9.X: Optional Implementation/ Comparison with other Implementation models
Rayan Bouaazzi

/* Comparison of our Database Operations to others. The coding language for this implementation is golang version 1.19 and this serves as a baseline example of how our backend service would make calls and query the pre-constructed databases containing supplier, product and store information within them. */

These examples show how we expect to implement querying availability and reducing cache internally. The examples used also serve the purpose of highlighting the implementation methods used in other areas of the field of our project, so these examples have dual purposes.

The code below is a rough example of how our operations would run when querying a database for customer and supplier information that is retrieved and then used to create the path, the front-end logic will receive this information from the backend and use it for the User Interface side of the application.

/* The function below represents the implementation of how our software would query the database to retrieve the request of items from the user and returning that request based on what is returned from the database */

```
func ValidRequest(context echo.Context, dbCall *db.query, customerAccount, supplierName, productName,
optionName, postRequest interface{}) error {
 if supplierID == "" || productID == "" {
   return obs.NewError("request is missing a supplier or a product", false, "")
 }

 if mode == http.MethodPost {
   return validateRequest(customerAccount, supplierName, productName, optionName,
postRequest.(obs.PostAvailabilityRequest), context, dbCall)
 }
 return validateGetRequest(customerAccount, supplierName, productName, optionName,
postRequest.(obs.PostAvailabilityRequest), context, dbCall.backendRequest)
}
```

/* This function queries the database to check and see if a particular item at the stores location is available or sold out or limited */

```
func getItemAvails(vacancy interface{}) store.AvailabilityStatus {
 if vacancy.(int64) < common.LimitedCapacity && vacancy.(int64) > common.ZeroCapacity {
   return store.LimitedAvailStatus
 } else if vacancy.(int64) == common.ZeroCapacity {
   return store.NoAvailCapacity
 }
```

```
 return store.AvailableStatus
}


/* This function is how we intend to make database calls to retrieve product information from a distinct
location */

func (i *InventoryAPIService) getProducts(supplierString string) (*store.Products, error) {
 params := &QueryParams{
   SupplierID: supplierString,
 }
 storeURL := retURL(i.client.cfg.BasePath, store.ProductsEndpoint)

 data, err := i.client.doRequest(http.GET, storeURL, "", params)
 if err != nil {
   return nil, store.NewError(err.Error(), false, err.Error())
 }
// sends the error to a logger so that we can monitor errors being sent from the database
 logRequestResponse(metrics.GetStoreProducts, storeURL, data)

 return &response, nil
}

// this function sets a new query parameter
func (c *context) StoreQuery(storeName string) string {
if c.query == nil {
 c.query = c.request.URL.Query()
}
return c.query.Get(name)
}

// These functions below is where we set query param values to call the database using the stores context
values
func (c *storeContext) ParamNames() []string {
return c.pnames
}

func (c *storeContext) SetParamNames(names ...string) {
c.pnames = names
*c.echo.maxParam = len(names)
}

func (c *storeContext) ParamValues() []string {
return c.pvalues[:len(c.pnames)]
}

func (c *storeContext) SetParamValues(values ...string) {
c.pvalues = values
```

}