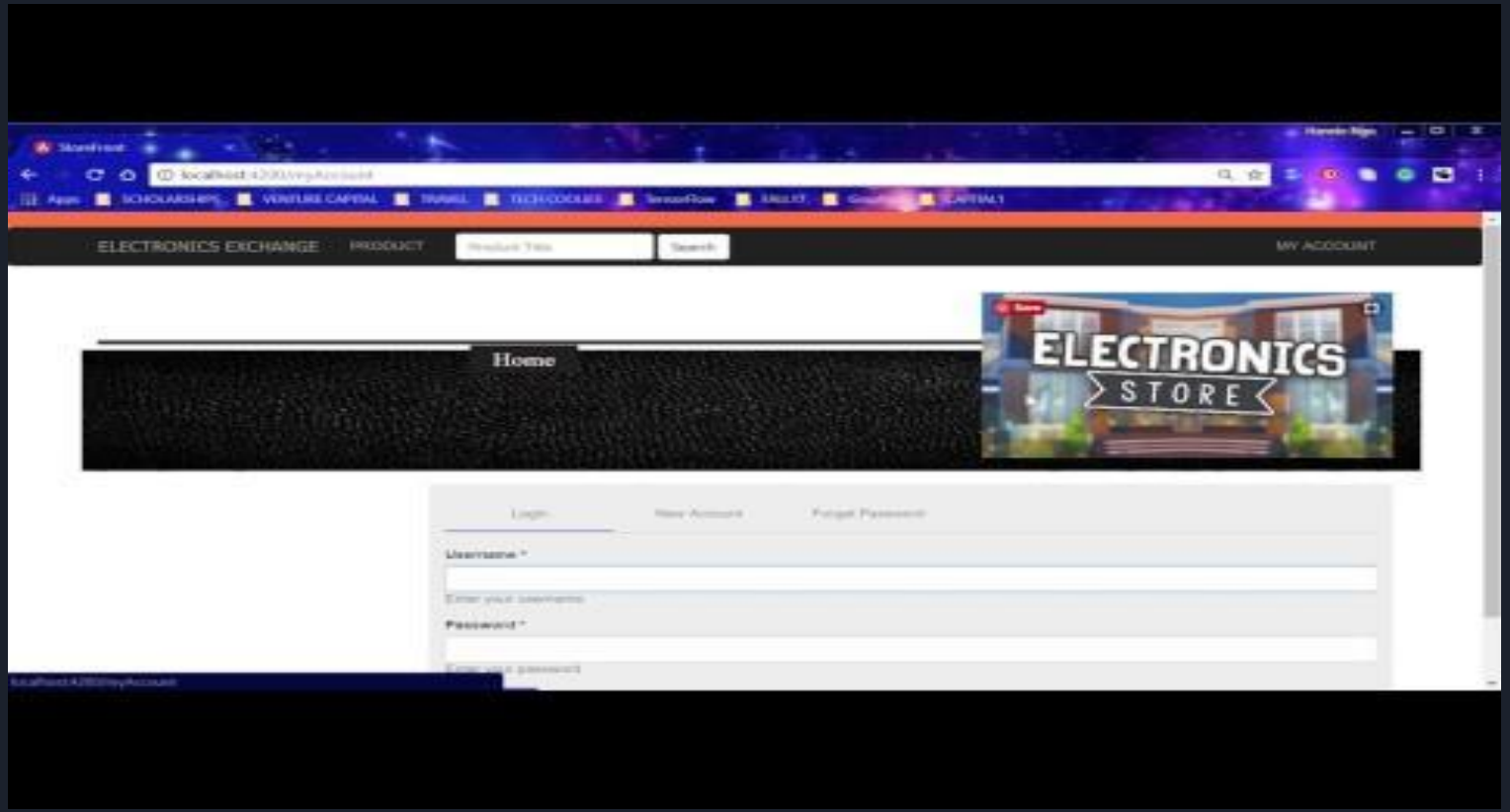
# ElectronXChange

Hannie Ngo, Robert Ladd,
Matthew Sredojevic

# Our Project

- An *e-commerce* web application where local customers can buy and sell electronics products
- Framework: AngularJS, Java Spring Technologies, MySQL
- Features:
  - Admin Portal
  - User sign-up, log-in
  - User profile management
  - Product inventory management
  - Shopping cart processing

# Demo

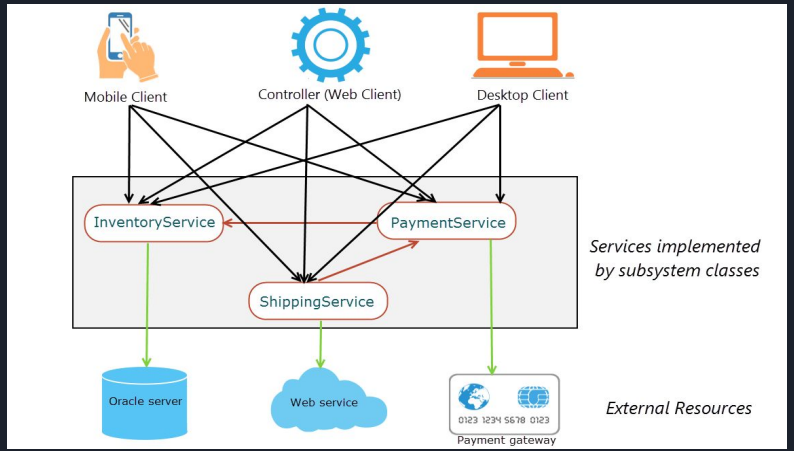

ELECTRONICS EXCHANGE
PRODUCT
Search
MY ACCOUNT
Home
ELECTRONICS
STORE
Login
New Account
Forgot Password
Username *
Password *

# Design Patterns

- Patterns to consider for our e-commerce webapp:
  - Proxy Design Pattern
    - Authorizing user/admin action
  - Simple Factory Design Pattern
    - Creating new classes  easily (i.e different categories of products)
  - Simple Data Access Layer Pattern
    - Creating data mapper to communicate with database
  - Observable Pattern
    - Used within Angular itself | EventListener equivalent
  - Facade Design Pattern
    - Simplifies actions between clients and sub systems

# Facade Pattern

- Part of the classic GoF structural pattern family which helps reduce complexity and simplify interactions that clients need to make with subsystem classes.
- Our problem: When a user places an order for a product, the following services complete the process:
  - Product inventory service
  - Payment service
  - Shipping service



# Facade Pattern

- Our approach: We categorize the participants of the Facade pattern as:
  - ❖ Facade: Delegates client requests to appropriate subsystem classes.
  - ❖ Subsystem classes: Implements subsystem functionalities. Subsystem classes are used by the facade, but not the other way around.
  - ❖ Client: Requests the facade to perform some action.

# Facade Pattern

implementation

Client

```java
public interface OrderServiceFacade {
    boolean placeOrder(int productId);
}
```

```java
public class OrderServiceFacadeImpl implements OrderServiceFacade {

    @Override
    public boolean placeOrder(int productId) {
        boolean orderFulfilled=false;
        Product product=new Product();
        product.setId(productId);

        ProductService productService;
        UserPaymentService userPaymentService;
        UserShippingService userShippingService;

        if(productService).isAvailable(product))
        {
            System.out.println("Product with ID: "+ product.productId+" is available.");
            boolean paymentConfirmed= userPaymentService.makePayment();
            if(paymentConfirmed){
                System.out.println("Payment confirmed...");
                userShippingService.shipProduct(product);
                System.out.println("Product shipped...");
                orderFulfilled=true;
            }
        }
        return orderFulfilled;

    }
```

```java
public interface ProductService {

    List<Product> findAll();
    Product findOne(Long id);
    Product save(Product product);
//  List<Product> blurrySearch(String t
    void removeOne(Long id);

}
```

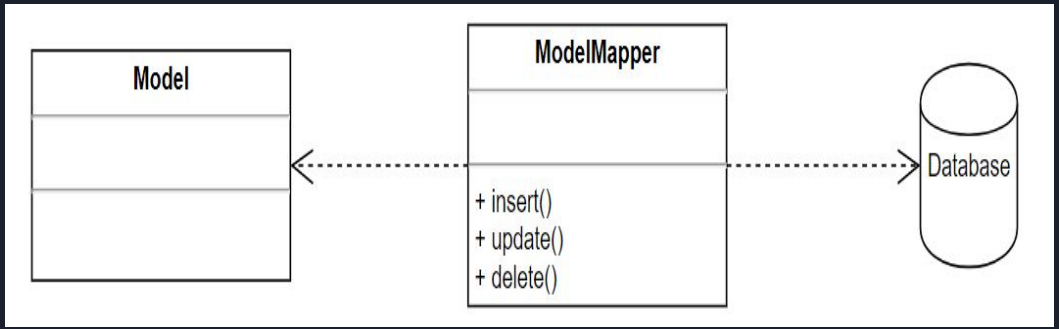
```java

public interface UserShippingService {

    UserShipping findById(Long id);

    void removeById(Long id);

}
```

```java
import com.electronicsstore.domain.UserPayment;

public interface UserPaymentService {

    UserPayment findById(Long id);
    void removeById(Long id);
}
```

# Data Access Layer Pattern Design

**AngularJS Components**

- Performs bi-directional transfer of data between the RDBS and in-memory data representation with RESTful API resources
- The goal of the pattern is to keep the in memory representation and the persistent data store independent of each other and the data mapper itself.



# Data Access Layer Pattern Design

**AngularJS Components**

- Our problem: A customer wants to add a product of interest to her shopping cart. We want to check if that product is in stock and proceed appropriate response.
- Our approach:
  - Created a single service called cart-service, which loads the product's quantity when a cart item is requested
  - This way we create pseudo-data mapper, which adapts our API according to the CRUD Single Page Application functionalities of Angular.

# Data Access Layer Pattern Design

Product class

product.ts

```ts
export class Product {
    public id: number;
    public title: string;
    public manufacturer: string;
    public category: string;
    public shippingWeight: number;
    public listPrice: number;
    public ourPrice: number;
    public active: boolean;
    public description: string;
    public inStockNumber: number;
}
```

cart-item.ts

```ts
import {Product} from './product';
import {ShoppingCart} from './shopping-cart';

export class CartItem {
    public id: number;
    public qty: number;
    public subtotal: number;
    public product: Product;
    public shoppingCart: ShoppingCart
    public toUpdate:boolean;
}
```

Cart-item class

cart-service.ts

```ts
import { Injectable } from '@angular/core';
import {Http, Headers} from '@angular/http';
import {AppConst} from '../constants/app-const';


@Injectable()
export class CartService {

  constructor(private http:Http) { }

  addItem(id:number, qty: number) {
    let url = AppConst.serverPath+"/cart/add";
    let cartItemInfo = {
      "productId" : id,
      "qty" : qty
    }
    let tokenHeader = new Headers({
      'Content-Type' : 'application/json',
      'x-auth-token' : localStorage.getItem("xAuthToken")
    });
    return this.http.post(url, cartItemInfo, {headers: tokenHeader});
  }

  getCartItemList() {
    let url = AppConst.serverPath+"/cart/getCartItemList";

    let tokenHeader = new Headers({
      'Content-Type' : 'application/json',
      'x-auth-token' : localStorage.getItem("xAuthToken")
    });
    return this.http.get(url, {headers: tokenHeader});
  }
```

Access the property of element cartItem in the html code

```html
<p *ngIf="cartItem.product.inStockNumber > 10" style="color:green;">In Stock</p>
```

# THANK YOU!