Bilkent University
Department of Computer Engineering

Senior Design Project

Kalas-Iris: Clothes recognition and rich attribute prediction using computer vision service for online clothing retail

Low Level Design Report

Olcay Akman, Doğaç Eldenk, Zeynep Korkunç, Hüseyin Ata Atasoy

Supervisor: Ayşegül Dündar

Low Level Design Report

1 February, 2021

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction

Online shopping is becoming more and more popular. There are around 7.1 million online retailers in 2020 and 1.8 billion people are shopping online each year. The demand on online shopping is requiring more resources to be spent on customers. There are two main ways customers find the goods that they buy in those websites: Browsing and searching. According to Oberlo [1], 75% of the search queries of the customers are brand new. Therefore, it is important for customers to find the good that they are searching for.

Currently big e-commerce websites such as Hepsiburada use textual matching for searching products. To get better results, most of the products are labeled and categorized. In addition to those labels, categories and descriptions, there are additional handwritten rules for better customer experience. This is very expensive and hard to maintain even for big e-commerce websites. Currently the search results of those products are not inspired from the images of the product. Machine learning is only used for showing relevant products to another product, which is affected by the behavior of the customers.

Most of the current search engines of small e-commerce websites recommend a good match to the given search query via matching the search queries with the labels associated with each product. Although successful to provide the customer with the products they are looking for, such services lack a high rate of accuracy. This mainly stems from the fact that small e-commerce owners do not have the flexibility to spend extensive time and effort on keeping a thorough and detailed log for their inventory on a digital platform. Required information about goods are put in the system by hand and sometimes directly gathered from the distributor. When the information lacks in the system, workers add the description based on the image of the product. Additionally, the products have different filters and labels. Labeling these products is time consuming and there can be missing information. The amount of detail in those descriptions also can be limited. Additionally, there can be semantic problems such as synonyms or similar words. In order to overcome those problems, we want to create an algorithm that can create descriptions and labels for items and also can show the best matching products according to a search query. We expect this algorithm to benefit from mostly the images of the products and additional descriptions.

This report is a detailed explanation about the low level design of the project, covering the technical details of the subsystem decomposition and contains a thoroughly explained UML diagram for its classes. Engineering decisions for this project are also evaluated in the Object Design Trade-offs section, section 1.1.

1.1 Object Design Trade-offs

- Development Time and Resources vs Flexibility

Fashion items can be identified by many aspects. We are primarily identifying a set of predefined categories and attributes. Each customer might want to have their set of categories and labels. In order to allow that we should have a more flexible machine learning model. It is time consuming and expensive for us to train such a model per customer. We will create a more generalized model which will have a vast amount of categories and attributes associated with it. If our customer provides us a big dataset with their labels/categories our model can be trained on those futures and labels. Additionally not every e-commerce platform uses a standard category/labeling system. They might have separate fields other than attributes. Covering all those fields comprehensively requires a vast amount of time and resources.

- Speed and Cost vs Functionality

We are sacrificing our response speeds for image annotation services for more accurate results and low costs. Our server design is asynchronous to the customers so that they are not expecting an immediate response from us. So that we can run the machine learning model on slower/cheaper cloud instances on certain times of the days using cron jobs. We can queue requests from our customers and we can run them in batches so that it will take less time and money. But we are not sacrificing any functionality. We are trying to use the best machine learning models that we can run with the best results.

- Compatibility vs Cost

Providing the services offered by Kalas-Iris to a certain platform, i.e. to e-commerce websites powered by Wix eCommerce [1], increases its compatibility to a certain number of e-commerce websites. When more platforms are supported, the compatibility of the service is increased. This, however, results in more work, more lines of code and more resources utilized, which all add up to an increased cost.

1.2 Interface documentation guidelines

Below we provide the interface documentation guideline which is given for each class in the UML diagram. This format is used in section 3 of this report, where the class interfaces of classes given in the diagrams in section 2 of this report.

Class Name	
Description	
Attributes	
type name	description
Operations	
functionName(params) : Return type	description

Table 1: The class interface table guideline.

1.3 Engineering standards

This report uses UML standards for class diagrams and IEEE standards for citations. We have used agile development for our development and we had a sprint driven development schedule using kanban board. For formatting our code we have complied to pep8 and eslint standards.

1.4 Definitions, acronyms, and abbreviations

- **Fashion Item:** Any type of clothing item that is sold in stores.
- **MMFashion:** The fashion annotation library used to annotate fashion images. [2][3]
- **Landmark Detection:** The detection of the landmarks of a clothing item from its image (i.e., its collar, hemline, etc.)
- **Attribute:** Attributes of a clothing product, detected via MMFashion (i.e., floral, long sleeve, striped, etc.)
- **Category:** Category of the fashion item, detected via MMFashion (i.e., summer dress, wedding dress, sneakers etc.)
- **Customer:** The e-commerce website owner who uses the Kalas-Iris services, unless otherwise stated (in some cases, those who purchase retail goods from e-commerce websites are also called customers, but in such cases it is explicitly stated. Otherwise, ‘customer’ has the meaning explained here).

2 Packages

For each subsystem of the project, there is a separate package containing classes, and in some cases, other packages. The entire subsystem decomposition and detailed package information is provided below in Figure 1.

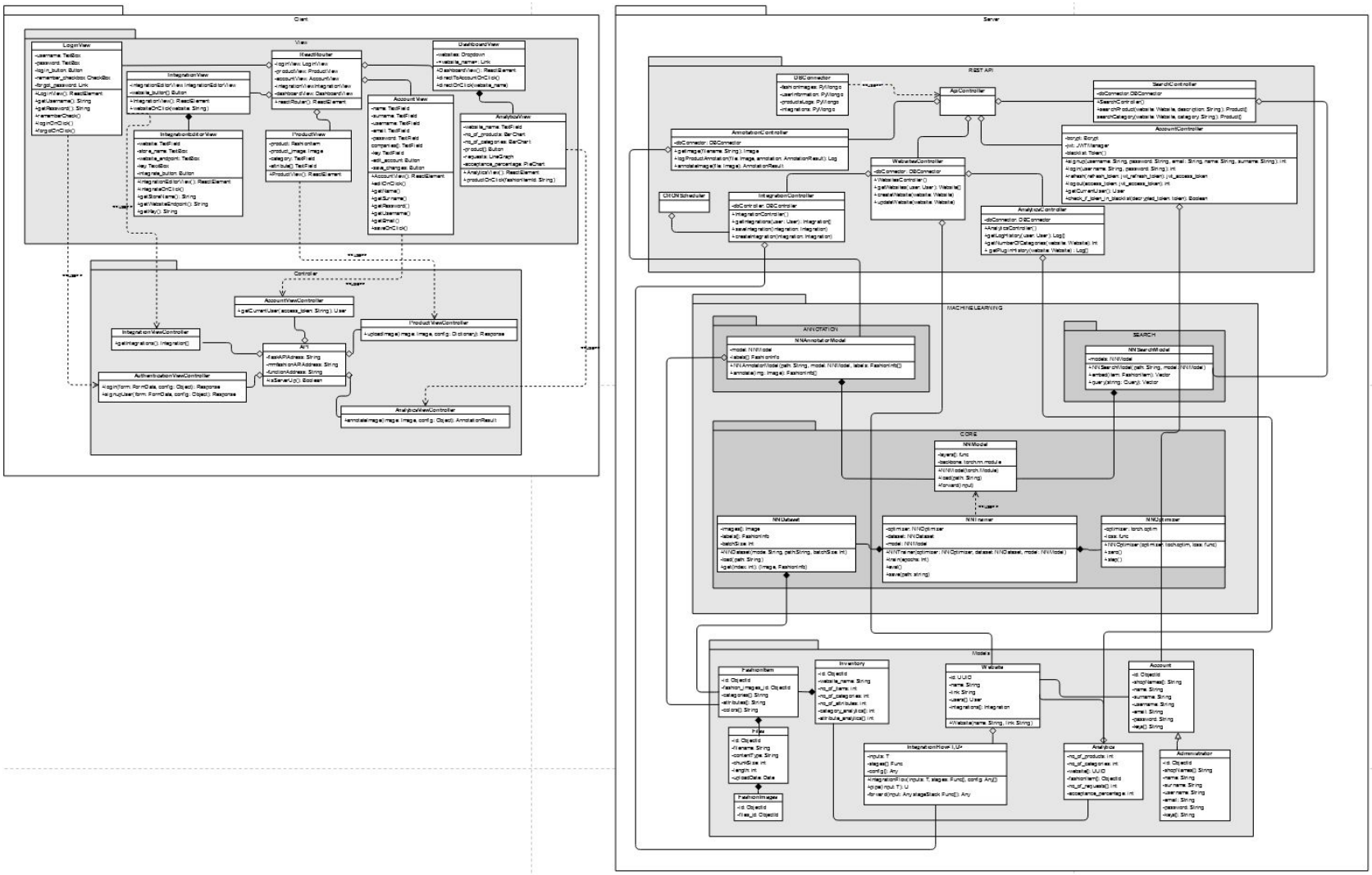

Figure 1: The entire subsystem decomposition.

2.1 Client Package

Client subsystem of the Client-Server architecture is composed of two subsystems, named View and Controller.

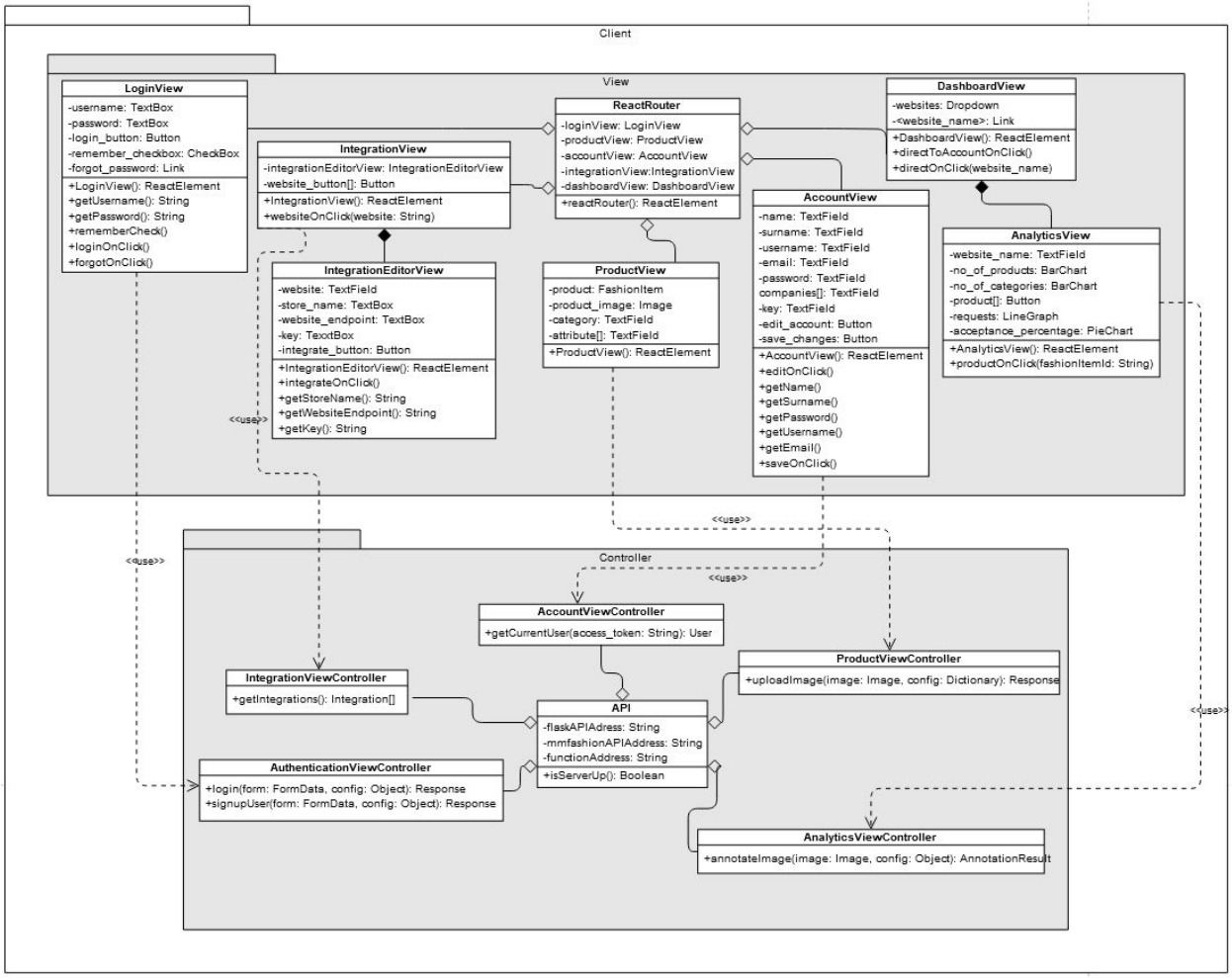

Figure 2: The Client package composition

2.1.1 View Package

The View subsystem consists of several views related to the front-end side of the API. They are all controlled by the ReactRouter. The views are:

- LoginView: The page the customers use to login or signup to Kalas-Iris service.
- ProductView: The page which shows the customers the products they have added to Kalas-Iris.

- IntegrationView: The page which aids the customer to integrate the Kalas-Iris service to their own e-commerce website.
- IntegrationEditorView: The visual editor tool for managing the workflow of the interactions between Kalas-Iris and e-commerce websites.
- AccountView: The page where the customer views and changes the details about their account on Kalas-Iris.
- DashboardView: The page which greets the customers. This page contains a brief information about the overall performance and metrics of the system. It has quick access buttons and mini views to help the user navigate.
- AnalyticsView: A detailed analytics page which shows the customer daily/weekly/monthly statistics. Those statistics can be a number of queries, system logs, system performance etc.

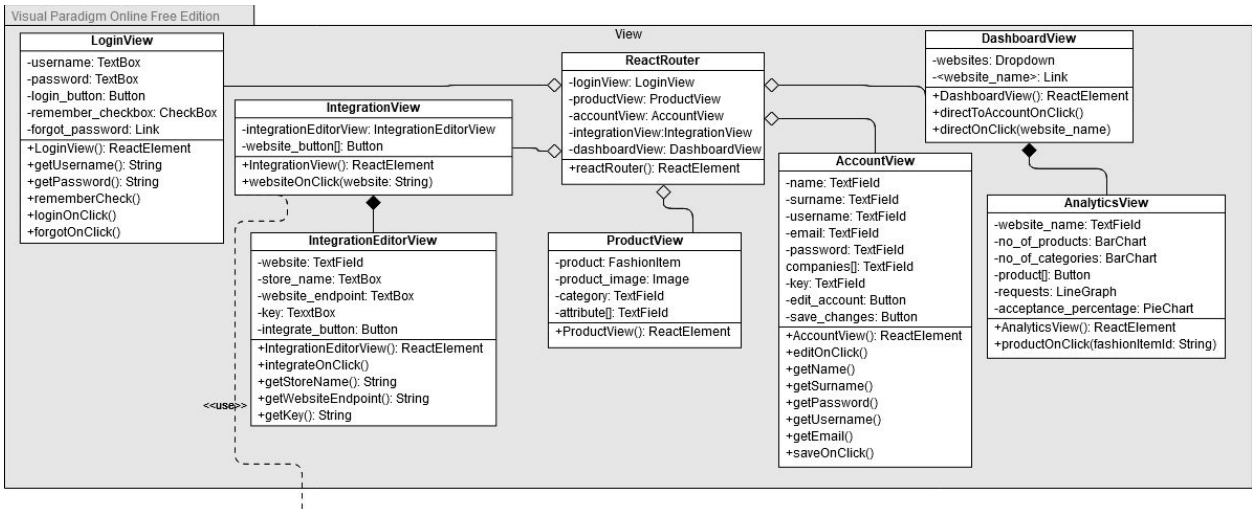

Figure 3: The View package composition.

2.1.2 Controller Package

The controller subsystem of Client consists of the controllers for the views mentioned in View. All those controllers are managed by the ApiConnectorController. The controllers are:

- AuthenticationViewController: Connects the authentication API with the front-end. Manages the states of the views.
- IntegrationViewController: Connects the integration API with the front-end. Manages the states of the views.
- AccountViewController: Connects the account information stored in the backend with the state of the view.
- ProductViewController: Controls the state of the products fetched from the backend and manages it's view on the frontend.

- AnalyticsViewController: Manages the analytics fetched from the backend and the state of the frontend widget.

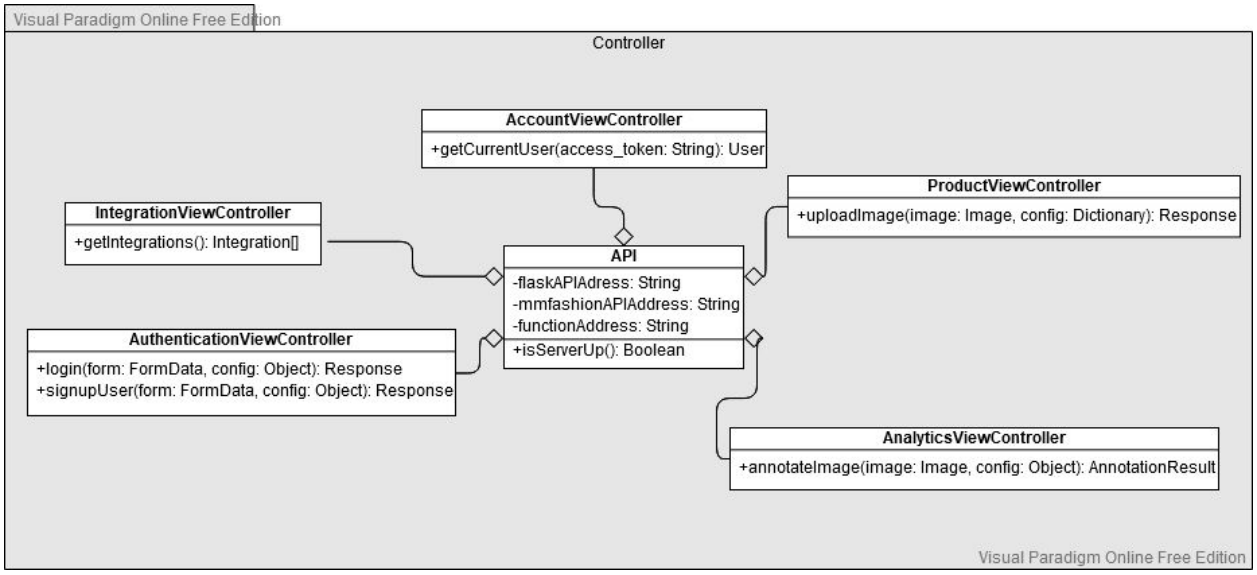

Figure 4: The Controller package decomposition.

2.2 Server Package

The server part of the Client-Server architecture deals with the back-end part of the Kalas-Iris service. It consists of the following subsystems:

- RestAPI, which controls the API's backend and communicates with the Front-end.
- Machine Learning, which controls the fashion annotation and semantic search part of the Kalas-Iris service.
- Models, which controls the models generated by Kalas-Iris and their representation in the Kalas-Iris website.

The REST API subsystem of the Client consists of the controllers for the Back-end service which will handle the data that is fetched from the machine learning and model subsystems and it communicates with the Server side in order to display the data. The controllers are managed by the ApiController as shown.

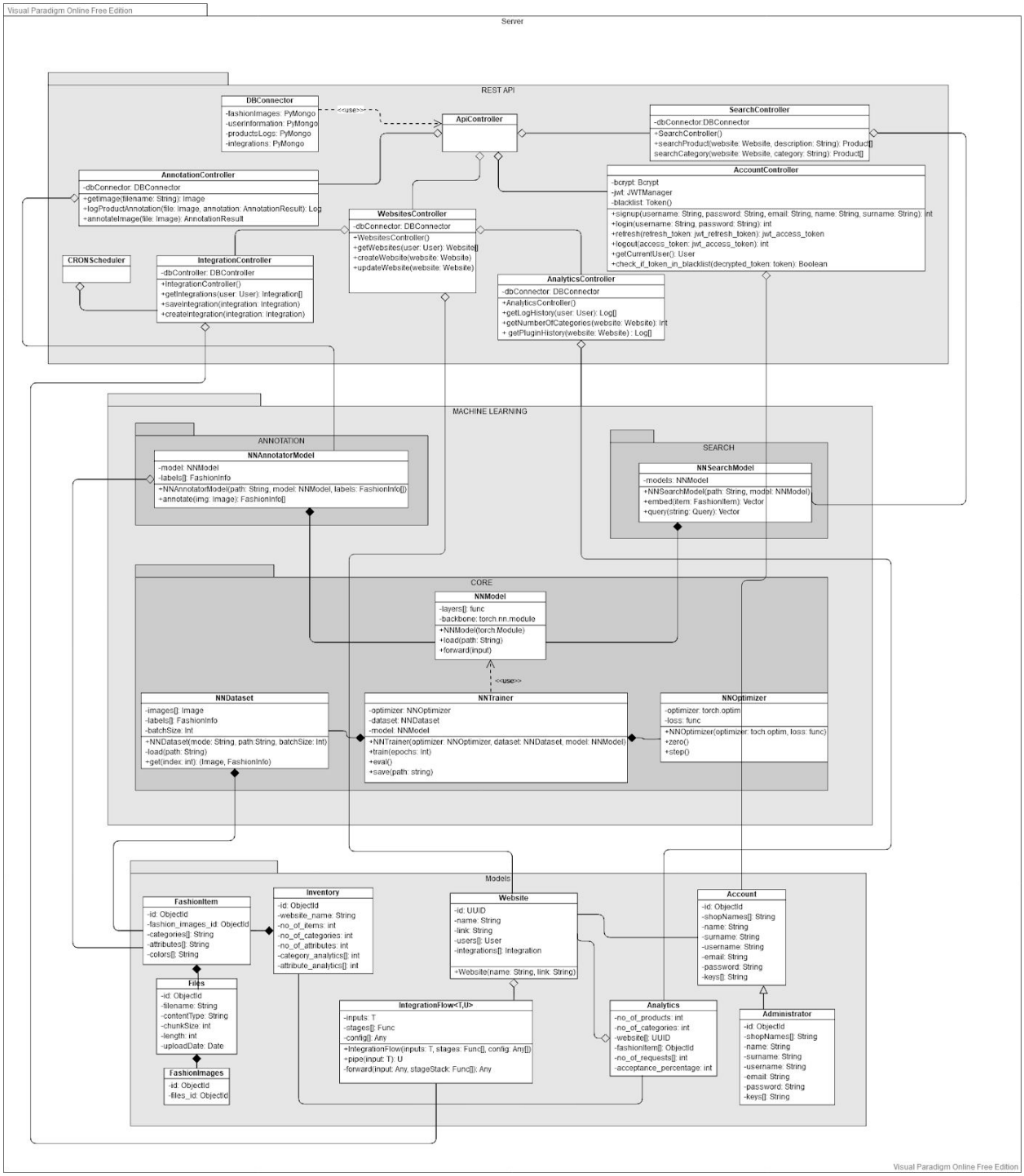

Figure 5: The Server package decomposition.

2.2.1 REST API

The classes in the REST API package are:

- AnnotationController: Fetches the data obtained from the fashion annotator and communicates with the ApiController to manage this data.
- SearchController: Fetches the data obtained from our search model and communicates with the ApiController.
- AccountController: Fetches the data related to the customers account from the model subsystem and sends the obtained results to the ApiController.
- WebsitesController: Communicates with both IntegrationController and AnalyticsController in order to obtain the related results about the website. Returns the results to the ApiController.
- AnalyticsController: Fetches data about the website analytics from the models subsystem and returns the data to the WebsitesController.
- IntegrationController: Controls the integration of the Kalas-Iris service to the customer's e-commerce website.
- CRONScheduler: A job scheduler for scheduling certain tasks if needed.
- DBConnector: Maintains the database connections of the REST API.

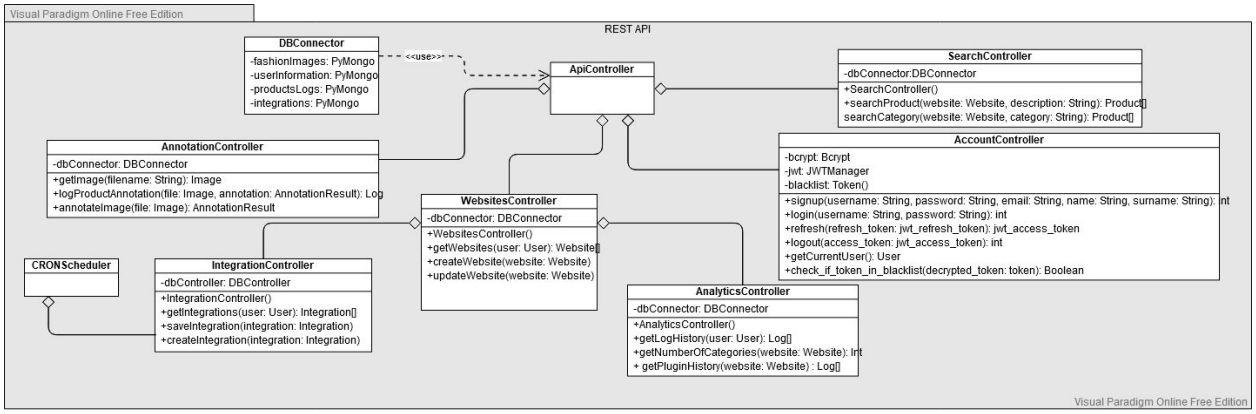

Figure 6: The REST API package decomposition.

2.2.2 Machine Learning

The Machine Learning package is composed of three packages, Annotation, Search and Core.

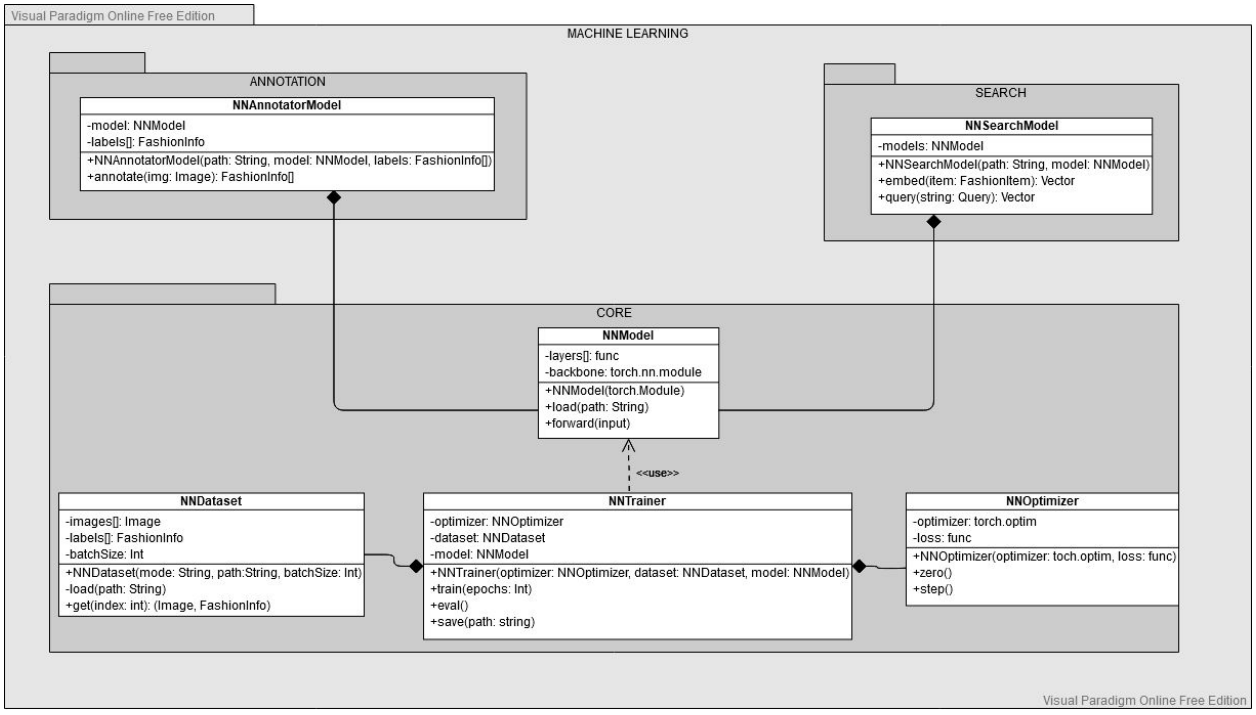

Figure 7: The Machine Learning package decomposition.

2.2.2.1 Annotation

The class in the Annotation package is:

- NNAnnotaterModel: Takes a fashion image and finds the attributes it is associated with. Also predicts the category of the image.

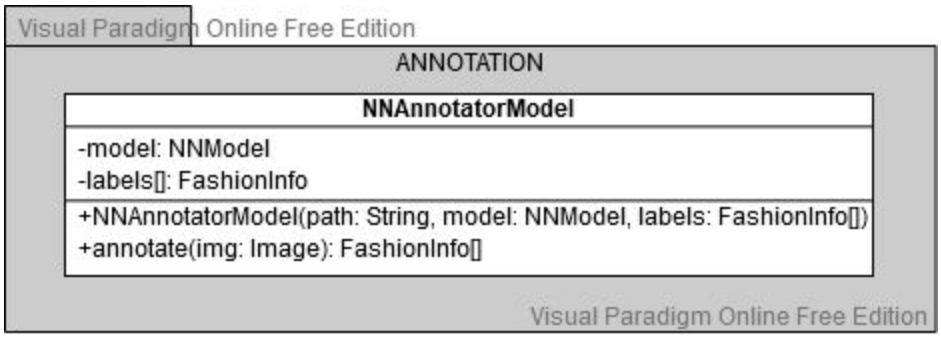

Figure 8: The Annotation package decomposition.

2.2.2.2 Search

The class in the Search package is:

- NNSearchModel: Takes a query and returns a list of images, which corresponds to the best matches to the query.

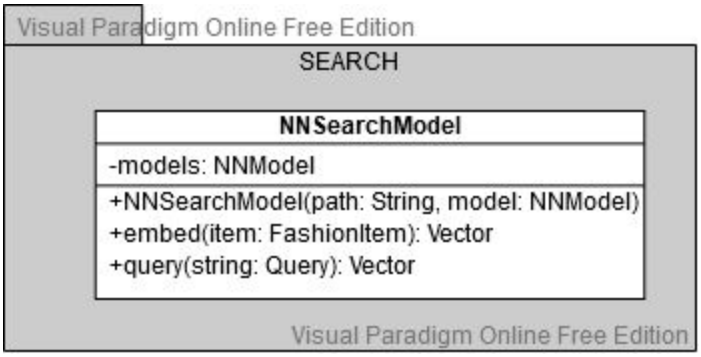

Figure 9: The Search package decomposition.

2.2.2.3 Core

The classes in the Core package are:

- NNDataset: The dataset we are using to train our models. We are currently using DeepFashion, a detailed dataset containing categories and attributes matched with clothes.
- NNOptimizer: An optimizer which uses modern algorithms such as Adam optimizer to help train our model.
- NNTrainer: The trainer class which is responsible for training the model and selecting the best model among the trained models. It trains the data coming from NNDataset and uses the optimizer defined in NNOptimizer.
- NNModel: A generic model which takes a tensor as an input, which is an 3d rgb image tensor in our case, and outputs a tensor, which corresponds to one hot encoding of the categories and attributes.

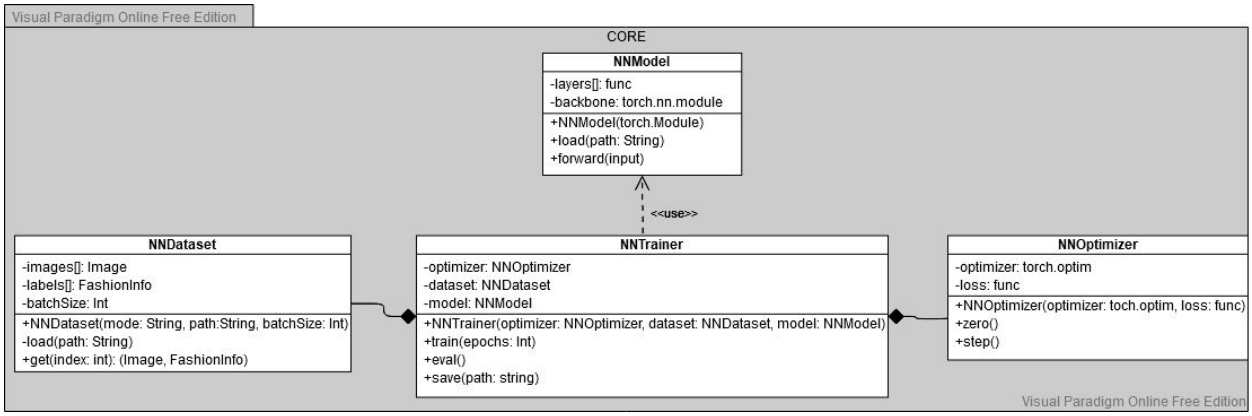

Figure 10: The Core package decomposition.

2.2.3 Models

The classes in the Models package are:

- FashionItem: A generic definition of a fashion item. Must contain an image, attributes and categories associated with it.
- Inventory: A collection of fashion items. Represents the products in the e-commerce website.
- IntegrationFlow: A model storing the flow of actions when there is a certain event happening such as an annotation request comes etc.
- Website: A model which contains information about the website, analytics, integrations and the inventory management systems.
- Analytics: A model which contains certain metrics such as performance, queries or logs.
- Account: A model for storing the user.

- Administrator: A super user that can manage all the possible websites. This is helpful for testing and helping clients in certain problems.

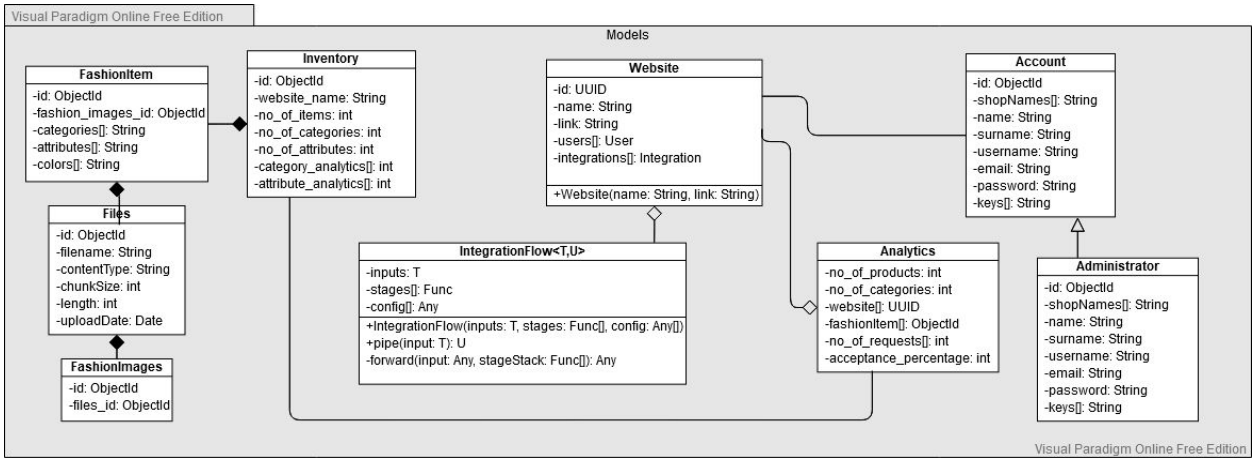

Figure 11: The Models package decomposition.

3 Class Interfaces

Below we provide the detailed information about class interfaces presented in the diagrams throughout section 2, where packages for Kalas-Iris are provided.

3.1 Client Package

3.1.1 View

ReactRouter	
This class manages the UI instances of Kalas-Iris.	
Attributes	
loginView: LoginView	LoginView that will be managed by ReactRouter
productView: ProductView	ProductView that will be managed by ReactRouter
integrationView: IntegrationView	IntegrationView that will be managed by ReactRouter
accountView: Accountview	AccountView that will be managed by ReactRouter
dashboardView: DashboardView	DashboardView that will be managed by ReactRouter
Operations	

reactRouter() : ReactElement	It will render and create a react application.

LoginView	
This class creates a view for the login page.	
Attributes	
username: TextBox	Text box for entering the username
password: TextBox	Text box for entering the password
login_button: Button	Button for logging in
remember_checkbox: CheckBox	Checkbox for remembering the username and password
forgot_password: Link	Link that redirects to the forgot password page.
Operations	
LoginView(): ReactElement	It will render and return the login page.
getUsername(): String	Returns the username.
getPassword(): String	Returns the password.
rememberCheck()	Activates the remember me functionality.
loginOnClick()	Activates the login process.
forgotOnclick()	Redirects to the forgot password page.

ProductView	
View class that displays the annotated product.	
Attributes	
product: FashionItem	Fashion item that will be displayed on this view.
product_image: Image	Image of the product.
category: TextField	Text field that shows the category of the product.
attribute[]: TextField	List of text fields that shows the attributes of the product.
Operations	

ProductView(): ReactElement	It will render and return the product page.

IntegrationView	
This class creates a view for e-commerce website integrations.	
Attributes	
integrationEditorView: IntegrationEditorView	IntegrationEditorView that will be managed by this class.
website_button[]: Button	Displays the e-commerce websites that Kalas-Iris allows to integrate.
Operations	
IntegrationView(): ReactElement	Renders and returns the integration page.
websiteOnClick(website: String)	Redirects to the specified website's integration editor page.

IntegrationEditorView	
Creates a view for an editor that will display customized preferences for the specified website integrations.	
Attributes	
website: TextField	Displays the name of the website.
store_name: TextBox	Name of the store that user wants to integrate.
website_endpoint: TextBox	Endpoint URL of the website that will be integrated.
key: TextBox	A key that was provided by Kalas-Iris in order to use the functionalities.
integrate_button: Button	A button for website integration.
Operations	
IntegrationEditorView(): ReactElement	Renders and returns the integration editor page.
integrateOnClick()	Initiates the integration process.

getStoreName(): String	Returns the store name.
getWebsiteEndpoint(): String	Returns the website endpoint.
getKey(): String	Returns the key.

AccountView	
This class creates a view for the account information for the user.	
Attributes	
name: TextField	Displays the name of the user.
surname: TextField	Displays the surname of the user.
username: TextField	Displays the username of the user.
email: TextField	Displays the email of the user.
password: TextField	Displays the password of the user.
companies[]: TextField	Displays the names of the stores that the user owns.
key: TextField	Display the key that provided by Kalas-Iris
edit_account: Button	Edit account button.
save_changes: Button	Save changes button.
Operations	
AccountView(): ReactElement	Renders and returns the account page.
editOnClick()	Redirects to an editable account page.
getName(): String	Returns the new name that the user enters.
getSurname(): String	Returns the new surname that the user enters.
getUsername(): String	Returns the new username that the user enters.
getPassword(): String	Returns the new password that the user enters.
getEmail(): String	Returns the new e-mail that the user enters.
saveOnClick()	Saves the changes.

DashboardView	
The dashboard where the user can see the fashion items they have annotated.	
Attributes	
account: Link	Directs the page to AccountView
websites: Dropdown	The items within the dropdown are names of the shop websites the account is linked to, each name links to the related AnalyticsView of the shop website.
<website_name>: Link	The website name the account owner linked to Kalas-Iris. Links to the AnalyticsView of that shop website.
Operations	
DashboardView(): ReactElement	Renders and returns the dashboard view.
directToAccountOnClick()	Directs to the AccountView of the user name when clicked.
directOnClick(website_name)	Directs to the analytics view of the website name when clicked.

AnalyticsView	
View page that displays the analytics for an e-commerce website that user has integrated.	
Attributes	
website_name: TextField	Displays the website name.
no_of_products: BarChart	Displays the number of logged products.
no_of_categories: BarChart	Displays the number of categories and number annotated products of that category.
product[]: Button	Displays the annotated products.
requests: LineGraph	Displays a graph of requests per hour.
acceptance_percentage: PieChart	Displays the acceptance percentage.
Operations	
AnalyticsView(): ReactElement	Renders and returns the analytics view.
productOnClick(fashionItemId: String)	Redirects to the product page when clicked.

3.1.2 Controller

API	
The controller API of the React App. Includes the functions to interact with the backend services.	
Attributes	
- flaskAPIAdress : String	Base URL for the Flask API.
- mmfashionAPIAddress: String	Base URL for the cloud deployment of the MMFashion service.
- functionAddress: String	Base URL for the Google Cloud API.
Operations	
+isServerUp(): Boolean	Makes a GET request to the MMFashion service to check if the server is online.

AccountViewController	
Controller class for the account page.	
Attributes	
Operations	
+getCurrentUser(access_token: String): User	Makes a GET request to the '/getCurrentUser' endpoint of the Flask API to retrieve information about the currently logged in user

IntegrationViewController	
Controller class for the integration page.	
Attributes	
Operations	
+getIntegrations(): Integration[]	Makes a GET request to the '/integrations' endpoint of the Flask API to retrieve information about the integrated web pages.

AuthenticationViewController	
Controller class for the authentication view.	
Attributes	
Operations	
+login(form: FormData, config: Object): Response	Makes a POST request to the '/login' endpoint of the Flask API for login operation.
+signupUser(form: FormData, config: Object) : Response	Makes a POST request to the '/signup' endpoint of the Flask API to signup a user to Kalas-Iris.

ProductViewController	
Controller class for the product page.	
Attributes	
Operations	
+uploadImage(image: Image, config: Dictionary) : Response	Makes a POST request to the '/uploadProductImage' of the Flask API, with a fashion image.

AnalyticsViewController	
Controller class for the analytics view.	
Attributes	
Operations	
+annotateImage(image: Image, config: Object) : AnnotationResult	Makes a POST request to the '/annotate' endpoint of the MMFashion service, with a fashion image.

3.2 Server Package

3.2.1 Rest API

AccountController	
Flask Blueprint to handle user account related operations.	
Attributes	
- bcrypt: Bcrypt	Bcrypt wrapper for the Flask app. Bcrypt is used to encrypt the passwords of the user.
- jwt: JWTManager	JWTManager wrapper for the Flask app. JWT is used to create and manage access and refresh tokens for the users.
- blacklist: set()	Set of blacklisted authentication and refresh tokens.
Operations	
+ signup(username: String, password: String, email: String, name: String, surname: String) : int	Signup logic for the POST request at the '/signup' endpoint. Registers the values to the users collection in the database.
+ login(username: String, password: String) : int	Login logic for the POST request at the '/login' endpoint. Returns (access_token, refresh_token) if the provided credentials match a user in the users collection.
+ refresh(refresh_token: jwt_refresh_token): jwt_access_token	Returns a new access token if refresh_token is valid.
+ logout(access_token: jwt_access_token): int	Blacklists the access_token of the user.
+ getCurrentUser(): User	Returns the identity of the currently logged in user. Identity of the user is the (access_token, username) tuple.

+check_if_token_in_blacklist(decrypted_token : token) : Boolean	Checks whether the decrypted_token is in the blacklist or not.

AnnotationController	
Flask Blueprint to handle operations related to the image annotation service.	
Attributes	
- dbConnector: DBConnector	Used to access the database.
Operations	
+ getImage(filename: String): Image	Retrieves the file with the given filename from the fashion_image_collection collection in the database.
+ logProductAnnotation(file: Image, annotation: AnnotationResult): Log	Logs the product image and its annotation to the database.
+ annotateImage(file: Image): AnnotationResult	Posts the fashion image to the MMFashion service and retrieves its annotation results.

DBConnector	
A Class for combining the MongoDB databases in a single file.	
Attributes	
+fashionImages : PyMongo	The fashionImages database in the MongoDB cluster.
+userInformation: PyMongo	The userInformation database in the MongoDB cluster.
+productsLogs: PyMongo	The productLogs database in the MongoDB cluster.
+integrations: PyMongo	The integrations database in the MongoDB cluster.
Operations	

SearchController	
Flask Blueprint which handles search operations throughout the product catalog.	
Attributes	
-dbConnector: DBConnector	DBConnector used to access the database.
Operations	
+SearchController()	Creates a new search controller.
+ searchProduct(website: Website, description: String): Product[]	Retrieves products in the product database with the given description.
+ searchCategory(website: Website, category: String): Product[]	Retrieves all products from a category.

IntegrationController	
Flask Blueprint for handling e-commerce website integrations.	
Attributes	
-dbConnector: DBConnector	DBConnector it uses to access the db.
Operations	
+IntegrationController()	Creates a new integration controller.
+getIntegrations(user: User) : Integration[]	Fetches all the integrations associated with the user.
+saveIntegration(integration: Integration)	Updates the configuration of the integration and updates it.
+createIntegration(integration: Integration)	Creates a new integration.

AnalyticsController	
Flask Blueprint for handling user analytic data.	
Attributes	
-dbConnector: DBConnector	DBConnector it uses to access the db.
Operations	
+AnalyticsController()	Creates a new analytics controller.
+ getLogHistory(user: User) : Log[]	Retrieves all of the products that have been logged to all platforms.
+ getNumberOfCategories(website: Website) : Int	Returns the number of product categories that have been logged so far.
+ getPluginHistory(website: Website) : Log[]	Retrieves the product information for a specific e-commerce plugin.

WebsitesController	
Flask Blueprint for managing e-commerce website related operations.	
Attributes	
-dbConnector: DBConnector	DBConnector it uses to access the db.
Operations	
+WebsitesController()	Creates a new website controller.
+getWebsites(user: User): Website[]	Gets all the websites tied to a user
+createWebsite(website: Website)	Creates a new website.
+updateWebsite(website: Website)	Updates the current website with the new configuration.

3.2.2 Machine Learning

NNDataset	
Pytorch Dataset class that contains images and attributes for training	
Attributes	
-images: Image[]	Array of test/training images.
-labels: FashionInfo[]	Detailed information of corresponding test/training images.
-batchSize: Int	Batch size of the data loader.
Operations	
+NNDataset(mode: String, path: String, batchSize: Int)	Constructor for the dataset. Mode can be tested or trained.
-load(path: String)	Loads the dataset and stores it in images and labels arrays.
+get(index: int): (Image, FashionInfo)	Gets the next tuple in the dataset.

NNOptimizer	
A class that bundles torch.optim and functionality for parameter tuning.	
Attributes	
-optimizer: torch.optim	The torch.optim for optimizing the model.
-loss: func	Loss function for the optimizer.
Operations	
+NNOptimizer(optimizer: torch.optim, loss: func)	Initializer with optimizer and loss function.
+zero()	Zeros grads for the next batch of training input.
+step()	Calculates the loss and steps the optimizer.

NNModel	
A generic pytorch NN Module class.	
Attributes	
-layers: func[]	The linear and convolutional layers for the neural network
-backbone: torch.nn.module	This backbone is a model that is pre-trained on ImageNet1000 dataset. This will be used as a feature extractor. Example: VGG16, AlexNet, ResNet or EfficientNet.
Operations	
+NNModel(torch.Module)	Constructor for initializing a neural network.
+load(path: string)	Load a pre-trained model's parameters.
+forward(input)	Forwards the inputs through its layers and calculates gradients.

NNTrainer	
A generic trainer model for training the machine learning model.	
Attributes	
-optimizer: NNOptimizer	A generic optimizer such as SGD or ADAM.
-dataset: NNDataset	The dataset which will be used to train the neural network.
-model: NNModel	The model for the neural network which wlll be trained.
Operations	
+NNTrainer(optimizer: NNOptimizer, dataset: NNDataset, model: NNModel)	Constructs a trainer object with the given optimizer to train the given model on the given dataset.
+train(epochs: Int)	Trains the model with the given amount of epochs.
+eval()	Runs a small evaluation test on the model to measure the current accuracy.
+save(path: string)	Saves model into a '.pth' file for later use.

NNAnnotatorModel	
This is the backbone of the image annotation service. It generates labels for a given image.	
Attributes	
-model: NNModel	The neural network which will be used to evaluate the input.
-labels: FashionInfo[]	The output labels gathered from the dataset.
Operations	
+NNAnnotatorModel(path: String, model: NNModel, labels: FashionInfo[])	The path will contain the pre-trained model's '.pth' file for using.
+annotate(img: Image): FashionInfo[]	Annotates the given image from a list of predefined labels using the model.

NNSearchModel	
This is the backbone of the semantic search service. It generates embeddings for products.	
Attributes	
-model: NNModel	The neural network which will be used to evaluate the input.
Operations	
+NNSearchModel(path: String, model: NNModel)	The path will contain the pre-trained model's '.pth' file for using.
+embed(item: FashionItem): Vector	Gets a fashion item and embeds it into a lower dimensional space.
+query(string: Query): Vector	Converts the query given into a vector lying in a lower dimensional space for searching.

3.2.3 Models

FashionItem	
Holds data about each fashion item annotated.	
Attributes	
id: ObjectId	ObjectId of the FashionItem.
fashion_images_id: ObjectId	ObjectId of the files containing the fashion
categories[]: String	The categories associated with the FashionItem after annotation
attributes[]: String	The attributes associated with the FashionItem after annotation
colors[]: String	The colors associated with the FashionItem after annotation
Operations	

FashionImages	
Holds the file information about files of fashion images.	
Attributes	
id: ObjectId	ObjectId of the FashionImages
Files_id: ObjectId	ObjectId of the Files
Operations	

Files	
Holds the file information of each file that holds the fashion images specific to a single FashionItem.	
Attributes	
id: ObjectId	ObjectId of the FashionImages
filename: String	Name of the file
contentType: String	Content type of the file

chunkSize: int	Size of the file
length: int	Length of the file
uploadDate: Date	Upload date of the file
Operations	

Inventory	
The inventory data of each user, consists of inventory information.	
Attributes	
id: ObjectId	ObjectId of the inventory
website_name: String	The name of the website the inventory belongs to
no_of_items: int	The number of fashion items in the inventory
no_of_categories: int	The number of categories in the inventory.
no_of_attributes: int	The number of attributes in the inventory.
category_analytics[]: int	The number of items per category.
attribute_analytics[]: int	The number of items per attribute.
Operations	

IntegrationFlow<T, U>	
A simple integration flow for a website integration plan. The input to this integration is T. T can be for example an object containing a webhook url and an image. The user provides stage functions. For example, the first stage preprocesses the image and forwards it to a second function. The second function sends a request to the webhook url and returns U. In this case U can be a response.	
Attributes	
-inputs: T	Any input that is forwarded to this integration
-stages: Func[]	Any function that is piped and applied to the input. Note that consecutive functions outputs should be equal to the next functions input.
-config: Any[]	A configuration of constants and variables to be used inside stages.

Operations	
+IntegrationFlow(inputs: T, stages: Func[], config: Any[])	Creates a new integration flow from the given inputs and the stages.
+pipe(input: T): U	Transforms the given input T by forwarding it into the stages and fetches the final output U.
-forward(input: Any, stageStack: Func[]): Any	Takes any input and pipes it through one stage. After it is done it pops the stageStack and recursively calls until all stages are finished.

Website	
Holds the information about the user's e-commerce website.	
Attributes	
id: UUID	Unique identifier of the website.
name: String	Name of the website.
link: String	Website link.
users[]: User	Owners of the website.
integrations[]: Integration	Integrations to the website.
Operations	
Website(name, link)	Creates a new website.

Analytics	
Holds the data about analytics.	
Attributes	
no_of_products: int	Number of products logged.
no_of_categories: int	Number of categories predicted.
website[]: UUID	The identifier of the website.
fashionItem[]: ObjectId	The identifier of each FashionItem.
no_of_requests[]: int	Number of requests made per hour.

acceptance_percentage: int	Acceptance percentage of products.
Operations	

Account	
Holds the account details and information for each user.	
Attributes	
id: ObjectId	ObjectId of the user
shopNames[]: String	Names of the e-commerce shops the account is used for.
name: String	Name of user
surname: String	Surname of user
username: String	Username of the user, it is unique
email: String	Email of the user, it is unique
password: String	Password of the account
keys[]: String	Keys for each shop the account owner is to use during integration. It is unique for each shop.
Operations	

Administrator	
Holds the account information about the super account, the admin account.	
Attributes	
id: ObjectId	ObjectId of the user
shopNames[]: String	Consists of all the shops registered in Kalas-Iris. This account can access them all since it is a super account.
name: String	Name of user
surname: String	Surname of user
username: String	Username of the user, it is unique

email: String	Email of the user, it is unique
password: String	Password of the account
keys[]: String	Keys for all the shops registered in Kalas-Iris. It is unique for each shop.
Operations	

4 References

[1] Wix. "Wix eCommerce." *Wix*, 2021, https://www.wix.com/ecommerce/website. Accessed 01 02 2021.