The full stack of an Angular app includes 3 tiers:

* Back end - Oracle & PL/SQL.
* Middle tier (Web service) - .Net & C#, uses the Windows Communication Foundation library (WCF).
* Front end (presentation tier) - Angular & TypeScript.

As an example, we will use DataChgSub objects (the "Data Change Subscriptions" screen in Admin):

**Back end:**

Usually, there are 4 procs:

* SELECT a list of entities.
* SELECT a single entity.
* SAVE (one proc for INSERT & UPDATE).
* DELETE.

For example:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

The above SAVE & DELETE procs demonstrate saving a single entity form dialog, which is the most popular.

If the data change dialog has multiple editable records, we use BULK SAVE & DELETE, sending a bunch of records.

The SAVE proc accepts them as a Json string:

A screenshot of a computer

Description automatically generated

The DELETE proc accepts the IDs of the entities as a comma-separated values string:

A white background with text

Description automatically generated

**Middle tier (Web service):**

Consists of 2 .Net solutions:

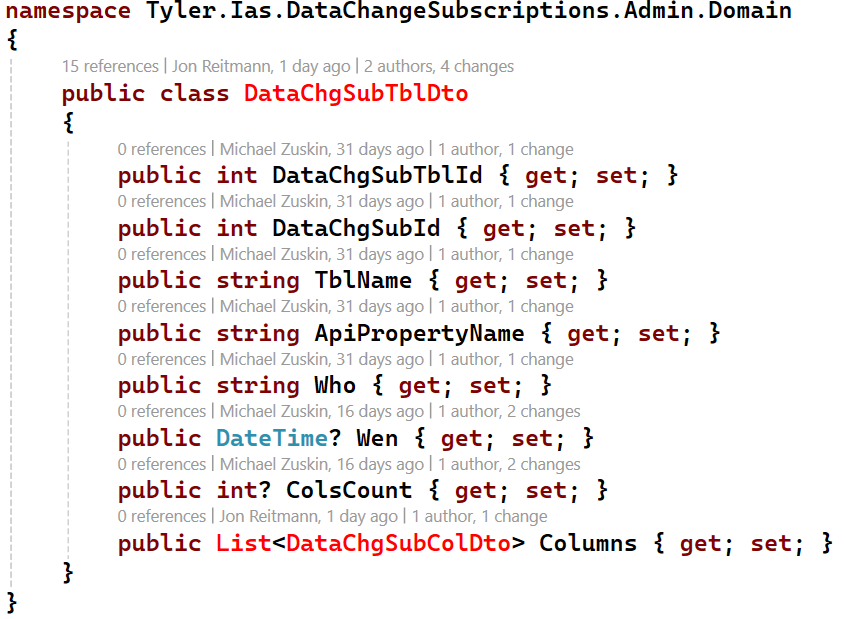
* MT – communicates with the DB back end (keep in mind that, in fact, the whole .Net part is the middle tier, but this is the terminology accepted in Tyler).
* PT – communicates with the presentation tier.

As an example, we will use DataChgSubTbl objects ("Data Change Subscriptions Table").

**--== MT ==--**

**The DTO (“Data Transport Object”) class:**

Describes an entity (so, each entity has a class).



**WCF WEB Service interface:**

Describes the functions, related to an entity (so, each entity has an interface).

A screenshot of a computer screen

Description automatically generated

For bulk, the SAVE function accepts and returns an array of the entities, and the DELETE function accepts an array of IDs:

**WCF Web Service class:**

Directly calls the Oracle package procs:

All the functions except DELETE return data: the SELECT list returns an array of entities, the SELECT single and SAVE – one record.

A few procs return recordsets with the same structure. To avoid code duplication, refactor reading the recordset to a separate ExecProc function:

A screenshot of a computer program

Description automatically generated

Then call ExecProc from the relevant functions. Notice that the SELECT single and SAVE functions return the 1st element of the recordset (since only one record is returned by the proc):

A screenshot of a computer program

Description automatically generated

The described SAVE & DELETE functions operate on a single entity.

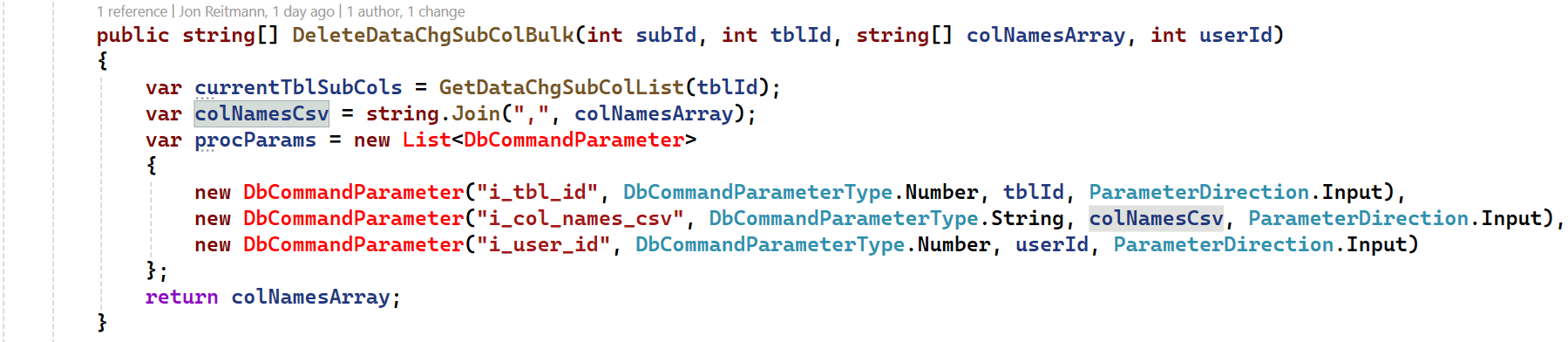
The next code demonstrates the SAVE & DELETE functions which operate on many entities at one stroke (“bulk”).

The SAVE function converts the accepted array of entities into a Jason string to send to the proc:

A screenshot of a computer code

Description automatically generated

The DELETE function converts the accepted array of IDs into a comma separated list to send to the proc:



**--== PT ==--**

**WCF Web Service client class:**

A proxy class that acts as a bridge between PT and MT. Calls the functions of the just described WCF Web Service object of MT.

The class uses Channel - a Web Service server object automatically created by WCF which is the actual connection to the Web Service.

That object implements the service contract – the functions defined in the Web Service object of MT.

It handles communication, serialization, and other low-level details, allowing the client class to forward method calls to the service seamlessly.



**A WCF Web Service client wrapper:**

An abstraction layer useful for decoupling and extensibility - for example, you can add additional logic to it.

While it adds a bit of boilerplate, it ensures your application is more adaptable to future changes.

The naming convention is simply “Service” since it represents the Service to the rest of the application, hiding the actual Service client class.

A screenshot of a computer program

Description automatically generated

**Web API Controller:**

Defines which .Net functions are called by different HTTP requests depending on the combination of the URL and the HTTP method.

The endpoints should be created in the following manner:

|  |  |  |  |
| --- | --- | --- | --- |
| CRUD OPERATION: | HTTP: | MAIN (PARENT) ENTITY: | CHILD ENTITY: |
| Get all entities | GET | /entities | /entities/{entityId}/child\_entities |
| Get a single entity | GET | /entities/{entityId} | /entities/{entityId}/child\_entities/{childEntityId} |
| Add an entity | POST | /entities | /entities/{entityId}/child\_entities |
| Update an entity | PUT | /entities/{entityId} | /entities/{entityId}/child\_entities/{childEntityId} |
| Delete an entity | DELETE | /entities/{entityId} | /entities/{entityId}/child\_entities/{childEntityId} |

Notice that the same SAVE function is called with HttpPost (for INSERT) and HttpPut (for UPDATE):



A black and white image of a spiral

Description automatically generated

**Front end (presentation tier)**

Uses:

* RxJS (Reactive Extensions for JavaScript) - a powerful library for reactive programming using Observables.
* NgRx (Angular Reactive Extensions) - a framework for building large scale, interpose level reactive applications. It’s an Angular/TypeScript framework based on the RxJS library of JavaScript.

Up to this point, we were using the traditional Imperative Programming.

It focuses on describing how a program operates through a sequence of statements that change the program state.

Imperative Programming uses statements that explicitly describe the steps to complete a task ("Do this, then do that").

Reactive Programming is a paradigm that focuses on working with asynchronous data streams and the propagation of changes.

It's an approach to building software that emphasizes responsiveness to data changes and events ("When this happens, respond by doing that").

You can manually implement Reactive Programming in a traditional environment (such as .Net) but that requires some boilerplate code to manage events, delegates, publishers and subscriptions.

NgRx is doing most of the work automatically behind the scenes, so you can concentrate on the business logic.

While Imperative is more intuitive for beginners and widely understood, Reactive has a steeper learning curve and requires a shift in thinking.

At first glance, it seems complicated because you must work with more abstract flows of program logic and get acquainted with new kinds of programming objects which support that flow.

But with experience, you understand that this is a well-organized and well-thought-out system that makes development easier.

**Observable**

An Observable is an object that can emit one or more values over time.

It represents a stream of data that can be observed by subscribers.

Instead of getting a single value, you subscribe to an Observable to receive multiple values over time.

Observables are a way to handle asynchronous data streams like HTTP requests, user input, and more.

This is the foundation and core tool of RxJS, so it is important to understand and "feel" it.

The next example is not related to the Tyler app but it demonstrates a simple use of an Observable.

It uses of() - a utility function that creates an Observable which emits the values provided as arguments, one after the other.

After emitting the last value, the Observable completes (i.e., it doesn't emit any more values).

Observable variables' names must end with $. That is not enforced by compiler but it's an accepted naming convention.

A computer code with green text

Description automatically generated

You could ask: what exactly does subscribe to the Observable?

The answer is simple, but you should memorize it because it's the cornerstone of reactive programming:

**The function passed to .subscribe() acts as the subscriber. Each time the Observer emits a value, that function is called, getting the emitted value as the parameter.**

In this case, the subscriber is:



As you can see, the name of the subscribe() method is misleading.

Calling it looks like the Observable is subscribing to something, when in fact it is being subscribed to.

It would be more correct to name this method getSubscribedBy().

Observables are lazy by default. This means that an Observable does not start emitting values until it is subscribed to.

When the subscribe() method called, that is what "activates" the Observable and it starts emitting values.

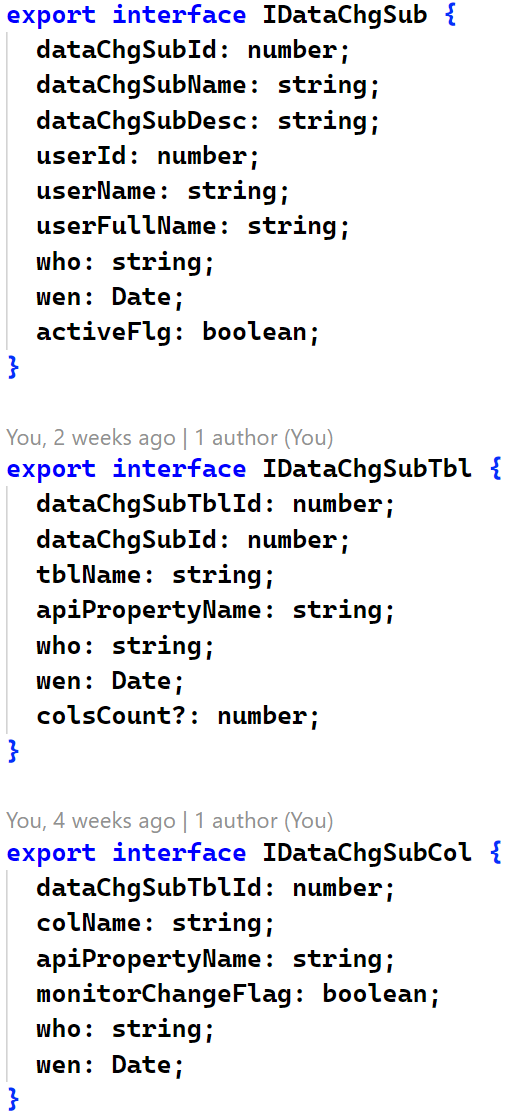
The flow steps after calling subscribe():

1. Value 1 is emitted: The subscriber function is called with 1, so console.log(1) runs.
2. Value 2 is emitted: The subscriber function is called with 2, so console.log(2) runs.
3. Value 3 is emitted: The subscriber function is called with 3, so console.log(3) runs.
4. The Observable completes, and no further emissions occur.

**Model:**

The file declares the data types which describe the entities used in the module (screen).

Usually, they have the same structure as the respective DTOs in .Net, just translated from C# to TypeScript.



**State**

State refers to a singleton object that holds the whole data of the module (normally, a screen with many related data objects) as it is at the current moment (i.e. the current state).

State:

* Centralizes data management providing a single source of truth for the module's data.
* Ensures that all parts of the module are synchronized with the same version of the data.
* Simplifies communication between components by avoiding direct data sharing and, worse, multiple copies of the same data.
* Ensures that growing amounts of data are handled efficiently and predictably.

The State file declares the data type for that singleton:

A screenshot of a computer

Description automatically generated

You can see that the file also creates the Initial State object of the declared type.

It provides the point for the module's State and ensures that the State is always defined, even before any data is retrieved.

**Action**

Action is a plain TypeScript object used to express an intention (usually, a data manipulation, or a change in the application’s state).

An Action is dispatched (launched) in one part of the application and captured in others, providing an easy global communication channel.

NgRx automatically manages the chain of fired events when an Action is dispatched, ensuring the appropriate consumers respond to it seamlessly.

Actions are one of the main building blocks in NgRx.

An Action object has two properties:

* *type* - a textual description of the intention.
* *payload* (optional) - additional data required for the action - to enforce type safety when the Action is dispatched.

IMPORTANT! The *type* property must be unique across the entire application since it serves as a unique identifier for each Action.

Conventionally, its structure is "[Module] Description". That allows different Modules to have Actions with a same Description.

Firstly, the Action file declares a private enum each constant of which will be used as the *type* property of an Action:



Notice that:

* Each CRUD operation of each entity has its dedicated Action.
* Each main CRUD Action has its own …Success action. The Success action is dispatched if its main Action has been executed successfully.

Then, the Action file declares the Actions themselves.

The props parameter to the createAction() function describes the Action’s *payload*:

* The *payload* of a main Action is the WEB Service input.
* The *payload* of a Success Action is the WEB Service output, which is the Action’s input.

A close up of text

Description automatically generated

**WEB Service**

The class which calls the Middle tier WEB Service directly.

Notice that:

* The http function (get, post, put & delete) must fit [HttpGet], [HttpPost] etc. in the PT’s Web API controller.
* The URL must fit the URL of the PT’s Web API controller formed by its [RoutePrefix] and [Route] directives.

A screenshot of a computer code

Description automatically generated

**Side Effect:**

Side Effects are operations that occur outside the context of Angular, such as external APIs calls or HTTP requests.

Each time you see "Effect" as an NgRx class or a file, keep in mind that "Side Effect" is meant.

Effects are implemented using RxJS Observables and are set up to listen for specific Actions and perform side effects without affecting the Store directly.

Our Effect class captures a dispatched main Action and calls the corresponding function of the WEB Service class.

Once a side effect is successfully completed, the Effect dispatches the respective "Success" Action to update the Store with the HTTP response data.

******

A computer screen shot of a computer code

Description automatically generated

**Reducer**

The file declares the Reducer function which captures a dispatched Action and creates a new State which reflects the result of the Action.

The mission of the Reducer is critical – it’s responsible for updating the State with the results of the WEB Service calls when a …Success Action is dispatched by the Effect.

Also, you can create and dispatch an Action whose purpose is to update the State without calling side effects (like two the first Actions - without “Success” - in the example below).

That is the correct way, we don’t update state from other functions – Reducer is one centralized file with all State updates.

A screenshot of a computer

Description automatically generated

**Component**

Simply speaking, an Angular Component is a piece of the screen.

For example, the Subscription List is one Component, the Subscription Card – another (“card” is “details form” in the Angular terminology).

Also, there is one Parent Component which represents the whole screen (the Module) and displays the Child Components.

Each Component consists of 3 parts:

* A TypeScript class with the logic.
* An HTML template.
* A CSS file with optional overrides of the application’s default CSS.

A typical folder of a Module contains 3 the mentioned files of the Parent Component, and sub-folders for each Child Component:

A screenshot of a computer

Description automatically generated

What makes a TypeScript class an Angular Component is the @Component() decorator.

An example for the Subscription Card:

A computer code with text

Description automatically generated with medium confidence

Obviously, the **templateUrl** and **styleUrls** properties contain the Component’s HTML and CSS files.

The **selector** property is a custom HTML tag which represents the Component.

When this tag is placed on the Parent Component’s HTML, the whole Child Component’s HTML is rendered in that spot in runtime.

Here is a fragment of the Parent Component which uses the Child’s selector as an HTML tag:



The tag also passes data from the Parent Component to the Child Component which will be rendered here.

**[contextDataChgSub]** represents the contextDataChgSub property of the Child Component. It was declared with the @Input() decorator to make this use possible.

**"contextDataChgSub"** represents the contextDataChgSub property of the Parent Component.

**Store**

Store is a singleton object that is an application-level data container.

It holds the States of multiple modules combined, which makes up the state of the whole application.

The Store is the single source of truth for the current state of the application.

It provides a way to access the state, dispatch actions, and subscribe to state changes.

The Store instance is created by Angular automatically, you only need to inject it into the constructor of your component.

While the Store is updated by the Reducer file only, any Component can retrieve data from it.

The next fragment of the Parent Component (Subscription) demonstrates working with the Store:

A screenshot of a computer code

Description automatically generated

Things to note:

* The application’s Store is injected into the constructor (“injected” means “automatically passed by Angular). It’s a TypeScript specific syntax which makes the code shorter. Each constructor’s parameter performs three tasks:
* Receives a value, as any regular parameter.
* Declares a class-level variable (as if you would declare it in the regular way).
* Populates it with the received value (otherwise you would populate the var in the constructor’s body, which can now be empty).
* Data is retrieved from the Store using it select() function, which returns an Observable.
* Respectively, the accommodating variables are Observables too. Each one is subscribing to a property of the State. If a property will be changed by the Reducer at any moment, its Observable will know that immediately, so the HTML template (the consumer of the Observables) will be automatically re-rendered to reflect the change.
* While HTML works with Observables directly, sometimes we need to read an observed value in TypeScript, in the “imperative way”. For that, a function, which assigns the value to a traditional, non-Observable variable, must subscribe to the respective Observable. In the example, it’s the contextDataChgSub variable. Even though it’s a regular variable, it will always be in sync with contextDataChgSub$ thanks to the arrow function which assigns a value to it, and which is subscribed to the changes in contextDataChgSub$.
* The subscription is managed using an instance of the Subscription class. Before the Component is destroyed, you need to unsubscribe manually since, potentially, the Observable can keep emitting values. No need to unsubscribe if the subscription was done not using the subscribe() function. For example, the select() function of the Store also creates a subscription, but its unsubscribing is managed by Angular.
* getDataChgSubListAction is dispatched for the initial populating of the Subscriptions List. The stored proc has no input parameters, so no load is passed to the Action.

The next fragment of the Subscription Table Component demonstrates dispatching Actions for the INSERT, UPDATE and DELETE operations, which accept load:

A screenshot of a computer

Description automatically generated