Shopify App: Better than Sum

ESOF 423, Software Engineering Applications, Spring 2022

Faith Nelson, Jacob Hofer, & Megan Fehres

Section 1: Program.

ESOF 423 Software Engineering Applications: 3 Credits (1 Lec, 2 Lab)

PREREQUISITES: ESOF 322

Application of software engineering techniques and methodologies acquired in previous courses to solve an open-ended software engineering problem provided by stakeholders. Students will use a team-based approach to requirements gathering, designing, implementation, testing, integration, and delivery of the software solution. CSCI 440 is recommended.

Section 2: Teamwork.

For our team, we wanted to make sure that we each were able to play to our strengths and do work equally. We split our work up into three parts:

Team Member 1 - The Database, 35%

Team Member 2 - The Admin UI, 35%

Team Member 3 - The Client UI, 30%

The database was developed by team member 1 using MySQL. He created CRUD methods in javascript to allow users to have access to the database through our user interface (UI). Team member 1 was our subject matter expert (SME) on this project. He had worked on this type of web development previously and had some knowledge about the best way to put it all together.

Team member 2 developed the admin UI system, which was a string of pages that allowed an administrator on a Shopify store to control the way the app behaved for them. Including giving access to changing the products and variations options associated with their store id that was set up by team member 1 on the database side. Team member 2 was given approximately 110 hours which is approximately ⅓ of the project. The UI admin side had a big

learning curve for writing a webpage using Javascript. Once that was completed, it became more about the programming logic issues. This, along with the planning stages, took up the first three sprints. The fourth sprint was where the functionality and some of the beautification of the site came to fruition. Finally, some issues came into play when trying to use the database, which made for more time programming the product page.

Team member 3 developed the client UI, which included the shop page that allowed the customer to customize the product and see the price change as they chose various options. This included making calls to the database with Javascript files to pull the product information and using a resulting JSON object to display the information. Next, stylization was achieved with CSS, linked to the main .liquid file for the app block. This process was time-consuming, as pushing code to the website was slow and needed to be done frequently. Additionally, development did not go smoothly in terms of javascript, so the estimate of 25 hours per sprint was appropriate and matched the actual. This was the final stage of the app and was the end goal for the client to be able to have the cost update dynamically at the top of the page.

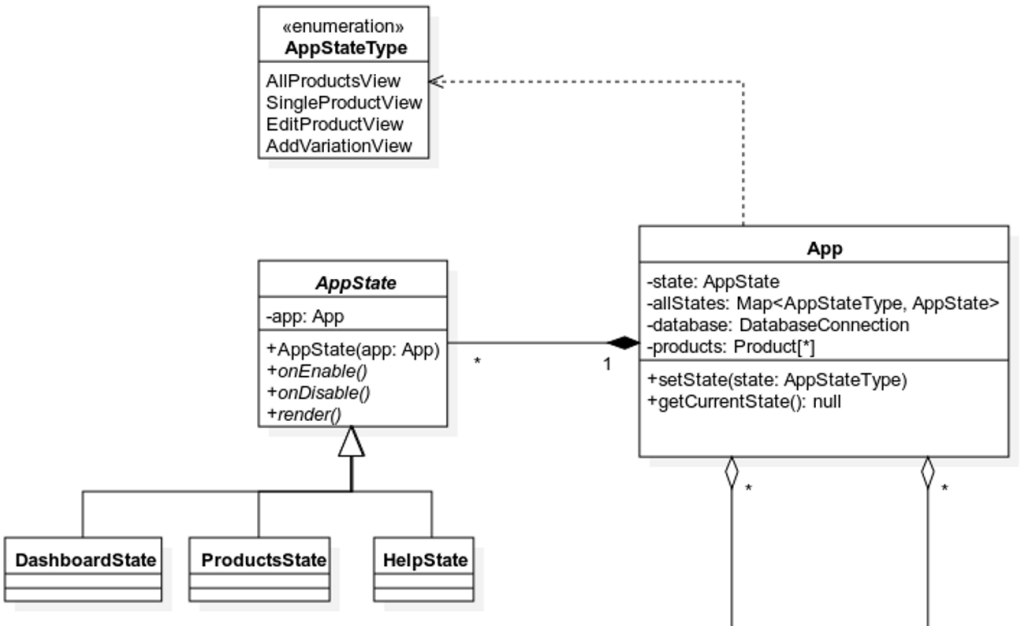
Section 3: Design pattern

The main design pattern we used in our project was the State Pattern. We used this pattern to control how our app functions and how it should render the user interface. We decided to use this pattern rather than ReactJS's page implementation, as this gave us better control over the app and the data being used. For example, the State Pattern provides the ability to quickly hot-swap between different pages without fetching all of the data again, allowing our other APIs to be RESTful. This also kept our code cleaner and easily manageable by enabling us to create better-isolated sections. This pattern is implemented using the following files:

```
App.js
AppState.js
```

```
AppStateType.js
DashboardState.js
HelpState.js
ProductState.js
```

The UML detailing this design pattern is shown below:



Section 4: Technical writing.

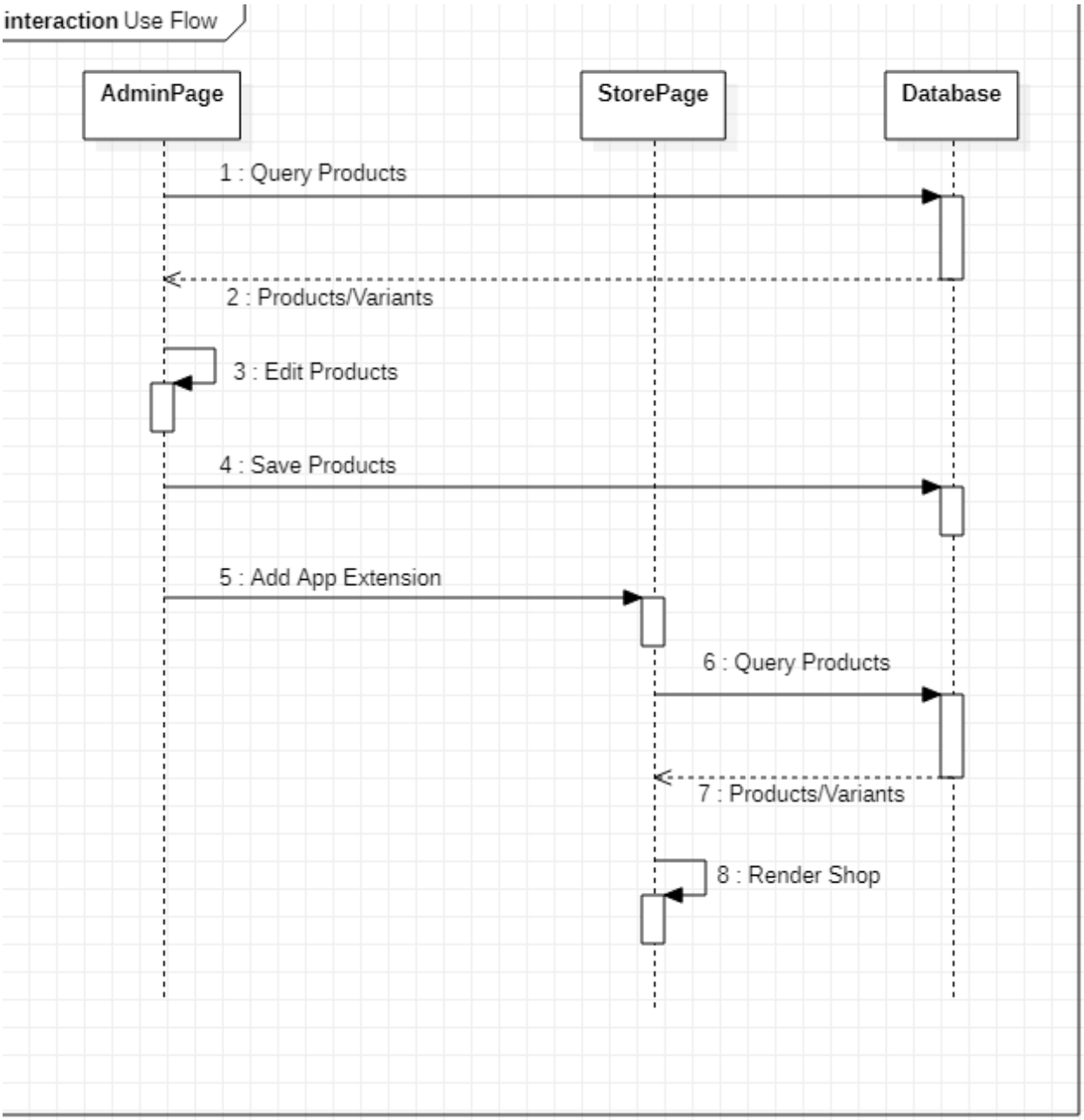
Include the technical document that accompanied your capstone project.

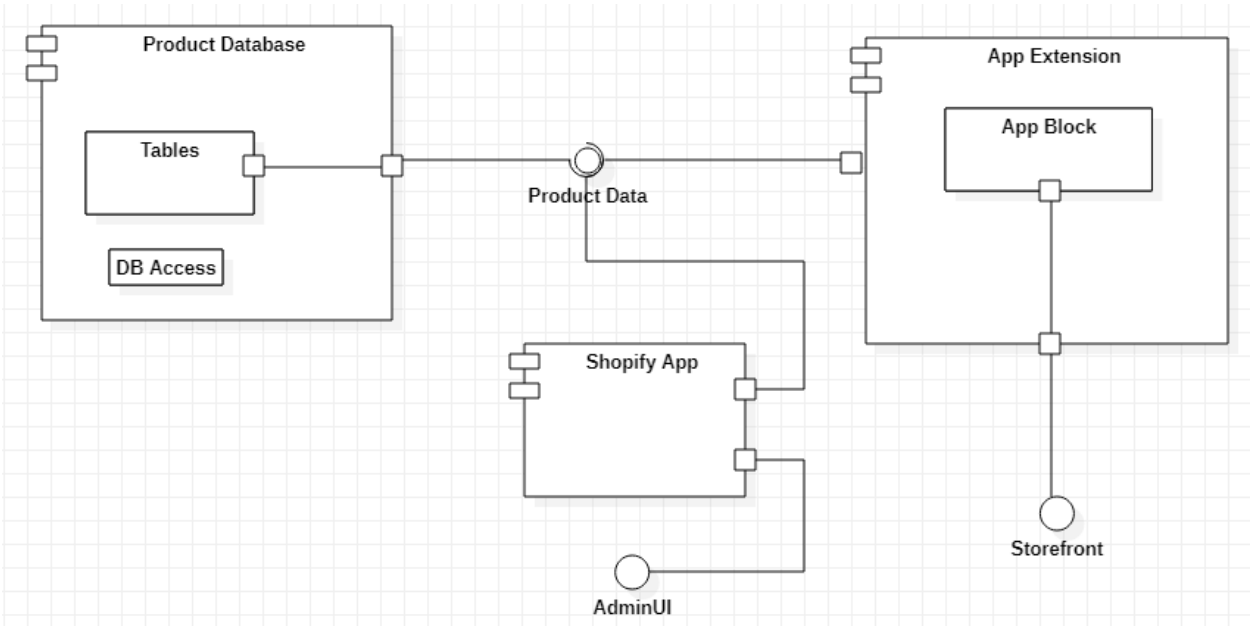
Developer Documentation

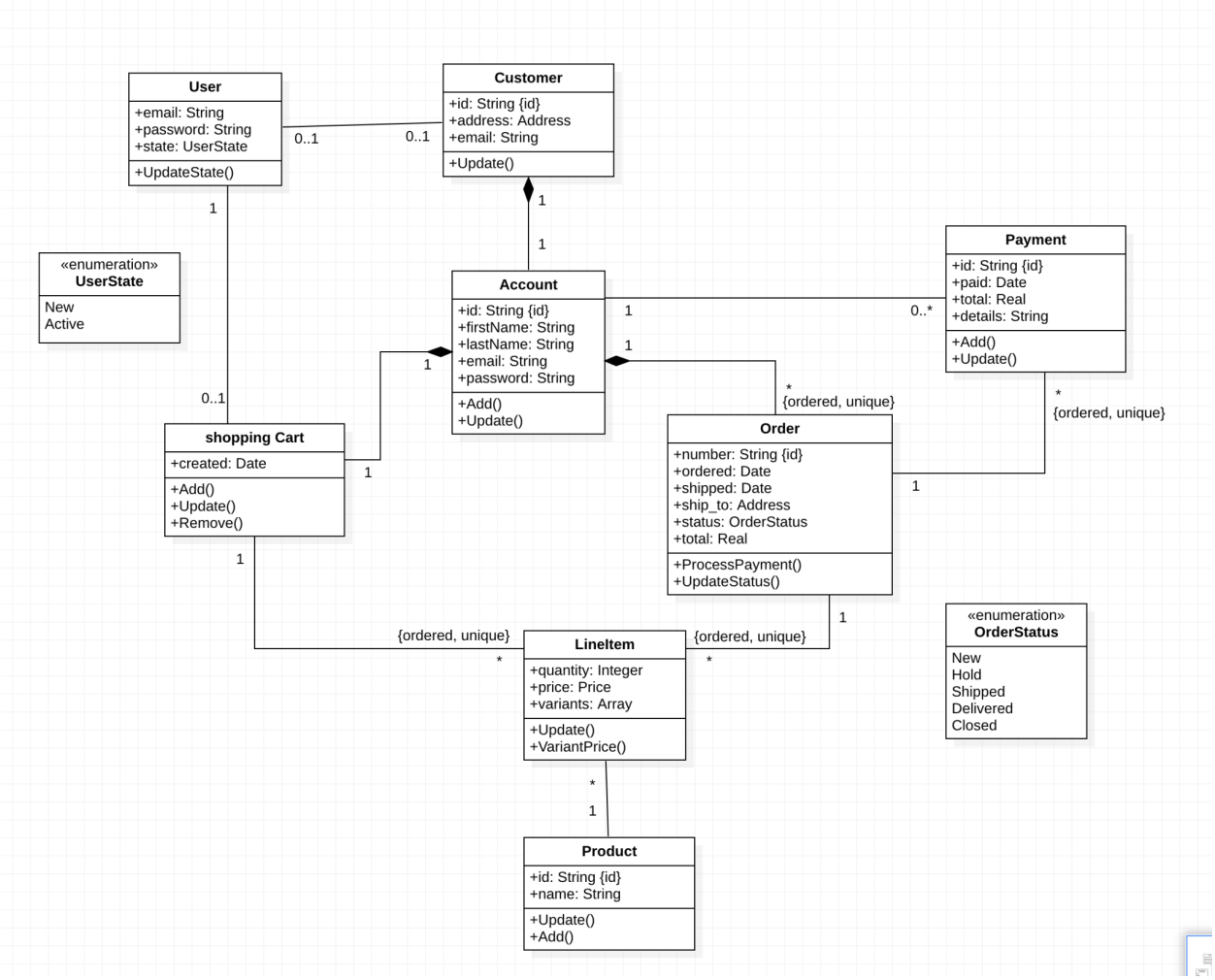
User Documentation

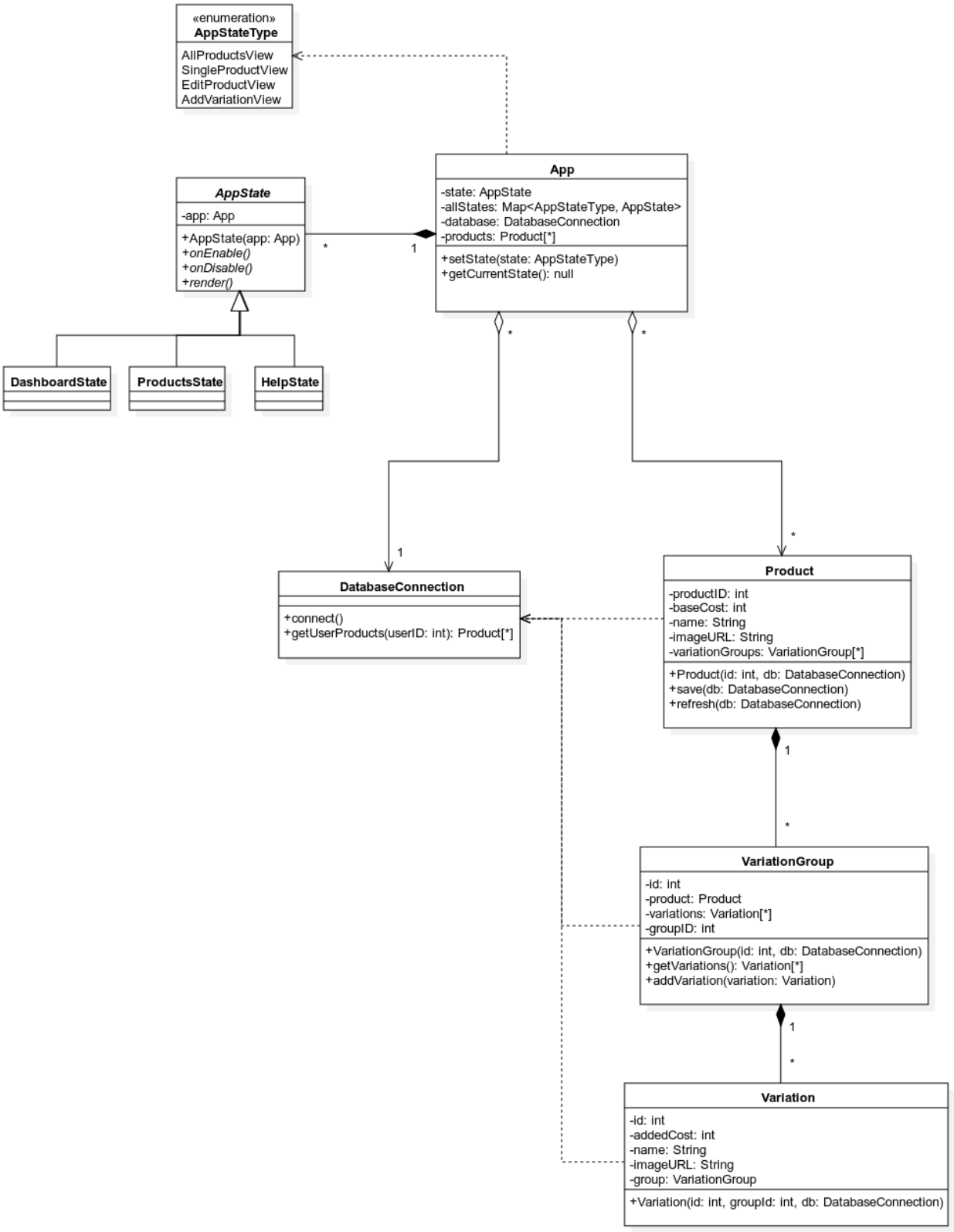
Section 5: UML.

Attach the UML design diagrams for your capstone project that were created before you began coding your project.




Product Database
Tables
DB Access
Product Data
App Extension
App Block
Shopify App
Storefront
AdminUI


User
+email: String
+password: String
+state: UserState
+UpdateState()
Customer
+id: String {id}
+address: Address
+email: String
+Update()
0..1
0..1
1
1
«enumeration»
UserState
New
Active
Account
+id: String {id}
+firstName: String
+lastName: String
+email: String
+password: String
+Add()
+Update()
Payment
+id: String {id}
+paid: Date
+total: Real
+details: String
+Add()
+Update()
0..*
1
1
1
0..1
*
{ordered, unique}
*
{ordered, unique}
shopping Cart
+created: Date
+Add()
+Update()
+Remove()
Order
+number: String {id}
+ordered: Date
+shipped: Date
+ship_to: Address
+status: OrderStatus
+total: Real
+ProcessPayment()
+UpdateStatus()
1
1
1
{ordered, unique}
*
{ordered, unique}
*
«enumeration»
OrderStatus
New
Hold
Shipped
Delivered
Closed
LineItem
+quantity: Integer
+price: Price
+variants: Array
+Update()
+VariantPrice()
*
1
Product
+id: String {id}
+name: String
+Update()
+Add()


«enumeration»
AppStateType
AllProductsView
SingleProductView
EditProductView
AddVariationView
App
-state: AppState
-allStates: Map<AppStateType, AppState>
-database: DatabaseConnection
-products: Product[*]
+setState(state: AppStateType)
+getCurrentState(): null
AppState
-app: App
+AppState(app: App)
+onEnable()
+onDisable()
+render()
DashboardState
ProductsState
HelpState
DatabaseConnection
+connect()
+getUserProducts(userID: int): Product[*]
Product
-productID: int
-baseCost: int
-name: String
-imageURL: String
-variationGroups: VariationGroup[*]
+Product(id: int, db: DatabaseConnection)
+save(db: DatabaseConnection)
+refresh(db: DatabaseConnection)
VariationGroup
-id: int
-product: Product
-variations: Variation[*]
-groupID: int
+VariationGroup(id: int, db: DatabaseConnection)
+getVariations(): Variation[*]
+addVariation(variation: Variation)
Variation
-id: int
-addedCost: int
-name: String
-imageURL: String
-group: VariationGroup
+Variation(id: int, groupId: int, db: DatabaseConnection)

Section 6: Design trade-offs.

The most impactful design trade-off we decided on was to use a rich-client implementation for the admin side of our application. Rather than storing more data within our database and having the client only issue requests, we decided to hold most of the data on the client's side. This allows for remarkably faster processing assuming the client uses a modern computer. This makes the code relatively simpler and ensures our server is less likely to be overloaded with requests. However, this choice did present a few problems. The client must use significantly more memory since all the database objects are loaded initially, rather than only when needed. This also opens the potential for security risks since accessing and modifying data within the database is handled significantly more on the client's side. Therefore, we have to implement authentication to ensure the request. CRUD operations are legitimate and not malicious. However, this complication is still simpler than the issues present with a less RESTful implementation. In addition, the processing is almost entirely handled on the client's end. This provides uncertainty since we do not have control over the hardware any given user may have. Our code must be lightweight and optimized to be compatible with as many devices as possible. However, this offloads the total processing power required to the individual clients rather than forcing it all onto the server. This saves on cost for the infrastructure on our end since the server is not handling nearly as many requests and data.

Section 7: Software development life cycle model

We used the AGILE model for development, which is meant to be an iterative cycle—allowing individuals to have multiple opportunities to fix issues that will show up during development. It also allows clients to make changes to the project but not alter the overall scope of the project in theory. This model also allows for multiple check-ins to increase communication

for teams. Allowing for blockers and issues to be found before someone struggles too long with it.

This model did help our group understand where each member stood in terms of the project goals and understanding. We were able to access blockers and help each other more by communicating well due to the stand-up meetings with the AGILE development model. We were also able to make sure that we were staying on task and were going to complete tasks that we needed to.

We found some issues with this model when going through our project. One of the problems was the amount of effort it put into documenting all of the artifacts took precious time. That time could have been used to work on the project itself. However, it was often spent typing out the files and going through the motions of the meetings. At times we ended up forgoing the documentation of the stand-ups in particular because those were so frequent and fast that it took longer to put the document together than just to explain issues and planned work.

SCRUM Artifacts - https://github.com/423s22/G4/tree/main/SCRUM