Fanlinc System Design

Ilir Dema

Software Engineering – CSCC01

October 20, 2019

Team Null

Ryan Sue

Michelle Kee

Terry Wong

Spencer Paulmark

Derick Amalraj

Table of Contents

CRC Cards

GraphQLProvider

Class name: GraphQLProvider
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:

- Creates the GraphQL Bean
- Wires the schema and the data fetchers together to create the /graphql endpoint

Collaborators:

- GraphQLFetchers

GraphQLFetchers

This is also called a resolver in GraphQL lingo.

Class name: GraphQLFetchers
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:

- Provides data for the graphql provider
- Returns the data fetchers that are used by the graphql provider. These data fetchers are the application logic behind graphql methods like createAccount().

Collaborators:

- AccountRepository
- FandomRepository
- FanlincSession

Account

Class name: Account
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:

- Data class that represents the Account stored in mongo

Collaborators:

- None

Fandom

Class name: Fandom
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:

- Data class that represents the Fandom stored in mongo

Collaborators:

- None

FandomMembership

Class name: FandomMembership
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:
- Data class that represents a Fandom membership of a user and fandom.
- Stores the users fandom level (This is the limited, …, expert and fan, cosplayer, vendor/artist distinction)

Collaborators:
- None

Application

Class name: Application
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:
- Represents the Spring Boot Application.

Collaborators:
- SpringBootApplication (Spring)

AccountRepository

Class name: AccountRepository
Parent class (if any): MongoRepository
Class Subclasses (if any): No
Responsibilities:
- DAO that has basic account CRUD operations and other account operations for the account collection in Mongo.
- Interact with the account collection in mongo

Collaborators:
- None

FandomRepository

Class name: FandomRepository
Parent class (if any): MongoRepository
Class Subclasses (if any): None
Responsibilities:
- DAO that has basic Fandom CRUD operations and other fandom operations for the fandom collection in Mongo.
- Interact with the fandom collection in mongo

Collaborators:
- None

FanlincSession

Class name: FanlincSession
Parent class (if any): None
Class Subclasses (if any): None
Responsibilities:

- Represents a Fanlinc session for an account
- Stores various session states for the front and back end.

Collaborators:

- AccountRepository

LoginService

Class name: LoginService
Parent class (if any): No
Class Subclasses (if any): No
Responsibilities:

- Login the user in given the user's credentials and register the session information.
- Log out the current user and invalidate the session.

Collaborators:

- FanlincSession
- AccountRepository

Environment

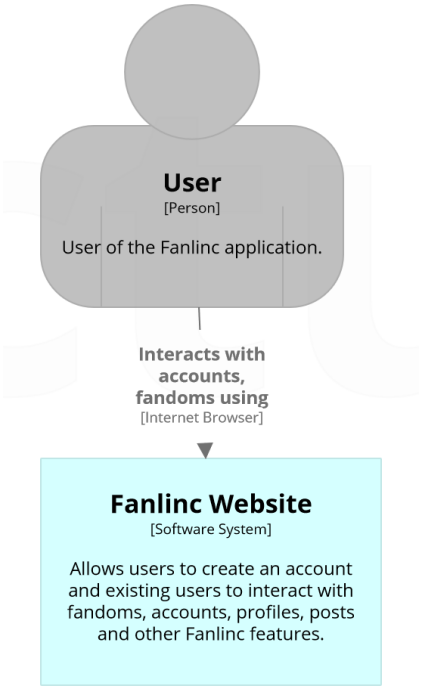
The Fanlinc website can be visited by any modern web browser. The development environment is a Linux (Ubuntu) or Windows 10 computer with Java 8, node and typescript. The GraphQL API Application is run from this along with the Fanlinc UI. The backend database is using MongoDB and is hosted using MongoDB Atlas. Specifically, it is the free plan. The GraphQL API Application should be the only thing accessing the Mongo database.

Software Architecture Diagram

We are using the C4 model for the diagram. This is basically a set of 4 diagrams that starts high level (at a contextual level) and goes down to a container view, component view then a UML view. The design undergoes many changes, so it is too volatile to generate UML diagrams currently. These diagrams were generated using Structurizr. They are accompanied by some system decomposition that relates the system architecture diagram to the design.

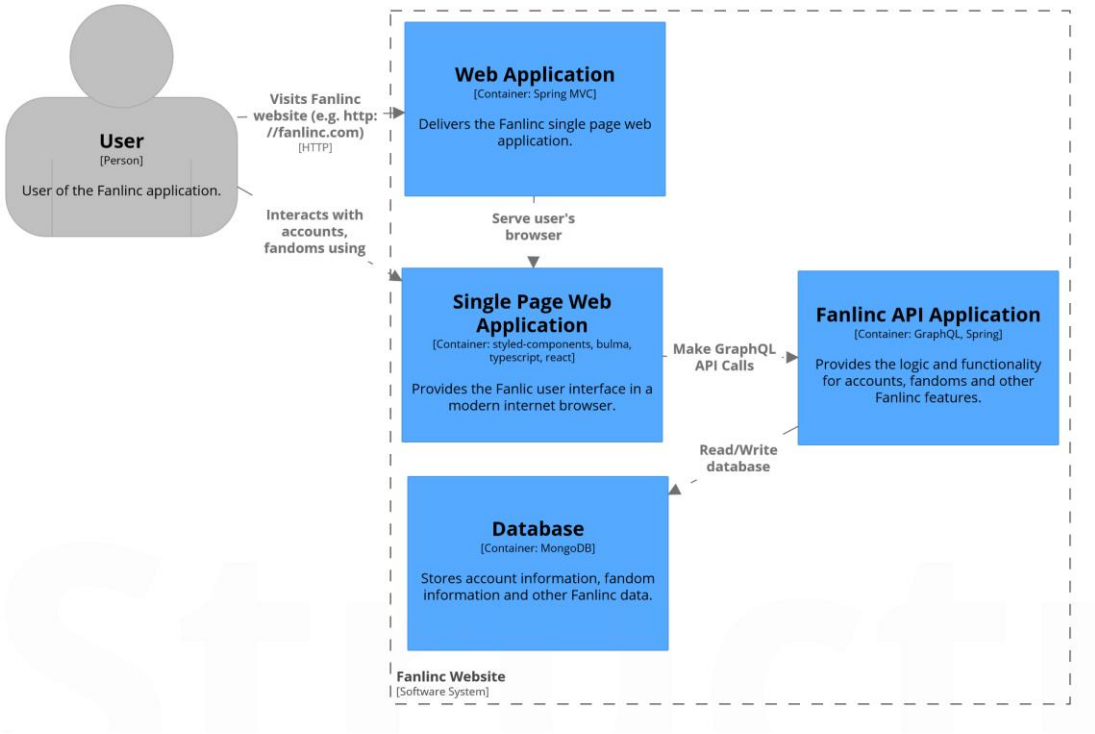
Legend for Diagrams

System Context View



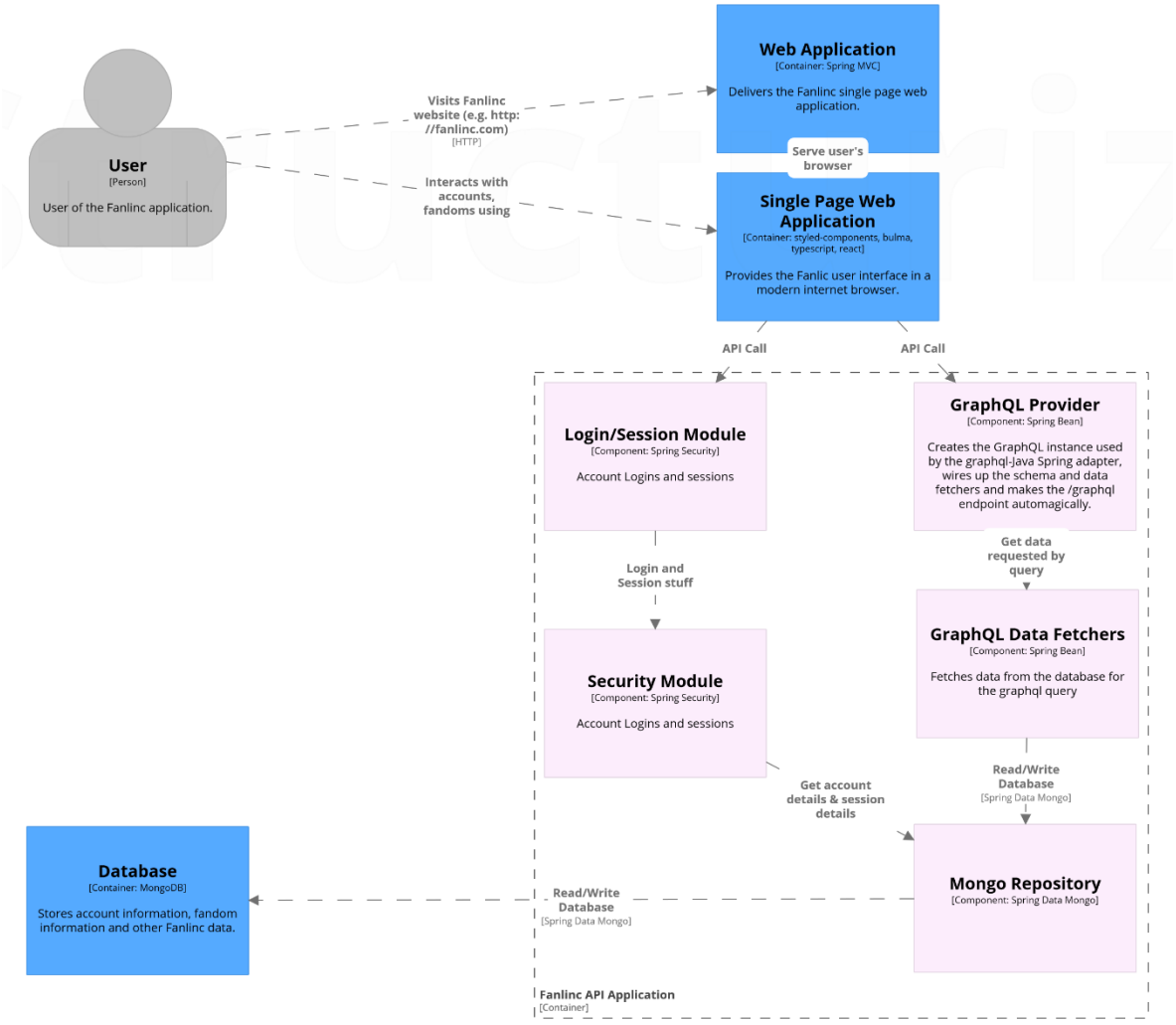
This is the highest-level diagram; it shows that the user will interact with the Fanlinc website. The website is the user interface which the user will interact with.

Container View



We will use Spring to serve the Single Page Web Application which makes heavy use of Typescript and React. That is what the user sees and interacts with. The single page web application will make API calls to the API which interacts with the backend database which will be a mongo database hosted on MongoDB Cloud, specifically the free plan that service.

Component View (API Application)



This is a zoom in which displays the components in the API Application.

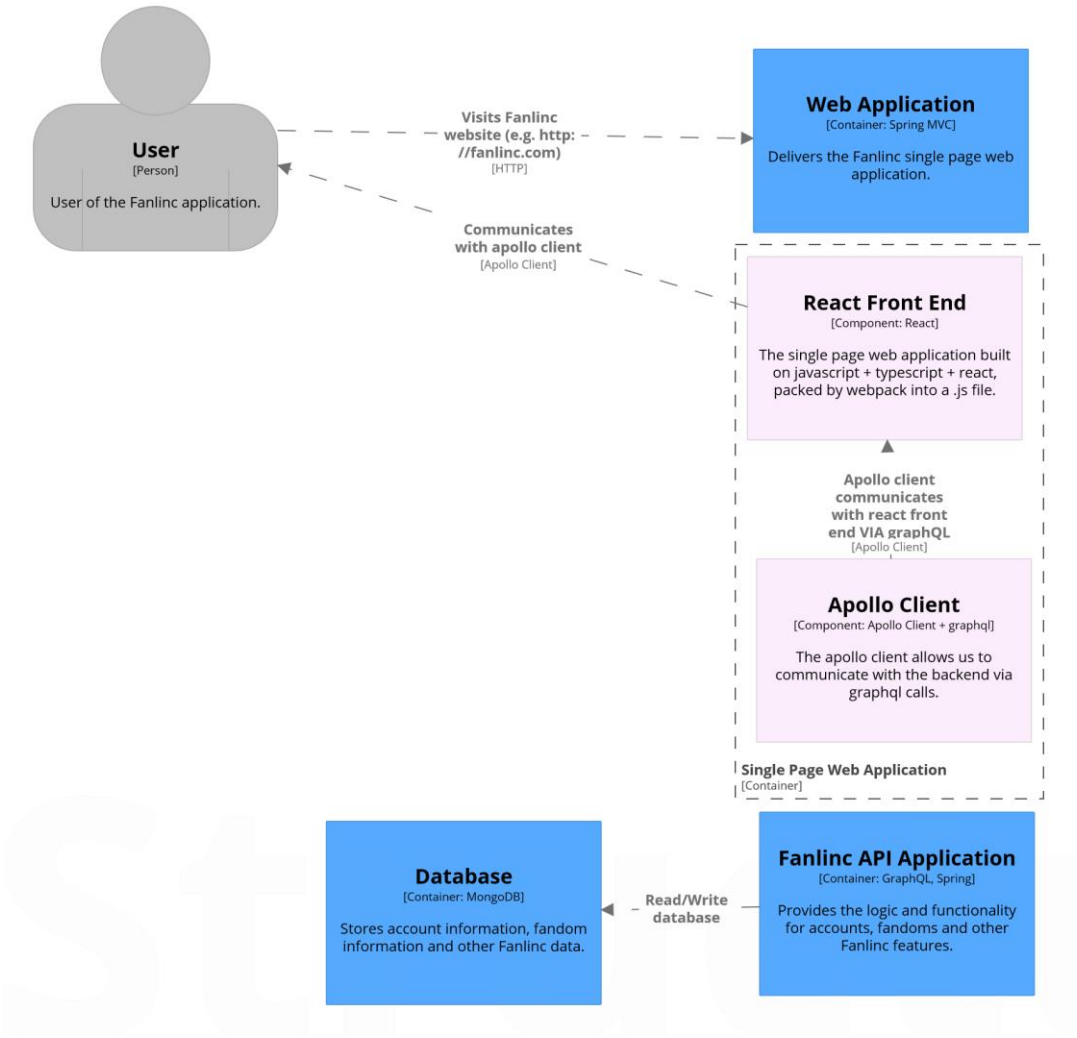
This follows the Model-View-Controller architectural pattern. The mongo database can be considered the model. The view is the single page web application and the controllers are the GraphQL data fetchers.

Account login and logout will use Spring Security along with Spring Session whereas everything else will use the GraphQL API. If the user enters invalid credentials, the login module will not log the user in. The session module will invalidate and expire inactive sessions after the user does not do anything for some time.

The provider wires up the schema and the data fetchers to create the graphql endpoint which the front end will interact with. The data fetchers are also known as resolvers. They contain the application logic to do a certain method defined in the GraphQL schema. For example, a data fetcher that creates accounts contains the logic to read/write from the AccountRepository and return appropriately per the GraphQL schema. The provider associates a part of the schema with a data fetcher. Invalid input into the data fetcher will yield an error response in graphql.

In this diagram, the Mongo Repository component is analogous to the AccountRepository and FandomRepository CRC cards. They are Data Access Objects (DAOs) that hold the functionality to preform CRUD operations on the Account/Fandom along with some more complex operations such as checking if an account is deleted. This isolates the logic of the API and the specific database we are using so that the API is less coupled to a specific database (in our case, Mongo). This also keeps the data classes cohesive.

Component View (Single Page Web Application)



The data flow in our system is as follows: The user will interact with the React Front End by clicking on links or buttons or perhaps trying to visit a page. A request to get or send data will be sent to graphql using a query or a mutation, which is handled by the Apollo Client. The Apollo Client will send this query to the graphql provider by a web request. The GraphQL provider will call the GraphQL Data fetcher that co-responds to this query. If necessary, this data fetcher will interact with the Mongo Repository to read or write data. After this, the query will return the data that needed to be fetched, or a list of errors. In the case that no errors occur, the list of errors would be empty. This response would

be received by the apollo client, and the apollo client would give the result to the react front end. If there are no errors, the desired result will take place. If there are errors, a modal box notifying the user of an error will occur. This error could be an internal server error, or something as simple as a notification saying that a username/email was already taken.

If a user tries to use a browser that doesn't have javascript enabled, they will get a message telling them to use a modern browser that can run javascript.

A possible error that we can encounter occurs when the graphQL query from the front end sent to the back end is incorrectly formatted or contains bad data that will cause an error. A badly formatted query will be caught by the GraphQL provider and will return an error that the query failed to validate. A query containing bad data will validate and be sent to the graphQL data fetchers. These data fetchers will validate the data that has been given to them, and if they find bad data, they will return an error to be sent back to the front end indicating to the user that the data they entered was bad.

If a network error prevents the front end and back end from connecting, the user will be notified on the front end that their request timed out by a modal box. This is made possible by the fact that graphql queries in the Apollo Client module will return a specific error when it fails to connect to an endpoint.

A possible error that we can encounter occurs if the user is not logged in or has an invalid session when trying to get or modify sensitive data. This is handled by our Login/Session Module and Security Module running spring security to validate user sessions.