2022 AIIT PBL

情報アーキテクチャコース
追 川 PT

大山 勇
池田 拓磨
砂川 俊明
御園生 銀平

主担当教員：追川 修一 教授
副担当教員：中鉢 欣秀 教授
黄 緒平 助教

DXサービス開発によるアジャイル開発の実践と開発スキルの習得

プロジェクトの目的

DXサービスの開発を通じ、実践的な開発スキルを習得する

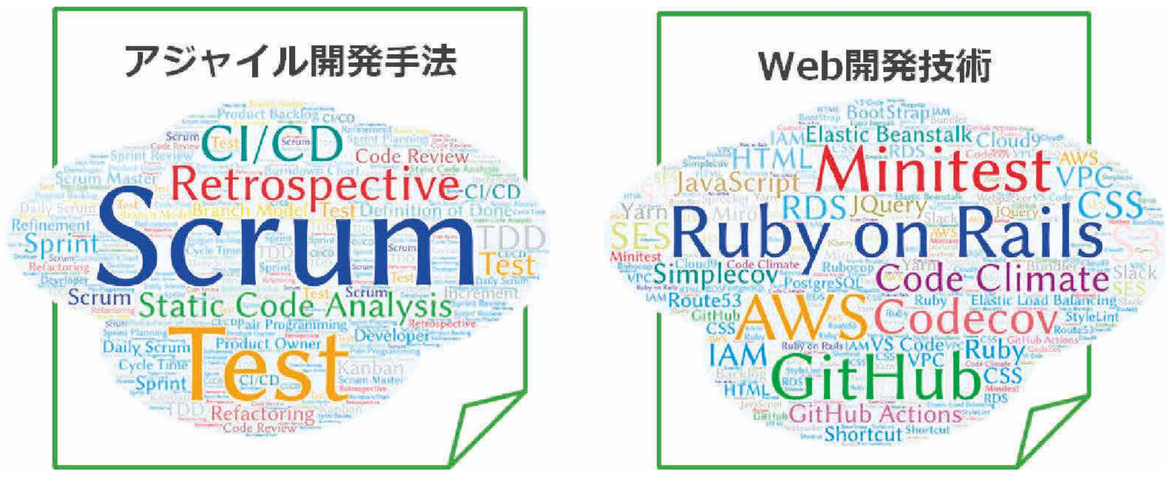
- アジャイル開発手法を用いてWeb開発を実践すると共に、開発の中でスクラムやCI/CD、リファクタリング等のプラクティスを習得
- Web開発で必要となる言語やツールを実際に利用する中で、それら技術の特性を理解すると共に、開発における実装能力を習得

背景
チームメンバー全員が、開発初心者の状態から脱却し、開発者として価値あるサービスを作り上げる実力を身に着けるために、より実践的な開発スキルの習得を当プロジェクトの目的とした

優先度: **開発スキルの習得** > 新規性のあるWebサービスの開発

当プロジェクトで扱ったプラクティスやツール

アジャイル開発手法やWeb開発技術に関連した以下のようなプラクティスやツールを利用し、開発スキルの習得を行った



※ 当プロジェクトで頻繁に扱ったプラクティスやツールを大きな文字で表現

活動概要

開発開始までの準備

- 当プロジェクトで習得したいスキルの具体化や開発するWebサービスの内容の検討
- 多くの既存サービスがあるナレッジ共有サービスを開発テーマとすることでアイディア検討を簡略化し、スプリントのプロセスの実践に注力
- スプリントの実施に向けてデプロイ環境やCI/CDパイプラインの構築の実施

スプリント#01–#10

- 2週間スプリントを計10回実施し、各スプリントでWebサービスのリリースやより効率的な開発を行うためのスプリントの改善活動を実施
- スプリントの合間に数回、学習活動期間を設け、開発を進めるにあたって足りないスキルの検討および書籍等による学習を実施

より良いコードを書きたいというモチベーションが生まれる

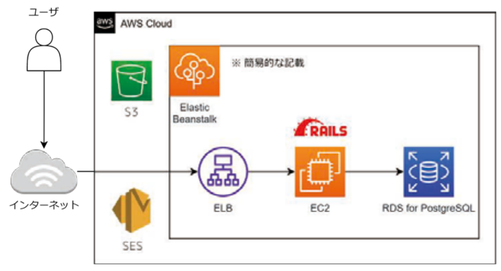
振り返り

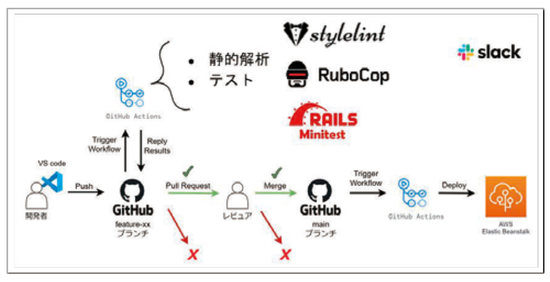
- これまでの活動を振り返り、学んだ内容を整理

開発したWebサービスと構築した環境

社内向けナレッジ共有サービス Knowledgekun
Markdown形式で記事を記述し、組織のメンバー間で知識を共有できるWebサービスを開発

Webサービスの構成要素
Web開発の実績、情報量等の観点から以下の技術を採用
- 開発フレームワーク: Ruby on Rails
- デプロイ環境: AWSの各種サービス





CI/CDパイプライン
- CIを用いてテストや静的コード解析を自動化することでレビュアは本質的な内容のレビューのみに集中
- 簡易的なCDを用いてデプロイを自動化することで手間を軽減

注力した学習内容: より良いコードを書くには？

当プロジェクトにおける『より良いコード』の探求プロセス

1 各自テーマを設けて学習
各自の興味がある分野をそれぞれ学習
- コードカバレッジ
- テスト駆動開発
- リーダブルコード
- リファクタリング
- etc...

2 より良いコードの定義
学習した内容を共有し、議論した上で『**より良いコード**』の定義を設定

3 施策への落とし込み
より良いコードの実現に向けた施策を検討し、開発プロセスに組み込む

当プロジェクトにおける『より良いコード』とは?

カイゼンしやすいコード

- コードの重複や構造的な問題が少なく、コードの意味や意図が理解しやすい
- モジュール単位で十分テストされ、コードの変更におけるバグの発生等が起きにくい

施策1
コードの保守性を向上させるコーディング規約の設定

施策2
テスト戦略を見直し、単体テストをより充実させる

※ これら施策をCIに組込むことで自動で施策を評価する仕組みを構築

施策1: コードの保守性を向上させるコーディング規約の設定

コードの保守性とは?
コードの保守のしやすさを表し、当プロジェクトではコードの重複やその他の構造的な問題の有無を基に定量化した

- コード品質を評価するサービスであるCode Climateを利用することで、Code Climateで定義される評価指標を基にコードの保守性を評価できる

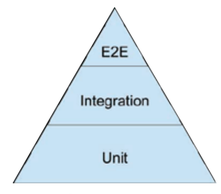
[具体的な評価項目例]
ファイルの長さ、メソッドの行数の長さ、制御フローの数、ネストの深さ 等々

Code Climateによるコードの保守性の評価

Code Climateのコード評価指標を基にコーディング規約を設定
- コードの保守性の指標について学習し、学習内容をコードの記述に活かす
- 既存コードの保守性を評価し、継続的にコードの書き方を改善する

施策2: テスト戦略を見直し、単体テストをより充実させる

以前のテスト戦略	E2Eテストが多く、単体テストは少ない (E2Eテストは実際の稼働に近いテストができるため、Webサービスを十分にテストできると考えた)
何が問題なのか？	E2Eテストは実行速度が遅く、テストのカバー範囲が広いため、問題発見時に原因特定が難しい
単体テストを充実させると？	E2Eテストの問題を軽減でき、コード変更の心理的ハードルが下がり、コードを改善がしやすい



テストピラミッド構造を意識
Google Testing Blog: Just Say No to More End-to-End Tests よりイメージを参照

E2Eテストを最小限に抑えつつ、単体テストの拡充に注力する
- 単体テストのカバレッジを可視化することで、単体テストを書くことを意識

その他学習内容

- リモート環境下における効率的なコードレビュー実施方法
- フィージビリティスタディの実施方法
- 効果的なNext Actionを生み出すスプリントレトロスペクティブ実施方法
- Ruby on Railsによるマルチテナント構成のWebサービス実装
- GitHubのブランチ戦略とコミット戦略
- AWSの各種マネージドサービスを利用したWebサービスの環境構築　　等々