**Garvice Eakins – Lead Software Development Engineer (Smartsheet)**

**Migrating your Javascript to Typescript**

**In software it’s rarely as simple as renaming a file extension to convert from one language to another. Yet in many ways that is the case for Typescript. So why isn’t everyone migrating? The reality is, there is a lot more to consider when your system has legacy code and a business that can't sit and wait while the code is migrated. In this talk I’m going to walk through a strategy to successfully manage the migration. Covering various scenarios you will encounter and present ways to overcome them.**

**Learning Points:**

**What a successful migration looks like - laying the ground rules.**

**How to execute the migration - a step by step process.**

**Putting it all together - strategies to handle the not-so straight-forward.**

**Sean Kelly**

**Bots in containers, oh my!**

**Containers are the new unit of deployment. Intelligent bots are all the rage. We'll look at Docker and the Microsoft Bot Framework and talk about how you can build and debug a bot in a container. Along the way we'll cover the basics of containers and why they are important tools you should be using daily. We'll also go over what's new in version 4 of the Microsoft Bot Framework.**

Sean (Salesforce) – improving on the performance of lightning platform. 11 engineer positions.

What are you in for?

* What is TypeScript and why migrate?
* Rules to measure success
* Define the migration process
* Develop strategies to handle “legacy patterns”

What is TypeScript?

* JavaScript, but better - .js to .ts (any valid JS is a valid TS)
* Statiscally typed
* Code has type annotations for field, method, and return types
* New ES 6+ language features
* Class keyword
* Arrow functions
* Let and const
* Transpiles down to ES 5 JavaScript at build (run) time

*Import are upto you, at smartsheet we have our own bundling system. You can choose to use import/require.*

*We have complete registry of system.*

JavaScript

function ToDoItem(params) {

this.displayNode = params.displayNode;

this.name = params.todoName;

this.dueDate = params.DueDate;

this.state = params.todoState;

this.elementId = params.ElementID;

this.name = params.todoName;

}

TypeScript

Class TodoItem {

public displayNode: HTMLElement;

….

constructor(params: {

htmlElement: HTMLElement,

elementId?: string | number,

isReadOnly?: boolen,

name: string,

dueDate?: number,

state: number,

}) {

This.displayNode = params.htmlElement;

}

*Question market*

*Null is defined by a million dollar mistake*

Why TypeScript?

* Type safety
* Code with types is easier to use and understand
* IDE intelligence – autocomplete, refactoring, etc.
* New ES 6+ language features
* Less code, more action
* Increased developer productivity and happiness 😊

*Constantly spending time to parse and understand, that’s a horrible place to be in.*

*In TS, look at the name, the params, poof. You know what it is.*

Why migrate?

* Full benefits of type safety require majority of code to be typed
* Understand code
* Refactor code
* Switching between JS and TS is error prone
* Once you type you’ll never any (TS joke)

*You can turn on TS lint.*

*You can fail build and make developer fix the mistake.*

Setting Goals for Success

* Zero business impact
* Zero bugs / no regressions
* Feature development continues
* Minimize developer impact
* Maximize speed
* Migrate incrementally
* Add type information to existing code
* All (owned) code is migrated

*They only care about the user experience.*

*Want to refactor the code? What’s the effect on my bottom dollar?*

*Migrate as seamless as possible*

*6000 lines of JS code is zero type safety*

*Your libraries should be in TS*

Minimize risk / maximize speed

* DO
* Type everything
* Disable TSLint on per-file basis
* Mark properties as optional or nullable
* Convert JSDoc typedefs to TypeScript interfaces
* Remove Comment-style type info (JSDoc)

*TSLint - Linter that imposes type restrictions*

*Don’t ship properties that are null across the wire*

*JSDoc typedefs to TypeScript interfaces – tells the client code this is how it looks like (interface)*

*See a bug in the code? Don’t fix it! You will end up fixing all the codes that are reliant to the bug.*

Minimize risk / maximize speed

* DON’T
* Change semantic behavior
* Change var to let or const
* Use ‘any’ type
* Refactor code
* Update to use es6 goodies

*If you use the ‘any’ type, why are you converting? It will be zero type safety.*

*Refactoring code requires a big testing scope. Make sure don’t introduce regressions.*

*Again, See a bug in the code? Don’t fix it! You will end up fixing all the codes that are reliant to the bug.*

Don’t change code!

* Primary goal is to add type information to existing code
* Transpiled output should be semantically equivalent to original JS
* Selectively disable TSLint rules to avoid semantic code changes
* Bugs and other code issues found during migration are tagged

If ((this.migratedChildClass as any).shouldRefreshChildred) {

This.migratedParentClass.refreshChildren();

}

*Don’t fix this bug. Just migrate it.*

Migrate incrementally

* Migrate one file at a time
* Reduce merge conflicts
* Share responsibility among developers
* Reduce cost per file
* Reduce burden of code reviews

*If do all things at once when you get to the end… not going to happen.*

*No customer value (if migrate all), the end of the day your code is thrown out.*

*Regex will help you out in migrating one file at a time.*

*We want both junior and senior engineer to be able to migrate.*

*i.e. 85 files to code review is a nightmare.*

how to migrate a file

* Create a wiki page detailing your process and your rules
* Create document tracking migration files (Smartsheet)
* Rough outline of process:
* Use TypeScript Migration sheet to track migration
* Migration single file on a personal branch
* Wait for code review
* Merger migrated file to develop

*For document tracking, you can use Git Hook.*

*Mark out items that should not be migrated.*

*Make it explicit that it is part of migration process and not part of feature development.*

*You are doing this for the team. That’s why developers will be motivated to do this.*

What about unmigrated types?

* The unmigrated types used in a migrated file are problematic
* When possible, migrate files used by new features
* Type aliasing and any to the rescue!

TypeScript migration declaration

* Single file stores the declartions:
* TypeScriptMigration.d.ts

// unmigrated classes

Type UserConfig = any;

declare let UserConfig: UserConfig;

// Global functions

declare function toLocalizeData(date, locale): Date;

// Global variables

declare var userConfig: userConfig;

* Unmigrated classes
* Global functions and variables
* Types and globals removed from file as they are migrated

*Types that we used in 1 file, is now linked with the new file.*

*Compiler error now, compiler will save your day.*

Migration examples – constructor

Migration examples -methods

this.isReadOnly = isReadOnly;

public getDueDateForDisply(): string {

return toLocalizeDate(this.dueDate, userConfig.Locale).toDateString();

}

public setIsReadOnly (isReadOnly: boolean): void {

this.isReadOnly = isReadOnly;

this.displayNode as any).disabled = isReadOnly

}

Gotcha – undefined properties

* Don’t change any logic If you get a TypeScript compiler error!
* Use as any to effectively disable type checks for a variable

Gotcha – constants on prototype

* Prototype is mostly invisible in TypeScript land
* Workaround: declare constants as fields, then define outside class

*Before*

TodoItem.prototype.CLASS\_PENDING = ‘todoItemPending’;

TodoItem.prototype.CLASS\_IN\_PROGRESS = ‘todoItemInProgress’;

TodoItem.prototype.CLASS\_COMPLETED = ‘todoItemCompleted’;

TodoItem.prototype.CLASS\_HOLD = ‘todoItemOnHold’;

*After*

class TodoItem {

public CLASS\_PENDING: string;

public CLASS\_IN\_PROGRESS: string;

public CLASS\_COMPLETED: string;

public CLASS\_HOLD: string;

}

Gotcha – enums

* JSDOC @enum is not quote the same thing as a TypeScript enum
* Use the regular number type when migrating enums to TypeScript

*Goal is not to change the code*

*Before*

TodoItem.State

*After*

Class TodoItem {

Public static readonly State = {

PENDING: 1;

}

}

*We proved the build process first.*

*Convinced that nothing bad will happen.*

*Then was allowed to do migration process.*

*First to migrate was only a button.*

*So far we had 0 bugs.*

*As engineers, we knew it was a great idea, so we did it methodically.*

*It’s main goal is the speed of developer. To avoid parsing through the code to understand what it does.*

*No one wants to touch a 10-year old JavaScript with 10,000 lines.*

*Learning TypeScript: by using Angular 2. Other people just played around with it. Passed on different resources. Creating a wiki – JavaScript variables vs TypeScript variables.*

*If it doesn’t conform to our code review standards, it is a liability.*

*If we are not sure, we don’t put it in code production review.*

*Pluralsight have TypeScript. Medium articles.*