Project 2 (Option B)

Steve Mazza

I will walk through the TacMark (Tactical Marketplace) product. I have been serving as the project lead since it started almost two years ago. It represents the full time effort of two dedicated government engineers, myself, and two part time junior government engineers. The full time resources have remained on the project since its inception but the junior engineer positions have rotated several times over the last twenty-four months. I chose to describe the project here not only because I am intimately familiar with it but also because it has sufficient complexity to be useful for discussion. I am fiercely proud of the fine work done on this project to which I give the entire credit to the talented government engineers[[1]](#footnote-1).

# General Component Hierarchy

In the following discussion I make heavy use of references to external work. This is in keeping with our philosophy of adopting and promoting a standard way of communicating within our development community. For those engineers who are familiar with the referenced work, the job of digesting a complex architecture is made considerably less onerous.

## Tactical Marketplace Server

The Tactical Marketplace (TacMark) server is a web server that serves content and RESTful services over HTTP. TacMark's Web API is feature complete: any task that can be performed in TacMark is exposed as RESTful services in the TacMark Web API.

In addition to serving RESTful services, the Tactical Marketplace server hosts documentation for its services. Live documentation allows developers to view and exercise[[2]](#footnote-2) TacMark's Web API through the documentation page located at `http://<tacmarkserver>/api/documentation`. For each endpoint, developers may pass parameters and view the returned output (formatted as JSON) from the server.

The server also features an Administrative web console for loading and unloading modules at `http://<tacmarkserver>/admin/modules`. This functionality is not exposed over HTTP as a web service because it should only be used by a super user who has access to the server machine that TacMark is running on.

## Platform

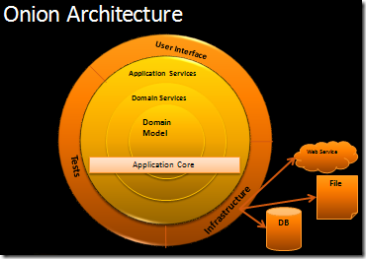
TacMark server runs on the ASP.NET web platform and is written against MS .NET 4.0 and ASP MVC 3. It follows that TacMark runs on IIS 7.5 on a Windows 7 Army Gold Master machine.

In addition to IIS and ASP MVC to power the web interface, TacMark implements its persistence in MS SQL Server 2008 R2 (Express), and implements its search functionality in Apache Solr configured to run on a Tomcat 7 service on localhost. This capability abstracts our search functionality from the underlying persistence. To wit, our port to Postgress has been almost trivial.

## Architecture

Tactical Marketplace server's architecture promotes extensibility and longevity. Instead of implementing a smart phone application marketplace for the Army, the team engineered a general purpose framework for the storage, management, and distribution of digital assets to consumer devices. The core framework is built on by core and extension modules in order to form a usable marketplace.

The closest description of the TacMark server's big picture architecture is Jeffrey Palermo's [Onion Architecture](http://jeffreypalermo.com/blog/the-onion-architecture-part-1/).



In the onion architecture, dependencies point inward from outer to inner layers but never the other direction. The next sections do the opposite of peeling an onion - start with the center moving outward.

### Core Domain

The domain, the heart of the application, is implemented in the `TacMark.Domain` library. Following Domain Driven Design, introduced in Eric Evan's [book](http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215), TacMark's domain defines the core objects central the problem that TacMark solves: the storage, management, and distribution of digital assets to warfighters. These [entity classes](http://martinfowler.com/bliki/EvansClassification.html) include Users, Digital Assets, Device Types, Digital Asset Types, User Devices, Teams, etc. TacMark avoids the [anemic domain model](http://www.martinfowler.com/bliki/AnemicDomainModel.html) anti-pattern by coding the relationships between all the domain entities within the domain entity classes themselves. To this end, TacMark identifies the following entities as [aggregate root entities](http://lostechies.com/jimmybogard/2008/05/21/entities-value-objects-aggregates-and-roots/):

* DigitalAsset
* User
* Team

Certain operations span across entity classes and don't belong in any one class. In these cases, TacMark hosts these operations in the [service layer](http://martinfowler.com/eaaCatalog/serviceLayer.html). Classes belonging to the service layer can be found in `TacMark.Domain.Tasks`. An example of one of these tasks is the `ProcessMissionPackEquipStatusTask` which is heavy with logic to answer questions about the mission packs[[3]](#footnote-3) for teams and users' devices. These questions, such as `GetUsersDevicesEquipppedInstallationReport(User user, Device device, DigitalAsset asset)`, clearly reach across entities and therefore do not belong in an entity class.

Like many other tasks in the service layer, the `ProcessMissionPackEquipStatusTask` is interface abstracted from the rest of the application; in this case, it is hidden behind the `IProcessMissionPackEquipStatusTask`. This loose coupling simplifies unit testing and allows the dependency injection framework to satisfy the application's need for an `IProcessMissionPackEquipStatusTask` at runtime. It also facilitates use of the [strategy pattern](http://en.wikipedia.org/wiki/Strategy_pattern).

#### Unit of Work and the Repository Pattern

The Tactical Marketplace abstracts its persistence implementation behind the [repository pattern](http://martinfowler.com/eaaCatalog/repository.html). This pattern exposes an easy [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete) interface for aggregate root entities. Querying the repository is abstracted with Microsoft's LINQ technology by exposing collections of aggregate root entities as `IQueryable` collections.

There is a separate repository interface defined for each aggregate root entity class e.g. `IDigitalAssetRepository`, `IUserRepository`, and `ITeamRepository`. These interfaces are very similar and seem to violate the [DRY principle](http://www.artima.com/intv/dry.html). Some developers prefer to use a [generic repository](http://codebetter.com/gregyoung/2009/01/16/ddd-the-generic-repository/) for entities instead of separate interfaces and internet flame wars result. At this time, TacMark uses the Managed Extensibility Framework which does not support open generics; however, community provided additions [can support open generics](http://codebetter.com/glennblock/2009/08/20/open-generic-support-in-mef/).

The `TacMark.Domain.Repositories.IUnitOfWork` interface is an abstraction on the [unit of work pattern](http://martinfowler.com/eaaCatalog/unitOfWork.html). The interface is available to the application in order to allow outer layers to commit and rollback the current unit of work session. TacMark employs a unit of work session per HTTP request pattern.

The implementation for both the unit of work and repositories is not defined in `TacMark.Domain`; it is left to be defined by a core module and the dependency injection system will satisfy dependencies at runtime. This is the basis for much of our extensibility.

### Core Framework

The `TacMark.Framework` library defines the next layer of the onion. This layer defines the interfaces for application services used by outer layers e.g. Authentication, Logging, and Digital Asset File Storage. In some cases, the implementation of these application services is left undefined so that a swappable core module may provide the implementation at runtime. For example, the Logging interface is defined in `TacMark.Framework`, but the implementation is defined in the core module `TacMark.Logging.NLogImplementation`.

The interfaces, whose implementations are left to modules, have their contracts defined in the `TacMark.Framework.ExtensionPoints` namespace. It follows that module authors need a reference to both the `TacMark.Framework.dll` and the `TacMark.Domain.dll` during development. Helper classes which assist module authors, such as `TacMarkControllerBase` and `SimpleTacMarkAssetApiController` are also defined in this layer.

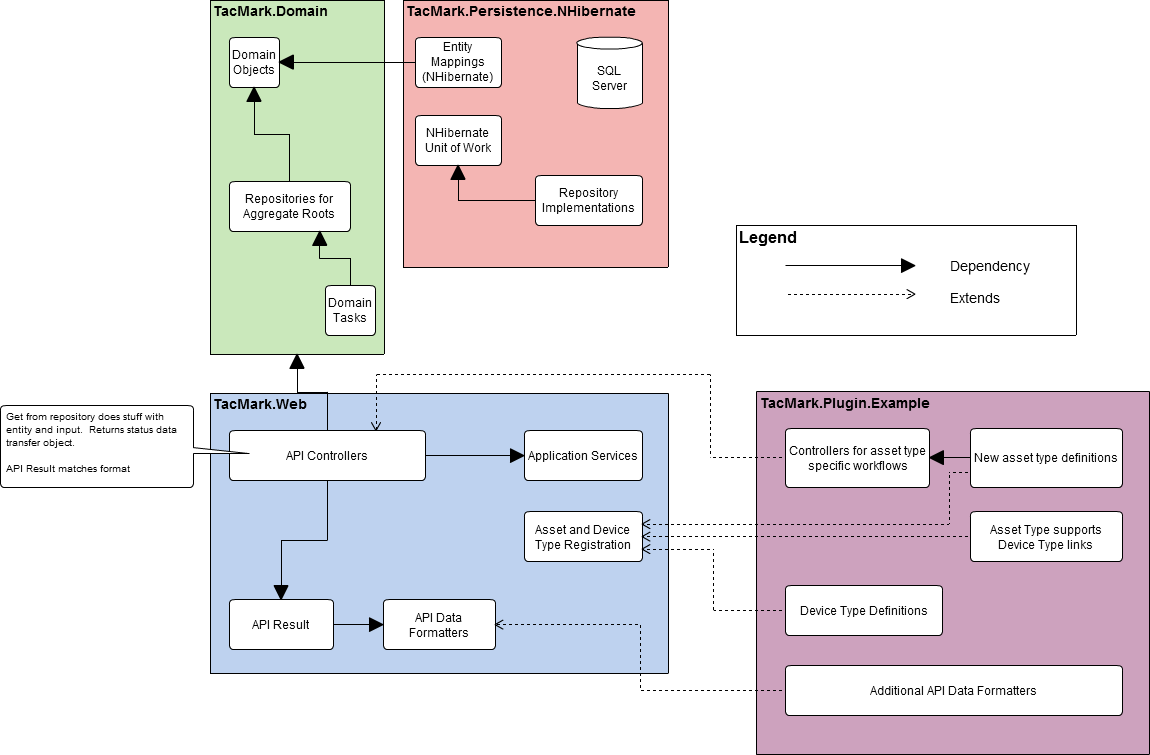
Finally, `TacMark.Framework` is the first layer to introduce references to the ASP MVC framework. Classes in the `TacMark.Framework.Web` namespace link the application services to the ASP MVC framework. For example, `RequiresPermissionsAttribute` implements MVC's `IAuthorizationFilter` which is used to protect ASP MVC Controller Actions from unauthorized users. The `RequiresPermissionsAttribute` depends on a `TacMark.Domain.Repositories.IUserRepository` to query whether or not the current user may perform some operation.

### Web Application

The `TacMark.Web` project is the ASP MVC 3 web application which is the application's point of entry. As the application point of entry, it is this project's job to register the dependency injection container with the ASP MVC application. This will inform ASP MVC how to satisfy the dependencies of framework initialized objects such as Controllers. ASP MVC projects inherently know about an [abstraction on dependency injection containers](http://msdn.microsoft.com/en-us/library/system.web.mvc.dependencyresolver%28v=vs.98%29.aspx); TacMark employs an [implementation based on the Managed Extensibility Framework](http://blog.maartenballiauw.be/post/2010/07/27/ASPNET-MVC-3-and-MEF-sitting-in-a-tree.aspx) to satisfy dependencies in a decentralized, extensible manner. After establishing the dependency injection container, the application augments the container with additional parts from core and extension modules. The application discovers and loads all the core and extension modules from their respective folders at startup. At this time, the dependency injection container becomes aware of the core module defined implementations of Core Framework interfaces, e.g. ILogger dependencies will be satisfied at runtime by instances of the discovered `TacMark.Logging.NLogImplementation` class found in a core module.  This dependency injection and module registration happens largely in the `TacMark.Web.Module` namespace.

This project uses the ASP MVC framework to define TacMark's REST API. The Controllers and Models that make up the REST API are self-contained in an MVC Area called "MvcApi". Instead of returning a generated HTML document, as is typical in ASP MVC, these Controllers return instances of `TacMark.Framework.Web.ApiResult`. The ApiResult uses data formatters (`TacMark.Framework.ExtensionPoints.IModelFormatter`) from modules to serialize a Model object into the data format asked for by the HTTP client. In addition to formats provided by extension modules, TacMark has built in support for JSON and XML data types.

In addition to the REST API, `TacMark.Web` defines a couple more interfaces to the system. The first of these interfaces is the module administration page at `/admin/modules`. From this page, administrators with the `manage\_modules` system permission may delete and upload extension modules while the server is running. After a module is added or removed, the server will restart to load or unload the changes.  The second of these interfaces is the API Documentation Page. The API Documentation at `/api/documentation` is useful for developers of TacMark clients to view and interact with all the REST endpoints the system exposes.



Together, the Core Domain (and related components) and the Web Application comprise the entire TacMark distribution. We have also authored clients for Android and IOS hand held devices. In all cases clients access TacMark through the RESTful API. This is strictly enforced as there is no “back door.”

# Selected Component Context Diagram

Let’s take a closer look at the abstraction on persistence. A full context diagram for all of the API calls that access persistence would be a very complex diagram. Instead I will describe *representative calls* and show the path of request, execution, and fulfillment.

All objects are stored in the RDBMS and this persistence is handled by [Hibernate](http://www.hibernate.org/), an abstraction that allows us not to worry about the underlying data layer implementation. In theory (and hopefully in practice, eventually) this will allow us to move to a schema-less ([NoSQL](http://en.wikipedia.org/wiki/NoSQL)) database like [CouchDB](http://couchdb.apache.org/). This would solve a large problem for us related to extending data types in ways we have not anticipated (i.e., that don’t fit the existing data schema). All requests from the API that access stored data make their way through the Hibernate layer. There is no direct access to data from the TacMark framework.

Enterprise class incremental full text indexed search (with completion and auto correction) is facilitated by [Solr](http://lucene.apache.org/solr/). This gives us parity with the robust search that drives customers at Amazon and eBay, among others. API calls requiring access to this search functionality (e.g., searching asset descriptions, document content, or comments) are passed first to the Solr layer and then to Hibernate.



A complete list of all API calls has been provided in the Appendix and comprises the entirety of what can be done with TacMark[[4]](#footnote-4).

# Selected Component Data Flow Diagram

I have constructed a diagram to extend our example of User Search. The lines represent bidirectional data flow and in our example we follow the data through the GetAsset() call of the RESTful API.



# Narrative Walkthrough of Data Flow Diagram

If the diagram above seems overly simplistic it’s because of the strict encapsulation and loose coupling between the components that comprise the TacMark architecture. Execution and fulfillment of a user search for an asset would be carried out, at least in part[[5]](#footnote-5), by the GetAsset() API call. This is one of over sixty-five calls that comprise the TacMark API. Since this call accesses indexed data, it relies on the Solr layer to facilitate the search results. TacMark knows nothing of the stored data and leverages Hibernate through its Unit of Work implementation which allows transaction control including commit and roll-back.

# Insights

We have created a very complex system that, despite its complexity, is amenable to refactoring. By following a strict paradigm of abstraction, encapsulation, and loose coupling we are able to exercise agile development strategies, respond quickly to user requests, empower third-party development of plug-ins and clients, and integrate new members of the development team into meaningful roles without being overwhelmed.

Throughout this process I have become an advocate of throwing away code. As the team matures its understanding of a given problem space, further development on a less-than-solid foundation of legacy code can only progress to the point that it reaches the boundaries of the existing infrastructure. Leveraging software patterns like those in “Design Patterns” (Gama, et. al), “Pattern-Oriented Software Architecture” (Buschmann, et. al.), and especially “Patterns of Enterprise Application Architecture” (Fowler) not only allow teams to communicate efficiently but also facilitate documentation and division of labor.

# Appendix

## RESTful APIs

### Device Installation Tracking

Report installation POST api/me/devices/{DeviceId}/installs/{AssetId}

Report uninstallation POST api/me/devices/{DeviceId}/uninstalls/{AssetId}

### Digital Asset

Delete digital asset DELETE api/digitalassets/{id}

Get asset GET api/digitalassets/{id}

Get asset collection GET api/digitalassets

Get content GET api/digitalassets/{id}/content

Get icon GET api/digitalassets/{id}/icon

Get install script GET api/digitalassets/{id}/instruct

### Digital Asset Comments

Create comment POST api/digitalassets/{da\_id}/comments

Delete comment DELETE api/digitalassets/{da\_id}/comments/{comment\_id}

Get single comment GET api/digitalassets/{da\_id}/comments/{comment\_id}

List asset comments GET api/digitalassets/{da\_id}/comments

### Digital Asset Dependencies

Get assets dependencies GET api/digitalassets/{da\_id}/dependencies

Introduce dependency POST api/digitalassets/{da\_id}/dependencies/{dependency\_id}

Remove dependency DELETE api/digitalassets/{da\_id}/dependencies/{dependency\_id}

### Digital Asset Tags

Add tag POST api/digitalassets/{da\_id}/tags/{value}

List all tags GET api/digitalassets/tags

List assets tags GET api/digitalassets/{id}/tags

Remove tag DELETE api/digitalassets/{da\_id}/tags/{value}

### Digital Asset Types

Get asset type default icon GET api/digitalassettypedefs/{typeKey}/icon

List asset type definitions GET api/digitalassettypedefs

### Kits

Create kit POST api/kits

Edit kit PUT api/kits/{id}

List kits GET api/kits

### Manage My Devices

Delete device DELETE api/users/me/devices/{id}

List my devices GET api/users/me/devices

Update devices metadata PUT api/users/me/devices/{id}

### Mission Pack

Get devices mission pack status GET api/me/devices/{DeviceId}/missionpack

Get team mission pack status GET api/teams/{team\_id}/missionpack/status

Get teams mission pack contents GET api/teams/{team\_id}/missionpack

Update teams mission pack PUT api/teams/{team\_id}/missionpack

### Register Devices

Register new device POST api/users/me/devices

### Roles

Create new role POST api/roles

Delete role DELETE api/roles/{Name}

Get single role GET api/roles/{Name}

Grant permission POST api/roles/{role\_name}/{system\_task}

List roles GET api/roles

Remove permission DELETE api/roles/{role\_name}/{system\_task}

Update role PUT api/roles/{Name}

### System Tasks

List system tasks GET api/systemtasks

### Team Membership: Leader's Management

Get single membership order GET api/teams/{team\_id}/membership/orders/{order\_id}

List membership orders for team GET api/teams/{team\_id}/membership/orders

List my teams GET api/me/teams

List team members GET api/teams/{team\_id}/membership

Order user to join team POST api/teams/{team\_id}/membership/orders

Remove member from team DELETE api/teams/{team\_id}/membership/{user\_id}

Remove membership order DELETE api/teams/{team\_id}/membership/orders/{order\_id}

### Team Membership: User's management

List membership orders received GET api/me/membership/orders

List team membership GET api/me/membership

Remove self from team DELETE api/teams/{team\_id}/membership/me

Swap device PUT api/teams/{team\_id}/membership/me/device

Update membership order status PUT api/teams/{team\_id}/membership/orders/{order\_id}

### Teams

Create team POST api/teams

Delete team DELETE api/teams/{team\_id}

Get single team GET api/teams/{team\_id}

List teams GET api/teams

### User Session

Log off DELETE api/session

Log on POST api/session

Register POST api/account

### Users

Get my profile GET api/users/me

List users GET api/users

User autocomplete GET api/users/autocomplete

### Users' Permissions

Grant user role POST api/users/{user\_id}/roles/{role\_name}

List user privileges GET api/users/{user\_id}/privileges

List user roles GET api/users/{user\_id}/roles

Remove user from role DELETE api/users/{user\_id}/roles/{role\_name}

Update user roles PUT api/users/{user\_id}/roles

1. Special thanks to John Gilday, Steve Domanski, Alex Hall, Andrew Clifton, Nora Belfar, and Zach Kjellberg. I also want to thank management who gave us the latitude we needed to be excellent. [↑](#footnote-ref-1)
2. The documentation is actually fully interactive and allows the user to interact with actual data. This is a fully home-grown capability that was motivated by our loathing for JavaDoc. [↑](#footnote-ref-2)
3. Aggregate asset bundles used to deliver content to teams. [↑](#footnote-ref-3)
4. The definition of a feature-complete API. [↑](#footnote-ref-4)
5. In practice, the client would likely also implement GetIcon(), ListAllTags(), and ListAssetComments(). [↑](#footnote-ref-5)