Antallagi
Examensarbete

By Christina Bögh & Mira Aeridou

EC-Utbildning Göteborg - FEU16

Table of Contents

Chapters sorted by author

	Christina Bögh	Mira Aeridou
Introduction	X	X
Background	X	X
Introduction of Aurelia	X	
Aurelia the good parts	X	
Routes: Main Router	X	
Routes: Child Routes		X
Routes: Breadcrumbs	X	
Routes: EventAggregator	X	
Structure, Design and CSS		X
Security: Firebase authentication		X
Security: Firebase – example 1 – the first step - WHY	X	
Security: AuthorizeStep	X	
Backend		X
Conclusion	X	X

Illustration Index

Introduction[1]

Antallagi is the name of an E-commerce web-application described in this paper. It was made during 23/10 – 31/12 2017 as part of 'LIA 1' period of this paper's authors Front-end Development education[2].

The application is not yet finished and half of the development is left to be done in the following 'LIA 2' in the spring of 2018. During LIA 1 the focus was on functionality, CRUD connections and log in, which will be discussed in this paper in length. Styling, Design, CSS and advanced functions are not yet fully applied[3]

The purpose of the project was to create an application where products are second-hand articles and are sold to users. The application supports the environment through motivating people to recycle and re-use their old stuff. Our aim with this project is to produce a functional application that will serve the purpose; the users to log in, connect and buy/sell their stuff.

The resources used for the front-end development are the Aurelia framework and the Materialize and Font Awesome libraries as well as PouchDB and Firebase for the back-end development.

Application structure and navigation is controlled by a router and breadcrumbs. The Aurelia EventAggregator connects and updates the ‘information state’ of the individual pages. These three parts give the application a simple structure that reduces user clicks and confusion.

---

[1]Chapter authors: Christina Bögh and Mira Aeridou.
[2]ECUtbildning Göteborg, Sweden.
[3]The code is being worked on every day! This means today's code and look of the website might be different tomorrow. Keep this in mind when downloading the repo.

Our consistent data is stored in a PouchDB in-browser database and user login/ signup authentication is stored in the Firebase cloud database. Access and security are provided by Firebase's methods and a custom-made router Pipeline.

The methods applied are dynamic content rendering, dynamic binding of user input and customized elements and behaviour. The data separation of the application pages is ensured by a Model-View-View-Model pattern. This ensures safety, scalability and avoids 'spaghetti code’.

Background[4]

Background of the project

With no facilities to recycle in Greece, it is difficult and complicated for the Greek people to reuse resources. According to the Greek Recycling Agency, there is currently no recycling of daily use articles (clothes, furniture, electronics etc.), besides the recycling of packaging (paper, plastic, glass, metal) through officially approved systems (Greek Recycling Agency, 2018, para.1).

There are few companies recycling clothes, for example, 'Recycom' and for electronics 'Greenfence' (Recycom, n.d.),(Greenfence, n.d.). But they deliver only a minuscule portion of what the users actually need and many donate to churches instead or use 'word-of-mouth'.

Since it's recognized that environmental issues like reusing resources are a global and national problem, collaboration to reach this goal should be unrestricted. Therefore the goal of our project is to address this challenge and propose a successful implementation for a solution of the problem mentioned above.

[4]Chapter authors: Christina Bögh and Mira Aeridou.

Target audience

The target audiences for our application are:

Families

Especially families with children need upgraded and new wardrobe constantly. Also, families have a limited budget and can earn money by selling used and save money by buying cheap.

Young people

The generation z, which are born mid 90's to mid 2000's are comfortable with new technology since they have been using the Internet from a young age.

Mentality of exchange/reuse

Modern society has several social movements that support environmental sustainability. For example: 'Eco-feminism' and the 'Green movement'(Encyclopedia.com, n.d.), (Brammer, 1998).

Students

The limited budget that many students live under makes buying cheap 2nd-hand 'things' attractive.

Collectors

Collectibles and antiques are found or sold easier through and an online, national exchange.

Retirees

Retired life has plenty of time and often little money, which gives a good opportunity for looking for and bidding on discounts.

Objective of the project

Our application supports the environment through motivating people to recycle and re-use their old stuff. Our aim with this project is to produce a functional application that will serve that purpose; the users to log in, connect and buy/sell their things.

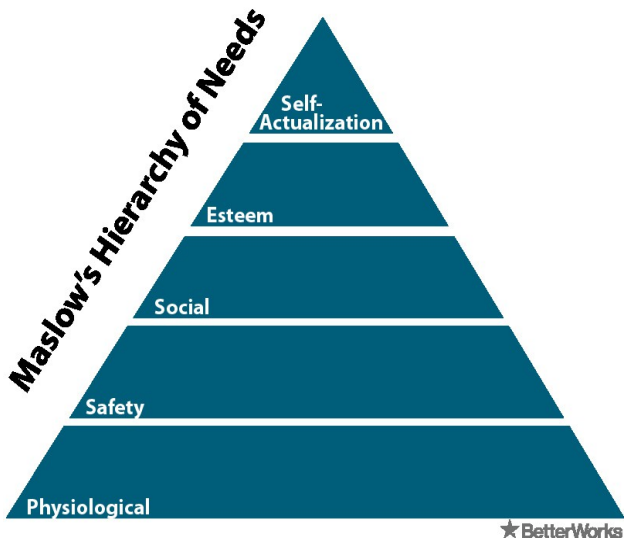


Illustration 1: Maslows pyramid of human needs.

Limitations

1: Cultural limitations:

According to Maslow's pyramid the basic needs are the physiological needs of water and food and the need for shelter and safety. Greece is a country with a difficult past with many wars and political instability. For many generations they have been on the bottom level of the pyramid. They were focused on providing for their **basic needs** until fascism and monarchy were abolished in 1973 (BBC, 2017).

The realization of needing to recycle is on the top of the pyramid where you have reached your full potential. You can only advance to a higher step if you have first fulfilled the current steps needs entirely (Khanacademy, 2014). In contrast to Greece, the Swedish society had already reached the top of the pyramid in the 1960's when they democratized and transitioned public space with the ‘du-reform’. This was when they exchanged the plural form 'ni' to the single form 'du' (Harrison, 2016). Society adapted a new morality and 'self-image', which are the characteristics of the top of the Maslows pyramid.

The Greek cultural limitation discussed above means for our application that this limitation works in our favor as Greece doesn't yet have a 2nd-hand exchange online service that is settled.

2: Economic limitations:

The Greek economic crisis started in 2008 and they have not yet recovered from that (The New York Times, 2016). As a result the income is low and unemployment high, which gives almost everyone a very limited budget. This means that selling and buying 2nd-hand articles becomes necessary and our application fills this demand.

Specifically for our application, the cultural and economic limitations, also mean that there aren't advanced payment applications in Greece (for example: no Swish, Direct Bankpayment & Klarna like in Sweden). For that reason, payment options on our page are going to be limited to credit card, Paypal and Pay on delivery.

Further study

In the future (during LIA 2 and after) further studies need to be done in the areas of payment options and marketing plans to accommodate the economic and cultural limitations.

Introduction of Aurelia[5]

WHAT is Aurelia

Aurelia is a framework for building Javascript applications.

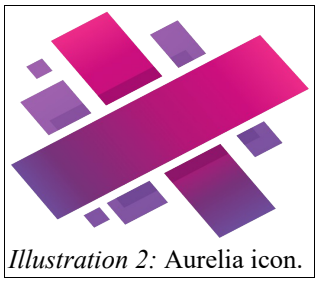


Illustration 2: Aurelia icon.

- It works on all modern browsers,
- It has no external dependencies,
- It has its own router (see chapter 'Routes'), and
- It is modularized.

This is done in such a way that you can choose the modules you need, for either full web-apps, normal websites, use the node.js web-server, or even create your own framework with it ("Introducing Aurelia," 2015, para. 2). What this means is, that you can simply download the Aurelia package with NPM, follow the guided setup and then run the app (on any system or browser) without needing to install and add much else.

WHY did we choose Aurelia

When we first decided to make this web application, the question of choice of Framework came up. Decision was made based on several considerations;

- small simple framework with easy to manage code → less code less bugs.
- mainly uses vanilla JavaScript but also the newest ES6 and ES7 (transformed to ES5) → we can learn the new cool syntaxes.
- has highly adaptable components, and individual independent views → adding a new component, module or view should be simple.
- and has all the features with no need to import other things → the all-in-one package is the way to go.

[5] Chapter author: Christina Bögh.

How does Aurelia back up our business

The Antallagi e-commerce website is based upon database requests and dynamic rendering of content. Users can add and remove products and the application as a whole is in constant flow and change.

To accommodate those needs we setup the Aurelia application in a MVVM pattern (Model-View-View-Model). The MVVM model has a binding-system to connect the relations between a Views elements and a ViewModels properties. The illustration below shows the connections.

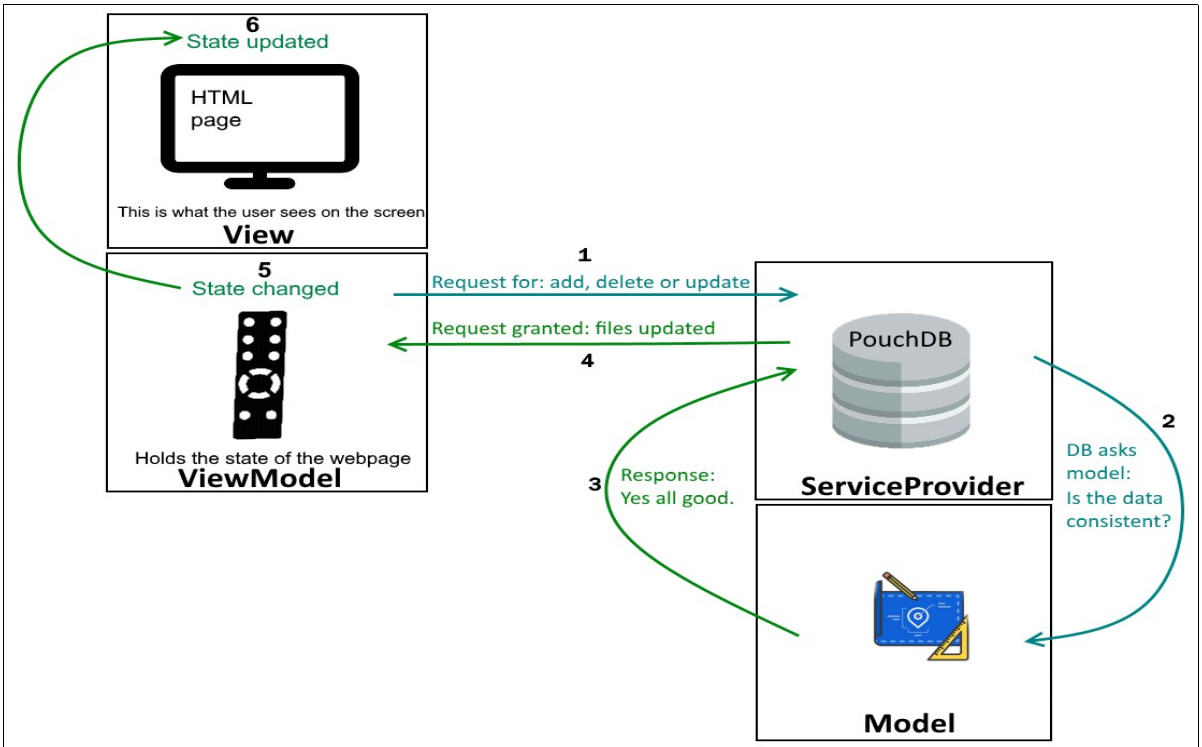


Illustration 3: MVVM pattern.

The 'ServiceAgent' is our PouchDB database and the 'Model' is a JavaScript class that describes how data added to the database has to be structured. The 'View-model' sends and receives information from the database and the 'View' actually displays the data.

This setup was chosen because the division between information creates a safer and more stable application. The consistency of data in the database is controlled by the model. The operations to add, remove and update the database (as well as error handing) are handled solely by the ServiceAgent. The connection to the database is handled by the View-Model. And the View, which is the part that is visible to the user, is kept separated from the logic of the application and only serves as a window to show selected data.

Aurelia, the good parts[6]

1) Binding

WHAT is binding?

The relations between a website (view, HTML-file) and its state (view-model, js-file) are really between the **HTML elements** in the view and **class properties** in the view-model. They are connected, or 'bound' together in such a way, that changes in one reflect in the other as well.

One could compare the view-model with some real object out in the world, and the view is the mirror image, reflecting how the object looks right now. If the object changes, moves or other items are added, the mirror will instantly reflect such changes also.

In order to bind an HTML element's attribute to its view-models property, a specific syntax has to be followed:

```
attribute.command="expression"
```

(Aurelia," n.d.-b).

Most HTML elements (tags); for example <input> or <button>, can have `attributes` (for example type='x'). The attribute can, with a `command` like 'bind', be bound to an `expression`. The expression is a property in the view-model and can be any kind of data (string, number, object etc.).

[6]Chapter author: Christina Bögh.

WHY is binding good for us?

Our e-commerce website is highly reactive to user input, user- and product-state and needs a system to update those changes in an easy and uncomplicated way. The goal is that the user wont notice latency and always see what he expects to see.

Aurelia's binding system allows for this to happen in a seamless manner. The Aurelia toolkit comes with several options for binding, depending on the needs. As you can see in the image below, the options are:

- From view to view-model: 'from-view'.
- From view-model to view: 'one-way' or 'one-time (runs once).
- Or both ways: 'two-way'.
- But if its hard to decide, you can let Aurelia do the binding automatically and simply choose 'bind'.

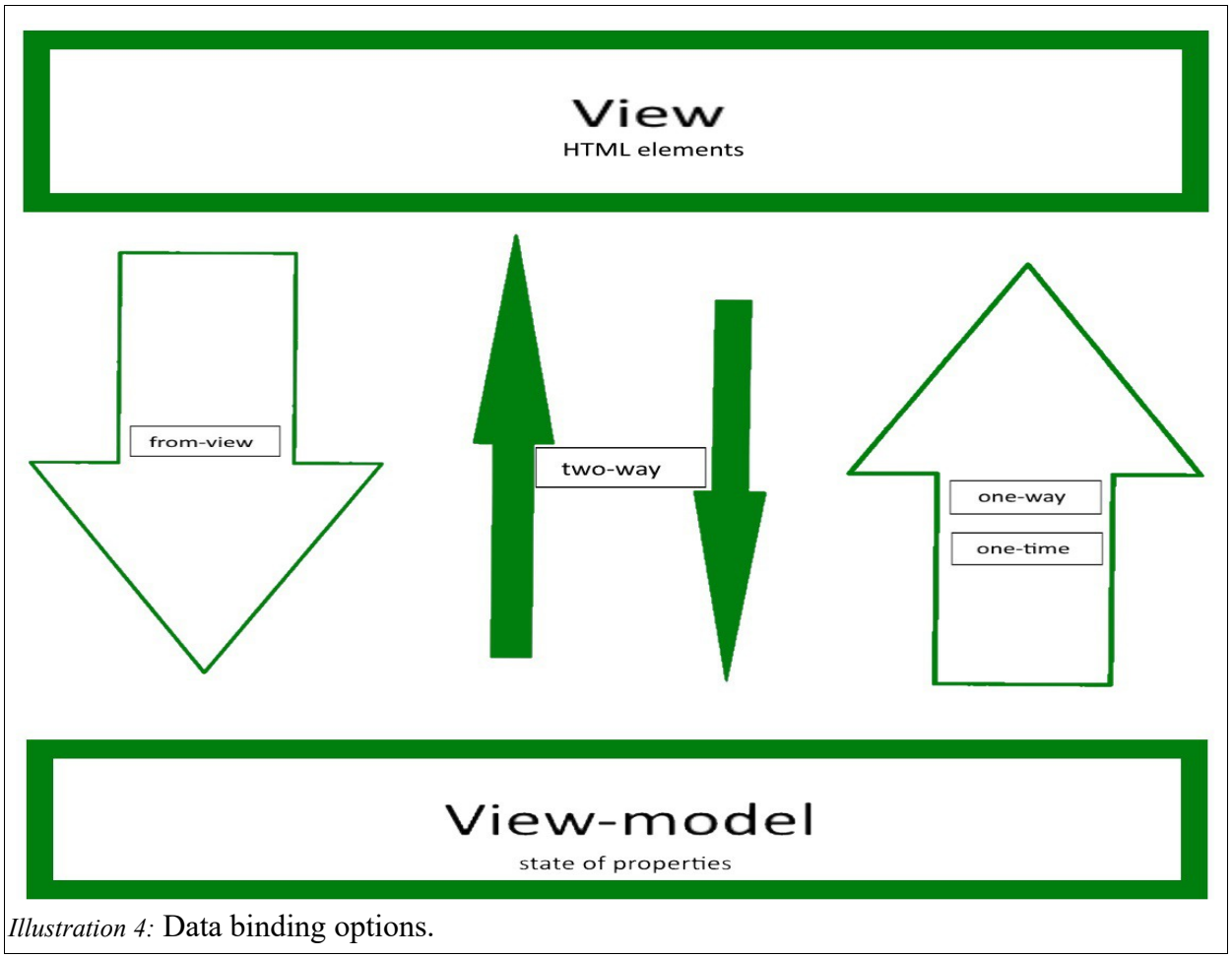


Illustration 4: Data binding options.

HOW does binding help us?

When Aurelia binds the HTML to the property, it creates an observer on the property so that when the property is changed, a notification of the change is triggered as an event.

- Binding data one-time for properties that do not change **improves performance** since the view only needs to load it from the view model once and doesn't need to check for changes.

- Binding data to or from the view-model (or both ways) gives us control over when and how data can be changed and makes data flow highly **flexible and secure**.

Example: binding in our app

Example 1: Bind value to state.

On our web page in 'sell' section, any user can enter a new product to be sold. Therefore there are several inputs fields as part of a form. The data entered in the input fields have to be handled with binding. See image below:

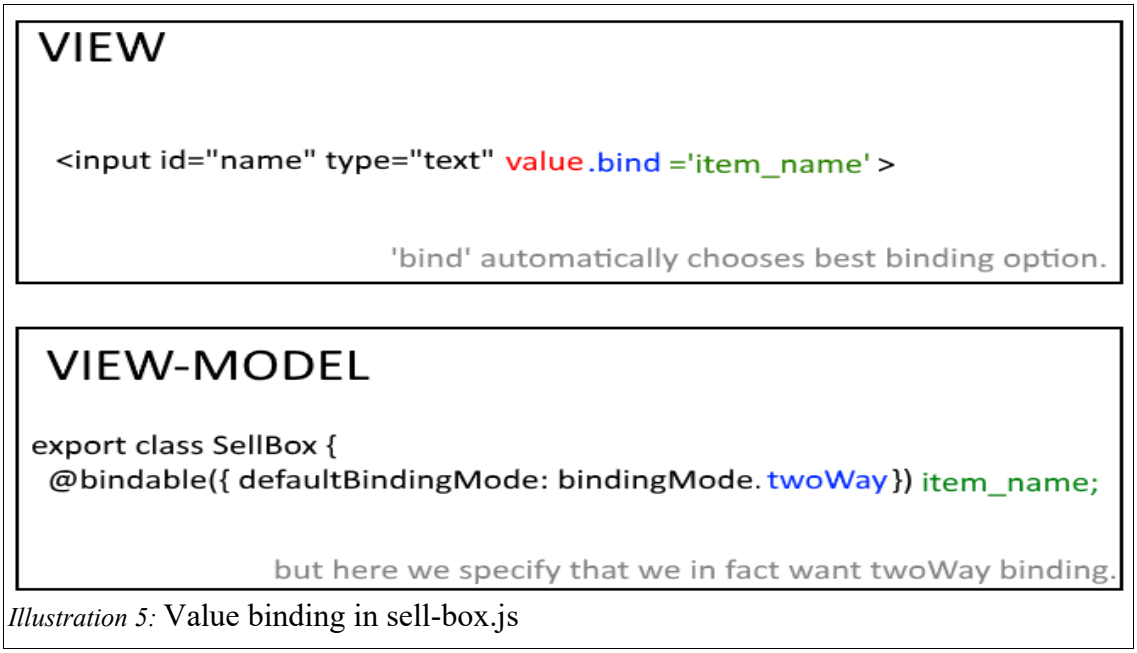
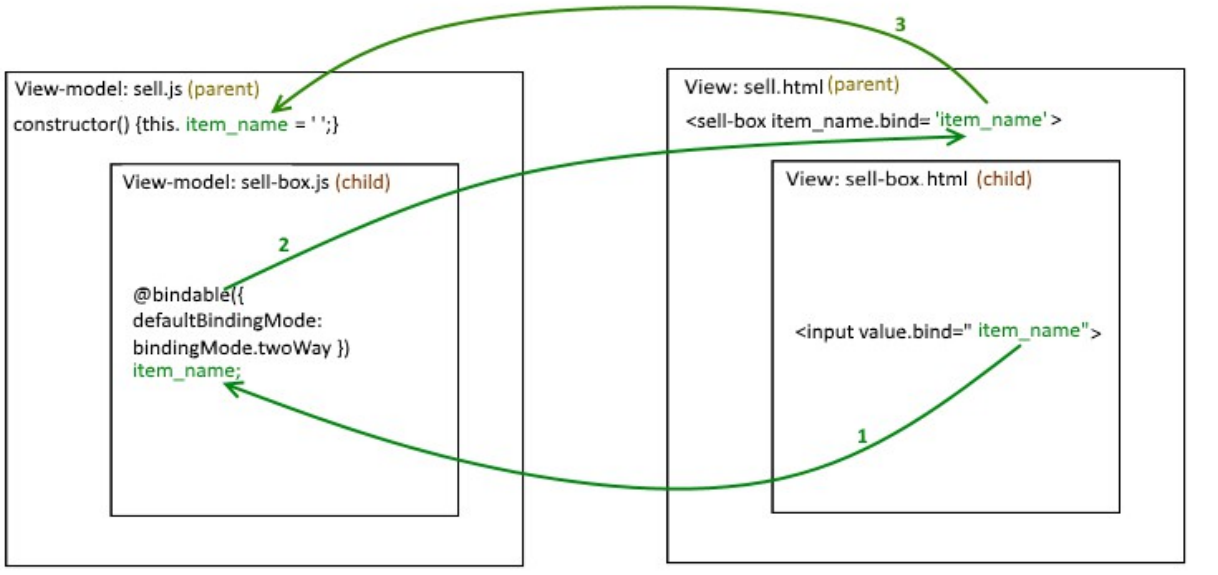


Illustration 5: Value binding in sell-box.js

The property `item_name` is two-way bound to the view. This means that if the value is changed in the state of the class where it resides or if it is changed in the input field, both will update. This way, the item_name is saved in the state of the view-model and when the form is submitted will be transmitted to the database.

Example 2: Share binding from one view to another.



I

llustration 6: Share binding from child to parent.

In the previous example, we used 'bind' in the input element and 'twoWay' in the view-model. The reason for this is that the property **`item_name`** belongs to the parent component. The view and view-model 'sell-box' is a child of the 'sell' component and the **`item_name`** property that the binding are updating is found in sell. See image below:

The view-model of the *sell.js* parent component has in its constructor the 'item_name' expression.

```
export class Sell { constructor() { this.item_name = '   '; } }
```

Aurelia's binding makes it easy, smooth and simple for us to pass data between view-models and views. Keeping the code simple and uncomplicated is important for easy debugging and also allows for changes to be made without many problems;

simple change the 'bindingMode.twoWay' to a 'bindingMode.one-way' for example.

2) Customized elements

WHAT are customized elements?

In Aurelia Views all HTML is placed inside **<template>** tags. According to MDN a template is a 'content fragment that is being stored for subsequent use in the document' (2017). This below is a simple example of a View template tag:

```
<template>
    <p>Hello, ${name}!</p>
</template>
```

In Aurelia rendering content is always defined by the logic in the JavaScript class that is linked to the template.

In this example the View-model defines the property **name**:

```
export class Test {constructor() {this.name = 'John Doe';}}
```

Every view is a customized element in Aurelia or can be used as such. The name of the view/view-model is the name of the custom element tag. The template tags are 'web components' and so are custom elements in Aurelia. Their name needs to have a dash, so the HTML-parser can tell them apart and ensures forward compatibility. For example <sell-box> from the example above.

WHY do we need to customize ?

The use of custom elements and web components creates a framework of encapsulated tags that can be reused and are poly-filled so every component can run even on browsers that don't support web-components (Webcomponents.org, n.d.).

HOW does customization help us?

Custom elements let us re-use elements anywhere and as many times as we want. This saves time and makes the code less complicated (and shorter). As you can see from the *sell.html* View below, the template holds two custom-elements; sell-progress-bar which is the sell-menu, and the sell-box, which holds the sell-form to add a new product – in less than 20 lines of code.

To add a custom element to a template do two things:

- require the child view and view-model/view files.

```
<require from="../../custom-elements/login/sell/sell-box"></require>
```

- add the child element anywhere within the <template> tags in the parents view

```
<sell-box item_name.bind='item_name' ...> </sell-box>
```

```
<template>
    <require from="../../custom-elements/login/sell/sell-progress-bar"></require>
    <require from="../../custom-elements/login/sell/sell-box"></require>

      <sell-progress-bar></sell-progress-bar>

      <sell-box item_name.bind='item_name'
                image_url.bind='image_url'
                startprice.bind='startprice'
                endprice.bind='endprice'
                category.bind='category'
                town.bind='town'
                condition_new.bind='condition_new'
                time.bind='time'
                description.bind='description'
                add-callback.call='addSellItem($event)'>
      </sell-box>
</template>
```

Illustration 7: View of sell.html.

3) Customized behavior

WHAT is customized behavior?

Customized behavior means that the actions and reaction that change the website's content can be changed and adjusted to fit specific needs. The keywords here are *events* and *dynamic* content. Events are actions and changes in the browser that either the user or another external or internal process made happen. This can be a mouse-click, a timer that runs 'something' after /before/during a certain time, a database update or a thousand other things. Examples could be:

- A text-box has a default attribute setting which we want to change.
- Display of lists of items depending on what is in the database.

WHY do we need it?

Making websites user-friendly is a front-end developers job. Websites need to be easy to use, pleasant and working as expected, which altogether is a difficult job. Many times a programmer wants/needs to change a little aspect of something and in Aurelia changing anything is simple and uncomplicated.

HOW does this benefit us?

In our website specifically, we have used custom attributes and dynamic rendering especially. Custom attributes are especially useful for form and button elements where the default behavior isn't the best option. Dynamic rendering makes showing lists simple and reliable.

Examples: customization in our app

Example 1: Custom attributes

An event listener can be added to the view-model to listen for specific events. For example, it will react if and when a user types into the input field.

In our date input, we wanted a 'placeholder' text to be displayed when the page loads. When the input field is clicked, we wanted a date picker to be displayed instead of the placeholder. See the image below.

Illustration 8: Date field in sell-box.js.

Input fields need a placeholder; to signal the user what to write and to improve user-friendliness. Therefore we have to customize the attribute of the input field. The attribute 'type' of an input element defines which type of data it accepts, and a date field as default has no 'placeholder' text. That is a problem! So, in stead, we start out with having it as a type 'text'.

```
<input type="text" placeholder="Choose date">
```

In the view-model, we add an event-listener to listen to the `onfocus` event attached(). See image below. When the input field is focused, it changes to a date field and the date can be entered. The `onfocus` is activated by either clicking on the field with the mouse or tabbing to it. If the input field looses focus (and if nothing has been entered into the field yet), the date input reverts back to a text field with the `onblur` event-listener that is also in the attached() function.

This way we have the placeholder like we wanted AND we have the input field accept dates and thus the customization of the input element has solved our problem.

```
attached(){
  var dtt = document.getElementById('date');
  dtt.onfocus = function (event) {
      this.type = 'date';
      this.focus();
  }
  dtt.onblur = function (event) {
      this.type = 'text';
  }
} // end attached
```

Illustration 9: attached() in sell-box.js.

Example 2: dynamic rendering

The user-interface can be rendered dynamically with template controllers (*if* and *repeat)*. This is especially useful when lists of items have to be displayed. For example a dynamic list of all the products. Below you see a tag with the Aurelia specific syntax 'repeat.for'.

```
<div class="col xs12 s6 m4 l3 center-align" repeat.for="item of allItems" id="item.id">
    <a route-href="route: id; params.bind: { id: item.id }">
```

Illustration 10: Dynamic list in View of categoryOne.js

The list is bound to the **expression** `'allItems'` and will create one new tag for every item in the list with the attribute **repeat** and the command **for**. Essentially it loops the array (`allItems`) and will add as many divs to the DOM as are elements in the list.

Now, `allItems` is a property in the view-model and has to be either an array or object – in this case, an array. Because the list is a list of items from the database, it is acquired and added to the view-model with a 'getter' function:

```
get allItems() {
    return this.SellItemsDatabase.sellitems;
}
```

Illustration 11: Getter function for allItems in categoryOne.js.

This way we show a dynamic list from the database in two simple steps without the need to write any loops ourselves. The dynamic rending displayed in this example is also used in the main and child menus (where the list is the route-array) as well as any other list on the website.

It is even possible to add conditionals to this:

```
<div if.bind="$first">First item</div>
```

This code would only show the first item of the list. Other options are (among others):

- $last would show the last,
- $middle, shows the middle item.
- $even and $odd show only either even or odd indexes.
- $index shows the index number of the item

Routes

Main Router[7]

WHAT are Routes?

In an Aurelia application routing to all pages is done with the help of Aurelias Router class. The Router makes it possible to link all the pages of the application together and change the view. All the pages in our application are displayed on the sitemap below. The main router in shell.js has 11 different routes. See image:

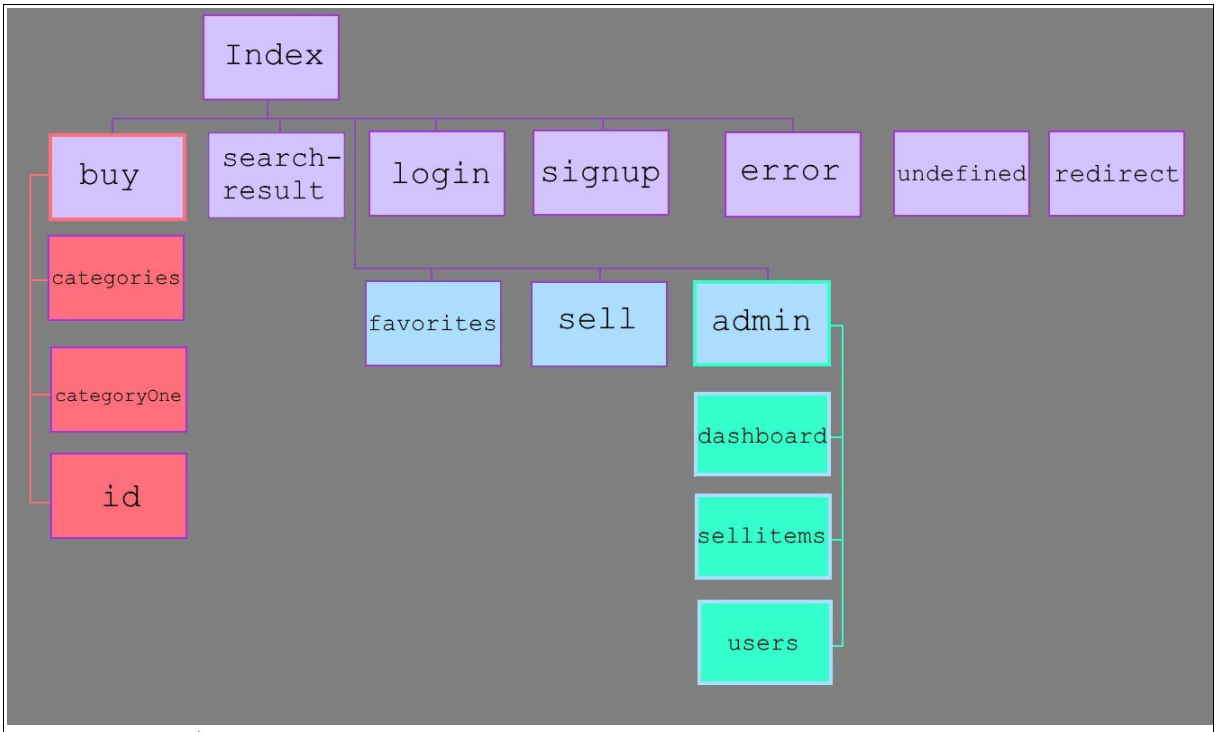


Illustration 12: Sitemap.

- The 8 purple routes: 'index', ,'buy', 'search-result', 'login', 'signup' and 'error', are available for anyone to see. The routes 'error', 'undefined' and 'redirect' are save-guards to account for 'odd' routes that might occur and to limit routing errors.

[7]Section author: Christina Bögh.

- The 3 light-blue routes: 'favorites', 'sell' and 'admin' are only available for logged in users, and admin only for registered admins of the website.

The three pink ('categories', 'categoryOne', 'item-selected') routes are part of the child-router in the buy component. The three turquoise routes ('dashboard', 'sellitems' and 'users') are part of the child-router in the admin component (see chapter 'Child Routers').

WHY do we have them?

In Aurelia apps, if you want more than a single page, you need a Router. An example of a route is index:

```
{
route: 'index',
name: 'index' ,
moduleId:'./routes/index/index',
nav: false,
title: 'Index',
roles: ['index']
}
```

A Router can be used to render menus dynamically and is a easy way to quickly overview the layout of the entire application (since every page of the app is found in the router). The property 'nav' is true/or false and defines if the route should show in the menu or not. Also, the router is used as a security measure for access (see chapter 'Security') and HTML pages that are not in the router can not be displayed.

Example: HOW is the router making our page better?

The properties of the router can be adjusted to set global properties for all routes with the ‘configureRouter’ function. This creates a ‘config’ object to hold all routes and settings. We can

add a application title: `config.title = 'Antallagi';` This simply adds the name of the app ‘Antallagi’ to be added to every route. This is equivalent to the <title> tag and will be shown in the browser head. In every route another page specific title can be added (as explained in the next chapter). In index the page header will look like this image:

Illustration 13: Index route title.

Child Routers[8]

Buy and Admin routers

Two of our routes in our main navigation have a second navigation level when clicked. These two routes are the 'buy' and 'admin'. That means that when either one of them is selected, we get redirected to the buy or admin component as defined by the moduleId in the main router.

Buy

```
moduleId: './routes/buy/buy'
```

Admin

```
moduleId: './custom-elements/login/admin'
```

In these two components we have created our child-routers. The child-routers follows the main router's rules and looks almost the same. The only difference is that we're not using the 'config.title' like we did in the main router. This is not needed because our child router already has a title that is defined in the main routes. In the illustration below, is the buy route from the main router, where you can see the 'title' property:

```
{ route: 'buy', name: 'buy', moduleId: './routes/buy/buy', title: 'Buy', roles: ['buy']}
```

Illustration 14: Buy route.

[8]Section author: Mira Aeridou.

WHY do we need the child-routers and HOW do they benefit our application

In order to create a secondary navigation that renders dynamically, you will have to create a child-router inside of the component where it is rendered. Placing, for example, the second navigation level in the main router would not allow you to render it in another component. That means that when we click on the buy or admin button we want to be redirected to the matching component that will display the secondary navigation.

In the picture below you can see our three components where we have our routers:

- index → main router
- buy → child router
- admin → child router

When you click on the buy-route (that is in the main router) you get redirected to the buy page, and when you click on the admin-route (in the main router) you get redirected to the admin page. See image below.

Illustration 15: Child-routers.

Another reason for using child routes is that it helps us keep our code structured. We are creating a big application with many lines of code that can very easily get chaotic, so structuring our code is something very important because it makes it easier for us to find what we want. When working in a team, not following some basic lines about structure can make the cooperation very difficult. Sometimes we are working with the same pieces of code and then it's important to find what we're looking for easy, to be able to edit it or add new things.

Breadcrumbs[9]

WHAT are breadcrumbs?

Breadcrumbs are navigational links that show the user where he has been compared to where he is now. Our breadcrumb scheme is location-based and shows the user where he is in the website hierarchy. We chose this design because our e-commerce website will have more than 3 levels and thus will provide users will an easy overview of a complicated structure. The illustration below illustrates the page hierarchy of the breadcrumbs logic.

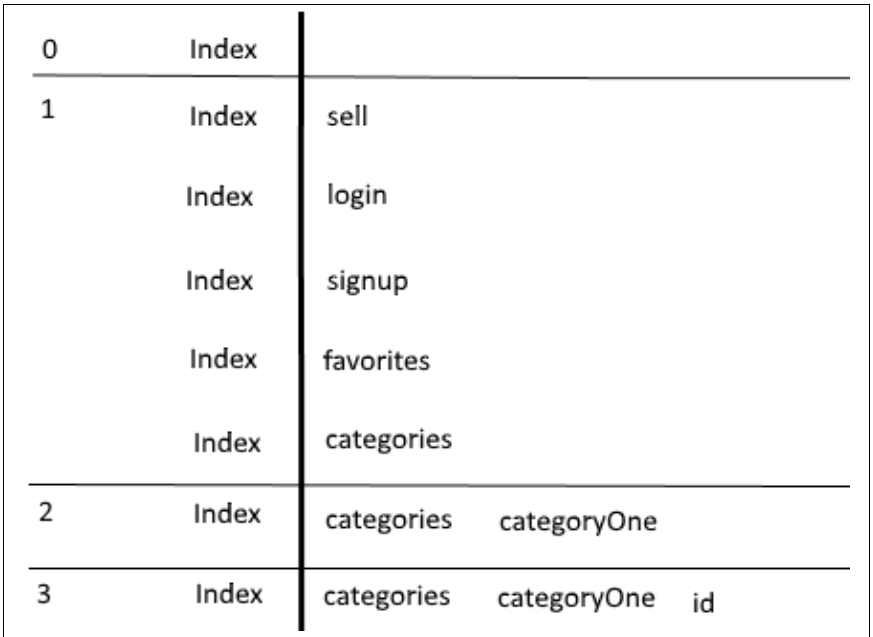


Illustration 16: Breadcrumbs page hierarchy.

For example, if you are on the categories page, you will be on level 1 of the breadcrumb hierarchy. Commonly breadcrumbs are horizontal list items separated by a "greater than" symbol (>) and are used in the format of *Parent category > Child category*.

[9]Section author: Christina Bögh.

Our breadcrumbs look like this below. The current page is styled in bold and is not a link In the illustration 17, the '**1**' to the far right is the current page. If you hover over the other breadcrumbs; a link in electric green with an underline appears as you can see.

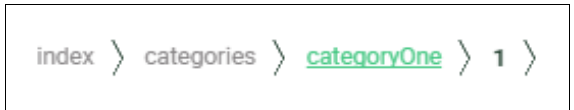


Illustration 17: Breadcrumbs.

The size of the breadcrumbs is smaller - or less noticeable than the primary navigation. this is because they are only a little helper and should not dominate the page.

WHY do we need them?

Breadcrumbs are not a built-in feature in Aurelia, so the breadcrumb algorithm and implementation we made is done entirely by ourselves and unique in this way. There are plug-ins and libraries we might have used, but we thought it would be better to 'own' our own code and not depend on a 3rd party.

We choose to have breadcrumbs because of these benefits:

- Breadcrumbs are a **secondary navigation**; whereas the normal menu structure is the primary. In websites with several levels, like ours, breadcrumbs give the user a fast way to reach the lower levels that they have visited before.
- The result of this is **reduced number of clicks** and the user doesn't need to use the 'back' button on the browser.
- Furthermore, because the breadcrumbs are small and they have a **very little negative impact on content overload** (Babich, 2017, para. 7).

HOW are they helpful?

1 Adding a breadcrumb component

The breadcrumbs are custom components that are added to a view-model at the position where they should be displayed. To add a breadcrumb component to any page, simply add the import and the breadcrumbs element tag.

Convention says to place breadcrumbs to the left under the header and above other content. Therefore we have our breadcrumbs right after the `<template>` tag on all pages like the illustration above shows.

```
<template>
<require from="../../services/breadcrumbs/breadcrumbs"></require>

<div class="categoriesContainer">
    <div class="row .breadcrumbsmargin">
        <breadcrumbs></breadcrumbs>
    </div>
</div>
```

Illustration 18: Adding breadcrumbs to buy-box.js.

2 Implementation

a) Parts

The breadcrumbs are divided into two sections, **two arrays**, which hold the breadcrumbs:

```
export class Breadcrumbs{

  constructor(eventAggregator) {
    this.eventAggregator = eventAggregator;
    this.start = []; // first item of path. this is always index.
    this.path = []; // all the middle
  } // end constructor
```

Illustration 19: Breadcrumbs.js constructor.

The property ‘start’ holds either nothing or the index route. Index is always going to be the first breadcrumb. Unless we are already on the index page, in which case it will be empty as there is no need to show breadcrumbs on the start page.

The property ‘path’ holds the rest of the breadcrumbs in hierarchical order, determined by the logic of a switch statement.

The **format** of the breadcrumbs is an object with two properties. The 'index breadcrumb', for example, looks like this:

```
{name:'index' , href: 'index'};
```

b) Algorithm

When a new page is loaded the switch statement checks which route it is. Depending on the name, the start and path arrays will be populated with different content. Because the breadcrumbs are location based, the order of and variation on breadcrumbs trails is limited. Therefore it fits well into the switch structure.

```
findBreadcrumbsfromDataChanged(data) {
  // these three lines get the number id of the page. this is used for the 'id' case.
  let d = window.location.href;
  let str = d.substring(d.lastIndexOf("/") + 1);
  let str2 = Number(d.substring(d.lastIndexOf("/") + 1));
  try {
    switch (data) {
    case 'sell':
        console.log('sell');
        this.start = [{name:'index' , href: ''}];
        this.path  = [{name:'sell' , href: '#/sell'}];
        break;
    case 'index':
        console.log('index');
        this.start = [{name:'index' , href: ''}];
        this.path = [];
        break;
    case 'buy':
        console.log('buy');
            this.start = [{name:'index' , href: ''}];
        break;
    case 'categories':
        console.log('categories');
        this.start = [{name:'index' , href: ''}];
        this.path  = [{name:'categories' , href: '#/buy/categories/'}];
        break;
    case 'categoryOne':
        console.log('categoryOne');
        this.start = [{name:'index' , href: ''}];
        this.path  = [{name:'categories' , href: '#/buy/categories/'},
                      {name:'categoryOne' , href: '#/buy/categories/categoryOne'}];
        break;
    case 'categoryTwo':
        console.log('categoryTwo');
        this.start = [{name:'index' , href: ''}];
        this.path  = [{name:'categories' , href: '#/buy/categories/'},
                      {name:'categoryTwo' , href: '#/buy/categories/categoryTwo'}];
        break;
    case 'id':
      console.log('id');
        // this checks if the id is NaN or not. If its NaN then the router has not yet loaded the page.
        if(this.reallyIsNaN(str2) !== true){
          this.start = [{name:'index' , href: ''}];
          this.path = [
                {name:'categories' , href: '#/buy/categories/'},
                {name:'categoryOne' , href: '#/buy/categories/categoryOne'},
                {name: str2 , href: '#/buy/categories//categoryOne/' + str2}
                ];
        }
        break;
    case 'login':
          console.log('login');
          this.start = [{name:'index' , href: ''}];
          let login = {name:'login' , href: '#/login'};
          this.path.push(login);
        break;
    case 'signup':
          console.log('signup');
          this.start = [{name:'index' , href: ''}];
          let signup = {name:'signup' , href: '#/signup'};
          this.path.push(signup);
        break;
    case 'favorites':
          console.log('favorites');
          this.start = [{name:'index' , href: ''}];
          let favs = {name:'favorites' , href: '#/favorites'}
          this.path.push(favs);
        break;

    } // end switch
    } // end try
```

Illustration 20: Breadcrumbs algorithm.

3) Rendering

Both arrays, start and path, are being dynamically rendered on the breadcrumbs view with an unordered list. See image to the right.

```
<ul id="crumbs">
    <li repeat.for = "row of start">
        <a href.bind =row.href >${row.name}</a>
    </li>
    <li repeat.for = "row of path">
        <a href.bind =row.href >${row.name}</a>
    </li>
</ul>
```

Illustration 21: Breadcrumbs list in the View.

When the breadcrumbs are rendered, the code looks like this below:

```
▼<ul id="crumbs"> == $0
  ▼<li>
     <a href.bind="row.href" class="au-target" au-target-id="48" href="index">index</a>
   </li>
   <!--anchor-->
  ▼<li>
     <a href.bind="row.href" class="au-target" au-target-id="51" href="#/buy/categories/">
     categories</a>
   </li>
  ▼<li>
     <a href.bind="row.href" class="au-target" au-target-id="51" href="#/buy/categories/
     categoryOne">categoryOne</a>
   </li>
  ▼<li>
     <a href.bind="row.href" class="au-target" au-target-id="51" href="#/buy/categories//
     categoryOne/1">1</a>
   </li>
   <!--anchor-->
</ul>
```

Illustration 22: Developer console view of breadcrumbs list.

In order to know which route we are on, the breadcrumbs need to be updated every time a new page is loaded. This is done by using a functionality in Aurelia called EventAggregator (see the next chapter 'EventAggregator').

EventAggregator[10]

WHAT is an eventAggregator?

The eventAggregator is a built in class in Aurelia that one can use to update and loosely pass messages with the methods 'publish' and 'subscribe' (and 'subscribeOnce' which is a subscription that only fires once), (Aurelia, n.d.-b). With those two methods the eventAggregator is a postal service for messages within the components of your app.

The class is imported and injected to the class that needs it, as illustrated below.

```
import {inject} from 'aurelia-framework';
import {EventAggregator} from 'aurelia-event-aggregator'
@inject(EventAggregator)
export class Xxx        {
  constructor(eventAggregator) {
    this.eventAggregator = eventAggregator;
  } // end constructor
```

Illustration 23: Import eventAggregator class.

Now the class can use the send and receive messages to and from any other component in the app which also has the eventAggregator injected.

WHY do we need it?

Our websites setup consists of view-model and their views that are independent and loaded based on a central router in the *shell.js* component. However, the individual pages do not know the state of each other, but only itself and perhaps its parent(by using binding options as explained in 'Binding').

However, in the case of the breadcrumbs for example, it was necessary for the breadcrumbs view-model to know about the state of every other page, because users behaviour is

[10]Chapter author: Christina Bögh.

random, chaotic and asynchronous. In essence, we needed a way for all the other view-models to send a message to the breadcrumbs view-model every time they are loaded; the eventAggregator does exactly that. This feature is my most favourite part about Aurelia, and it is as beautiful as it is simple.

HOW?

1) Publish

The publish() method is how you send data. There are only two arguments to pass into the publish method, the **event name** and the **message** to be sent with the eventAggregator. See image.

```
this.ea.publish('data changed', 'sell')
```

Illustration 24: Publish method.

The first argument is the event name, which in this example is 'data changed' and it has to be a string. The second argument is the message, which here is the string 'sell'. It can be a number, string, array or object, or nothing at all.

Every component (that should have a breadcrumb) has a publish method in the attached() function of its view-model. This way, whenever the component loads, the breadcrumbs component will receive a message and know which breadcrumbs to display.

The attached() method runs every time an update is done to the DOM - also when the page first is loaded. The eventAggregator here is publishing the string ‘sell’ to the ’data changed’ event. The whole process is illustrated in the image below.

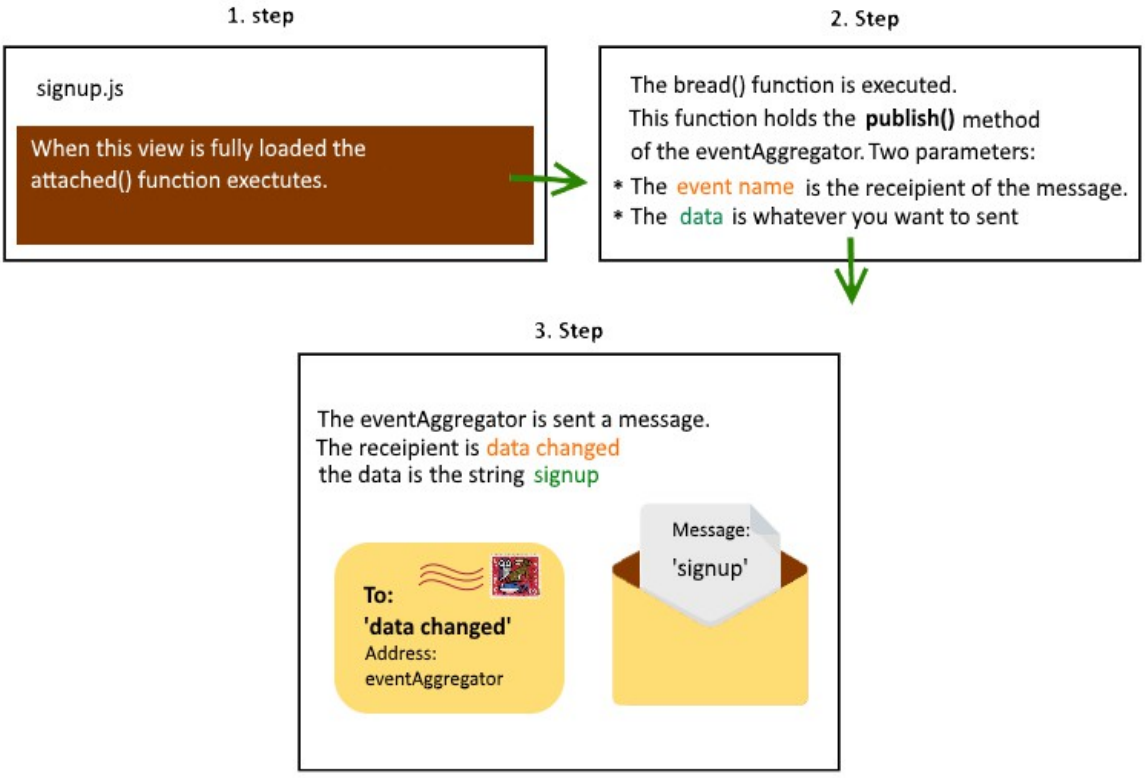


Illustration 25: Publish method example.

2) Subscribe

The subscribe() method listens to published events. This means it listens for the eventAggregator to receive a message with the string name 'data changed' and when it does, the subscription will respond and open the message.

```
bind() {  // breadcrumbs subscription. subsription is bound when component is created.
  this.subscription = this.eventAggregator.subscribe('data changed', data =>
    this.findBreadcrumbsfromDataChanged(data));
}
```

Illustration 26: Subscribe method in breadcrumbs.js.

In our case, the 'data changed' is the event name. And whenever a message with this event name is received the function 'findBreadcrumbsfromDataChanged' is executed. This is the switch algorithm for finding breadcrumbs as described in chapter XXX.

Structure, Design and CSS[11]

Simplicity and user-friendliness

Our basic focus when creating the structure, design and CSS of our application was on two things: simplicity and user-friendliness. Simplicity, to avoid pages that are overloaded with information that can create overwhelm, and user-friendliness to avoid any confusion and latency. In the sections of this chapter I am analyzing how we support these two concepts with the structure of our application and the software that we chose to use.

Wireframes

One of our first tasks in our project was creating wire-frames. We decided to do a basic structure and guideline that would be simple and user-friendly, but at the same time not focus all too much on the appearance/design in our LIA 1. The reason was because we wanted to make the basic functionalities of our application first, and then later give more weight on improving the appearance of our website. This order of doing things seemed more logical to us because when the application is ready (or almost ready), we are going to do user-friendly tests to improve the user experience of our website and that can lead to major changes in the appearance and even in the structure of the web-page.

In the two following paragraphs I will be analyzing two examples.

[11]Chapter author: Mira Aeridou

Example 1: Landing page

The first example is our landing page and our header when somebody is not logged in. The header just displays the 'logotype', a 'search bar' and 'Login'/ 'Signup' buttons. On the content of the main page we decided to display two large buttons so the visitor can easily choose what they want to do: 'Buy' or 'Sell' – see image below. These are the functionalities that we chose for our landing page in order to provide the user with the information that he needs, without making the page overloaded. The signup/login options, that serves the basic concept of our website and the sell and buy options, to make a clear separation of the two basic purposes that somebody would visit our application. So, without any description of our application anywhere, the visitor can understand what this website is all about.

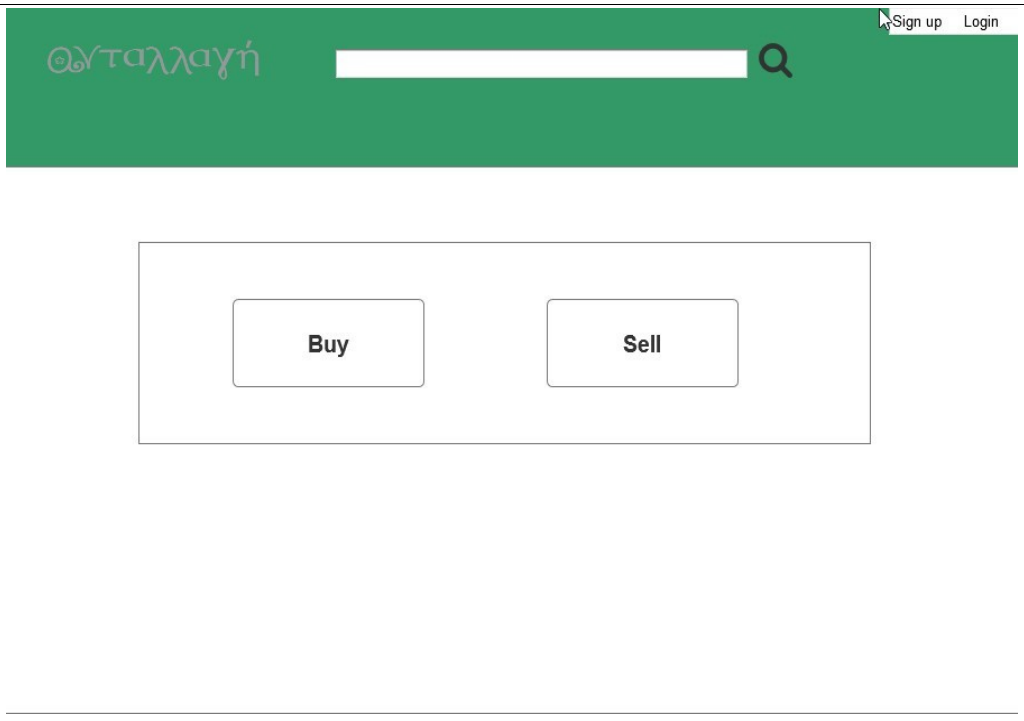


Illustration 27: Landing page wireframe

When the visitor has chosen the 'Buy' or 'Sell' options, we thought that it is very important to make it easy for him to switch to the other section at any moment without needing to search the respective button in a menu. If the user needs to search for a basic functionality, like switching buy/sell section, it can make the user experience low and that's why we wanted to place the switch button somewhere visible. For the service of that purpose we added a switch option on the header, under our logotype. See image below.

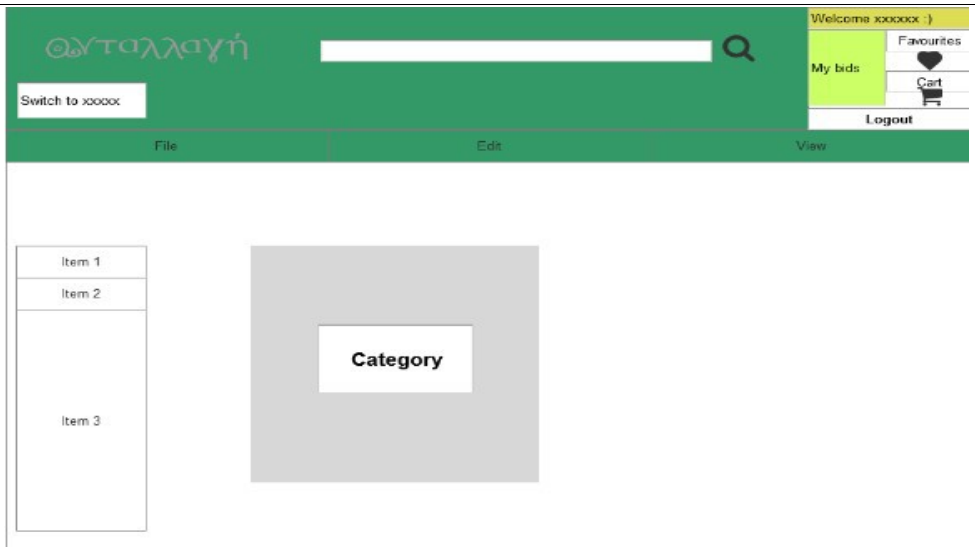


Illustration 28: Wireframe with user-menu and switch button.

Example 2: User-menu

The second example, is the user menu that appears when logged in. We created this little box on the top right corner of the header that will display some user information that we wanted to be easily accessible – see image above. Our point here is to prevent confusion and make it easy for the user to navigate through our application and make the features that are mostly used easy to find.

Colors

We are creating an application that promotes the concept of selling used objects for reusing them instead of throwing them away and creating waste. The color we decided to use is green because it's a color that represents nature and therefore, recycling. Green is also the most restful color for the eyes and has an overall positive impact when looking at it (Adobe Flash Enabled, n.d, para. 13- 16). One of the most important emotions that it emerges is safety and that is something we definitely want our visitors to feel.

Safety and positive emotions are something that every business would want their customers to feel but we believe that big emphasis should be given to these two concepts when creating and application that doesn't have a physical store that you can visit and especially when bank transactions or other payment methods are being used. The appearance of our application and the impression that it gives, is the image that we want to show about our company and how we want to be conceived by our visitors.

Materialize

For our CSS code, we have been using the Materialize framework. Materialize was created by Google Material Design. You can either download it, link to it with a CDN link or install it like a package in node.js (Materialize, n.d -a).

Originally we were using the CDN link but after a while, we decided to download the file. We did that because in many cases we needed to overwrite some code or create new classes and it was much easier having the file locally.

WHY did we choose Materialize

The reasons why we chose Materialize are many. First of all and most importantly, Materialize has its main focus on user-friendly design. Materialize accomplishes the user friendliness with a clear and simple design that is combined with a 'fresh' look. Apart from that, it also gives its elements a natural look by adding shadow to them. Except from the natural look, the shadow also makes an element stand out in a very subtle way. Simplicity is combined with this concept because you can draw the attention of the visitor without needing to use text explanation, bold or other intense methods to make something stand out. One more thing that we liked about Materialize were the animations.

We have not implemented any animation yet in our application, but we intend to, in the near future. Last but not least, the grid system that makes it easy to build a responsive design (Materialize, n.d. -b). Responsive design is very important as we would want our application to be accessible in all kinds of devices.

HOW does Materialize backup our business/ idea

Recycling is a concept that is not used so much in Greece. The mentality of throwing something away instead of selling it, still exists. The reason for this, may be that many people think that it's complicated, hard and requires time to find somebody that is interesting in buying their old stuff, so they just throw them away or keep them in the closet. We wanted to promote that it's easy, simple and fun to use our website and be a part of our concept. **Easy** because of the user-friendliness, **simple** because of our uncomplicated design and **fun** because of the animations that gives a sense of lightness.

HOW we adapted Materialize to our needs

The Materialize CSS code offers three 'screen class-prefix" classes: 's', 'm' and 'l'. These three scales were not enough for the responsive design of our application. We discovered that, when we were creating the categories page. We wanted 4 categories displaying on each row for large screens, 3 categories for medium screens, 2 for small screens, and 1 for mobile screens (Illustration x). In order to accomplish that we created the 'xs' class.

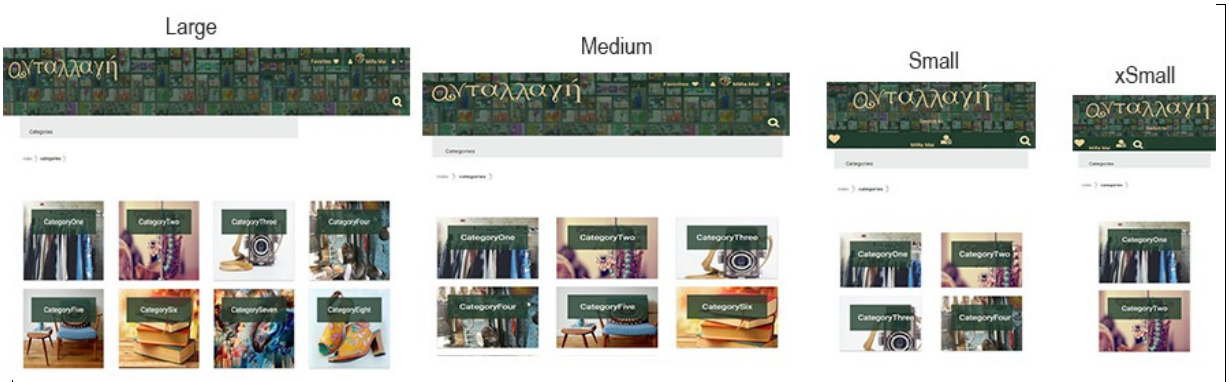


Illustration 29: Categories page in all scales.

Creating the xs class was easy. All the prefix classes in the materialize css file (l, m and s) have exactly the same properties and values, but are valid for different screen sizes. That means that only the Media Queries for each and one of them is different.

We first created the xs class and gave it all the properties that the rest of the classes have, and then we just added a media Media Queries on the s class.

```
@media only screen and (min-width: 450px)
```

What this code says now, is that the s class is valid on a screen that is wider than 450px. For smaller screens we can now use the xs class. This simple solution gave even more flexibility to our layout and we are using the new class in several pages of our application.

Font awesome

In our application, we are using several icons as the 'heart icon', the 'user icon', the 'search icon' and more. At first, we used Materialize icons, by linking to their website from our header section in the index.html file. We realized that, that solution would not work in the long term because we are creating an application that will have to handle a huge amount of data and we are always looking for solutions that will not have any chance to burden our site speed. Keeping that in mind, we thought that it would be better to find an icons software that is available for downloading and using it in our application folder. That way the users bandwidth will not get slowed down by needing to download the icons and we will not depend on a third party server to show the icons on our website.

Font awesome is a free source software that gave us that option. Another reason why we chose it is the huge collection of icons that it offers and the fact that you can customize them according to your preferences with simple CSS coding. This gave us the possibility to adapt the icons according to the needs of our application (Gandy, n.d).

On the screenshots below you can see the icons that we have used in our application. We have adapted the size and the colors. We used a light beige nuance to match with the colors of our application and when the heart icon is clicked we turned it into red.

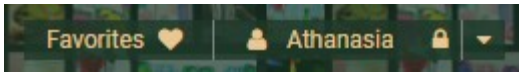


Illustration 31: User- menu.

Illustration 30: Heart icon when clicked.

Illustration 32: Search button.

Security

Security is one of the most important things when creating an e-commerce application if you want a trustworthy and reliable company that will have potentials to grow. Our website represents our brand, and without any physical store, it's the only connection we will have with our visitors; therefore, the impression that it gives is vital for the growth of our company. Returning customers are based on trust and most people wont buy or sell with anything but what they conceive to be a 'reputable, secure businesses'. (Laverty, n.d., para.6).

Firebase authentication[12]

WHAT is Firebase

Firebase is a platform owned by Google and provides a series of services like cloud messaging, authentication, real-time database, storage , hosting, crash reporting and much more (Google Developers, n.d.). Firebase have made it easy to use back-end services by creating it's own methods that you can use in your Javascript code. That makes it very practical and useful especially for front-end developers because the amount of back-end code that needs to be written is minimum.

Among all the services that Firebase offers, we decided to use Firebase Authentication, which is a service that provides a series of functions for signing up, using authentication from another platform, for example Facebook, Twitter, email etc (Google Developers, 2018-a).

[12]Section author: Mira Aeridou.

WHY did we choose Firebase authentication & HOW does it serve us

What is authentication? “Authentication is the process of determining whether someone or something is, in fact, who or what it is declared to be.” (Rouse, 2015).

One of the basic functionalities of our application is creating an account. This is necessary in order to be able to create an announcement about an item that you want to sell, or if you want to buy something. For that purpose, sensitive user information like address, email and payment info are needed. This information has to be protected, so one reason why we chose Firebase Authentication is because authentication ensures that the right person is accessing their account.

Another reason why we chose it, is because we are building a platform where we want our visitors to have as many options as possible available for the sign-up. Providing sign-up methods with authentication from other applications makes the sign-up procedure very simple and effortless because the only thing needed is a password or verification. This improves our user-friendliness because all the procedure sums down to the 1/5 of the time that you would need if performing a normal sign-up.

Lastly, we chose Firebase authentication because we needed to get our application up and running quickly so that we could focus on developing other parts of the application that we had prioritized. When we started with our project, we understood that we had to experiment with some software options, in order to gain the experience needed. As new developers, it's sometimes necessary to try out programs and software and see over time, if it's something that serves the needs of our project. The possibility of changing our authentication provider exists, and that is something that we plan on investigating in, in the near future. The two factors that will be crucial upon that decision, is the safety of our users accounts and the ability to support a big amount of users.

The 'onAuthStateChanged' method

The 'onAuthStateChanged' method is provided by Firebase and it's an authentication state observer. That means that it gives us the user information from the user that is currently logged in (Google Developers, 2018-b). It's very useful and necessary for our application not only to have access to the users data and be able to perform our basic functionalities like selling and buying, but also for displaying the user menu and other pages that are adapted to the user's preferences. Below I'm going to explain how we connected Firebase to other functionalities of our website, using the onAuthStateChanged method.

Example 1: How we made the users menu display

Whenever a user is logged in, we will want the user menu to display. To make that happen, we needed to connect 'onAuthStateChanged' method to our user menu.

We first added a statement inside of the onAuthStateChange method that says that if the 'user' is true, 'userLoggedIn' will also be true, otherwise it will be false. The variable 'user', is an object with all the user data that we got from the observer (Illustration x).

```
315    firebase.auth().onAuthStateChanged(user => {
316        this.userLoggedIn = user ? true : false;
```

Illustration 33: 'userLoggedIn' inside of 'onAuthStateChanged' method.

Then we bound the userLoggedIn property to our user menu in shell.html. See image below. That means that it will only show when the user is logged in.

```
100    <div class="col s12" if.bind="userLoggedIn">
```

Illustration 34: 'userLoggedIn' binding in shell.html

The user menu has to be displayed in order for the user to have access to their account and be able to edit it (change user-name, image, card inf etc). The onAuthStateChanged method is a secure way of accessing the users data, as it will only give access to it, when you are logged in.

Example 2: How we connected Firebase Authentication to Pouch DB

The database that we chose for our application is Pouch DB (See chapter 'Backend'). When somebody signs-up with Email or Facebook authentication the user is first created in Firebase authentication users database, and then we add this user to our database, Pouch DB. This is necessary in order to create our users database and keep all the users info there. Imagine that each user has a file with all their data that is reachable only when they log in. The file is their account. They can store new things there, add favorite items, bank account info etc, and they can also modify the existing info in there. Connecting Firebase authentication to Pouch DB has several steps that are illustrated in the image below. A more detailed description of each step follows.

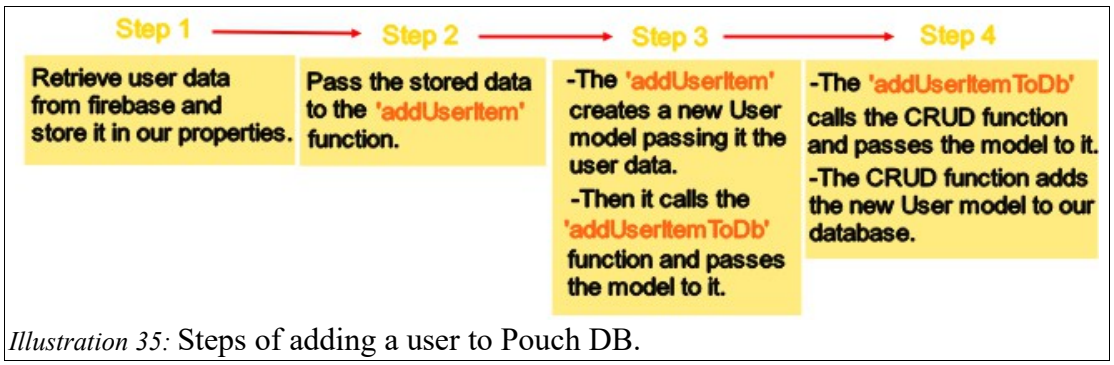


Illustration 35: Steps of adding a user to Pouch DB.

- **The first step** is to retrieve the user data from Firebase and store it in the properties that we have declared in the constructor of the 'Signup' class (Illustration 4, lines 12-15). We are doing that in the 'checkIfUserExistsAndAdd' function that is being called every time somebody tries to sign-up (Illustration 5, lines 350, 351 & 353). The data of the user is stored in those properties which are "carrying" the email, the image and the username of the user that is currently logged in.

```
8   export class Signup {
9
10      constructor() {
11
12          this.item_name = '';
13          this.item_email = '';
14          this.item_password = '';
15          this.item_img = '';
16
17      }
```

Illustration 36: The constructor in signup.js

```
348     checkIfUserExistsAndAdd() {
349
350         this.item_email = firebase.auth().currentUser.email;
351         this.item_img = firebase.auth().currentUser.photoURL;
352
353         this.item_name = firebase.auth().currentUser.displayName;
```

Illustration 37: The 'checkIfUserExistsAndAdd' in singup.js

In the above lines we are using the 'currentUser' property provided by Firebase (Google Developers, 2018-b). This property retrieves the users data when somebody is logged in.

Note: The 'currentUser property' is an alternative way to retrieve the data apart from the 'onAuthStateChanged'.The reason why we used it, is because the 'onAuthStateChanged' method was returning undefined inside of this function.

- **The second step** is to pass the stored data to the 'addUserItem' function. We call the function inside of 'checkIfUserExistsAndAdd', giving it an object with the user data as an argument. See image below.

```
364         this.addUserItem({
365             name: this.item_name,
366             email: this.item_email,
367             password: this.item_password,
368             img: this.item_img
369         });
```

Illustration 38: Calling 'addUserItem' in 'checkIfUserExstsAndAdd' in signup.js.

This step is preparing the data so that we can add it to PouchDB.

- **In the third step**, the 'addUserItem' function creates an instance of the user model, giving it the assigned properties (see illustration 39, line 461). In the next line it's calling the 'addUserItemToDb' function, passing it the instance of the user model that we just created. See image below.

```
460   addUserItem(addeduser) {
461       let user = new User(addeduser.name, addeduser.email, addeduser.password, addeduser.img);
462       this.addUserItemToDb(user)
```

Illustration 39: The 'addUserItem' in signup.js

- **In the fourth step**, we are passing the user data to the CRUD function that is responsible for adding the user to the Pouch DB.

```
471       addUserItemToDb(addeduser) {
472           this.UserDatabase.addUserItemToDb(addeduser);
473       };
```

Illustration 40: The 'addUserItemToDb' in signup.js

The CRUD function resides in a different file and that's why the 'addUserToDb' function is calling the instance of the user database and then the CRUD function called 'addUserItemToDb' -see image above.

WHY[13]: Division of steps in this way was done to improve the code.

- Firstly, it is logical to have model-creation and DB-operations in two different functions, since they do two different things, that each may later require specific updates and changes, where changing the other then wont be necessary.
- Secondly, and most importantly because of error recovery/checks and the MVVM model we use which is a separation of concerns and finding, preventing and correcting errors should be done for each operation separately to improve efficiency.

[13]Paragraph author: Christina Bögh.

AuthorizeStep[14]

What are pipelines

A pipeline is, as the name suggests, a 'flow' from start to finish, through some predetermined path. Much like an oil pipeline, in Aurelia routing events run through steps and you can add a pipeline in between those steps.

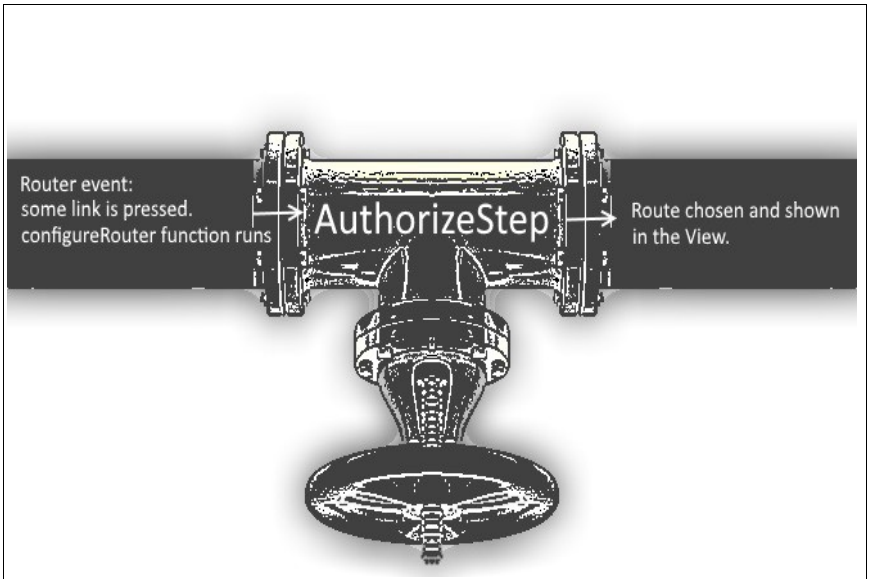


Illustration 41: Pipeline.

As you can see in the illustration above, the AuthorizeStep pipeline is a middle step that is executed in between the user presses a link and the new page is displayed. The process is in 5 steps:

1. User clicks link →
2. Aurelia app looks at configureRouter() function →
3. executes AuthorizeStep pipeline →
4. pipeline does several checks on the requested route →
5. either: continue to route or be redirected.

[14]Section author: Christina Bögh.

Any kind of pipeline can be added, even several, but in the case of our application, the pipeline is a way of improving the security of our page.

WHY do we need a pipeline?

Some parts of the website are only for logged-in users, and some parts are only for admins. In order to control content and secure admin- or user-specific data, the pipeline effectively controls access.

The AuthorizeStep pipeline pauses the natural flow of the router for a short 'check' before continuing normally. In our website, it is a filter, to stop unwanted routing and redirect users to alternate paths if they are **not** authorized by the pipeline.

HOW do we do it?

Pipelines are like an ordered line of **slots** that can be executed before routing and can be added to an Aurelia router. There are 4 available slots and any pipeline must be added to one of those slots. To add a pipeline to a slot, simply add a pipeline **step**. Aurelia comes with 4 pre-made step functions you can use, but you can also simply make your own instead, which is what we did on our website.

Below you see the config option that is found in our main Router. The 'addPipelineStep' function is the custom step, and the first argument 'authorize' is the slot. The class we want to use for our custom step is the second argument, 'AuthorizeStep'.

```
config.addPipelineStep('authorize', AuthorizeStep);
```

Illustration 42: config pipeline in router.

Example: How a pipeline step works.

The pipeline step in itself is an **object** created from the AuthorizeStep class that contains a function called

```
run(navigationInstruction, next)
```

Inside of this function, the logic behind the pipeline is added. Our authorize pipeline has three different 'checks' that are done prior to any routing.; a test for 'sell', 'admin' and 'favorites'. See image:

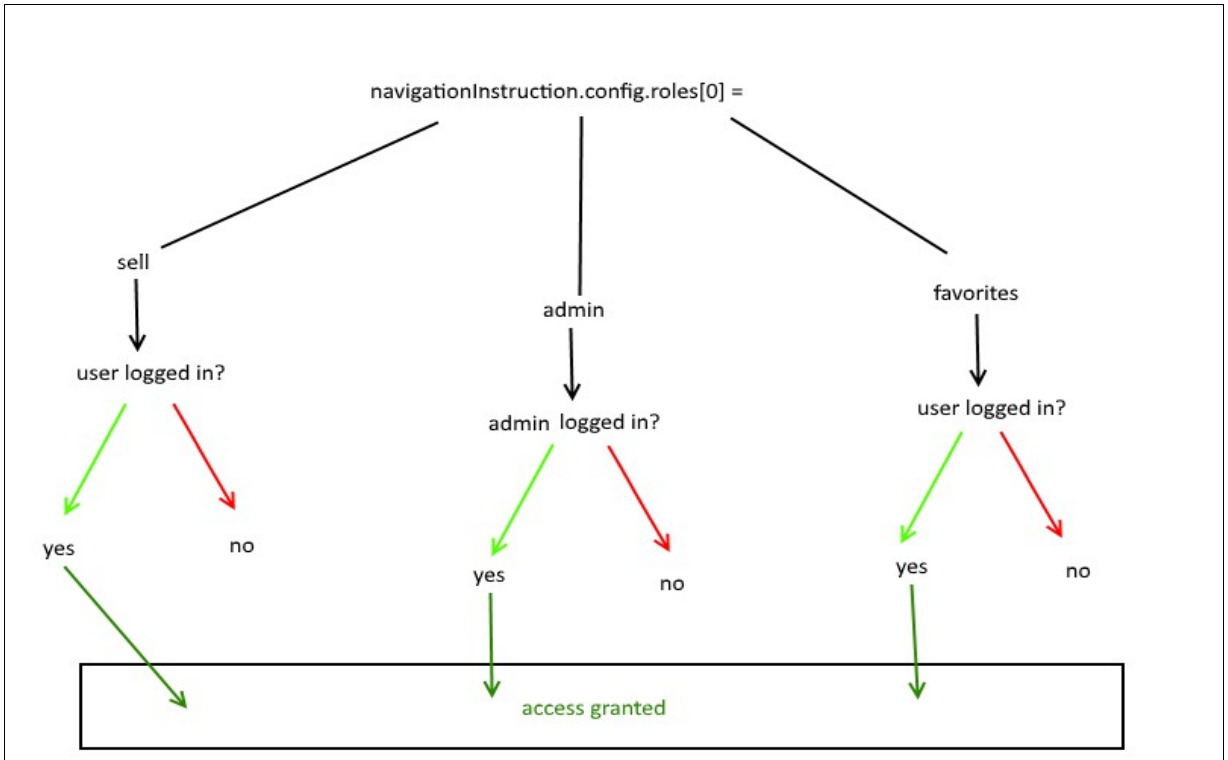


Illustration 43: Pipeline checks.

The function run() is passed two arguments. The first one ' navigationInstruction' is the one that tells the class everything about the navigation: it is the entire Router navigation object, with all the routes, their properties and most importantly; which is the current route that we want to access.

How does it work? Well, first, a simple if statement checks if we are trying to access the admin route. If yes, the user database is consulted and asked if the currently logged-in user is an admin. If its false it directs to the index route. If its true, the pipeline does nothing (and just lets the Router show the page). See image below:

```
class AuthorizeStep { // is is to redirect the sell to login if no user is logged in.

  constructor() { this.UserDatabase = new UserDatabase();}

  get adminOrUser() {return this.UserDatabase.admin;}

  run(navigationInstruction, next) {


    // start test admin
    if (navigationInstruction.config.roles[0] === 'admin') {
      try{
          let test = this.adminOrUser;
          if(test === false){ // not logged in as admin
            console.log('test was false: you are redirected');
            return next.cancel(new Redirect('index'));
```

Illustration 44: admin check.

Backend[15]

PouchDB

WHAT is Pouch DB

PouchDB is a database that was inspired by Apache Couch DB. It runs in the browser and in node.js, and stores the data locally so that the user can use the application even when offline. When the user goes online it can synchronize the data with a NoSQL server, that could for example be Couch DB (PouchDB, n.d.-a). It is open source, and to implement it, you use Javascript methods, that are provided by the Pouch DB documentation (PouchDB, n.d.-b).

WHY we chose Pouch DB and HOW it serves our project

There are several reasons why we chose Pouch DB for our application. The fact that it is an in-browser database that can be synchronized to many servers will make the features of our application available even when offline. How is this going to benefit our application? When a user has no internet or a slow connection, the in-browser database, Pouch DB, will be used. This way, latency is prevented because what an in-browser database does, is using the data that has been locally cached until the network connection comes back, and then it synchronizes the data with the server (Slater, 2014, para. 7). We will adapt this to the user environment and needs, so that the Pouch DB will store data that is necessary for the user when offline. This is very important for us, as we're aiming for the best possible user experience.

For the moment we are only using Pouch DB without synchronizing it to any server. Choosing and implementing the suitable server for our application will be a task of LIA 2.

[15]Mira Aeridou.

Something that is not less important is that Pouch DB is compatible with all popular browsers like Chrome, Firefox, Opera, Safari and IE (PouchDb, n.d. -a) . This is something vital for our application as we would not want it to be restricted from people because of the browser they use.

A big advantage of Pouch DB that needs to be mentioned is the good documentation and instructions that it provides and that makes it very easy to implement. This benefits us because it makes our job a lot easier and our working hours less.

HOW we implemented Pouch DB in our application

First, we created our database by calling a new instance of the Pouch DB object with the name we wanted to give to the database, and the adapter we wanted to use. Specifically, the name for our users database is 'users-db' and the adapter, 'websql'. See image below.

```
6       this.userDB = new PouchDB('users-db', {adapter: 'websql'});
```

Illustration 45: Create a Pouch DB database.

To be able to view our in-browser database, we need to go to the 'Developer tools' in our browser, and then choose 'Application' → 'Web SQL'. In the screenshot below, I have chosen the users database and on the right of the image, the database's object is displaying. See image below.

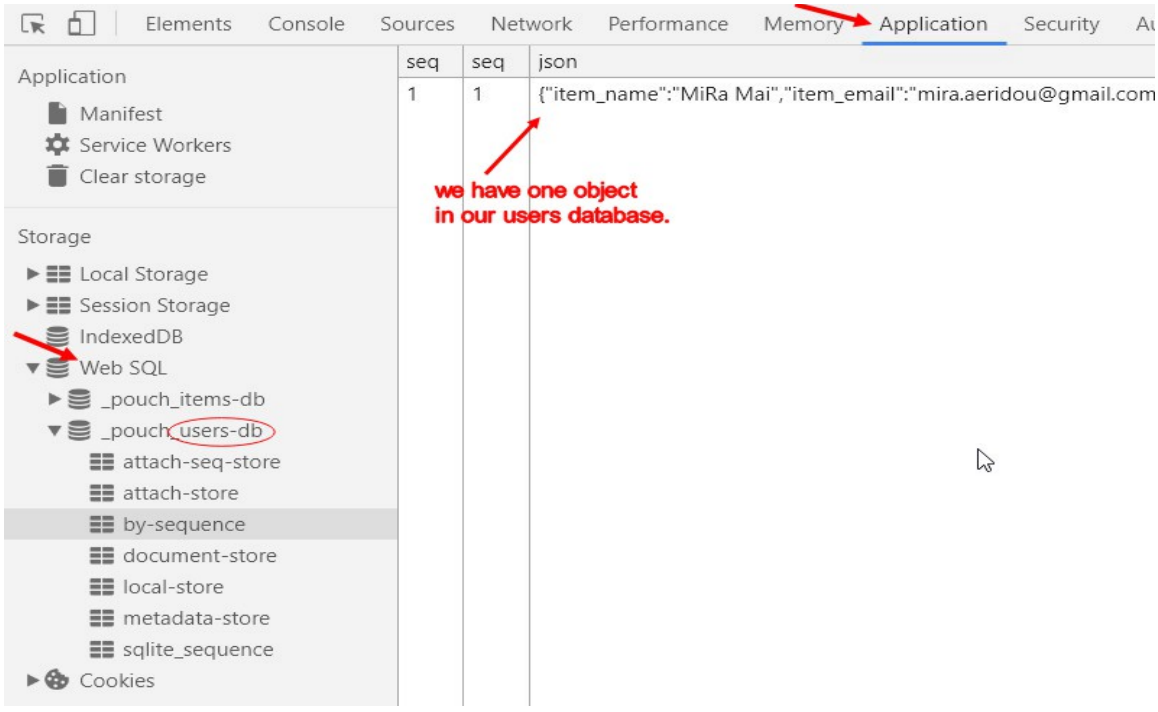


Illustration 46: Developer tools in our browser.

Our next steps was to create a 'model' and then the CRUD operations. Below you see an illustration of these steps and a more detailed explanation is following.

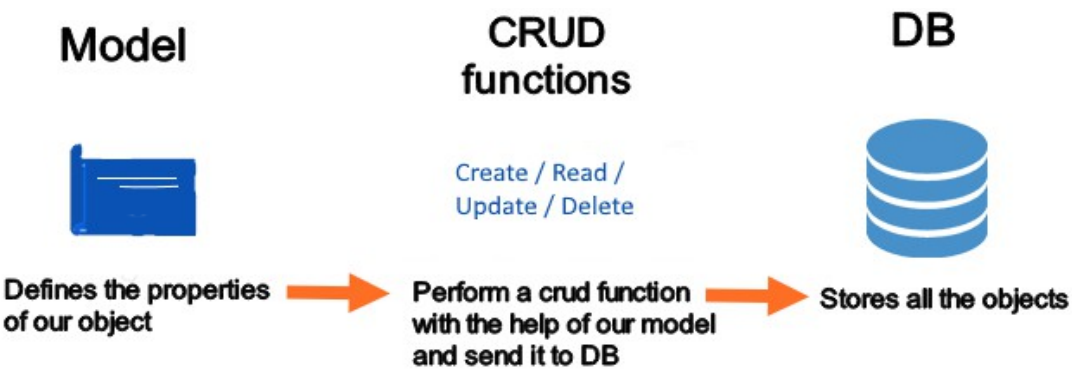


Illustration 47: Pouch DB implementation.

Model

WHAT is a model

The model is a code that defines the properties and the basic structure of the objects that we want to store in our database. When a new object is being created and added to our database we need to do it calling the model class. An example is when a new user is being created.

The users model looks like this:

```
export class User {

  constructor(item_name, item_email, item_password, item_img, admin) {

    this.item_name = item_name;
    this.item_email = item_email;
    this.item_password = item_password;
    this.item_img = item_img;
    this.item_favorites = [];
    this.admin = admin;

  }
}
```

Illustration 48: The user model in user.js

When we want to create a new user we must call the instance of the User class and pass the four first properties to it.

```
new User (name, email, password, image);
```

HOW does our application benefit from the model and WHY do we need it

A model is necessary because we want all our objects in our database to have the same structure and properties so that we can access the given values by calling the property name. It's a baseline that all of them need to follow. This is very important for our e-commerce application because it helps us in various features.

Other benefits with using a model is that we can require and filter the model properties. For example we can require the password property. That means that an account will not be created unless a password has been given. For the moment we don't use require because Firebase authentication sign-up does that for us. Filter is being used to ensure that the information that the user gives, meets certain requirements. This can be used to avoid “ugly” language or to limit the characters in the password. We are not yet filtering our model properties, but we intend to in the future. Two examples of how the model is useful in our application are following:

Example 1: The first example is when we're displaying the users name in the users menu. We are first calling the class name of the model, and then the value of the property that we want to display (Image below: line 132).

```
User.item_name
```

```
128     <div class="divTableCell">
129         <span>
130             <i class="fa fa-user fa-1x" aria-hidden="true" style="color: #e9c893; padding: 5px;"></i>
131         <img class="circular" src=${User.item_img} alt=""/>
132         ${User.item_name}
133     </span>
134     </div>
```

Illustration 49: Displaying user-menu.

This is a simple way to access the data of the user that is currently logged in by calling the model properties.

Example 2: The second example is when we are displaying the favorites page that has to be customized to the user's preferences. In order to do that, we need all our users to have the favorites array as a property. In this array we add the users favorite items, and we display them in the favorites page by calling the favorite list property of the user that is currently logged in. See the two images below.

```
<div class="col s12 center-align" repeat.for="item of favorites" id="favlist">
```

Illustration 50: Displaying the favorites list in favorites view-model.

```
this.favorites = this.allUsers[i].item_favorites;
```

Illustration 51: The favorites list in the favorites model.

CRUD operations

WHAT are the CRUD operations

The word CRUD stands for create, read, update and delete. These are the operations needed in order to handle the data in our database. Pouch DB documentation provides us with methods to be able to perform and implement these operations into our application (source).

The methods that we are using in our application is the 'get()' / 'allDocs()' for reading one or all the documents that exists in the database, the 'put()' for creating or updating a document, and 'remove()' for deleting a document (PouchDb, n.d -b).

HOW does our application benefit from the CRUD operations and WHY do we need them

The CRUD operations are absolutely essential for our application because we are creating an e-commerce application where our users will be creating their account and then creating an announce about a product that they want to sell. Imagine these two functionalities in levels. See image below :

- In level one, we create an account by creating a new object in the users database using the CRUD method 'put'.
- In level two we create an announcement about a product that we want to sell. That means that you're adding a new object in the sell-items database using the CRUD method 'put'.
- There is one more level, the third, where you can edit/ update your announce, delete it and add a product to your favorites. In each of these functions we are using the respective CRUD methods (PouchDb, n.d -c).

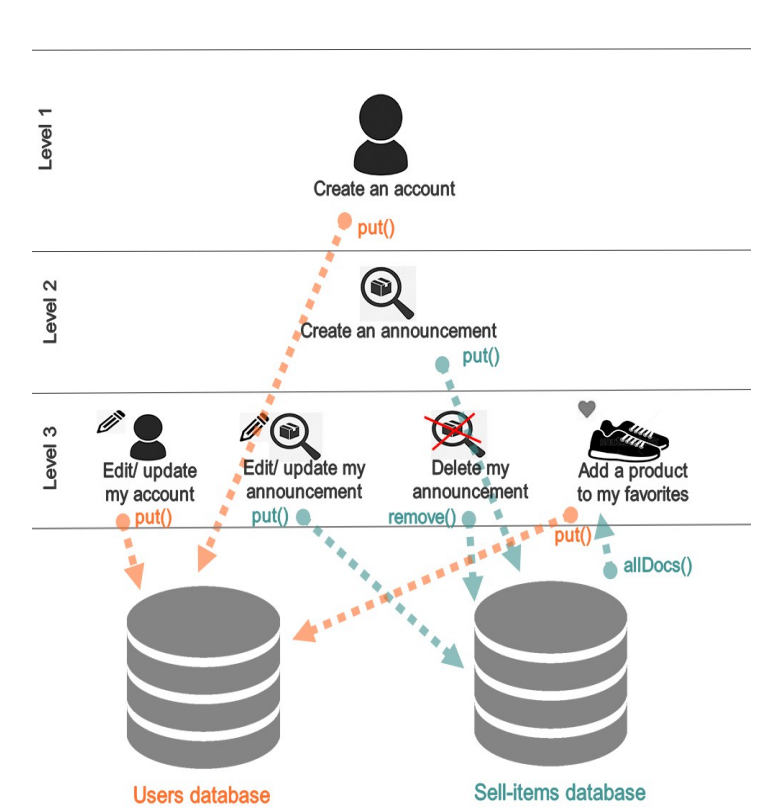


Illustration 52: database connections.

Now it's easy to understand that all the functionalities of our application are going to be related to the CRUD operations or at least depend on them, and that's the reason why they are essential for our application.[16]

Lastly, I want to mention that Pouch DB provides us with an asynchronous API. In other words, we are using promises to implement the CRUD methods. This benefits our application a lot because it prevents latency and spaghetti code. It also helps us organize our code, something that is vital when creating such big applications like ours.

[16]Note: This explanation and the illustration below contains just examples of few functionalities that we are using in our application.

Conclusion[17]

Our Antallagi e-commerce website fulfills the Greek demand for recycling used things and thereby fills a niche. At this point, the application allows users to log in and register. As of now products can be added, but not yet bought.

[17]Chapter authors: Mira Aeridou. Christina Bögh

References

Adobe Flash Enabled. (n.d.). Color Meaning. Retrieved from http://www.color-wheel-pro.com/color-meaning.html

Aurelia. (n.d.-a). Technical Benefits. Retrieved from http://aurelia.io/docs/overview/technical-benefits#web-component-standards

Aurelia. (n.d.-b). EventAggregator. Retrieved from http://aurelia.io/docs/api/event-aggregator/class/EventAggregator

BBC. (2017, June 19). Greece profile. Retrieved from http://www.bbc.com/news/world-europe-17373216

Brammer, L. R. (1998). ECOFEMINISM, THE ENVIRONMENT, AND SOCIAL MOVEMENTS. Retrieved from http://homepages.gac.edu/~lbrammer/Ecofeminism.html

Encyclopedia.com. (n.d.). Green Movement - Dictionary definition of Green Movement.

Gandy, D. (n.d.). Font Awesome, the iconic font and CSS toolkit. Retrieved from http://fontawesome.io/

Harrison, D. (2016, February 27). När infördes du-reformen? Retrieved from https://www.svd.se/nar-infordes-du-reformen

Google Developers. (n.d.). Documentation. Retrieved from https://firebase.google.com/docs/

Google Developers. (2018, January 20-a). Firebase Authentication. Retrieved from https://firebase.google.com/docs/auth/

Google Developers. (2018, January 20-b). Manage Users in Firebase. Retrieved from https://firebase.google.com/docs/auth/web/manage-users#get_the_currently_signed_in_user

Greek Recycling Agency. (2018). The "alternative" recycling. Retrieved from https://www.eoan.gr/el/content/164/i-alli-anakuklosi

Greenfence. (n.d.). Home. Retrieved from http://greenfence.gr/en/

HTML Living Standard. (2018, January 17). 4.13 Custom elements. Retrieved from https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements

Introducing Aurelia. (2015, January 26). Retrieved from http://blog.aurelia.io/2015/01/26/introducing-aurelia/

Khanacademy. (2014). Maslow's hierarchy of needs [Video file]. Retrieved from https://www.khanacademy.org/test-prep/mcat/behavior/theories-personality/v/maslow-hierarchy-of-needs

Materialize. (n.d. -a). Getting Started. Retrieved from http://materializecss.com/getting-started.html

Materialize. (n.d. -b). Grid - Materialize. Retrieved from http://materializecss.com/grid.html

MDN, & Brettz9. (2017, November 7). The HTML template. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template

PouchDb. (n.d. -a). About PouchDB. Retrieved from https://pouchdb.com/learn.html

PouchDB. (n.d. -b). Introduction to PouchDB. Retrieved from https://pouchdb.com/guides/

PouchDB. (n.d. -c). Updating and deleting documents. Retrieved from https://pouchdb.com/guides/updating-deleting.html

PouchDB. (n.d. -d). Asynchronous code. Retrieved from https://pouchdb.com/guides/async-code.html

Recycom. (n.d.). Recycling. Retrieved from http://www.recycom.gr/cms/?page_id=23

Rouse, M. (2015, February). What is authentication? - Definition from WhatIs.com. Retrieved from http://searchsecurity.techtarget.com/definition/authentication

Slater, N. (2014, August 21). An Introduction To PouchDB. Retrieved from https://www.engineyard.com/blog/an-introduction-to-pouchdb

Webcomponents.org. (n.d.). Introduction - What are web components? Retrieved from https://www.webcomponents.org/introduction