卒業論文

デジタル版しおりアプリの開発

関西学院大学理工学部

情報工学課程 西谷研究室

37022463 山本果音

2026年3月

概 要

旅行アプリ「旅しお」では旅行計画を簡単に立てることが可能であるが, 予定作成時の情報整理や視覚的な把握が不十分であり, 継続的な利用が困難である. 本研究では, 旅行前の計画段階において, ユーザーがスムーズに情報を整理・共有できることを目的とし, デジタル版しおりの開発を行った. 具体的な特徴として, 旅行名や説明文をキーとした検索機能, 旅行日付に基づく時系列整理, 目的地の地図表示機能, 移動時間表示機能, 訪問順序入れ替え機能, チェックリスト機能を実装し, 新たな予定を立てやすくした. 結果としてユーザーは旅行計画を効率的に立てることができ, 継続的な利用が可能になった.

目 次

図 目 次

第1章　序論

修学旅行などのイベントでは, 参加者に対して事前にしおりが配布されることが多い. しおりは旅程の把握や所持品の確認, 集合時間の共有などを目的とした重要な情報媒体であり, 旅行の準備や当日の行動において欠かせない役割を果たしている. 近年では, スマートフォンの普及により, 紙媒体に代わって Web サービスを活用した旅行情報の閲覧・共有のニーズが高まっており,「旅しお」[1] のような Web アプリも登場している. しかし, これらのサービスにはいくつかの課題が存在する.

1 つ目は, 旅行履歴の表示が作成日や更新日を基準としており, 実際の旅行日付に基づいた時系列での整理がされていない点である. そのため, 過去の旅行の実施状況を視覚的に把握することが困難である.

2 つ目は, 旅行名や説明文に対するキーワード検索機能が存在しないため, ユーザーはスクロール操作によって目的のしおりを探す必要がある点である.

3 つ目は, 地図表示機能がないため, 目的地の位置関係や全体像を視覚的に理解することが難しく, 利用者は外部ツールや手作業で補完しなければならない.

これらの課題は, 旅行計画の際にユーザーにとって大きな障壁となっており, 情報の整理や視覚的な理解の低下が考えられる. そこで本研究では, 視覚的な理解と旅行後の振り返りを行える「デジタル版しおり」として機能する旅行アプリの開発を目指す.

第2章　手法

2.1　開発手法

本研究では, 限られた開発期間の中で, 機能単位に計画・設計・実装・テストを小刻みに繰り返す手法を採用した. 各段階で得られた知見や, 研究室内のメンバーからのフィードバックをもとに, 仕様や UI の見直しを行いながら, 柔軟に改善を重ねた. これにより, 利用者の視点を反映した機能の追加や調整が可能となり, アプリケーションの完成度と実用性を段階的に高めることができた.

2.1.1　Dango

開発環境として,Python で実装されたサーバーサイド Web アプリケーションフレームワークである Django を選定した [2]. Django を選定した理由は以下の 3 つである.

1 つ目は, ユーザー認証や管理画面, データベース操作などの機能が初期状態で組み込まれており, 追加設定を行わずとも迅速に開発を開始できる点である. これにより, 開発初期の環境構築にかかる手間を大幅に削減することが可能となった.

2 つ目は, データベース操作をすべて Python で記述できるため, 学習コストが低い点である. Django では, モデルの定義からマイグレーションの実行までを一貫して Python で記述できるため,SQL の詳細な知識がなくても複雑なデータ構造の設計や変更が容易に行える.

3 つ目は, セキュリティ対策がフレームワークに標準で備わっている点である. Django はクロスサイトスクリプティング (XSS) や SQL インジェクションなどの脆弱性に対して, 初期状態で対策が施されている. そのため, 開発者が個別にセキュリティ機能を実装する必要がなく, 安全性の高い Web アプリケーションを効率的に構築することができた.

また, 実装には Python, HTML, JavaScript, CSS を使用し,UI ライブラリとしては Bootstrap を採用した．開発にあたっては Copilot[3] を参照しながら, 開発を行った.

2.1.2 Heroku

第3章　結果と考察

3.1　アプリ機能

今回，開発したアプリでは大きく分けて 4 つの機能を実装した．それぞれの機能は図 3.1 の通りである．

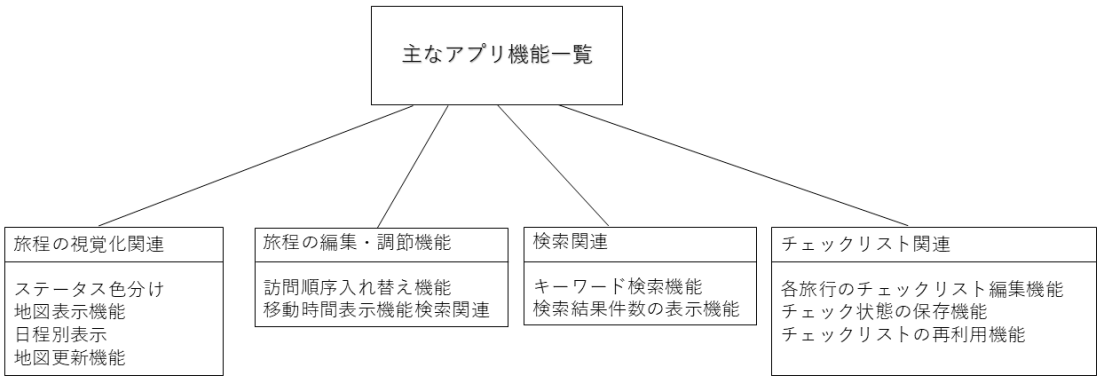

図 3.1: アプリ機能一覧.

3.1.1　旅程の視覚化管理

本システムでは, ユーザーが旅行の全体像や日程ごとの計画を直感的に把握できるよう, 複数の視覚化機能を実装した.

まず, 登録された旅行期間に基づいて, 各旅行を「予定」「旅行中」「終了」「未定」の 4 つのステータスに分類し, 色分けによって一覧画面上に表示する機能を実装した．これに

より, ユーザーは「どこに行ったか」「これからどこに行くか」といった旅行の進行状況を一目で把握できるようになった.

また, 旅行詳細画面では,Google Maps API を用いて登録された観光場所を地図上に表示する機能を導入した. これにより, 各観光地の位置関係や移動ルートを視覚的に確認でき, 旅行計画全体の空間的な構造を把握しやすくなった. さらに, 旅行に含まれる観光場所を「全体」「1 日目」「2 日目」など日程ごとに切り替えて表示する機能を実装した. ユーザーはタブを操作することで, 旅行全体の流れを理解しつつ, 各日の予定を個別に確認・編集することが可能となった. タブの切り替えに連動して, 地図上に表示される観光地もその日に対応するもののみに自動で更新されるため, 日程ごとの旅程を視覚的に管理しやすい設計となっている.

これらの機能により, ユーザーは旅行の計画段階において, 旅程を効率的に把握・調整できるようになり、より直感的な旅行プランニングが可能となった.

3.1.2 旅程の編集・調節機能

2 つ目は

3.1.3 検索機能

3 つ目は検索機能であり, 旅行一覧画面に多くの旅行が表示されている場合, 目的の旅行を探し出すのに時間がかかる. そこで, 旅行名や説明欄に含まれる単語をもとに, 部分一致によるキーワード検索を行い, 該当するデータを絞り込めるようにする. タイトルの語句をキーとして検索するキーワード検索に分けて検索機能を実装した.

3.1.4 チェックリスト機能

3.1.5 通知機能

5 つ目は通知機能であり，今回開発したような情報共有アプリに通知機能を搭載するメリットは以下の 4 点である.

図 3.2: 検索画面.

1. リアルタイムな情報提供.

2. エンゲージメント向上.

3. ユーザーエクスペリエンスの向上.

4. 重要な情報の警告.

まず 1. に関して，通知機能により，ユーザーはリアルタイムで新しい情報や重要なイベントにアクセスできる．例えば，新しいメッセージ，更新されたデータ，または特定のアクションが実行されたことを通知できる．次に 2. に関して，ユーザーに対して新しい情報やアクションを通知することで，アプリへのエンゲージメントが向上する．通知はユーザーの注意を引きつけ，アプリの利用頻度を増加させる効果がある．次に 3. に関して，通知はユーザーエクスペリエンスを向上させる．ユーザーはアプリを開かなくても新しい情報を受け取ることができ，アプリの使用が便利になる．最後に 4. に関して，出席連絡や期限に関する通知は，重要事項であるため，高確率でユーザーに行き届かないと意味がない．通知機能はユーザーにこれらの情報を素早く，確実に伝達することができる．Habit man はメールでの通知を基本とする．メール通知は Create，Update 処理をした際にグループメンバー全員に行き届くようにした．

3.2　日付軸管理

投稿されたデータは保存したい日付を 1 つのカラムに保持しているため，日付ごとのデータ管理が容易となった．また，投稿されたデータ一覧画面では，タグやキーワードでのデータ絞り込み機能が利用可能である．また，どちらのアプリもログイン済みであることを前提として，投稿されたデータを参照するまでのアクション数を比較する．Discord ではサーバーの選択，チャンネルの選択，画面のスクロールなど最低でも 3 つのアクションを要する．Habit man では日付選択の 1 アクションでデータを参照することが可能である．

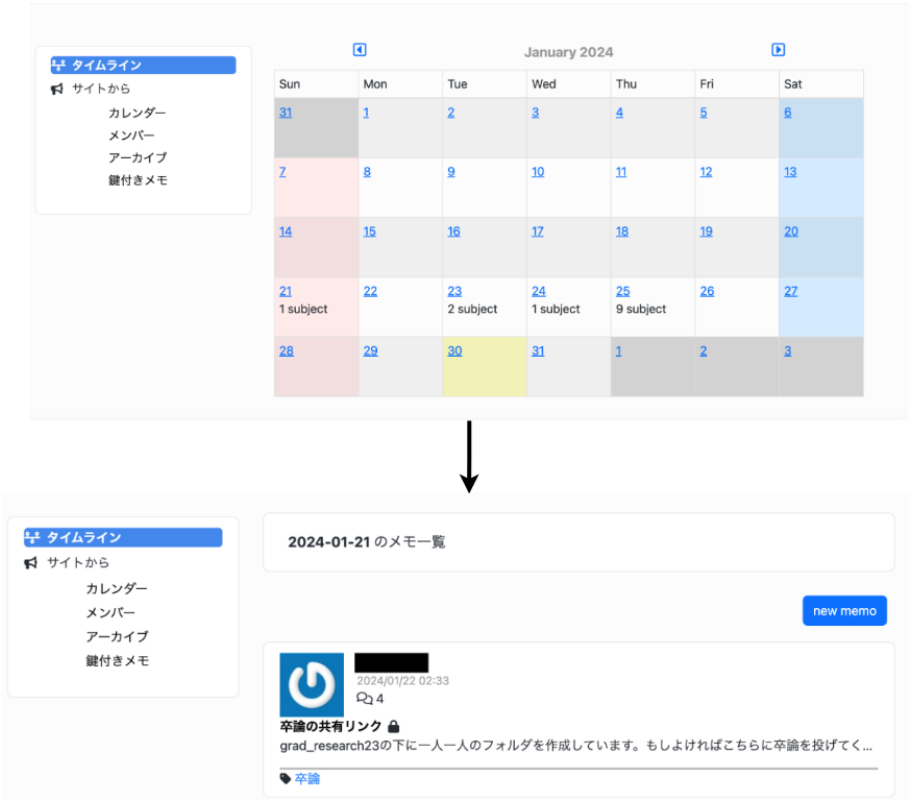

図 3.3: 参照したい日付に保存されたデータを参照する一連の動作.

3.3　ローカル環境から本番環境へ

3.3.1　Heroku × Github でのデプロイ方法

CLI を使用した Heroku への Web アプリケーションのデプロイ方法が存在するが，今回は Github 連携を使用したデプロイ方法を記す．Heroku は Github と統合することで，Github 上のソースコードを Heroku 上で実行中のアプリに容易にデプロイできるようになる．最初に，Heroku 上でアプリケーションを作成し，Heroku ダッシュボードの Deploy タブを押下し，Deploy 方法を選択する箇所から Github を選択する．Github を選択した後

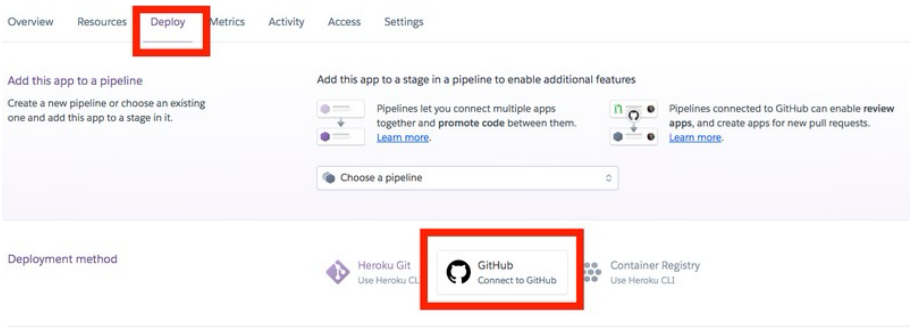

図 3.4: Heroku と Github の連携方法

は,「Connect to Github」というボタンが表示され，そのボタンを押下する．そうすれば，Github での認証画面が表示されるため,「Authorize heroku」というボタンを押下し，認証を成功させる．次に，Github との統合が完了すると図 3.9 のような画面が表示されるため，連携したい Github リポジトリを検索し，Connect ボタンを押下する．最後に，任意の

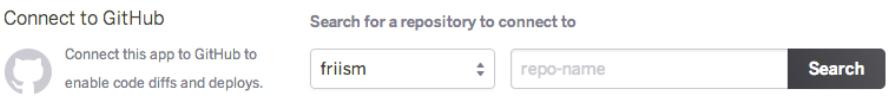

図 3.5: 任意の Github リポジトリと Heroku の連携.

Github リポジトリとの連携が完了すると Manual deploy という欄が表示される．Manual deploy の欄にある「Deploy Branch」というボタンを押下し，Heroku は自動的にビルドを開始する．ビルドに成功した場合はリリースすることができる．

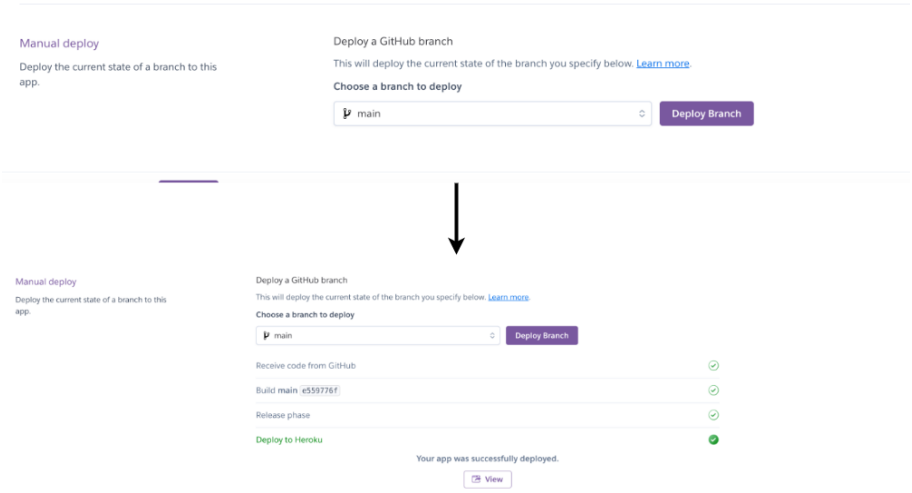

図 3.6: Heroku への手動デプロイ方法

3.3.2 Heroku Add-ons

今回，本番環境用のアドオンとして 2 つのアドオンを選定した．アドオンは Heroku Dashboard または CLI を使用してアプリケーションにインストールすることができるが，Heroku Dashboard を使用してインストールした．1 つ目のアドオンは「Heroku Postgres」である [13]．データベースとのインターフェースを提供する Rails の ActiveRecord[14] のデフォルトが PostgreSQL を対象としている [15]．そこで，Heroku 上でも PostgreSQL を動作させる．次のアドオンは「Heroku Scheduler」である [16]．アーカイブ機能を Heroku 上で実現するために Heroku が提供している専用のスケジューラーアドオンを使用しなければいけない．Rails の場合は規則として rake タスクを設定する必要があり，さらに lib/tasks フォルダ直下にバッチ処理を組み込んだファイルを配置しなければならない．

第4章　まとめ

本研究では，Ruby で書かれたサーバー側 Web アプリケーションフレームワークである Rails を開発環境として，データを共有・見返す作業を癖付けることを目的としたグループウェアアプリ Habit man を開発した．

開発手法として事前に全ての機能やサービスの詳細な要件，作業スケジュールを立てるウォーターフォール開発ではなく，優先順位を付けて重要な機能やサービスを段階に分けて開発し，ユーザーから得た反応から方向性や仮説を見出し，ユーザーへ価値を素早く届け，実戦投入の学びから素早く改善を行うというサイクルを確立するアジャイル開発をした．結果として，研究室のメンバーから早い段階でフィードバックを頂けたのでより良いグループウェアアプリへと修正することができた．また，機能的な側面としてカレンダー表示やタグ付け機能などの複雑な実装はパッケージマネージャーである gem に配布されているライブラリを用いて実現させた．

日付を第 1 キーとしてデータを保存することによって，日付ごとのデータ管理が容易になり，データの検索が高速化した．加えて，データをただ単に溜めていくだけではなく，独自のアルゴリズムから不必要と判定されたデータをアーカイブに移動することによってデータベースやストレージの使用量を最適化できた．

結果として，グループメンバーのデータ共有・見返す作業の習慣化に繋げることができた．

謝辞

本研究を進めるにあたり, 西谷滋人教授には指導教員として終始熱心なご指導を頂きました. 心から感謝いたします. また, アジャイル手法でのアプリ開発を進めていくうえで実際にアプリを使用していただいたり, 貴重な意見を下さった西谷研究室に所属している皆さまにも大変お世話になりました. お礼申し上げます.

参考文献

[1] Discord - https://discord.com (accessd on 21 Nov 2023).

[2] 野口 悠紀雄, 「超」整理法—情報検索と発想の新システム (中公新書), 中央公論新社, (1993).

[3] Slack - https://slack.com/intl/ja-jp/help/articles/115004071768-Slack-%E3%81%A8%E3%81%AF (accessd on 6 Jan 2024).

[4] Agile software development - https://en.wikipedia.org/wiki/Agile_software_development (accessd on 20 Nov 2023).

[5] Ruby on Rails - https://rubyonrails.org/ (accessd on 20 Nov 2023).

[6] まつもとゆきひろ - https://ja.wikipedia.org/wiki/%E3%81%BE%E3%81%A4%E3%82%82%E3%81%A8%E3%82% (accessd on 18 Dec 2023).

[7] Qiita - https://help.qiita.com/ja/articles/qiita (accessd on 18 Dec 2023).

[8] RubyGems - https://rubygems.org/ (accessd on 18 Dec 2023).

[9] Heroku - https://www.heroku.com/what (accessd on 18 Dec 2023).

[10] Heroku Add-ons - https://elements.heroku.com/addons (accessd on 18 Dec 2023).

[11] Convention over Configuration - https://en.wikipedia.org/wiki/Convention_over_configuration (accessd on 18 Dec 2023).

[12] Habit man - https://membermanagementapp-d0c147b97826.herokuapp.com/

[13] Heroku Postgres - https://devcenter.heroku.com/ja/articles/heroku-postgresql (accessd on 29 Jan 2024).

[14] Active Record の基礎 - https://railsguides.jp/active_record_basics.html (accessd on 29 Jan 2024).

[15] Rails コア開発環境の構築方法 - https://railsguides.jp/development_dependencies_install.html (accessd on 29 Jan 2024).

[16] Heroku Scheduler - https://devcenter.heroku.com/ja/articles/scheduler (accessd on 29 Jan 2024).

[17] Strong parameters - https://railsguides.jp/action_controller_overview.html#strong-parameter (accessd on 3 Dec 2023).

付 録A　コード詳細

A.1　データベース(DB)設計

今回，開発したWebアプリのデータベース構造を図A.1に示す．例えばグループには複数のユーザーが所属し，ユーザーは複数のグループに所属できるという仕様にしたためUserテーブルとGroupテーブルの関係性は多対多になる．したがって，中間テーブルとなるGroupUserテーブルを作成した．また，ユーザーは複数のメモを作成することができ，グループは複数のメモを保持するという仕様にしたためGroupテーブルとUserテーブルと同じ関係性のようにGroupテーブルとUserテーブルの中間テーブルとしてMemoテーブルを作成した．

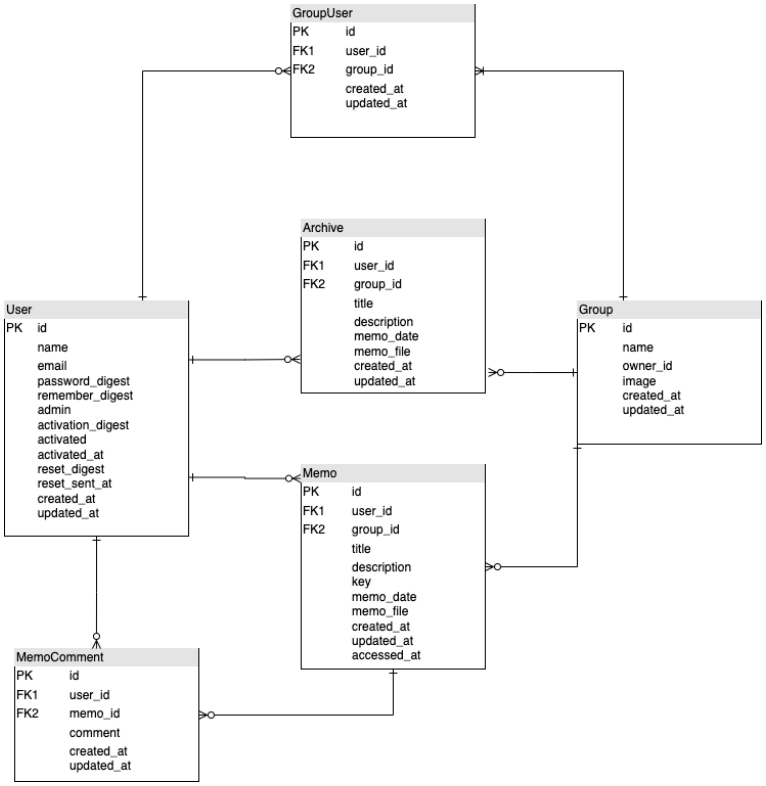

図 A.1: Webアプリの DB設計図.

A.2　グループ関連機能

A.2.1　グループ作成機能

```
class GroupsController < ApplicationController
  def create
    @group = Group.new(group_params)
    @group.owner_id = current_user.id
    @groups = current_user.groups
    if @group.save
      @group.users << current_user
      flash[:success] = "You succeeded in creating new group!"
      redirect_to @group
    else
      render "index", status: :unprocessable_entity
    end
  end

  private
    def group_params
      params.require(:group).permit(:name, :image)
    end
end
```

グループを作成するアクションは create メソッドで行う．group_params メソッドでは strong parameters[17] を明示している．strong parameter とはユーザーがモデルの重要な属性を誤って更新してしまうことを防止するための，より優れたセキュリティ対策のことであり，params メソッドでフォームから送信された情報を受け取り，require メソッドでパラメーターの中にモデルに対応するキーが存在するかを確認し存在する場合にそのバリューを返し，permit メソッドで保存するパラメーターの許可処理を行う．group_params メソッドは Groups コントローラの内部でのみ実行され，Web 経由で外部ユーザーに公開

する必要はないため，Ruby の private キーワードを使って外部から使えないようにする．create アクション内では，@group というインスタンス変数にフォームから送られてきた内容を元に Group モデルのインスタンスが生成されたものを代入する．そして，内容の保存が成功した場合に，グループの中に作成したユーザーを入れ，フラッシュメッセージを表示し，グループの詳細画面へと遷移する．失敗した場合には，エラーメッセージをグループ一覧画面，即ちグループを作成していた画面で表示させる．

A.2.2 メンバー招待機能

```
class GroupsController < ApplicationController
  def invite
    @user = User.find_by(email: params[:group][:email].downcase)
    @group = Group.find(params[:group_id])
    @members = @group.users
    # 招待したメールアドレスがユーザーデータベースに存在すれば招待メールを送信
    if @user
      GroupMailer.invite_member(@group, @user).deliver_now
      flash[:success] = "You succeeded in inviting new member!"
      redirect_back(fallback_location: group_member_path(@group))
    else
      flash.now[:danger] = "Invalid email or Not registered email"
      render "member", status: :unprocessable_entity
    end
  end

  def invite_member(group, user)
    @group = group
    @user = user
    mail(to: @user.email, subject: "Invitation to join the group")
  end
```

```
end
```

メンバーを招待するアクションは invite メソッドで行う．@user というインスタンス変数にはフォームから送られてきたメールアドレスを小文字に変換したものを代入している．そして，そのメールアドレスが登録されているユーザーがデータベースに存在するならば，そのユーザーにメールを通してグループ参加の招待メールを送り，存在しなければ，”Invalid email or Not registered email”というエラーメッセージをメンバー一覧画面に表示させる．

A.2.3　メンバー参加機能

```
class GroupsController < ApplicationController
  def join
    user = User.find(params[:user])
    reset_session
    log_in user
    @group = Group.find(params[:group_id])
    if @group.users.exclude?(user)
      @group.users << user
      flash[:success] = "You succeeded in joing a group!"
      redirect_to group_path(@group)
    else
      redirect_to group_path(@group)
      flash[:warning] = "You have already joined this group"
    end
  end

  def log_in(user)
    session[:user_id] = user.id
    session[:session_token] = user.session_token
```

```
  end
end
```

招待されたメンバーがグループに参加するアクションはjoinメソッドで行う．user変数にはメールのリンクから遷移したユーザー情報を格納している．そして，Rails標準メソッドであるreset_sessionを用いてセッション情報を一度削除してからlog_inメソッドを用いて新たなセッション情報を生成する．条件分岐ではグループ内にメールのリンクから遷移したユーザーが含まれていない場合には新たにグループ内にユーザーを追加するようにし，含まれている場合にはグループの詳細画面に遷移し，”You have already joined this group”というフラッシュメッセージを表示させる．

A.3　メモ関連機能

A.3.1　コメント機能

あるグループメンバーがメモを生成した後に補足情報を追加したい時，あるいは他のグループメンバーがそのメモに対して意見や質問をする際，再度メモを生成するのは利用するグループメンバーの負担が大きくなってしまうリスクがある．そこで，コメント機能を実装することによって，スムーズな情報共有が可能になり，結果としてアプリからユーザー離れが起こりづらくなる．

```
class MemoCommentsController < ApplicationController
  before_action :correct_memo_comment_user, only: [:destroy]


  def create
    @memo = Memo.find(params[:memo_id])
    @group = Group.find(params[:group_id])
    @comment = current_user.memo_comments.new(memo_comment_params)
    @comment.memo_id = @memo.id
    if @comment.save
      # ポストに関わった人たち全員にメールで通知をする
```

```
      NotificationMailer.comment_notification(@memo, @comment, @group).deliver_now
      flash[:success] = "You succeeded in creating new comment!"
      redirect_to group_memo_path(@group, @memo)
    else
      @memo_comment = @comment
      render "memos/show", status: :unprocessable_entity
    end
  end

  def destroy
    MemoComment.find(params[:id]).destroy
    flash[:success] = "comment deleted"
    redirect_to group_memo_path(params[:group_id], params[:memo_id]),
    status: :see_other
  end



  private
    def memo_comment_params
      params.require(:memo_comment).permit(:comment)
    end

    def correct_memo_comment_user
      redirect_to(root_url, status: :see_other) unless
      MemoComment.find(params[:id]).user == current_user
    end
end
```

コメントを作成するアクションはcreateメソッドで行う．memo_comment_paramsメソッドではgroup_params同様にstrong parametersを明示している．createアクション内では，

@comment というインスタンス変数にフォームから送られてきた内容を元に現在ログインしているユーザーが新規コメントを生成したものを代入する. そして，コメントの保存が成功した場合に，メモに関わったグループメンバー全員にコメント内容の通知が送信されるようにしている. 失敗した場合には，エラーメッセージをメモの詳細画面で表示させる.

```
class NotificationMailer < ApplicationMailer
  def comment_notification(memo, comment, group)
    @memo = memo
    @comment = comment
    @group = group
    array_email = []
    @memo.memo_comments.each do |memo_comment|
      array_email.append(memo_comment.user.email)
    end
  array_email.append(@memo.user.email)
    mail bcc: array_email, subject: "#{@comment.comment} from #{@comment.user.name}"
  end
end
```

上記のように，Rails に標準搭載されている Action Mailer を使用すると，アプリケーションのメーラークラスやビューで電子メールを送信できる. 引数として受け取ったメモとコメント，グループをそれぞれインスタンス変数に格納する. そして，メモに紐付けられているコメントユーザー全員のメールアドレスとメモを作成したグループユーザーのメールアドレスを array_email 配列に代入する. 最後に，複数のグループユーザーあてに電子メールを同時送信する際，受取人以外の送信先メールアドレスを伏せて送信することができるブラインドカーボンコピー (bcc) を利用して配列 array_email に格納されたメールアドレス宛に電子メールを送信する.

A.4　検索機能

A.4.1　時間軸検索

時間軸検索とは「超」整理法と呼ばれる新たな整理法の一種である．従来の整理法では，データや書類の内容から分類する「図書館方式」が一般的であった．つまり，整理とは分類であるという考えが古くから定着されている．しかし，従来の整理法にはいくつか問題が生じる．

1. どの分類項目に入れて良いか分からない問題

2. その他問題

まず 1. については，対象となるデータが，複数の内容または属性を持っている場合に，どの分類項目に入れて良いか分からなくなる．例として，pdf の資料をグループ全体に共有しようとした際に，pdf という項目に入れるのか，その pdf 資料の内容に関する項目に入れるのかという問題が生じてしまう．個人的にデータを管理するのならば，図書館方式でも良いかもしれないが，グループとしてデータを管理していくのならば，グループ 1 人 1 人が共通の分類法を身につけなければならない．これは非常に危険であり，非効率である．次に 2. についてはデータはどの分類項目に入らないものもある．この場合には,「その他」などといった分類項目に残しておくというのが，ごく常識的な対処であろう．しかし，これこそが最大の陥穽なのであり,「その他」はハードルが低く便利な分類項目だから，どんどんデータが入ってくる．その結果，とどまるところをしらず膨れ上がり，収拾がつかなくなる．また，データを分類した瞬間は正しい項目に入れておいたか覚えていたとしても，時間が経過すればどの項目に入れてしまったか忘れてしまうこともある．使用頻度が低いデータは特にそうである．そこで時間軸による検索は，極めて有効的である．理由として以下の 2 点が挙げられる．

1. 使用する書類，データの大部分は，最近使ったものの再使用である．

2. 人間の記憶は，時間順に関しては強い．

A.4.2　タグ検索

タグ検索は日付キーを忘れて時間軸検索ができない場合に用いることを想定する．図のように共有されたデータ一覧画面の検索欄からタグとなる単語を入力し該当するタグに紐付けられたデータを絞り込むことができる．また，1つ1つの投稿データのタグリンクをクリックすることによっても検索することができる．さらに動画のリンクや音声・画像ファイル共有した場合にはそれらの内容の特徴を表したタグを付けておくことによってデータの検索を容易にすることもできる．

図 A.2: タグでデータを絞り込む一連の動作．

A.4.3　キーワード検索

キーワード検索はタグ付けがされていないメモ，タグを忘れてしまったメモを検索する場合に用いることを想定する．タグ検索と同様データ一覧画面の検索欄からメモのタイト

ルを入力し単語がヒットすれば該当するデータを絞り込むことができる.

```
class GroupsController < ApplicationController
  def show
    @group = Group.find(params[:id])
    @memo = @group.memos.new
    @pagy, @memos = pagy(@group.memos.order(updated_at: :desc).limit(20))
    if params[:key_word]
      @pagy, @memos = pagy(@memos.where("title LIKE ?", "%#{params[:key_word]}%"))
      unless @memos.present?
 @pagy, @memos = pagy(@group.memos.tagged_with(params[:key_word]))
      end
    end
  end
end
```

key_word というパラメータを受け取った時にのみ，Rails の検索用メソッドである where と like 句を用いて受け取ったパラメータからタイトルにヒットするデータを絞り込み@memos というインスタンス変数に格納する． その後，@memos の中身が存在するかどうかを present メソッドで判定し，存在しなければ受け取ったパラメータからタグ検索をする.

A.5　アーカイブ機能

lib/tasks/create_archive.rake

```
namespace :create_archive do
  desc "アーカイブの定期実行プログラム"
  task add_archive: :environment do
    Group.all.each do |group|
      group.memos.each do |memo|
 if memo.accessed_at == nil ||
 (memo.accessed_at < 2.weeks.ago && memo.key == false)
```

```
    archive = group.archives.new(
            title: memo.title,
            description: memo.description,
            user_id: memo.user_id,
            group_id: memo.group_id,
            memo_date: memo.memo_date,
            memo_file: memo.memo_file
           )
    archive.tag_list << memo.tag_list
    archive.save
    memo.destroy
  end
      end
    end
  end
end
```

グループのメモ 1 件 1 件に対してアクセスされた日時を表す accessed_at が nil または 2 週間以上経過しており，key カラムが false の場合に新たに Archive インスタンスを生成し，保存する．そして，アーカイブに移動することになったメモは削除される．これにより，アクティブなデータとアーカイブデータを分離することが可能になる. これを毎日 0 時に定期実行されるようにバッチ処理として組み込む.

A.6 通知機能

```
class NotificationMailer < ApplicationMailer
  def notification_for_member(notification, group)
    @notification = notification
    @group = group
    mail bcc: @group.users.pluck(:email),
```

```
    subject: "#{@notification.title} from #{@notification.user.name}"
  end
end
```

上記のように，Rails に標準搭載されている Action Mailer を使用すると，アプリケーションのメーラークラスやビューで電子メールを送信できる．引数として受け取ったメモとグループをそれぞれインスタンス変数に格納する．そして，複数のグループユーザーあてに電子メールを同時送信する際，受取人以外の送信先メールアドレスを伏せて送信することができるブラインドカーボンコピー (bcc) を利用して電子メールを送信する．