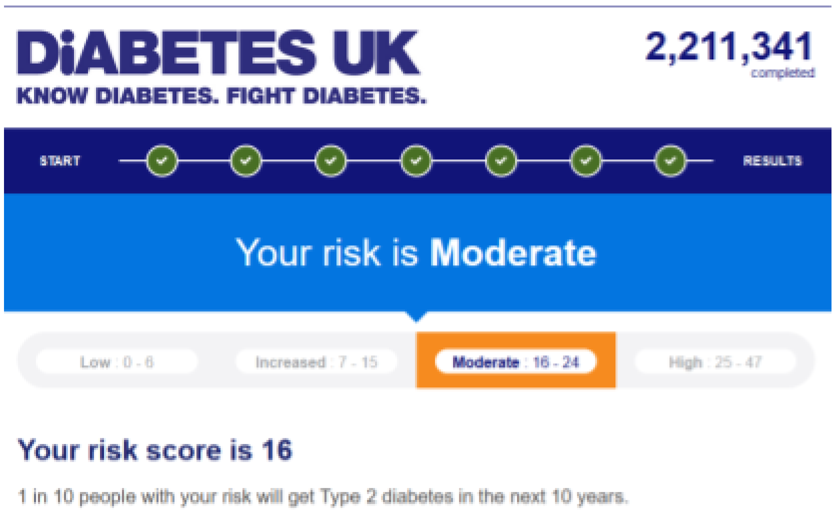
4 % of pregnancies with similar health status are affected by this complication.

Placenta Praevia

Low risk

2.8 % of pregnancies with similar health status are affected by this

Our client had already seen several tools that did calculate scores for some of the complications. Some were really good, like Risk Score Diabetes UK. The web site is clean, has clear and concise questions, and most people have no problem taking the test. (University of Leicester, n.d.)



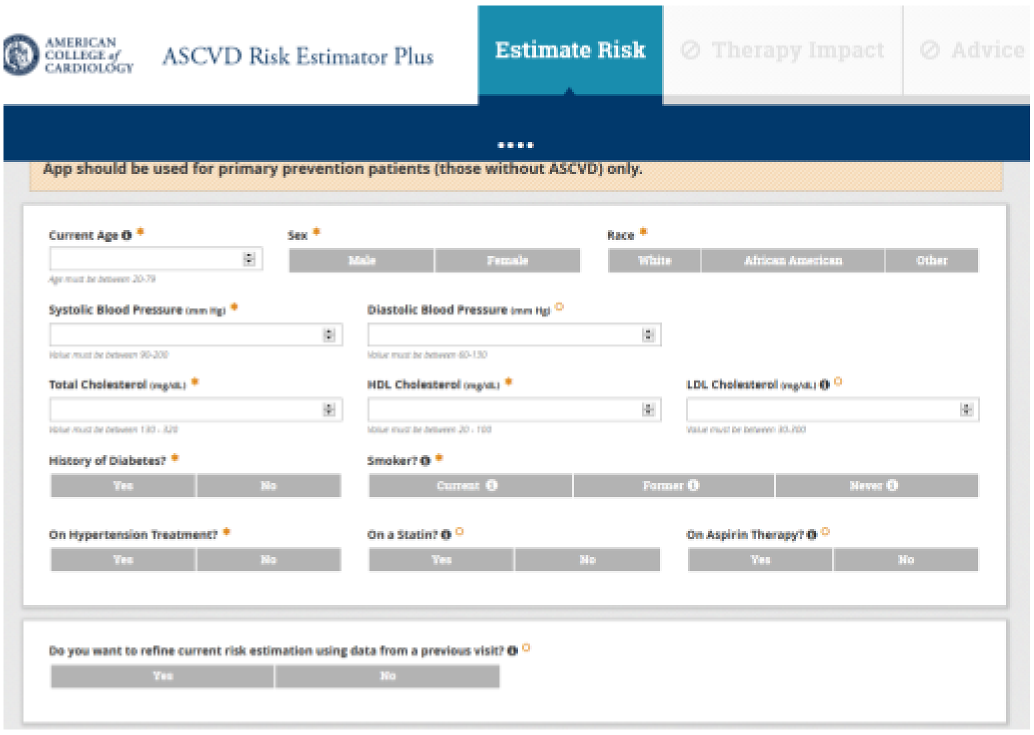
Your risk is **Moderate**

Your risk score is 16

1 in 10 people with your risk will get Type 2 diabetes in the next 10 years.

Screenshot of Diabetes UK test site

The problem is that it only calculates the risk factors for Type 2 diabetes. The same goes for ASCVD Risk Estimator Plus from American College of Cardiology, but this one looks rather dated compared to the former one. (American College of Cardiology, n.d.)



App should be used for primary prevention patients (those without ASCVD) only.

Current Age *
Age must be between 20-79

Sex * Male | Female

Race * White | African American | Other

Systolic Blood Pressure (mm Hg) *
Value must be between 90-200

Diastolic Blood Pressure (mm Hg)
Value must be between 60-130

Total Cholesterol (mg/dL) *
Value must be between 130 - 320

HDL Cholesterol (mg/dL) *
Value must be between 20 - 100

LDL Cholesterol (mg/dL)
Value must be between 30-300

History of Diabetes? * Yes | No

Smoker? * Current | Former | Never

On Hypertension Treatment? * Yes | No

On a Statin? Yes | No

On Aspirin Therapy? Yes | No

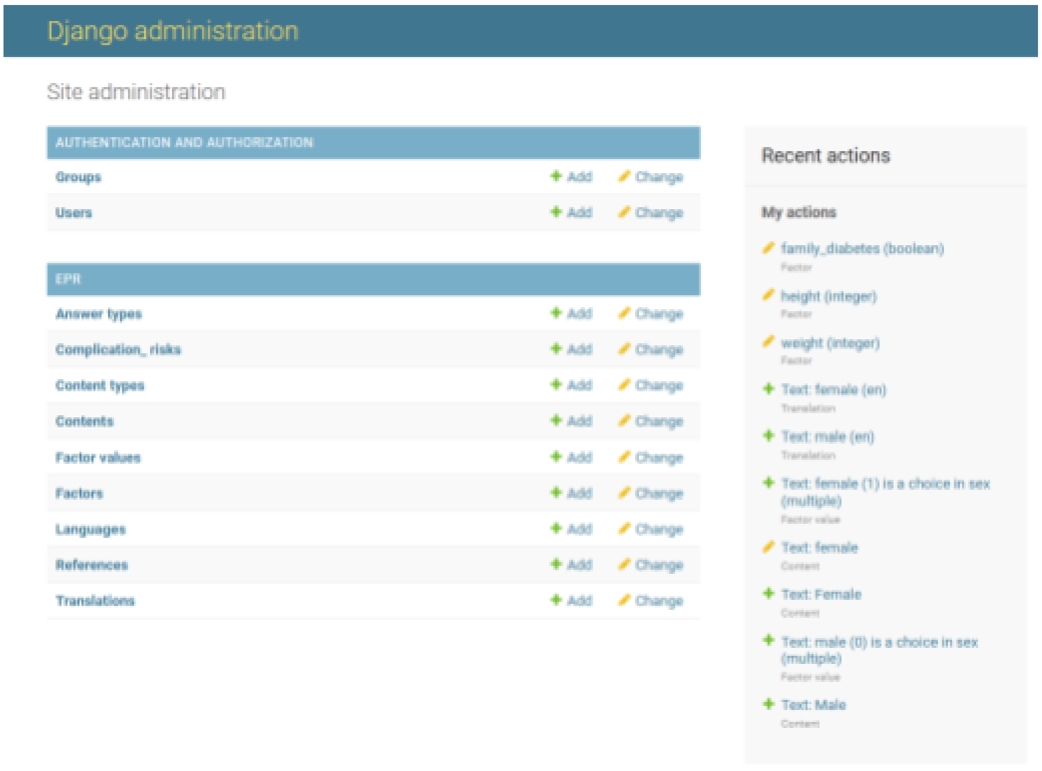
Do you want to refine current risk estimation using data from a previous visit? Yes | No

Screenshot of ASCVD Risk Estimator Plus

Our client wanted the tool to calculate risk factors for multiple complications, as opposed to the above mentioned, which only did calculations for one at a time.
We also had some more challenges, as we could not have as much static content as in these other examples.

Database

The database is implemented in a way that is easy to expand. New questions can easily be added by using the Django Admin panel. New translations, complications and factors can also be added or edited with the admin panel.



Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

EPR		
Answer types	+ Add	Change
Complication_risks	+ Add	Change
Content types	+ Add	Change
Contents	+ Add	Change
Factor values	+ Add	Change
Factors	+ Add	Change
Languages	+ Add	Change
References	+ Add	Change
Translations	+ Add	Change

Recent actions

My actions

- family_diabetes (boolean)
  Factor
- height (integer)
  Factor
- weight (integer)
  Factor
- + Text: female (en)
  Translation
- + Text: male (en)
  Translation
- + Text: female (1) is a choice in sex (multiple)
  Factor value
- Text: female
  Content
- + Text: Female
  Content
- + Text: male (0) is a choice in sex (multiple)
  Factor value
- + Text: Male
  Content

Illustration of the Django Admin panel where one can change and add questions, translations and more

Discussion

Backend Design Reflection

Many people would consider the Django web framework a very bloated one, where you get a lot of unnecessary tools that you never use. Despite this we landed on Django for several reasons.

The first reason is that it is more or less ready to use straight out of the box. Whereas Python web frameworks such as Flask come with no “bloat”, it requires you to manually install all necessary modules. Further, choosing Django gave us the ability to utilize their pre-built admin panel. Manually maintaining a database through a terminal can often be seen as a daunting task, even for the tech-savvy user. Our client, who is unfamiliar with exactly how this application is built, wanted to be able to add translations, edit questions, etc, even after we had left the project. By using Django’s admin panel we were able to easily allow our client to maintain the database in the future.

Django does not come without its limitations, though. Firstly it is a very monolithic framework. This means that the entire backend is more or less one giant application, and works as a single point of failure. Secondly, Django can often feel quite opinionated and inflexible. Such design decisions from the Django team were probably made so that the framework is “beginner friendly”. Features, such as having to specify all of your models in a single file felt very restrictive for us, and made it hard to keep track of the database models.

By choosing Django we also locked ourselves into the Python ecosystem. The Python community is currently thriving with a lot of available packages, but it is very slow compared to other languages. This may cause performance related issues further down the road. We are currently only querying the database and doing simple arithmetic operations on the data received from the frontend. If the project grows in the future, however, Python might become a bottleneck for computationally intensive tasks.

One of the biggest problems working on the calculation models is expandability. Every single complication introduced new factors and was impacted differently from shared ones. Therefore, each calculation had to be written individually so that the different risk factors had correct weights corresponding to each complication. This means that in order to add an additional complication at a later stage, one will have to write a new python-file with the computation. A solution to this problem would be to write each calculation model in a separate text-file with a specified format. Then we would have to implement a parser in python that could compute the risk based on the contents of each file. This could help people without python knowledge to add complications. Ultimately, we decided that the latter option was a little ambitious based on the time we had to work on the project. In addition, there is not a certainty that more complications will be added to the application and therefore such a solution could prove to be more complicated than what is necessary.

Frontend Design Reflection

While React Native allowed us to develop a web site and applications with the same code base, it did bring with it some limitations. Mostly it meant that we were limited to react native components, or build them from scratch ourselves. Some of the features that we wanted to include were simply not possible due to these limitations. One of them was to have a dynamic node network that showed how all the questions related to one another. This would have been a really cool feature, and probably been helpful to the ones taking the questionnaire. But React Native had no such component. We could of course build it ourselves, but with React Native this means that you have to build custom ones for all the different platforms that you want to build to. Not only would this affect the responsiveness of the web and mobile applications, but it would also take a lot of time to code.
One of the great things about React Native is that it is often updated, and new features are added all the time. Most likely there will come a native module that allows us to implement the node network that our client wanted, or the drop down functionality that we were looking at.
The thing is, it is also one of it’s big weaknesses. Because with rapid development and updates, also comes the risk of your old components ending up being

deprecated. In the end, you might end up having to redo a lot of the basic features. This is not strictly a React Native problem, but more of a general problem you get when you want to support many different platforms. If they make a change, chances are that you have to make changes to your codebase too.
For our project, that meant that we had to check the React Native documentation a lot, to confirm the method that we used would not be on the short list of deprecated features.

React native imposed some restrictions on what functions and features we could implement. It also has some syntax differences compared to vanilla JS which made for some frustrating moments. We discovered an alternative made by Google, Flutter, which seemingly works more like Java, both in form and function. The syntax, while similar to Javascript, is more like plain Java, and it also works more like it. Some examples of this are having a normal entrypoint in the form of a main function and having static typing. It also uses null safety notation in the newest versions, which would have been very useful in this project. Like React native, it also is a multi platform system, and supports Android, IOS, Windows Desktop, Web, and others. (Google LLC, n.d.)

Conclusion

In conclusion, have we managed to deliver what the client wanted? Yes, we have a website that can calculate risk for 11 of the pregnancy risks. We have a website that is easy to use and to understand.

The initial goal of the project was to have a website that can calculate pregnancy risks. And that we have, currently we have most but not all of the requested risk calculations. However from this point we can add more risk calculations.All we have to do is add them to the backend. So our goal of having a web tool that gives pregnant women and healthcare workers a toll to calculate the risk for pregnancy complications, has been achieved.

Another of our initial goals was to have a website that was easy to use and understand. We have received feedback from people that have taken the test, and they were overall positive about the design and features that were implemented. We always had to have in mind to keep the site minimalistic and uncluttered. This was not the easiest when we came to reference overlay or the result page. These are the two pages that needed to contain a lot of information but it needed to be structured in a way that is easy to read and understand. However, we still would like the questions to be easier to understand. Since some of them are hard to get. We would also like to have a short summary for each risk factor for the user to read. since right now we just have a lot of references.

But we have learned a lot about how to work effectively in teams, especially now that we can not work together. We learned how a team works together when we have such a small work load. We also have learned how to handle sudden changes from the client side. If the client suddenly has a wish to implement a certain function, we had to find a way to either implement it or say that we can implement that right now either out of time restrictions or the programming restrictions. Since we chose React-Natve, there are not a lot of libraries that are supported on both mobile and web.This leads to some restrictions of what we can achieve.

We also learned more about both frontend and backend programming and how they talk to each other.

But how does the future of this project look? This is not a finished project, it works and can be used, but there are some restrictions and limitations. The client wanted a Node net, that shows how the different risk factors are connected to the different complications. However this is not something that we managed to do. So this is something that future developers can work on. We would also have a small summary for each factor question to be displayed when a user asks for references. This is because right now the user only gets a lot of references and links to other reports. For a normal person this is a lot and does not tell them anything. Other than that there is not much to do. Of course if we had to redo the entire project we probably would focus more on the frontend and then use a different framework than React-Native. This is because we want to focus on the website and other ways of developing the site.

But what should the future developers on this page do? They should start looking at how questions are asked, since some of the questions are hard to understand. They should also look at how the translations work, since right now the translations are just google translate.
After that they should look into splitting the codebase, having the website in React and using Flutter for the app part. This is just to make the development easier and it is easier to implement more things. At the same time the developer does not have to think about both CSS for the website and the mobile app.

So in conclusion, we have delivered a website that calculates the risk for 11 pregnancy complications. It is easy to use and to understand, however it is not finished. There are still some things that can be implemented and some things that should be changed

Acknowledgments

Anagha Joshi - For giving us the opportunity to work on this project.
Noeska Smit - For giving good and well structured feedback and guidance.
Jarle Hjellvik Wallevik(Jarle the TA Guy) - For help and moral support in the hard times.

Bibliography

American College of Cardiology. (n.d.). *ASCVD Risk Estimator Plus*. Retrieved 05 19, 2021, from https://tools.acc.org/ASCVD-Risk-Estimator-Plus/#!/calculate/estimate/

Cafasso, J. (n.d.). *Complications During Pregnancy and Delivery.* Retrieved 04 24, 2021, from https://www.healthline.com/health/pregnancy/delivery-complications

Diabetes UK, University of Leicester, University Hospital of Leicester NHS Trust. (n.d.). *Diabetes UK*. Retrieved 05 19, 2021, from https://riskscore.diabetes.org.uk/start

Google LLC. (n.d.). *Flutter for React Native developers*. Retrieved 05 14, 2021, from https://flutter.dev/docs/get-started/flutter-for/react-native-devs

Stackshare. (n.d.). *Django - Reviews, Pros & Cons*. Django. Retrieved May 15, 2021, from https://stackshare.io/django

Appendix A

Communication between frontend and backend

We are using the RESTful API to communicate between the frontend and the backend.

The formatting of the response from the backend looks like this if successful:

```
// Default success response from the backend
{
    "success": true,
    "payload": {
       //Payload based on request
    }
}
```

And will look like this if it is not successful:

```
// Default error response from the backend
{
    "success": false,
    "error": {
        "Name": str,
        "Message": str,
        "Code": int
    }
}
```

API Endpoints is using the POST method for requests.
Below is an example of the request data for Gestational Diabetes Mellitus, but the rest of the complications are similar in function, albeit with different variables:

```
{
      "age": int,
      "parity": int,
      "gdm": bool,
      "congenital": bool,
      "stillbirth": bool,
      "miscarriage": bool,
      "preterm": bool,
      "macrosomia": bool,
      "height": int,
      "weight": int,
      "white": bool,
      "family_diabetes": bool,
      "polycystic": bool,
      "blood_pressure_family": bool,
      "blood_pressure_not_family": bool,
      "diet_not_varied": bool,
      "diet_sugar": bool,
      "diet_sweets": bool,
      "diet_processed_meat": bool,
      "diet_whole_grain": bool,
      "diet_diary": bool,
      "diet_vitamin_d": bool,
      "activity_walking_minute": bool,
      "activity_vigorous": bool,
      "activity_stairs": bool
}
```

The response from the backend to the frontend is based on the criteria from their respective complications.
You get an integer "severity", that ranges from 0-4, see below table for what each value represents.

0	Low
1	Increased
2	Moderate
3	High
4	Very High

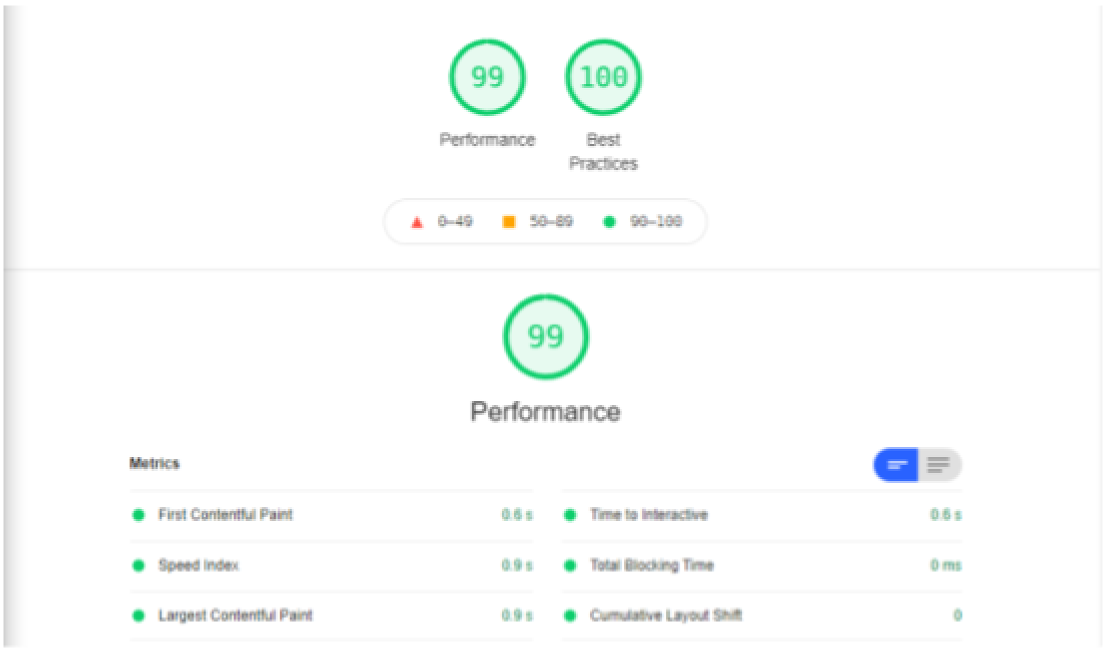
```
{
    "success": true,
    "payload": [
        {
            "complication": "Gestational Diabetes Mellitus",
            "severity": 4,
            "severity_str": "High",
            "risk_str": ">50 % of pregnancies",
            "risk_score": 53
        },
        {
            "complication": "Gestational Diabetes Mellitus",
            "severity": 4,
            "severity_str": "High",
            "risk_str": ">50 % of pregnancies",
            "risk_score": 53
        }
    ]
}
```

Factors for questions.

```
// Standard response from the backend. “?” means that it is
optional, and only used for the questions that need them
{
    "factors": [
      {
            "factor": string,
           "question": string,
           "answertype": string,
           "skippable": boolean,
           "maxdigits"?: int,
           "requirement"?: string,
           "subfactors"?: [
                    {},
                   {},
                   {},
               ]
        }
    ]
}
```

External performance and test analysis

Below you will see screenshots of external test results:



99 Performance

100 Best Practices

▲ 0–49 ■ 50–89 ● 90–100

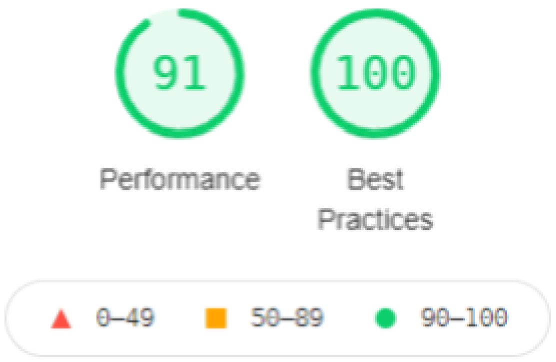
99

Performance

Metrics

●	First Contentful Paint	0.6 s	●	Time to Interactive	0.6 s
●	Speed Index	0.9 s	●	Total Blocking Time	0 ms
●	Largest Contentful Paint	0.9 s	●	Cumulative Layout Shift	0

Screenshot of desktop performance from Google Lighthouse

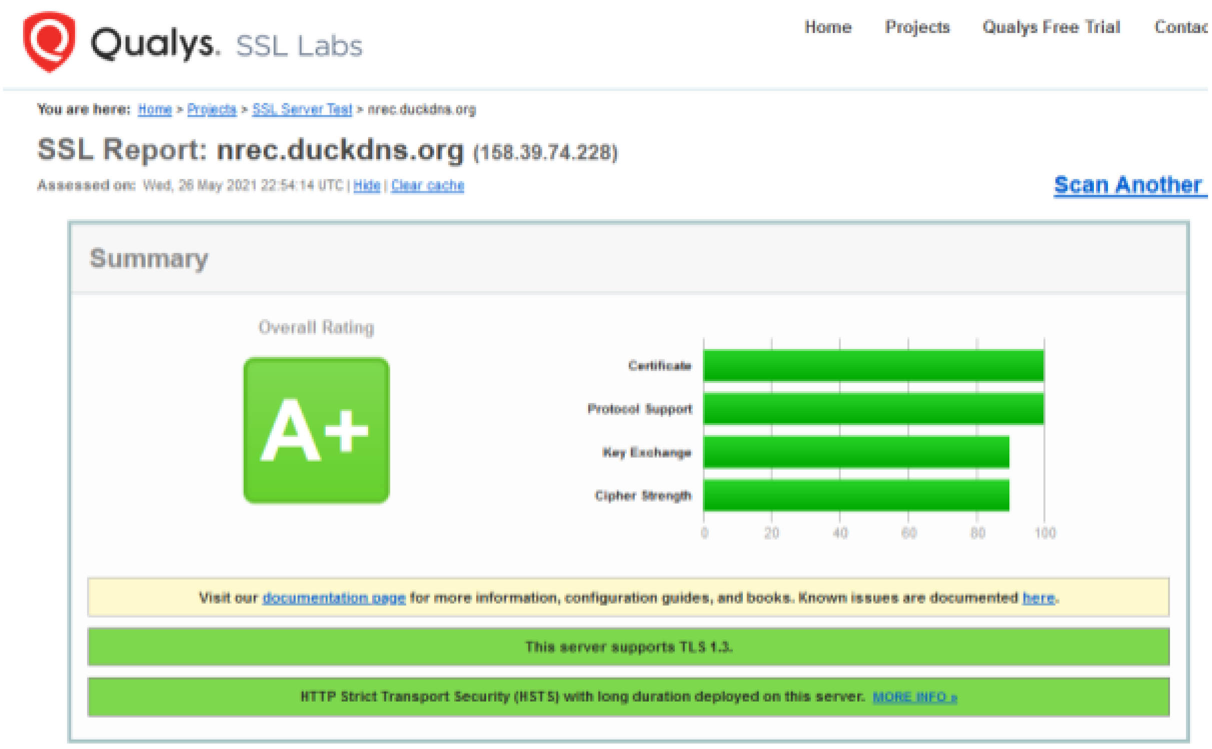


91 Performance

100 Best Practices

▲ 0–49 ■ 50–89 ● 90–100

Screenshot of mobile performance from Google Lighthouse



You are here: Home > Projects > SSL Server Test > nrec.duckdns.org

SSL Report: nrec.duckdns.org (158.39.74.228)

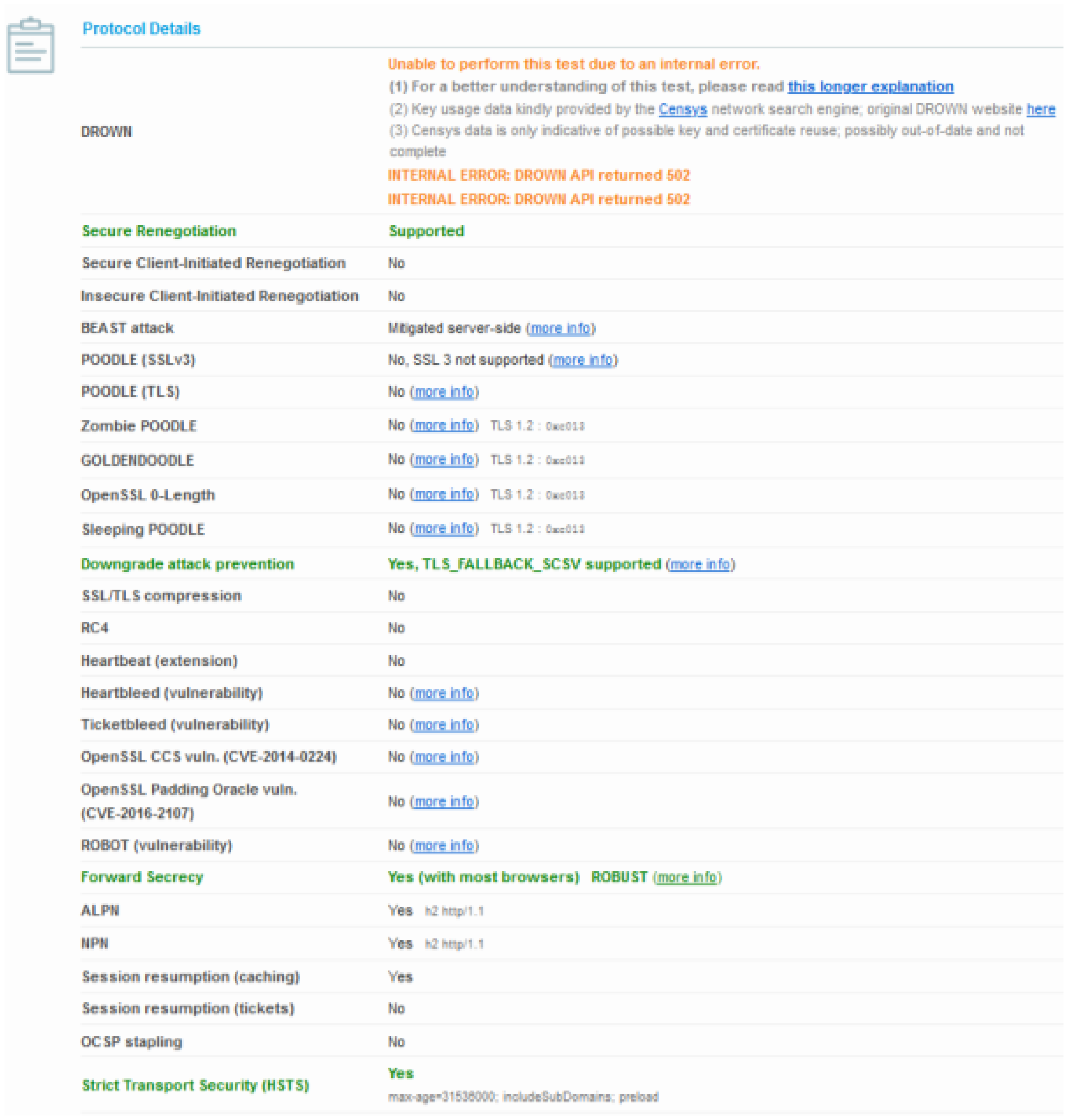
Assessed on: Wed, 26 May 2021 22:54:14 UTC | Hide | Clear cache

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server supports TLS 1.3.

HTTP Strict Transport Security (HSTS) with long duration deployed on this server. MORE INFO »

Test results from Qualys SSL Labs



Protocol Details

DROWN	Unable to perform this test due to an internal error. (1) For a better understanding of this test, please read this longer explanation (2) Key usage data kindly provided by the Censys network search engine; original DROWN website here (3) Censys data is only indicative of possible key and certificate reuse; possibly out-of-date and not complete INTERNAL ERROR: DROWN API returned 502 INTERNAL ERROR: DROWN API returned 502
Secure Renegotiation	Supported
Secure Client-Initiated Renegotiation	No
Insecure Client-Initiated Renegotiation	No
BEAST attack	Mitigated server-side (more info)
POODLE (SSLv3)	No, SSL 3 not supported (more info)
POODLE (TLS)	No (more info)
Zombie POODLE	No (more info) TLS 1.2 : 0xc013
GOLDENDOODLE	No (more info) TLS 1.2 : 0xc013
OpenSSL 0-Length	No (more info) TLS 1.2 : 0xc013
Sleeping POODLE	No (more info) TLS 1.2 : 0xc013
Downgrade attack prevention	Yes, TLS_FALLBACK_SCSV supported (more info)
SSL/TLS compression	No
RC4	No
Heartbeat (extension)	No
Heartbleed (vulnerability)	No (more info)
Ticketbleed (vulnerability)	No (more info)
OpenSSL CCS vuln. (CVE-2014-0224)	No (more info)
OpenSSL Padding Oracle vuln. (CVE-2016-2107)	No (more info)
ROBOT (vulnerability)	No (more info)
Forward Secrecy	Yes (with most browsers) ROBUST (more info)
ALPN	Yes h2 http/1.1
NPN	Yes h2 http/1.1
Session resumption (caching)	Yes
Session resumption (tickets)	No
OCSP stapling	No
Strict Transport Security (HSTS)	Yes max-age=31536000; includeSubDomains; preload

Qualys SSL Labs continued test results

As you can see, the work with continuously trying to optimize and do best practices has not been for nothing. We have achieved excellent test results from Google Lighthouse, with over 90% on performance, 100% best practice, and an A+ overall score on the network security. This shows that it is important to keep in mind the performance of the application when you are developing, and that it will pay off in the end. The reason why the mobile application is lower than the desktop version, is that it is not resizable. This is by design, and the overall score is still high. Although the security of the website is not crucial, this is still important for potential SEO (Search engine optimization), and it is important to get a good result here as well. It also contributes to the overall experience from the end user, and delivering as good an experience as possible is crucial for people to continue to use the site.

We are currently running it on NREC (Norwegian Research and Education Cloud), on anUbuntu instance. We have built the react native web version, but not mobile applications in expo. The site is then moved to NGINX subfolders (the site itself is now static, but the content is dynamic).
We are using Let’s Encrypt to get SSL certificates, and Certbot to auto renew them when needed. We are also enforcing HTTP/2, but still the initial request is HTTP 1.1. We are also supporting the ALPN extension, which should allow for initial request for HTTP/2
With cache control and expires headers, we have reduced the times necessary for the browser to send server requests, as it now can store and fetch content from the browser’s cache, or the source itself.
We also are utilizing the gzip on NGINX for compressing some of the content served. This brings down the page size from 492.6 KB to a respectable 172 KB.