Architectural Decision-Making Log

Project Group: *11*
Course: *IV1201 - Design of Global Applications*
Date: March 16, 2025

Contents

1 Decision-Making Process

Date: January 23, 2025

Decision

1. **Propose:** One team member proposes an initial idea or approach.
2. **Consult:** Another team member reviews the proposal, providing feedback and potential improvements.
3. **Finalize:** The refined proposal is presented to the full team for discussion and final approval. Decision gets documented in this document.

Rationale

- Proposing an initial approach ensures a structured starting point.
- Consulting a team member helps refine the idea with additional perspectives.
- Finalizing the proposal with the entire team ensures alignment and buy-in.

Alternatives Considered

- **Making an immediate decision:** Rejected due to the risk of overlooking critical input.
- **Full team discussion from the start:** Rejected to avoid inefficiencies and delays in reaching a conclusion.

Implications

- Ensures a balanced approach that incorporates diverse insights.
- May require adjustments based on feedback at each stage.
- Helps maintain efficiency while still allowing for collaborative decision-making.

2 Application Architecture

Date: January 25, 2025

Decision

Start with a **Monolithic Architecture** using **Client-Side Rendering (CSR)**.

Rationale

- Simpler to develop and deploy initially.
- Easier for team members to collaborate on a single codebase.
- CSR offers better user experience with faster interactions.

Alternatives Considered

- **Microservices Architecture:** Rejected due to increased complexity and team's limited experience with service orchestration.
- **Server-Side Rendering (SSR):** Rejected in favor of CSR for a more dynamic and responsive UI.

Implications

- Potential need to refactor into microservices in the future as the application scales.
- Modularization will make future transitions easier.

3 Version Control System

Date: January 25, 2025

Decision

Adopt **Git** as the version control system for the project.

Rationale

- Widely used and industry-standard for version control.
- Distributed nature ensures every team member has a full backup of the repository.
- Powerful branching and merging capabilities facilitate collaborative development.
- Active community support and integrations with popular platforms (e.g., GitHub, GitLab).

Alternatives Considered

- **SVN (Subversion):** Rejected due to its centralized architecture, which lacks the flexibility and resilience of Git.

Implications

- Team members will need training on Git workflows (e.g., branching, merging, resolving conflicts).
- Establishing a clear branching strategy (e.g., Git Flow or trunk-based development) will be essential for smooth collaboration.
- Potential learning curve for team members unfamiliar with Git commands.

4 Frontend and Backend Frameworks

Date: January 25, 2025

Decision

Adopt **React** for the frontend and **Express** for the backend.

Rationale

- **React:**
  - Component-based architecture makes UI development modular and reusable.
  - Widely adopted in the industry with strong community support and a rich ecosystem.
  - Flexibility to implement complex and dynamic user interfaces.
- **Express:**
  - Minimal and unopinionated framework that is simple to set up and extend.
  - Well-suited for creating RESTful APIs and integrating with various databases.
  - Lightweight and performant, allowing rapid backend development.
- Both technologies have a large talent pool, making it easier to onboard new developers.

Alternatives Considered

- **Angular (Frontend):** Rejected due to steeper learning curve and team preference for React's flexibility.
- **Vue.js (Frontend):** Rejected because of less widespread enterprise adoption compared to React.
- **Django (Backend):** Rejected in favor of Express for its lightweight and JavaScript-based ecosystem, allowing shared language between frontend and backend.
- **Spring Boot (Backend):** Rejected due to higher complexity and a need for more specialized skills.

Implications

- Need to establish best practices for state management in React (e.g., using Redux, Context API, or others).
- Backend team will focus on building RESTful APIs with Express, with possible future integration of middleware for authentication and data validation.
- Training and documentation will be necessary to ensure efficient use of both frameworks.
- Seamless communication between frontend and backend will require setting clear API contracts.

5 Authentication Mechanism

Date: January 25, 2025

Decision

Use **JWT (JSON Web Tokens)** for authentication and session management.

Rationale

- Stateless authentication mechanism, reducing the need for server-side session storage.
- Compact and self-contained tokens that can securely transmit user information and claims.
- Industry-standard solution for authentication and authorization.
- Easy integration with both frontend (React) and backend (Express) frameworks.
- Tokens can be easily verified on the backend without querying a database for each request, improving performance.

Alternatives Considered

- **Session-Based Authentication:** Rejected due to the overhead of managing session storage on the server and less scalability in distributed systems.
- **OAuth 2.0 with OpenID Connect:** Rejected for now due to increased complexity, which may not be necessary for the initial phase of the project.
- **API Keys:** Rejected as they lack user-specific authentication and are better suited for service-to-service communication.

Implications

- Backend must securely sign and verify tokens using a secret or a public/private key pair.
- Tokens must be securely transmitted (e.g., via HTTPS) and stored on the client side (e.g., HTTP-only cookies or localStorage).
- Token expiration and refresh mechanisms will need to be implemented to maintain secure and seamless user sessions.
- Additional considerations will be required to manage token invalidation in cases of logout or compromised tokens.

6 Primary Communication Tool

Date: January 25, 2025

Decision

Adopt **Discord** as the primary communication and collaboration platform for the team.

Rationale

- Provides real-time messaging, voice, and video capabilities, fostering effective team collaboration.
- Channels allow for organized communication by topic or project area.
- Widely accessible across devices (desktop, mobile, and web), ensuring team members can stay connected.

Alternatives Considered

- **Slack:** Rejected due to limitations in the free plan (e.g., restricted message history) and lower adoption by the team compared to Discord.
- **Microsoft Teams:** Rejected because of its heavier, more formal interface and lack of team familiarity.

Implications

- All team members will need to set up and join the designated Discord server.
- Clear communication guidelines (e.g., naming conventions for channels, etiquette) will be established to maintain professionalism.

7 Email-based Password Reset Mechanism

Date: February 19, 2025

Decision

Adopt an **email-based password reset mechanism** by sending a URL containing a temporary JWT token for resetting user passwords.

Rationale

- **Security:**
  - A JWT (JSON Web Token) is used to securely transmit information between the backend and the frontend, ensuring that the password reset process is protected from unauthorized access.
  - The temporary token can be configured with an expiration time to prevent misuse.
- **User Experience:**
  - The email-based reset provides a familiar and straightforward way for users to reset their passwords without needing to remember existing credentials.
  - Including a direct URL with a temporary token reduces the steps required for users to complete the reset process.
- **Scalability:**
  - This mechanism is easy to implement at scale across a variety of applications, as it leverages common standards for user authentication.
  - JWTs can be handled efficiently and are lightweight, minimizing backend overhead during the reset process.
- **Email Reliability:**
  - Most users have an email account, making this an accessible and reliable method for initiating the reset process.
  - By integrating with existing email services, we avoid the need for complex phone-based authentication or other third-party systems.

Alternatives Considered

- **SMS-based reset:** Rejected due to reliance on third-party services (e.g., Twilio), higher costs, and potential security concerns (e.g., SIM swapping).
- **Security Questions:** Rejected due to lower security, as answers to common security questions can be easily guessed or found online.
- **Biometric Authentication:** Rejected as an alternative because it requires additional hardware and may be less accessible to all users.

Implications

- Need to integrate JWT token creation and validation in the backend, ensuring proper expiration management and secure storage.
- Email system integration will require careful handling of sensitive data, including ensuring that the reset URL is sent only to the correct user.
- Frontend will need to provide a user-friendly interface for entering the new password once the user clicks the reset link.
- Token expiration and validation logic will need to be thoroughly tested to avoid vulnerabilities.
- Possible future enhancements include rate limiting password reset requests to prevent abuse.

8 Git branch for each function

Date: February 11, 2025

Decision

Maintain a dedicated Git branch for each individual feature or function developed within the project.

Rationale

- **Isolation of Features:** Each feature branch allows developers to work independently without affecting the main codebase, minimizing conflicts and disruptions.
- **Clear Version History:** Maintaining separate branches ensures a well-documented history of changes, making it easier to identify and revert problematic code if necessary.
- **Simplified Code Review:** Pull requests from feature branches facilitate focused code reviews, improving the quality and maintainability of the codebase.
- **Parallel Development:** Multiple developers can work on different features simultaneously, accelerating project progress.

Alternatives Considered

- **Single Branch Development:** Rejected due to the risk of overlapping changes and difficulty in managing conflicts, especially in collaborative projects.
- **Minimal Branching (Only Major Features):** Rejected as it limits flexibility and makes it harder to track incremental progress on smaller features.

Implications

- Team members need to follow best practices for branching, committing, and merging code.
- Clear naming conventions (e.g., feature/feature-name) will be established for consistency.
- Merge conflicts must be resolved before integrating branches into the main branch.
- Regular merging of the main branch into feature branches will help keep code up to date.

9 Create a documentation for project planning

Date: February 11, 2025

Decision

Create a Google Document to maintain comprehensive project planning documentation to guide the project's progress, ensure alignment among team members, and support efficient project management.

Rationale

- **Clear Objectives and Milestones:** Documenting project goals, deliverables, and timelines ensures that all team members understand their responsibilities and deadlines.
- **Improved Coordination:** Centralized documentation promotes collaboration, minimizes miscommunication, and aligns team efforts.
- **Progress Tracking:** Regularly updated documentation helps monitor progress, identify delays, and adjust plans as needed.

Alternatives Considered

No alternative considered, Google document suits all.

Implications

- Project documentation will be maintained using collaborative tools such as Google Docs or Notion.
- The documentation will cover key areas such as project scope, roles and responsibilities, task assignments, deadlines, and risk management.
- Regular updates will ensure the documentation remains accurate and reflects the latest project developments.
- Access permissions will be managed to ensure data security while enabling team collaboration.

10 Commenting Every Exported Function

Date: March 6, 2025

Decision

Adopt a standard practice of adding comprehensive comments to every exported function in the codebase. This includes detailed descriptions of the function's purpose, parameters, return values, and any important logic within the function.

Rationale

- **Code Readability:**
  - Commenting each exported function ensures that the code is self-documenting and easier to understand for developers who may not be familiar with the codebase.
  - Well-documented functions help in faster code reviews, debugging, and understanding the flow of logic within the system.
- **Maintainability:**
  - As the project grows, the codebase becomes more complex, and a comment-based system ensures that each function can be easily identified and understood by future developers.
  - Comments help in preventing issues with functionality as future modifications can be done with a clear understanding of the function's expected behavior.
- **Collaboration:**
  - This approach promotes collaboration among developers by providing clear context and explanations for each function. This is particularly useful when working in teams with different experience levels.
  - Helps in avoiding misunderstandings during code handovers, ensuring that new developers can easily work with existing functions.
- **Best Practices:**
  - Commenting functions aligns with industry best practices for writing clean, maintainable, and readable code.
  - It ensures that the codebase adheres to standards expected by most development teams and fosters better long-term project health.
- **Debugging and Testing:**
  - Adding detailed comments allows for easier debugging and error tracking, as developers can more quickly understand the purpose and behavior of each function when issues arise.
  - Tests can be better written and understood when the expected behavior of functions is clearly stated in the comments.

Alternatives Considered

- **No Comments:** Rejected due to the lack of clarity in the codebase, leading to slower development, harder debugging, and a higher chance of introducing errors.
- **Minimal Comments:** Rejected as insufficient comments would still leave too much ambiguity in understanding the function logic, especially for complex or non-trivial implementations.
- **External Documentation:** Rejected as it would require keeping documentation separate from the code, leading to potential inconsistencies and extra overhead in maintaining both the code and documentation.

Implications

- Developers will need to ensure that comments are added when implementing new exported functions and when modifying existing functions.
- The comments should be consistently formatted and should include descriptions of the function's parameters, return values, exceptions thrown, and any key logic used.
- A process will need to be established to ensure that functions are properly commented as part of the code review process.
- A potential challenge is ensuring that comments remain up-to-date, as out-of-date comments could become misleading. Developers will need to be diligent about keeping comments aligned with code changes.
- The project may require tooling or linting setups to enforce this commenting standard, ensuring consistency across the codebase.

11 Use of Heroku as a Cloud Platform and Postgres as an Add-on for Backend Application

Date: March 1, 2025

Decision

Adopt Heroku as the cloud platform and Postgres as an add-on for the backend application.

Rationale

- **Ease of Use and Rapid Deployment:**
  - Heroku simplifies deployment and scaling, allowing developers to focus on building the application rather than managing infrastructure.
  - Its support for multiple programming languages (e.g., Python, Node.js) and frameworks accelerates development.
  - Pre-configured PostgreSQL add-ons reduce the need for manual database setup, making it quick to deploy.
- **Scalability:**
  - Heroku's infrastructure can scale up or down based on demand, which is ideal for dynamic applications with varying workloads.
  - Provides flexibility for scaling the database through the Postgres add-on, which can handle high transaction volumes as the application grows.
- **Integration and Ecosystem:**
  - Heroku offers a wide array of third-party add-ons (like Postgres) that integrate seamlessly with the platform.
  - Built-in support for CI/CD pipelines, monitoring, and error tracking streamlines development workflows.
- **Cost-Effectiveness:**
  - Heroku offers a variety of pricing tiers, starting with a free tier that can help teams prototype or run small applications at minimal cost.
  - Postgres on Heroku is offered as a fully-managed service, reducing overhead costs associated with database management.
- **Security and Compliance:**
  - Heroku provides a secure environment for data storage and application deployment with SSL encryption, regular backups, and compliance certifications.
  - PostgreSQL on Heroku is fully managed, including automated backups and updates, ensuring high availability and disaster recovery.

Routine

- Set up a Heroku account and deploy the backend application to Heroku.
- Add Postgres as an add-on to Heroku and configure the database for use with the backend application.
- Implement necessary environment variables and manage configurations through Heroku's dashboard or CLI.
- Use Heroku's monitoring tools to ensure smooth operation and performance of the application.
- Regularly back up the database to safeguard against data loss.
- Test the application for performance and scalability, adjusting resource allocation as needed.

Alternatives Considered

- **AWS or Google Cloud:** Rejected due to higher complexity in setting up and managing cloud infrastructure.
- **Self-Hosting:** Rejected because it would require significant time and effort to manage both infrastructure and databases, increasing overhead.
- **DigitalOcean:** Rejected as Heroku's ease of use, native integrations, and managed services outweigh DigitalOcean's more manual configuration process.

Implications

- Developers must familiarize themselves with the Heroku platform and how to integrate Postgres as an add-on.
- Deployment, scaling, and monitoring of the application will be managed on Heroku, reducing administrative overhead.
- Ongoing management of the database, including security patches, backups, and scaling, will be handled by Heroku's managed Postgres service.
- Future developers may need to adhere to best practices for application deployment on Heroku to ensure smooth operation.

Decision

Adopt a standardized routine to ensure that all web pages function correctly across Edge, Firefox, Chrome, and Safari. This routine includes manual testing methods to verify compatibility.

Rationale

- **Consistency Across Browsers:**
  - Ensures that users experience a uniform and fully functional interface regardless of their browser choice.
  - Prevents browser-specific rendering or functionality issues.
- **Improved Debugging and Maintenance:**
  - Identifies browser-specific bugs in development, reducing post-deployment issues.
  - Maintains long-term project health by ensuring continued compatibility as browsers update.
- **Industry Best Practices:**
  - Ensures compliance with accessibility and performance expectations across browsers.

12 Routine for broweser compatibility

Routine

- Check that all functionality that applicants may use work as intended.
- Sign up, Login, add competences, apply for job and reset password.
- Error Monitoring: Capture and document browser-specific issues.
- Do this check before group presentation.
- www.browserling.com can be used if a member does not have access to a specific web browser.

Alternatives Considered

- No Routine: Rejected due to it is a mandatory task.
- Automated Testing: Rejected since some UI/UX issues seems to require manual review. Also, why make it harder than it needs to be?

Implications

- Developers must integrate cross-browser testing into their workflow.
- QA processes will need to include browser verification before releases.
- Continuous updates may be required to accommodate browser updates and new web standards.

13 Error Handling Strategy

Febrauri 25, 2025

Decision

Implement a comprehensive error handling strategy using try-catch blocks for synchronous code and Error Boundary Component in the frontend (React), along with error middleware in the backend (Express). This ensures that errors are caught, logged, and handled gracefully across the application.

Rationale

- User Experience: Users receive clear, friendly error messages instead of unhandled exceptions or cryptic codes.
- Debugging and Maintenance: Proper error handling and logging help developers quickly diagnose and fix issues, improving stability.
- Security: Sensitive error details (e.g., stack traces) are not exposed to users, preventing vulnerabilities.
- Consistency: A unified error handling strategy ensures consistent error management, making the codebase easier to maintain

Implications

- Consistency: Errors are handled uniformly, improving maintainability.
- User Experience: Users get meaningful error messages, reducing frustration.
- Debugging: Errors are logged and tracked, simplifying issue diagnosis.
- Security: Sensitive error details are hidden, reducing security risks.

Alternatives Considered

None. This approach was deemed essential for user experience, debugging, security, and consistency.

14 Frontend and Backend testing

March 11, 2025

Decision:

Conduct a study on how to implement unit testing in both the frontend (React) and backend (Express) to ensure code reliability and maintainability.

Rationale:

- Code Quality: Unit testing ensures that individual components and functions work as expected, reducing the likelihood of bugs in production.
- Maintainability: Tests act as documentation and make it easier to refactor or extend the codebase without introducing regressions.
- Debugging: Unit tests help identify issues early in the development process, reducing debugging time.
- Higher Grade Requirement: Implementing unit testing is a requirement for achieving higher grades, as outlined in the Tasks Affecting Final Grade document

Implications:

- Development Time: Writing unit tests requires additional time during development but pays off in reduced debugging and maintenance efforts.
- Code Coverage: Aim for high code coverage to ensure most of the codebase is tested.
- Continuous Integration: Integrate unit tests into the CI/CD pipeline to run tests automatically on every code change.

Alternatives Considered:

- No Unit Testing: Rejected because it would lead to lower code quality and make debugging and maintenance more difficult.
- Manual Testing: Rejected because it is time-consuming and less reliable than automated unit tests.

15 Higher Grade Tasks

Date: January 25, 2025

Decision

Focus on implementing higher-grade tasks to achieve a better grade, as outlined in the Tasks Affecting Final Grade document.

Rationale:

- Grade Improvement: Completing higher-grade tasks increases the likelihood of achieving a higher grade (e.g., Grade A or B).
- Skill Development: Implementing advanced features (e.g., internationalization, transactions, CI/CD) enhances technical skills and knowledge.
- Project Quality: Higher-grade tasks often involve best practices (e.g., testing, security, scalability), improving the overall quality of the application.

Implications:

- Increased Effort: Higher-grade tasks require more time and effort but result in a more robust and feature-rich application.
- Team Collaboration: All team members must contribute equally to ensure higher-grade tasks are completed on time.

Alternatives Considered:

- Aim for Lower Grade: Rejected because it would limit the potential grade and the quality of the application.
- Partial Implementation: Rejected because incomplete higher-grade tasks may not contribute to the final grade.

16 Logging Decision

Date: March 01, 2025

Decision

Adopt **Winston** as the logging tool for the backend.

Rationale

- **Widely used** and robust logging library in the Node.js ecosystem.
- **Flexible logging configuration** supports different log levels, log file rotation, and output formats.
- Supports **logging to multiple transports**, such as files and external services, making it versatile.
- **Active community support** ensures timely updates, bug fixes, and improvements.

Log Details

- **Where it is logged:** Logs will be stored in the `logs` folder in the backend server.
- **Log rotation:** Logs will be rotated daily to manage file sizes and avoid excessive disk usage.
- **What is logged:** Major events, including login activities, retrieving and modifying data in the database.
- **How it is logged:** A series of logging function calls will be implemented in the service layer of the backend, as it is the layer that has the most knowledge of what is received from the frontend and decides what is sent to the model layer and later to the database. This ensures that all significant interactions between the frontend and backend are logged effectively.

Alternatives Considered

- **Winston alternatives (e.g., Bunyan, Pino):** Rejected due to either lack of flexibility or less robust features for managing log levels, formats, and log rotation.

Implications

- Developers will need to become familiar with Winston's configuration, logging calls, and best practices for log rotation.
- A well-defined logging strategy should be established, including what constitutes a major event worth logging.
- Logs will need to be monitored to ensure they are rotating properly and not consuming excessive disk space.

17 CI/CD Pipeline Tool Decision

Date: March 05, 2025

Decision

Adopt **GitHub Actions** for Continuous Integration (CI) and Continuous Deployment (CD) pipeline.

Rationale

- **Native integration with GitHub:** GitHub Actions seamlessly integrates with GitHub repositories, simplifying the setup and management of the CI/CD pipeline.
- **Customizable workflows:** Allows for flexible and customizable workflows, which can be easily tailored to meet project-specific requirements.
- **Built-in actions and community support:** GitHub Actions provides a wide range of pre-built actions for tasks like static analysis, unit testing, and deployment, which can be easily incorporated into workflows.
- **Cloud deployment integration:** Supports deployment to various cloud platforms (e.g., AWS, Azure, Heroku), ensuring the pipeline can be adapted to different deployment environments.
- **Scalability and reliability:** GitHub Actions is highly scalable and reliable, with strong community support and constant improvements.

Pipeline Workflow

The CI/CD pipeline will follow these stages:

- **Static Analysis:** Code will be analyzed for potential issues (e.g., syntax errors, style violations) using tools like ESLint or Prettier, ensuring code quality and consistency.
- **Unit Testing:** Unit tests will be executed using frameworks like Jest or Mocha to ensure code correctness and reliability.
- **Deployment:** Upon successful static analysis and unit testing, the code will be automatically deployed to the cloud platform (e.g., AWS, Azure, Google Cloud) for staging or production environments.

Alternatives Considered

- **Jenkins:** Rejected due to its more complex setup and maintenance requirements compared to GitHub Actions, which integrates more seamlessly with GitHub repositories.
- **GitLab CI/CD:** Rejected due to reliance on GitLab repositories, which is not ideal given the project's usage of GitHub.
- **Heroku Pipelines:** Rejected due to limited flexibility and customization options when compared to GitHub Actions. While Heroku Pipelines provides an easy-to-use environment, it lacks the same level of control and integrations available with GitHub Actions for custom workflows and deployment to multiple cloud platforms.

Implications

- Developers will need to become familiar with GitHub Actions workflows, YAML configurations, and setting up various stages of the pipeline (static analysis, testing, deployment).

- The project will benefit from automatic deployment and testing, but manual review of failed pipelines may be required in some cases.
- Ensuring that all code changes pass the static analysis and unit testing stages will be critical before deployment to production.