

AP in Computer Science at Business Academy Southwest

United Markets

3rd Semester Examination

<https://unitedmarketsapp.web.app/>

<https://unitedmarketsapp.azurewebsites.net/api/markets>

Radoslav Backovsky and Anne Luong
December 18, 2020



**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

Preface

This report documents a project assignment completed for the 3rd Semester Examination in the Computer Science study at the Business Academy SouthWest (EASV).

The project took place from November 18, 2020 until December 18, 2020, where the United Markets system was developed.

We were able to complete the project with the guidance and support from the lecturers of the course and would like to thank:

Lars Bilde

Ole Eriksen

Anne-Mette Tønnesen

Henrik Kühl

Radoslav Backovsky and Anne Luong,
AP in Computer Science
Business Academy SouthWest (EASV)



Contents

Preface	1
1. Introduction	4
1.1 Project definition	4
2. SDP and CDS.....	5
2.1 Introduction	5
2.2 System architecture	5
2.3 GUI (including UI-design patterns).....	7
2.4 Conceptual data model.....	10
2.5 Implementation (examples, design patterns and principles)	11
2.5.1 Update.....	11
2.5.2 Security	14
2.5.3 Improvements.....	21
2.6 Conclusion.....	21
3. SDM.....	22
3.1 Introduction	22
3.2 Selected development and management approach.....	22
3.3 Applied XP practices (process documentation and reflections)	22
3.3.1 Planning.....	22
3.3.2 Test driven development	24
3.3.3 Pair programming	28
3.3.4 Progress recording and tracking	28
3.3.5 Collective Code Ownership	29
3.4 Branching model	29
3.5 CI setup and configuration.....	30
3.6 Conclusion.....	31
4. References	32
5. Appendices.....	33
5.1 Appendix 1: Initial project definition	33



**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

5.2	Appendix 2: Overall project schedule	34
5.3	Appendix 3: Software usage	34
5.4	Appendix 3 Scrumwise documentation	35
5.5	Appendix 4 Prototypes.....	39



1. Introduction

In this project, the team practiced and polished skills from the courses Software Architecture and Distributed Programming (SDP), Computer Network and Distributed Systems (CDS) and Software Development Methods (SDM) in a self-defined project. The team wanted to create a web store for grocery delivery as it is a service much needed during the current corona pandemic. There are not many options on the Danish market and especially not one which combines products from the supermarkets' own brands. As such, the team made up a customer, *X Company*, which intends to provide online grocery delivery service with products from the two prominent supermarket retailers *Coop Danmark* and *Salling Group*.

1.1 Project definition

X Company is a new business which will be launched soon. It is an online grocery delivery service, which will help customers get groceries from selected supermarket chains under our partners *Coop Danmark* and *Salling Group*.

We need a website for our potential customers to use our services.

Initial requirements:

1. The website should be aesthetically pleasing and have an intuitive user interface.
2. The customer should first choose a supermarket chain in order to browse the selection of products.
3. The customer can shop by adding products to a digital shopping cart.
4. The customer can see all products in the cart on a "Shopping Cart" page. The customer can place an order with the products in the cart on this page.
5. Only an authenticated super user (admin) can see an overview of all orders.
6. The admin can update orders
7. The admin can delete orders.
8. All customers are guest users in the initial version. The intention is to introduce customer accounts later.

2. SDP and CDS

2.1 Introduction

In this project, the team practiced how to design and implement distributed systems with security considerations in mind. The team developed a distributed web application (SDP) using .NET Core as the back-end and Angular as the front-end. The web-application was deployed using Microsoft Azure (back-end) and Firebase (front-end).

2.2 System architecture

The United Markets application consists of 2 tiers.

- 1) Front end, which is Angular web application.
- 2) Back end, which is ASP.NET Core REST API.

Front end

Job of Angular application is to process a request from user, send it to back end and process a response. The team defined root app module for customer site and child admin module for admin. Angular follows MVVM pattern. [1]

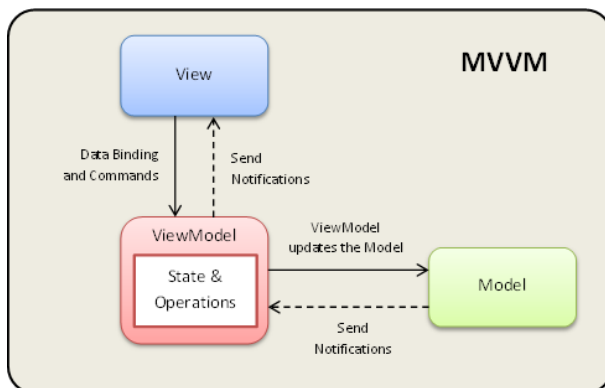


Figure 1. MVVM diagram [2]

Example:

1. User manipulates with component's template.
2. Component processes task and call service.
3. Service calls http client that constructs end sends http request to back end.
4. Response from back-end -> Service -> Component -> component's template. [1]



Back end

Job of REST API is to accept request from front end, prepare data and return them back. REST API follows clean architecture, that means UI and Infrastructure depends on the Core. [1]

UI:	REST API – Access point for a request
CORE:	<ul style="list-style-type: none">- Business entities, unit tests- Application services contain contract and business logic- Domain services contain contract for Infrastructure implementations
INFRASTRUCTURE:	Data initialization for development and data access

Clean Architecture Layers (Onion view)

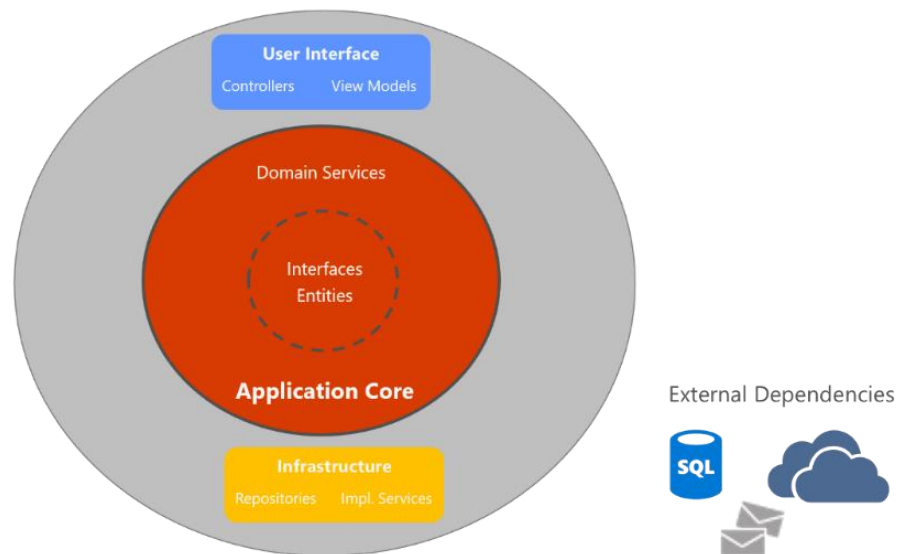
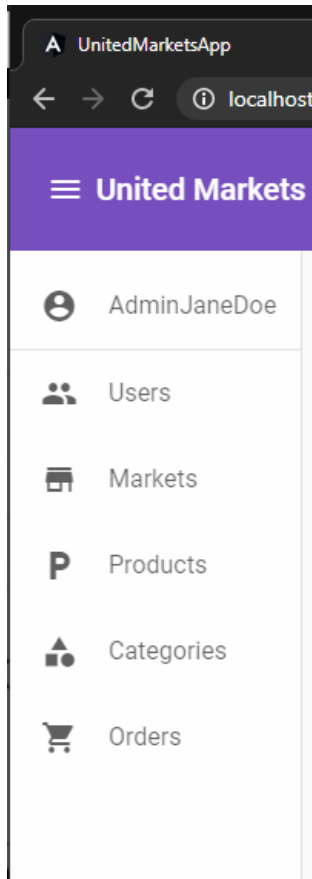


Figure 2. Clean Architecture [3]



2.3 GUI (including UI-design patterns)



The team decided to use Angular Material for designing the application.

Navigation tabs UI design pattern was used for navigating between different markets.

Notifications UI design pattern was used for successful order creation, order update and order deletion.

Module tabs UI design pattern was used for the Admin page. In case of further development, other sections are prepared.

Shopping Cart UI design pattern was used for the Cart page.

Figure 3. Admin module tabs

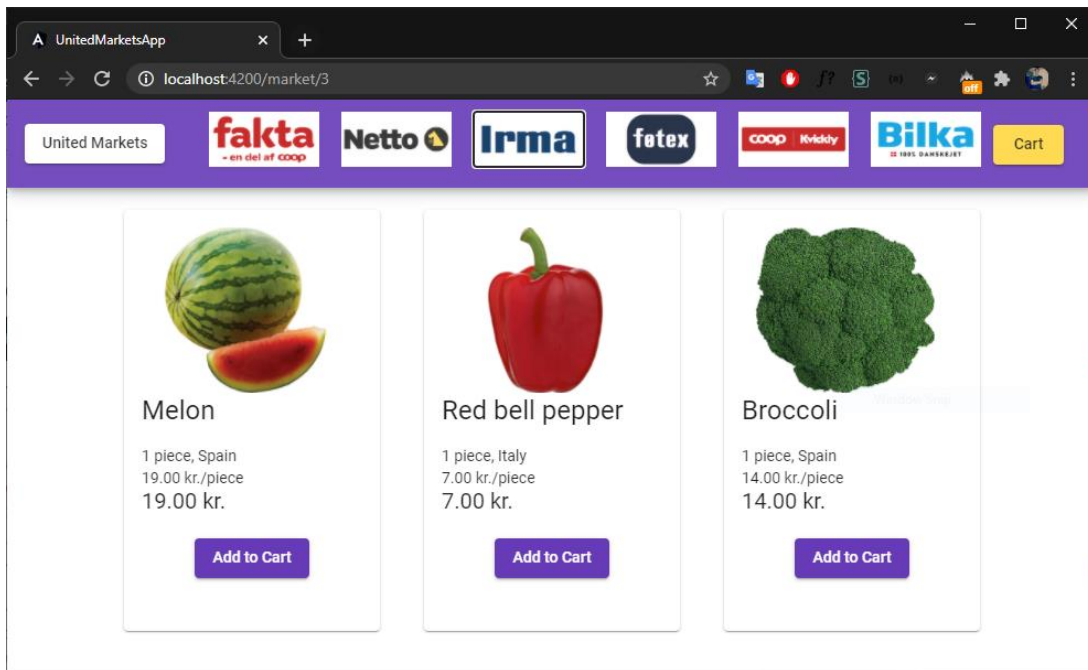


Figure 4. Product view

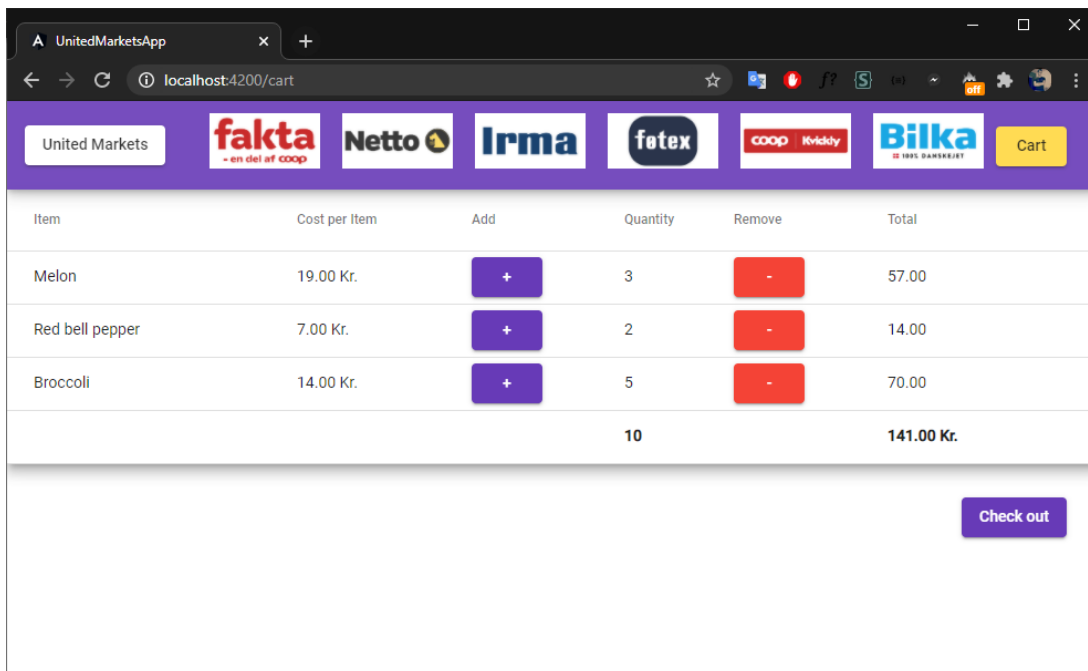


Figure 5. Cart view



**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

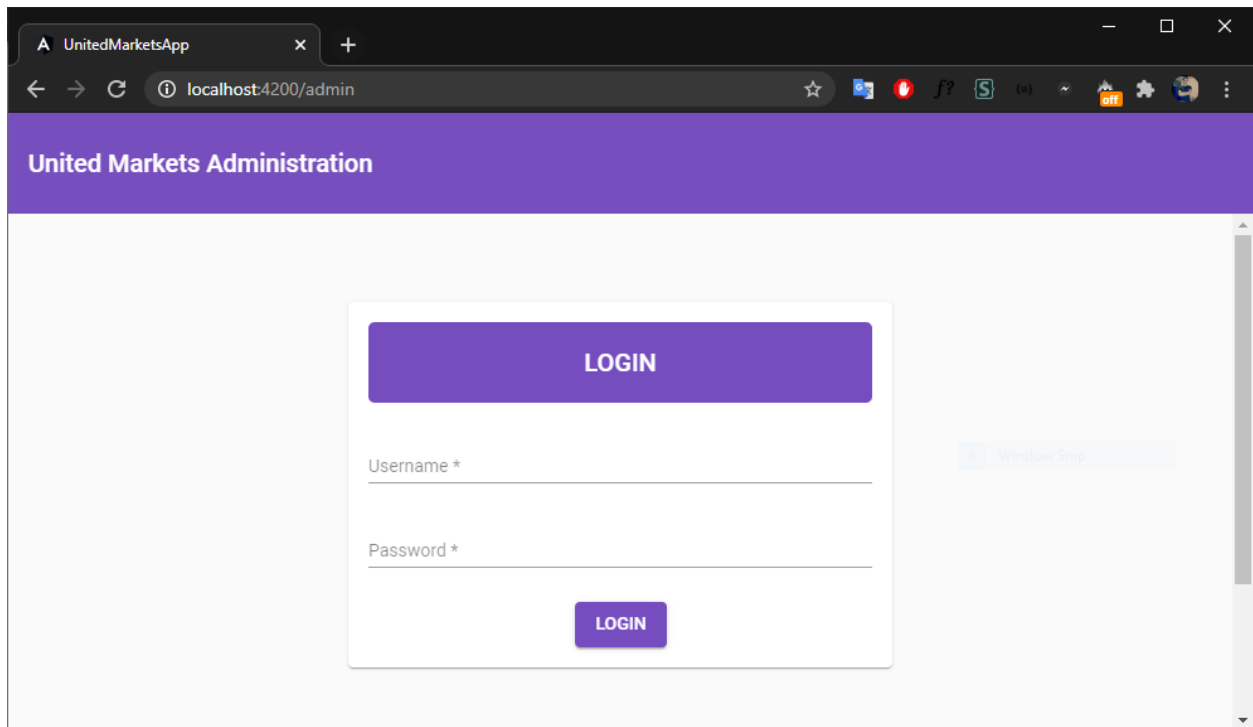


Figure 6. Admin login view

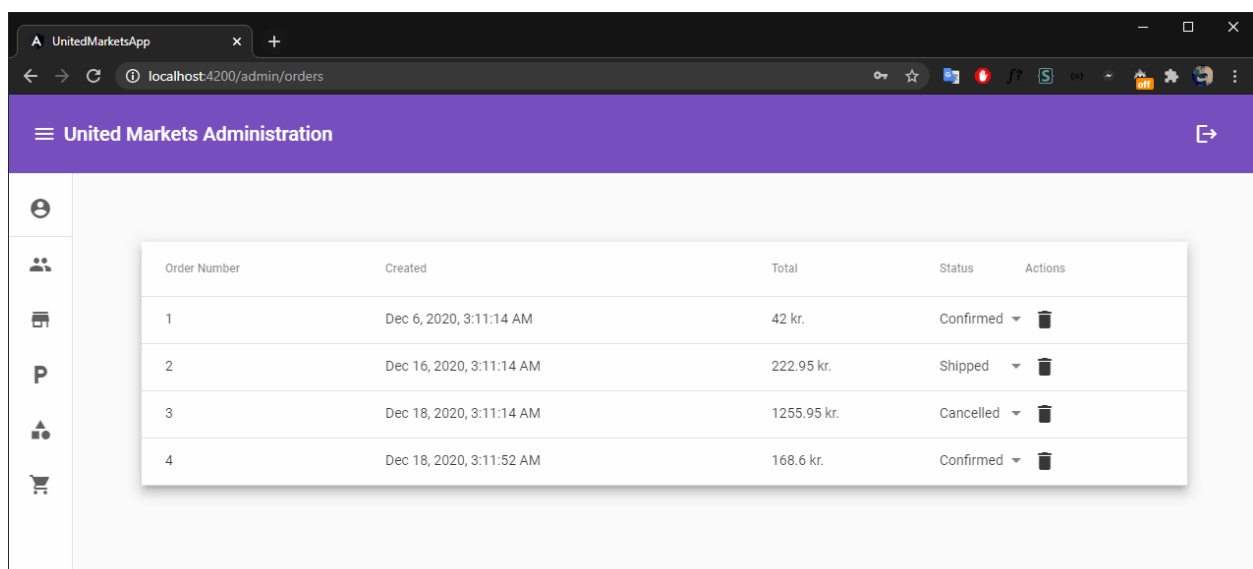


Figure 7. Admin order overview view

2.4 Conceptual data model

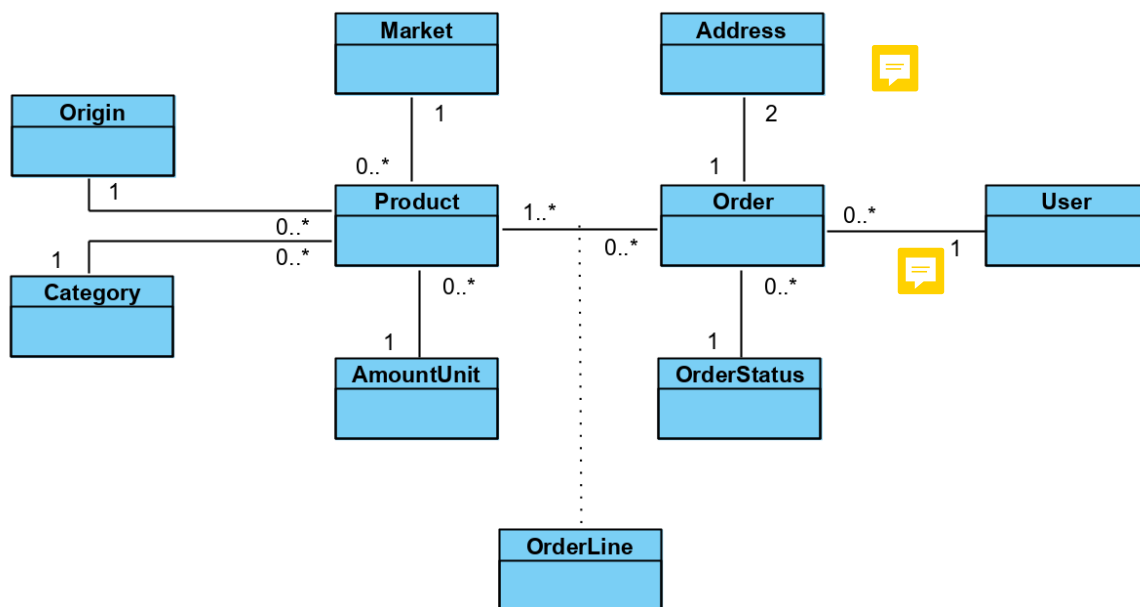


Figure 8. Conceptual data model

A conceptual data model was created to identify the entities and their relations. A one-to-many relation exists between Market and Product, where one Market can be related to zero to many Products and one Product is related to exactly one Market. One-to-many cardinality also exists between the pairings: Origin-Product, Category-Product, AmountUnit-Product, OrderStatus-Order and User-Order. Product and Order has a many-to-many relation, where one Product can belong to zero-to-many Orders and one Order can have one-to-many Products. To implement this relation an associative table (join table) called OrderLine is used. Lastly, an Order has exactly two Addresses (billing and shipping) and an Address has exactly one Order.

The above model was used throughout the project however some relations were not used. In the current system, an Order is not related to any Address and User relation. An Order has the billing and shipping addresses stored as string properties instead. This choice was made to simplify the implementation due to time constraints. During checkout, the Order is placed with hard-coded billing and shipping addresses by a guest user. Effectively, there is no relation between Order and User.

The User relation is currently used for admin users only, so an admin can log in and perform administrative services, which include viewing, editing and deleting orders. The current implementation is open for introducing Order-Address and Order-User relations in the future.

2.5 Implementation (examples, design patterns and principles)

The team used several design patterns throughout the project, which includes Unit of Work Pattern (back-end), Dependency Injection (back-end and front-end), and Observable Pattern (front-end). Interfaces were also used in the back-end for the services, validators and repositories to lower coupling. When possible, generic interfaces were used.

2.5.1 Update

The team decided to implement full update functionality for order in back end in one request. Update(order) method from front end is using back end method for updating. At this point only order status id is updated in Angular application. The team did not implement full update for order in front end because of time constrains.

Front-end

```

21     <!-- Status column -->
22     <ng-container matColumnDef="status">
23         <th mat-header-cell *matHeaderCellDef>Status</th>
24         <td mat-cell *matCellDef="let order" (click)="$event.stopPropagation()">
25             <mat-select [(value)]="order.statusId" (valueChange)="updateOrder(order)">
26                 <mat-option *ngFor="let s of statuses" [value]="s.id">{{ s.name }}</mat-option>
27             </mat-select>
28         </td>
29     </ng-container>

```

Figure 9. order-list.component.html

\$event.stopPropagation() is ignoring redirection to order-item component after user clicks on order item. Each order has a default value of its statusId which is formatted as status name of status fetched from status array in component.ts. After value of option is changed, order object gets new value of statusId and updateOrder(order) is triggered.



```
97     updateOrder(order: Order): void {
98         this.orderService.updateOrder(order).subscribe(
99             response => {
100                 console.log(response);
101                 this.snackBar.open('Status of order #' + response.id + ' was successfully updated.', '', {
102                     duration: 6000,
103                     horizontalPosition: 'center',
104                     verticalPosition: 'top',
105                     politeness: 'polite',
106                     panelClass: ['mat-toolbar', 'mat-accent']
107                 });
108             }
109         );
110     }
111 }
```

Figure 10. order-list.component.ts

```
42     updateOrder(order: Order): Observable<Order> {
43         httpOptions.headers =
44             httpOptions.headers.set('Authorization', 'Bearer ' + this.authService.getToken());
45         return this.http.put<Order>(this.ordersApiUrl + '/' + order.id, order, httpOptions);
46     }
47 }
```

Figure 11. order.service.ts

updateOrder in component.ts will call service method. Service method calls http Clients put method that will construct PUT request to REST API which has to be authenticated for administrator. Returned value is of type Observable<Order>, that is why subscribing to service method is necessary. After response is received, component.ts is notified and then it can display notification of successful update with returned order information.



Back-end

```
65         // PUT
66         [Authorize(Roles = "Administrator")]
67         [HttpPut("{id}")]
68         public IActionResult Put(int id, [FromBody] Order order)
69         {
70             if (id != order.Id)
71             {
72                 return BadRequest("Order id must match.");
73             }
74
75             try
76             {
77                 return Ok(_orderService.Update(order));
78             }
79             catch (System.Exception e)
80             {
81                 return StatusCode(500, e.Message);
82             }
83         }
```

Figure 12. Put method in OrdersController.cs

Access point for request from client. Order object passed as JSON and converted to Order BE.

```
54         public Order Update(Order order)
55         {
56             _orderValidator.IdValidation(order.Id);
57             order.DateUpdated = DateTime.Now;
58             foreach (var orderLine in order.Products)
59             {
60                 _orderLineValidator.DefaultValidation(orderLine);
61                 _orderLineValidator.UpdateValidation(orderLine);
62             }
63
64             _orderValidator.DefaultValidation(order);
65             _orderValidator.UpdateValidation(order);
66             return _orderRepository.Update(order);
67         }
```

Figure 13. Update order in OrderService.cs

Validation is heavy responsibility of the core. In case of invalid data input, exception with clear warning is returned to the client. The team decided to assign dateUpdated value in the core. This way the team can unit test this system requirement.



```
63     public Order Update(Order orderFromRequest)
64     {
65         _ctx.OrderLines.RemoveRange(_ctx.OrderLines.Where(ol => ol.OrderId == orderFromRequest.Id));
66         _ctx.OrderLines.AddRange(orderFromRequest.Products);
67         _ctx.Update(orderFromRequest);
68         _ctx.Entry(orderFromRequest).Property(x => x.DateCreated).IsModified = false;
69         _ctx.SaveChanges();
70
71         var orderForClient = _ctx.Orders
72             .Include(o => o.Products)
73             .FirstOrDefault(o => o.Id == orderFromRequest.Id);
74
75         if (orderForClient == null)
76             throw new NullReferenceException("Order does not exist in database.");
77
78         return orderForClient;
79     }
```

Figure 14. Update order in OrderSqlLiteRepository.cs

Line 65, 66: Removing existing OrderLines in order and adding new list from request.

Line 67: Updating order details.

Line 68: Ensuring the DateCreated is ignored for update order.

Line 69: Executing SQL statement that will update Orders Table and OrderLines Table.

2.5.2 Security

Authentication

The administrative features are accessed through the URL paths with /admin. From a realistic point of view, the management services would not share the same domain name. The project itself became disorganized due to bad planning, where the team did not make sure all basic CRUD operations could be covered (a requirement in the exam project description document). Implementation of *update* and *delete* from possible customer-oriented features of the web store were not considered and are not many. The only update and delete operations would be updating user information or deleting a user account. This was discouraged by a teacher, who suggested to implement *update* and *delete* orders, which are admin features. This way the implemented features would include CRUD operations for orders.

A user will be directed to a login page when trying to access any administration features. Only authorized users (admin users) can access, because an authorization guard (AuthGuard) has been added to all admin related components in the AdminRoutingModule.

```

8  const appRoutes: Routes = [
9    { path: 'admin',
10     component: AdminComponent,
11     children: [
12       { path: 'orders', component: OrderListComponent, canActivate: [AuthGuard], },
13       { path: 'order/:id', component: OrderItemComponent, canActivate: [AuthGuard], }
14     ]
15   }
16 ];

```

Figure 15. admin-routing.module.ts

If a user types any path with /admin (e.g. /admin/orders) and is not logged in, the AuthGuard will redirect the user to the admin component.

```

5  @Injectable()
6  export class AuthGuard implements CanActivate {
7
8    constructor(private router: Router, private authService: AuthenticationService) { }
9
10   canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
11     if (this.authService.getToken()) {
12       // Logged in so return true
13       return true;
14     }
15
16     // Not logged in so redirect to login page with the return url
17     this.router.navigate([ '/admin/' ], { extras: { queryParams: { returnUrl: state.url } } });
18     return false;
19   }
20 }

```

Figure 16. auth.guard.ts

In the admin component, if the user is not authenticated, the LoginAdminComponent is shown.

```

45  <div class="selection-content">
46    <app-login-admin *ngIf="!isAuthenticated()"></app-login-admin>
47    <router-outlet *ngIf="isAuthenticated()"></router-outlet>
48  </div>

```

Figure 17. admin.component.html

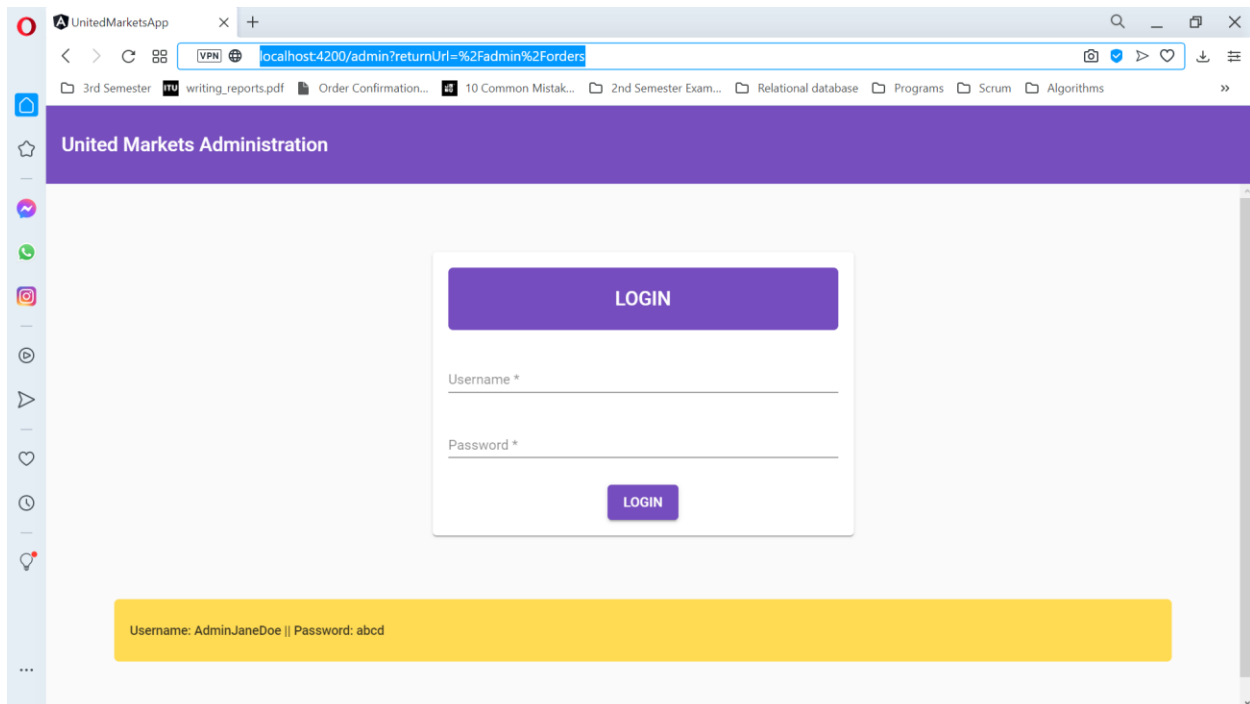


Figure 18. Admin login view from `admin.component.html` and `login-admin.component.html`

When the user has filled out the login information and clicked the login button, a method in the AuthenticationService below will be called. The method will send a POST request to the back-end with the username and password.

```
46 onSubmit(): void {
47   if (this.loginForm.invalid) {
48     return;
49   }
50
51   this.loading = true;
52   this.authenticationService.login(this.username.value, this.password.value)
53     .subscribe(
54       next: success => {
55         this.router.navigateByUrl(this.returnUrl);
56       },
57       error: error => {
58         this.submitted = true;
59         // TODO: Get the error message from back-end.
60         this.errorMessage = error.message;
61         this.loading = false;
62       }
63     );
64 }
```

Figure 19. `login-admin.component.ts`



```
7  @Injectable()
8  export class AuthenticationService {
9
10     constructor(private http: HttpClient) {}
11
12     Login(username: string, password: string): Observable<boolean> {
13         return this.http.post<any>({ url: environment.apiUrl + 'login', body: { username, password } })
14             .pipe(map( project: response => {
15                 const token = response.token;
16                 // Login successful if there's a jwt token in the response
17                 if (token) {
18                     // Store username and jwt token in local storage to keep user logged in between page refreshes
19                     localStorage.setItem('currentUser', JSON.stringify({ value: { username, token } }));
20                     // Return true to indicate successful login
21                     return true;
22                 } else {
23                     // Return false to indicate failed login
24                     return false;
25                 }
26             }));
27     }
```

Figure 20. authentication.service.ts

If the user and password match the stored record in the database, the username and token will be sent back as a reply. The username and token are stored in local storage (for authorization) and the user is redirected to the return or default URL.

If the login is unsuccessful, an error message will be shown, and text will prompt the user to try again with valid user login information.

In the back-end, the POST request will call an authentication method in the UserService.

```
20  [HttpPost]
21  // anneluong
22  public IActionResult Login([FromBody] LoginInputModel loginInputModel)
23  {
24      try
25      {
26          var token :string = _userService.AuthenticateUser(loginInputModel);
27
28          return Ok(new
29              {
30                  loginInputModel.Username,
31                  token
32              });
33      }
34      catch (Exception e)
35      {
36          return Unauthorized(e.Message);
37          //return BadRequest(e.Message);
38      }
39  }
```

Figure 21. LoginController.cs

The method FindUser(string username) will then try to find the user in the database through the UserRepository.



```
30 public string AuthenticateUser(LoginInputModel loginInputModel)
31 {
32     _loginInputModelValidator.DefaultValidation(loginInputModel);
33     var user = FindUser(loginInputModel.Username);
34
35     if (!_authenticationHelper.VerifyPasswordHash(loginInputModel.Password, user.PasswordHash,
36         user.PasswordSalt)) throw new ArgumentException( message: "Invalid password.");
37
38     return _authenticationHelper.GenerateToken(user);
39 }
40
41 private User FindUser(string username)
42 {
43     var user = _userRepository.ReadByName(username);
44
45     if (user == null) throw new ArgumentException( message: "Non-existing username.");
46
47     return user;
48 }
```

Figure 22. UserService.cs

The query method used to find the user in the UserRepository is shown below.

```
39 public User ReadByName(string username)
40 {
41     return _ctx.Users.ToList().FirstOrDefault( predicate: user => user.Username == username);
42 }
```

Figure 23. UserSqlLiteRepository.cs

Next, the password needs to be verified. All passwords are stored in the database after computing password hash and salt values using the HMACSHA512 algorithm. The password salt results in different password hash values for identical passwords, which is effective against rainbow table attack. The verification below will check if the computed hash value is equal to the stored hash value. The method will return true if they are identical.

```
28 public bool VerifyPasswordHash(string password, byte[] storedHash, byte[] storedSalt)
29 {
30     using var hmac = new HMACSHA512(storedSalt);
31     var computedHash :byte[] = hmac.ComputeHash( buffer: Encoding.UTF8.GetBytes(password));
32
33     for (var i = 0; i < computedHash.Length; i++)
34         if (computedHash[i] != storedHash[i])
35             return false;
36
37     return true;
38 }
```

Figure 24. AuthenticationHelper.cs



If the verification is successful, a JSON Web Token (JWT) will be generated using the method below. The token will be returned to the LoginController, which will send the username and token as the response of the HTTP POST request.

```
40 public string GenerateToken(User user)
41 {
42     var claims :List<Claim> = SetUpClaims(user);
43
44     //Create a token with a header (token type JWT,private key and algorithm used for signing token) and payload
45     var token = new JwtSecurityToken(
46         new JwtHeader(new SigningCredentials(
47             new SymmetricSecurityKey(_secretBytes),
48             SecurityAlgorithms.HmacSha256)),
49         new JwtPayload( issuer: null, // issuer - not needed (ValidateIssuer = false)
50             audience: null, // audience - not needed (ValidateAudience = false)
51             claims.ToArray(),
52             notBefore: DateTime.Now, // notBefore
53             expires: DateTime.Now.AddMinutes(10))); // expiration time
54
55     return new JwtSecurityTokenHandler().WriteToken(token);
56 }
```

Figure 25. AuthenticationHelper.cs

```
58 /// Create a list of claims containing username and permission. ...
63 private List<Claim> SetUpClaims(User user)
64 {
65     var claims = new List<Claim>
66     {
67         new Claim( type: ClaimTypes.Name, value: user.Username)
68     };
69
70     if (user.IsAdmin) claims.Add( item: new Claim( type: ClaimTypes.Role, value: "Administrator"));
71     else claims.Add( item: new Claim( type: ClaimTypes.Role, value: "User"));
72
73     return claims;
74 }
```

Figure 26. AuthenticationHelper.cs

Authorization

As mentioned earlier, the token is stored in local storage. The token is used to assert the user has been verified and used for authorization to actions in the OrdersController. The actions are marked with the [Authorize] attribute.



United Markets

Radoslav Backovsky and Anne Luong

```
29 getToken(): string {
30     const currentUser = JSON.parse(localStorage.getItem( key: 'currentUser'));
31     if (currentUser) {
32         return currentUser.token;
33     } else {
34         return null;
35     }
36 }
```

Figure 27. authentication.service.ts

```
27 getOrders(): Observable<Order[]> {
28     // Add authorization header with jwt token
29     httpOptions.headers =
30     | httpOptions.headers.set('Authorization', 'Bearer ' + this.authenticationService.getToken());
31
32     // Get from api
33     return this.http.get<Order[]>({this.ordersApiUrl, httpOptions});
34 }
```

Figure 28. order.service.ts

```
21 // GET
22 [Authorize(Roles = "Administrator")]
23 [HttpGet]
24 | anneluong
25 public ActionResult<IEnumerable<Order>> GetAll()
26 {
27     try
28     {
29         return Ok(_orderService.GetAll());
30     }
31     catch (Exception e)
32     {
33         return BadRequest(e.Message);
34     }
35 }
```

Figure 29. OrdersController.cs



2.5.3 Improvements

Setting order statusId in the core method. Preventing client to set different statusId on order creation other than “Pending”. In this case the team must make sure that “Pending” status has id equal to 1 in database. This approach can be improved by injecting StatusSqlLiteRepository to OrderService and getting all order statuses. Id of specific status might change, therefor hardcoded statusId in the OrderService is bad practice.

```
31         public Order Create(Order order)
32         {
33             order.DateCreated = DateTime.Now;
34             order.DateUpdated = DateTime.Now;
35             order.StatusId = 1;
36             foreach (var orderLine in order.Products) _orderLineValidator.DefaultValidation(orderLine);
37             _orderValidator.DefaultValidation(order);
38             _orderValidator.CreateValidation(order);
39             return _orderRepository.Create(order);
40         }
41     }
```

Figure 30. Order statusId set in the Core

2.6 Conclusion

The team gained valuable experience in developing a distributed web application and the deployment of it.



3. SDM

3.1 Introduction

The team practiced several SDM disciplines such as XP development methodology, TDD, branching models and CI. The usage and reflections are documented in the following sections.

3.2 Selected development and management approach

In this project, the team used Extreme Programming (XP) as the development approach and Scrum as the management approach. It was not possible to purely use XP as the short duration of the project was unsuitable for XP's release cycles. The team would not be able to finish a release cycle by the end of the project. Hence, Sprints from the Scrum methodology was used instead.

3.3 Applied XP practices (process documentation and reflections)

3.3.1 Planning

Exploration

First, an overall project schedule was made to get an overview (see Appendix 2: Overall project schedule). The team also decided which software would be used in the project (see Appendix 3: Software usages).

User stories were written by the team as the project didn't have a customer. The table shows the stories formulated from the initial project definition (Appendix 1: Initial project definition). The team also divided the stories into engineering tasks and added tasks which were necessary during the developing process.

User stories from the initial project definition
As a customer, I should see the minimum order requirements and delivery fees clearly.
As a customer, I should be able to select the supermarket I want to shop in.
As a customer, I should be able to view products of selected shop.
As a customer, I should be able to see different categories of products in the selected shop.
As a customer, I should be able to choose a product and see the product information on a new page. (price, weight/amount, price per kg, origin)
As a customer, I should be able to increase the amount of the product before adding it to the shopping cart.
As a customer, I should be able to add products to a digital shopping cart.
As a customer, I should be able to see all my products in the shopping cart on a "Shopping Cart" page before order confirmation.



As a customer, I should be able to place an order as a guest user from the “Shopping Cart” page by clicking a “Checkout” button which will redirect me to a page for filling out order placement information.
As a customer, I should get validation for my inputs when I fill out the order placement information (name, email, phone number and address).
As a customer, I should be able to choose a delivery time for my order.
As a customer, I should finalize my order by clicking a “Place order” button.
As an admin, I should be able to view all orders.
As a customer, I should be able to choose if I want to continue checkout as a guest user or login as an existing user.
As a customer, I should be able to place my order with a personal user account.

Table 2. User stories from the initial project definition

Planning

In XP methodology, a commitment schedule with user stories in the current release would be made. However, the team deemed it unsuitable and decided to use Sprints from the Scrum methodology instead. The following Scrum practices were used:

- Initial Product Backlog (prioritizing user stories)
- Sprint Backlog (choose user stories after prioritization)
- Sprint planning
- Sprint Review (casually done)
- Sprint Retrospective (casually done)

In both Sprints, each developer of the team volunteered for the tasks and estimated the ideal engineering time.

Sprint Backlogs can be viewed in the project RA_Web_Store_CS2019 on Scrumwise. The appendix also contains screen captures (Appendix 4: Scrumwise documentation).

Steering

The project plan was influenced by the team as the project developed. The project definition changed, and new user stories were created. The changes were necessary, because the initial intended features did not fulfill the requirements of the exam project description. The team was not aware that the implementation of basic *CRUD* operations in both front-end was compulsory, so the initial plan only had *create* and *read* related operations in both front-end and back-end. The focus was on letting a customer browse a web store, filtering products by supermarket and categories, adding products to a shopping cart, editing the cart contents, and place an order.

The team also got behind schedule throughout the project as the team struggled with front-end implementation. User stories had to be revised (initial expectations were too high) and some were

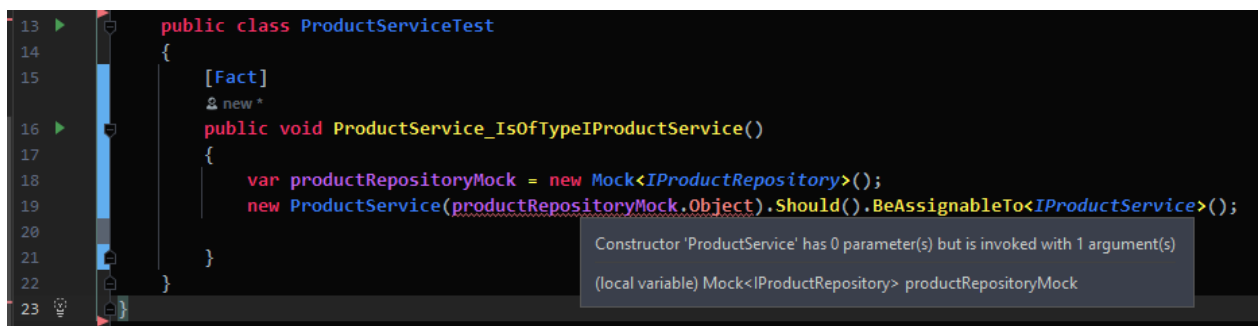
dropped to make sure all CRUD operations would be implemented. Initially, the team wanted to have three Sprints (each around 1 week) but changed to only two Sprints due to the delay in progress.

3.3.2 Test driven development

Motivation behind test driven development is clean code that works. With TDD a developer must think about every requirement and handle what should happen if something goes wrong. The important thing is that developer must write test first and then write functional code for passing test and then refactor. With this approach the development team is saving a lot of time in a long run. The team wrote 86 tests that are passing, therefor ensuring well exception handling for end user.

	Missing unit tests:
Product Validation	Name CategoryId AmountUnitId
Order Validation	Correct total price calculated from order lines.
OrderLine Validation	Correct sub total price calculated from product prices.

TDD Example 1:



```

13 public class ProductServiceTest
14 {
15     [Fact]
16     public void ProductService_IsOfTypeIProductService()
17     {
18         var productRepositoryMock = new Mock<IProductRepository>();
19         new ProductService(productRepositoryMock.Object).Should().BeAssignableTo<IProductService>();
20     }
21 }
22
23

```

Constructor 'ProductService' has 0 parameter(s) but is invoked with 1 argument(s)
(local variable) Mock<IProductRepository> productRepositoryMock

Figure 31. ProductService test

ProductService did not have dependency on IProductRepository therefore the test cannot be executed.

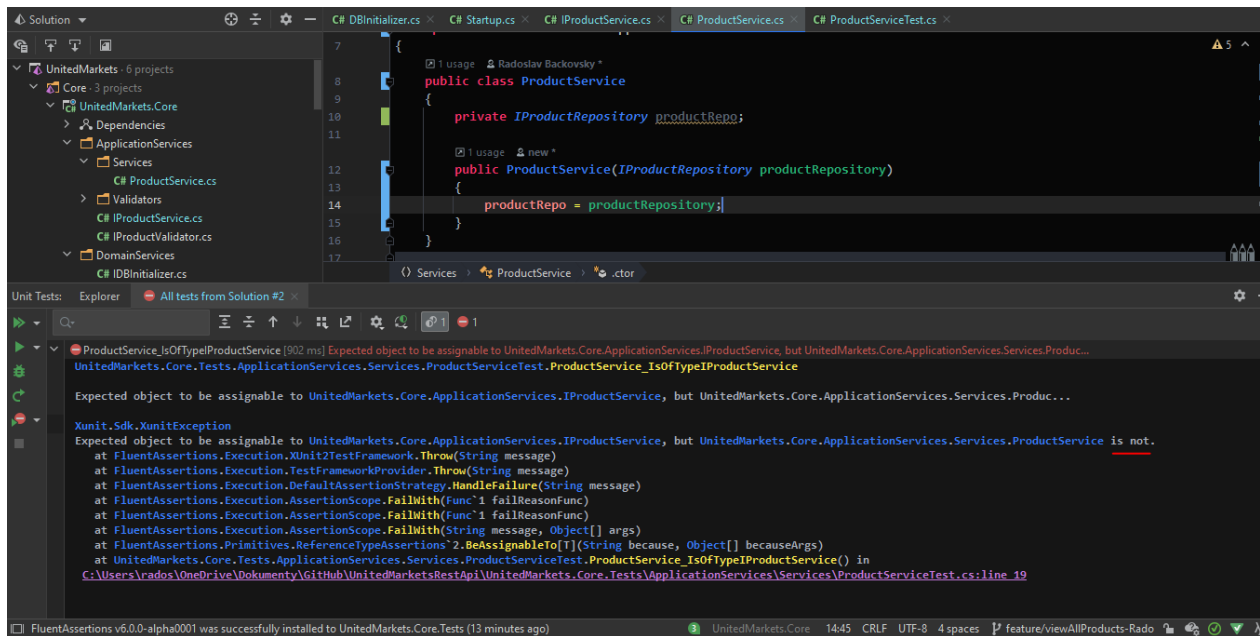


Figure 32. ProductService dependency injection

The team had to dependency inject ProductRepository to ProductService and run the tests.

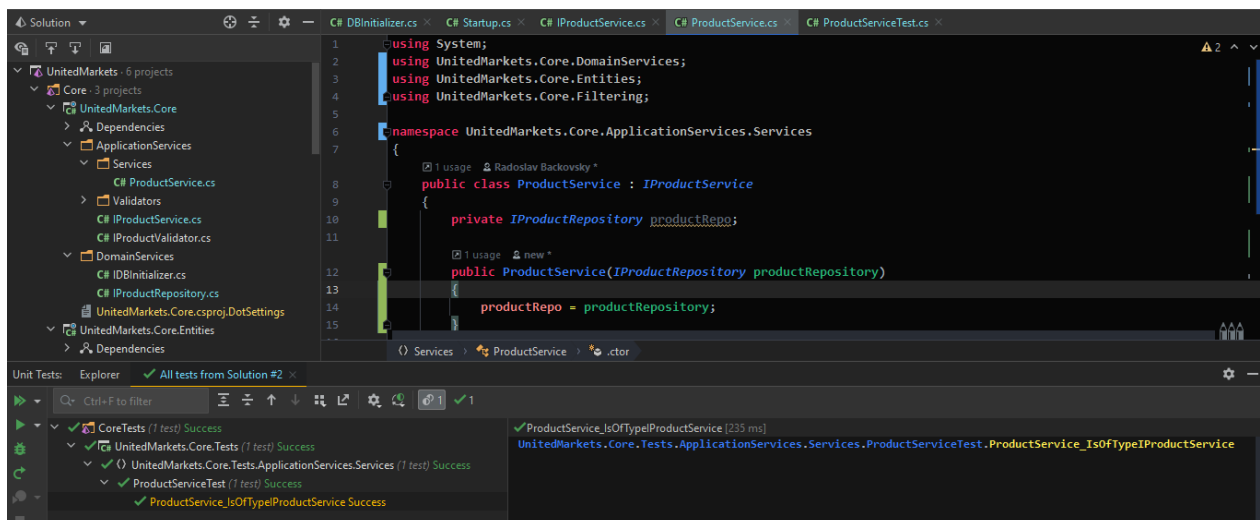


Figure 33. Test passing

ProductService is implementing IProductService interface, test is successful.



TDD Example 2:

```
22 [Fact]
23 new *
24 public void NewProductService_WithNullRepository_ShouldThrowException()
25 {
26     var productRepositoryMock = new Mock<IProductRepository>();
27     Action action = () => new ProductService(null as IProductRepository);
28     action.Should().Throw<NullReferenceException>().WithMessage("Product Repository Cannot be Null.");
29 }
30 }
```

Figure 34. Testing New ProductService with Null Repository

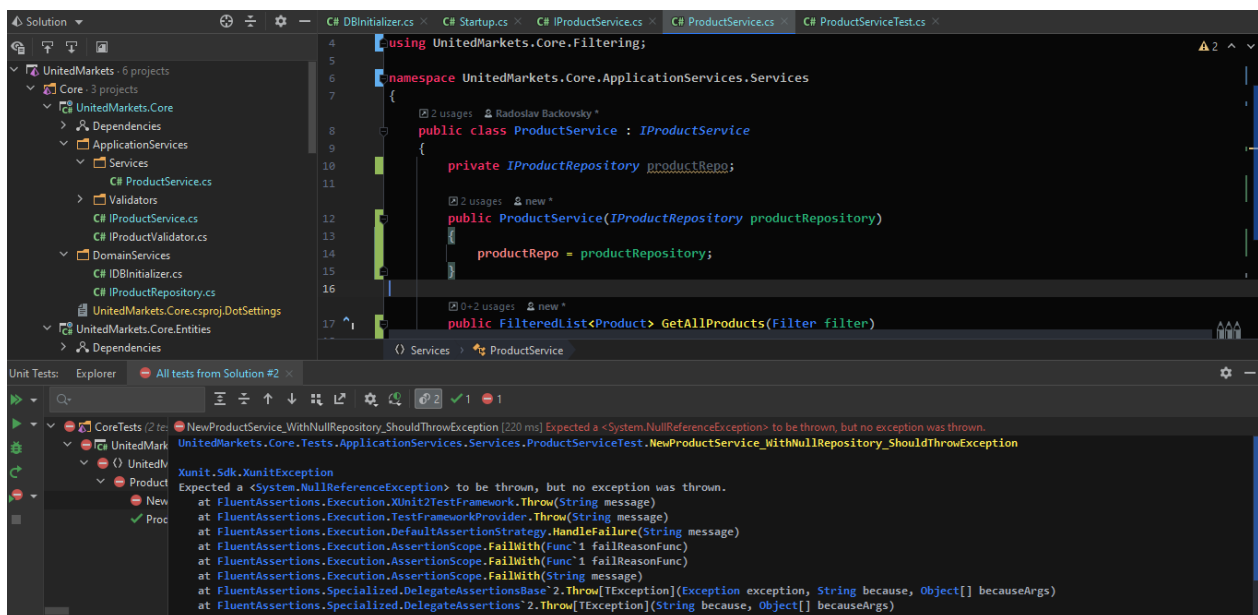


Figure 35. Failing test

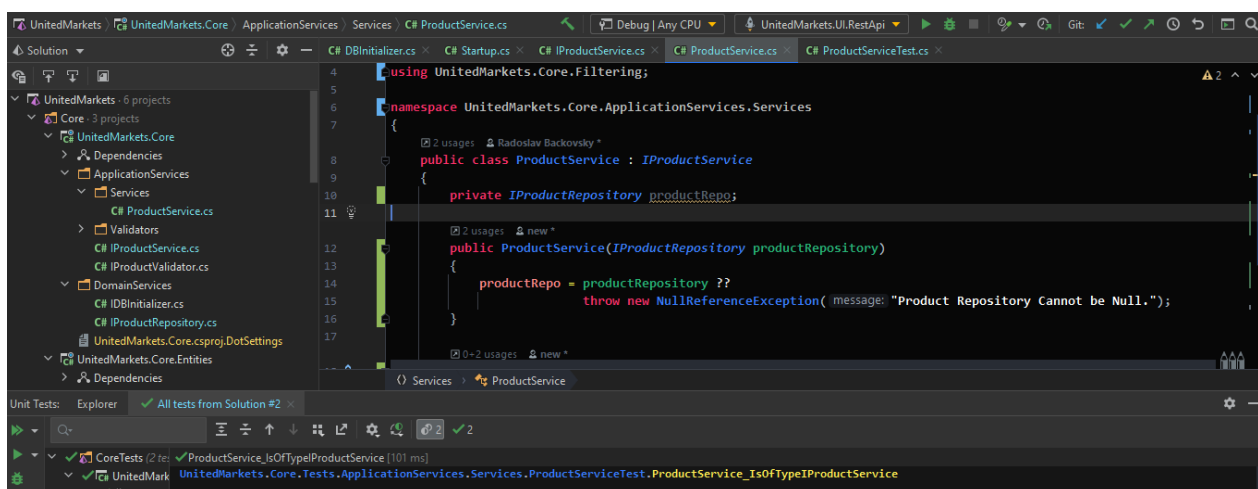




Figure 36. Correct implementation, passing test

Example 3:

```
C# DBInitializer.cs x C# Startup.cs x C# IProductService.cs x C# ProductService.cs x C# ProductServiceTest.cs x
29
30
31 [Fact]
32 new *
33 public void GetAllProducts__ShouldCallRepoWithFilterInParams_Once()
34 {
35     var productRepositoryMock = new Mock<IProductRepository>();
36     IProductService productService = new ProductService(productRepositoryMock.Object);
37     Filter filter = new Filter();
38     productService.GetAllProducts(filter);
39     productRepositoryMock.Verify( expression: repo => repo.GetAllProducts(filter), Times.Once);
40 }
41 }
```

Services > ProductServiceTest > GetAllProducts__ShouldCallRepoWithFilterInParams_Once

Failed: GetAllProducts__ShouldCallRepoWithFilterInParams_Once [11 ms] System.NotImplementedException: The method or operation is not implemented.

Application failed: O

allRepo

NullRep

Product

System.NotImplementedException: The method or operation is not implemented.

The method or operation is not implemented.

at UnitedMarkets.Core.ApplicationServices.Services.ProductService.GetAllProducts(Filter filter) in C:\Users\rados\OneDrive\Documents\GitHub\UnitedMarketsRestApi\UnitedMarkets.Core\ApplicationServices\Services\ProductService.cs:line 20

at UnitedMarkets.Core.Tests.ApplicationServices.Services.ProductServiceTest.GetAllProducts__ShouldCallRepoWithFilterInParams_Once() in C:\Users\rados\OneDrive\Documents\GitHub\UnitedMarketsRestApi\UnitedMarkets.Core.Tests\ApplicationServices\Services\ProductServiceTest.cs:line 37

Figure 37. Failing test

```
C# DBInitializer.cs x C# Startup.cs x C# IProductService.cs x C# ProductService.cs x C# ProductServiceTest.cs x
14
15 productRepo = productRepository ??
16     throw new NullReferenceException( message: "Product Repository Cannot be Null.");
17 }
18
19 0+3 usages new *
20 public FilteredList<Product> GetAllProducts(Filter filter)
21 {
22     return productRepo.GetAllProducts(filter);
23 }
24 }
```

Services > ProductService

ProductService_IsOfTypeIProductService [111 ms]

UnitedMarkets.Core.Tests.ApplicationServices.Services.ProductServiceTest.ProductService_IsOfTypeIProductService

Application

Figure 38. Correct implementation resulting in test passing

The same approach was used throughout the development.

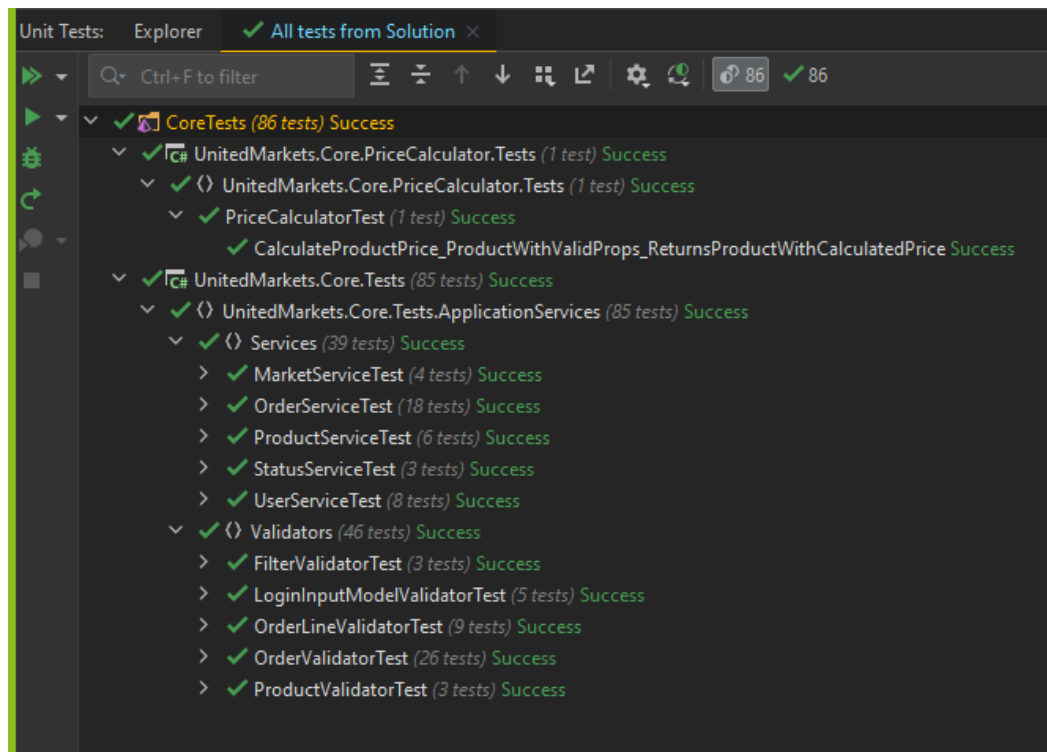


Figure 39. Results from all tests

3.3.3 Pair programming

An attempt at pair programming was made, when Radoslav needed help from Anne regarding unit testing and fluent assertions for `Create(Order order) + dateCreated`. The process consisted of asking questions and teamwork. Both members were able to use their computers, therefore the experience is not considered as proper pair-programming. During development, there was a lot of discussion and both members had their responsibilities split. The team considers pair-programming as difficult practice to implement in daily development, because of time constraints and ease of use. Team members also prefer to work alone and ask questions as needed.

3.3.4 Progress recording and tracking

The team did not have a dedicated tracker as described in XP. Instead, the Scum Burndown Chart was used to keep track of the progress together with the Scrum Taskboard. Both can be seen in the project `RA_Web_Store_CS2019` on Scrumwise. Burndown Charts are also included in Appendix 3: Scrumwise documentation.

3.3.5 Collective Code Ownership

The team strives for high code quality and was open to suggestions and improvements. Especially, since the project development had a high learning curve. The team is aware that refactoring can improve code quality, but also further development and knowledge of development team.

3.4 Branching model

The team used Git version control and decided on a branching model with one main branch, one development branch and several feature branches as needed.

- Main branch contains the Sprints
- Develop branch contains user stories from all iterations
- Each feature branch contains one user story (feature)

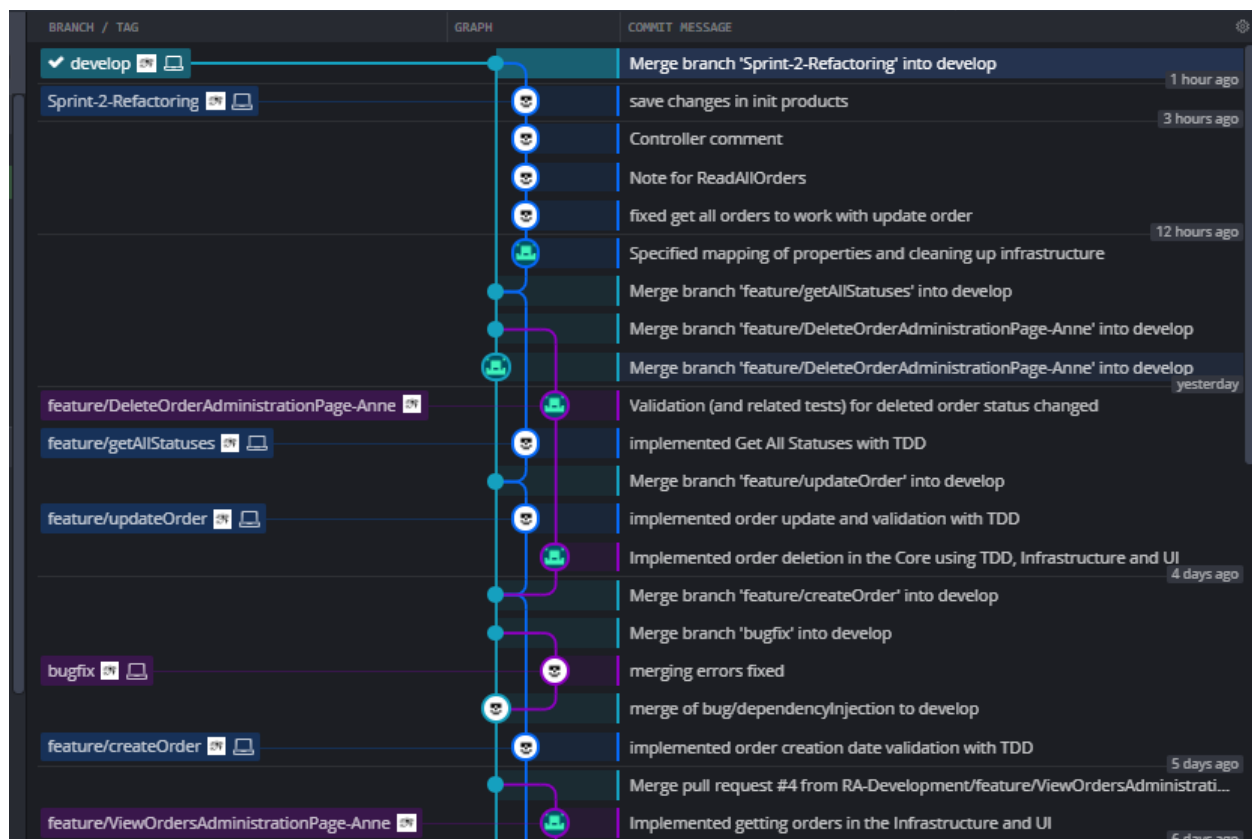


Figure 40. Branching model example from GitKraken

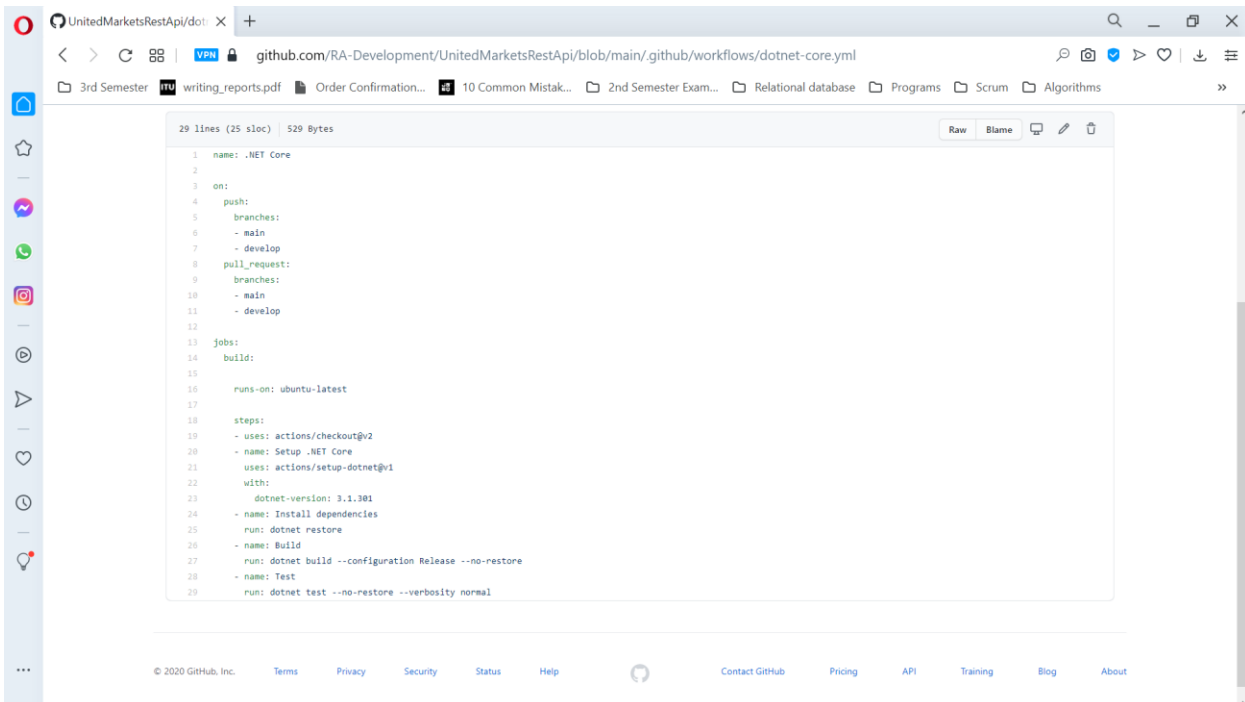
The above strategy was used throughout the project, except the committing user stories from Sprint 1 to the main. The progress after Sprint 1 was very minimal, so the team decided to forego merging to the main branch.

3.5 CI setup and configuration

Continuous integration (CI) was used for automated testing during the development process ensuring the team would discover any problems early on. GitHub Actions was chosen as the tool to help with CI. The team decided to use GitHub Actions, because it proved to be the easiest and most convenient compared to Travis CI and TeamCity. The setup is fast and as GitHub is used for version control, it is convenient to use the inbuilt service.

The team had no unknown problems found by the CI tool, so the advantages were not prominently felt. It did, however, serve its role as a preventive measure providing peace of mind. One of the reasons for the few merge conflicts could be due to the small team size of two.

The image below shows the YAML (.yml) file, which defines the configurations for the CI. Both the main and develop branch have CI, where the workflow is triggered on pull requests and merges. If any jobs fail, a notification email will be sent to the person performing a change to the branch (enabled in the personal account settings).



```
1 name: .NET Core
2
3 on:
4   push:
5     branches:
6       - main
7       - develop
8   pull_request:
9     branches:
10      - main
11      - develop
12
13 jobs:
14   build:
15
16     runs-on: ubuntu-latest
17
18     steps:
19       - uses: actions/checkout@v2
20       - name: Setup .NET Core
21         uses: actions/setup-dotnet@v1
22         with:
23           dotnet-version: 3.1.301
24       - name: Install dependencies
25         run: dotnet restore
26       - name: Build
27         run: dotnet build --configuration Release --no-restore
28       - name: Test
29         run: dotnet test --no-restore --verbosity normal
```

Figure 41. YAML (.yml) configuration file



**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

3.6 Conclusion

This project enabled the team to practice several SDM disciplines such as XP development methodology, TDD, branching models and CI. Pair programming was briefly attempted, but time constraints and a general preference for working alone meant it was quickly abandoned. The team also realized at the end that the project could have benefited from code reviews. Again, the lack of time (due to inexperience and lack of skills) meant the focus was on developing the features. Even discussions were remarkably sparse compared to previous projects, let alone code reviews. Overall, it was a very fruitful experience, where the team learned a lot.



4. References

- [1] Okta, "MVC in an Angular World," *scotch.io*, Feb. 25, 2019. [Internet]. Available: <https://scotch.io/tutorials/mvc-in-an-angular-world>. [Accessed: Dec 18, 2020].
- [2] P. Naveen, "MVC, MVVM and Angular," *Naveen Pete*, Sep. 07, 2016. [Online Image]. Available: <https://naveenpete.wordpress.com/2016/09/07/mvc-mvvm-and-angular>. [Accessed: Dec 18, 2020].
- [3] Contributors, "Common web application architectures," *Microsoft*, Dec. 01, 2020. [Online Image]. Available: <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>. [Accessed: Dec 18, 2020].



5. Appendices

5.1 Appendix 1: Initial project definition

Project definition:

X Company is a new business which will be launched soon. It is an online grocery delivery service, which helps customers get groceries from supermarket chains under our partners Coop Danmark and Salling Group.

We need a website for our potential customers to use our services.

Initial requirements:

1. The website should be aesthetically pleasing and have an intuitive user interface.
2. The minimum order requirements and delivery fees should be displayed clearly for the customer.
3. The customer should first choose a supermarket chain in order to browse the selection of products.
4. The customer can shop by adding products to a digital shopping cart.
5. The customer can see all products in the cart on a “Shopping Cart” page. The customer can place an order with the products in the cart on this page.
6. It should not be possible to place orders which do not meet the minimum order value.
7. There must be input validation, when customers fill out the order placement information (name, email, phone number and address).
8. The customer can choose one of three delivery time options: 12:00-16:00, 14:00-18:00 or 16:00-20:00.
9. The delivery fee is 40 kr. for shopping at one shop. If the customer wants to shop at more than one shop, the fee increases by 15 kr. per additional shop (total 55 kr.).
10. There must be a super user (admin), which has an overview of all orders.
11. All customers are guest users in the initial version. The intention is to introduce customer accounts later.



5.2 Appendix 2: Overall project schedule

Week	Date	Time	Activity
47	Nov 16, 2020		Project definition hand-in to Henrik and Lars
47	Nov 18, 2020 – Nov 19, 2020		Pre-Game (Sprint 0) planning
47	Nov 20, 2020		Sprint 1 planning
48	Nov 23, 2020 – Dec 01, 2020		Sprint 1
49	Dec 02, 2020		Sprint 1 Review
49	Dec 02, 2020		Sprint 1 Retrospective
49	Dec 02, 2020		Sprint 2 planning
49-51	Dec 02, 2020 – Dec 17, 2020		Sprint 2
51	Dec 17, 2020		Sprint 2 Review
51	Dec 17, 2020		Sprint 2 Retrospective
51	Dec 17, 2020 – Dec 18, 2020		Report writing
51	Dec 18, 2020	12:00	Hand-in deadline of the report and program on WiseFlow.

5.3 Appendix 3: Software usage

Name	Type
Zoom	Video communications
Microsoft Teams	Text, image, video and audio communications File hosting and synchronization service
Visual Studio	Integrated development environment (IDE)
Rider	Integrated development environment (IDE)
WebStorm	Integrated development environment (IDE)
Postman	API testing tool
Microsoft Azure	
Firebase	
Git	Version control
GitHub Actions	Continuous integration
GitHub (GitHub Desktop)	Git client
GitKraken	Git client
DB Browser for SQLite (DB4S)	SQLite database tool
Scrumwise	Scrum project management tool
Balsamiq Wireframes	Wireframing
Visual Paradigm	Diagramming
Microsoft Word	Word processing



5.4 Appendix 3 Scrumwise documentation

Sprint 1

Sprint 1 Backlog at the end of the Sprint

The screenshot displays the Scrumwise interface for Sprint 1. The top navigation bar includes tabs for Overview, Projects, People, Messages, Backlog, Sprints, Task board, Burndown, Kanban, Retrospec..., Time, and More. The main content area shows Sprint 1 details, including team members (Anne, Rado), progress (27 hours behind), and a list of backlog items. The backlog items are categorized by Epic and include tasks like 'Initial Setup', 'Browsing Page', 'Navigation Bar', 'Product Page', 'Shopping Cart Page', and 'Checkout Page'. Some items are marked as 'Done' or 'To do'.

The screenshot displays the Scrumwise interface for Sprint 1. The top navigation bar includes tabs for Overview, Projects, People, Messages, Backlog, Sprints, Task board, Burndown, Kanban, Retrospec..., Time, and More. The main content area shows Sprint 1 details, including team members (Anne, Rado), progress (31 hours completed), and a list of backlog items. The backlog items are categorized by Epic and include tasks like 'Initial Setup', 'Browsing Page', 'Navigation Bar', 'Product Page', 'Shopping Cart Page', and 'Checkout Page'. Some items are marked as 'Done' or 'To do'.



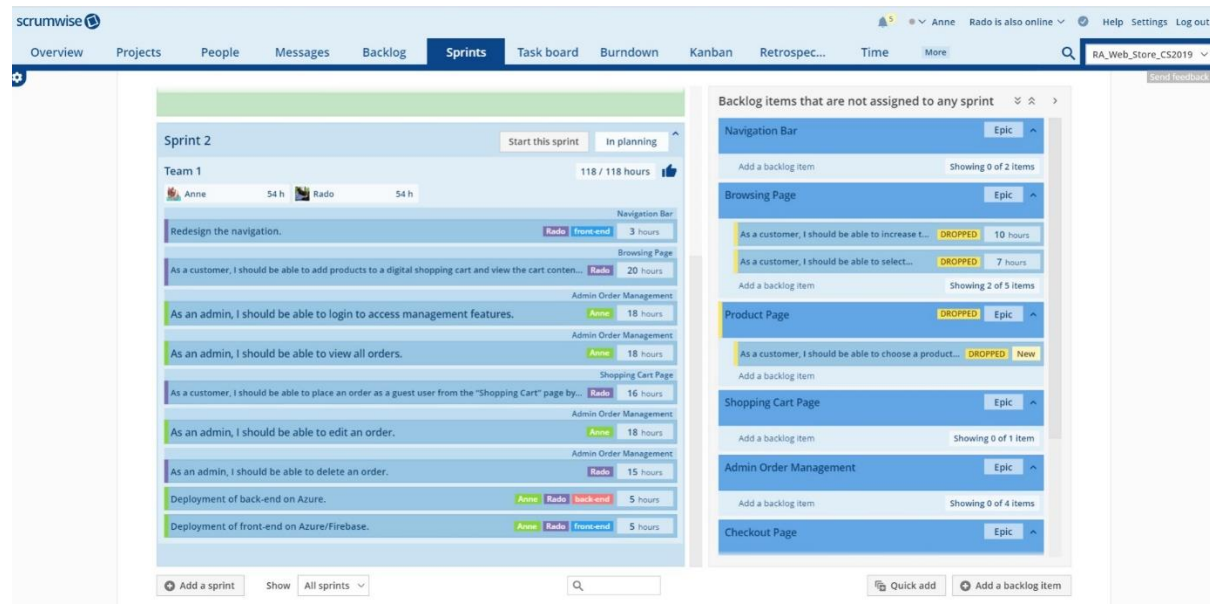
Sprint 1 Burndown chart where the two last user stories (tagged DROPPED) were removed resulting in assigned hours to drop drastically.





Sprint 2

Sprint 2 Backlog before starting the Sprint.



Sprint 2 Burndown chart at the intended end of Sprint. The Sprint was extended so the team could finish the required user stories. The team ended up stopping at December 17, 2020 with deployment left.





**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

scrumwise

Overview Projects People Messages Backlog Sprints Task board Burndown Kanban Retrospec... Time More

RA_Web_Store_CS2019

Sprint 2 Complete this sprint In progress

Team 1

Anne 0 h left Rado 22 h left 32 hours behind

Navigation Bar

Redesign the navigation. Rado front-end 0 h left To test

Browsing Page

As a customer, I should be able to add products to a digital shopping cart and view the... Rado 0 h left To test

Admin Order Management

As an admin, I should be able to login to access management features. Anne 0 h left To test

Admin Order Management

As an admin, I should be able to view all orders. Anne 0 h left To test

Shopping Cart Page

As a customer, I should be able to place an order as a guest user from the "Shopping... Rado 0 h left To test

Admin Order Management

As an admin, I should be able to edit an order. Rado back-end 13 h left In progress

Admin Order Management

As an admin, I should be able to delete an order. Anne 9 h left In progress

Deployment of back-end on Azure. Anne Rado back-end 5 h To do

Deployment of front-end on Azure/Firebase. Anne Rado front-end 5 h To do

Backlog items that are not assigned to any sprint

Navigation Bar Epic

Add a backlog item Showing 0 of 2 Items

Browsing Page Epic

As a customer, I should be able to increase t... DROPPED 10 hours

As a customer, I should be able to select... DROPPED 7 hours

Add a backlog item Showing 2 of 5 Items

Product Page DROPPED Epic

As a customer, I should be able to choose a product... DROPPED New

Add a backlog item

Shopping Cart Page Epic

Add a backlog item Showing 0 of 1 Item

Admin Order Management Epic

Add a backlog item Showing 0 of 4 Items

Checkout Page Epic

Add a sprint Show All sprints Quick add Add a backlog item

scrumwise

Overview Projects People Messages Backlog Sprints Task board Burndown Kanban Retrospec... Time More

RA_Web_Store_CS2019

Task board

Backlog item To do In progress To test Done

Admin Order Management

As an admin, I should be able to edit an order. Rado Team 1 18 hours In progress 28%

Admin Order Management

As an admin, I should be able to delete an order. Anne Team 1 15 hours In progress 40%

Deployment of back-end on Azure. Anne Rado back-end Team 1 5 hours To do

Deployment of front-end on Azure/Firebase. Anne Rado front-end Team 1 5 hours To do

front-end Rado 12 h 0%

back-end Rado 9 h 100%

delete order Rado 9 h 0%

delete order Anne 6 h 100%

Sprint status In progress Complete this sprint

Edit this board Sprint Sprint 2 Show All teams



**BUSINESS
ACADEMY
SOUTHWEST**

United Markets

Radoslav Backovsky and Anne Luong

5.5 Appendix 4 Prototypes