The New York Times Developers

Swordfighting with Dagger

Dependecy Injection Made Less Simple

Mike Nakhimovich @FriendlyMikhail

Android Framework Team

@NYTDevs | developers.nytimes.com

What is Dagger?

Alternative way to instantiate and manage your objects

- Guice - Google (Dagger v.0)
- Dagger 1 - Square
- Dagger 2 - Back to Google :-)

Why Do We Need It?

Good Code = **Testable Code**

Why Do We Need It?

More **Tests** = Less **Anxiety**

Why Do We Need It?

Proper Code Organization is

a **requirement** for testing

Untestable Code (Me in the Beginning)

```
public class MyClass {
   private Model model;
   public MyClass() {this.model = new Model();}

   public String getName() {return model.getName();}
}
```

How can we test if `model.getName()` was called?

Internet Told Me to Externalize My Dependencies

```
public class MyClass {
...
   public MyClass(Model model) {this.model = model;}
   public String getName() {return model.getName();}
}...
public void testGetData(){
    Model model = mock(Model.class);
    when(model.getName()).thenReturn("Mike");
    MyClass myClass = new MyClass(model).getName();
    verify(myClass.getName()).isEq("Mike");
}
```

Where Does Model Come From?

Dependency Injection

to the rescue!

Dagger Helps You Externalize Object Creation

```
@Provides
Model provideModel(){
    return new Model();
}
```

Provide from Modules

```
@Module
public class AppModule{
}
```

A **module** is a part of your application that *provides* some functionality.

Provide from Modules

```
@Module
public class AppModule{
@Provides
Model provideModel(){
   return new Model();
}
...
```

A **module** is a part of your application that *provides* some functionality.

Components are Composed of Modules

```
@Singleton
@Component(modules = {MyModule.class})
public interface AppComponent {
  void inject(MyActivity activity);
}
```

A Component is the manager of all your module providers

Next, Create a Component Instance

```
component = DaggerAppComponent.builder()
        .myModule(new MyModule(this))
        .build();
```

Register with Component

```
protected void onCreate(Bundle savedInstanceState) {
  getApplicationComponent().inject(this);
```

Injection Fun

Now you can inject dependencies as fields or constructor arguments

```
@Inject
Model model;
```

```
@Inject
public Presenter(Model model)
```

Dagger @ NY Times

Now for the fun stuff!

50% Recipes 50% Ramblings

Dagger @ NY Times

- Module/Component Architecture
  - Working with libraries
  - Build Types & Flavors
- Scopes
  - Application
  - Activity (Now with Singletons!)
- Testing
  - Espresso
  - Unit Testing

AmazonRelease

AmazonDebug

GoogleDebug

Code Organization

How Dagger manages 6 build variants & 6+ libraries

AmazonBeta

GoogleRelease

GoogleBeta

Application Scoped Modules

- App Module
- Library Module
- Build Type Module
- Flavor Module

App Module Singletons

- Parser (configured GSON)

App Module Singletons

- Parser (configured GSON)
- IO Managers

App Module Singletons

- Parser (configured GSON)
- IO Managers
- Configs (Analytics,AB, E-Commerce)

Example Library Module: E-Commerce

```
@Module
public class ECommModule {


@Provides
@Singleton
public ECommBuilder provideECommBuilder(                    )
```

E-Comm using App Module’s Dep

```
@Module
public class ECommModule {

@Provides
@Singleton
public ECommBuilder provideECommBuilder(ECommConfig config){
return new ECommManagerBuilder().setConfig(config);
}
```

Amazon & Google Flavors

- Amazon Variants needs Amazon E-Commerce

- Google Variants needs to contain Google E-Commerce

How can Dagger help?

E-Comm Qualified Provider

```
@Module public class ECommModule {

@Provides @Singleton
public ECommBuilder provideECommBuilder(ECommConfig config){
return new ECommManagerBuilder().setConfig(config);
}

@Provides @Singleton @Google
public ECommManager providesGoogleEComm (ECommBuilder builder,
GooglePayments googlePayments)
```

E-Comm Qualified Provider

```
@Module public class ECommModule {


@Provides @Singleton
public ECommBuilder provideECommBuilder(ECommConfig config){
return new ECommManagerBuilder().setConfig(config);
}
...
@Provides @Singleton  @Amazon
public ECommManager providesAmazonEComm (ECommBuilder builder,
AmazonPayments amazonPayments)
```

Flavor Module: src/google & src/Amazon

```
@Module public class FlavorModule {









}
```

Flavor Module Provides Non-Qualified E-Comm

```
@Module public class FlavorModule {

@Singleton
@Provides
ECommManager provideECommManager(@Google ECommManager ecomm)
}
```

Note: Proguard strips out the other impl from Jar :-)

Type Module

Brings build specific dependencies/providers in Type Module

Type Module

Brings build specific dependencies/providers in Type Module

- Logging
  - Most logging for Beta Build
  - No-Op Release

Type Module

Brings build specific dependencies/providers in Type Module

- Logging
  - Most logging for Beta Build
  - No-Op Release
- Payments
  - No-Op for debug

Type Module

- Brings build specific dependencies/providers in Type Module
  - Logging
    - Most logging for Beta Build
    - No-Op Release
  - Payments
    - No-Op for debug
  - Device ID
    - Static for Debug

Component Composition

How we combine our modules

Start with Base Component

- Base Component lives in src/main
- Contains inject(T t) for classes & Services that register with Dagger (non flavor/build specific)

```
interface BaseComponent {
  void inject(NYTApplication target);
}
```

Src/Google & Src/Amazon Contain a FlavorComponent

- Create FlavorComponent that inherits from BaseComponent
- Register inject for flavor specific classes
- Anything not in src/flavor that needs component registers here ie:
  - Messaging Service
  - Payment Activity

```
public interface FlavorComponent extends BaseComponent {
  void inject(ADMessaging target);
}
```

App Component debug, beta, release

One for src/debug src/beta src/release

```
public interface ApplicationComponent {
}
```

App Component

Inherits from Flavor Component

```
public interface ApplicationComponent extends FlavorComponent {
}
```

App Component

- Adds @Component @Singleton annotations

```
@Singleton @Component
public interface ApplicationComponent extends FlavorComponent {
}
```

App Component

- Adds modules

```
@Singleton @Component(modules =
{ApplicationModule.class, FlavorModule.class, TypeModule.class,
AnalyticsModule.class, ECommModule.class, PushClientModule.class })

public interface ApplicationComponent extends FlavorComponent {
}
```

Anything registering with App Component

gains access to all providers for the Flavor/Type

Usage of Generated App Component

App Component Factory

```
public class ComponentFactory {
   public AppComponent buildComponent(Application context) {
      return componentBuilder(context).build();
   }

   // We override it for functional tests.
  DaggerApplicationComponent.Builder componentBuilder(Application context) {
      return DaggerApplicationComponent.builder()
                        .applicationModule(new ApplicationModule(context)}
}
```

Component Instance

- NYT Application retains component

```
private void buildComponentAndInject() {
  appComponent = componentFactory().buildComponent(this);
  appComponent.inject(this);
}

public ComponentFactory  componentFactory() {
  return new ComponentFactory();
}
```

Introducing Activity Scope

Activity Component

- Inherits all “provides” from App Component
- Allows you to add “Activity Singletons”
  - 1 Per Activity
  - Many views/fragments within activity can inject same instance

ActivityComponent

```
@Subcomponent(modules = {ActivityModule.class, BundleModule.class})
@ScopeActivity
public interface ActivityComponent {
  void inject(ArticleView view);
}
```

Add to AppComponent:

```
Activitycomponent plusActivityComponent(ActivityModule activityModule);
```

ActivityComponentFactory

```
public final class ActivityComponentFactory {

public static ActivityComponent create(Activity activity) {
              return ((NYTApp)activity.getApplicationContext).getComponent()
               .plusActivityComponent(new ActivityModule(activity));
   }

}
```

Activity Component Injection

```
public void onCreate(@Nullable Bundle savedInstanceState) {

  activityComponent = ActivityComponentFactory.create(this);

  activityComponent.inject(this);
```

Activity Component Modules

Activity Module

- Publish Subjects (mini bus)
- Reactive Text Resizer
- Snack Bar Util

Font Resizing

```
@Provides @ScopeActivity @FontBus
PublishSubject<Integer> provideFontChangeBus() {
	return PublishSubject.create();
}

@Provides @ScopeActivity
FontResizer provideFontResize( @FontBus PublishSubject<Integer> fontBus) {
	return new FontResizer(fontBus);
}
```

Usage of Font Resizer

```
@Inject
FontResizer fontResizer;

private void registerForFontResizing(View itemView) {
   fontResizer.registerResize(itemView);
}
```

Usage of Font Resize “Bus”

```
@Inject
public SectionPresenter(@FontBus PublishSubject<Integer> fontBus) {
   fontBus.subscribe(fontSize -> handleFontHasChanged());
}
```

Dagger helps us inject only what we need

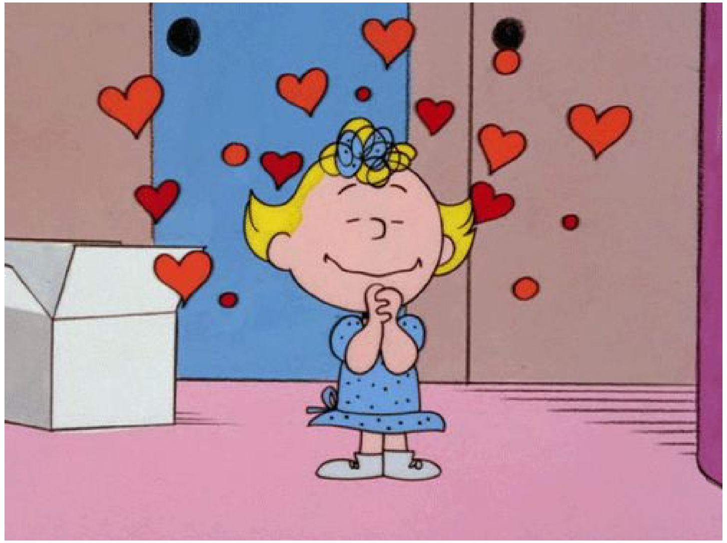
SnackBarUtil

```
@ScopeActivity
public class SnackbarUtil {
    @Inject Activity activity;
public Snackbar makeSnackbar(String txt, int duration) {
      return Snackbar.make(...);}
}
….
```

In some presenter class:

```
public void onError(Throwable error) {
  snackbarUtil.makeSnackbar(SaveHandler.SAVE_ERROR, SHORT).show();
}
```

Bundle Module, A Love Story



Bundle Management

Passing intent arguments to fragments/views is painful

- Need to save state
- Complexity with nested fragments
- Why we not inject intent arguments instead?

Create Bundle Service

```
public class BundleService {
  private final Bundle data;

  public BundleService(Bundle savedState, Bundle intentExtras) {
      data = new Bundle();

      if (savedState != null) {
          data.putAll(savedState);
      }
      if (intentExtras != null) {
          data.putAll(intentExtras);
      }
  }
```

Instantiate Bundle Service in Activity

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
  bundleService = new BundleService(savedInstanceState, getIntent().getExtras
());

//Never have to remember to save instance state again!
protected void onSaveInstanceState(Bundle outState) {
  outState.putAll(bundleService.getAll());
```

Bind Bundle Service to Bundle Module

```
@Provides
@ScopeActivity
public BundleService provideBundleService(Activity context)
{
return ((Bundler) context).getBundleService();
}
```

Provide Individualized Intent Values

```
@Provides
@ScopeActivity
@AssetId
public Long provideArticleId(BundleService bundleService) {
   return bundleService.get(ArticleActivity.ASSET_ID);
}
```

Inject Intent Values Directly into Views & Presenters

```
@Inject
public CommentPresenter(@AssetId String assetId){
//fetch comments for current article
}
```

Old Way

Normally we would have to pass assetId from:

ArticleActivity to

ArticleFragment to

CommentFragment to

CommentView to

CommentPresenter

:-|

Testing

Simple Testing

JUNIT Mockito, AssertJ

```
@Mock
AppPreferences prefs;
@Before public void setUp() {
   inboxPref = new InboxPreferences(prefs);
}
@Test
public void testGetUserChannelPreferencesEmpty() {
   when(prefs.getPreference(IUSER_CHANNELS,emptySet())) .thenReturn(null);
   assertThat(inboxPref.getUserChannel()).isEmpty();
}
```

Testing with Dagger

Dagger BaseTestCase

```
public abstract class BaseTestCase extends TestCase {

protected TestComponent getTestComponent() {
  final ApplicationModule applicationModule = getApplicationModule();
  return Dagger2Helper.buildComponent(
          TestComponent.class,
          applicationModule));}
```

TestComponent

```
@Singleton
@Component(modules = {TestModule.class,ApplicationModule.class, FlavorModule.class, TypeModule.class, AnalyticsModule.class, EcommModule.class, PushModule.class})
public interface TestComponent {
   void inject(WebViewUtilTest test);
```

Dagger Test with Mocks

```
public class WebViewUtilTest extends BaseTestCase {
  @Inject NetworkStatus networkStatus;
  @Inject WebViewUtil webViewUtil;

  protected ApplicationModule getApplicationModule() {
     return new ApplicationModule(application) {
              protected NetworkStatus provideNetworkStatus() {
                    return mock(NetworkStatus.class);
               }
        };
}
```

Dagger Test with Mocks

```
public class WebViewUtilTest extends BaseTestCase {
  @Inject NetworkStatus networkStatus;
  @Inject WebViewUtil webViewUtil;

 …

  @Test
  public void testNoValueOnOffline() throws Exception {
     when(networkStatus.isInternetConnected()).thenReturn(false);
     webViewUtil.getIntentLauncher().subscribe(intent -> {fail("intent was launched");});}
```

Dagger Test with Mocks Gotchas

- Must have provides method
- Must be in module you are explicitly passing into Dagger

Functional/Espresso Testing

NYTFunctionalTestApp

- Creates Component with overridden providers

NYTFunctionalTestApp

- Creates Component with overridden providers
- Mostly no-op since this is global
    - Analytics
    - AB Manager
    - Other Test impls (network, disk)

NYTFunctionalTestApp

- Creates Component with overridden providers
- Mostly no-op since this is global
    - Analytics
    - AB Manager
    - Other Test impls (network, disk)
- Functional test runner uses custom FunctTestApp

NYTFunctionalTestApp

- Creates Component with overridden providers
- Mostly no-op since this is global
    - Analytics
    - AB Manager
    - Other Test impls (network, disk)
- Functional test runner uses custom FunctTestApp
- Test run end to end otherwise

NYTFunctionalTestApp

```
public class NYTFunctionalTestsApp extends NYTApplication {

ComponentFactory componentFactory(Application context) {
return new ComponentFactory() {
protected DaggerApplicationComponent.Builder componentBuilder(Application context) {
return super.componentBuilder(context)
        .applicationModule(new ApplicationModule(NYTFunctionalTestsApp.this) {
            protected ABManager provideABManager() {
              return new NoOpABManager();
}
```

NYTFunctionalTestRunner

```
public class NYTFunctionalTestsRunner extends AndroidJUnitRunner {
  @Override
  public Application newApplication(ClassLoader cl,String className, Context context)
      {  return newApplication(NYTFunctionalTestsApp.class, context);
 }
}
```

Sample Espresso Test

```
@RunWith(AndroidJUnit4.class)
public class MainScreenTests {

  @Test
  public void openMenuAndCheckItems() {
     mainScreen
           .openMenuDialog()
           .assertMenuDialogContainsItemWithText(R.string.dialog_menu_font_resize)
           .assertMenuDialogContainsItemWithText(R.string.action_settings);
  }
```

Questions?

@NYTDevs | developers.nytimes.com

The New York Times Developers

Thank You!

(We’re hiring)

@FriendlyMikhail

@NYTDevs | developers.nytimes.com