Programming Paradigm - Object Oriented Programming

For this application we have opted to use Object Oriented Programming. Object oriented programming or OOP is a programming paradigm in which real world things are represented as objects in code. These objects consist of two things: attributes and methods.

Using a real world example, let us represent a bank account. So a bank account's attributes would be what make a bank account, it needs to have an account number, balance, maybe an account type, like checking or savings. A method would be a process that this object, the bank account performs such as depositing, withdrawing, showing balance.

```
class Account:
    # Initializing the parameters of the class
    def __init__(self, account_type, id, balance=0.0):
        # 'checking', 'savings', etc.
        self.account_type = account_type
        # Balance of the account
        self.balance = balance
        # Account ID
        self.id = id
        # Determines rates for checking and savings accounts
        self.rate = 1.03 if account_type == 'checking' else 1.05

    def deposit(self, amount):
        # Checks for valid deposit amount, must be positive
        if amount > 0:
            # Adds input to current balance
            self.balance += amount
        else:
            # Prints if not valid
            print(f"{error_color}Deposit amount must be positive.{r
```

Example of bank account object in Python.

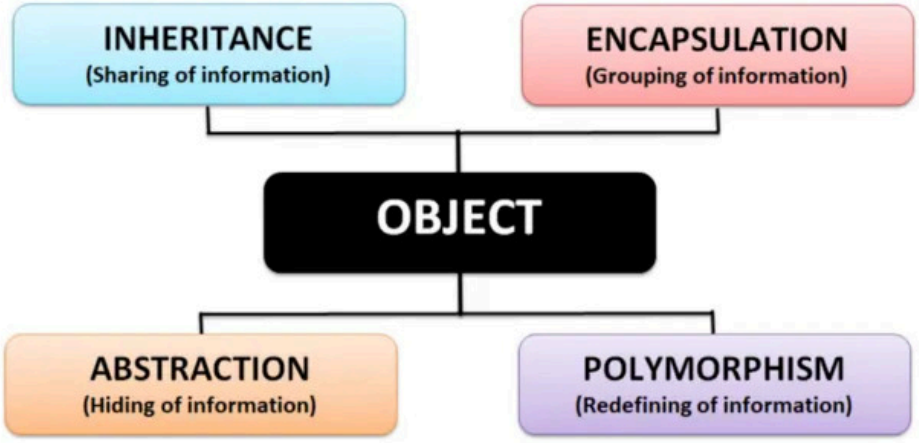
In the above image it shows a bank account represented as a blueprint of an object. Without going into too much detail, it simply contains attributes such as account type, balance, and id. It's methods it can perform are deposit and withdraw. All actions a bank account performs in the real world are represented as an object in code.

Four pillars of OOP

So as stated before objects in real life are represented in code such as the previous example above with the bank account. The representations of the object in code that outline the attributes and methods are called 'classes'. Classes are the blueprint of the object. Classes allow us to create 'instances' which are the objects created from the blueprint classes.

There are four pillars of writing OOP code we always must keep in mind:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism



Encapsulation

The principle of encapsulation states that objects should not be accessed or manipulated directly but through methods built into the object class. This is to avoid unwanted data manipulation, hence for safety we allow only methods in the object's class to manipulate the data. Methods that are used to access data are called 'getters' and methods used to manipulate data are called 'getters'.

```
class LoginAccount {
    constructor(username, password) {
        this._username = username // Private property using undersc
        this._password = password
    }

    getUsername() {
        return this._username
    }

    getPassword() {
        return this._password
    }

    changePassword(change) {
        this._password = change
    }
}

// Create a Login Account object
const newAccount = new LoginAccount("username1", "secret_password25

// Accessing properties through getter methods
console.log(newAccount.getUsername()) // Outputs: username1
console.log(newAccount.getPassword()) // Outputs: secret_password25

// Using setter methods to change properties
newAccount.changePassword("new_password")
console.log(newAccount.getPassword()) // Outputs: new_password
```

In the example show above we have a class LoginAccount which has private properties 'username' and 'password'. These properties are accessed using the methods defined in the class.

Abstraction

Abstraction states that only necessary information is shown to a user. Only information that is specifically requested should be exposed and the complex details are hidden.

Take for example a car, a function we can look at is starting the engine when you turn the key, you know that turning the key starts the engine, however you do not need to know about the complex parts of the car that work to start the engine. Or when you use a 'print' function or 'console.log' function you only know that the text will be displayed on the console but you do not know the functionality details of the functions.

Let us take a look at a basic example of abstraction using the force function [F = ma]

```
function ForceCalculate(mass, acceleration) {
    return mass * acceleration // The logic of the function is hidd
}

console.log(ForceCalculate(30, 9.81)) // Outputs: 294.3
```

Here the formula of the function is not shown only the result of it is. This is abstraction in a function object.

Inheritance

Inheritance is where a class can 'inherit' or take on the attributes and methods of another class. A class that inherits from another is called a 'child' class of the 'parent' class that it inherits from.

Let us say we have a range of vehicles, so we create a class called 'vehicles'. However, there are different types of vehicles such as buses, cars, bicycles. These vehicles are their own class but inherit attributes and/or methods. Cars, buses, and bicycles would inherit an attribute 'wheels' for example since all these vehicles have wheels.

Let us look at a coded example below:

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log(`Hello ${this.name}!`);
    }
}

// inheriting parent class
class Student extends Person { // extends keyword denotes that the

    constructor(name) {
        // call the super class constructor and pass in the name pa
        super(name);
    }
}

const student1 = new Student('Earvin');
student1.greet(); // Outputs: Hello Earvin
```

In the above example the Student class does not need to redefine the greet function as it inherits the function from the person class. Inheritance allows reusability of code

for child classes.

Polymorphism

Polymorphism simply means "having many forms", so in the context of programming it means an object being able to have different characteristics and functions. When a class inherits methods from a parent class the methods of the child class can be the same as the parent class but also be changed to suit different situations. One way to do this is through method overriding. Method overriding is when the child class implements it's own functionality over the parent class' method.

Let's look at an example:

```
class Animal {
    speak() {
      console.log("This animal makes a sound.");
    }
  }

  class Dog extends Animal {
    speak() {
      console.log("The dog barks."); // override the speak() functi
    }
  }

  class Cat extends Animal {
    speak() {
      console.log("The cat meows."); // override the speak() functi
    }
  }

  const animals = [new Animal(), new Dog(), new Cat()];

  animals.forEach(animal => animal.speak());
```

We see in this example the Dog and Cat child classes redefine the function of the same name, speak() from their parent class Animal.

App Architecture - MVC

The software architecture we have opted to use for this project is MVC - Model View Controller.

MVC separates an application into three components; the model, the view, and the controller. The model represents the data and business logic, the view is responsible for presentation of the data to the user, and the controller is the middleman between

the view and model, it updates the model according to input from the view but also does the opposite, updating the view to reflect changes in the model.

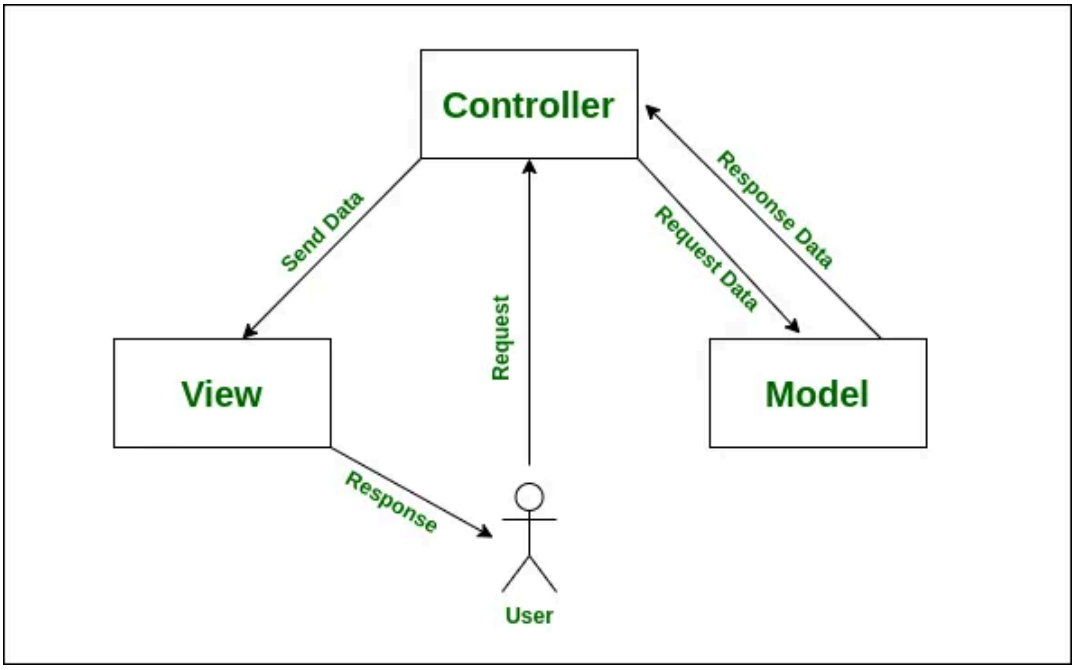


Diagram of how MVC works within an application.

Model

Additionally from representing the data a model will also be responsible for sanitizing and validating data. When a request is sent to the model from the controller if it is requesting to fetch data the model will request this data using a query language (we will be using mongodb) from the database the application is connected to.

```
const mongoose = require("mongoose");

// Make a schema with data properties
const UserSchema = new mongoose.Schema({
    email: String,
    password: String,
    username: String
});


// Make a model that uses the schema
//                              Name in DB, schema to use for its v
const UserModel = mongoose.model('User', UserSchema);

// Export the model
module.exports = {
    UserModel
}
```

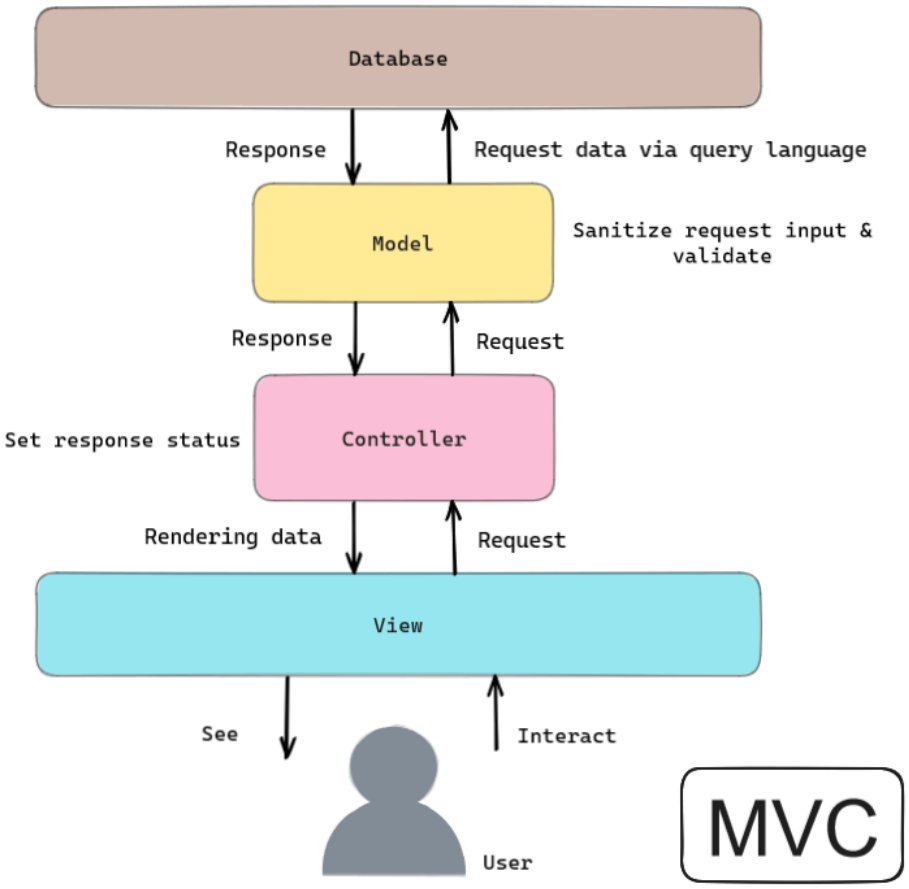
Above is an example of a model and schema for an entity in NodeJS using the mongoose library. The data will be represented in a MongoDB database. The model is responsible for checking what data goes in the database.

View

As the view is responsible for the presentation of data it uses frontend frameworks such as React to render the data requested from the database and allows users to input data if they wish.

Controller

The controller routes requests and responses from the user input in the view to either fetch or manipulate data from the database through the model. The controller additionally is responsible for send status codes, this can be for data not found, invalid, or forbidden in some cases. The controller is responsible for what is known as CRUD operations which deals with Creating, Reading, Updating, and Deleting data.



How data flows from Database to user and vice-versa via MVC.

Advantages of MVC

- Codes are easy to maintain and they can be extended easily.
- The MVC model component can be tested separately.
- The components of MVC can be developed simultaneously.
- It reduces complexity by dividing an application into three units.
- It supports Test Driven Development.
- It works well for Web apps that are supported by large teams of web designers and developers.

References

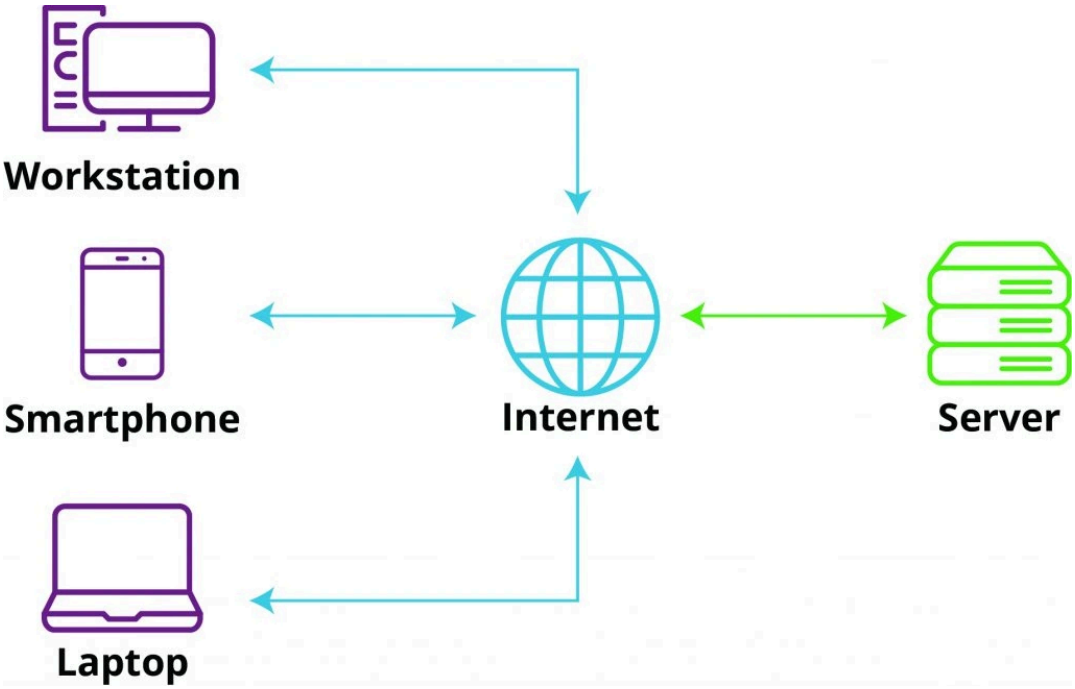
- https://levelup.gitconnected.com/principles-of-object-oriented-programming-33a08094afe6
- https://medium.com/@tayfunkalayci/abstraction-in-software-exploring-real-world-examples-b2e95d496bef#:~:text=One of the most prevalent,creating instances known as objects.
- https://logicmojo.com/inheritance-in-oops#:~:text=Inheritance is a technique of,are all utilised for transportation.
- https://medium.com/@kerimkkara/understanding-mvc-model-view-controller-with-c-and-net-5197bbbc7ed5
- https://www.geeksforgeeks.org/polymorphism-in-javascript/
- https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm#:~:text=The Model-View-Controller ,development aspects of an application.
- https://www.interviewbit.com/blog/mvc-architecture/

Explanation of how client/server architecture works

With the client/server architecture, the system is dived into two components of the front-end; the client that is the user-facing application e.g mobile app, web app or desktop app. These will then send requests to the server.

The server is the back-end that processes the requests and data that then sends the information back to the client.

This client/server relationship communicate over a network such as the internet using protocols like http/https.



Client Server Architecture Diagram

How it will work in our system

1. Client (Frontend)

The client is the application that users interact with. It could be:

- A mobile app which will be the main focus (e.g., iOS or Android).
- A web app (e.g., accessed through a browser).
- A desktop app.

The client handles:

- User Interface (UI): Displays book information, reviews, reading progress, recommendations, and activity feeds.
- User Input: Collects data from users (e.g., ratings, reviews, reading progress updates).
- Sending Requests: Sends requests to the server for data or processing (e.g., "Show progress for Book X" or "Submit a review for Book Y").

2. Server (Backend)

The server is the backend system that manages:

- **Data Storage**: Stores all data in a database (e.g., user information, book details, reviews, reading progress, recommendations).
- **Business Logic**: Processes requests from the client (e.g., calculating recommendations, updating reading progress, or generating activity feeds).
- **Authentication**: Verifies user credentials (e.g., during login or account creation).
- **APIs**: Exposes endpoints for the client to interact with (e.g., /login, /books, /reviews, /progress).

How Client and Server Interact

Step-by-Step Example: User Submits a Review

1. **Client (Frontend)**:

a. The user writes a review for a book and clicks "Submit."

b. The client collects the review data (e.g., rating, comment, book ID, user ID).

c. The client sends an `HTTP POST request` to the server (e.g., to the /reviews endpoint).

2. **Server (Backend)**:

a. The server receives the request and validates the data (e.g., checks if the user and book exist).

b. The server processes the request by saving the review to the database (e.g., in the Review table).

c. The server updates related data (e.g., recalculates the book's average rating).

d. The server sends a **response** back to the client (e.g., "Review submitted successfully").

3. **Client (Frontend)**:

a. The client receives the response and updates the UI (e.g., displays a success message and shows the new review).

Another Example: User Views Their Reading Progress

1. **Client (Frontend):**

a. The user navigates to their reading progress page.

b. The client sends an `HTTP GET request to the server` (e.g., to the /progress endpoint with the user ID).

2. **Server (Backend):**

a. The server retrieves the user's reading progress data from the database (e.g., from the `ReadingProgress` table).

b. The server sends the data back to the client in a structured format.

3. **Client (Frontend):**

a. the client receives the data and displays it in the UI (e.g., shows a progress bar or list of books being read).

Key Components of Client/Server Architecture in Your System

1. **Database**

Stores all the entities and relationships defined in your ERD:

- `User`, `Book`, `Review`, `ReadingProgress`, `Recommendation`, `Friendship`, `Category/Genre`, `ActivityFeed`.
- Managed by the server, which reads from and writes to the database.

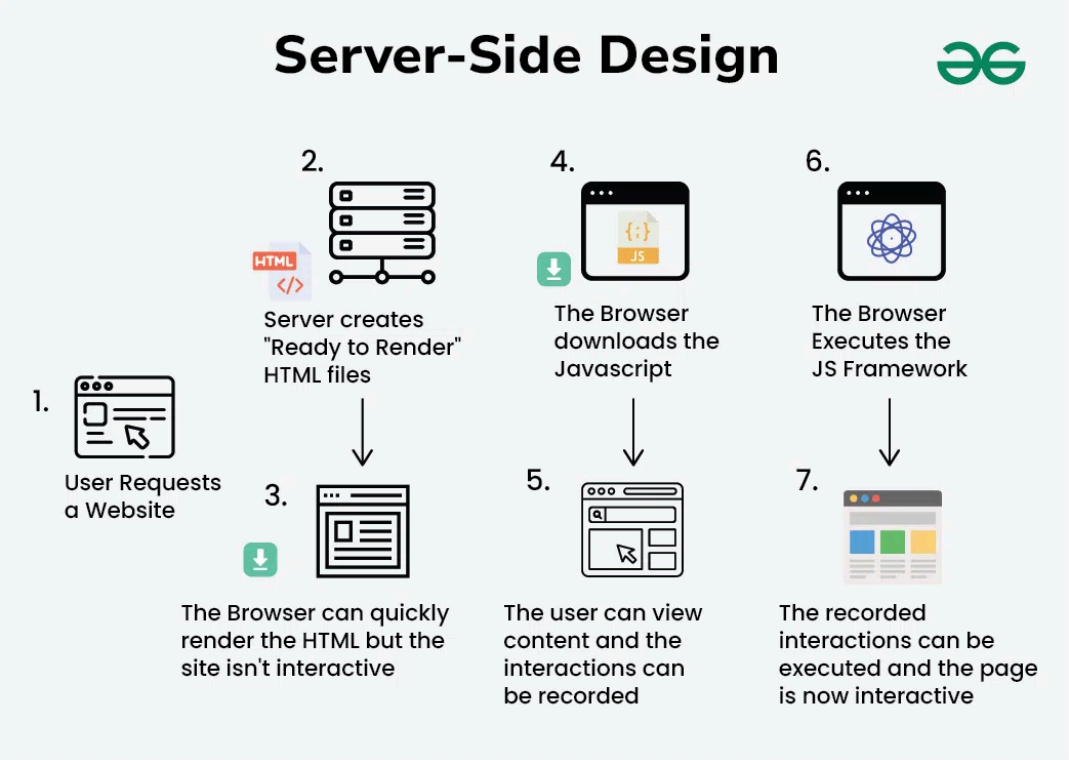
2. **APIs (Application Programming Interfaces)**

- The server exposes APIs for the client to interact with. Examples:
    - **Account Creation & Authentication:**
        - `POST`/ signup (create a new user).
        - `POST` / login (authenticate a user).
    - **Book Rating System:**
        - `POST` /reviews (submit a review).
        - `GET` /reviews (fetch reviews for a book).

- **Reading Progress Tracker:**
  - POST /progress (update reading progress).
  - GET /progress (fetch reading progress).
- **Recommendations:**
  - GET /recommendations (fetch personalized book recommendations).
- **Social Features:**
  - POST /friends (follow a user).
  - GET /activity (fetch activity feed).

3. **Network Communication**

- The client and server communicate over the internet using HTTP/HTTPS.
- Data is typically exchanged in JSON format (e.g., for reviews, progress updates, recommendations).



Client Server Architecture - System Desgin

Benefits of Client/Server Architecture

1. **Scalability:**

a. The server can handle multiple clients simultaneously (e.g., thousands of users accessing the app).

2. **Centralized Data Management:**

a. All data is stored and managed on the server, ensuring consistency and security.

3. **Separation of Concerns:**

a. The client focuses on the UI and user experience, while the server handles data processing and storage.

4. **Security:**

a. Sensitive data (e.g., passwords) is stored on the server, not the client.

b. Authentication and authorization are managed by the server.

Introduction

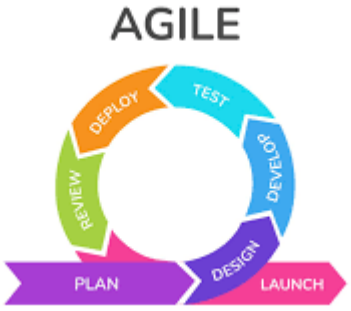
This research investigates how Agile and Scrum techniques, along with Trello for task management, were used in the creation of a book application. Account setup, book rating, reading progress monitoring, suggestions, and social interactions are some of the app's features.

Agile Methodology

Agile is a project management approach that focuses on flexibility, ongoing improvement, and producing high-quality results. It consists of breaking projects down into brief periods of work and often reassessing and adapting strategies. This procedure is repeated to guarantee that the product satisfies the expectations of its consumers and can adjust to new demands as they arise.

The following are some important practices of Agile:

- Sprints: These are brief, well-defined cycles (often lasting one to two weeks) that focus on the development of features of the application.
- User Stories: Each feature of the software is broken down into small, doable tasks that can be done in a single sprint cycle.
- Daily Stand-ups: These are short meetings that take place every day, during which team members share updates on their progress and discuss their future moves.
- Retrospectives: These sessions take place at the end of each sprint to discuss what went well and to determine areas where improvements may be made.

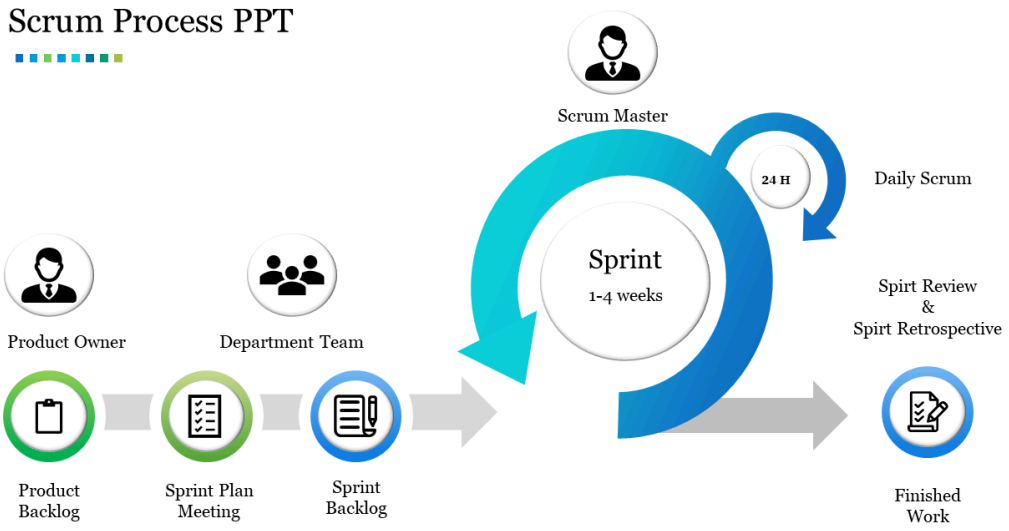


Scrum Methodology

Scrum is an Agile framework that organizes software development into cycles called Sprints, which typically run between two and four weeks. At the beginning of each sprint, the team holds a planning meeting in which they select items from a backlog that they think they will be able to do within the sprint.

The following are the components of Scrum that are utilized:

- Product Backlog: This is a set of project needs that have been prioritized and presented as user stories. For example, "As a user, I want to rate books so that I can influence the recommendations I receive."
- Sprint Planning: This is the process of determining which features will be developed in the next sprint. For example, in Sprint 1, the authentication system will be put up.
- Daily Scrum: These are short meetings that take place every day to talk about how things are doing and any problems that have come up.
- Sprint Review: At the end of each sprint, the work that has been accomplished is shown to stakeholders so that they may provide their input.
- Sprint Retrospective: This is a meeting that takes place at the conclusion of each sprint to reflect on what went well and what may be improved for the next sprint.



Task Management Tool: Trello

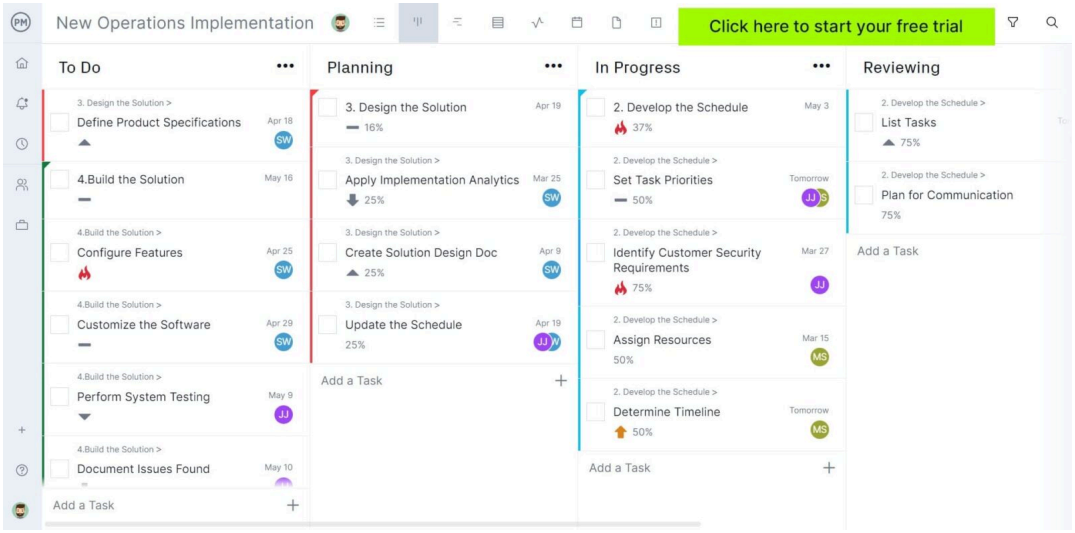
Trello is a tool that helps you graphically organize and keep track of development activities by using a system of boards, lists, and cards.

This is the way Trello is set up:

- Boards: These are created for various stages of the project, including Development, Testing, and Deployment.
- Lists: These show the several stages of task completion, such as To Do, In Progress, Review, and Done, and illustrate how tasks are progressing.
- Cards: These are individual tasks or user stories that are moved around the lists as they are being worked on and completed. Scenario

The following is an example of how Sprint Planning and Trello may be used together in a project:

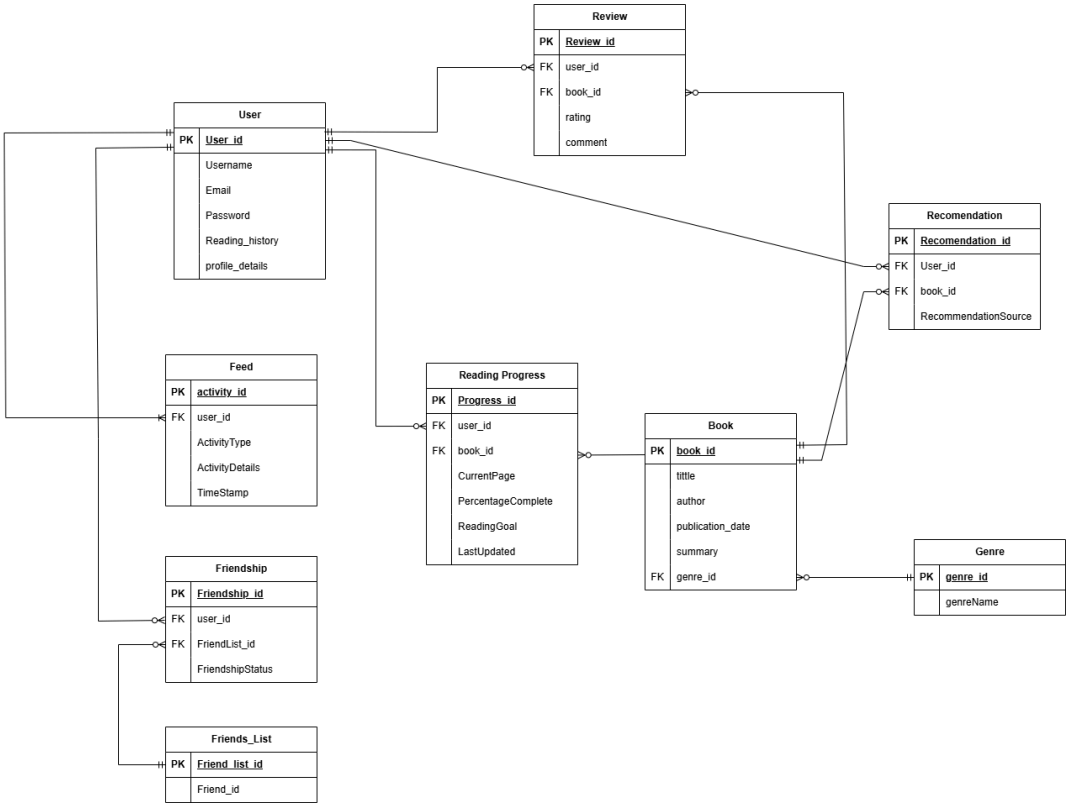
- Sprint Planning: The team selects user stories from the Product Backlog, such as "Implement email authentication for account creation," and transfers them to the Sprint Backlog.
- Trello Setup: Each user narrative that is picked is represented by a card, which is originally placed in the "To Do" column on our Trello board.
- Daily Scrum: Each day, team members update the status of their cards, moving them to "In Progress," "Review," or "Done" as they complete their duties.
- Sprint Review: At the end of the sprint, the team goes over the cards that have been finished to make sure that the work fulfills the acceptance criteria that were established.
- Retrospective: The team talks about possible improvements for the upcoming sprint and adjusts the Trello lists and cards based on this feedback.



References

- https://asana.com/resources/agile-methodology
- https://www.atlassian.com/agile
- https://www.coursera.org/articles/what-is-agile-a-beginners-guide
- https://www.geeksforgeeks.org/what-is-agile-methodology/
- https://trello.com/templates/engineering/scrum-board-dFzygb01
- https://www.atlassian.com/blog/trello/how-to-scrum-and-trello-for-teams-at-work
- https://www.scrum.org/resources/what-scrum-module
- https://aws.amazon.com/what-is/scrum/

ERD


Review
PK Review_id
FK user_id
FK book_id
rating
comment
User
PK User_id
Username
Email
Password
Reading_history
profile_details
Recomendation
PK Recomendation_id
FK User_id
FK book_id
RecommendationSource
Feed
PK activity_id
FK user_id
ActivityType
ActivityDetails
TimeStamp
Reading Progress
PK Progress_id
FK user_id
FK book_id
CurrentPage
PercentageComplete
ReadingGoal
LastUpdated
Book
PK book_id
tittle
author
publication_date
summary
FK genre_id
Genre
PK genre_id
genreName
Friendship
PK Friendship_id
FK user_id
FK FriendList_id
FriendshipStatus
Friends_List
PK Friend_list_id
Friend_id

User Stories

User Story Example

"**As a** frequent reader who loves exploring new books,
I want to receive personalized book recommendations based on my reading history and preferences,
so that I can easily discover books I'm likely to enjoy without spending hours searching."

1. **User/Persona**

- "Frequent reader who loves exploring new books."
- This persona represents a user who is actively engaged in reading and values discovering new books.

2. **Need/Justification:**

- "I can easily discover books I'm likely to enjoy without spending hours searching."
- This highlights the user's need for a more efficient way to find books tailored to their interests.

3. **Want/Feature/Function:**

- "Receive personalized book recommendations based on my reading history and preferences."
- This describes the specific feature the user wants, which is a recommendation system powered by their activity and preferences.

How This User Story Translates to Development

- **Feature:** Implement a recommendation engine that analyzes the user's reading history, ratings, and preferences to suggest books. (e.g book rated over 3 stars will be generate similar books)
- **Backend:** The server processes the user's data and generates recommendations using algorithms (e.g., collaborative filtering or content-based filtering).
- **Frontend:** The client displays the recommendations in an easy-to-navigate section of the app (e.g., a "Recommended for You" carousel).

"**As a** busy professional who struggles to find time to read,
I want to set reading goals and track my progress,

so that I can stay motivated and ensure I’m making time for reading."

1. **User/Persona:**

- "Busy professional who struggles to find time to read."
- This persona represents a user with limited free time who values structured reading habits. (e.g reading goal of 5-10 pages a night)

2. **Need/Justification:**

- "Stay motivated and ensure I’m making time for reading."
- this highlights the user’s need for a tool to help them prioritize reading in their busy schedule. (e.g reward pop-up congratulating for reaching goal)

3. **Want/Feature/Function:**

- "Set reading goals and track my progress."
- This describes the specific feature the user wants, which is a reading progress tracker with goal-setting functionality.

How This User Story Translates to Development

- **Feature**: Implement a reading progress tracker that allows users to set goals (e.g., "Read 20 pages per day") and track their progress (e.g., percentage completed, pages read).
- **Backend:** The server stores the user’s goals and progress data in the database and calculates metrics like completion percentage.
- **Frontend:** The client displays the progress visually (e.g., a progress bar or checklist) and sends reminders or notifications to keep the user on track.

Account Creation and Authentication

User Story: As a new user, I would like to be able to simply create an account by using my email address, social media, or a username and password. This would allow me to access my personalized reading environment in a secure manner.

- **Need/Justification:** When it comes to user adoption and confidence, security and simplicity of access are extremely important factors. In order to respond to the preferences of users and improve accessibility, providing different sign-up alternatives is essential.
- **Want/Feature/Function:** It should be possible for users to sign up for the app and log in using their email addresses, social media accounts, or a username that is accompanied by a password.

- **Development Translation:** The use of OAuth for social media integrations, a secure email verification mechanism, and encrypted password storage are all desirable features. Create an authentication module for users that can accommodate a variety of different login methods.

Book Rating System

User Story: Being a bookworm, I want to evaluate books I have read using a 1-5 rating system and give reviews so I may share my thoughts with the community and impact book suggestions.

- **Need/Justification:** Users are able to browse and express their thoughts on books through rating systems, which helps to cultivate a community of readers and provides assistance with decision-making capabilities.
- **Want/Feature/Function:** In addition to being able to post written reviews, users are also able to evaluate books on a scale ranging from one to five stars.
- **Development Translation:** Create a database schema that includes user IDs, book IDs, ratings, and reviews. Implement front-end components for users to submit ratings and reviews and backend logic to calculate average ratings and display reviews.

User feed/Activity stream

User Story: As a reader who likes to discuss books with friends, I would like to be able to see what my friends are reading and recommending, as well as what they rate their books. This would allow me to seek other options of books and know what my friends are interested in too.

- **Need/Justification:** To fulfil the social aspect of this app, allowing users to see their friends activity is a must-have feature.
- **Want/Feature/Function:** Users should be able to see recommended books from friends and rating of books.
- **Development Translation:** We use the engine developed to analyze a user's reading history, ratings, and preferences to suggest books. However we now make it public for the user's friends so that recommendations come through a feed. We also allow friends data to be seen by the user. This data will all be requested from the server. It will be displayed to the client randomized to similar to other social media feeds.

Ethical Principles

1. Accessability

We want to make sure that this application is accessible to everyone. Some common feature we will implement will include:

- Alternative text for images or visuals. This allows people to understand what is meant to be shown in case an image or visual aid does not load properly on the web-page for any reason
- We will ensure the application is keyboard friendly but also touch screen friendly: we want as many devices as possible able to use our app, the exception perhaps only smart watches or any with very small screen.
- It would be great to allow a large number of users to test the features however this is a very small scale project it will not have as much resources to allow for such a large number of user testing.
- We will strive to make forms inclusive taking into account all user needs for the UI.

2. Responsive design

We will strive to make the application responsive to as many device types as possible.

- We will be following the bootstrap media breakpoints. Media breakpoints are the points at which the applications User interface will change view based on screen size (width in px).
    - For mobile view: 576px and below
    - For tablet view: 768px to 992px
    - For desktop view: 992px and above (for any devices larger as well such as TV or multi-monitor set ups)

3. Security and Authorization The following steps will be taken to keep user security:

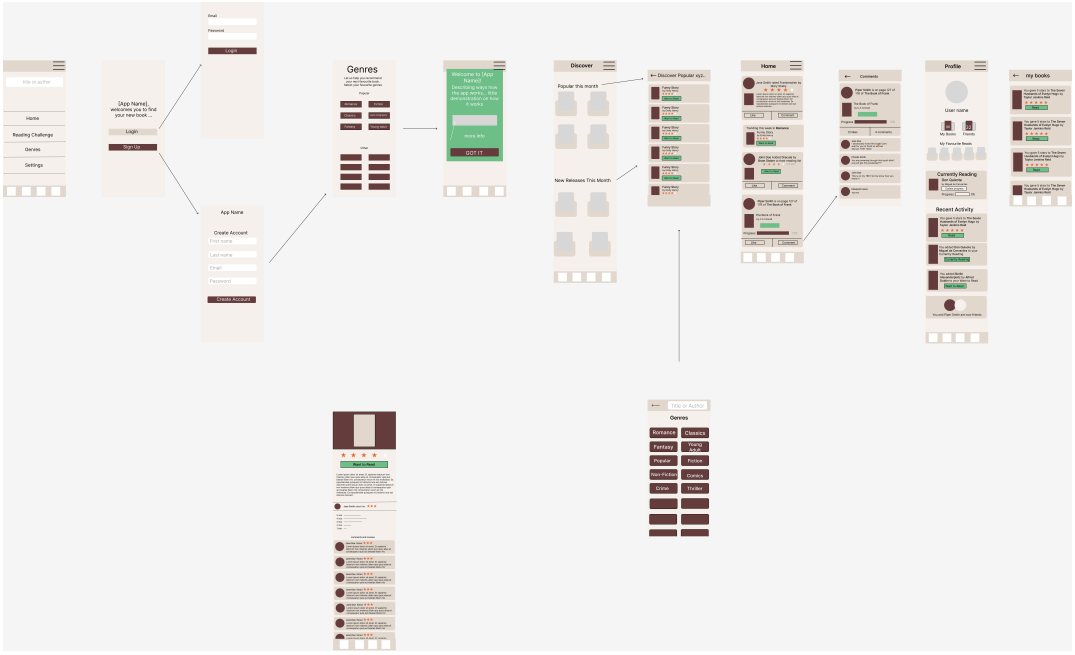
    - As our application has user login features to protect information it requires password encryption and authentication.
    - To store passwords we will make sure to hash them.
    - We will consider only accepting from users 'strong' passwords which contain special characters numbers and lower/upper case letters
    - Styling the password input field so that a password is not shown in plain text on the user interface while typing it in.

4. Documentation

Documentation is important for allowing our fellow peers in web development to understand the workings of our app. But documentation is not just for our peers but potential users looking to test or use our app.

- Once we start developing we will document more in a README.md file in out GitHub repository.
- Additionally we will have frequent commits on our GitHub to document the whole development stage.
- Every relevant object, whether class or function or block of code will contain doc strings, comments, and notes. Again this is for fellow developers to look at and understand our code.
- Our ERD, Wireframes, User stories all give different insight to our application's design process.

Mobile size wireframes [576px and below]



Tablet size wireframes [576px to 992px]

Desktop size wireframes [992px and above]