CS5033

Software Architecture Report

220031985, 190020857, 190018240, 190015715

University of
St Andrews

6th April 2023

Architecture Driven Development

Abstract

This technical report outlines the assignment aimed at enhancing the practical application of software architecture concepts. By working as a group, the task is to specify a software architecture for the given system (*A Plot Management Application for Aspiring Writers*) and the creation of an outline implementation of the system in Java that satisfies the designed architecture.

The report outlines the target system's assumptions, functional requirements, qualities, and design and operational limitations. It includes a comprehensive summation of how the architecture was developed, a specification of the architecture in terms of the appropriate views, and a justification of the viewpoints, notations, styles, and design choices that were ultimately decided upon.

The report also describes how the outline implementation was guided by and corresponds to the architecture specification, as well as an analysis of the architecture and implementation plan, as well as a critical evaluation of the solution in terms of significant quality attributes and risks.

Introduction

The group has been tasked with designing and developing the architecture for the target system (A Plot Management Application for Aspiring Writers), which is a non-web-based Java application that will enable individual and collaborative plot development for stories. The application is anticipated to be installed on laptops and desktop computers. The name "**StoryArch**" is used to characterize the application.

Functional Requirements

The following is the application’s feature list, expanded to include the basic functional requirements and assumptions on which StoryArch will be built. As part of our system's design, we categorized features into three priority levels: high-priority, which is crucial for proper functioning; medium-priority, which forms the core of its identity; and low-priority, which is minor quality-of-life enhancements.

1. Users must register with the application before they can use it – High Priority.
    - In-house login via email, password (SHA-256 hashed).
2. Each storyboard has a project associated with it and each chapter has a storyboard associated with it – High Priority.
    - A project is a group of people working on a thing, like an organisation.
    - Storyboards live inside projects.
    - Chapters live inside storyboards.
    - One project can contain multiple storyboards.
3. When there is a team of writers, the Project is assigned a chief writer who sets up a project in the application for that item and invites the rest of the team to join – Medium Priority.
    - The project creator is the default chief writer.
    - A storyboard or chapter can only be deleted by its creator.
4. An individual or a team of writers can develop a storyboard – High Priority.
    - Writers have read-only or read-write to a project; only the creator can delete a project.
5. Chapters do not have to be developed linearly – Medium Priority.
    - Each chapter is stored as its separate entity in a storyboard using a chapter ID. This means they can be re-ordered as you wish.
6. When there is a team of writers, each writer can work on a different part of the plot simultaneously – High Priority.
    - Each chapter is locked for editing while one writer is working on it.
7. The application generates chapter outlines as text files to be used with other word-processing applications – Low Priority.
8. The application should maintain versions of the plot based on commits by writers – Low Priority.

- Each time you save a chapter, it is saved as a revision linked to the previous version.

9. The application should allow writers to visualise the story in chronological or chapter order – Low Priority.
   - Chronological = in order of date modified.
10. The free version of the application supports individual writers, with each allowed a maximum of 10 projects – Medium Priority.
    - No collaboration for free; all work is solo.
11. The paid version (via annual subscription) adds support for teams and unlimited projects – Medium Priority.
    - API calls to an external payment gateway which manages subscription renewal and charges automatically.
12. The application allows the users of the free version to upgrade to the paid version at any time – Medium Priority.
13. The paid version reminds users of upcoming subscription deadlines and allows them to renew using an external payment service – Medium Priority.
    - Has access to an external email service to do this.
    - Will need scheduled tasks to regularly check for upcoming expiries.
    - Will notify the user via the inbuilt message system of the upcoming renewal.
14. The application provides graphical and textual interfaces for plot development – High Priority.
    - A standard desktop application GUI covers this requirement.
15. It should be possible for writers to work in 'offline' mode when they are not connected to the network – Low Priority.
    - The application checks for an internet connection in the local machine and displays messages appropriately.
    - When the network is back, it attempts to upload changes automatically.

- If someone else has uploaded changes while one user was working offline, a merged interface is available.

16. Writers should be able to use and pay for external illustration services – Low Priority.
    - A Fiverr-type service for illustrators to sign up and advertise themselves for work.
    - They get access to a chapter to do the illustration.
    - Illustrators and authors communicate through the built-in message system.
17. Writers should be able to submit plots to publishers (who are external to the system) via existing APIs – Medium Priority.
    - Able to supply story content to arbitrary endpoints.

Non-Functional Requirements

The following is a list of Story Arch's quality requirements and the desired characteristics and properties of the system that ensure it meets the needs of its users and stakeholders. These are listed in order of priority.

1. **Scalability:** The application should be scalable to handle an increasing number of users and projects.
2. **Usability:** The application's user interface should be user-friendly and easy to navigate.
3. **Security:** The application should ensure the privacy and security of all transactions and any user data collected during user registration and stored within projects.
4. **Reliability:** The application should always be reliable and available, due to the presence of cloud-based collaboration features; test cases have been written to check the application's model and service functionalities.
5. **Performance:** The application should be responsive and efficient.

Design and Business Constraints

- The application will be developed using the agile development methodology.
- Due to a tight timescale, the business is prioritising releasing a functional application over producing high-quality code. Technical debt can be fixed when the product is on the market.
- The application will be developed using the Java programming language.
- The application will be compatible with laptops and desktops manufactured within the last 3 years.

Description of Process Used

The architecture was developed iteratively. The team met and developed the basic feature list into a set of assumptions and requirements as outlined above, to decide exactly what we were to build.

We then researched options for styles, viewpoints, and notations, creating a checklist of the plans we had to make. Based on our team discussion and note-taking session, we individually worked on a different representation of the architecture and shared regular progress updates.

We thought working separately would (a) allow each team member to check their understanding of the system architecture, and (b) find different problems to address in the system. This worked well and sparked lots of productive discussions as we each thought more deeply about the system from another perspective.

We brought all the representations together and reviewed them as one, checking for consistency across the notations and viewpoints, before writing the analysis of the architecture presented later in this document.

Justifying the Viewpoints

Viewpoints are a necessity when creating and developing a software architecture. They allow the system to be viewed from different perspectives and abstractions, which can allow the identification of possible conflicts and trade-offs between different system concerns. Viewpoints are also important when communicating the software architecture to the different stakeholders, stakeholders have a wide range of technical experience and backgrounds and it can be hard to coherently explain the architecture to the stakeholders. By using viewpoints software architects can tailor the communication to different audiences to best explain the system and ensure all the stakeholders have a firm understanding of the system. For these reasons the choice of the viewpoints for the system is important, the four viewpoints chosen are logical, development, process and physical.

The logical viewpoint is concerned with the functional requirements of the system and how they are organized into coherent and meaningful structures. It aids the design of the overall structure of the architecture and ensures that it meets the desired functional requirements. It shows the structurally significant elements of the architecture such as components and their interactions. The development view explains the implementation details of the system, it provides a low-level understanding of the components of the system and how they are linked to form the system, using this viewpoint potential implementation risks and pitfalls can be identified. The process view explains the operational aspects of the system, it helps architects design the system's operations processes, for example, deploying and monitoring the system. The process view helps ensure the system operates efficiently as possible. Lastly, the physical viewpoint shows how the logical structure is mapped onto the application hardware. This can cover a wide variety of components including hardware, network infrastructure and the physical environment. It is essential as it allows the architects to design the system's physical infrastructure ensuring that it can support the system's performance and does not hinder the functionality of the system.

The viewpoints cover the entire system, and a different set of viewpoints could be used to allow more precise views of the system, however, given the time constraints and the nature of the project, a larger set of viewpoints was unnecessary, with the four viewpoints covering all the necessary aspects of the system. Given this set of viewpoints the system can be viewed from a variety of perspectives, it allows the system to be explained easily to stakeholders whilst also providing architects with a useful resource for the design of the system. It can be ensured that the system meets the requirements by using these viewpoints and can be maintained and monitored throughout its lifecycle.

System Architecture

The application is a three-tier client-server architecture where the three tiers are:

Presentation: This tier contains the application's user interface components, which allow users to interact with the system. In this case, it features graphical and textual plot development and visualisation interfaces, as well as choices for internal illustration services.

Application: This tier contains the application logic, which handles the system's business logic and orchestration. In this scenario, it would include the MVC architecture's Controller component, which would manage the interaction between the Model via Service Layer and View components as well as the logic for upgrading to the paid version, establishing teams, and generating chapter outline.

Data: This tier contains the application's data storage and retrieval components. In this scenario, it would include the MVC architecture's Model component and Service Layer, which would manage data storage and retrieval, such as user registration information, narrative projects, storyboard and the chapters, and payment information.

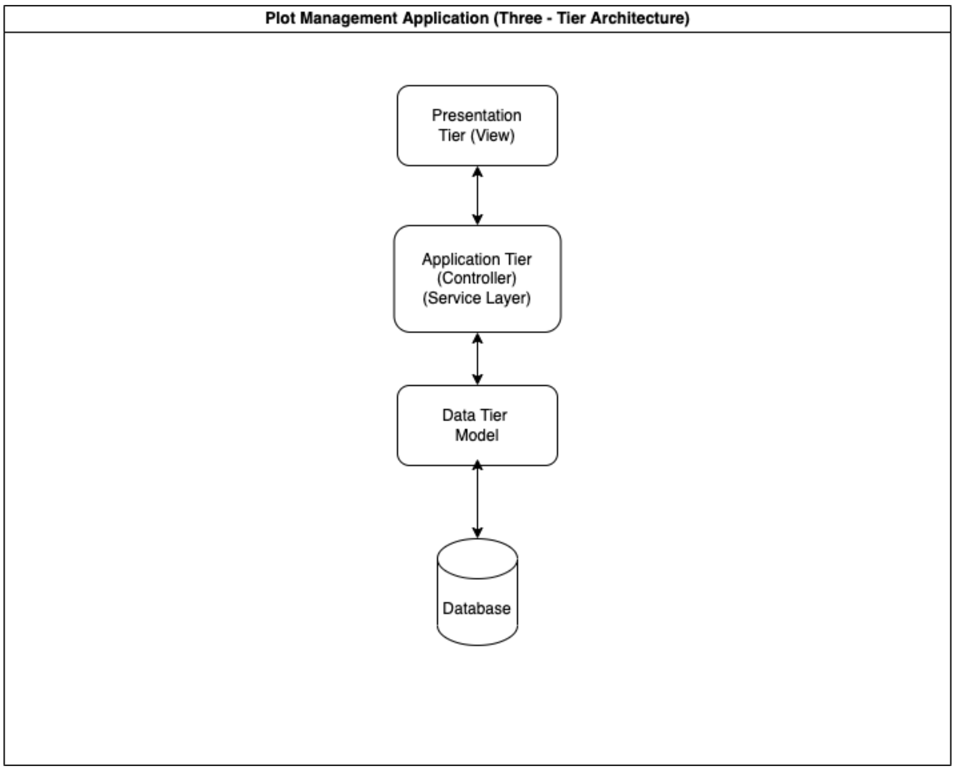

Plot Management Application (Three - Tier Architecture)
Presentation Tier (View)
Application Tier (Controller) (Service Layer)
Data Tier Model
Database

Figure 1: Three–Tier Architecture Diagram for the implemented application

Architecture pattern: Model-View-Controller

The team used the MVC (*Model-View-Controller*) [1] pattern, to develop the application. It is a prominent design pattern that divides the logic of an application into three interrelated components: **the Model**, **the View**, and **the Controller**. The Model is responsible for implementing StoryArch's domain logic, including data, validation, rules, data access, and aggregation logic. The View handles the user interface and is responsible for the presentation of data to the user. Lastly, the Controller manages user interactions and coordinates the flow of certain information between the Model and the View [4].

MVC has several strengths over other alternatives, but the most important is the separation of concerns inherent to the pattern. This allows you to break up the frontend and backend code into separate components, a structure which simplifies the process of maintaining and updating components without interfering with the others, making the system reliable and extensible.

In addition, to further segregate issues and promote scalability, a Service Layer containing the business logic is sometimes utilised in conjunction with the MVC architecture.

Overall, the three-tier architecture and the MVC design pattern are a good choice for this application because they provide a scalable, reliable, maintainable, and extensible architecture.

Logical viewpoint

As mentioned previously, the logical viewpoint organises the program into coherent structures.

An entity-relationship diagram was used to identify the data which would be stored, how it would be stored, and how data relates to other data. While this is not a particularly common notation from the logical viewpoint, it was decided that because MVC puts such emphasis on the application data model, it was important to ensure this had a solid and clear foundation. You could consider this an early-stage class diagram, since the MVC model component maps to the diagram effectively 1:1.

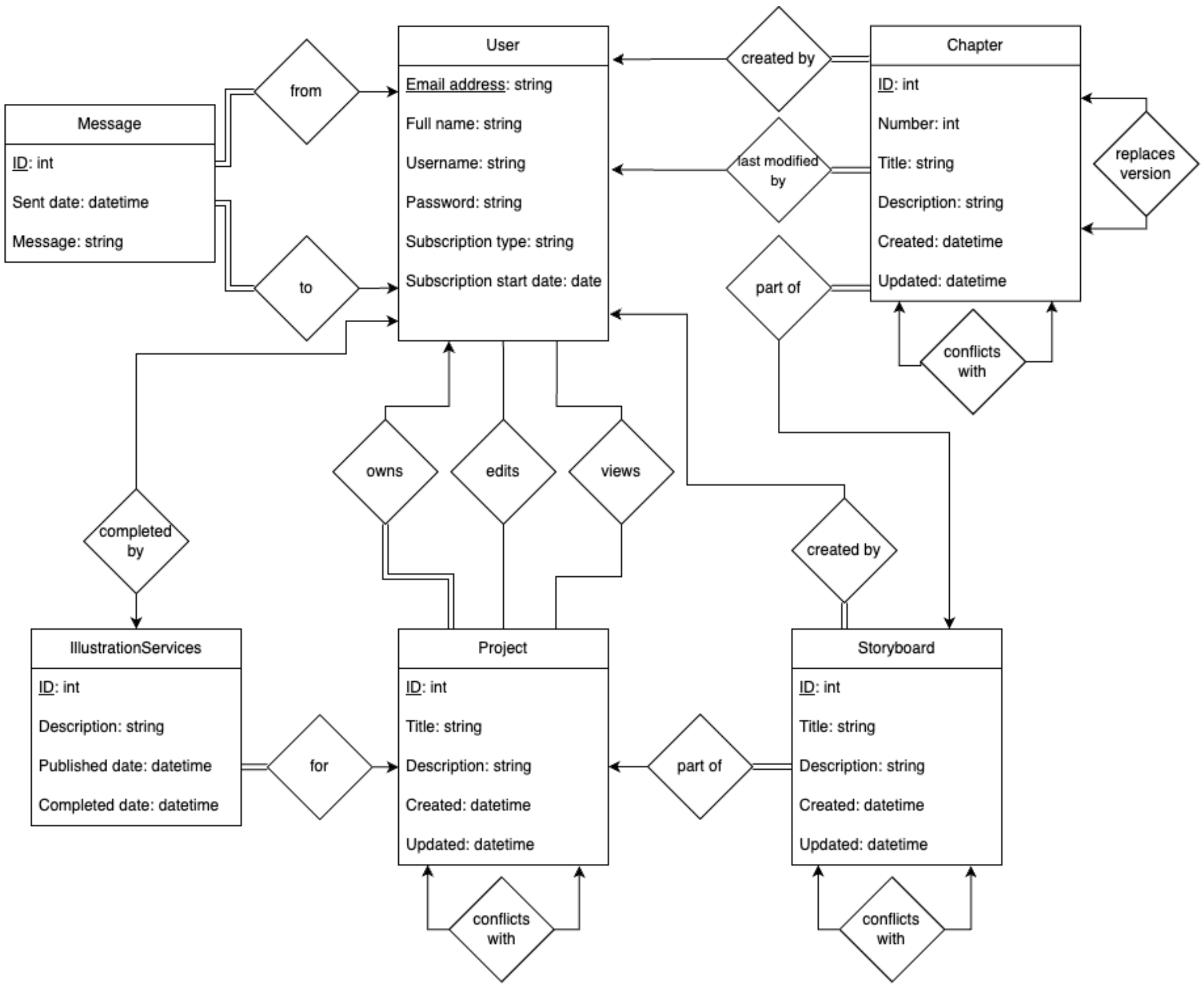


Figure 2: Entity-Relationship diagram for the Model layer of StoryArch

Some key design decisions in this model are:

- The data model itself enforces some constraints which are useful at higher layers in the stack. For example, Projects have total participation in the Owns relationship with the User, so every project must have an owner, but the Edits and Views relationships don't have this constraint, since projects don't *need* to be shared.
- Version control in chapters is handled by every 'save' being stored as a new Chapter entity, which has a relationship to the previous version of that same chapter. This is more data-intensive than storing only differences in a Git-like system but is simpler to implement. We don't store a version history for projects

and stories themselves, since they only contain titles and basic descriptions, which the user is less likely to look back through.

- In cases such as the IllustationServices completion date attribute, we store a date time instead of a Boolean. A null value should be treated as false, and a valid timestamp as true. This is simple to deal with at higher layers of the stack and allows us to encode more information (the time of some action) without adding extra attributes and complexity at the model layer.
- Role-based access to projects is handled by different relationship types between the two entities.
- If a merge conflict occurs when uploading offline edits, the upload can still happen and the merge resolved later using the "conflicts with" relationship.

This was developed into a class diagram, presented in Appendix 1. Entities of the ER model were turned into classes at the model layer, and then more classes were added for the component and view layers. Appropriate methods were added to each class to create the necessary functionality.

In the diagram, you can see the clear split between the layers of the application. The model layer is on the first lines of the figure, and you can see its classes have no methods associated – they are strictly for storing data about entities. Moving to the next row, the Controller layer comprises the ArchController and Services, where data is processed and updated. Finally, the View layer is the CommandLine class, which handles the application UI. This visual representation reinforces the benefits of this style which were covered previously. The boundaries between different 'types' of action in the application are very clear.

Development viewpoint

The development view illustrates a system from the programmer's perspective. The development view is concerned with software management and is represented below using a component diagram. A UML component diagram has been chosen as they are used to model a static view of the components in the system and is a standard for representing the development viewpoint. The diagram below depicts the components in the system and how they relate to each other, the diagram can be split into 3 rough sections: the components concerned with the user, the projects, and the messaging services. Additionally, the illustration services are related to the project and the payment service is associated with the user components.

The component diagram is presented in Appendix 2.

Process viewpoint

The process view for the system depends on the specific use case of the system and the role the user has. Some specific cases have been chosen to highlight the functionality of the system and to guide the implementation of the system. In all use cases of the system, the user will begin by logging into the software or registering, this can be performed online or offline, the user will then access the main dashboard and proceed depending on the service they wish to access.

To represent the process views, UML activity diagrams have been used. This is a standard choice when representing process views and is recommended by many different sources [2]. Activity diagrams represent the behaviour of the system using boxes to represent events that take place on the system and arrows as the flows between these events. The three use cases selected are a user registering with the system, sending a message, and creating a project. The process view for project creation incorporates illustration and collaboration services. These are shown below:

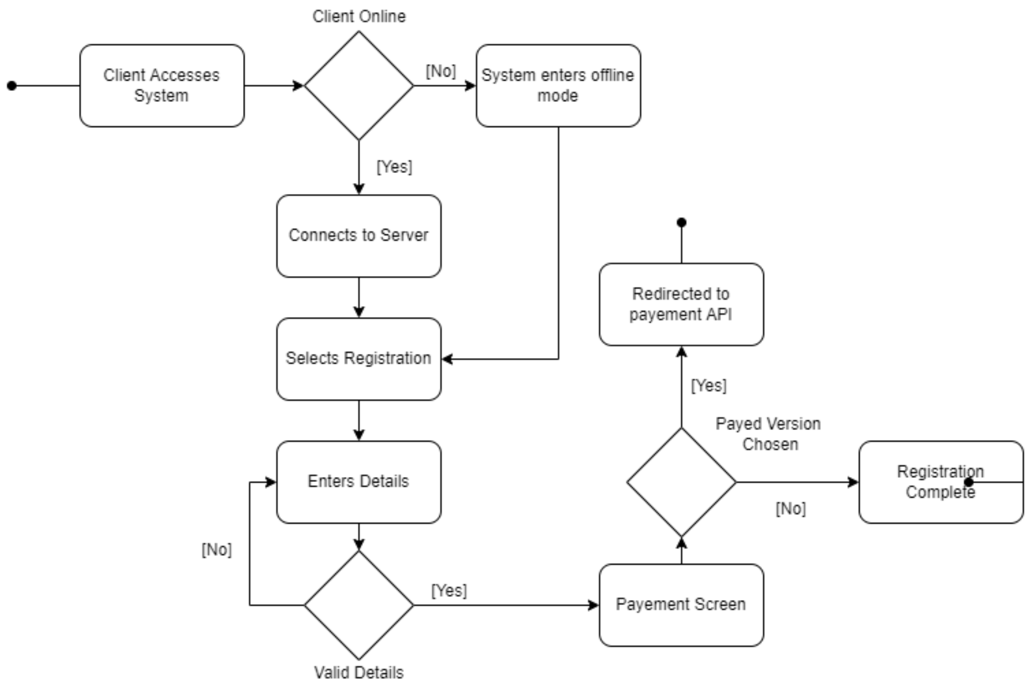


Figure 3: User Registration

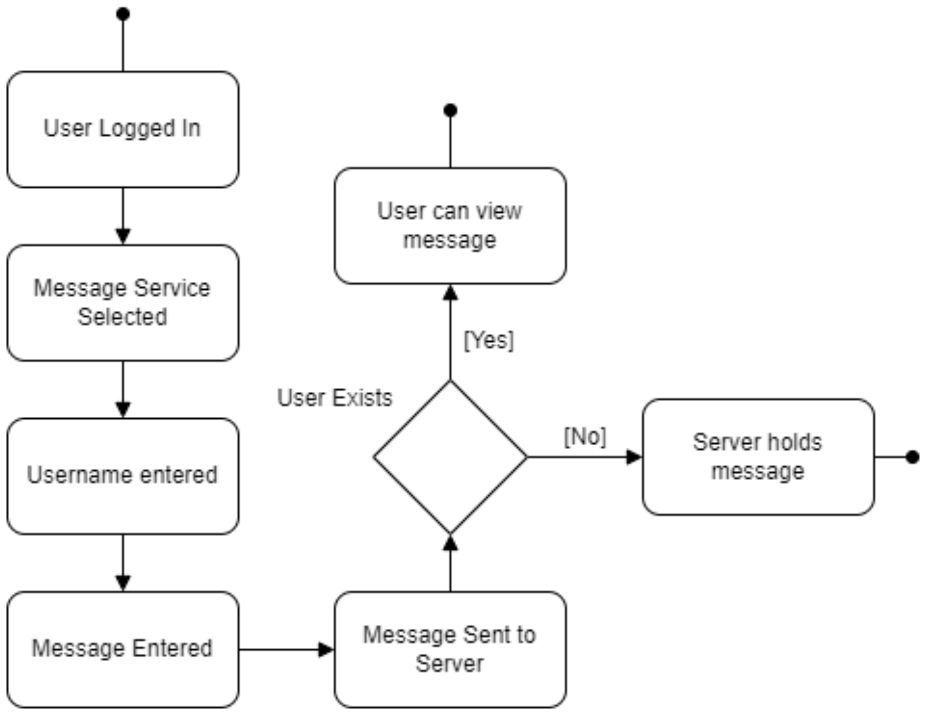
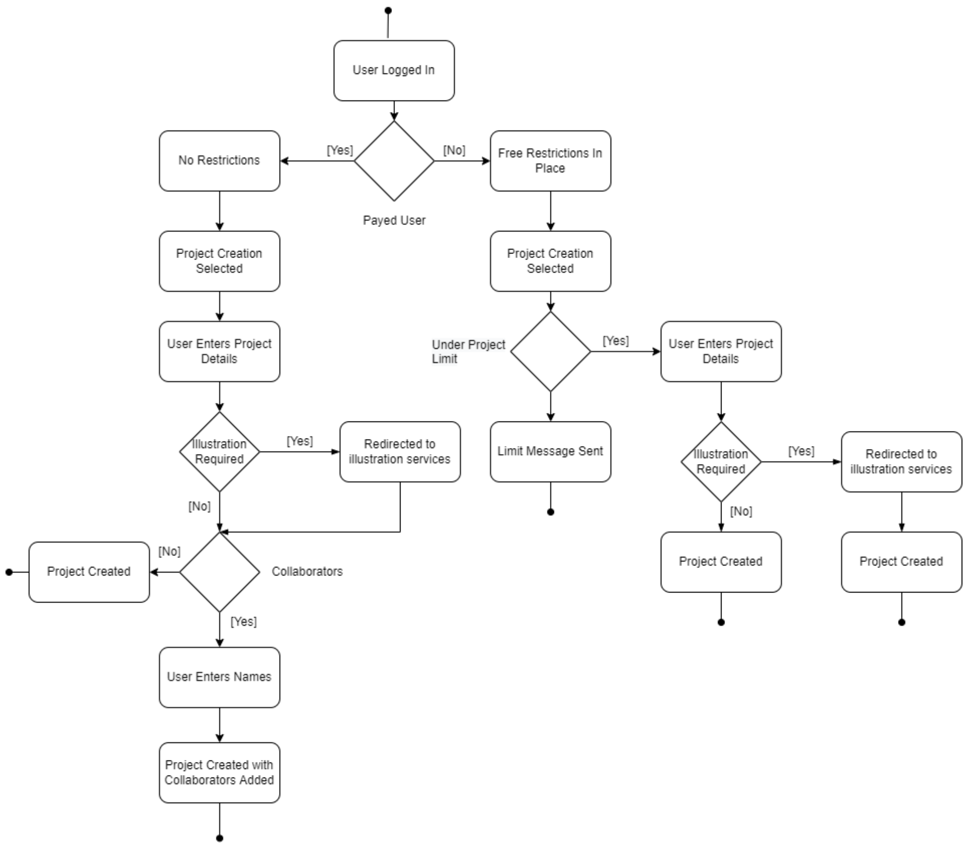


Figure 4: User sending a message



User Logged In
Payed User
[Yes]
[No]
No Restrictions
Free Restrictions In Place
Project Creation Selected
Project Creation Selected
User Enters Project Details
Under Project Limit
[Yes]
User Enters Project Details
Illustration Required
[Yes]
Redirected to illustration services
Limit Message Sent
Illustration Required
[Yes]
Redirected to illustration services
[No]
[No]
Project Created
Collaborators
Project Created
Project Created
[Yes]
User Enters Names
Project Created with Collaborators Added

Figure 5 Creating a new project

Physical viewpoint

The physical viewpoint presents a system from the perspective of a system engineer. It showcases the distribution, physical configuration, and interconnections of hardware and software components and how software components are mapped to hardware resources [3].

For StoryArch, a UML deployment diagram displayed below illustrates where the Model, View, Controller and Service components are located within the system topology and how they interact. The View and Controller are contained within a client-side application running on an end-user device. The View communicates with the Controller, which communicates with the Services component on the application server. Here, either user requests are passed to the Model component, located within a database server, which retrieves the appropriate data from a database, or an API makes a call to an external server or gateway for services like subscription payments or publisher services.

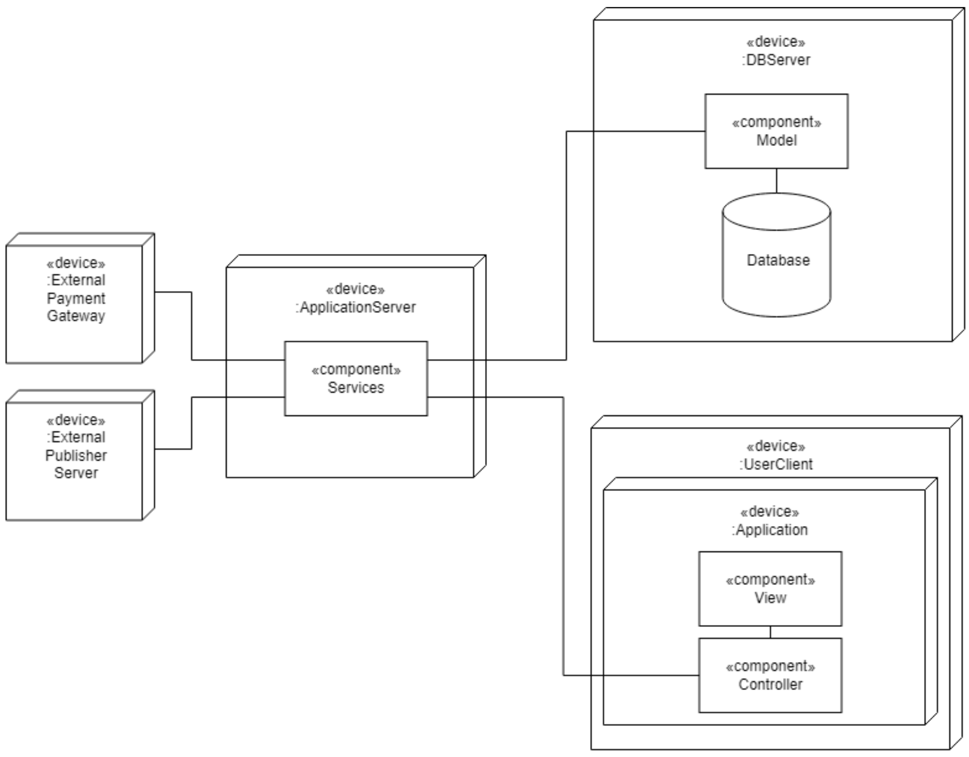


Figure 6 Deployment diagram for StoryArch

Matching the Description and Implementation

Some of the diagrams presented in this report, such as the class diagram, were generated from Java source code. Effectively, we were treating Java as an architecture definition language (ADL). While it isn't an ADL, we thought the best use of our team resources would be to treat it as such, rather than learning a new ADL or diagramming language. It also means we know that our implementation matches the described architecture, because the two are so tightly coupled, reducing the chance of architecture drift.

Where drift does happen, it will be particularly evident because a change to the code which impacts the architecture will change the generated graph, which is a clear signal that the work you are doing must be reviewed more closely.

We can match our viewpoints and descriptions to the generated ones and check for consistency. If the other notations match the generated notations, and the generated notations come directly from the implementation, then we know the architecture and implementation are in sync and as intended.

Analysis

The quality attributes that we established during the initial design phase of our system have been fully incorporated into its architecture and implementation.

The separation of components inherent to the MVC architecture pattern makes it simple for the system to scale horizontally or vertically by adding more instances of a component or increasing the resources allocated to it. For example, it's possible to create multiple View components, which is useful for an application which contains a range of users with varying roles, permissions, and features, such as free and paid StoryArch members, each with multiple projects on which they are authoring or collaborating.

In terms of usability, the ability to easily spin up and customise new instances of the View without affecting other parts of the system means that developers can continuously improve the user experience with frequent updates to the user interface without affecting the application's underlying business logic and data.

Next, the layered structure of the system makes it more secure by reducing the number of possible exploits and attack vectors that a bad actor could use. The View component does not have a direct connection to the Model component, meaning a vulnerability in the user interface would not necessarily lead to compromised user data.

StoryArch is also highly reliable due to the use of the MVC pattern. The relative independence of each component means that failure in one component may not propagate due to the small number of dependencies between them, if any. Additionally, the ability to scale the system horizontally means that the load can be distributed across multiple servers, reducing overloading and downtime. 93 test cases were written to support system reliability.

Lastly, the separation of concerns leads to a more performant system. Each component's optimisation can be tailored to its responsibilities. For example, the View can be optimised for rendering while the Model can be optimised for database access. Furthermore, introducing multiple servers as part of horizontal scaling would allow the application to quickly handle a larger number of requests.

References

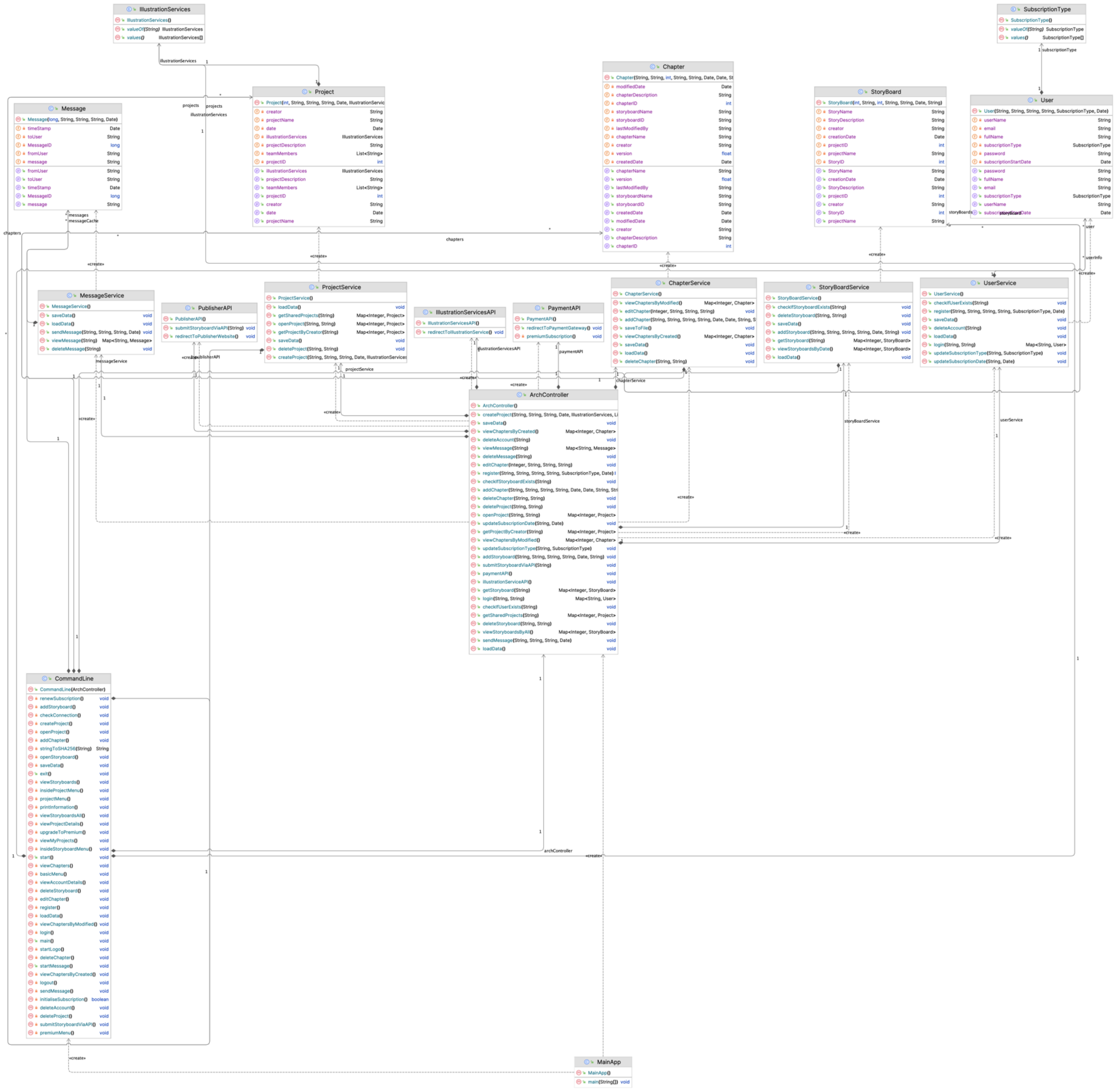
[1] Software Architect's Handbook (2018), Joseph Ingeno, Accessed on 27th March 2023 (Pg. 232)

[2] Jayawardene, P.D. (2021). *4+1 Architectural view model in Software.* [online] Java revisited. Available at: https://medium.com/javarevisited/4-1-architectural-view-model-in-software-ec407bf27258.

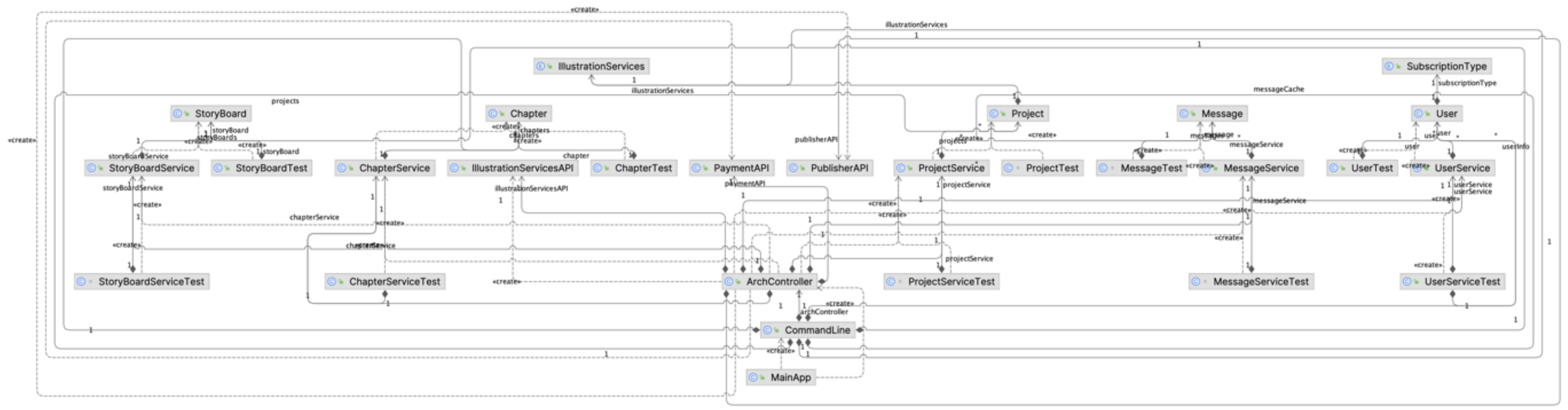
[3] Lucid chart. (n.d.). Deployment Diagram Tutorial. [online] Available at: https://www.lucidchart.com/pages/uml-deployment-diagram.

[4] Tutorialspoint (2019). MVC Framework - Introduction - Tutorialspoint. [online] Tutorialspoint.com. Available at: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm .

Appendix 1: Class Diagram


IllustrationServices
IllustrationServices()
valueOf(String) IllustrationServices
values() IllustrationServices[]
illustrationServices
Project
Project(int, String, String, String, Date, IllustrationServic
creator String
projectName String
date Date
illustrationServices IllustrationServices
projectDescription String
teamMembers List<String>
projectID int
illustrationServices IllustrationServices
projectDescription String
teamMembers List<String>
projectID int
creator String
date Date
projectName String
projects
illustrationServices
Message
Message(long, String, String, String, Date)
timeStamp Date
toUser String
MessageID long
fromUser String
message String
fromUser String
toUser String
timeStamp Date
MessageID long
message String
messages
messageCache
chapters
Chapter
Chapter(String, String, int, String, String, Date, Date, St
modifiedDate Date
chapterDescription String
chapterID int
storyboardName String
storyboardID String
lastModifiedBy String
chapterName String
creator String
version float
createdDate Date
chapterName String
version float
lastModifiedBy String
storyboardName String
storyboardID String
createdDate Date
modifiedDate Date
creator String
chapterDescription String
chapterID int
StoryBoard
StoryBoard(int, String, int, String, String, Date, String)
StoryName String
StoryDescription String
creator String
creationDate Date
projectID int
projectName String
StoryID int
StoryName String
creationDate Date
StoryDescription String
projectID int
creator String
StoryID int
projectName String
storyBoards
SubscriptionType
SubscriptionType()
valueOf(String) SubscriptionType
values() SubscriptionType[]
subscriptionType
User
User(String, String, String, String, SubscriptionType, Date)
userName String
email String
fullName String
subscriptionType SubscriptionType
password String
subscriptionStartDate Date
password String
fullName String
email String
subscriptionType SubscriptionType
userName String
subscriptionStartDate Date
user
userInfo
«create»
MessageService
MessageService()
saveData() void
loadData() void
sendMessage(String, String, String, Date) void
viewMessage(String) Map<String, Message>
deleteMessage(String) void
messageService
PublisherAPI
PublisherAPI()
submitStoryboardViaAPI(String) void
redirectToPublisherWebsite() void
publisherAPI
ProjectService
ProjectService()
loadData() void
getSharedProjects(String) Map<Integer, Project>
openProject(String, String) Map<Integer, Project>
getProjectByCreator(String) Map<Integer, Project>
saveData() void
deleteProject(String, String) void
createProject(String, String, String, Date, IllustrationServices
projectService
IllustrationServicesAPI
IllustrationServicesAPI()
redirectToIllustrationService() void
illustrationServicesAPI
PaymentAPI
PaymentAPI()
redirectToPaymentGateway() void
premiumSubscription() void
paymentAPI
ChapterService
ChapterService()
viewChaptersByModified() Map<Integer, Chapter>
editChapter(Integer, String, String, String) void
addChapter(String, String, String, String, Date, Date, String, S
saveToFile() void
viewChaptersByCreated() Map<Integer, Chapter>
saveData() void
loadData() void
deleteChapter(String, String) void
chapterService
StoryBoardService
StoryBoardService()
checkIfStoryboardExists(String) void
deleteStoryboard(String, String) void
saveData() void
addStoryboard(String, String, String, String, Date, String) void
getStoryboard(String) Map<Integer, StoryBoard>
viewStoryboardsByDate() Map<Integer, StoryBoard>
loadData() void
storyBoardService
UserService
UserService()
checkIfUserExists(String) void
register(String, String, String, String, SubscriptionType, Date)
saveData() void
deleteAccount(String) void
loadData() void
login(String, String) Map<String, User>
updateSubscriptionType(String, SubscriptionType) void
updateSubscriptionDate(String, Date) void
userService
ArchController
ArchController()
createProject(String, String, String, Date, IllustrationServices, Li
saveData() void
viewChaptersByCreated() Map<Integer, Chapter>
deleteAccount(String) void
viewMessage(String) Map<String, Message>
deleteMessage(String) void
editChapter(Integer, String, String, String) void
register(String, String, String, String, SubscriptionType, Date)
checkIfStoryboardExists(String) void
addChapter(String, String, String, String, Date, Date, String, Str
deleteChapter(String, String) void
deleteProject(String, String) void
openProject(String, String) Map<Integer, Project>
updateSubscriptionDate(String, Date) void
getProjectByCreator(String) Map<Integer, Project>
viewChaptersByModified() Map<Integer, Chapter>
updateSubscriptionType(String, SubscriptionType) void
addStoryboard(String, String, String, String, Date, String) void
submitStoryboardViaAPI(String) void
paymentAPI() void
illustrationServiceAPI() void
getStoryboard(String) Map<Integer, StoryBoard>
login(String, String) Map<String, User>
checkIfUserExists(String) void
getSharedProjects(String) Map<Integer, Project>
deleteStoryboard(String, String) void
viewStoryboardsByAll() Map<Integer, StoryBoard>
sendMessage(String, String, String, Date) void
loadData() void
archController
CommandLine
CommandLine(ArchController)
renewSubscription() void
addStoryboard() void
checkConnection() void
createProject() void
openProject() void
addChapter() void
stringToSHA256(String) String
openStoryboard() void
saveData() void
exit() void
viewStoryboards() void
insideProjectMenu() void
projectMenu() void
printInformation() void
viewStoryboardsAll() void
viewProjectDetails() void
upgradeToPremium() void
viewMyProjects() void
insideStoryboardMenu() void
start() void
viewChapters() void
basicMenu() void
viewAccountDetails() void
deleteStoryboard() void
editChapter() void
register() void
loadData() void
viewChaptersByModified() void
login() void
main() void
startLogo() void
deleteChapter() void
startMessage() void
viewChaptersByCreated() void
logout() void
sendMessage() void
initialiseSubscription() boolean
deleteAccount() void
deleteProject() void
submitStoryboardViaAPI() void
premiumMenu() void
MainApp
MainApp()
main(String[]) void

Appendix 2: Component Diagram


IllustrationServices
SubscriptionType
StoryBoard
Chapter
Project
Message
User
StoryBoardService
StoryBoardTest
ChapterService
IllustrationServicesAPI
ChapterTest
PaymentAPI
PublisherAPI
ProjectService
ProjectTest
MessageTest
MessageService
UserTest
UserService
StoryBoardServiceTest
ChapterServiceTest
ArchController
ProjectServiceTest
MessageServiceTest
UserServiceTest
CommandLine
MainApp
«create»
illustrationServices
projects
messageCache
subscriptionType
storyBoard
storyBoards
storyBoardService
chapters
chapter
publisherAPI
messageService
userService
userInfo
user
message
paymentAPI
illustrationServicesAPI
chapterService
projectService
archController
1
*