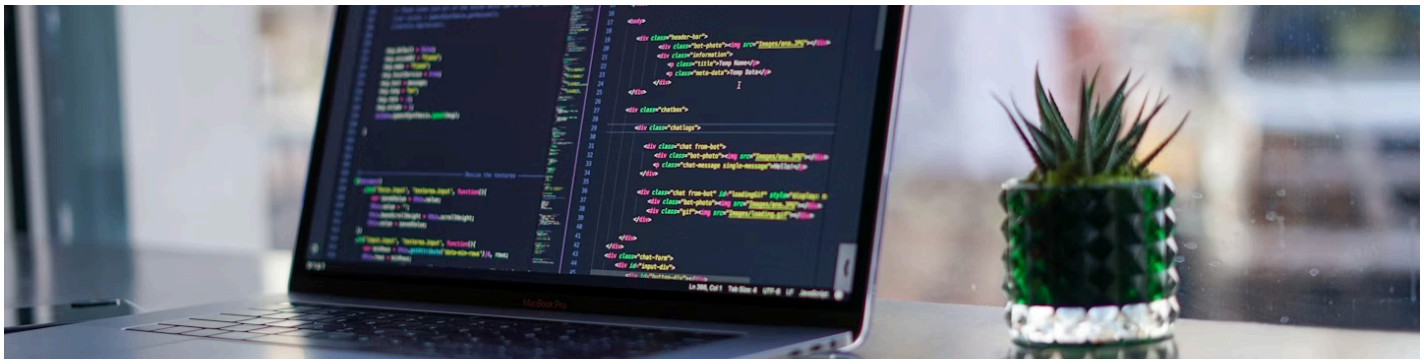

Development Project Group 1

This Coda Doc aims to help with organisation for the 3rd project of Semester 2 at Fontys ICT & media Design

Team Charter

This page serves to structure the Team Charter of the 3rd project of Semester 2 at Fontys ICT & Media Design

Team Members

Stefi:

- **Strengths:** Organization, documentation, design, front-end
- **Weaknesses:** Translating complex elements into code, back-end

Yana:

- **Strengths:** Front-end development, documentation, dependability, communication
- **Weaknesses:** Creativity, procrastination

Core Values

What do we prioritize?

- **Commitment to Quality**
  We strive to deliver clean, maintainable, and well-documented code that meets the project's objectives and user needs.
- **Open Communication**
  We maintain honest and respectful communication. Everyone is encouraged to voice ideas, ask questions, and raise concerns without fear of criticism.
- **Reliability & Accountability**
  Each member takes full responsibility for their tasks and respects deadlines. If challenges arise, we inform the team early and ask for support.
- **Collaboration Over Competition**
  We work together toward a common goal. We value teamwork over individual success and help each other improve through feedback and pair programming.
- **Learning-Oriented Mindset**
  We treat every challenge or mistake as an opportunity to grow and improve, both as individuals and as a team.

Group Norms

Our team is committed to working efficiently, transparently, and respectfully throughout the semester. These norms ensure smooth collaboration and uphold our shared values.

Tools & Workflow

- **Documentation:** All project documentation is maintained in **Coda**, ensuring both team members have 24/7 access for viewing, editing, and downloading.
- **Task Management:** We use **Trello** to assign, track, and prioritize tasks.
- **Version Control:** Code is merged and reviewed via **Git** (GitLab).
- **Communication:** All daily communication is conducted via **WhatsApp** and **Instagram**.

Response Time

- Team members must respond to all forms of communication within **12 hours** unless a valid reason is provided (e.g., illness or emergency).

Meeting Attendance

- Attendance is expected for all scheduled **status update meetings**. If one member is unavailable, the meeting may be rescheduled or followed by a written text message on Instagram or WhatsApp.

Deadlines

- Tasks are expected to be submitted on time.
  If a task cannot be completed:
  - Notify your teammate as early as possible.
  - Collaborate on solutions, which may include extending the deadline or task reassignment by mutual agreement.

Feedback Culture

- Feedback is welcomed and received with an **open mind**.
- We treat feedback as an opportunity for **personal and project improvement**, not as criticism.
- We commit to reflecting on suggestions and acting upon them constructively.

Roles

Stefania Melyova – UX Designer & Frontend Developer

Yana Spasova – Frontend Developer & Tester

Standards of quality

- Code is clean, readable, and reviewed before merging
- Design is responsive and matches the UX prototype

- All features are tested and bug-free before submission
- Website works on major browsers (Chrome, Firefox, Edge)
- Tasks are tracked in Trello and completed on time
- Communication is clear and progress is regularly shared
- User feedback is collected and applied where possible
- Documentation is kept up to date in Coda

Sample Agreement (Rules)

As a two-person team, we rely on clear communication, mutual respect, and shared responsibility. These rules define how we work together and maintain accountability throughout the project.

Working Hours

- Our agreed working hours are **Monday to Friday, 9:00 AM – 5:00 PM**.
- Availability outside these hours will be communicated in advance when possible.

Response Time

- We commit to replying to each other within **12 hours**, unless a valid reason (e.g., illness, emergency) is communicated.

Attendance

- Participation in all **classes and check-in meetings** is expected.
- If one team member is unavailable, they must notify the other and ensure their tasks are on track or adjusted accordingly.

Deadlines

- Tasks are expected to be submitted on time.
- If one of us cannot meet a deadline, we discuss it early and decide together whether to support or reassign the task.

Respect & Communication

- We commit to a respectful, inclusive, and professional working relationship.
- All communication is conducted in **English**, through WhatsApp or in person.

Feedback

- We give each other **constructive feedback**, focusing on:
  - Strengths and successes
  - Areas for improvement
  - Specific, respectful suggestions
- Feedback is always aimed at helping each other grow.

Development Responsibilities

- Each of us is responsible for our own **Git branch**.
- Changes to each other's code will only be made **after discussion and agreement**.

Design Decisions

- We agree on design changes **together**.
- If one of us wants to suggest a major visual change, we create a **separate frame or version** to propose it.

AI Usage

- AI tools can be used only if:
  - Both members **agree**
  - The output is **fully understood**
  - It aligns with course guidelines and academic integrity

⚠ Conflict Resolution

If either team member fails to follow the agreement repeatedly:

- The issue will be discussed openly and respectfully.
- If unresolved after **two incidents**, the matter will be escalated to the teacher for further guidance.

Project Plan

This page serves to structure the Team Charter of the 3rd project of Semester 2 at Fontys ICT & Media Design

1. Project Definition

- The project focuses on developing a new website for the **BELCO Alliance**, a nonprofit collaboration between five European universities.
- The website will serve as a **central information hub**, linking to **existing exchange programs** run by member universities.
- Unlike the BELCO Education page, this platform **does not promote BELCO-owned programs**, but rather facilitates visibility of already established university exchanges.
- The site will clearly showcase the **mission, goals, partnerships**, and **opportunities** of the alliance.
- A modern, accessible, and user-friendly design (from previous project) will improve communication and promote **international collaboration**.

1.1 Client:

- **BELCO Alliance** – A nonprofit partnership between universities in France, Germany, the Netherlands, Finland, and Denmark. BELCO supports international education through joint research, short academic modules, and mobility programs.
- Website: **belco-edu.com**
  Mission: Promote academic internationalization via university-driven exchange programs and collaborative academic offerings..

1.2 Team :

- **Position: Project Manager**

Contact person: Stefania Melyova

email: stefania.melyova@gmail.com

phone number: +359885738855

- **Position: Front-End Developer; Checkpoint Writer**

Contact person: Yana Spasova

email: yanadspasova@gmail.com

phone number: +31 6 25 1319 33

1.3 Current situation

- The current BELCO Alliance website lacks structure, visual appeal, and clarity.
- Key sections are either **non-functional** or **missing**.
- Information about university exchange programs is **not properly represented or linked**.
- The website fails to communicate the **purpose and reach** of the alliance.

1.4 Problem description

The Belco-edu.com website serves as a digital hub for fostering collaboration among members, partners, and stakeholders. However, the website is currently not fully functional and lacks essential content, limiting its ability to effectively showcase the alliance's mission and initiatives.

Key Challenges:

1. **Incomplete Functionality** – Some sections of the website are not fully operational, engagement and accessibility.
2. **Content Gaps** – Crucial information about the alliance, its projects, and opportunities for collaboration is missing or insufficiently detailed.
3. **User Experience Issues** – Without a seamless and informative experience, visitors may struggle to understand how they can contribute or benefit from the alliance.

1.5 Project Goal

- Create a **fully functional**, modern, and engaging website for BELCO Alliance.
- Provide **clear structure**, **intuitive navigation**, and **direct access to exchange programs** from university partners.
- Improve overall **usability**, **aesthetic**, and **alignment with BELCO's mission**.
- Increase awareness and international participation in alliance initiatives.

1.6 Deliverables

- A redesigned, responsive BELCO Alliance website (HTML/CSS/JS).
- Figma-based UI prototypes and mockups.
- Pages presenting BELCO's **mission, partners**, and **linked exchange programs**.
- Accessibility improvements (WCAG compliance).
- Final project documentation with usage guide and update instructions.

1.7 Non-deliverables

- No student enrollment systems or backend development.
- No third-party integrations (LMS, CRMs, etc.).
- No branding or logo redesign unless specifically requested.
- No ongoing site maintenance after delivery.

1.8 Constraints

- Limited original content and assets.
- Timeframe restricted to academic semester.
- Client availability may vary, affecting feedback cycles.
- Dependency on timely content delivery from client (e.g. exchange program links, partner info).

1.9 Risk Assessment

Risk	Mitigation

Delayed Content	Set early internal deadlines for receiving client-provided material.
Scope Creep	Stick to defined deliverables; require formal approval for extras.
Design Misalignment	Share mockups early and get regular client feedback.
Platform Limitations	Use simple, flexible technologies and document constraints clearly.

2. Development Phasing

Phasing Schedule

Week	Weekly Goals
Week 1	- Set up Next.js project; explore folder structure; configure routing - Research & implement shared layout: Navbar, Footer - Try implementing 1–2 key pages (Homepage, Programs) with dummy data - Explore authentication (NextAuth) + data model setup (MongoDB schema) - Review results; write decision summary with challenges & recommendations
Week 2A In case Research went well.	- Finish remaining page scaffolding (Partners, Blog, Articles, Account) - Integrate MongoDB (content for Blog/Partners pages) - Implement login/register flow + user session state - Admin dashboard (add/delete content) - Polish UI, finalize placeholder content, deploy on Vercel
Week 2B In case research was insufficient. Develop with strictly HTML|CSS|JS	- Create folder structure and navbars for both sites - Develop BELCO Alliance site (Homepage, Programs, Blog, Partners) - Develop BELCO Education site (simplified structure) - Refine layouts, apply responsive styling - Review, compress assets and folders, prepare for handoff
Week 3	- Conduct informal user testing with peers/stakeholders - Fix navigation/UX issues, improve accessibility - Optimize performance, clean code - Prepare final presentation/demo - Submit all deliverables + usage guide and update instructions

3. Target Audience

- **Students** – Exploring international exchange options through their home universities.
- **BELCO Members** – Coordinating and promoting academic mobility and module collaboration.
- **Future Partners** – Institutions interested in joining BELCO or aligning with its mission.

Checkpoints

Checkpoint 1

Presented to Paul:

- Project plan (Emphasis on Phasing)

Key Takeaways:

- The current structure of the phasing section, based on design principles (like Empathize and Define), is counterproductive since these were already completed in the previous group project.

- Any path we explore—such as testing and researching React for development—should be documented, including any challenges and failures.

- The phasing should be reorganized into a time-based format, such as a weekly or daily schedule, with clear goals and deliverables.

Checkpoint 2

Presented to Dirk:

- Updated Figma wireframes

- Questions about GitLab workflow (branching, etc.)

- Ideas for incorporating different programming languages into the project

Key Takeaways:

- The updated wireframes were well-received. The pages are now coherently linked, unlike the previous version.

- Hyperlink positions should be prioritized based on expected user behavior.

- The "Forgot password?" link should be grouped with the password input field to follow the Gestalt principle of association.

- Sending users an automatically generated password without giving them the option to change it introduces security risks. A safer reset method should be implemented. A user verification would not be possible for this project of our level yet so it will not implemented.

- Adopting a new programming language at this stage may lead to a lack of visual polish due to time constraints. HTML, CSS, and JavaScript remain more manageable for maintaining visual quality.

- Dirk recommended meeting with the client to clarify whether functionality or visual design is the higher priority. The MoSCoW list should be updated and deliverables clearly defined.

- Dirk visually demonstrated GitLab branching. We now understand how to use feature branches and merge into main once tested and stable. This helps avoiding loss of code

Checkpoint 3

Presented:

- Trello workspace

- Updated phasing in the project plan

Key Takeaways:

- No major changes needed to the overall timeline.

- Focus should now shift to maintaining and updating Trello with daily, actionable tasks.

- In a professional setting, the **project owner** typically creates and assigns tasks, while **developers select** which ones to take on.

- Each Trello task should be **labeled with its corresponding Git branch name** and **referenced in commit messages** to maintain clear traceability between code and task.

- Establish a routine for reviewing and updating Trello tasks daily or during stand-ups.

- Consider assigning a team member to oversee task clarity and consistency.

Presented to Paul

- **Trello Work Board**
- **Two BELCO Websites deployed by Vercel (Belco Education and Belco Alliance)**
- **Git Repository**

🔑 Key Takeaway

Although we worked on separate websites, we used this structure as an opportunity to **learn Next.js collaboratively**, helping each other debug and improve our code during work sessions at university. This showed that collaboration can still be meaningful even when working on parallel components, as long as **knowledge-sharing and support** remain central.

- Assign **relevant and specific titles** to each page (e.g., "BELCO Alliance Programs – International Exchange"). Helps improve **Google search visibility** and **user navigation**.
- Use **branches** effectively for collaborative development:
- Create separate branches for new features or individual contributions.

- Merge to `main` only after peer review and testing.

Research

Building the BELCO Alliance Website with Next.js and MongoDB

BELCO Alliance website sitemap (from the Figma design) outlining the main sections: homepage, programs, blog (with individual article pages), partners, user account, and an admin-only content management section. This roadmap will guide our implementation.

Introduction and Project Overview

In this guide, we will help you (two motivated university students) build the **BELCO Alliance** website as a full-stack application using **Next.js** (React framework) and **MongoDB** as the database. We'll follow the provided Figma sitemap to structure the site and implement all major features: a dynamic homepage, programs page, blog with articles, partners listing, user authentication (login/register), and an admin-only CMS for managing articles and partner universities. By the end, you will have a deployed website on Vercel that goes beyond your course requirements – an impressive achievement! 🎉

How We'll Proceed: We'll tackle the project in a sensible order to finish quickly and effectively. First, we'll set up the Next.js project and create the shared layout (navigation bar and footer) that appears on all pages. Next, we'll build each page (homepage, programs, blog, article detail, partners) according to the sitemap, initially using placeholder content. Then we'll set up the **MongoDB Atlas** database and **Mongoose** models to store users, articles, and universities, and replace the placeholder content with real data from the database. After that, we'll integrate **NextAuth** for authentication, enabling users to register and log in (with one designated admin user). Finally, we'll create the admin dashboard for content management (adding/deleting articles and universities) and ensure that only the admin can access it. We'll also cover protecting routes (so that, for example, only logged-in admins can reach the CMS) and how to deploy your finished site on **Vercel** with the necessary environment configuration.

Throughout the guide, we'll include code snippets, clear instructions, and visual references from the Figma design. The tone will be instructional and motivational – even if you're new to some of these technologies, don't worry! We'll break down each step so you can follow along, leveraging your solid HTML/CSS skills while learning the required JavaScript/React concepts. By following this step-by-step guide, you'll not only meet your course criteria but **exceed**them by delivering a fully functional, full-stack website. Now, let's get started!

Setting Up the Project Environment

Before building specific features, let's set up the development environment and project structure:

1. **Install Node.js and create a Next.js app:** Ensure you have Node.js and npm installed on your macOS system. In VSCode (or your terminal), use **Create Next App** to bootstrap a new Next.js project. For example:

```
npx create-next-app@latest belco-alliance
# or, if you prefer TypeScript: npx create-next-app@latest --typescript belco-alliance
```

1. Follow the prompts (you can accept defaults or choose TypeScript if comfortable). This will create a `belco-alliance` directory with a basic Next.js project inside.
2. **Open the project in VSCode:** Navigate into the project folder and open it in VSCode:

```
cd belco-alliance
code .
```

1. You should see a standard Next.js project structure with a `pages` folder, `public` folder, etc. Running `npm run dev` will start the development server on http://localhost:3000.
2. **Plan the page routes:** According to the sitemap, our site will have the following pages:
   - **Homepage** – the landing page (route: `/`).
   - **Programs** – information about BELCO programs (route: `/programs`).
   - **Blog** – a listing of news/research articles (route: `/blog`).
   - **Article detail** – individual article pages (route: `/blog/[id]` for each article).
   - **Partners** – list of partner institutions (route: `/partners`).
   - **Account** – user login and registration (routes: `/login`, `/register`; possibly a profile page).
   - **Admin Dashboard** – for admin user to manage content (route: `/admin` and subpages).
   - **About** and **Contact** – (if needed, these could be additional static pages or sections on the homepage; the nav has links for them).
3. In Next.js (using the Pages Router), each page is a React component file under the `pages/` directory. The router is file-system based: for example, a file `pages/about.js` becomes accessible at `/about`. We will create files for each page listed above. Dynamic routes (like the article page) are created by using square brackets in the filename. For instance, creating a file **`pages/blog/[id].js`** means any request to `/blog/<some-id>` will be handled by that file. We will use this for article pages.
4. To clarify, here's an overview of the file structure we will end up with for the main pages (you can create these files now as placeholders):

```
pages/
├── _app.js               // Custom App to include global layout
├── index.js              // Homepage
├── programs.js           // Programs page
├── blog/
│   ├── index.js          // Blog listing page
│   └── [id].js           // Article detail page (dynamic route)
├── partners.js           // Partners page
├── login.js              // Login page
├── register.js           // Registration page
└── admin/
    └── index.js          // Admin CMS dashboard (could include all forms or link to
others)
```

1. We'll fill these in as we go. Next.js will automatically route these pages based on file name. For example, `index.js` is the root route, and `blog/index.js` is `/blog`, etc.
2. **Install required dependencies:** We will need a few additional packages:
   - **Mongoose** (for MongoDB object modeling) and **MongoDB driver** (if not using mongoose's built-in) – to connect to MongoDB.
   - **NextAuth** – for authentication.
   - **bcrypt** (or `bcryptjs`) – to hash passwords for storing in the database and to verify them on login.
   - (Optionally) any UI/styling libraries you prefer, but since you have strong CSS skills, you can write custom CSS. Next.js supports global CSS and CSS modules. You might also consider installing **Sass** if you want to use SCSS for easier styling (e.g., `npm install sass` and then you can create `.scss` files).
3. Install the main packages by running:

```
npm install mongoose next-auth bcryptjs
```

1. This will add entries to your `package.json`. Now, restart your dev server if needed to ensure everything is in sync.
2. **Set up a Git repository (optional but recommended):** Initialize a git repo and make commits as you build features. This will help you track changes and deploy to Vercel later (Vercel integrates smoothly with GitHub/GitLab repos). For example:

```
git init
git add .
git commit -m "Initial Next.js setup"
```

1. You can push to GitHub when ready.

With the project created and dependencies installed, we can start building out the site's structure and features.

Creating a Shared Layout (Navbar and Footer)

The BELCO Alliance site has a common **navigation bar** and **footer** across all pages (as seen in the Figma design). We will create these as reusable components and include them on every page, so the site has a consistent frame.

Navbar: The top navigation bar will contain the BELCO logo and links to the main sections: Programs, Blog, Partners, About, Contact, and an Account icon or link (for login/profile). We'll implement it as a React component and include it globally. Next.js provides a convenient way to wrap all pages with a layout – by customizing the

`_app.js` file to include our Navbar and Footer on every page. Alternatively, you can add the Navbar in each page, but it's better to define it once.

- Create a new file `components/Navbar.js` (or `.jsx`). Inside, define a functional component that returns the navigation bar JSX. For example:

```
// components/Navbar.js
import Link from 'next/link';
import { useSession, signOut } from 'next-auth/react';

export default function Navbar() {
  const { data: session } = useSession();  // to determine if user is logged in

  return (
    <nav className="navbar">
      {/* Logo and site name */}
      <div className="logo">
        <Link href="/"><img src="/logo.png" alt="BELCO Alliance Logo" /></Link>
      </div>

      {/* Nav links */}
      <ul className="nav-links">
        <li><Link href="/programs">Programs</Link></li>
        <li><Link href="/blog">Blog</Link></li>
        <li><Link href="/partners">Partners</Link></li>
        <li><Link href="/about">About</Link></li>
        <li><Link href="/contact">Contact</Link></li>
      </ul>

      {/* Account section on the right */}
      <div className="account-link">
        {!session && (
          // If not logged in, show Login (and maybe Register) links
          <Link href="/login">
            <a><i className="fas fa-user-circle"></i> Login</a>
          </Link>
        )}
        {session && (
          // If logged in, show either a profile link or logout button
          <>
            <span>Welcome, {session.user.name}!</span>
            {session.user.isAdmin && (
              <Link href="/admin"><a>Manage Content</a></Link>
            )}
            <button onClick={() => signOut()}>Logout</button>
          </>
        )}
      </div>
    </nav>
  );
}
```

- In the above snippet, we use `next-auth`'s `useSession` hook to check if a user is logged in. If not, we display a Login link (which goes to the login page). If yes, we greet the user by name and show a Logout button, and if the user is an admin, we also show a "Manage Content" link to the admin dashboard. You can adjust this to your design (for example, you might use an icon for the account and a dropdown menu). The links use `next/link` which is the Next.js way to do client-side navigation between pages.
- *Styling:* You can write CSS for `.navbar`, `.nav-links`, etc., in a global stylesheet (e.g., `styles/globals.css`) or use a CSS module. The Figma shows the navbar with a white background and the site logo on the left, with links on the right. Use flexbox or grid to layout these elements. If you prefer, you can integrate a responsive design for mobile (hamburger menu), but given time constraints, focus on the desktop layout first.
- **Footer:** Similarly, create `components/Footer.js` with the site's footer. The Figma design shows a large blue footer with presumably the BELCO Alliance info, partner logos, contact info, and links like Privacy Policy and Cookie Policy at the bottom. For now, implement a simplified footer with placeholders for those elements:

```
// components/Footer.js
export default function Footer() {
  return (
    <footer className="footer">
      <div className="footer-content">
        <p>&copy; {new Date().getFullYear()} BELCO Alliance. All rights reserved.</p>
        <p>Privacy Policy | Cookie Policy</p>
      </div>
    </footer>
  );
}
```

- You can expand this with actual content: e.g., logos of founding organizations, a short description or address, etc., as in the design. Style it according to the Figma (blue background, white text).
- **Include Navbar and Footer globally:** Open `pages/_app.js`. This file wraps every page in your app. Modify it to include the Navbar at the top and Footer at the bottom:

```
import Navbar from '../components/Navbar';
import Footer from '../components/Footer';
import { SessionProvider } from "next-auth/react";
import '../styles/globals.css';  // import global styles

function MyApp({ Component, pageProps: { session, ...pageProps } }) {
  return (
    <SessionProvider session={session}>
      <Navbar />
      <main>
        <Component {...pageProps} />
      </main>
      <Footer />
```

```
      </SessionProvider>
    );
  }

  export default MyApp;
```

- We wrap the app with `SessionProvider` from NextAuth, so that `useSession` (in Navbar and elsewhere) knows about the logged-in state. Now, every page will show the Navbar at top and Footer at bottom by default. If you need certain pages without nav/footer (for example, maybe an admin or landing page), you could conditionally render them or implement a custom layout, but in our case the design shows nav/footer everywhere (even login page has the nav menu visible).

With the layout set, any page content we create next will appear between the navbar and footer. Now let's implement each section of the site one by one.

Implementing the Homepage

The **homepage** is the front door of the BELCO Alliance site (accessible at `/`). According to the Figma design, the homepage includes several elements: a welcome banner or hero section, an introduction to BELCO Alliance, a highlight of the alliance founders, a carousel or grid of partner logos, an “About us” snippet, and a “Contact us” form section. We’ll create the `pages/index.js` page to reflect this structure.

The Figma design of the homepage (left) shows a welcome section, an introduction to the alliance, a display of BELCO founders or organization info, followed by a grid of partner logos and a contact form at the bottom. We will implement these sections in Next.js to match the look and feel.

Open `pages/index.js` and start building the JSX. You can break the homepage into subsections (could even create smaller components for each section if you like, but it’s okay to keep it in one file for now):

- **Hero/Welcome Section:** This might include a welcome message like “Welcome to BELCO Alliance” and some graphic or image. Use a large heading (`<h1>`) for the welcome, and a paragraph for a subtext. You can also include a call-to-action button if needed (like “Learn more” linking to About or Programs).
- **About/Introduction:** A section describing what BELCO Alliance is, maybe with a title like “Our alliance” (as seen in design) and a text block. Possibly mention the founding institutions or mission statement.
- **Founders/Organization Section:** The design shows logos of organizations (Fontys, etc.) and maybe a short text “BELCO Founders” or “Our organization”. You can implement this as a series of images (logos) in a carousel or a simple grid. If you want a carousel effect (scrolling logos), you might use a library or custom CSS animation. But a simpler approach is a responsive grid of logos for now, given time constraints. Place these images in the `public` folder (e.g., `public/founder1.png`, etc.) and reference with `<img src="/founder1.png" alt="Founder 1" />`.
- **What’s included / Highlights:** The homepage might list some key offerings (for instance, if BELCO offers exchange modules, etc., maybe highlighted on homepage). If the Figma shows specific content here, you can add it as sub-sections or cards.
- **Contact Us section:** At the bottom, there’s a contact form (the design shows input fields for name, email, message). You can create a basic form with those fields and a Submit button. For now, since implementing

form submission (email sending or database storing of inquiries) is secondary, you can have this form do a simple `onSubmit={...}` that triggers an alert or logs the input. If you want to be thorough, you could create an API route to send an email via Nodemailer or store the message in MongoDB, but given your focus, it's okay to leave actual sending for later or as a future enhancement. The main goal is to have the form in the UI.

Here's a skeletal example of how `pages/index.js` might look:

```
// pages/index.js
export default function Home() {
  return (
    <div className="homepage">
      {/* Hero Section */}
      <section className="hero">
        <h1>Welcome to BELCO Alliance</h1>
        <p>Connecting top institutions worldwide to drive innovation in education.</p>
        {/* Maybe a call-to-action button */}
      </section>

      {/* Alliance Introduction Section */}
      <section className="alliance-intro">
        <h2>Our Alliance</h2>
        <p>BELCO Alliance is a consortium of leading universities collaborating on ... [some
introduction text].</p>
      </section>

      {/* Founders / Partners logos Section */}
      <section className="founders-logos">
        <h2>Founding Institutions</h2>
        <div className="logos-grid">
          <img src="/logos/fontys.png" alt="Fontys University logo" />
          <img src="/logos/partner2.png" alt="Partner2 logo" />
          {/* ...other logos... */}
        </div>
      </section>

      {/* About snippet or Alliance at a Glance */}
      <section className="about-glance">
        <h2>Alliance at a Glance</h2>
        <p>Brief stats or overview of the alliance (e.g., number of programs, countries,
etc.).</p>
      </section>

      {/* Contact Us Section */}
      <section className="contact-us">
        <h2>Contact Us</h2>
        <p>Please fill out the form below to get in touch with us.</p>
        <form onSubmit={(e) => { e.preventDefault(); /* handle submission */ }}>
          <div>
            <label>Name</label>
            <input type="text" name="name" required />
          </div>
```

```
            <div>
              <label>Email</label>
              <input type="email" name="email" required />
            </div>
            <div>
              <label>Message</label>
              <textarea name="message" rows="4" required></textarea>
            </div>
            <button type="submit">Send</button>
          </form>
        </section>
      </div>
    );
  }
```

This is a rough outline; you will need to adjust the content to match the Figma text and style. Use proper HTML semantic tags and classes so you can style each section in CSS. Keep your JSX well-structured (each section wrapped in a `<section>` with a clear class name as above).

After building the homepage structure, add CSS rules in your `globals.css` or a dedicated `Home.module.css` (and import it) to apply the color scheme, spacing, and fonts as per design. For example, the design uses a bright yellow and blue palette, and likely specific fonts. If the Figma doesn't specify exact CSS, just approximate it (e.g., use a clean sans-serif font, match colors from the screenshots).

Tip: It's often helpful to use Flexbox or CSS Grid for the layout of sections. For instance, the hero section can be centered text on a colored background; the logos grid can use CSS grid to layout images in rows/columns; the contact form can use flex or grid for the form fields.

At this point, you can run the dev server and verify that the homepage looks like the design (it's okay if it's not perfect; you can refine CSS later). The Navbar and Footer should appear as well (since we added them in `_app.js`).

Implementing the Programs Page

Next, create the **Programs** page (`pages/programs.js`). This page will detail the various programs offered by the BELCO Alliance. From the design, it seems to have a header (perhaps "BELCO Programs") and then sections for each program (e.g., "BELCO Exchange Modules", "BELCO Tripartite Programme (3I)", etc.) with descriptions and maybe bullet points of what's included.

Structure this page as static content for now (we don't necessarily need dynamic data here unless you plan to load program info from a database, which might not be necessary). Use your HTML/CSS skills to format it similar to the design.

For example, `pages/programs.js` could be:

```
export default function Programs() {
  return (
    <div className="programs-page">
```

```
        <h1>BELCO Programs</h1>
        <section className="program">
          <h2>BELCO Exchange Modules</h2>
          <p>Description of the exchange modules program... (from Figma content).</p>
          <h3>What's included:</h3>
          <ul>
            <li>Courses at partner universities</li>
            <li>Cultural exchange opportunities</li>
            <li>Credits transfer and recognition</li>
            {/* etc based on design */}
          </ul>
        </section>

        <section className="program">
          <h2>BELCO Tripartite Programme (3I)</h2>
          <p>Description of the 3I program... (from Figma content).</p>
          <h3>What's included:</h3>
          <ul>
            <li>International internships</li>
            <li>Industry collaboration projects</li>
            <li>Joint degrees</li>
          </ul>
        </section>

        {/* Add other programs similarly if any */}
      </div>
    );
  }
```

Style the page to match the alliance branding (e.g., headings in the alliance's colors). The Figma screenshot for Programs shows a design with a blue background on the header and some decorative shapes. You can incorporate a styled header by giving the container a background color, and adding images or shapes (from the Figma assets if provided). Again, focus on structure first, style second.

After adding content, test that you can navigate to http://localhost:3000/programs and see the page. The Navbar should highlight "Programs" (you can manually add an active class in Navbar if `router.pathname` matches "/programs" to style the active link).

Implementing the Blog Listing Page

The **Blog** page will list articles, news, and research updates. This page (route `/blog`) is dynamic in that it will eventually display a list of articles fetched from the database. For now, we'll set up the structure and a way to display multiple article "cards" or previews, then later we'll connect it to MongoDB data.

According to the design, the blog page likely has a title like "Articles, news and research" and then a grid of article cards. Each article card might show an image, a title, maybe a short summary or category. In the Figma screenshot, we see cards like "Acceleration academies", "The rise of virtual exchange programs", etc. We will create a layout to display these.

Steps to build the Blog page (`pages/blog/index.js`):

- Start with a heading for the page:

```
<h1>Articles, News and Research</h1>
```

- (Use whatever exact title the design uses.)
- Display a list of article previews. For now, create a placeholder array of articles in the code to simulate data. For example:

```
const sampleArticles = [
  { id: '1', title: 'Acceleration academies', summary: 'Accelerators in education ...',
image: '/images/acceleration_academies.jpg' },
  { id: '2', title: 'Global Outreach Programs', summary: 'The value of global programs ...',
image: '/images/global_programs.jpg' },
  // ... more items
];
```

- Then map over this array to render a grid of article cards:

```
<div className="articles-grid">
  {sampleArticles.map(article => (
    <div key={article.id} className="article-card">
      <img src={article.image} alt={article.title} />
      <h3>{article.title}</h3>
      <p>{article.summary}</p>
      <a href={`/blog/${article.id}`}>Read more</a>
    </div>
  ))}
</div>
```

- The above uses a normal `<a href>` linking to the article page. You could use Next's `<Link>` for client-side transitions:

```
<Link href={`/blog/${article.id}`}>
  <a className="read-more">Read more</a>
</Link>
```

- The important part is that each card links to `/blog/[id]` which we will implement as the article detail page.
- Style the grid with CSS so that on desktop, the cards are in a 3-column or 4-column grid (depending on design). Ensure the images are nicely sized (maybe use a fixed height or aspect ratio for images for

consistency). Each card should have a hover effect if you want (like a slight lift or shadow).

Dynamic Data: Once we connect to MongoDB, we won't use `sampleArticles` anymore. Instead, we'll fetch real articles from the database. We can use Next.js data fetching methods to do this. For a simple approach, we'll use **Server-Side Rendering** via `getServerSideProps` on this page to load articles at request time (so it always shows the latest). We will implement this after setting up the database. Essentially, `getServerSideProps` will connect to the DB, fetch all articles, and pass them as props to the page. But *don't worry about coding that just yet* – we'll revisit it once our database and Article model are ready. For now, confirm the blog page looks correct with dummy content.

Navigate to http://localhost:3000/blog to ensure the page is rendered. Clicking on "Read more" won't work until we create the dynamic article page, which is our next step.

Implementing Article Detail Pages

Each blog article will have its own page, which we've set up to use a dynamic route: `pages/blog/[id].js`. This will display the full content of a single article. Our goal is to route from the Blog listing to an article page (e.g., clicking "Acceleration academies" leads to `/blog/1` or some identifier).

Setting up the dynamic route: In Next.js pages router, a file named `[id].js` inside the `blog` folder means any URL `/blog/<id>` will render that page. We can access the `<id>` value in the page's code to know which article to load.

- Create `pages/blog/[id].js`. This page component needs to determine which article to show based on the URL. Next.js gives us `context.params` (for `getServerSideProps`) or a router query (for client side) to get the `id`. We will likely fetch the article from our database by ID. But we can start by handling it with placeholder data similar to the blog page, then integrate real data later.
- For now, let's assume the `id` is simply the index or some slug. We can use the same `sampleArticles` array approach or a simple switch:

```
import { useRouter } from 'next/router';

export default function ArticlePage() {
  const router = useRouter();
  const { id } = router.query;
  // You could find the article from sampleArticles by id here if data is client-side.
  // But better, we'll use server-side fetching to load the article.
  return (
    <div className="article-page">
      <h1>Article Title</h1>
      <img src="/images/placeholder.jpg" alt="Article image" />
      <p>Full content of the article goes here...</p>
    </div>
  );
}
```

- The above is a simple placeholder. We will not rely on client-side `useRouter` for the real implementation because it's better to fetch the data on the server side. So let's plan the server-side approach:

Server-side Data Fetching for Article: We will use `getServerSideProps` in this page to fetch the article from the database by ID once the DB is set up. The pattern will look like:

```
import dbConnect from '../../lib/dbConnect';
import Article from '../../models/Article';

export async function getServerSideProps({ params }) {
  await dbConnect();
  const article = await Article.findById(params.id);
  if (!article) {
    return { notFound: true };
  }
  return { props: { article: JSON.parse(JSON.stringify(article)) } };
}

export default function ArticlePage({ article }) {
  // render using article prop
}
```

What's happening here is:

- We connect to the database,
- Find the article document by the ID from the URL,
- If not found, return a 404 page,
- Otherwise pass the article data as a prop to the component.
  We will implement the `dbConnect` utility and the `Article` model soon in the database section. The `JSON.parse(JSON.stringify(article))` trick is often used to convert a Mongoose document into a plain object safe for serialization (because Next.js will serialize props to JSON to send to the client).

For now, if you want to see an article page in action, you can temporarily hardcode some content or reuse the sample array:

```
// pseudo-code for demonstration only
export async function getServerSideProps({ params }) {
  const articleData = sampleArticles.find(a => a.id === params.id);
  return { props: { articleData: articleData || null } };
}

export default function ArticlePage({ articleData }) {
  if (!articleData) return <h1>Article not found</h1>;
  return (
    <div>
      <h1>{articleData.title}</h1>
      <img src={articleData.image} alt={articleData.title} />
```

```
      <p>{/* Imagine this is the full content, for now use summary */}{articleData.summary}
</p>
    </div>
  );
}
```

This way, while developing, you can navigate to `/blog/1` and see the page. Once the backend is ready, we'll replace this with real content and possibly a rich text for the article body.

Design-wise, style the article page to make it readable: maybe the title is large and bold, the image is full-width or nicely framed, and the content text is well spaced. You might also include the author name and publication date if those are relevant (if the sitemap or design indicated those). You can store and display those once we have the Article model (e.g., `article.author`, `article.date`).

Now our public-facing pages (home, programs, blog list, article, partners) are scaffolded. Next, we'll integrate the database to provide real data for the blog and partners pages, and implement the account system.

Implementing the Partners Page

The **Partners** page (`/partners`) will showcase the alliance's partner institutions. The design shows a list (or grid) of university names with images, likely with their logos or campus photos, and possibly categorized by country. It also has some text at the bottom like "Why we partner with top institutions..." and a call to action "Join our alliance network".

We will create `pages/partners.js` to display all partner universities. Similar to the blog, this data will come from the database (the admin will be able to add universities via the CMS). To build quickly, we can hardcode some example partners and then later fetch from MongoDB.

- Outline the structure:

```
export default function Partners() {
  return (
    <div className="partners-page">
      <h1>Our Partners</h1>
      <p>We partner with top institutions worldwide to ... [some introductory text].</p>

      <div className="partners-grid">
        {/* Partner items */}
      </div>

      <section className="partners-info">
        <h2>Why we partner with the best institutions</h2>
        <p>[Some text from Figma about the value of partnerships...]</p>
        <h2>Join our alliance network</h2>
        <p>If you're interested in becoming a partner, [instructions or link to contact].
</p>
        <a href="/contact" className="button">Contact Us</a>
      </section>
```

```
    </div>
  );
}
```

- In the `.partners-grid`, we'll list each partner. Suppose we have a `sampleUniversities` list (for initial development):

```
const sampleUniversities = [
  { id: 'uni1', name: 'Fontys University', country: 'Netherlands', image:
'/partners/fontys.jpg', website: 'https://fontys.edu' },
  { id: 'uni2', name: 'University of X', country: 'Country Y', image: '/partners/uni_x.jpg',
website: 'https://universityx.edu' },
  // ... etc.
];
```

- We can map this:

```
<div className="partners-grid">
  {sampleUniversities.map(uni => (
    <div key={uni.id} className="partner-card">
      <img src={uni.image} alt={uni.name} />
      <h3>{uni.name}</h3>
      <p>{uni.country}</p>
      <a href={uni.website} target="_blank" rel="noopener noreferrer">Visit site</a>
    </div>
  ))}
</div>
```

- This will display each partner's image, name, country, and a link to their website (opening in a new tab with `target="_blank"`).
- Style this page: likely a grid of cards (the Figma [partners page screenshot] shows a neat grid of images with names next to them, possibly a two-column layout of logo and name). You can achieve that either with a CSS grid or flex wrapping. Make sure images are uniform size (perhaps same width and height) or at least consistently styled (maybe all logos or campus images with a similar dimension).
- The bottom section ("Why we partner… / Join our network") is static text. Just ensure to include it for completeness, as it gives context and a call to action.

The "Partners" page design (center) displays a list of partner institutions with their logos/photos and names, likely arranged in a scrollable section. We will fetch the partner entries from MongoDB and render them in a similar grid layout with images and names. The design also includes a descriptive section at the bottom explaining the partnerships and inviting new partners.

Dynamic Data: As with the blog, once connected to MongoDB, we'll retrieve the partners list from the database. We'll have a **University** model for partners. On this page, we can use `getServerSideProps` to fetch all universities from the DB and pass them as props. The code would mirror the earlier approach:

```
// (to be implemented after DB setup)
export async function getServerSideProps() {
  await dbConnect();
  const uniRecords = await University.find({});
  const universities = uniRecords.map(doc => {
    const uni = doc.toObject();
    uni._id = uni._id.toString();
    return uni;
  });
  return { props: { universities } };
}
```

Then use `universities` prop instead of `sampleUniversities` for rendering. We'll integrate this soon.

At this point, our front-end for all main pages is in place. You can navigate through the site (using the Navbar) to see homepage, programs, blog, an example article, partners. They all use static/dummy data now. The next step is to make the site functional by setting up the **account system and database**.

Setting Up MongoDB Atlas and Data Models

Now we'll connect our Next.js application to a **MongoDB database** to store persistent data: users (for accounts), articles (for blog), and universities (for partners). We'll use **MongoDB Atlas** (a cloud-hosted MongoDB service) so you don't need to install MongoDB locally. We'll also use **Mongoose** as an ODM (Object Data Modeling) library to define schemas and interact with the DB conveniently.

1. Create a MongoDB Atlas cluster: If you haven't already:

- Go to the MongoDB Atlas website and sign up (there's a free tier).
- Create a new project and then a new cluster (choose a free cluster).
- In the Atlas UI, create a database user (with username/password) that our app will use to connect.
- Obtain the connection URI. Atlas provides something like:
  `mongodb+srv://<username>:<password>@cluster0.abcd123.mongodb.net/<dbname>?retryWrites=true&w=majority`
  Replace `<username>`, `<password>` with your DB user credentials, and `<dbname>` with the name of your database (e.g., `belco`).
- In the Network Access settings on Atlas, allow access from your IP (or set to allow all IPs for development).

2. Store the connection string in environment variables: In your Next.js project, create a file `.env.local` (Next.js will load this for environment vars in development and Vercel can use them in production). Add:

```
MONGODB_URI=mongodb+srv://<username>:<password>@cluster0.../<dbname>?
retryWrites=true&w=majority
NEXTAUTH_SECRET=someRandomStringForAuth
```

The `NEXTAUTH_SECRET` can be any random string; it's used by NextAuth to encrypt session data (you can use an online generator to create a secure random value). In production (Vercel), you will set these as well. **Never commit `.env.local` to git**, as it contains sensitive info (you can add it to `.gitignore` if not already).

3. Create a Mongoose connection utility: We'll make a file to handle connecting to MongoDB, ensuring we don't open multiple connections during development (Next.js hot-reloading can re-run code). A common pattern (from Next.js official examples) is to use a global cached connection. Create a folder `lib/` in your project, and inside it `lib/dbConnect.js`:

```
// lib/dbConnect.js
import mongoose from 'mongoose';

const MONGODB_URI = process.env.MONGODB_URI;
if (!MONGODB_URI) {
  throw new Error('Please define the MONGODB_URI environment variable inside .env.local');
}

/**
 * Global is used here to maintain a cached connection across hot reloads in development.
 * In production, a new connection is created for each invocation.
 */
let cached = global.mongoose;
if (!cached) {
  cached = global.mongoose = { conn: null, promise: null };
}

async function dbConnect() {
  if (cached.conn) {
    return cached.conn;
  }
  if (!cached.promise) {
    const opts = {
      bufferCommands: false,
    };
    cached.promise = mongoose.connect(MONGODB_URI, opts).then(mongoose => mongoose);
  }
  cached.conn = await cached.promise;
  return cached.conn;
}

export default dbConnect;
```

This function `dbConnect()` will ensure we have a single connection to the database. When first called, it uses `mongoose.connect()` to connect to Atlas, then caches the connection. Subsequent calls reuse the cached connection. This is important in serverless environments (like Vercel) to avoid connection overhead on each request.

4. Define Mongoose schemas and models: We need models for **User**, **Article**, and **University** data. We'll create a `models/` directory (you can put it in `lib/models/` or directly under project root). Let's define each:

- **User Model:** Fields: `name` (String), `email` (String, unique), `password` (String), `role` or `isAdmin` (Boolean). We will store hashed passwords, not plain text. We'll also mark `email` unique to prevent duplicates. In Mongoose, once a schema is defined, we create a model or use an existing one if already created (to avoid overwriting during hot reload).
- Create `models/User.js`:

```
import mongoose from 'mongoose';

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  isAdmin: { type: Boolean, default: false }
});

// This is important to avoid model recompilation in dev
export default mongoose.models.User || mongoose.model('User', UserSchema);
```

- This defines a User with those fields. We'll treat `isAdmin` as a flag to differentiate the admin user(s) from regular users.
- **Article Model:** Fields might include `title`, `content` (the full text or HTML of the article), maybe an `excerpt` or `summary`, an `image` URL, an `author` name, and perhaps a `date` field (default to current date). We can also include `tags` or `category` if needed, but to keep it simple, we'll stick to essentials.
- Create `models/Article.js`:

```
import mongoose from 'mongoose';

const ArticleSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  summary: { type: String },  // short summary for preview (optional)
  image: { type: String },    // URL or path to an image
  author: { type: String },   // could also reference a User
  createdAt: { type: Date, default: Date.now }
});

export default mongoose.models.Article || mongoose.model('Article', ArticleSchema);
```

- Here, we allow summary and image to be optional (admin might or might not include them). The `createdAt` will timestamp the article.
- **University Model:** Fields: `name` (String), `country` (String), maybe `city` (String) if needed, `website` (String), `image` (String for a logo or photo URL). Create `models/University.js`:

```
import mongoose from 'mongoose';

const UniversitySchema = new mongoose.Schema({
  name: { type: String, required: true },
  country: { type: String, required: true },
  city: { type: String },
  website: { type: String },
  image: { type: String }  // URL or path to logo image
});

export default mongoose.models.University || mongoose.model('University', UniversitySchema);
```

- We mark `name` and `country` as required. We can add `unique: true` on name if we want to avoid duplicates, but it's possible different countries have universities with the same name, so maybe not unique globally.

Each model file uses the pattern `mongoose.models.ModelName || mongoose.model('ModelName', Schema)` to prevent re-defining the model on every HMR (Hot Module Reload) during development.

5. Connect and test the database connection (optional): You can write a quick test by creating a dummy API route or even using `getServerSideProps` in one page to see if connection succeeds. For example, add a temporary route `pages/api/test-db.js`:

```
import dbConnect from '../../lib/dbConnect';
import User from '../../models/User';

export default async function handler(req, res) {
  try {
    await dbConnect();
    const users = await User.find({});
    res.status(200).json({ count: users.length });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Database connect failed' });
  }
}
```

Start your dev server and hit http://localhost:3000/api/test-db. If everything is set up correctly, it should return `{"count":0}` (assuming no users yet) or at least not error out. If you get an error, check the console for issues

(like incorrect connection string or schema issues). Once confirmed, you can remove this test route.

At this stage, we have configured our Next.js app to talk to MongoDB and defined the data models. Now we can integrate this into our pages and features:

- We will use `dbConnect()` and the models in our Next.js data fetching functions (`getServerSideProps`) to load data for the Blog and Partners pages, instead of the dummy data.
- We will implement **NextAuth** to handle login, which will use the User model (for checking credentials and creating new users).
- We will build API routes for the admin CMS to create and delete Article/University entries, using our models.

Let's proceed to the account system next, since having authentication in place will also allow us to secure the admin routes when we get to them.

Implementing User Authentication with NextAuth (Login & Register)

User accounts are needed for the "Account" section of the site – especially to distinguish an admin user who can manage content. We'll use **NextAuth.js** to handle authentication in our Next.js app. NextAuth supports many providers (Google, etc.), but here we will use the **Credentials Provider** for classic email/password login since we want users to register with a custom account.

1. NextAuth configuration: NextAuth in the pages directory expects an API route at `pages/api/auth/[...nextauth].js`. Create this file (inside `pages/api/auth/`). In it, we'll set up the credentials provider.

```
// pages/api/auth/[...nextauth].js
import NextAuth from "next-auth";
import CredentialsProvider from "next-auth/providers/credentials";
import dbConnect from "../../../lib/dbConnect";
import User from "../../../models/User";
import bcrypt from "bcryptjs";

export default NextAuth({
  providers: [
    CredentialsProvider({
      name: "Credentials",
      credentials: {
        email: { label: "Email", type: "text", placeholder: "you@example.com" },
        password: { label: "Password", type: "password" }
      },
      async authorize(credentials) {
        // Connect to DB and find the user by email
        await dbConnect();
        const user = await User.findOne({ email: credentials.email }).select("+password");
        if (!user) {
          throw new Error("No user found with that email");
        }
        // Check password
        const pwValid = await bcrypt.compare(credentials.password, user.password);
```

```
          if (!pwValid) {
            throw new Error("Incorrect password");
          }
          // If credentials are valid, return the user object (omit password)
          return {
            id: user._id.toString(),
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin
          };
        }
      })
    ],
    secret: process.env.NEXTAUTH_SECRET,
    session: {
      strategy: "jwt"
    },
    callbacks: {
      // Include user.id and isAdmin in the JWT
      jwt({ token, user }) {
        if (user) {
          token.id = user.id;
          token.isAdmin = user.isAdmin;
        }
        return token;
      },
      session({ session, token }) {
        if (token) {
          session.user.id = token.id;
          session.user.isAdmin = token.isAdmin;
        }
        return session;
      }
    }
  });
```

Let's break down what this configuration does:

- We import `CredentialsProvider` and define it with an `authorize` function. This function runs whenever someone tries to log in with credentials. We:
  - Connect to the database.
  - Find the user by the provided email. We use `.select("+password")` because by default our User schema might not return password (we didn't set `select: false`, but if we did, this is how to override). We want the hashed password to compare.
  - If user not found, throw an error.
  - If found, use `bcrypt.compare()` to compare the provided password with the hashed password in the DB.
  - If password doesn't match, throw an error.

- If it matches, return a plain object representing the user (NextAuth will attach this to `token`).
- We deliberately construct our own object to return (id, name, email, isAdmin) and leave out the password. NextAuth will generate a JWT that stores these fields.

- We set `session.strategy: "jwt"` meaning we'll use JSON Web Tokens (default in NextAuth for credentials) to track sessions.
- We define **callbacks** for `jwt` and `session`:
  - The `jwt` callback takes the token and user (if first sign in) and adds `id` and `isAdmin` to the token. This ensures our JWT payload contains the admin flag.
  - The `session` callback takes the token and makes those available in the session object. So `session.user.isAdmin` will be true/false based on the DB.
    These callbacks implement role-based session info, which is how we will later restrict admin routes (only allow session.user.isAdmin).
- We pass `secret` from env (the random string we set) for token signing.

This configuration means:

- NextAuth exposes endpoints at `/api/auth/signin`, `/api/auth/signout`, etc., which the NextAuth client will use.
- We can use NextAuth's React hooks (`useSession`, `signIn`, `signOut`) in our components (we already used `useSession` in Navbar). The `SessionProvider` in `_app.js` is set, so this is all wired up.

2. Registration flow: NextAuth's credentials provider doesn't include a built-in "register" – we have to create a user ourselves (e.g., via a separate API route for sign-up). We'll implement a registration route at `pages/api/register.js`. This will allow new users to sign up by sending their name, email, password.

Create `pages/api/register.js`:

```
import dbConnect from "../../lib/dbConnect";
import User from "../../models/User";
import bcrypt from "bcryptjs";

export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ message: "Method not allowed" });
  }
  const { name, email, password } = req.body;
  if (!name || !email || !password) {
    return res.status(400).json({ message: "Missing required fields" });
  }
  try {
    await dbConnect();
    // Check if user already exists
    const existing = await User.findOne({ email });
    if (existing) {
      return res.status(409).json({ message: "User with that email already exists" });
    }
    // Hash the password
```

```
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
    // Create new user (we can decide if the first user should be admin)
    const isFirstUser = (await User.countDocuments({})) === 0;
    const user = await User.create({
      name,
      email,
      password: hashedPassword,
      isAdmin: isFirstUser ? true : false  // e.g., make first user an admin
    });
    return res.status(201).json({ message: "User created", userId: user._id });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ message: "Server error" });
  }
}
```

This route:

- Ensures it only handles POST.
- Validates name, email, password are present.
- Connects to DB, checks if a user with that email exists (to prevent duplicate registration).
- Uses `bcrypt` to hash the password with a salt (10 rounds).
- We also included a little logic: if this is the very first user in the database, we mark them as admin (`isAdmin: true`). This is one way to ensure you (as the first registrant) become admin. Alternatively, you could skip this and manually flag the desired admin in the database, or check if the email matches a certain domain. But making the first user admin is a quick solution.
- Creates the user in the DB.
- Returns a success response (we include `userId` just in case we want to use it, but we might not need to).

Now we have a way to create new users via an API call. Next, we build the frontend pages for login and register and hook them up to these routes.

3. Login Page (`pages/login.js`): This page will present a form for email and password, and when submitted, call NextAuth's `signIn` function for the credentials provider.

```
import { signIn } from "next-auth/react";
import { useState } from "react";
import { useRouter } from "next/router";

export default function LoginPage() {
  const router = useRouter();
  const [error, setError] = useState(null);

  const handleLogin = async (e) => {
    e.preventDefault();
```

```
      setError(null);
      const email = e.target.email.value;
      const password = e.target.password.value;
      const result = await signIn("credentials", {
        redirect: false, // we'll handle navigation
        email,
        password
      });
      if (result.error) {
        setError(result.error);
      } else {
        // Logged in successfully
        router.push("/");  // or redirect to profile/dashboard as needed
      }
    };

    return (
      <div className="account-page login-page">
        <h1>Welcome!</h1>
        {error && <p className="error">{error}</p>}
        <form onSubmit={handleLogin}>
          <label>Email:</label>
          <input name="email" type="email" required />
          <label>Password:</label>
          <input name="password" type="password" required />
          <button type="submit">Login</button>
        </form>
        <p>Forgot password? (We will implement password reset later or manually.)</p>
        <p>Don't have an account? <a href="/register">Register here</a></p>
      </div>
    );
  }
```

Explanation:

- We use `signIn("credentials", {...})` from NextAuth to attempt login with the given email/password. We set `redirect: false` so it doesn't automatically redirect (which by default goes to the homepage on success). Instead, we handle success by redirecting ourselves (`router.push("/")` or maybe to `/admin` if admin – you can decide).
- If there's an error, NextAuth will give an error message (from our throw in authorize). We capture it and display to the user.
- The form is straightforward. We also provide a link to register page.

You can style this page to match the Figma (the design shows a simple centered form with a heading "Welcome!", input boxes and a bright yellow Login button). Add appropriate classes and CSS. For example, `.account-page` can set max-width and margin auto to center the form, etc.

Also note: the Figma shows a "Forgot password" screen. Implementing password reset fully (sending emails with tokens) is a bit complex for the time frame. You can include a placeholder text or link for "Forgot password" that

maybe just informs user to contact support, or implement a simple version (like if a user clicks it, auto-generate a random new password and email it, but emailing requires setting up an email server or API). Given the scope, it's okay to omit actual password reset functionality or mark it as future work.

4. Register Page (`pages/register.js`): This will have a form for name, email, password, and will call our `/api/register` route. After successful registration, we can automatically log the user in or redirect them to login.

```
import { useState } from "react";
import { useRouter } from "next/router";
import { signIn } from "next-auth/react";

export default function RegisterPage() {
  const router = useRouter();
  const [error, setError] = useState(null);

  const handleRegister = async (e) => {
    e.preventDefault();
    setError(null);
    const name = e.target.name.value;
    const email = e.target.email.value;
    const password = e.target.password.value;
    try {
      const res = await fetch("/api/register", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ name, email, password })
      });
      const data = await res.json();
      if (!res.ok) {
        throw new Error(data.message || "Failed to register");
      }
      // Registration successful
      // Option 1: Automatically sign the user in
      await signIn("credentials", { email, password, redirect: false });
      router.push("/");  // redirect to home or profile
    } catch (err) {
      setError(err.message);
    }
  };

  return (
    <div className="account-page register-page">
      <h1>Register</h1>
      {error && <p className="error">{error}</p>}
      <form onSubmit={handleRegister}>
        <label>Name:</label>
        <input name="name" type="text" required />
        <label>Email:</label>
        <input name="email" type="email" required />
        <label>Password:</label>
```

```
          <input name="password" type="password" required minLength={6} />
          <button type="submit">Register</button>
        </form>
        <p>Already have an account? <a href="/login">Login here</a></p>
      </div>
    );
  }
```

Key points:

- On submit, we `fetch("/api/register")` with the form data. We expect a JSON response. If not OK, we throw an error to display.
- On success, we call `signIn("credentials", { email, password })` to log the user in immediately (so they don't have to go back to login form). This uses the same NextAuth flow, so it will set the session cookie. We then redirect to homepage (or you could choose to redirect to `/admin` if the user is admin, but we might not know here; we can always redirect to home, and the Navbar's "Manage Content" will appear if they are admin).
- We display any errors (like email taken, etc., which our API returns as 409 with message).

Styling: similar to login, ensure it matches design (it likely looks almost the same as login form).

Now test the flow in the browser:

- Go to `/register`, fill in details, submit. It should create the user and then redirect you (while automatically logging in).
- Check that the Navbar now maybe shows "Welcome, [Name]!" and a Logout (and Manage Content if you were made admin).
- Then try logging out (`signOut()` from Navbar), then go to `/login`, try logging in with the credentials you made.
- Try an incorrect password to see error message.

This verifies the account system. If something fails:

- Check terminal output for any errors (like connection issues or unique constraint violation if you reused an email).
- Make sure the NextAuth secret is set (NextAuth will warn in console if not).
- Ensure `bcrypt.compare` and `bcrypt.hash` are working (we used `bcryptjs` which is pure JS and should be fine).

NextAuth session and roles: Thanks to the callbacks we set in NextAuth, you can access `session.user.isAdmin` to see if the logged-in user is admin. We should use this to conditionally show the "Manage Content" link (we did) and more importantly, **protect the admin pages and API routes** from unauthorized access. We will address route protection in the next section.

One more detail: The `authorize` function returns error messages which NextAuth by default might show as generic. We threw errors like "No user found" or "Incorrect password". By catching `result.error` in our login

form, we displayed it. That's user-friendly enough. If you want to customize further (like showing a specific message on the login page via query param), NextAuth allows an error redirect, but our approach is fine.

Now, with authentication in place, let's implement the **Admin CMS** functionality, where an admin can add or remove articles and partner universities.

Building the Admin CMS Dashboard (Add/Delete Articles and Universities)

The admin dashboard is where an authorized admin user can manage content:

- **Post Article:** Add a new blog article.
- **Delete Article:** Remove an existing blog article.
- **Post University:** Add a new partner institution.
- **Delete University:** Remove a partner.

According to the sitemap and Figma, these functions might be on one page with a simple internal menu or tabs. The design shows buttons or tabs for each action ("Post Article", "Delete Article", "Post University", "Delete University") and the relevant form or list below when selected. We will implement this in the `pages/admin/index.js` page for simplicity.

Only the admin should access this page. We will enforce that using a **server-side redirect**: check the session on the server; if not admin, redirect them away.

1. Protect the Admin page (server-side): Next.js allows us to use `getServerSideProps` to run on each request. We can use `getSession` from NextAuth to get the session server-side. This requires passing the context. Let's add at top of `pages/admin/index.js`:

```
import { getSession } from "next-auth/react";
import dbConnect from "../../lib/dbConnect";
import Article from "../../models/Article";
import University from "../../models/University";

export async function getServerSideProps(context) {
  const session = await getSession(context);
  if (!session || !session.user?.isAdmin) {
    // Not logged in or not an admin -> redirect to home (or login)
    return {
      redirect: { destination: "/", permanent: false }
    };
  }
  // If admin, fetch current articles and universities for display
  await dbConnect();
  const articles = await Article.find({}, "title");  // only get title (and _id)
  const universities = await University.find({}, "name");
  // Serialize data for transfer
  const articlesData = articles.map(doc => ({ id: doc._id.toString(), title: doc.title }));
  const universitiesData = universities.map(doc => ({ id: doc._id.toString(), name: doc.name
}));
  return {
```

```
    props: { articlesData, universitiesData, session }
  };
}
```

This will:

- Redirect any non-admin away (so even if someone manually visits `/admin`, they can't see it unless logged in as admin).
- If admin, connect to DB and fetch all articles and universities, but only selecting basic fields (title and name) since for deletion listing we just need an identifier and a name/title.
- We convert the Mongoose docs to simple JS objects (`id` as string, etc.) because Next.js doesn't serialize ObjectIds or complex objects directly.

Now, the component itself will receive `articlesData` and `universitiesData` as props to use in rendering the lists for deletion.

2. Admin Dashboard UI and State: We'll present the admin with a set of options and forms. There are many ways to do it. For clarity:

- We could show all forms at once one below the other. But that might be cluttered.
- Better, use a small internal menu or toggle. For example, render two tabs: "Manage Articles" and "Manage Universities", or four buttons and show corresponding form/list when that button is clicked.

To keep it simple, let's do two sections on the page: one for Articles management and one for Universities management, separated by headings. Within each section, we'll have both the add form and the delete list.

(Alternatively, you could implement a tab interface where only one is visible at a time, but doing both in one page is okay if well-structured.)

We will implement:

- **Add Article Form**: fields for title, content, perhaps image URL.
- **List of Articles with Delete buttons**: using `articlesData` from props.
- **Add University Form**: fields for name, country, (city, website, image URL).
- **List of Universities with Delete buttons**: using `universitiesData`.

And we'll handle form submissions with API calls to our endpoints (which we will write next). Each form submission or delete action will call an API route, which will modify the DB. After an add, we might want to refresh the list (to show the new item); after delete, remove from list. We can handle that with a simple client-side update or by refreshing the page. To avoid complexity, we can cheat by using page reload on success (simplest way to get updated data).

Let's outline `pages/admin/index.js` component:

```
import { useState } from "react";
```

```
export default function AdminDashboard({ articlesData, universitiesData }) {
  const [articles, setArticles] = useState(articlesData);
  const [universities, setUniversities] = useState(universitiesData);
  const [loading, setLoading] = useState(false);
  const [msg, setMsg] = useState(null);

  // Handler for adding article
  const handlePostArticle = async (e) => {
    e.preventDefault();
    setLoading(true);
    setMsg(null);
    const title = e.target.title.value;
    const content = e.target.content.value;
    const image = e.target.image.value;
    try {
      const res = await fetch("/api/articles", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ title, content, image })
      });
      if (!res.ok) throw new Error("Failed to add article");
      const data = await res.json();
      setMsg("Article added!");
      // Update articles list UI:
      setArticles(prev => [...prev, { id: data.article._id, title: data.article.title }]);
      e.target.reset();  // clear form
    } catch (err) {
      console.error(err);
      setMsg(err.message);
    } finally {
      setLoading(false);
    }
  };

  // Handler for deleting article
  const handleDeleteArticle = async (id) => {
    setLoading(true);
    setMsg(null);
    try {
      const res = await fetch(`/api/articles/${id}`, { method: "DELETE" });
      if (!res.ok) throw new Error("Failed to delete article");
      setArticles(prev => prev.filter(a => a.id !== id));
      setMsg("Article deleted.");
    } catch (err) {
      console.error(err);
      setMsg(err.message);
    } finally {
      setLoading(false);
    }
  };

  // Similar handlers for University:
```

```
  const handlePostUniversity = async (e) => {
    e.preventDefault();
    setLoading(true);
    setMsg(null);
    const name = e.target.name.value;
    const country = e.target.country.value;
    const website = e.target.website.value;
    const image = e.target.image.value;
    try {
      const res = await fetch("/api/universities", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ name, country, website, image })
      });
      if (!res.ok) throw new Error("Failed to add university");
      const data = await res.json();
      setMsg("University added!");
      setUniversities(prev => [...prev, { id: data.university._id, name:
data.university.name }]);
      e.target.reset();
    } catch (err) {
      console.error(err);
      setMsg(err.message);
    } finally {
      setLoading(false);
    }
  };

  const handleDeleteUniversity = async (id) => {
    setLoading(true);
    setMsg(null);
    try {
      const res = await fetch(`/api/universities/${id}`, { method: "DELETE" });
      if (!res.ok) throw new Error("Failed to delete university");
      setUniversities(prev => prev.filter(u => u.id !== id));
      setMsg("University deleted.");
    } catch (err) {
      console.error(err);
      setMsg(err.message);
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="admin-dashboard">
      <h1>Admin Dashboard</h1>
      {msg && <p className="message">{msg}</p>}

      <section>
        <h2>Manage Articles</h2>
        <h3>Add New Article</h3>
```

```
        <form onSubmit={handlePostArticle}>
          <input name="title" placeholder="Article title" required /><br/>
          <textarea name="content" placeholder="Article content" rows="5" required>
</textarea><br/>
          <input name="image" placeholder="Image URL (optional)" /><br/>
          <button type="submit" disabled={loading}>Post Article</button>
        </form>

        <h3>Existing Articles</h3>
        <ul>
          {articles.map(article => (
            <li key={article.id}>
              {article.title}
              <button onClick={() => handleDeleteArticle(article.id)} disabled=
{loading}>Delete</button>
            </li>
          ))}
          {articles.length === 0 && <li>No articles yet.</li>}
        </ul>
      </section>

      <section>
        <h2>Manage Universities</h2>
        <h3>Add New University</h3>
        <form onSubmit={handlePostUniversity}>
          <input name="name" placeholder="University name" required /><br/>
          <input name="country" placeholder="Country" required /><br/>
          <input name="website" placeholder="Website URL" /><br/>
          <input name="image" placeholder="Logo/Image URL" /><br/>
          <button type="submit" disabled={loading}>Add University</button>
        </form>

        <h3>Partner Universities</h3>
        <ul>
          {universities.map(uni => (
            <li key={uni.id}>
              {uni.name}
              <button onClick={() => handleDeleteUniversity(uni.id)} disabled=
{loading}>Delete</button>
            </li>
          ))}
          {universities.length === 0 && <li>No partner universities yet.</li>}
        </ul>
      </section>
    </div>
  );
}
```

That’s a lot, but essentially we have four handlers for four operations, and corresponding form or list UI. To summarize:

- We maintain state arrays `articles` and `universities` to reflect current list (starting from server-provided data).
- On adding, we call the respective API, and if successful, update the list state with the new item (taking the returned data from API which includes the created document).
- On deleting, we call API and filter out the removed item from state.
- We show a message (`msg`) for feedback on operations, and a `loading` state to disable buttons while an operation is in progress (to avoid double submission).

The forms are simple. We assumed minimal fields (article: title, content, image; university: name, country, website, image). You can expand if needed (e.g., add an "author" field for article if multiple authors).

Also, note that the `content` for article is a textarea. It could be a long text or even HTML. For now, plain text is fine. If you want richer text, you might consider using a WYSIWYG editor or instruct admin to input basic HTML. But that's an enhancement. Plain text content is acceptable (later you could parse line breaks to `<p>` tags on render if needed).

Styling the Admin page: Since this is internal, we might not need fancy styling, but it should still be neat. Perhaps structure the sections side by side if wide screen, or stacked. The design was likely simple forms. The Figma admin page ("cms - post article" etc.) shows a very minimal design with basic fields and a Post button. We've essentially implemented that. You can apply CSS to `.admin-dashboard section` to add spacing, to forms and lists for clarity.

The admin CMS interface design shows forms for posting new articles/universities and lists for deleting them. For example, the "Post Article" panel (left) contains fields for title, content, and an image upload, with a Post button. The "Delete Article" panel (second from right) lists existing article titles each with a delete (trash can) button. Similarly, "Post University" (second panel) has fields like country, name, etc., and "Delete University" (far right) lists universities with delete buttons. We will mirror this functionality in our admin dashboard implementation.

3. Creating API routes for articles and universities: We referenced `/api/articles` (POST) and `/api/articles/[id]` (DELETE), similarly for universities. We need to implement those endpoints:

- `pages/api/articles.js` (to handle creating an article, maybe also listing if needed):

```
import { getServerSession } from "next-auth/next";
import { authOptions } from "./auth/[...nextauth]";  // need to import our NextAuth config
import dbConnect from "../../lib/dbConnect";
import Article from "../../models/Article";

export default async function handler(req, res) {
  // Only allow POST (for creating new article)
  if (req.method !== "POST") {
    return res.status(405).end();
  }
  // Authenticate the request
  const session = await getServerSession(req, res, authOptions);
  if (!session || !session.user.isAdmin) {
    return res.status(401).json({ message: "Not authorized" });
```

```
  }
  // Get data from request body
  const { title, content, image } = req.body;
  if (!title || !content) {
    return res.status(400).json({ message: "Title and content are required" });
  }
  try {
    await dbConnect();
    const newArticle = await Article.create({ title, content, image });
    return res.status(201).json({ message: "Article created", article: newArticle });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ message: "Error creating article" });
  }
}
```

- This ensures only an admin (session with isAdmin) can create an article. It then inserts a new Article document. We return the `newArticle` in the response so the frontend can use its `_id` and other fields (title, etc.) immediately. We imported `getServerSession` and `authOptions` to validate the session on the server. We need to export `authOptions` from our NextAuth config. E.g., modify `...nextauth.js` to also export the config:

```
// at bottom of [...nextauth].js
export const authOptions = { ... }  // the object passed to NextAuth
export default NextAuth(authOptions);
```

- Then in API routes we can import `{ authOptions }` easily.
- `pages/api/articles/[id].js` (for deleting an article by ID):

```
import { getServerSession } from "next-auth/next";
import { authOptions } from "../auth/[...nextauth]";
import dbConnect from "../../../lib/dbConnect";
import Article from "../../../models/Article";

export default async function handler(req, res) {
  if (req.method !== "DELETE") {
    return res.status(405).end();
  }
  const session = await getServerSession(req, res, authOptions);
  if (!session || !session.user.isAdmin) {
    return res.status(401).json({ message: "Not authorized" });
  }
  try {
    await dbConnect();
    const { id } = req.query;
    await Article.findByIdAndDelete(id);
```

```
    return res.status(200).json({ message: "Article deleted" });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ message: "Error deleting article" });
  }
}
```

- This looks up the `id` from query and deletes that document. If the ID is invalid or not found, `findByIdAndDelete` will just return null, which is fine for our case (we respond 200 anyway). You might choose to handle not found specifically, but not critical.
- Similarly, `pages/api/universities.js` for POST and `pages/api/universities/[id].js` for DELETE:
  - For `universities.js`:

```
import { getServerSession } from "next-auth/next";
import { authOptions } from "./auth/[...nextauth]";
import dbConnect from "../../lib/dbConnect";
import University from "../../models/University";

export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).end();
  }
  const session = await getServerSession(req, res, authOptions);
  if (!session || !session.user.isAdmin) {
    return res.status(401).json({ message: "Not authorized" });
  }
  const { name, country, website, image } = req.body;
  if (!name || !country) {
    return res.status(400).json({ message: "Name and country are required" });
  }
  try {
    await dbConnect();
    const newUni = await University.create({ name, country, website, image });
    return res.status(201).json({ message: "University added", university: newUni });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ message: "Error adding university" });
  }
}
```

  - For `[id].js` under universities:

```
import { getServerSession } from "next-auth/next";
import { authOptions } from "../auth/[...nextauth]";
import dbConnect from "../../../lib/dbConnect";
import University from "../../../models/University";
```

```
export default async function handler(req, res) {
  if (req.method !== "DELETE") {
    return res.status(405).end();
  }
  const session = await getServerSession(req, res, authOptions);
  if (!session || !session.user.isAdmin) {
    return res.status(401).json({ message: "Not authorized" });
  }
  try {
    await dbConnect();
    const { id } = req.query;
    await University.findByIdAndDelete(id);
    return res.status(200).json({ message: "University deleted" });
  } catch (err) {
    console.error(err);
    return res.status(500).json({ message: "Error deleting university" });
  }
}
```

All these routes enforce that the request is coming from an authenticated admin session. Because we call these via `fetch` on the client side (which will send the cookie with session token automatically), `getServerSession` will find the session. If not (somehow someone without session calls it), it returns 401.

Now, our admin page's forms and buttons should trigger these routes appropriately:

- Add Article form -> POST /api/articles -> creates article in DB.
- Delete button for article -> DELETE /api/articles/[id] -> removes from DB.
- Same for universities.

Test the admin functionality:

- Log in as admin, go to `/admin`.
- Try adding an article (fill in title and content). After submission, it should show "Article added!" message and the article appears in the list below with a Delete button. Also check the database (if you have MongoDB Compass or Atlas UI open) to see the new entry.
- Try deleting an article using the button. It should remove from list and DB.
- Add a university (name & country at least). Check it appears in list and in DB.
- Delete a university, ensure removal.

If anything doesn't work:

- Open browser dev console network to see if the fetch requests got correct responses or errors.
- Check the Next.js server logs for error messages from our try/catch.
- Common issues: forgot to export authOptions, or not using getServerSession correctly, or missing dbConnect etc. Ensure the import paths are correct (notice in `authOptions` import, our path might differ since it's in a

dynamic folder; we used `../auth/[...nextauth]` which should be correct for both articles and universities folder structures).

- Also ensure NextAuth session cookie is working. If `getServerSession` isn't finding the session, you might need to pass `req, res` correctly. We did that by using `getServerSession(req, res, authOptions)` in API route. That should work as shown in NextAuth docs for securing API routes.

At this point, you have a fully functional admin dashboard. The admin experience is not super fancy (no modals or rich text editor), but it covers the requirements: an admin can add blog articles, which will show up on the Blog page for all users, and add partner universities, shown on Partners page; and remove them if needed.

Let's ensure we hook up the front-end pages to use the database now:

- Update **Blog listing page** to fetch from DB instead of dummy:
  We can now use `getServerSideProps` in `pages/blog/index.js` similar to how we did in admin (but without needing session). For example:

```
import dbConnect from "../../lib/dbConnect";
import Article from "../../models/Article";

export async function getServerSideProps() {
  await dbConnect();
  const articles = await Article.find({}, "title summary image");
  const articlesList = articles.map(doc => {
    const article = doc.toObject();
    article._id = article._id.toString();
    return article;
  });
  return { props: { articles: articlesList } };
}

export default function BlogPage({ articles }) {
  // render similar to before but using articles prop
}
```

- Now the blog page will always show whatever articles are in the DB. (You might sort them by date if you want newest first: `.find({}).sort({ createdAt: -1 })`.)
- Update **Article page** to fetch one article:
  We sketched this earlier. Implement in `pages/blog/[id].js`:

```
import dbConnect from "../../lib/dbConnect";
import Article from "../../models/Article";

export async function getServerSideProps({ params }) {
  await dbConnect();
  try {
    const article = await Article.findById(params.id);
    if (!article) {
```

```
      return { notFound: true };
    }
    const articleObj = article.toObject();
    articleObj._id = articleObj._id.toString();
    return { props: { article: articleObj } };
  } catch {
    return { notFound: true };
  }
}

export default function ArticlePage({ article }) {
  return (
    <div className="article-page">
      <h1>{article.title}</h1>
      {article.image && <img src={article.image} alt={article.title} />}
      <p>{article.content}</p>
    </div>
  );
}
```

- We handle not found (in case someone uses a bad URL or an article was deleted). Now the article page will show the full content. Note: if `content` is very long or has line breaks, you might want to render it with `<div dangerouslySetInnerHTML={{__html: content}}>` if it contains HTML. But if just plaintext, the `<p>` is fine (line breaks won't be preserved though). Optionally, you could split by `\n` and render multiple `<p>`.
- Update **Partners page** to fetch all universities:
  In `pages/partners.js`:

```
import dbConnect from "../lib/dbConnect";
import University from "../models/University";

export async function getServerSideProps() {
  await dbConnect();
  const unis = await University.find({});
  const uniList = unis.map(doc => {
    const uni = doc.toObject();
    uni._id = uni._id.toString();
    return uni;
  });
  return { props: { universities: uniList } };
}

export default function PartnersPage({ universities }) {
  // use universities prop instead of sample array
  // e.g., universities.map(uni => ...)
}
```

- Then render similarly, making sure to use the actual fields (name, country, etc.). E.g., in the card:

```
{universities.map(uni => (
  <div key={uni._id} className="partner-card">
    {uni.image && <img src={uni.image} alt={uni.name} />}
    <h3>{uni.name}</h3>
    <p>{uni.country}{uni.city ? `, ${uni.city}` : ""}</p>
    {uni.website && <a href={uni.website} target="_blank">Visit site</a>}
  </div>
))}
```

- If you want to keep the logos small, you might add styling or set a max-width in CSS.

Now the blog and partners pages are fully dynamic. Try using the admin dashboard to add an article and a university, and then see them appear immediately on the Blog and Partners pages (since SSR will pull the new data on refresh).

We've essentially covered all main functionality. The final step is to **deploy** the site so it's live and accessible.

Deploying the Site on Vercel (and Environment Setup)

Deploying a Next.js app is easiest using **Vercel**, as it's the platform made by the creators of Next.js. Here's how to deploy your BELCO Alliance website:

1. **Push your code to GitHub (or GitLab/Bitbucket):** If you haven't already, create a repository and push the project code. Ensure that your `.env.local` is not committed. On Vercel, you'll set those environment variables separately.
2. **Create a Vercel account:** Go to vercel.com and sign up (you can log in with GitHub).
3. **Import project:** Once in Vercel's dashboard, choose "Add New..." -> Project. Import from your GitHub repository (you might need to install the Vercel GitHub app and authorize access to your repo).
4. **Configure build settings:** Vercel will auto-detect it's a Next.js project. Make sure the root is correct (the repository root if that's where the Next.js project is). No need to change build command (`npm run build`) or output (default `.next`).
5. **Set Environment Variables on Vercel:** In the project settings on Vercel, find the Environment Variables section. Add the same variables from your `.env.local`:
   - `MONGODB_URI` – with the connection string (make sure to update any special characters appropriately; Vercel will handle it, just copy-paste).
   - `NEXTAUTH_SECRET` – use the same secret or generate a new secure one.
   - Also add `NEXTAUTH_URL` – set this to your deployment URL (e.g., `https://your-site.vercel.app`). NextAuth uses this to generate correct callback URLs. In production, it's recommended to set it. (In development NextAuth uses default localhost).
6. If you have others (like API keys for email or such), add them too. For our use-case, just these.
7. Set these for **Production** environment (and Preview if you want to test on preview deploys).

8. **Deploy:** Click "Deploy". Vercel will build the project (install packages, run `npm run build`, then host it). On first deployment, it will also assign a domain like `your-project-name.vercel.app`. After deployment, test the site at that URL:
    - The pages should load.
    - Test the database functionality: You might need to update your MongoDB Atlas IP access list to allow Vercel's servers. Atlas can allow access from anywhere (0.0.0.0/0) which is simplest for a demo project, or you can find the IPs Vercel uses (they can vary). Easiest: in Atlas, add an IP whitelist of `0.0.0.0/0` (but remember this is less secure long-term, however your DB has username/password so it's fine for development).
    - Try registering a user on the live site, logging in, etc. Ensure everything works in production as it did locally.
9. **Set up a custom domain (optional):** If this is for a project demo, the Vercel URL is fine. If you have a custom domain (like belcoalliance.com), you can add it in Vercel and point DNS to it.

Deployment debugging: If something doesn't work on Vercel:

- Check the Deployment logs in Vercel for errors.
- A common issue is forgetting to set env vars, so NextAuth or DB connection fails. Ensure those are set and redeploy if needed.
- Another is build errors if your code had any references to things that don't exist, but if it ran locally, it should be fine.
- If the NextAuth callback URL is off, you might get NextAuth errors. Setting `NEXTAUTH_URL` fixes that.

Once deployed, you have a live full-stack site! 🎉

Conclusion and Next Steps

You have now built a comprehensive website for BELCO Alliance with full-stack capabilities. Let's recap what we accomplished:

- **Set up a Next.js project** and structured pages according to the sitemap (homepage, programs, blog, article, partners, account, admin).
- **Implemented a consistent layout** with a responsive Navbar and Footer on all pages.
- **Translated the Figma design** components into React components and JSX structure, matching the visual style and content sections (using the provided screenshots as a guide).
- **Established a connection to MongoDB Atlas** and defined Mongoose models for users, articles, and universities, enabling persistent data storage.
- **Developed an authentication system** using NextAuth (Credentials Provider), allowing users to register and log in, with secure password hashing and session management. We also set up role-based access (admin flag).
- **Created protected admin functionality** for the designated admin user to add new blog articles and partner universities, and delete existing ones through a simple internal interface. These actions are secured on both

frontend and backend – only the admin sees the controls, and API routes double-check the user's admin status.

- **Made the blog and partners pages dynamic**, so they display content directly from the database. When the admin adds or removes items, the changes reflect on the user-facing pages immediately on refresh.
- **Deployed the application on Vercel**, making it accessible on the web, and configured environment variables for production.

This is a significant achievement, especially given your initially limited JavaScript experience. 🎉 You've gone beyond static coursework to implement a real-world full-stack app with authentication and content management. Along the way, you likely picked up skills in React components, state handling, server-side programming with Next.js API routes, and database integration.

A few final tips / potential enhancements:

- You might want to improve the security and polish: e.g., add confirmation modals for deletions ("Are you sure?"), or implement a richer text editor for writing articles (so admin can format content).
- Implementing a proper **password reset** flow: using email to reset password, or at least an admin ability to reset a user's password, would be useful in a real scenario.
- Adding more admin features like editing an existing article or university (not just add/delete) would make the CMS more complete. This could be done by creating a similar form that loads the current data and saves changes.
- You can also expand the user profile: right now, aside from login, regular users don't have much to do (since it's mostly an informational site). If needed, you could allow users to edit their name, or have a profile page, etc. For example, if you envision a member-only area or comments on articles, the authentication system is already in place to facilitate that.
- **UI/UX refinements:** Since you have good CSS skills, you could add more responsiveness (make sure the site looks good on mobile devices), and use the exact colors/spacing from the Figma for a pixel-perfect result. Little things like hover effects on buttons, transitions, etc., can elevate the feel of the site.
- **Testing:** It's good to test as a non-admin user as well. For instance, ensure that if a non-admin tries to access `/admin` or the API routes directly, they are indeed blocked (we set that up, but always good to verify no loopholes).

By following this guide, you tackled the project in a logical sequence: setting up the foundation, building the UI, then adding authentication, database, and admin features. This approach helps to always have a working version of the site at each step (you could always navigate through pages even when some data was placeholder).

Finally, **stay motivated and keep learning**. You've effectively built a mini content management system and a multi-page website with modern tools – this is no small feat! If you ever get stuck, refer back to this guide and the cited resources for specific implementation details. The citations (like Next.js and NextAuth documentation) can provide deeper explanations if you want to understand the why's and how's in greater depth.

Good luck with your BELCO Alliance website project, and congratulations on pushing yourself to go beyond the basics. With the site deployed and running, your alliance and peers can now see the results of your hard work. 🚀

Decision Report: Not Using React/Next.js for Belco 3-Week Project

Balancing Ambition with Feasibility in a 3-Week Development Sprint

Introduction

This report outlines our team's decision to **not use React or Next.js** for the Belco website project, which has a **3-week development timeline**. We explain our original plan, why it proved impractical after research and teacher feedback, the challenges we faced, and how we adjusted our approach. The tone is kept clear and professional, aimed at informing our UAS teachers of our reasoning.

Original Plan and Rationale

Initially, our plan was ambitious. We intended to:

- **Use React or Next.js:** Build the client's website as a modern single-page application using React (or the Next.js framework) to leverage dynamic content and a responsive user interface.
- **Implement a Custom CMS:** Develop a custom **Content Management System** so that Belco staff could easily update site content through an admin interface (instead of hard-coding changes).
- **Include User Account System:** Incorporate an **account/login system** to manage user authentication and perhaps restrict certain content to registered users.

Rationale: We believed this stack would provide a cutting-edge solution. React/Next.js promised modular development and a rich ecosystem, which we thought could handle all requested features in one unified app. A custom CMS and account system, in theory, would meet the client's needs for content control and user interactivity. We were also excited to learn these modern technologies as part of the project.

Research and Feedback: Feasibility Issues

After some research and discussions with our teachers, we discovered significant feasibility issues with our original approach:

- **Steep Learning Curve of React/Next.js:** As second-semester students with no prior experience in these frameworks, we realized that learning React (and Next.js, which builds on React) on the fly would be challenging. Mastering React basics alone typically takes *"a few weeks to months"* of practice – time we simply did not have within a 3-week project. Next.js would add even more concepts (like routing and server-side rendering) on top of React, increasing complexity.
- **Complexity of Building a CMS:** Through research, we learned that developing a custom CMS from scratch is a major undertaking. In fact, a basic custom CMS can take on the order of *months* to build (roughly **three months** for even a simple version). This far exceeds our project timeline. The CMS would require setting up

databases, content editing interfaces, and admin controls – tasks that are unrealistic to complete (and test) in such a short span, especially with our limited experience.

- **User Account System Challenges:** Implementing secure user authentication and account management is also non-trivial. We'd need to handle password security, user roles, sessions, etc., and ensure it's all safe. Proper security testing (like penetration tests) is *"not trivial"*, and doing this in 3 weeks could jeopardize security. Our teachers advised that tackling authentication without prior backend experience could lead to serious issues or an incomplete feature.
- **Overhead of Advanced Frameworks for a Small Project:** We reconsidered whether using a heavy framework was necessary. For a relatively straightforward website, adding the overhead of React/Next.js (project setup, state management, build configuration) could slow us down instead of speeding development. External guidance and literature confirmed that while React excels for **complex, dynamic apps**, pure HTML/CSS/JS offers *"simplicity and control, perfect for smaller projects"*. In other words, our project's scale and timeline make a simpler approach more suitable.

Our instructors reinforced these findings. They noted that given our current skill set, trying to implement a full React application with custom CMS and auth in three weeks was not feasible. We heeded this feedback to avoid overcommitting to an unrealistic plan.

Key Challenges and Constraints

Several practical constraints drove us to pivot away from the original plan:

- **Time Constraint (3 Weeks):** The project's duration is very short. Building even a basic version of the CMS and account features (let alone learning a new framework) would likely take much longer than 3 weeks. We risked running out of time with half-finished features.
- **Limited Technical Experience:** At Semester 2, our team has a foundation in HTML, CSS, and some JavaScript, but **no prior exposure to React or Next.js**. Jumping into a sophisticated framework under time pressure is risky. It would require learning JSX, component structures, state management, and Next.js conventions from scratch. As research indicated, picking up React basics alone isn't instantaneous. Our lack of experience also extends to backend development – we have never built a CMS or an auth system before.
- **Steep Learning vs. Building Balance:** Given our inexperience, a large portion of the 3 weeks would be spent simply learning the technologies rather than building the actual project. This learning curve would leave insufficient time to implement and polish the client's website. We had to acknowledge that we couldn't become proficient with the required tools fast enough to deliver a quality result within the deadline.
- **Quality and Security Risks:** Attempting complex features without expertise could lead to poorly implemented or insecure solutions. For example, a rushed account system might have security vulnerabilities. We did not want to deliver a flawed product to the client. Ensuring robustness (especially for user accounts) would require extensive testing and debugging, which was not realistic under our constraints.
- **Scope Creep and Project Focus:** The original plan's scope was very broad – essentially building a full web application platform (frontend and backend) from scratch. This wide scope threatened to pull our attention in too many directions (design, frontend coding, backend logic, database, security). With limited time, we risked not doing any one aspect well. We recognized the need to narrow the scope and focus on the **core priorities** (in this case, the website's content and design) to ensure a successful delivery.

These challenges made it clear that continuing with the initial approach would likely result in an incomplete or subpar outcome. The prudent choice was to adjust our plan to something achievable and ensure we could excel in the most important areas for the client.

Revised Plan: Two Static Websites Focused on Design

After evaluating the situation, we decided to **simplify the project and play to our strengths**. The revised plan is to build **two separate websites** for the client – one for **Belco Alliance** and one for **Belco Education** – using standard HTML, CSS, and JavaScript (no React/Next.js). The emphasis will be on creating a strong visual design and clear content for each site. Key aspects of this plan:

- **Static Website Approach:** We will develop the pages as static or lightly scripted websites. This means writing well-structured HTML for content, using CSS for layout/styling (to reflect Belco's branding), and adding vanilla JavaScript for any basic interactivity or animations. By avoiding heavy frameworks, we eliminate setup overhead and can start building the site layout and visuals immediately.
- **Belco Alliance and Belco Education Sites:** The two parts of the project will be handled as two cohesive but separate sites. Each site will have its own set of pages to showcase relevant information. We will ensure a consistent design language between them (since they belong to the same client) but tailor the content and navigation to each site's focus (Alliance vs. Education). This separation allows us to manage each site's development more straightforwardly, without needing a complex multi-section single application.
- **Focus on Visual Design and UX:** With more time freed up (since we're not wrestling with unfamiliar tech), our team can concentrate on what we do well: designing an attractive, user-friendly interface. We plan to produce a modern, clean design for both sites, with responsive layouts and intuitive navigation. The goal is to give Belco a professional online presence that satisfies their immediate needs. This addresses the client's priority of having a polished website, even if some advanced features are deferred.
- **Content Presentation:** All content (text, images, static info) will be hard-coded or manually placed for now. We will ensure it's organized and easy for users to read. While this means the client cannot yet edit content through a CMS, we will work closely with them during development to get the content right. Any future updates would have to be done by editing the code or adding new pages, but in the short term, this is manageable given the tight deadline.

This revised approach is much more **in line with our current capabilities**. It allows us to deliver **functional, good-looking websites** within 3 weeks. Essentially, we are prioritizing a solid front-end product over unfinished back-end features. The client will get two live websites focusing on their branding and information, which was the core of their request, with the understanding that dynamic functionality can come later.

Postponing CMS and Account Features

One of the biggest decisions we made was to **postpone the CMS and user account system features**. Instead of trying to cram these complex systems into the initial 3-week build (and likely failing or delivering poor results), we will lay the groundwork for them to be added in the future. Here's our thinking on this:

- **Visual Foundation First:** By completing the static sites now, we create a strong foundation – all the pages, design elements, and content will be in place. This foundation can later be enhanced with dynamic features.

For example, once we have more time or experience, a CMS could be integrated to manage the existing content, or a login system could be added to specific sections. Building the visuals first means any future functionality will plug into an established UI/UX, which might simplify integration.

- **Future Implementation (When Feasible):** We conveyed to the client (and acknowledge for ourselves) that the CMS and account system are **deferred, not canceled**. In the future, possibly in a follow-up project or when our team has advanced skills (perhaps later semesters), we can explore implementing those features properly. This might involve using a proven solution or framework once we're ready, rather than a rushed custom build now. For instance, we could eventually set up a custom API or adopt a headless CMS service to connect with our static front-end when time allows.
- **Client Understanding:** We will document these features as "Phase 2" possibilities. The client, Belco, remains interested in having content manageability and user accounts, so we assure them that the current websites are built with the intention to expand later. In our delivery, we'll note which parts of the site would benefit from a CMS (e.g. news updates or course lists) and which sections might need user accounts. Having the site live and usable now is the priority; adding those enhancements can be scheduled once it's realistic.

By postponing these features, we avoid compromising the current project. This decision reflects good project management – it's better to deliver a **focused, high-quality product** that meets core needs than an over-ambitious product full of half-working features. Our teachers supported this phased approach as a sensible solution given the circumstances.

Summary of Key Reasons and Outcomes

In summary, our choice to not use React/Next.js (and to delay the CMS/account functionality) was driven by several key factors and has led to a clear plan moving forward:

- **Insufficient Time for New Technology:** With only 3 weeks available, there wasn't enough time for us to learn and implement React/Next.js along with building complex backend features. Research shows learning React basics alone can take weeks or months, and developing a custom CMS could take **months** – far beyond our timeline.
- **Limited Experience:** Our current skill set just wasn't aligned with the original plan's requirements. Attempting unfamiliar frameworks and systems under deadline pressure posed too high a risk. We prioritized a path that fits our knowledge (HTML/CSS/JS) to ensure we can deliver results.
- **Project Complexity vs. Value:** The advanced features (dynamic content editing, user logins) added a lot of complexity but were not absolutely essential for an initial web presence. We recognized that a simpler static site can still fulfill the client's core needs (displaying information beautifully) without the complexity that might derail the project.
- **Quality and Professionalism:** This decision was about maintaining quality. By narrowing the scope, we can devote time to refining the design and content. The outcome will be two polished websites that reflect well on both Belco and our team, rather than an unstable application. We are treating the client's project professionally by not using them as a guinea pig for technologies we aren't comfortable with yet.

Outcome

We will deliver the **Belco Alliance** and **Belco Education** websites as static, well-designed sites within the 3-week deadline. The sites will have the agreed-upon content and a visually appealing, user-friendly interface. All critical information will be accessible to users, meeting the client's immediate requirements. The more advanced features (CMS for content updates and user account system) are documented as future enhancements. Once we have the necessary experience or additional time, we can implement those in a Phase 2, ensuring they are done correctly.

By making this choice, our team learned an important lesson in project scoping and feasibility. We believe this approach best balances the client's needs with our capabilities and timeline, resulting in a successful project delivery. We appreciate the guidance from our teachers in reaching this decision, and we are confident that focusing on a strong visual foundation now will set the stage for any needed functionality later.

How to Use Trello Board

A simple guide to help us stay organized and focused without pressure

🔁 *Scroll down for full Bulgarian translation / Плъзни надолу за пълен превод на български*

Link: https://trello.com/b/PlY90m8z

⚠️ Important Note About Deadlines

You are responsible for setting your own due dates. I will not be enforcing any deadlines. What matters is that the task gets done, not when. Please set realistic dates to stay on track.

🔑 Step 1: Join the Board

- Click the link above
- Log in or sign up to Trello
- Request access if needed

📚 Step 2: Understand the Layout

- **Lists** = columns (e.g., To Do, In Progress, Done)
- **Cards** = tasks (each task is a card inside a list)

✅ Step 3: Work with Tasks (Cards)

- Click a card to open it
- Read the description
- Do the task
- Move the card to the next list when done

✏ What You Can Do in a Card

- **Comment**: say what you did or ask questions
- **Checklists**: tick off smaller tasks
- **Add yourself**: click “Members”
- **Due date**: set your own realistic deadline

🔔 Tips

- **Watch** cards to get notified about changes
- **Keep cards updated**

- Keep it simple and clean 💡

If you're not sure about something, just comment on the card! 🙌

🇧🇬 Как да използваш нашия Trello борд

Линк: https://trello.com/b/PIY90m8z
Подзаглавие: *Просто ръководство, за да останем организирани и фокусирани без напрежение*

⚠️ Важно относно сроковете

Ти сам си определяш крайните срокове по задачите си. Аз няма да ги следя или налагам. Важно е задачите да се свършат – не кога точно. Моля, слагай реалистични срокове, за да си следиш напредъка.

🔑 Стъпка 1: Влез в борда

- Натисни линка горе
- Влез или се регистрирай в Trello
- Ако трябва – поискай достъп

📚 Стъпка 2: Разбери структурата

- **Списъци** = колони (напр. To Do, In Progress, Done)
- **Карти** = задачи (всяка задача е отделна карта)

✅ Стъпка 3: Работа със задачите (картите)

- Натисни карта, за да я отвориш
- Прочети описанието
- Свърши задачата
- Премести картата в следващата колона, когато е готова

✏️ Какво можеш да правиш в карта

- **Коментар**: пиши какво си направил или попитай нещо
- **Чеклист**: отбелязвай малките подзадачи
- **Добави себе си**: чрез бутона “Members”
- **Краен срок**: сложи си реалистична дата за изпълнение

🔔 Съвети

- **“Watch”** – така ще получаваш известия
- **Обновявай картите** ако нещо се промени
- Дръж нещата прости и чисти 💡

Heuristic evaluation form

1= cosmetic problem, 2= minor problem, 3 = major problem, 4= catastrophe

Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Table 1

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**
1	The website currently lacks feedback when a post is added, leaving users unsure if the action was successful.	3
2	The website currently lacks feedback when a message is sent, leaving users unsure if the action was successful.	**3**

Match between system and the real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

Table 2

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**

1	The contact Info form is presented like a business card in a physical form	0

User control and freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Table 3

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**
	The website currently lacks an option to undo or cancel actions like post submissions. If a user adds a post by mistake, there is no clear way to reverse it. Providing options like a "Cancel" button during creation or an "Undo" after submission would enhance control and prevent user frustration.	**2**
	There is no confirmation prompt or way to recover what was written. This lack of control can lead to frustration, especially if the form was lengthy. Implementing a "Are you sure you want to leave?" prompt or autosave feature would give users more freedom to recover from unintended navigation.	4

Consistency and standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Table 4

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**

Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Table 5

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**
1	The website currently lacks an option to undo or cancel actions like post submissions. If a user adds a post by mistake, there is no clear way to reverse it. Providing options like a "Cancel" button during creation or an "Undo" after submission would enhance control and prevent user frustration.	2
1	There is no confirmation prompt or way to recover what was written. This lack of control can lead to frustration, especially if the form was lengthy. Implementing a "Are you sure you want to leave?" prompt or autosave feature would give users more freedom to recover from unintended navigation.	4

Recognition rather than recall

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be

visible or easily retrievable whenever appropriate.

Table 6

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**
1	**We constantly remind users of a “How it works?” step-by-step process, when applying, searching or interacting with our student programs. A specific guide is at the top of all pages, in which it is applicable.**	**0**

Flexibility and efficiency of use

Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Table 7

Column 1	Column 2	Column 3

Nr	Description	Ernst
1	For our Application Form, next to some input boxes, there is an additional explanation, where applicable, to avoid confusion and aid user experience.	**0**

Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Table 8

Column 1	Column 2	Column 3
Nr	**Description**	**Ernst**
1	**Hero - homepage; Simple shapes with easy for the eyes colors, that provide aesthetic, without confusing our user.**	**0**
2	**Easy and well explained application form.**	**0**

Help users recognize, diagnose, and recover from errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Table 9

Column 1	Column 2	Column 3
Nr	**Omschrijving**	**Ernst**

Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Table 10

Column 1	Column 2	Column 3
Nr	**Omschrijving**	**Ernst**
1	For some of the terminology within the site some extra information is needed eg. Slug, iso things	3