The Agile Way to Architect Web Apps

Doguhan Uluca
Technical Fellow
Excella
Doguhan Uluca
@duluca  linkedin.com/in/duluca

Excella
What is a Line-of-Business App?

"set of critical computer applications perceived as vital to running an enterprise"

• What do we develop?
• Why/when/where do we develop it?
• Who develops it?
• How do we develop it?
  ▪ Architecture
  ▪ API Design
  ▪ Web App Design
  ▪ Imperative vs Reactive
  ▪ Router-first architecture
What do we develop?
When/where/why do we develop?
When/where/why do we develop?

**Personal**
1. On your own time
2. Passion-driven
3. Educational
4. Likely under engineered

**Enterprise**
1. On company time
2. Requirement-driven
3. Likely over engineered
Personal development

Apps can grow but as industry apps can be under-utilized.

Small exps.

Under engineered.

Small exps.

Trouble begins.

Costly re-engineering.
Enterprise development
80% of useful apps

Personal

Line-of-Business Apps

Enterprise
The Enterprise Niche

Personal

Line-of-Business Apps

Enterprise

extremely complicated or large LOB apps, tools or products
1. When we start coding, we really never know how much or how little our app will be used.

2. Our apps can quickly become a LOB app.

3. LOB apps are really important to businesses.
How should we develop?

- Ideal scenario
  - Maintain optimal engineering overhead

- Size vs. features
1. Must deal with demands of iterative and incremental delivery
2. Must achieve a constant forward flow of features
3. Must do all the cloud things
4. Be scalable
5. Serve dozens of screen and input types
6. Must do usability, accessibility
7. Can't have more than 8 people on the team
8. Must do backlog grooming

However...
Who develops it?

1. CS grads
2. Career switchers
3. Reluctant companies
4. Foot draggers
5. Lone wolves
6. Team players
7. Learners
8. Stack-Overflows
How?

The right architecture, tools and patterns/practices.

- Size of our app
- Reason we are developing the app
- Skill level of developers
- Iterative and incremental delivery
- Constant forward flow of features
- All the cloud things
Fundamental Architecture & Engineering Fundamentals
High Level Goals of Architecture

- High Cohesion
- Low coupling
N-tier Architecture

User ➔ Web Server

Front-End
Angular Web Application

OpenWeatherMap API

API Server
Optional Back-End

Database
Optional data store
SOLID

DRY
Modern Web Architecture

- High cohesion
- Low coupling
- Asynchronous
- Non-blocking I/O
- Multi-threaded
- Parallel execution
- High availability
- Self-healing
- Auto-scaling
- RTO and RPO expressed in minutes/seconds
STATELESS

COMPOSABLE

REACTIVE
- API DESIGN
- WEB APP ARCHITECTURE
API design with REST

- REST is robust, but inflexible
- API documentation must be enabled manually
  - i.e. Swagger
- Must optimize every API call
  - Amount of data transferred over the wire matters, a lot
app.get('/users', function (req, res) {
    // return list of users
})

app.get('/user/{id}', function (req, res) {
    // return a single user with id
})

app.put('/user', function (req, res) {
    // update a user
})
**LemonMart 1.0**

[Base URL: localhost:3000/v1]

LemonMart API

### Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
</tr>
</tbody>
</table>

### default

<table>
<thead>
<tr>
<th>Method</th>
<th>Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/users</td>
</tr>
<tr>
<td>GET</td>
<td>/user/{id}</td>
</tr>
<tr>
<td>PUT</td>
<td>/user/{id}</td>
</tr>
</tbody>
</table>
PUT /user/{id}

**Models**

**Role**  
*string*

*Enum:*

- clerk, cashier, manager

**Name**

- **first**  
  *string*
- **middle**  
  *string*
- **last**  
  *string*
API design with GraphQL

- GraphQL makes it possible to iteratively implement
- Documentation is baked in
- The consumer decides how data to pull
- Inherently stateless
- Inherently composable
- Lends itself to functional reactive programming
type Query {
  content(
    source: Audience!
    type: [ContentType]!
    id: Int
    zoneId: Int
    count: Int = 30
    topNews: Boolean = false
    dayCount: Int = 365
    htmlContent: Boolean = false
  ): [Article]
  ...
}
GraphQL Schema
export const resolvers = <IResolvers>{
  Query: {
    content: async (obj: any, args: IContentQueryParams) => {
      return await getContent(args)
    },
    ...
  },
  Article: {
    type: (obj: any) => {
      return EnumValues.getNameFromValue(ContentType, obj.type)
    },
    authors: (obj: any) => {
      return obj.authors ? wrapAsArray(obj.authors.author) : []
    },
    ...
  },
  ...
}

GraphQL Metrics

- 11 mobile apps in 14 languages
- 9 months
- Kanban
- Multiple redesigns
- One major technology shift
- On time, on budget
- Deployed on serverless infrastructure
- Apps work fast in rural areas
- Two developers
Web App Architecture

- Exposed to a lot of change by
  - End-users
  - APIs
- Must be able to evolve the architecture without rewrites
  - Isolation of components is key
  - Custom elements will help
Web App Architecture

View
- View
- Model & Controller
- Service & Model

API
Local Storage

Data

HTML/Template

JavaScript/Typescript
**Angular**

- app.ts
- rootRouter
- services
- filters
- directives
- /a: default
- /master
- /detail
- /b/
- /c
- childRouter
- /d
- /e
- /f
React

App.js

react-router

react-redux

Component Legend
- Presentational
- Container
- Provider
- Router
Imperative vs Reactive
Must define every step of code execution
Easy to introduce state
Easy to forget a step
Difficult to test
Imperative

- OOP
- Functional
Reactive

- Programming with asynchronous data streams
- Functional
- Composable
- Stateless
What is a Data Stream?

- Event-driven
- Pub-Sub
- Asynchronous data stream
Event-Driven Data

Event Source

user clicks

Event Handler

window.alert('Are you sure?')

onClick='confirmDelete()'
Publish-Subscribe Pattern

- updated data
- updateCache()
- fetchDetails()
- showToastMessage()
Pub/Sub or Event-based use case

todos

What needs to be done?

- [ ] get ready for talk
- [ ] give talk
- [x] enjoy rest of the day

2 items left

All  Active  Completed  Clear completed
Asynchronous Data Stream

Event Source: mouse clicks

- throttle(250ms)
- map(list.length)
- filter(x >= 2)

<li *ngFor="let i in list | async">
  window.alert('Are you sure?')
</li>
Reactive or Data Stream use case

- DigitalOcean
- Erik Meijer
- Avatar.ai
1. Grab a hose
2. Spray water into the heater
3. Turn on the faucet for hot water
4. Send a text to the utility company
5. Don't forget to undo your steps, when done
1. Turn on/off the faucet for hot water
```javascript
Reactive

products$ = this.http.get<Product[]>(this.productsUrl)
  .pipe(
    tap(data => console.log('getProducts: ', JSON.stringify(data))),
    shareReplay(),
    catchError(this.handleError)
  )
```
Reactive

```typescript
productsWithCategory$ = combineLatest(
  this.products$,
  this.productCategoryService.productCategories$
).pipe(
  map(([products, categories]) =>
    products.map(
      p =>
        ({
          ...p,
          category: categories.find(c => p.categoryId === c.id).name
        } as Product) // <-- note the type here!
      ),
    ),
  shareReplay()
)
```
Reactive

Thinking Reactively: Most Difficult | Mike Pearson
https://www.youtube.com/watch?v=-4cwkHNguXE

Data Composition with RxJS | Deborah Kurata
https://www.youtube.com/watch?v=Z76QlSpYcck
What is Router-First Architecture?

A way to

- **enforce** high-level thinking
- **ensure** consensus on features, *before* you start coding
- **plan** on your codebase/team to grow
- **little** engineering overhead
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
How to implement Router-First?

1. **Develop a roadmap and scope**
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
Let's build a weather app
Local Cast Weather

Your city, your forecast, right now.

Enter city or zip:

Current Weather

Bethesda, US

Sep 18th

Overcast

72°F

<table>
<thead>
<tr>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>80°F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Your city, your forecast, right now!

Enter city or zip

Bursa

 Specify country code like ‘Paris, US’

Current Weather

Bursa, TR  Wednesday
Dec 27th

36 °F
clear sky
Lemon Mart
Stakeholders
Site Map

Landing/Login Page

Manager

Inventory

Point of Sale (POS)

User Profile
1. Develop a Roadmap and Scope

- Get high-level architecture right
- Define the map before getting on the road
- Capture the vision concretely
- Bring tools in only when necessary
- Implement iteratively
- Drive for perfection only after the fundamentals are in place and agreed upon
- Document every artifact you create
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
First-Paint Matters, A Lot

- 53% of mobile users* abandon if load times > 3 secs
- Content is consumed mostly on mobile*
- 70% in the US
- 90% in China
- Hybrid client/server-side rendering is hard

*Source: Angular Team, Google Analytics, 2018
Define User Roles

- manager
- inventory
- pos
- unauthorized
Hello, Lemonite!

Login
Hello, Lemonite!

E-mail

Password
Minimum 8 characters

Login
Lazy Loading Tips

1. Eager-load resources in the background
2. Optimize your chunk sizes
3. Watch for amounts of assets and compression
4. Component level lazy loading will be possible
2. Design with Lazy-Loading in Mind

- First-paint matters a lot
- Lazy loading is low hanging fruit
- Requires user roles to be defined early on
- Very difficult implement lazy-loading after the fact
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
LemonMart

POS  Inventory  Manager
Manager's Dashboard

User Management  Receipt Look-up

Monthly Sales  Product Mix  Inventory Status
manager-home works!
inventory-home works!
Filter if children, otherwise display +

If checked, display +/- icons
LemonMart

Manager
Users
Receipts
Inventory
Stock Entry
Products
Categories
Gather feedback from users
Workout fundamental workflow and integration issues quickly
Concrete representation of scope
Sets the stage for multiple teams to work in tandem
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
Reactive Streams

Event Source

mouse clicks

throttle(250ms) → map(list.length) → filter(x >= 2)

<li *ngFor="let i in list | async"> window.alert('Are you sure?')
User
id
email
password
name {
  first
  middle
  last
}
picture
role [clerk, cashier, manager]
userStatus
RxJS/BehaviorSubject as Data Anchors

```javascript
1  readonly currentUser$ = new BehaviorSubject<IUser>(
2      this.getItem('user') ||
3          new User()
4  )
```
Using RxJS/BehaviorSubject

```html
  <div *ngIf="currentUser$ | async as currentUser">
    <div>{{currentUser.firstName}}</div>
    ...
  </div>
```
4. Be Stateless & Data-Driven

- Define observable "data anchors"
- Leverage RxJS features
- Write functional reactive code
- Don't store state in components
- Data across components will be kept in sync
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
## Users

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>Jane</td>
</tr>
<tr>
<td>✓</td>
<td>Jack</td>
</tr>
</tbody>
</table>

## User Details

Create

...
Ways to decouple

- @Input bindings
- @Output bindings
- Router orchestration
Let the router orchestrate

```javascript
1 {
2   path: 'users',
3   component: UserManagementComponent,
4   children: [
5     { path: '', component: UserTableComponent, outlet: 'master' },
6     {
7       path: 'user',
8       component: ViewUserComponent,
9       outlet: 'detail',
10      resolve: {
11        user: UserResolve,
12      },
13     },
14   ],
15   canActivate: [AuthGuard],
16   canActivateChild: [AuthGuard],
17   data: {
18     expectedRole: Role.Manager,
19   },
20 }
```
Review and update your user profile.

John S Smith
this.Role.Manager

E-mail
manager@test.com

Date of Birth
Feb 23, 1975
<table>
<thead>
<tr>
<th>Name</th>
<th>E-mail</th>
<th>Role</th>
<th>Status</th>
<th>View Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample text</td>
<td>Sample text</td>
<td>clerk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>E-mail</td>
<td>Role</td>
<td>Status</td>
<td>View Details</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>Sample text</td>
<td>Sample text</td>
<td>clerk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Items per page: 5
1 - 1 of 1
<table>
<thead>
<tr>
<th>Name</th>
<th>E-mail</th>
<th>Role</th>
<th>Status</th>
<th>View Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample text</td>
<td>Sample text</td>
<td>clerk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample text

Sample text

Sample text
clerk

E-mail

Sample text

Date of Birth

Feb 28, 2018
5. Be Decoupled

- Every component should be responsible of loading their own data
- Allows for composition of components
- Router enables URL driven composition
- Don't abuse the router
- Ok to design for a parent component to contain multiple hard-coded components
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
What's a User Control?

- i.e. Custom Date Input
- Highly coupled, convoluted, complicated code
- Using Angular features no one has ever heard of before
What is a component?

- i.e. Form with date inputs
- Code must be easy to read and understand
- Stick to Angular basics
- So code is stable and easy to maintain
Reusability

- User controls encapsulate complicated code
- User controls can be shipped in libraries
- User controls can be open sourced
- Composable components
- Save time and resources
6. User Controls vs Components

- Wire-framing allows to identify reusable elements early on
- Keep user interaction code separate from business logic
- Encapsulate domain specific behavior and share it
How to implement Router-First?

1. Develop a roadmap and scope
2. Design with lazy loading in mind
3. Implement a walking-skeleton navigation experience
4. Achieve a stateless, data-driven design
5. Enforce a decoupled component architecture
6. Differentiate between user controls and components
7. Maximize code reuse with ES6/TypeScript
• Don't Repeat Yourself
Object-Oriented Design

- Move behavior to classes
- i.e. Hydration, ToJSON, Calculated Properties
- Remain stateless and functional
• Documents shape of data
• Pass abstractions, not concretions
• Separate internal data shape from external shape
• Aim for flat data hierarchy
• Arrays and simple shapes for common objects are okay
• i.e. name object, domain-specific object
Use Enums

- No string literals in code
- No string literals in code
- No string literals in code
7. Use ES6 & TypeScript

Features

- Design around abstractions
- Use classes to reuse context specific behavior
- Use interfaces to communicate the shape of internal and external data
- Use enums instead of string literals
- Leverage Angular Validators and Pipes to reuse logic
Router-First Metrics

- 10 projects
- 50 developers
- 1 year of code written
- 115 of 1000 components written
- ~25% reduction in package dependencies, variations, and versions
- Feedback loop cycle reduced from 20 minutes to 1 minute
- ~25% reduction in software bugs reported
- ~25% increase in agile velocity
Think Router-First
Master the fundamentals
Be reactive
Keep it simple
What do we develop?
Why/when/where do we develop it?
Who develops it?
How do we develop it?

- Architecture
- API Design
- Web App Design
- Imperative vs Reactive
- Router-first architecture
Angular 6 for Enterprise-Ready Web Applications

Deliver production-ready and cloud-scale Angular web apps

By Doguhan Uluca

AngularForEnterprise.com
@duluca
linkedin.com/in/duluca
github.com/duluca