Story books

1. Application description (2)
   1. Introduction
   2. Story book
   3. Author experience
   4. User experience
2. Architecture choice (3)
3. Application modules (4)
4. Technical description (8)
5. Application server API (9)
   1. Books
   2. Catalog
   3. Identity
   4. Basket
   5. Orders
   6. Play
6. Client applications (12)
   1. Ionic applications
   2. Admin application
7. Development timeline (12)
   1. Initial development
   2. Language support improvements
   3. Purchase changes and new policies
   4. Introduce complexity to gameplay

Application description

Introduction

Story books is application about interactive book reading. Its main goal is to make reading interesting and entertaining for the young audience. Each person will have the opportunity not only to read about his favorite character but to navigate him trough his story. Each discission will lead you to different story branches with positive or negative effects. You are challenged to reach the end without facing early death or dead end. This application will give more freedom to the authors, chance to expand their fan bases, sell or give for free their stories and receive feedback from their buyers. Users will have the opportunity to increase their libraries, knowledge and achievements.

**What is story book?**

Story book is interactive book that feels like playing game. At the beginning each player has some starting predefined stats. They will change and are the main factor of how well are you making decisions and progress through the book. Each book is consisted of chapters like normal book. The main difference is that that at the end of every chapter, the user will be given choices and he have to make a decision how his story will continue. At reaching of some chapters there will be set define effects over the user stats. They can be positive and negative. If the user accumulates enough negative decisions the story ends for him. The goal for the user is to reach the end of the story. Some chapters can be marked as checkpoints and this is something like save game. This way the user will restart at the last checkpoint and not from the beginning. Each book can be purchased from the store and added to the users library.

Author experience

Each user can become an author. He will have creative space where he can write his stories. If he wants, he can publish his books and they will be available in the book store. Each user can add them to their libraries. If the book has price, small percent of the purchase will be taken by the application. Each user can subscribe or become a fan of given author. Also, each book can receive reviews. The author can read reviews from his fans and see stats about his books in a report page.

User experience

Each free guest will open the home page consisting of the most popular books. Guest can see short description about the book and will have the option to create account in order to purchase or play the story book. In user profile page he will be present with library section with all added books and their progress. Book details will contain start/continue play controls, progress, short description of the book, achievements and link to the store page.

Architecture choice

There were two choices for application architecture:

* Domain driven design with clean architecture
* Vertical slices architecture

I have decided to get the best of the mentioned above. We will have domain driven architecture with bounded contexts separated in vertical slices.

Each slice will consist of all necessary layers for its autonomous function. That means that the full stack will be included as follows:

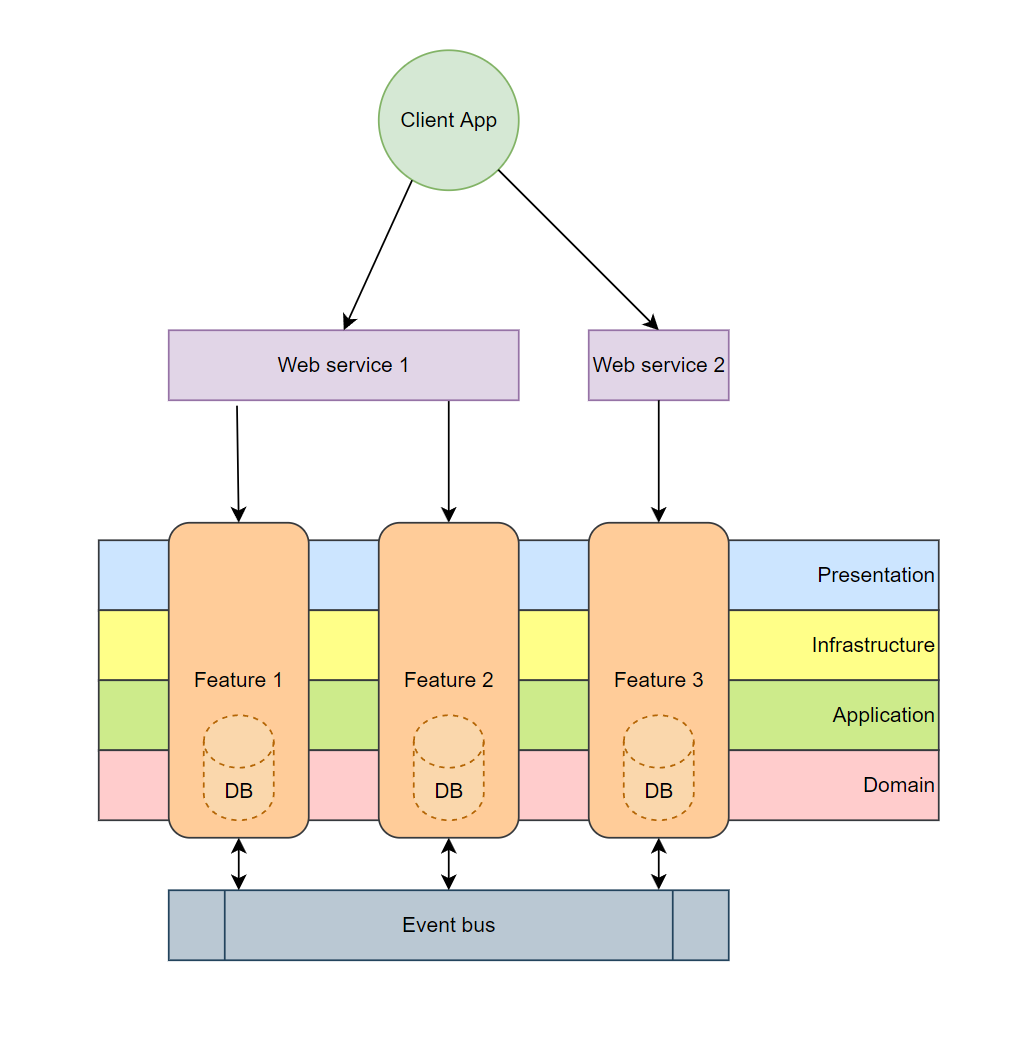
Data layer, Domain, Application, Infrastructure, Presentation, Configuration.

DDD will be preserved as much as possible within each slice. Infrastructure layer can be separated into Services and Persistent layers.

Slices (also called features or contexts) are just library projects distributed across one or many web services. We can start with monolith approach. We will have one API gateway that will be web project referencing all of the slices. They will communicate with each other with event bus, even if they are part of the same web service. This will help with future separation into multiple microservices.

We should consider each slice to be independent and ready to become its own microservice. If it is simple, small and well defined, it can be a single project.

For client application I think that is good idea to use web framework with possibility of using it in hybrid mobile platform. This way we will achieve maximum user coverage and we won’t depend on their preferred hardware and OS.



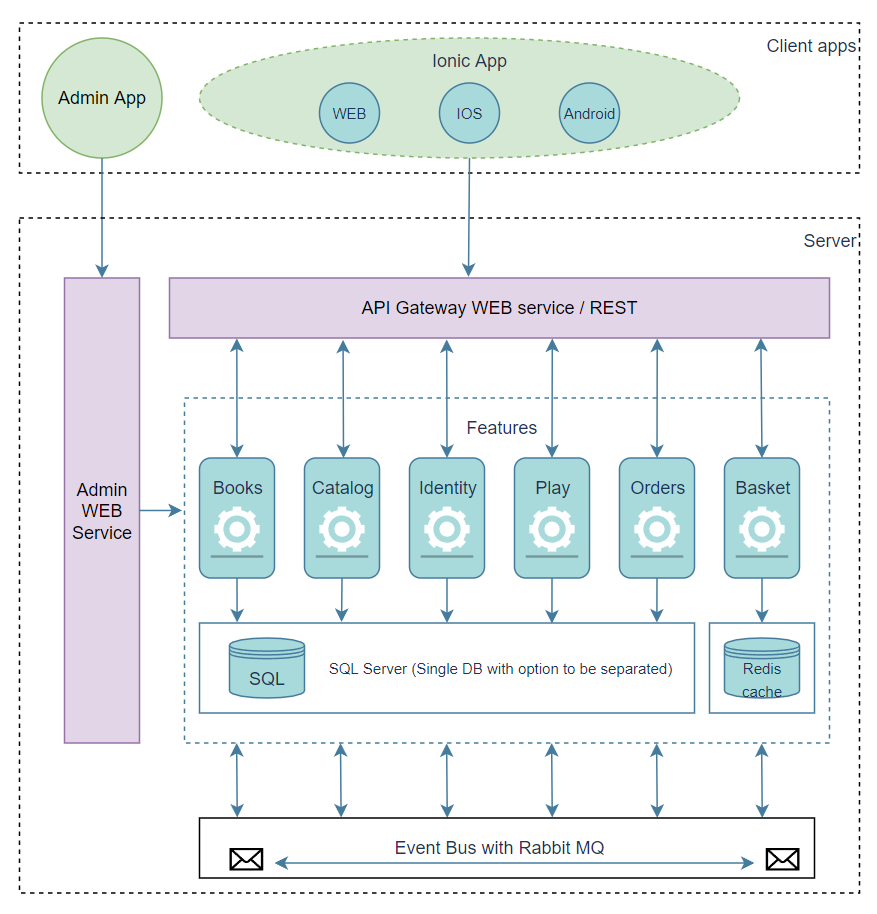
Application modules

Based on the description we can define several independent modules:

* Identity
* Catalog
* Basket
* Orders
* Books
* Play
* Payment
* Logging

These modules are logically separated but, in the beginning, will be part of one single web service. If everything is implemented according to the selected architecture if future separation is required, it will be only matter of configurations and data migrations.

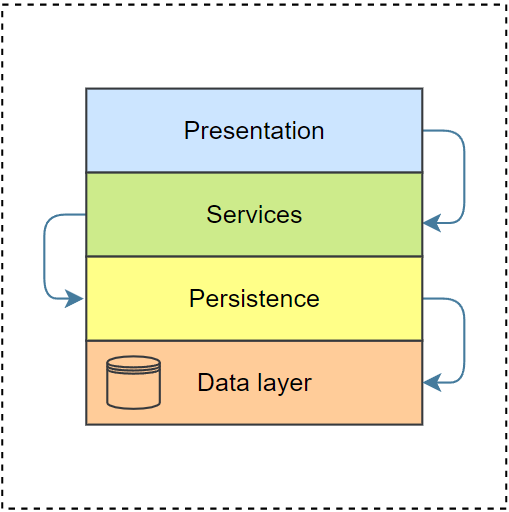
Database also will be shared across all modules in the beginning. If we keep each modules data in separate context, we will be fine for future separation if needed. Home page, basket and other data that can be cached will be stored in Redis DB. This will help with performance and the first-time loading time to be optimal. See below the diagram for the core modules:



**Identity module**

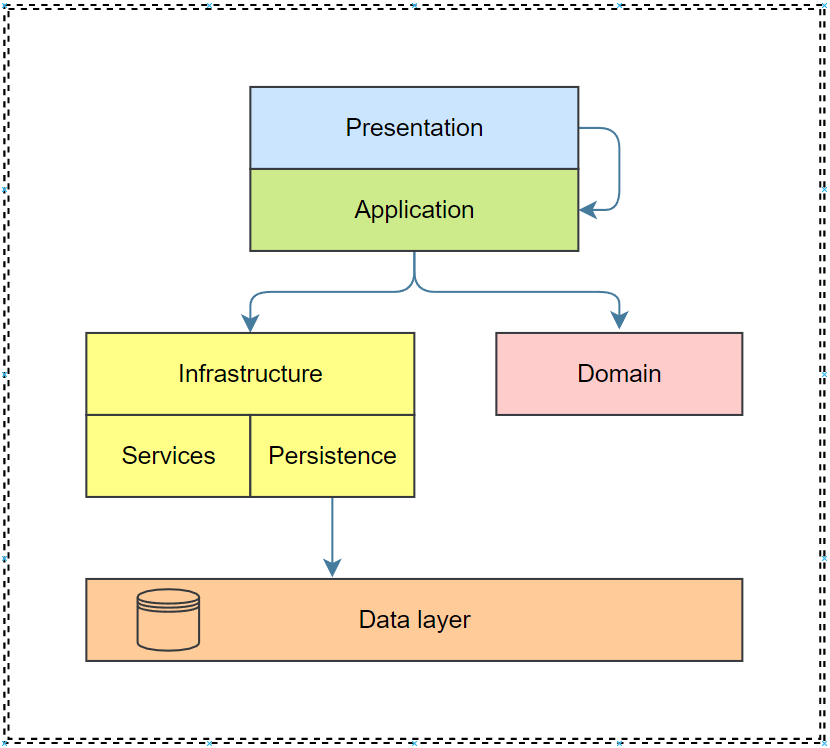
This module will be responsible for registering and managing user accounts and data related to sign up and register workflows. User should be able to register, login, change his password, enable two factor authentication and sign up with Facebook. In future can be added other login providers but this is the most famous and most used one. All possible actions will be documented in the API description.

Technology stack will rely on the .NET platform. Database will be SQL Server and we will use the identity library from .NET SDK. This way we will have needed user account objects and their management. We do not need cache or any other optimizations. Structure:



**Books/Authors module**

This module will store all the data of the story books that each author is writing. Author will be able to create story book and apply different modifications to it. For example: setting user stats, adding chapters, setting chapter choices, choice effects, etc. This module will store only the data related to the story book that defines its content and way to be played.

Technology stack will be again .NET platform. The complex data structure will need relational database. The choice is the same as in the identity module: SQL server. This layer will have more complex structure in order to maintain all of the complexity that may expand with new features in the future. Structure:

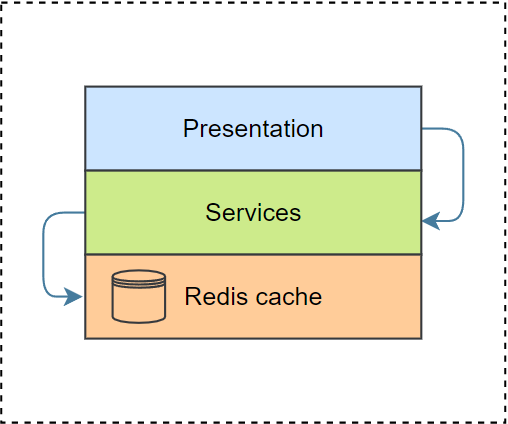
**Catalog module**

This module will be responsible for listing all published story books to the end user. This will be the shop part of the project. This will be the home page of our application. Most popular books will be shown on the first row. On the next rows we will see most popular books separated by genre. This will be similar to HBO GO and Netflix home pages. From home page each user will be able to open general search page will full books list and many filters.

This module again has complex objects with relations between them. The structure will be the same as the one in the ‘Books’ module.

**Basket module**

Basket module will have the purpose to save the baskets/shopping carts of every user. This will be relatively small and simple module. Data should not be persisted for long time. .Net platform will be used but with combination of Redis cache DB instead of SQL server.



**Orders module**

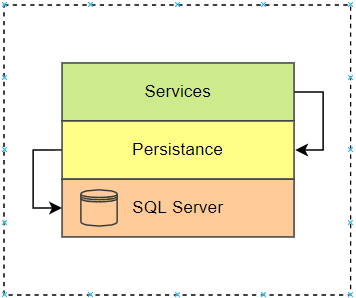
This module takes after basket workflow to finish and track order history of every user. Orders module also will show data for authors and their sales statistics. This is another example of more complex feature and workflow. Domain driven concepts will be user to their full extend. You can see the diagram for Books module for reference. Same stack of .NET and SQL server will be used here. This module will work tightly with the payment module. In order to complete order and transfer money to the author and the application cut. They can be united in one single module but we will talk about them as separate entities and contexts.

**Payment module**

For the payment module, the most important thing will be the simple, fast and securely execution of each payment. We should choose payment provider and to log every user and external interaction. We have two choices for payment provider:

* PayPal
* Sage Pay

Technology stack still remains .NET. Presentation layer can be skipped. Successful payments will take effect into the orders module. We will log all possible data related to the payments here and not only in the logging module. SQL server will be preferred here for better structuring of the payment, user and external data.



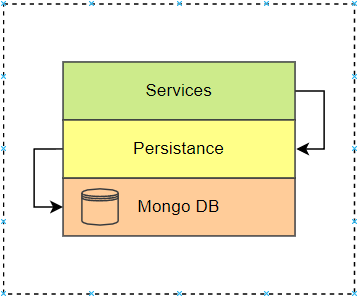
**Play module**

In this module we have the main functionality of the application. This can be defined as the core module this brings the main idea behind the app. Here users will interact with books from their libraries. We need to track progress, user stats and achievements for every book. There will be a lot of communication between client and serve apps. We need to be sure to save all user activity in order to maintain their progress.

Tech stack is the same as in Orders and Book modules: .Net + SQL server. Full workflow will be described in API description and workflows section.

**Logging module**

This module will be responsible for tracking all essential data related to users and authors interactions, errors and communication with external APIs. We will record:

* ****Each write/edit/delete request from the users
* Payment requests
* Unhandled errors occurred in the application pipeline
* Communication with external services and their responses
* Performance issues

Every module will send data to this and at moment the load can be very big. We will use Mongo DB for storing all of this data fast and easy. We do not have transaction or relations that are required. We just have a lot of row data. Also, we will need to take into account and the retention period of each stored document type and archive older data in separate database.

Technical description

In this section will be define main part of the tech stack, used libraries and technologies. There is and big chunks of the application that should be researched and analyzed how they will be integrated into the application. Let’s look at our slices structure layer by layer.

**Domain**

Domain layer should be the most inner one. It does not depend on any other layer. Here is stored all domain models with their create, update, validate logic. For every slice we have one bounded context. Entity creation should be executed only from factory classes. Public or internal setters should not be used. Each change should be performed through well named method with action from the real world. It is a good idea to research using strongly typed ids for each entity. This way we will decrease possibility of using wrong identifiers.

**Application**

Application layer will define all business actions related to the business logic and the domain entities. Clean architecture pattern should be preserved. CQRS pattern will be implemented with the help of ‘MedatR’ library. All in/out request will go through mediator handler. Here we can attach any middle wares for tracking performance, error handling, logging information.

Validation middleware will use use fluent validation library for easier and better model validation. With this we have separated validation logic from the rest. Mapping between object will be handles by Auto-Mapper library.

**Infrastructure**

This layer consists of few smaller parts. Persistence, services and any other logic that does not fit in domain and application layers.

For persistence we will use Entity framework to make calls to the SQL server database.

**Deployment**

We will use to Docker for containerizing our application. At first there will be few containers:

* Web API project containing initially all of the features
* Client Web application
* SQL Server
* Mongo DB
* Rabbit MQ
* Redis DB

For automation and orchestration, we can use Kubernetes.

Application server API

**Authors**

Path:/api/v1/author/{id}

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| Get book full details | GET /book/{bookId} | 200 OK 404 Not Found |
| List books by author | GET /books?name=&genre… | 200 OK 400 Bad Request |
| Add new book | POST /book | 201 Created 400 Bad Request |
| Update book details | PUT /book/{bookId} | 200 OK 400 Bad Request 404 Not Found |
| Remove book | DELETE /book/{bookId} | 200 OK 404 Not Found |
| Add chapter | POST /book/{bookId}/chapter | 201 Created 400 Bad Request |
| Update chapter details | PUT /book/{bookId}/chapter/{chapterId} | 200 OK 400 Bad Request 404 Not Found |
| Remove chapter | DELETE /book/{bookId}/chapter/{chapterId} | 200 OK 404 Not Found |
| Add user stats | POST /book/{bookId}/user-stats/ | 201 Created 400 Bad Request |
| Update user stats | PUT /book/{bookId}/user-stats/{chapterId} | 200 OK 400 Bad Request 404 Not Found |
| Remove user stats | DELETE /book/{bookId}/user-stats/{chapterId} | 200 OK 404 Not Found |

**Catalog**

Path: /api/v1

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| Get books for the home page | GET /catalog/home | 200 OK |
| Books search page | GET /catalog/story-books?filters=value… | 200 OK 400 Bad Request |
| Get book details | GET /catalog/story-book/{id} | 200 OK 404 Not Found |

**Identity**

Path: /api/v1

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| User details | GET /account | 200 OK 404 Not Found |
| Register | POST /account | 201 Created 400 Bad Request |
| Update user details | PUT /account | 200 OK 400 Bad Request 404 Not Found |
| Disable user account and delete its history | DELETE /account | 200 OK 404 Not Found |
| Login user | POST /account/login | 200 OK 400 Bad Request 404 Not Found |
| Change password | PUT /account/manage/password | 200 OK 400 Bad Request 404 Not Found |
| Confirm email | GET /account/manage/email | 200 OK 404 Not Found |

**Basket**

Path: /api/v1

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| Create basket | POST /basket | 201 Created 400 Bad Request |
| Get basket details | GET /basket/{id} | 200 OK 404 Not Found |
| Add book to the basket | PUT /basket/{id} | 200 OK 400 Bad Request |
| Remove book from the basket | DELETE /basket/{id} | 200 OK 404 Not Found |

**Orders**

Path: /api/v1

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| Checkout and create new order | POST /oder | 201 Created 400 Bad Request |
| Get order details | GET /order/{id} | 200 OK 404 Not Found |
| Change order details | PUT /order/{id} | 200 OK 400 Bad Request 404 Not Found |
| Update payment status | PUT /order/{id}/payment | 200 OK 400 Bad Request 404 Not Found |
| Get orders history per user | GET /orders/{userId} | 200 OK 404 Not Found |

**Play**

Path: /api/v1/play

|  |  |  |
| --- | --- | --- |
| Functionality | Path | Return Codes |
| Start new play session | POST /session | 201 Created 400 Bad Request |
| Get session current progress | GET /session/{id} | 200 OK 404 Not Found |
| Make progress to next chapter and update session progress | PUT /session/{id} | 200 OK 400 Bad Request 404 Not Found |
| Reset session progress | PUT /session/{id}/reset | 200 OK 400 Bad Request 404 Not Found |

Client applications

In order to reach maximum number of users we need to cover the maximum number of platforms. The good part is that the user interface will be relatively simple and we can create single web application with Ionic. From there we can deploy Web, iOS, Android and desktop applications that will use single code base. Other benefit of the Ionic framework is its flexibility of using JavaScript framework of our choice.

We have the opportunity to use ether Angular, React, Vue or even pure JavaScript with TypeScript. This will depend on the team’s preference.

Admin application can be developed as WEB app that communicates either with WEB API gateway for the admin or to use the existing WEB server with special policies for accessing admin endpoints. This will be discussed further with the team.

Development timeline

**Initial development**

At first the application will be simpler. In the API section are defined the core functionality that should be implemented at launch. Core play mechanism and ability to search and buy books are required. Authors to have the ability to create story books and users to see their library with purchase books and their play progress.

**Language support improvements**

At launch we will support only one language per story book. With the first update we will introduce multi-language support for the books. This functionality will be complex and take a lot of time and that’s way is exported as later update. Author will be responsible for adding additional language versions of their books.

**Purchase changes and new policies**

If our application has growing userbase and books count, we can create new type of payment. We can introduce something like XBOX pass or Netflix subscription. This will take some financial consulting and very careful consideration about this new payment policy. The risk is big but can attract larger number of users.

**Introduce complexity to gameplay**

When we have stable platform and userbase we can invest in more complex gameplay elements. Let’s hope we get there. Good luck.