**Team Gizmo**

**Business Problems**

1. Clients call the Help Desk for problems that repeat over and over, e.g. missing Learning Essentials tile, where a root cause may never be fully investigated or investigated at all.
2. Developers on support rotation solve a problem that some other developer on support rotation just solved. This wastes the developer's time, and the client's time in waiting for a resolution.
3. The software development cycle is prone to inefficiencies, and avoidable technological costs. Time spent considering scenarios for the use of specific technologies (e.g. all forms of Spring) should instead be used to solve business problems.

**Potential Solutions**

1. Create an INC tracker (web based) that categorizes problems at a more granular, and FLEX-specific level, than does ServiceNow
2. Adopt a business oriented mindset, and not technological, when designing a software application.

For example:

* "I want to use Spring Cloud Stream in this project because I think it's just the coolest thing out there, and it begins with the word Spring."

versus:

* "I have a business requirement to read messages off a broker. What's the most efficient and cost effective technology to accomplish that?"

**Summary of Application**

* Categorize a ticket by keywords, e.g. LMS, W2CHECK, HRO toggle, Web132C, FSA, etc., and store the INC number along with those keywords
* Keywords will be created per Assignment Group (e.g. DEV - User Access, DEV – API, DEV – Ent Data Integration, etc.), so that each group can decide what best describe their issues.
* A maximum of two (2) keywords will be allowed to be associated to a ticket. Preferably only one will be selected. A greater number will dilute the usefulness when searching, i.e. will bring back too many records
* Keywords would not be assigned until the ticket was with the final group that owns the issue
* The next time an LMS issue arises for User Access (as an example), the keyword(s) can be specified and (hopefully) an existing solution can be retrieved
* The solution to the ticket can be associated when the record is created, or later

**PROS**

* The "Solution" to the ticket can be stored under that INC number and therefore retrieved by keyword
* In addition to the keyword search, each column in the table display for all INC's will be searchable per the use of ag-Grid (a business problem solved via the correct technology)
* All INC's that have something to do with (e.g.) "LMS" can be displayed at one time
* Should help to reduce developer frustration and time during INC rotation: previously solved INC's can be used as a basis for finding a solution for new INC's with the same problem
* Should help to reduce client frustration since INC's can be solved faster
* Over time, a pattern (per keyword) may emerge as to the root cause, which could lead to a fix in the data and/or code
* Updates/Deletes against an INC will not require having to click a "Save" button - just tabbing off the field will persist the value to the db (a business problem solved via the correct technology)

**CONS**

* Creating the association between an INC and its keyword(s) will be a manual operation
* The keywords that best describe an INC will be determined by the person currently on rotation, therefore it will be a subjective evaluation
* Not able to employee a REST interface to ServiceNow to pull additional data. Have been told by the ServiceNow Applications Administrator, "Unfortunately with our multi-sso configuration you can’t use your regular credentials to access the table APIs."

**Approach to designing the INC Tracker application**

Three C's will be considered in promoting efficiency - Coupling, Cohesion, and Cost.

**Coupling**

Attempting to achieve: **LOW**

* Each module will exist and can change independent of all other modules
* Minimize developers stepping on each other’s toes during development
* Minimize merge conflicts
* Back-end can be used to serve x number of interfaces simultaneously
* Front-end can switch between back-end modules with minimal configuration changes, i.e. modify only the url and port for the REST endpoints

Strategy:

* Other than additions/deletions/data modifications to REST endpoints, changes to the front-end will result in no changes needed in the back-end.
* The front-end's knowledge that a back-end module exists is limited to a URL and Port number being specified in a front-end config file
* The back-end module will have zero knowledge that a front-end module exists. To reinforce that, there will be no java class names that mimic ts/js names which might indicate a hard link if the two code bases are compared
* The persistence module in use will NOT be hard-coded into the app, but instead change based on which application server the back-end is deployed. For example, if the application is deployed to:
* WebLogic, Payara, GlassFish: uses EclipseLink
* TomEE, WebSphere, Geronimo: uses OpenJPA
* WildFly: uses Hibernate
* etc.

NOTE: there will be no need to import Spring dependencies since the back-end code will be written as a minimalistic Jakarta EE application (fewest dependencies -> smallest war file -> fastest start-up). If the code is deployed to an EE compliant application server (e.g. WebLogic, Payara, WildFly, TomEE, GlassFish, etc.), it will function as designed

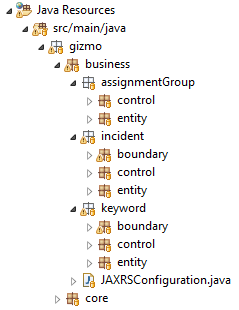
**Cohesion**

Attempting to achieve: **HIGH**

Each element within each module does its own thing - promotes robustness, reliability, reusability, and understandability

Strategy:

* The BCE (Boundary-Control-Entity) standard will be used to structure back-end code packages into specific business concerns. This strategy helps eliminate the practice of packaging code by technical responsibility, e.g. all @Entity classes exist in a single entity folder.
* A *\*Resource* class (within the Boundary directory) will be the interface to the outside world for a specific business need
* A *\*Manager* class (also within the Boundary directory) will contain the business logic just for that *\*Resource* interface and will be @Injected into the Resource.
* Both will be stateless session beans, i.e. they will represent business logic only - they will have no data, just operate on data
* Persistence entities (@Entity annotated classes) will be packaged by business need and not lumped within a single generic directory. Since their creation will satisfy a specific business function, each one will be tied to a *\*Resource* class in the applicable Boundary layer. To avoid potential duplication of effort, each will also be available via import... to whatever other business function may have a need.
* To promote the low coupling concept from a developer standpoint, the front-end and back-end code bases will be (1) in separate GIT projects, and (2) developed using different IDE's.



**A BCE packaging example**

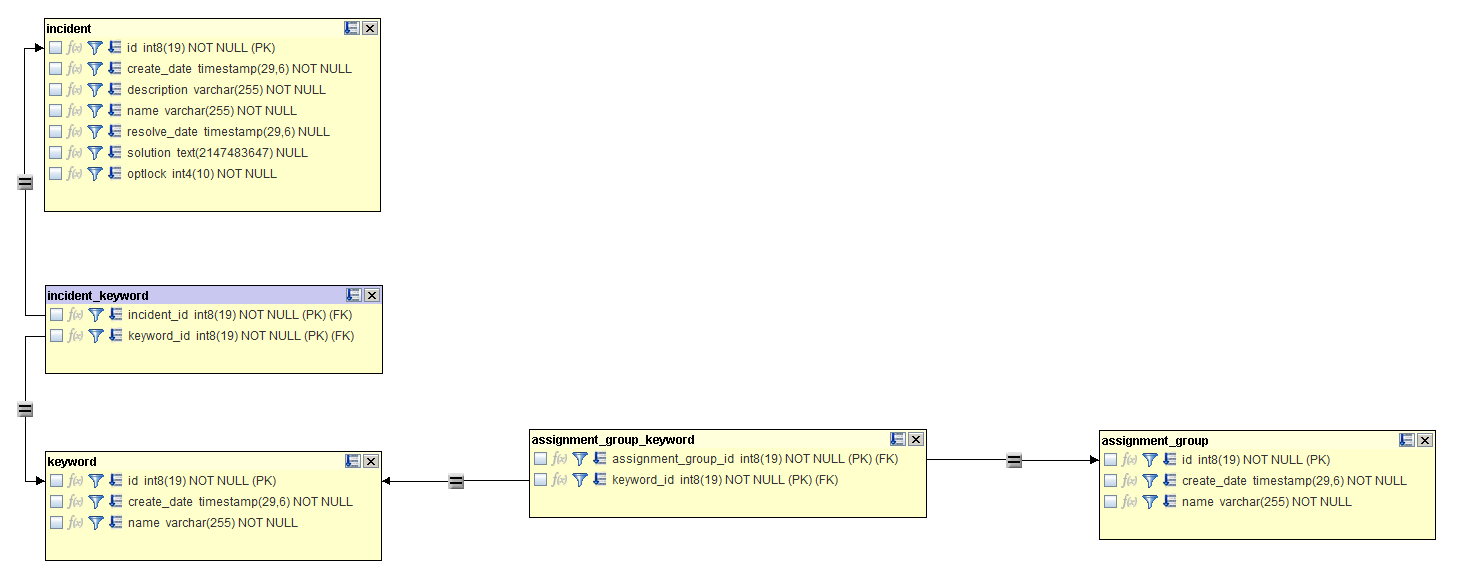
**Cost**

Attempting to achieve: **ZERO**

Technologies used:

* Web server: NG Live Development Server (changes to a file cause an automatic server refresh)
* Application Server: WildFly (JBoss)
* RDMS: PostgreSQL
* Programming language: Java
* Web to application server interface: RESTful
* IDE's: Eclipse and Visual Studio Code
* UI Scripting: Angular and Typescript
* UI grid layout: ag-Grid
* Dependency manager: Maven
* Persistence auditing: EntityListeners (javax.persistence.EntityListeners), which provide automatic auditing of CRUD operations (INSERT/DELETE/UPDATE) without the expense or complexity of third-party software/hardware.
* Testing platforms: Postman and cURL
* Db backup: a scheduled nightly task will create INSERT scripts and store them to disk

Cost of all proposed technologies: $0.00



**Entity Relationship Diagram**