# CONTINUOUS INTEGRATION

CONINT Semester Project Documentation

Kazem Gholibeigian, Lorenz Stoiber

## Table of Contents

## Documentation

The following document provides an insight on our project-, app- and pipeline-setup.

### Application and Dockerfiles

We hosted our source code in one main repository on our GitLab Server on AWS. It consists of three application folders: Backend, Frontend_green and Frontend_blue. In the beginning we also had a MariaDB application, as we had not set up AWS yet. The folder is stil included, but not used by our project.

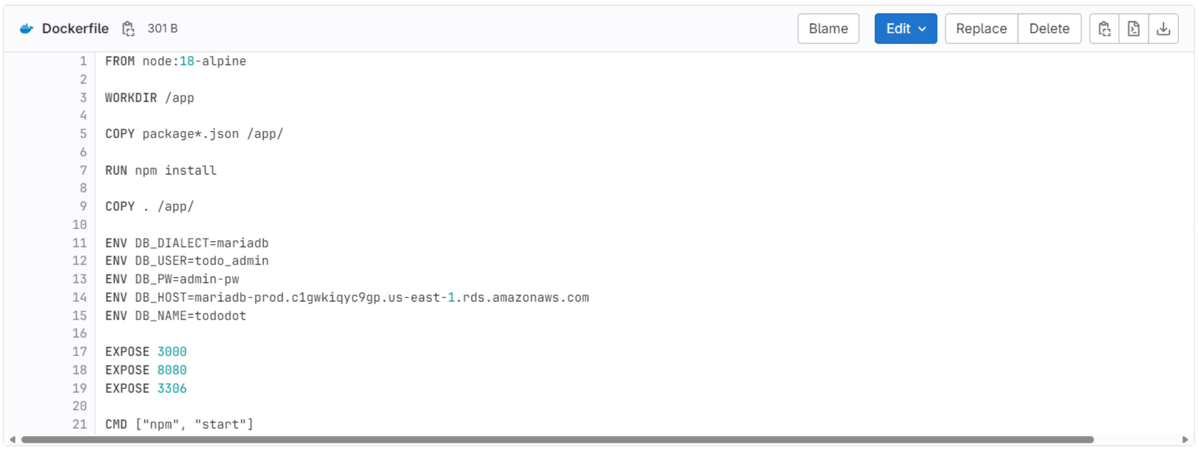
We split the frontend folders into two folders that each have their own Dockerfile. They are used for blue-green deployment in our pipeline. Each frontend as well as the backend have their own AWS EC2 instance, which just consists of a docker-compose.yml file which pulls the current image from our Docker Hub registry, then builds and runs it with necessary environment variables and configuration.

Our Dockerfiles for blue and green frontends are the same, but in case we want to configure something else in our staging environment, it would be possible that way.

For local testing, we used two different docker-compose files, one for production and one for staging. These files are not used currently but could be used for local tests.

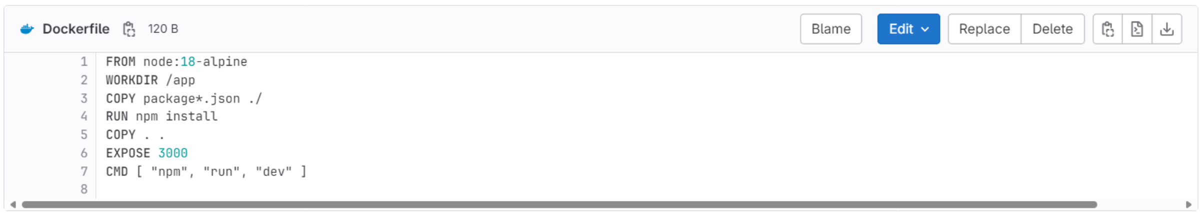
### Backend

Our backend’s Dockerfile defines environment variables for our database connection. We tried moving them to an .env file, but as we had difficulties with that, we just kept them there.



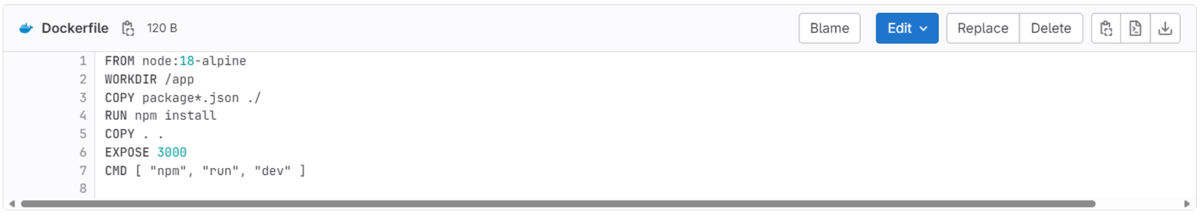
```
FROM node:18-alpine

WORKDIR /app

COPY package*.json /app/

RUN npm install

COPY . /app/

ENV DB_DIALECT=mariadb
ENV DB_USER=todo_admin
ENV DB_PW=admin-pw
ENV DB_HOST=mariadb-prod.c1gwkiqyc9gp.us-east-1.rds.amazonaws.com
ENV DB_NAME=tododot

EXPOSE 3000
EXPOSE 8080
EXPOSE 3306

CMD ["npm", "start"]
```

### Frontend Blue



```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm", "run", "dev" ]

```

## Frontend Green



```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm", "run", "dev" ]

```

## Version Control

We do not currently use semantic versioning. We simply push our latest version of our blue or green environments respectively, once all tests have succeeded. We planned on implementing more sophisticated version control, however we ran out of time to implement such functionalities at the project end.

This of course means, that conflict handling, traceability or revoking faulty releases is difficult or not possible with our pipeline at the moment.

## AWS Infrastructure

On AWS, we currently use 7 instances. A t2.large for our GitLab Server, which we upgraded as our initial setup with a t2.medium lacked power. Our GitLab Runner is hosted on a t2.small instance, which was sufficient most of the time. It sometimes took quite long for our pipeline to run through, so upgrading here could improve our workflow.

Our SonarQube instance is deployed on a t2.medium instance, which was, again, sufficient but not the quickest.

Operation of our application relies on the remaining four instances. A t2.large instance for our backend, a t2.medium instance for both of our frontends – one for blue and one for green – and lastly a t2.small instance that hosts our nginx configuration, as well as a python server which should handle our deployment switches from blue to green and vice versa.

For EBS we use an RDS MariaDB instance. Here we had to play around with creating multiple security groups, which was new to us, as we never had to use EBS before.

Two improvements we could have done in the future are narrowing down our security groups to increase security as well as using private IP addresses for communication between our instances. At the moment we configured noip.com hostnames, so that we do not have to change IPs in our servers' configurations.

## Docker Hub

We use Docker Hub for storing three different images. Our backend, our blue frontend as well as our green frontend. Green and blue are differentiating tags in our conint_frontend registry. Our backend registry is named conint, which is a name to be improved for sure.

As mentioned earlier, we configured our application's backend and frontend instances to create our app from docker-compose files, which are only hosted within our servers. They are not on our GitLab server, only in the filesystem of our servers. This could potentially be improved, but it works.

## CI Server Configuration

### GitLab Server

Our GitLab server is connected to via the following command.

*ssh -i "labsuser.pem" ubuntu@44.193.198.166 -p 2424*

The port has to be specified to 2424, as GitLab need port 22 for its own SSH connections.

After the first login, we updated using apt and installed docker using docker.docs' tutorial.

Then, we changed the SSH port to 2424 in the config file

*/etc/ssh/sshd_config*

We then restarted the session using

*sudo systemctl restart ssh*

We set up the volume location using the following commands:

*sudo mkdir -p /srv/gitlab*

*export GITLAB_HOME=/srv/gitlab* (in bash profile)

Afterwards, we created the docker container for GitLab with the following command:

```
sudo docker run --detach \
  --hostname gitlab.example.com \
  --env GITLAB_OMNIBUS_CONFIG="external_url 'http://gitlab.example.com'" \
  --publish 443:443 --publish 80:80 --publish 22:22 \
  --name gitlab \
  --restart always \
  --volume $GITLAB_HOME/config:/etc/gitlab \
  --volume $GITLAB_HOME/logs:/var/log/gitlab \
  --volume $GITLAB_HOME/data:/var/opt/gitlab \
  --shm-size 256m \
  gitlab/gitlab-ee:<version>-ee.0
```

We grep'd the password

```
sudo docker exec -it gitlab grep 'Password:' /etc/gitlab/initial_root_password
```

We made further configurations down the line, to ensure our hostnames were correct.

We singed in, imported our repository and added our SSH Keys, so that we could easily push an d pull changes.

## GitLab Runner

We set up our AWS instance for GitLab Runner, but used a less powerful processor. We SSHed into the instance and then installed docker once more.

We used the installation variant with a local system volume mount and changed the config.toml for privileged mode and our certs directory. We then registered a new runner using the registration roken from our GitLab instance. After a few tries we managed to make it work and the runner appeared in out GitLab Server, after which we set up our first dummy pipeline to test if our runner actually executes our pipeline – and it did.

For the installation, we used this tutorial: [Registering runners | GitLab]

## Code Quality Server

We have set up a SonarQube server on AWS to ensure our code quality remains high. Here is a detailed description of the setup process:

We created and launched a new instance, then SSHed into it.

```
ssh -i "labsuser.pem" ubuntu@ec2-public-dns
```

We updated the server software

```
sudo apt-get update

sudo apt-get install -y docker.io
```

Once again installed Docker and Docker-Compose Then we set up our compose file for our SonarQube instance

```
version: '3'

services:

  sonarqube:

    image: sonarqube:latest

    container_name: sonarqube

    ports:

      - "9000:9000"

    volumes:

      - sonarqube_conf:/opt/sonarqube/conf

      - sonarqube_data:/opt/sonarqube/data

      - sonarqube_logs:/opt/sonarqube/logs
    - sonarqube_extensions:/opt/sonarqube/extensions
volumes:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_logs:
  sonarqube_extensions:
```

Then we started it.

```
sudo docker-compose up -d
```

Afterwards, we accessed the app using our browsers, logged in and configured our project.

*http://sonarqube.zapto.org:9000*

We then configure SonarQube for our project

1. We created a new project in SonarQube and generated a project token.
2. Integrated SonarQube with our CI/CD pipeline using the token.

## Snyk Setup

We integrated Snyk into our project to bolster our security measures effectively. Initially, we created an account on Snyk's platform and configured our project within their interface. This setup involved linking our repository to Snyk, ensuring seamless integration.

Next, we automated security checks by integrating Snyk into our CI/CD pipeline. This automation enabled Snyk to conduct scans automatically with each code change, meticulously scrutinizing our dependencies and third-party libraries for vulnerabilities.

Throughout our build process, Snyk diligently analyzed our codebase, promptly alerting us to any security vulnerabilities it discovered. These alerts were accompanied by comprehensive reports and actionable remediation advice, empowering us to address issues promptly and maintain a robust security posture.

We opted for a setup using Snyks Website, as the project specification did not require us to use an AWS instance.

## Feature Toggles and A/B-&Testing

We integrated PostHog to manage feature toggles and conduct A/B testing within our project. Here's how we set it up:

1. **Account Setup and Project Configuration**: We began by creating an account on PostHog's platform. After signing up, we configured our project settings within PostHog, linking it to our repository where our application code resides.
2. **Feature Toggle Integration**: Utilizing PostHog's feature toggle functionality, we implemented dynamic feature toggles directly within our application. This allowed us to control feature releases and experiment with different functionalities easily.
3. **A/B Test Setup**: Leveraging PostHog's A/B testing capabilities, we designed experiments to compare different versions of features and interfaces. This setup enabled us to gather valuable user feedback and make data-driven decisions.
4. **Monitoring and Analytics**: PostHog provided comprehensive analytics and monitoring tools, allowing us to track user interactions with toggled features and analyze experiment results in real-time. This data-driven approach helped us optimize user experiences effectively.

## Pipeline Stages

### Lint Stage:

- **Purpose**: Ensures code quality and adherence to coding standards through static code analysis and linting.
- **Tools Used**: Utilizes Snyk and SonarQube for vulnerability scanning and code quality checks.
- **Actions**:
  - **snyk-check**: Authenticates with Snyk using $SNYK_TOKEN, scans for vulnerabilities across all projects in the organization, and monitors them for changes.
  - **sonarqube-check**: Executes the SonarQube scanner to analyze code quality, identify bugs, security vulnerabilities, and code smells.

### Build Stage:

- **Purpose**: Compiles the application code and prepares it for deployment.
- **Steps**:
  - **build_backend**: Builds Docker image cooopeir/conint:latest for the backend application located in the backend/ directory.
  - **build_blue**: Builds Docker image cooopeir/conint_frontend:blue for the blue version of the frontend located in frontend_blue/.
  - **build_green**: Builds Docker image cooopeir/conint_frontend:green for the green version of the frontend located in frontend_green/.

### Test Stage:

- **Purpose**: Validates the functionality and reliability of the application through automated testing.
- **Steps**:
  - **test_backend**: Runs backend tests (if implemented) to ensure backend functionality is working as expected.
  - **test_blue**: Executes Vue.js frontend tests located in frontend_blue/, ensuring the blue version behaves correctly.
  - **test_green**: Executes tests for the green version of the frontend located in frontend_green/.

### Deliver Stage:

- **Purpose**: Prepares Docker images for deployment by pushing them to a Docker registry.
- **Steps**:
  - **deliver_backend**: Pushes Docker image cooopeir/conint:latest to Docker Hub after successful testing of the backend.
  - **deliver_blue**: Pushes Docker image cooopeir/conint_frontend:blue to Docker Hub after successful testing of the blue frontend.
  - **deliver_green**: Pushes Docker image cooopeir/conint_frontend:green to Docker Hub after successful testing of the green frontend.

### Deploy Stage:

- **Purpose**: Automates the deployment of Dockerized applications to staging or production environments.
- **Steps**:
  - **deploy_backend**: Deploys the backend application using docker-compose.yml to a specified server (10.0.0.31), handling container lifecycle management (down, prune, up).
  - **deploy_blue**: Deploys the blue version of the frontend application using docker-compose.yml to another server (10.0.0.18), ensuring smooth transition and updates.
  - **deploy_green**: Deploys the green version of the frontend application using docker-compose.yml to yet another server (10.0.0.35), maintaining high availability and reliability.

## Failing Stage Handling

In our CI/CD pipeline, we handle failing stages to ensure smooth and efficient development and deployment processes. Here's how we manage failing stages:

- **Purpose**: The primary goal is to quickly identify and resolve issues that cause stages to fail, minimizing downtime and ensuring rapid feedback to developers.

- **Strategies**:
  - **Immediate Notification**: We receive immediate notifications upon stage failure via our CI/CD tool (CircleCI). This allows us to promptly investigate the root cause.
  - **Detailed Logging**: Each stage logs detailed information about its execution. This helps in diagnosing failures by providing insights into what went wrong.
  - **Automatic Rollback**: For deployments, we implement automatic rollback mechanisms in case of deployment failures. This ensures that the application remains in a consistent state.
  - **Manual Intervention**: Sometimes, certain failures require manual intervention. We have predefined procedures and runbooks for handling such scenarios to expedite resolution.

## Feature Documentation

Our project's features and user stories are documented comprehensively to ensure clarity and alignment across the development team. Here's how we structure our feature documentation:

- **Feature Overview**: Each feature is described in terms of its purpose and functionality. This includes a high-level summary to provide context and importance.
- **User Stories**: Features are broken down into user stories that outline specific functionalities from an end-user perspective. Each user story includes:
  - **Title**: A concise title that describes the functionality.
  - **Description**: Detailed information about what the user story entails and its expected outcome.
  - **Acceptance Criteria**: Clear criteria that define when a user story is considered complete and meets expectations.
- **Dependencies**: Any dependencies related to each feature or user story are documented to ensure that all prerequisites are met before implementation.
- **Updates and Changes**: Documentation is updated to reflect any changes or updates to features and user stories throughout the development lifecycle.

## A/B Testing Documentation

A/B testing is conducted to compare two versions of a product or feature to determine which one performs better. Here's how we set up and document A/B testing:

- **Purpose**: A/B testing helps us make data-driven decisions about features and improvements by comparing user responses to different variations.
- **Steps to Recreate**:

1. **Define Experiment**: Clearly define the objective of the A/B test and identify the metrics that will be used to measure success.
2. **Setup Variations**: Create multiple variations (A and B) of the feature or interface being tested. These variations should differ in specific aspects you want to evaluate.
3. **Random Assignment**: Randomly assign users to either variation A or B to ensure unbiased results.
4. **Measure and Analyze**: Use analytics tools to measure user engagement, conversion rates, or other relevant metrics for both variations.
5. **Draw Conclusions**: Analyze the results to determine which variation performed better based on the defined metrics.

## Feature Toggle Documentation

Feature toggles allow us to enable or disable features in real-time without redeploying the application. Here's how we document and implement feature toggles:

- **Implementation**: Feature toggles are implemented using configuration files or environment variables that control feature visibility.
- **Steps to Recreate**:
    1. **Define Toggle Points**: Identify specific points in the application where feature toggles will be implemented (e.g., UI components, backend services).
    2. **Toggle Configuration**: Create configuration files or use environment variables to define toggle states (enabled or disabled) for each feature.
    3. **Integration**: Integrate feature toggles into the application logic to conditionally enable or disable features based on toggle states.
    4. **Testing**: Test feature toggles thoroughly to ensure they work as expected across different environments and scenarios.
- **Documentation**: Document each feature toggle, including its purpose, toggle point, configuration details, and any dependencies. This documentation helps in managing feature releases and conducting controlled rollouts.

## Blue/Green Deployment

Our deployment strategy involves using a nginx.config file on a separate server to toggle between the live (blue) and staging (green) environments. Here's how it works:

- **Purpose**: Blue/Green deployment ensures zero downtime during deployments by switching traffic between two identical environments.
- **Setup**:
    - **Environment Setup**: Maintain two identical environments: one serves live traffic (blue) while the other (green) serves as a staging environment.

  - **nginx.config File**: Configure nginx.config to route traffic between the blue and green environments based on specific criteria (e.g., URL paths, headers).
  - **Toggle Mechanism**: Modify nginx.config to direct incoming requests to either the blue or green environment, facilitating controlled deployments and A/B testing.

- **Deployment Process**:
  - **Deploy to Staging**: Push changes to the green environment using Docker and docker-compose.yml.
  - **Testing**: Validate changes in the green environment to ensure functionality and performance.
  - **Toggle Traffic**: Update nginx.config to direct traffic from the blue environment to the green environment, making the green environment live.
  - **Rollback**: In case of issues, revert nginx.config to redirect traffic back to the blue environment while issues are addressed.

## Logging and Monitoring Outlook

Incorporating effective logging and monitoring practices enhances visibility and facilitates proactive management of our project. Here's our outlook on logging and monitoring:

- **Current Practices**:
  - **Logging**: Use centralized logging tools (e.g., ELK Stack, Fluentd) to aggregate and analyze logs from various components within the application.
  - **Monitoring**: Implement monitoring tools (e.g., Prometheus, Grafana) to track metrics, performance, and health of the application and infrastructure.
- **Suggestions for Improvement**:
  - **Enhanced Instrumentation**: Increase the granularity of logging to capture detailed insights into application behavior and user interactions.
  - **Alerting Mechanisms**: Set up alerts based on predefined thresholds for metrics such as CPU usage, memory consumption, and response times.
  - **Dashboard Customization**: Customize monitoring dashboards to display relevant metrics and KPIs, providing actionable insights at a glance.
  - **Automated Remediation**: Implement automated responses to detected issues, such as scaling resources or triggering rollback procedures.
- **Continuous Optimization**: Regularly review and optimize logging and monitoring configurations based on observed patterns and performance metrics. This iterative process ensures that our project remains resilient and responsive to evolving demands.

# Project Management

## Definition of Done

**A feature is considered done when:**

- All specified requirements and user stories are implemented.
- The code is reviewed and approved by at least one other developer.
- Unit tests are written and passing with a minimum of 80% code coverage.
- Integration tests are written and passing.
- End-to-end tests are written and passing.
- The feature is documented, including usage instructions and any configuration settings.
- The feature is deployed to a staging environment and verified to work as expected.
- User acceptance testing (UAT) is completed and signed off by a stakeholder.
- Any identified bugs are fixed, and regression testing is performed.
- The feature is merged into the main branch and deployed to production.
- Post-deployment monitoring is set up to ensure feature stability.

## User Stories

### EPIC 1: Feature Allowing Multiple Users with Registration and Login

**User Story: User Registration**

As a new user, I want to register an account, so that I can access the application's features.

*Acceptance Criteria*:

- The registration page includes fields for username, email, password, and password confirmation.
- The system validates the email format and ensures the password meets security requirements (e.g., minimum length, contains numbers and special characters).
- Users receive a confirmation email with an activation link upon successful registration.
- The system prevents duplicate registrations using the same email.
- Error messages are displayed for invalid input or if the email is already registered.

**User Story: User Login**

As a registered user, I want to log in to the application, so that I can access my personal data and settings.

*Acceptance Criteria*:

- The login page includes fields for email and password.
- Users can log in using their registered email and password.
- The system provides feedback for incorrect email or password.
- A "Forgot Password" link directs users to a password recovery page.
- Users stay logged in until they log out or their session expires.

**User Story: Password Recovery**

As a user who forgot their password, I want to recover it, so that I can regain access to my account.

*Acceptance Criteria*:

- The "Forgot Password" page includes a field for the registered email.
- Users receive a password reset link via email.
- The password reset link directs users to a page where they can enter a new password.
- The system validates the new password using the same criteria as registration.
- Users can log in with the new password immediately after resetting it.

## EPIC: Sorting Todos with Unfinished Todos Moving to the Next Day + Feature Toggle

**User Story: Sorting Todos**

As a user, I want my Todo items to be sorted by due date, so that I can prioritize my tasks effectively.

*Acceptance Criteria*:

- Todos are displayed in ascending order of their due dates.
- Overdue Todos are highlighted and remain at the top of the list until marked as completed.
- Completed Todos are moved to the bottom of the list or a separate section.

**User Story: Unfinished Todos Moving to the Next Day**

As a user, I want my unfinished Todos to automatically move to the next day, so that I can continue working on them.

*Acceptance Criteria*:

- Unfinished Todos are automatically rescheduled to the next day at midnight.
- The system notifies users of the rescheduling action.
- Users can manually adjust the due date of any Todo item.

**User Story: Feature Toggle for Todo Rescheduling**

As an admin, I want to enable or disable the automatic rescheduling of Todos, so that I can control the feature availability.

*Acceptance Criteria*:

- An admin interface includes a toggle to enable or disable the Todo rescheduling feature.
- The system checks the toggle state before rescheduling Todos.
- Changes to the toggle state take effect immediately.
- A notification is displayed confirming the toggle state change.

## EPIC: A/B Testing with Different Register Button Colors and Headlines

**User Story: A/B Testing Framework Setup**

As a developer, I want to set up an A/B testing framework, so that I can test different UI elements and measure their effectiveness.

*Acceptance Criteria*:

- An A/B testing framework is integrated into the application.
- The framework supports multiple experiments running simultaneously.
- Experiment data is logged for analysis.

**User Story: Implementing Different Register Button Colors**

As a developer, I want to implement different colors for the register button, so that I can test which color has a higher conversion rate.

*Acceptance Criteria*:

- Two or more variants of the register button with different colors are created.
- Users are randomly assigned to see one of the button variants.
- Click events on the register button are tracked and logged.

**User Story: Implementing Different Headlines**

As a developer, I want to implement different headlines on the registration page, so that I can test which headline attracts more users.

*Acceptance Criteria*:

- Two or more variants of the registration page with different headlines are created.
- Users are randomly assigned to see one of the headline variants.
- Page view and conversion data are tracked and logged for each variant.

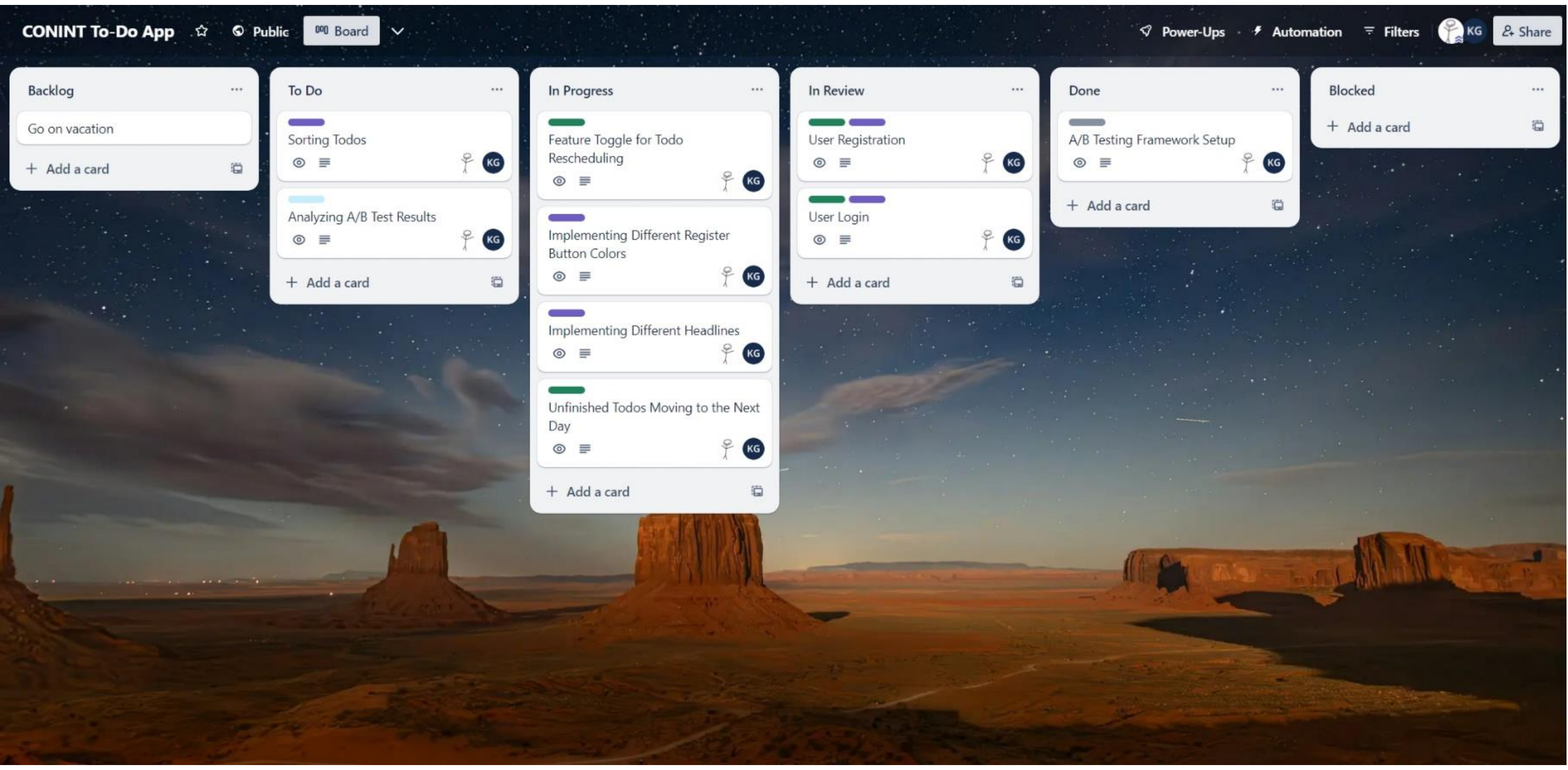
**User Story: Analyzing A/B Test Results**

As a developer, I want to analyze the results of the A/B tests, so that I can determine which variant performs better.

*Acceptance Criteria*:

- Data from the A/B tests is collected and stored in a database.
- A dashboard or report is created to display the test results, including conversion rates and statistical significance.
- The results are reviewed to determine the winning variant.
- The winning variant is deployed to production after the test concludes.

## Trello Board



Backlog
- Go on vacation
- + Add a card

To Do
- Sorting Todos
- Analyzing A/B Test Results
- + Add a card

In Progress
- Feature Toggle for Todo Rescheduling
- Implementing Different Register Button Colors
- Implementing Different Headlines
- Unfinished Todos Moving to the Next Day
- + Add a card

In Review
- User Registration
- User Login
- + Add a card

Done
- A/B Testing Framework Setup
- + Add a card

Blocked
- + Add a card