Dagger

Jake Wharton


#DV13 #Dagger

Dependency Injection

Dependency Injection

- Every single app has some form of dependency injection

Dependency Injection

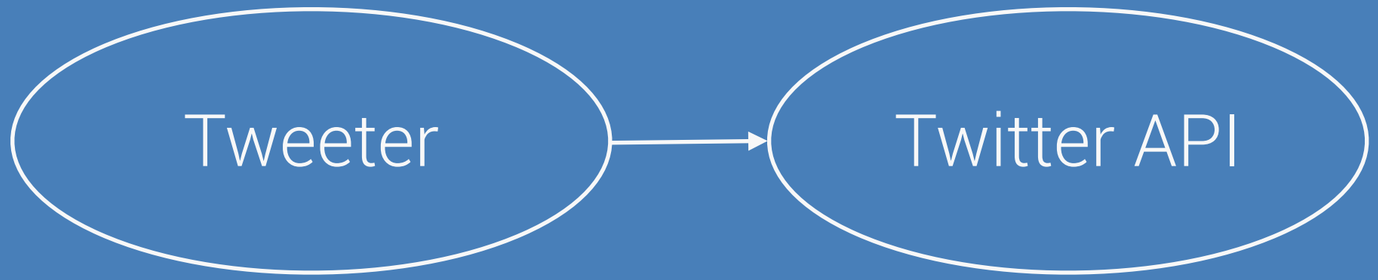
- Every single app has some form of dependency injection
- Separate behavior of something from its required classes

Dependency Injection

- Every single app has some form of dependency injection
- Separate behavior of something from its required classes

Dependency Injection

- Every single app has some form of dependency injection
- Separate behavior of something from its required classes



Dependency Injection

- Every single app has some form of dependency injection
- Separate behavior of something from its required classes

```
public class Tweeter {
  public void tweet(String tweet) {
    TwitterApi api = new TwitterApi();
    api.postTweet("JakeWharton", tweet);
  }
}
```

```
public class Tweeter {
  public void tweet(String tweet) {
    TwitterApi api = new TwitterApi();
    api.postTweet("JakeWharton", tweet);
  }
}

public class TwitterApi {
  public void postTweet(String user, String tweet) {
    OkHttpClient client = new OkHttpClient();
    HttpUrlConnection conn = client.open("...");
    // ... POST blah blah blah ...
  }
}
```

```
public class Tweeter {
  public void tweet(String tweet) {
    TwitterApi api = new TwitterApi();
    api.postTweet("JakeWharton", tweet);
  }
}

public class TwitterApi {
  public void postTweet(String user, String tweet) {
    OkHttpClient client = new OkHttpClient();
    HttpUrlConnection conn = client.open("...");
    // ... POST blah blah blah ...
  }
}

Tweeter tweeter = new Tweeter();
tweeter.tweet("Hello, Devoxx 2013! #DV13 #Dagger");
```

```
public class TwitterApi {
  public void postTweet(String user, String tweet) {
    OkHttpClient client = new OkHttpClient();
    HttpUrlConnection conn = client.open("...");
    // ... POST blah blah blah ...
  }
}
```

```
public class TwitterApi {
  private final OkHttpClient client = new OkHttpClient();

  public void postTweet(String user, String tweet) {
    HttpUrlConnection conn = client.open("...");
    // ... POST blah blah blah ...
  }
}
```

```
public class TwitterApi {
  private final OkHttpClient client;

  public TwitterApi(OkHttpClient client) {
    this.client = client;
  }

  public void postTweet(String user, String tweet) {
    HttpUrlConnection conn = client.open("...");
    // ... POST blah blah blah ...
  }
}
```

```
public class Tweeter {
  public void tweet(String tweet) {
    TwitterApi api = new TwitterApi();
    api.postTweet("JakeWharton", tweet);
  }
}
```

```
public class Tweeter {
  public void tweet(String tweet) {
    TwitterApi api = new TwitterApi(new OkHttpClient());
    api.postTweet("JakeWharton", tweet);
  }
}
```

```
public class Tweeter {
  private final TwitterApi api = new TwitterApi(new OkHttpClient());

  public void tweet(String tweet) {
    api.postTweet("JakeWharton", tweet);
  }
}
```

```
public class Tweeter {
  private final TwitterApi api = new TwitterApi(new OkHttpClient());
  private final String user;

  public Tweeter(String user) {
    this.user = user;
  }

  public void tweet(String tweet) {
    api.postTweet(user, tweet);
  }
}
```

```
Tweeter tweeter = new Tweeter();
tweeter.tweet("Hello, Devoxx 2013! #DV13 #Dagger");
```

```
Tweeter tweeter = new Tweeter("JakeWharton");
tweeter.tweet("Hello, Devoxx 2013! #DV13 #Dagger");
```

```
Tweeter tweeter = new Tweeter("JakeWharton");
tweeter.tweet("Hello, Devoxx 2013! #DV13 #Dagger");
tweeter.tweet("Devoxx 2013 is amazing! #DV13");
tweeter.tweet("mind == blown. #DV13 #Dagger");
tweeter.tweet("OMG you better be at Devoxx!");
tweeter.tweet("#Devoxx #Dagger #Devoxx #Dagger #Devoxx");
tweeter.tweet("Hungover... #DV13");
```

```
Tweeter tweeter = new Tweeter("JakeWharton");
tweeter.tweet("Hello, Devoxx 2013! #DV13 #Dagger");
tweeter.tweet("Devoxx 2013 is amazing! #DV13");
tweeter.tweet("mind == blown. #DV13 #Dagger");
tweeter.tweet("OMG you better be at Devoxx!");
tweeter.tweet("#Devoxx #Dagger #Devoxx #Dagger #Devoxx");
tweeter.tweet("Hungover... #DV13");
```

```
Timeline timeline = new Timeline("JakeWharton");
timeline.loadMore(20);
for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
```

```java
public class Timeline {
  private final List<Tweet> tweetCache = new ArrayList<>();
  private final TwitterApi api = new TwitterApi(new OkHttpClient());
  private final String user;

  public Timeline(String user) {
    this.user = user;
  }

  public List<Tweet> get() { /* ... */ }
  public void loadMore(int amount) { /* ... */ }
}
```

```
public class Timeline {
  private final List<Tweet> tweetCache = new ArrayList<>();
  private final TwitterApi api;
  private final String user;

  public Timeline(TwitterApi api, String user) {
    this.api = api;
    this.user = user;
  }

  public List<Tweet> get() { /* ... */ }
  public void loadMore(int amount) { /* ... */ }
}
```

```
public class Tweeter {
  private final TwitterApi api = new TwitterApi(new OkHttpClient());
  private final String user;

  public Tweeter(String user) {
    this.user = user;
  }

  public void tweet(String tweet) {
    api.postTweet(user, tweet);
  }
}
```

```
public class Tweeter {
  private final TwitterApi api;
  private final String user;

  public Tweeter(TwitterApi api, String user) {
    this.api = api;
    this.user = user;
  }

  public void tweet(String tweet) {
    api.postTweet(user, tweet);
  }
}
```

```
Tweeter tweeter = new Tweeter("JakeWharton");
tweeter.tweet("Hello, Devoxx 2013!");

Timeline timeline = new Timeline("JakeWharton");
timeline.loadMore(20);
for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
```

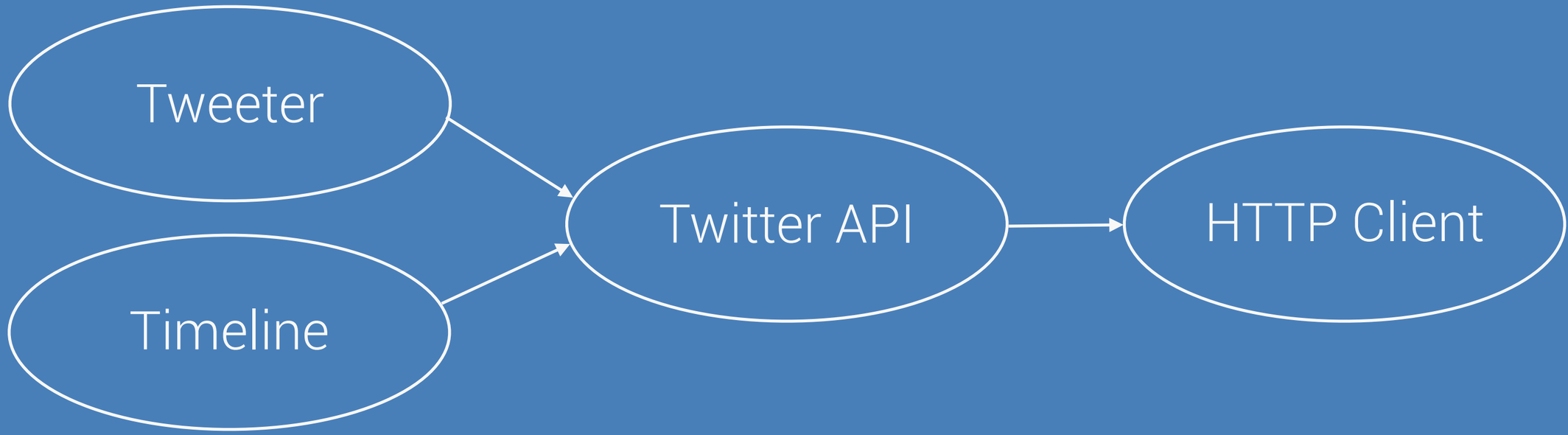
```
OkHttpClient client = new OkHttpClient();
TwitterApi api = new TwitterApi(client);
String user = "JakeWharton";

Tweeter tweeter = new Tweeter(api, user);
tweeter.tweet("Man, this takes a lot of code!");

Timeline timeline = new Timeline(api, user);
timeline.loadMore(20);
for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
```

Dependency Injection

- Every single app has some form of dependency injection
- Separate the behavior from the classes required to perform it

Dependency Injection

- Every single app has some form of dependency injection
- Separate the behavior from the classes required to perform it



Dependency Injection

- Every single app has some form of dependency injection
- Separate the behavior from the classes required to perform it

- How do we avoid the boilerplate that comes with the pattern?

We used Guice

We used Guice

- All of our Java services heavily use Guice

We used Guice

- All of our Java services heavily use Guice
- Powerful, dynamic, well-tested, wide-spread, etc...

We used Guice

- All of our Java services heavily use Guice
- Powerful, dynamic, well-tested, wide-spread, etc...
- Canonical standard for dependency injection

We used Guice

- All of our Java services heavily use Guice
- Powerful, dynamic, well-tested, wide-spread, etc...
- Canonical standard for dependency injection
- Configuration problems fail at runtime

We used Guice

- All of our Java services heavily use Guice
- Powerful, dynamic, well-tested, wide-spread, etc...
- Canonical standard for dependency injection
- Configuration problems fail at runtime
- Slow initialization, slow injection, memory problems

Romain Guy
@romainguy

Follow

Startup time slowed down by 30% because of an injection framework. That's what Chet and I warned against at #Devoxx m.facebook.com/notes/facebook...

Reply Retweet Favorite More

“ObjectGraph” Goals

"ObjectGraph" Goals

- Static analysis of all dependencies and injections

"ObjectGraph" Goals

- Static analysis of all dependencies and injections
- Fail as early as possible (compile-time, not runtime)

"ObjectGraph" Goals

- Static analysis of all dependencies and injections
- Fail as early as possible (compile-time, not runtime)
- Eliminate reflection on methods and annotations at runtime

"ObjectGraph" Goals

- Static analysis of all dependencies and injections
- Fail as early as possible (compile-time, not runtime)
- Eliminate reflection on methods and annotations at runtime
- Have negligible memory impact

“ObjectGraph” Development

"ObjectGraph" Development

- ~5 weeks of heads-down work by Jesse Wilson

"ObjectGraph" Development

- ~5 weeks of heads-down work by Jesse Wilson
- Bob Lee served as technical advisor

"ObjectGraph" Development

- ~5 weeks of heads-down work by Jesse Wilson
- Bob Lee served as technical advisor
- “Giant" boolean switch in our applications

"ObjectGraph" Development

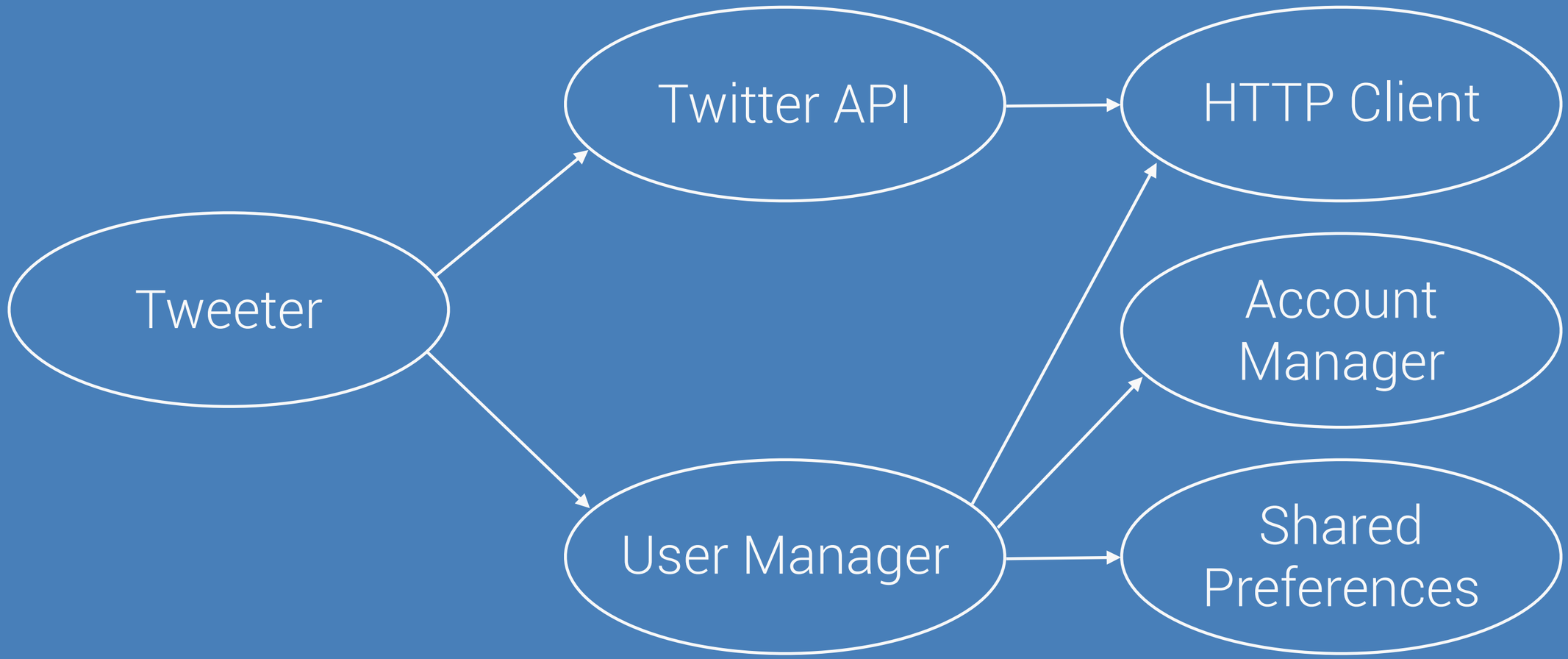
- ~5 weeks of heads-down work by Jesse Wilson
- Bob Lee served as technical advisor
- "Giant" boolean switch in our applications
- 2 weeks after, dropped Guice completely

"ObjectGraph" Development

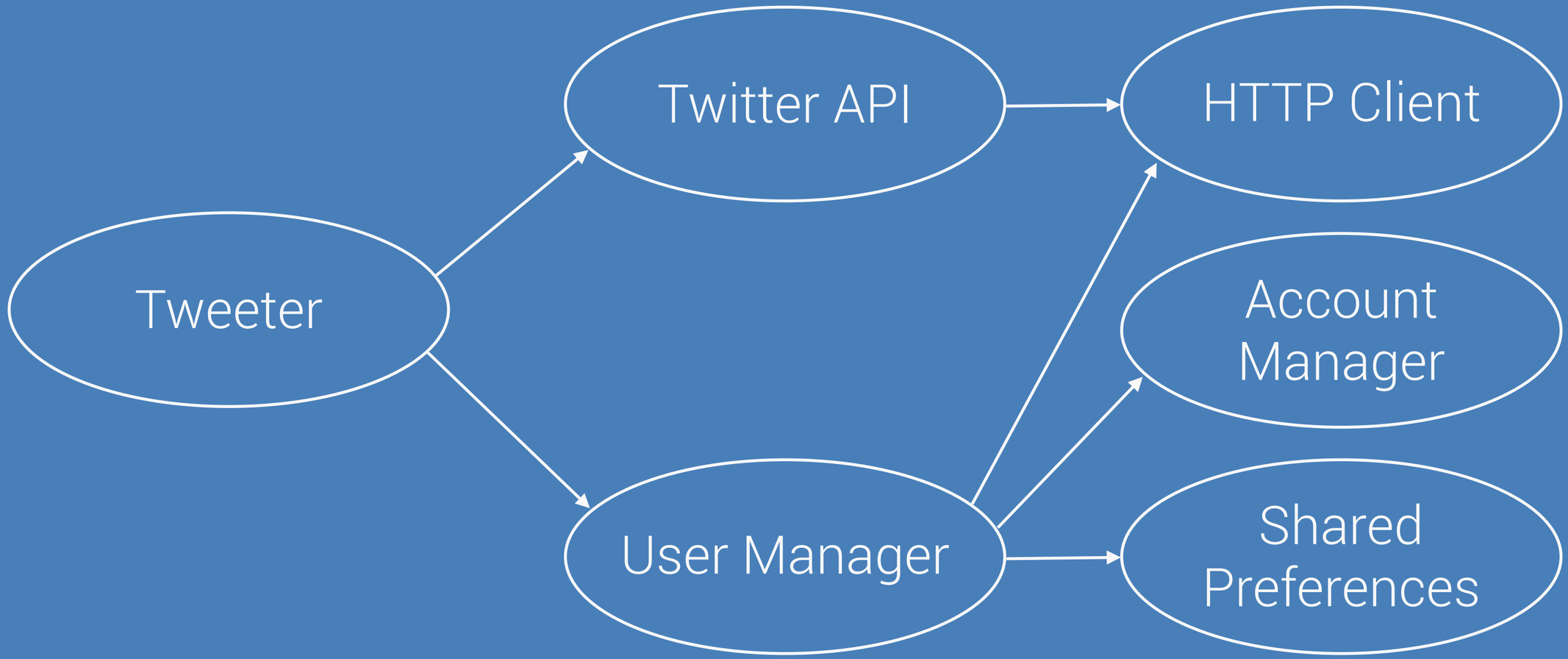
- ~5 weeks of heads-down work by Jesse Wilson
- Bob Lee served as technical advisor
- "Giant" boolean switch in our applications
- 2 weeks after, dropped Guice completely
- Renamed to Dagger before first release

Dagger

Dagger


Twitter API
HTTP Client
Tweeter
Account Manager
User Manager
Shared Preferences

DAG-er


Twitter API
HTTP Client
Tweeter
Account Manager
User Manager
Shared Preferences

Dagger

Dagger

- `ObjectGraph`: central dependency manager and injector

Dagger

- `ObjectGraph`: central dependency manager and injector
- `@Module` + `@Provides`: mechanism for providing dependencies

Dagger

- `ObjectGraph`: central dependency manager and injector
- `@Module` + `@Provides`: mechanism for providing dependencies
- `@Inject`: mechanism for requesting dependencies

Dagger

- `ObjectGraph`: central dependency manager and injector
- `@Module` + `@Provides`: mechanism for providing dependencies
- `@Inject`: mechanism for requesting dependencies
- Plus some other sugar, magic, and conventions

Providing Dependencies

Providing Dependencies

- Modules are classes that provide dependencies

Providing Dependencies

- Modules are classes that provide dependencies
- `@Module` annotation on the class

Providing Dependencies

- Modules are classes that provide dependencies
- `@Module` annotation on the class
- `@Provider` annotation on a method indicates that its return type is a dependency

```
public class NetworkModule {

  public OkHttpClient provideOkHttpClient() {
    return new OkHttpClient();
  }


  public TwitterApi provideTwitterApi(OkHttpClient client) {
    return new TwitterApi(client);
  }

}
```

```
@Module
public class NetworkModule {

  public OkHttpClient provideOkHttpClient() {
    return new OkHttpClient();
  }


  public TwitterApi provideTwitterApi(OkHttpClient client) {
    return new TwitterApi(client);
  }

}
```

```
@Module
public class NetworkModule {
  @Provides
  public OkHttpClient provideOkHttpClient() {
    return new OkHttpClient();
  }

  @Provides
  public TwitterApi provideTwitterApi(OkHttpClient client) {
    return new TwitterApi(client);
  }

}
```

```
@Module
public class NetworkModule {
  @Provides @Singleton
  public OkHttpClient provideOkHttpClient() {
    return new OkHttpClient();
  }

  @Provides @Singleton
  public TwitterApi provideTwitterApi(OkHttpClient client) {
    return new TwitterApi(client);
  }

}
```

Providing Dependencies

Providing Dependencies

`OkHttpClient`

`TwitterApi`

Providing Dependencies

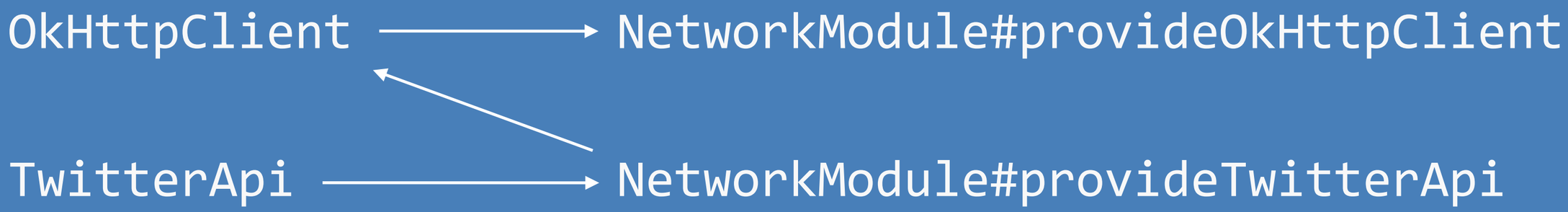
`OkHttpClient`	`NetworkModule#provideOkHttpClient`
`TwitterApi`	`NetworkModule#provideTwitterApi`

Providing Dependencies

`OkHttpClient` → `NetworkModule#provideOkHttpClient`

`TwitterApi` → `NetworkModule#provideTwitterApi`

Providing Dependencies



Providing Dependencies

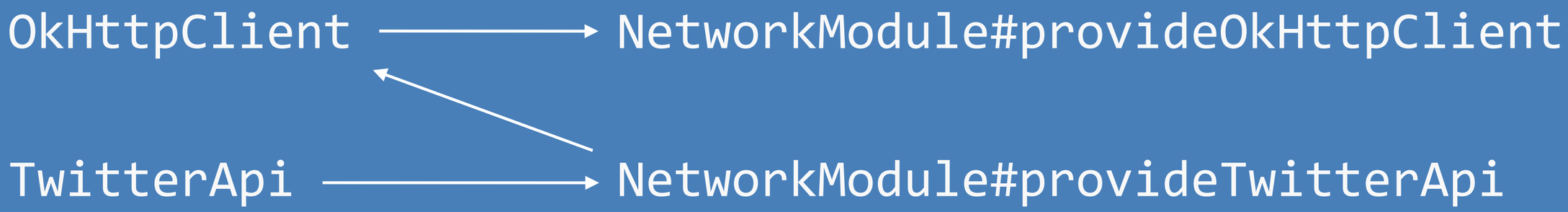
- Modules are classes that provide dependencies
- `@Module` annotation on the class provides static analysis hints
- `@Provider` annotation on a method indicates that its return type is a dependency

Providing Dependencies

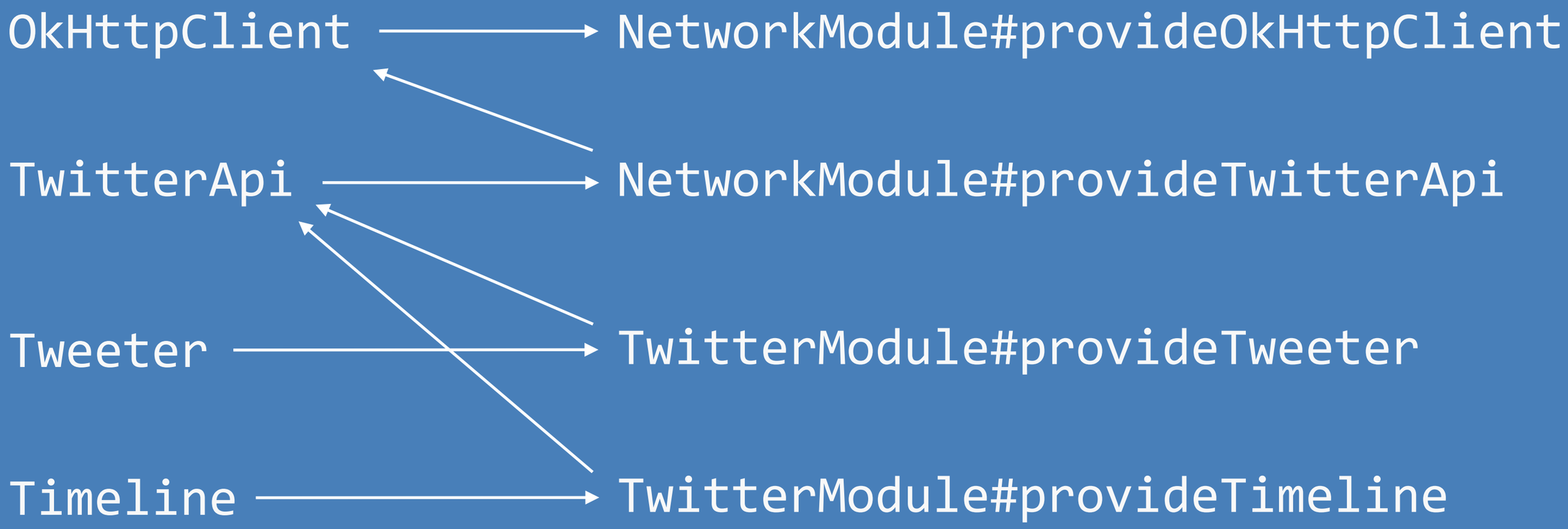
- Modules are classes that provide dependencies
- `@Module` annotation on the class provides static analysis hints
- `@Provider` annotation on a method indicates that its return type is a dependency
- Designed to be composed together

```
@Module
public class TwitterModule {
  private final String user;

  public TwitterModule(String user) {
    this.user = user;
  }

  @Provides @Singleton
  public Tweeter provideTweeter(TwitterApi api) {
    return new Tweeter(api, user);
  }

  @Provides @Singleton
  public Timeline provideTimeline(TwitterApi api) {
    return new Timeline(api, user);
  }
}
```

Providing Dependencies



Providing Dependencies



Requesting Dependencies

Requesting Dependencies

- `@Inject` annotation required

Requesting Dependencies

- `@Inject` annotation required
- Field injection and constructor injection only

Constructor Injection

Constructor Injection

- `@Inject` on a single constructor

Constructor Injection

- `@Inject` on a single constructor
- All constructor arguments are dependencies

Constructor Injection

- `@Inject` on a single constructor
- All constructor arguments are dependencies
- Dependencies can be stored in `private` and `final` fields

```
public class TweeterApp {
  private final Tweeter tweeter;
  private final Timeline timeline;

  @Inject
  public TweeterApp(Tweeter tweeter, Timeline timeline) {
    this.tweeter = tweeter;
    this.timeline = timeline;
  }

  // ...
}
```

Constructor Injection

- `@Inject` on a single constructor
- All constructor arguments are dependencies
- Dependencies can be stored in `private` and `final` fields

Constructor Injection

- `@Inject` on a single constructor
- All constructor arguments are dependencies
- Dependencies can be stored in `private` and `final` fields
- Dagger must create the object

Constructor Injection

- `@Inject` on a single constructor
- All constructor arguments are dependencies
- Dependencies can be stored in `private` and `final` fields
- Dagger must create the object
- `@Provides` not required for downstream injection

Field Injection

Field Injection

- `@Inject` on fields which are dependencies

Field Injection

- `@Inject` on fields which are dependencies
- Field may not be `private` or `final`

```
public class TweeterApp {
  @Inject Tweeter tweeter;
  @Inject Timeline timeline;

  // ...
}
```

```
public class TweeterActivity extends Activity {
  @Inject Tweeter tweeter;
  @Inject Timeline timeline;

  // ...
}
```

Field Injection

- `@Inject` on fields which are dependencies
- Field may not be `private` or `final`

Field Injection

- `@Inject` on fields which are dependencies
- Field may not be `private` or `final`
- Injection happens after the object is “alive”

Field Injection

- `@Inject` on fields which are dependencies
- Field may not be `private` or `final`
- Injection happens after the object is “alive”
- Object often must be responsible for injecting itself

The `ObjectGraph`

The `ObjectGraph`

- Central module (dependency) manager

The `ObjectGraph`

- Central module (dependency) manager
- Injector

```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new TwitterModule("JakeWharton")
);
```

```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new TwitterModule("JakeWharton")
);


// Using constructor injection:
TweeterApp app = og.get(TweeterApp.class);
```

```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new TwitterModule("JakeWharton")
);


// Using constructor injection:
TweeterApp app = og.get(TweeterApp.class);

// Using field injection:
TweeterApp app = new TweeterApp();
og.inject(app);
```

The `ObjectGraph`

- Central module (dependency) manager
- Injector

The `ObjectGraph`

- Central module (dependency) manager
- Injector
- Can be extended to create “scopes”

```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new TwitterModule("JakeWharton")
);
```

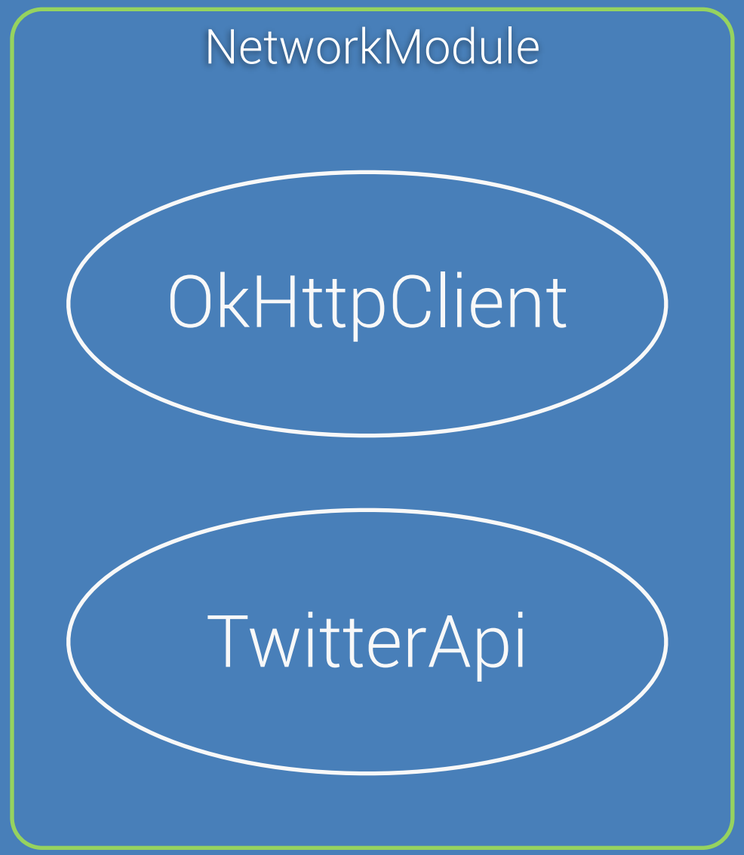
```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule()
);
```

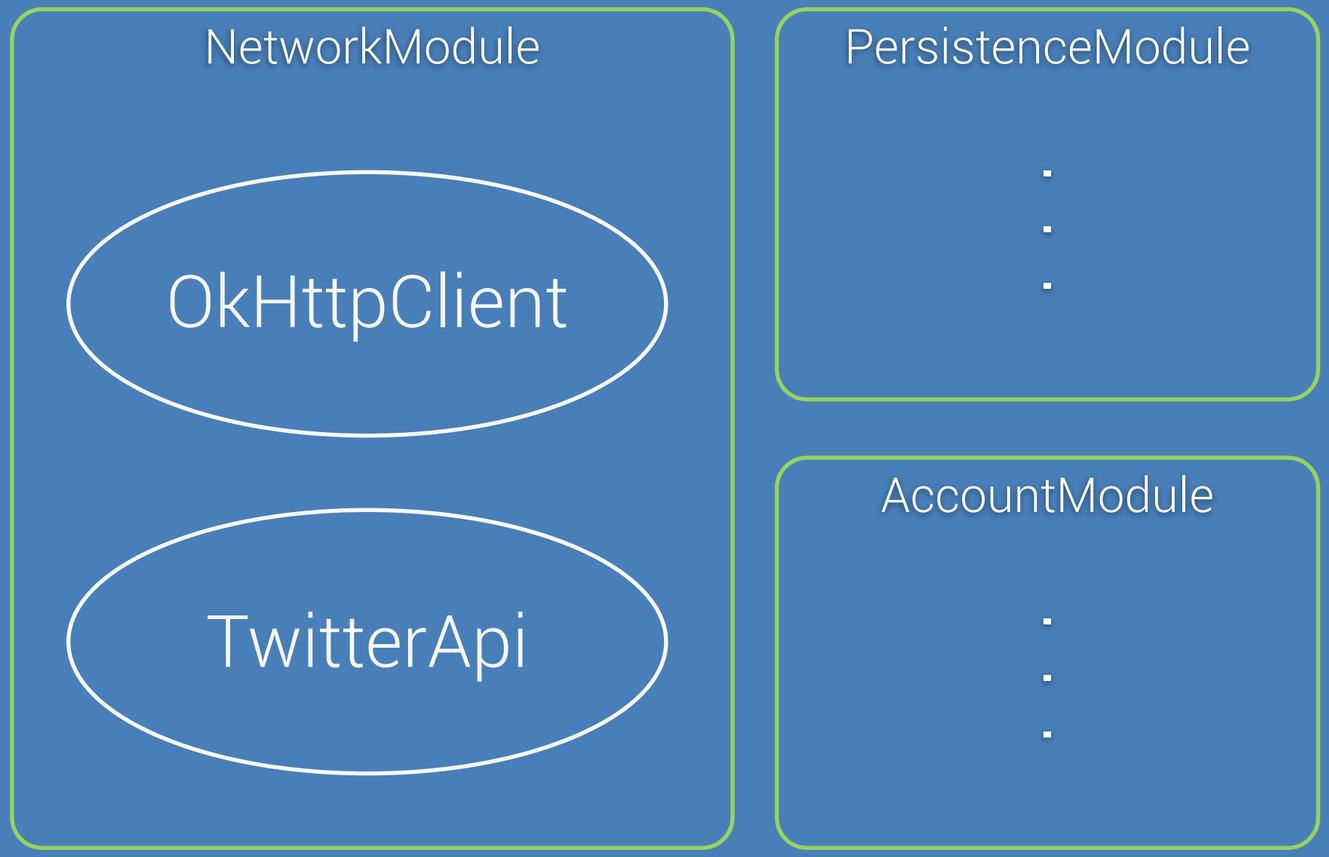
```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new PersistenceModule(),
  new AccountModule()
);
```

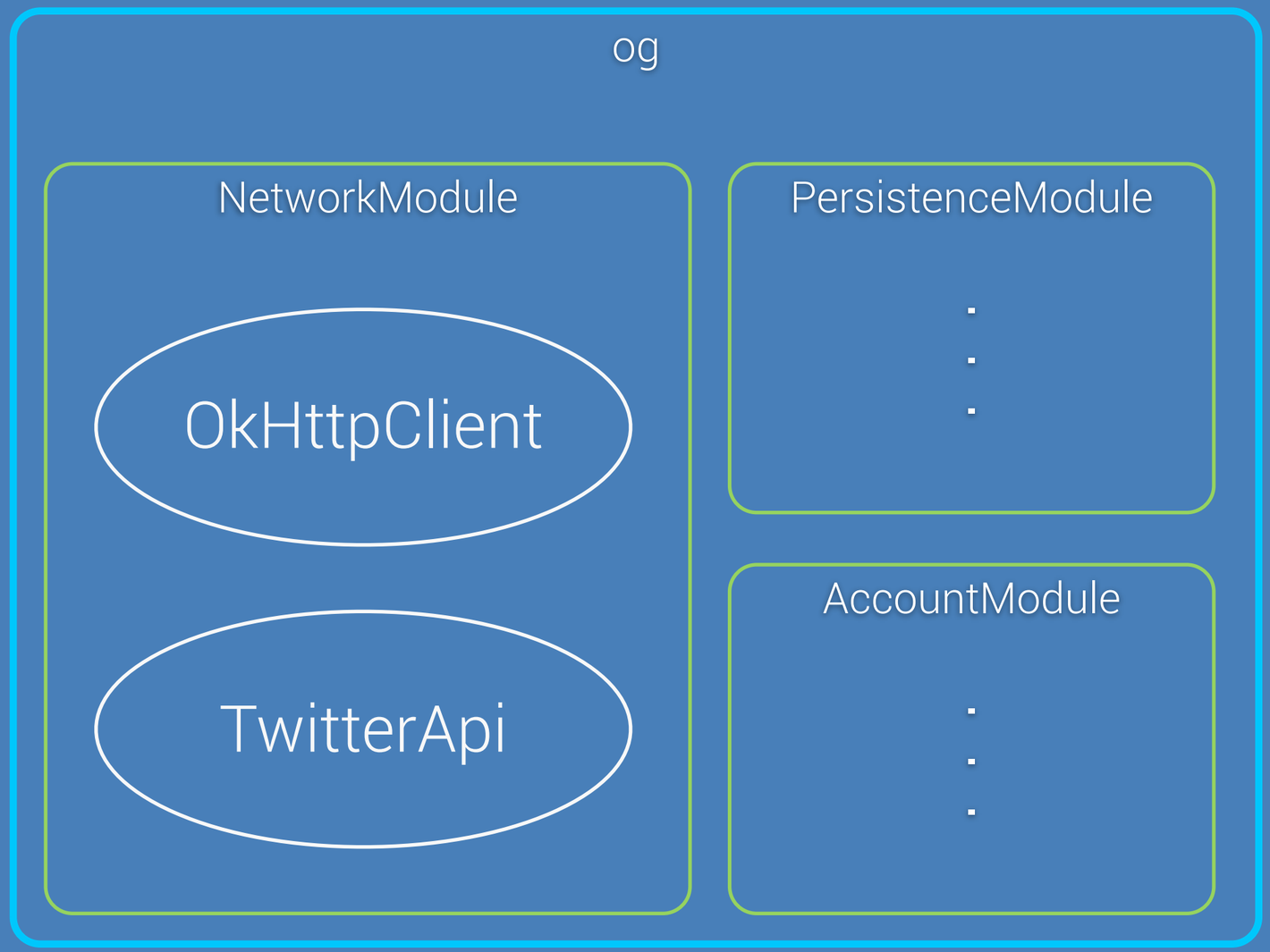
```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new PersistenceModule(),
  new AccountModule()
);
// Inject app things using ‘og’...
```

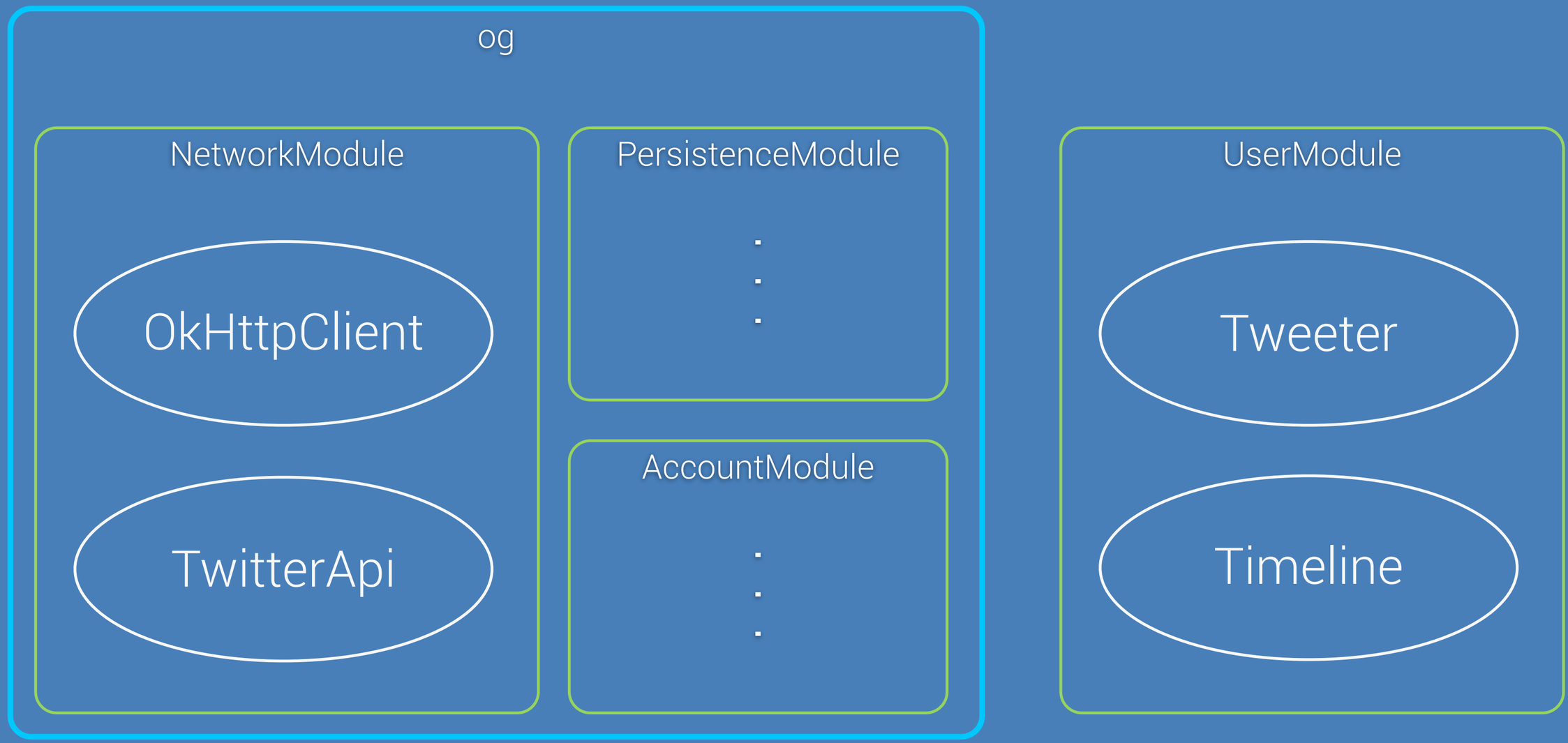
```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new PersistenceModule(),
  new AccountModule()
);
// Inject app things using ‘og’...

// Later...
String user = "JakeWharton";
```

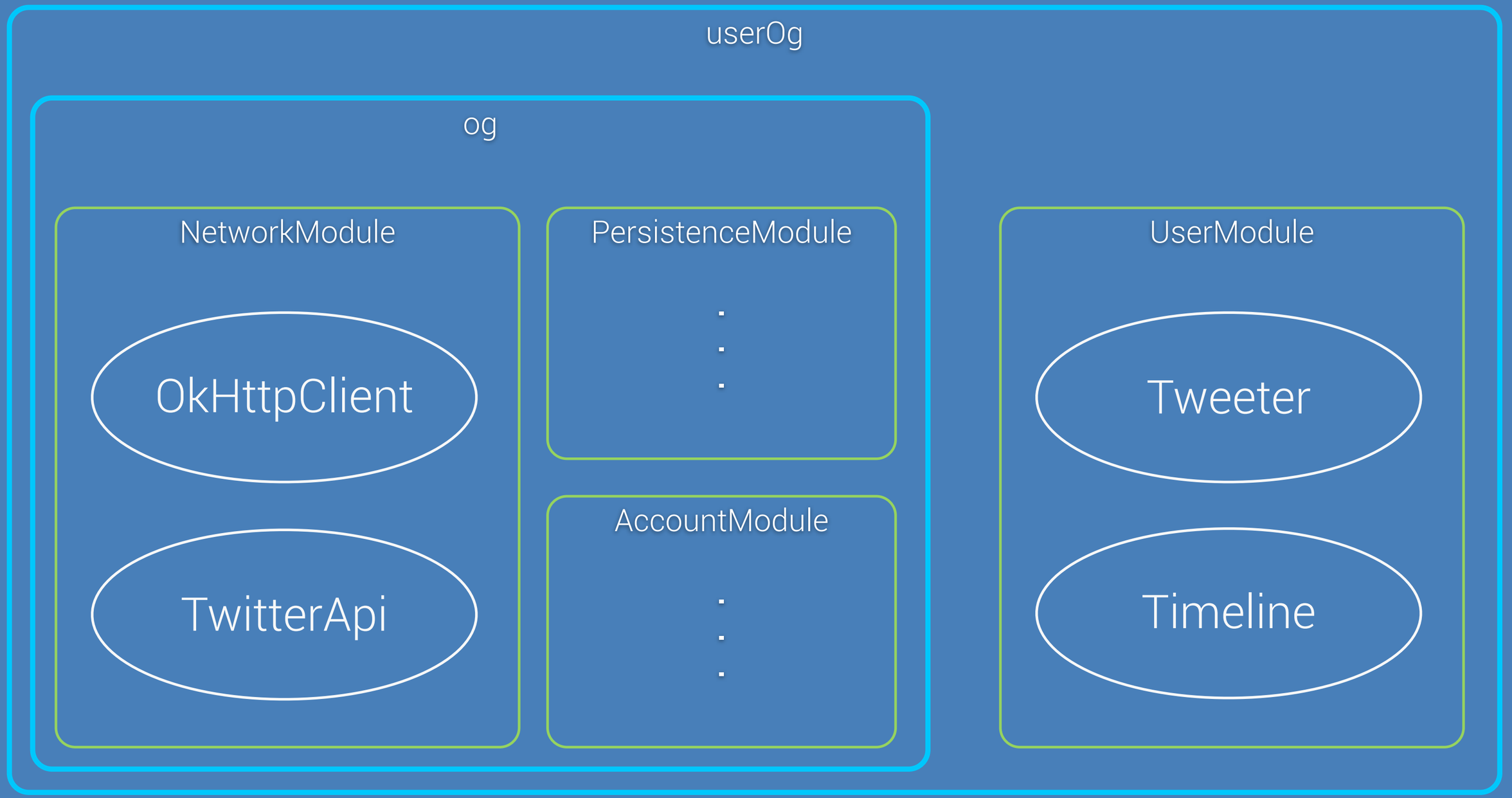
```
ObjectGraph og = ObjectGraph.create(
  new NetworkModule(),
  new PersistenceModule(),
  new AccountModule()
);
// Inject app things using ‘og’...

// Later...
String user = "JakeWharton";
ObjectGraph userOg = og.plus(new TwitterModule(user));

// Inject user things using ‘userOg’...
```

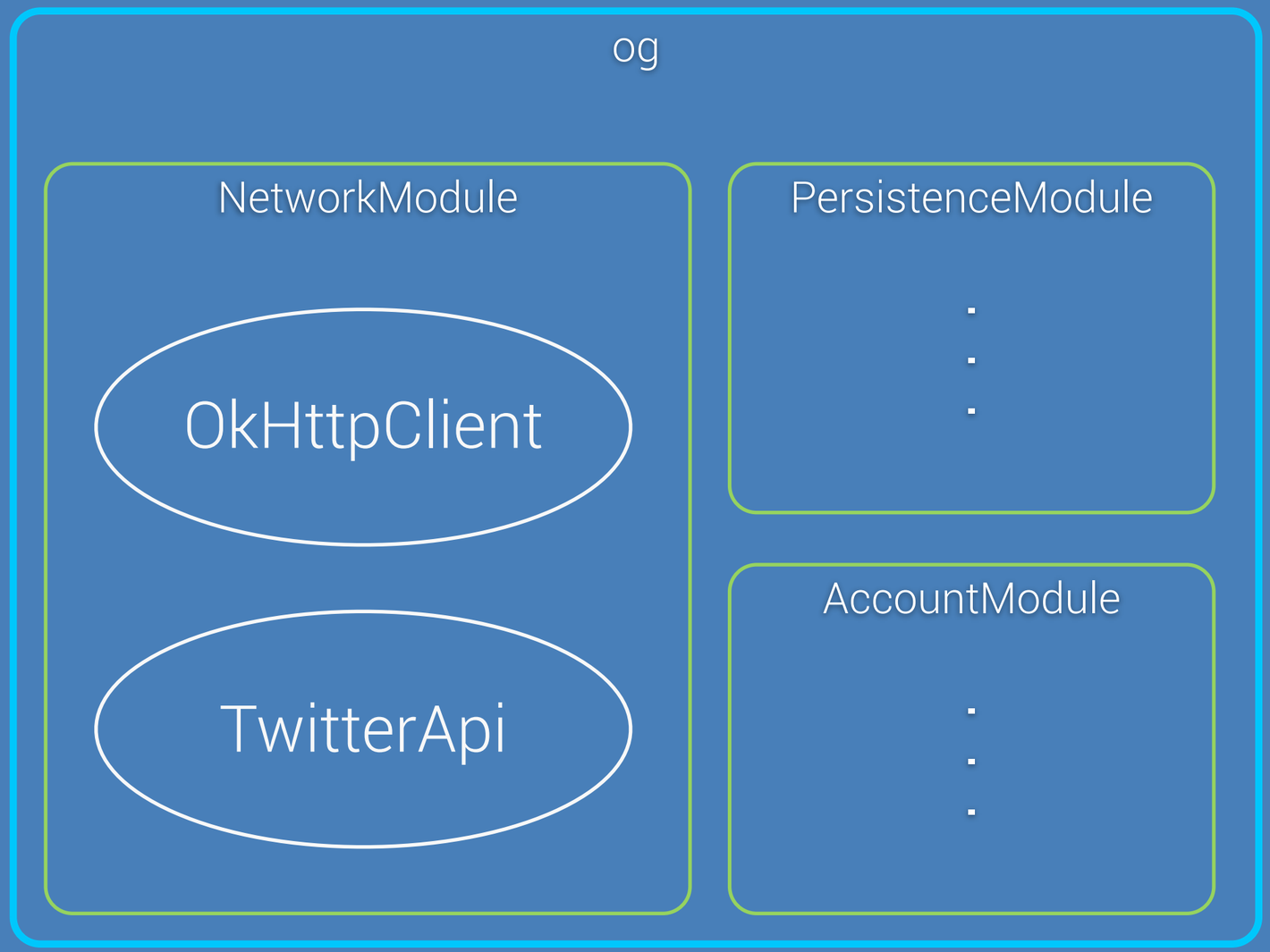

NetworkModule
OkHttpClient
TwitterApi

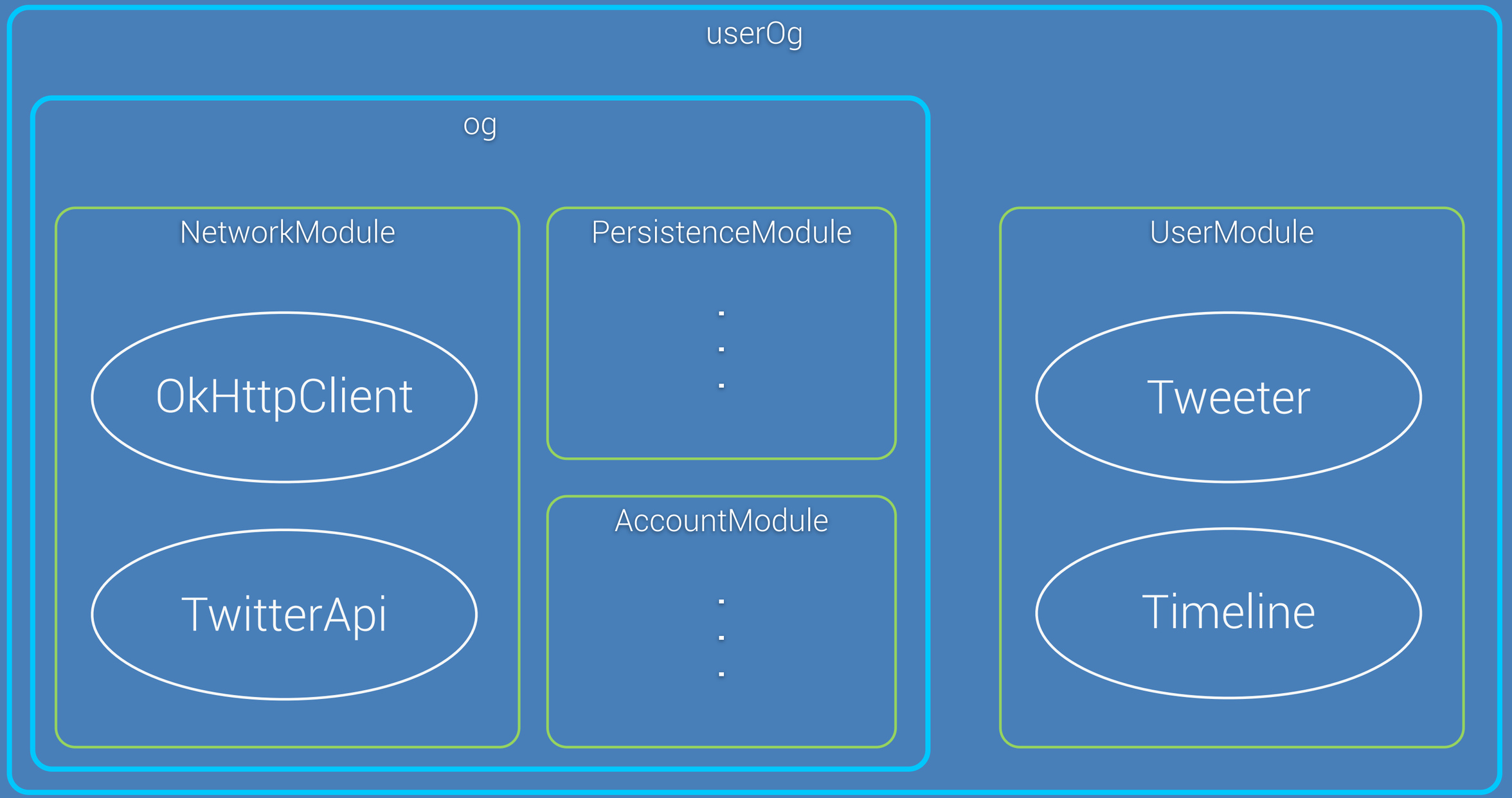

NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
.
.
.
AccountModule
.
.
.


og
NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
.
.
.
AccountModule
.
.
.


og
NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
AccountModule
UserModule
Tweeter
Timeline


userOg
og
NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
AccountModule
UserModule
Tweeter
Timeline


og
NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
.
.
.
AccountModule
.
.
.


userOg
og
NetworkModule
OkHttpClient
TwitterApi
PersistenceModule
AccountModule
UserModule
Tweeter
Timeline

Android

Android

- Entry objects are managed objects constructed by OS

Android

- Entry objects are managed objects constructed by OS
- Multiple services, activities, etc. required shared state

Android

- Entry objects are managed objects constructed by OS
- Multiple services, activities, etc. required shared state
- Platform is *very* difficult to test

Android

- Entry objects are managed objects constructed by OS
- Multiple services, activities, etc. required shared state
- Platform is *very* difficult to test
- Build system allows for dynamic flavors and build types

Android

- Entry objects are managed objects constructed by OS
- Multiple services, activities, etc. required shared state
- Platform is *very* difficult to test
- Build system allows for dynamic flavors and build types
- Many libraries require keeping singletons or long-lived objects

The `ObjectGraph`

- Central module (dependency) manager
- Injector
- Can be extended to create “scopes”

The `ObjectGraph`

- Central module (dependency) manager
- Injector
- Can be extended to create “scopes”
- Eagerly or lazily created on the Application by convention

```
public class ExampleApp extends Application {
  private ObjectGraph objectGraph;

  @Override public void onCreate() {
    super.onCreate();

    objectGraph = ObjectGraph.create(
      new ExampleModule()
    );
  }

  public ObjectGraph getObjectGraph() {
    return objectGraph;
  }
}
```

```
public class ExampleActivity extends Activity {
  @Inject Foo foo;
  @Inject Bar bar;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ExampleApp app = (ExampleApp) getApplication();
    app.getObjectGraph().inject(this);

    // ...
  }
}
```

Listing Injection Points

Listing Injection Points

- All injection points must be listed on a module

Listing Injection Points

- All injection points must be listed on a module
- Used for aggressive static analysis

Listing Injection Points

- All injection points must be listed on a module
- Used for aggressive static analysis
- Potentially not needed for full compilation...

Listing Injection Points

- All injection points must be listed on a module
- Used for aggressive static analysis
- Potentially not needed for full compilation...
- ...but absolutely required for incremental compilation

```
@Module
public class ExampleModule {
  @Provides @Singleton Foo provideFoo() {
    return new Foo();
  }

  @Provides @Singleton Bar provideBar() {
    return new Bar();
  }
}
```

```
@Module(
  injects = {
    ExampleActivity.class
  }
)
public class ExampleModule {
  @Provides @Singleton Foo provideFoo() {
    return new Foo();
  }

  @Provides @Singleton Bar provideBar() {
    return new Bar();
  }
}
```

Use in Android

Use in Android

- Root `ObjectGraph` on Application

Use in Android

- Root `ObjectGraph` on Application
- Activities, services, fragments, views obtain and inject

Use in Android

- Root `ObjectGraph` on Application
- Activities, services, fragments, views obtain and inject
- Modules and their “injects” segment parts of your app

Use in Android: `.plus()`

Use in Android: `.plus()`

- Extend the graph with additional modules

Use in Android: `.plus()`

- Extend the graph with additional modules
- Allows creating “scopes” of dependencies

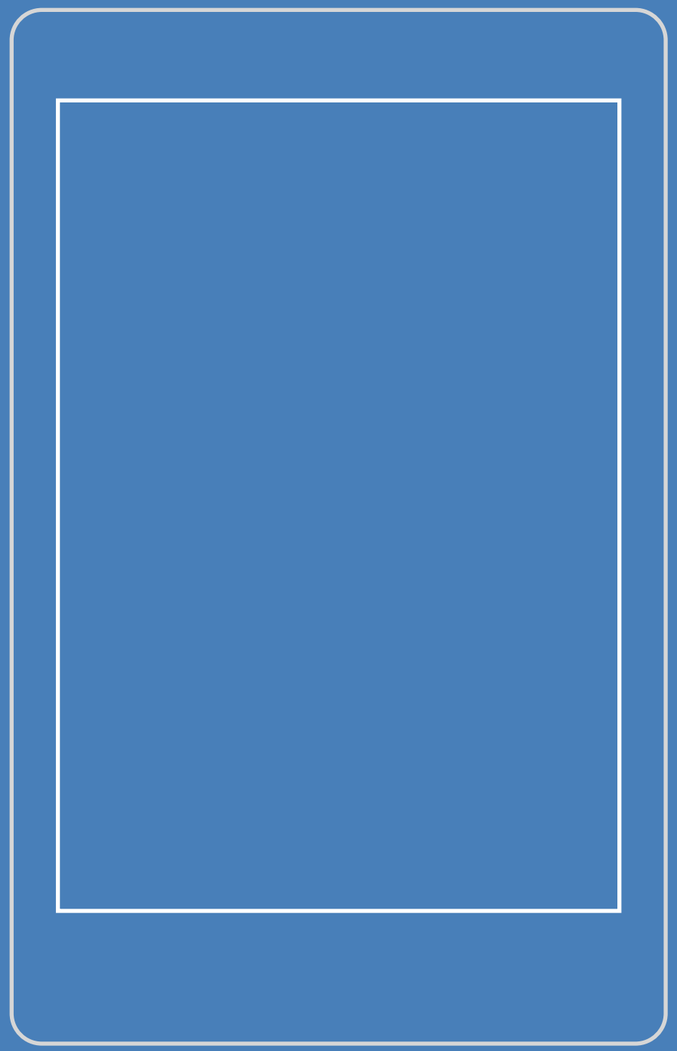
```
public class TweeterApp extends Application {
  private ObjectGraph objectGraph;

  @Override public void onCreate() {
    super.onCreate();

    objectGraph = ObjectGraph.create(
      new NetworkModule(),
      new PersistenceModule(),
      new AccountModule(),
    );
  }

  public ObjectGraph getObjectGraph() {
    return objectGraph;
  }
}
```

```java
public class LandingActivity extends Activity {
  @Inject UserManager userManager;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ExampleApp app = (ExampleApp) getApplication();
    app.getObjectGraph().inject(this);

    if (userManager.hasUser()) {
      // Start TimelineActivity, finish, return...
    }

    // Show log in / sign up...
  }
}
```

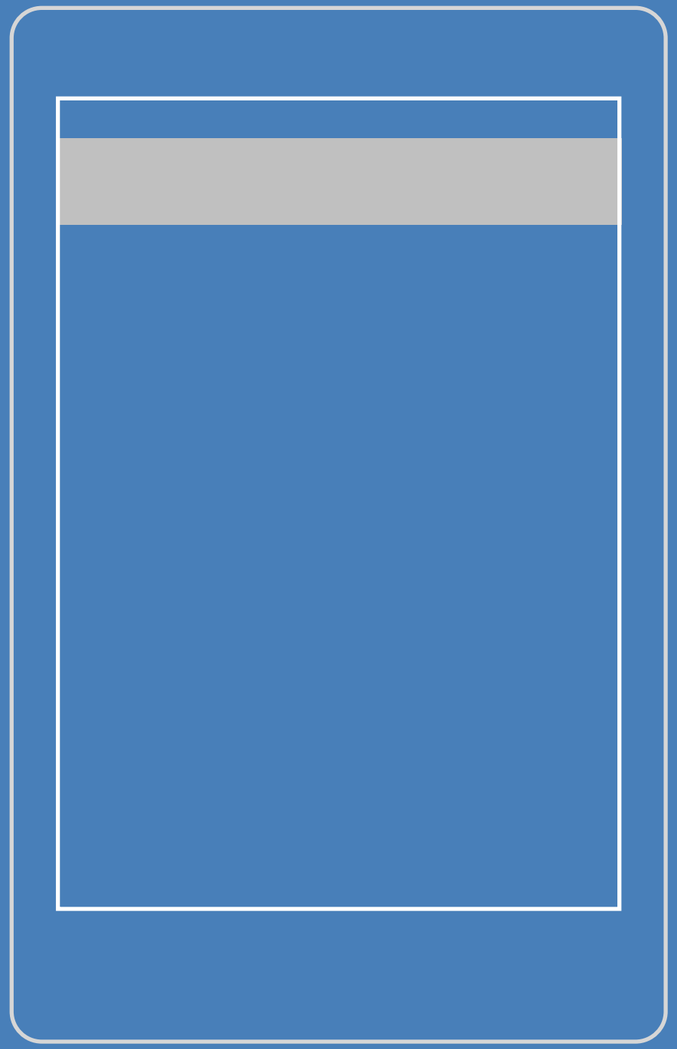
```
public class TimelineActivity extends Activity {
  @Inject Timeline timeline;
  @Inject Tweeter tweeter;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ExampleApp app = (ExampleApp) getApplication();
    ObjectGraph og = app.getObjectGraph();

    String user = og.get(UserManager.class).getUser();
    // TODO if user == null, finish and start LandingActivity...
    og.plus(new TwitterModule(user)).inject(this);

    // Set up timeline and tweeter UI...
  }
}
```

Use in Android: `.plus()`

Use in Android: `.plus()`



Use in Android: `.plus()`



Use in Android: `.plus()`

Use in Android: `.plus()`

```
@Module(
  injects = {
    SignUpView.class, ForgotPasswordView.class
  }
)
public class SignUpModule {
  private final SignUpActivity signUpActivity;

  public SignUpModule(SignUpActivity signUpActivity) {
    this.signUpActivity = signUpActivity;
  }

  @Provides @Singleton ActionBar provideActionBar() {
    return signUpActivity.getActionBar();
  }
}
```

```
public class SignUpActivity class Activity {
  private final ObjectGraph childOg;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ExampleApp app = (ExampleApp) getApplication();
    ObjectGraph og = app.getObjectGraph();

    childOg = og.plus(new SignUpModule(this));

    // Content set up and injection...
  }
}
```

```
public class SignUpView class LinearLayout {
  @Inject ActionBar actionBar;

  public SignUpView(Context context, AttributeSet attrs) {
    super(context, attrs);
    // Note: Injection happens by parent activity.
  }

  @Override public void onAttachedToWindow() {
    super.onAttachedToWindow();
    actionBar.setTitle(R.string.sign_up);
  }
}
```

Use in Android: `overrides`

Use in Android: `overrides`

- Modules whose dependencies override others

Use in Android: `overrides`

- Modules whose dependencies override others
- Only apply in the same `ObjectGraph`

Use in Android: `overrides`

- Modules whose dependencies override others
- Only apply in the same `ObjectGraph`
- Useful for customizing flavors and testing

```
public class Timeline {
  private final List<Tweet> tweetCache = new ArrayList<>();
  private final TwitterApi api;
  private final String user;

  public Timeline(TwitterApi api, String user) {
    this.api = api;
    this.user = user;
  }

  public List<Tweet> get() { /* ... */ }
  public void loadMore(int amount) { /* ... */ }
}
```

```
public class MockTimeline extends Timeline {
  private final List<Tweet> tweets = new ArrayList<>();

  @Inject MockTimeline() {
    tweets.add(new Tweet("MockUser", "Hello, mock data!");
  }

  @Override public List<Tweet> get() {
    return tweets;
  }

  public void addTweet(Tweet tweet) {
    tweets.add(new Tweet("MockUser", tweet), 0);
  }

  @Override public void loadMore(int amount) {}
}
```

```
public class Tweeter {
  private final TwitterApi api;
  private final String user;

  public Tweeter(TwitterApi api, String user) {
    this.api = api;
    this.user = user;
  }

  public void tweet(String tweet) {
    api.postTweet(user, tweet);
  }
}
```

```
public class MockTweeter extends Tweeter {
  private final MockTimeline timeline;

  @Inject MockTweeter(MockTimeline timeline) {
    this.timeline = timeline;
  }

  @Override public void tweet(String tweet) {
    timeline.addTweet(tweet);
  }
}
```

```
@Module(
  overrides = true
)
public class MockTwitterModule {
  @Provides @Singleton
  public Tweeter provideTweeter(MockTweeter mockTweeter) {
    return mockTweeter;
  }

  @Provides @Singleton
  public Timeline provideTimeline(MockTimeline mockTimeline) {
    return mockTimeline;
  }
}
```

```
ObjectGraph og = ObjectGraph.create(
    new NetworkModule(),
    new TwitterModule("JakeWharton"),

    new MockTwitterModule(),
);
```

```
Timeline timeline = /* injected */
Tweeter tweeter = /* injected */
```

```
Timeline timeline = /* injected */
Tweeter tweeter = /* injected */


for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
//  MockUser: Hello, mock data!
```

```
Timeline timeline = /* injected */
Tweeter tweeter = /* injected */

for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
//  MockUser: Hello, mock data!

tweeter.tweet("Showing off more mock data! #DV13 #Dagger");
```

```
Timeline timeline = /* injected */
Tweeter tweeter = /* injected */

for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
//  MockUser: Hello, mock data!

tweeter.tweet("Showing off more mock data! #DV13 #Dagger");

for (Tweet tweet : timeline.get()) {
  System.out.println(tweet);
}
// MockUser: Showing off more mock data! #DV13 #Dagger
// MockUser: Hello, mock data!
```

Mock Mode

Mock Mode

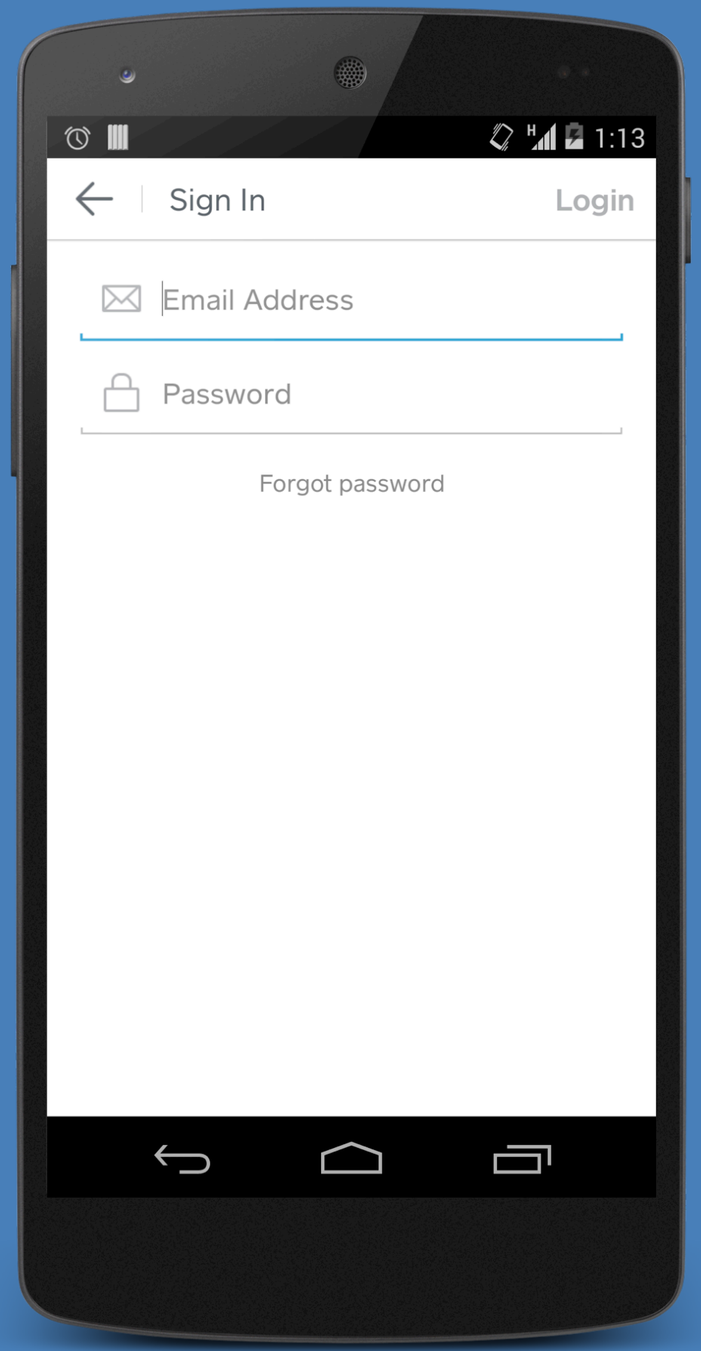
- Modules which override all network-calling classes

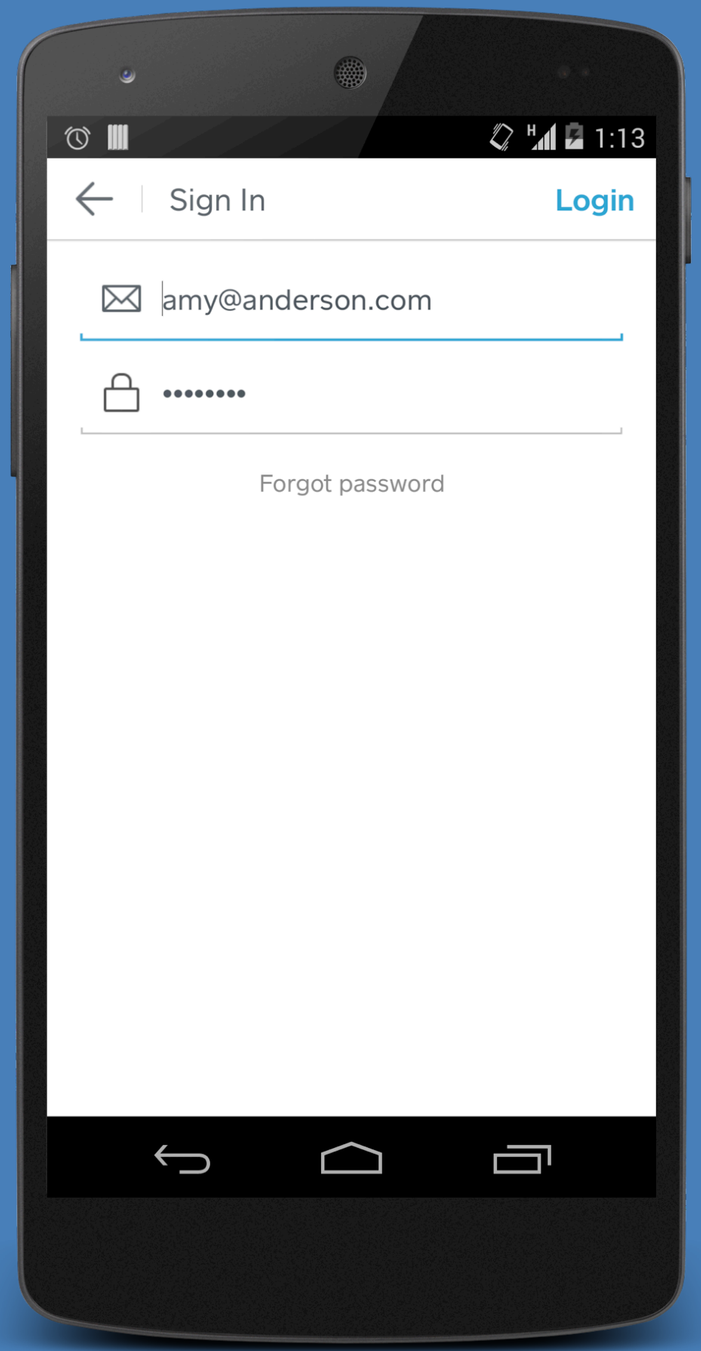
Mock Mode

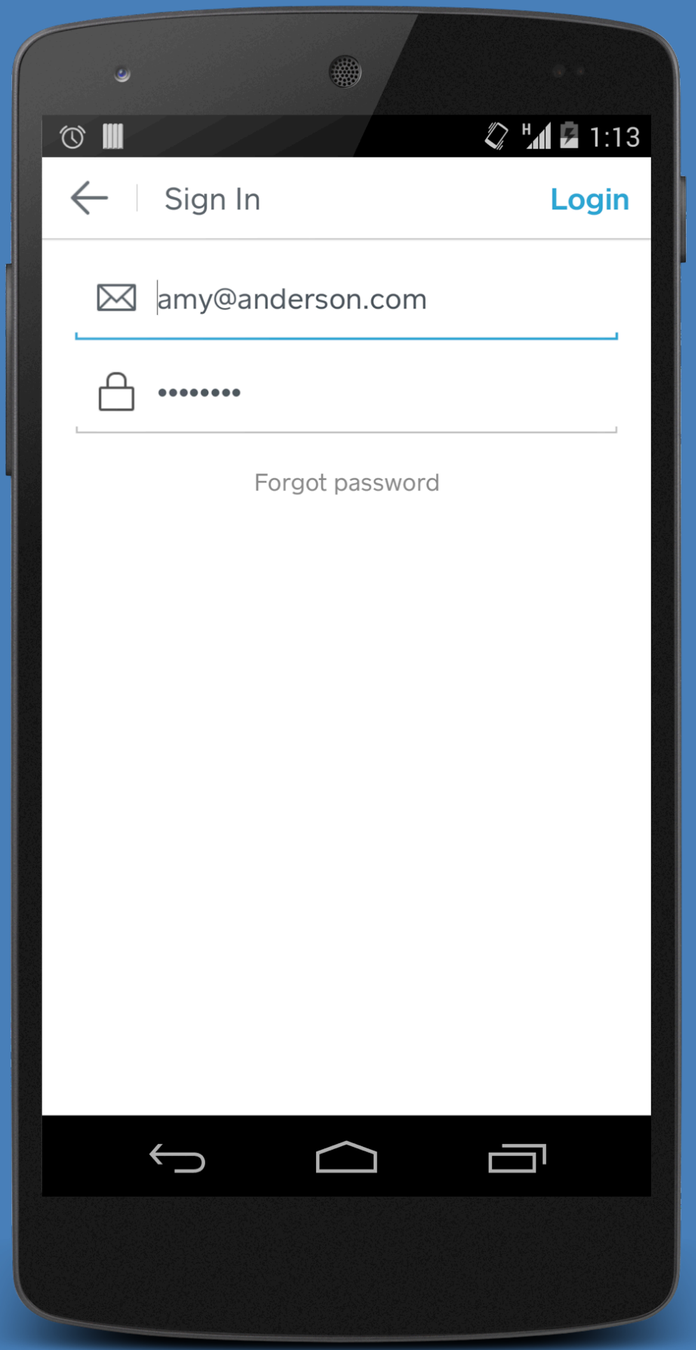
- Modules which override all network-calling classes
- Alternate implementations of network-calling classes which emulate a remote server but in-memory

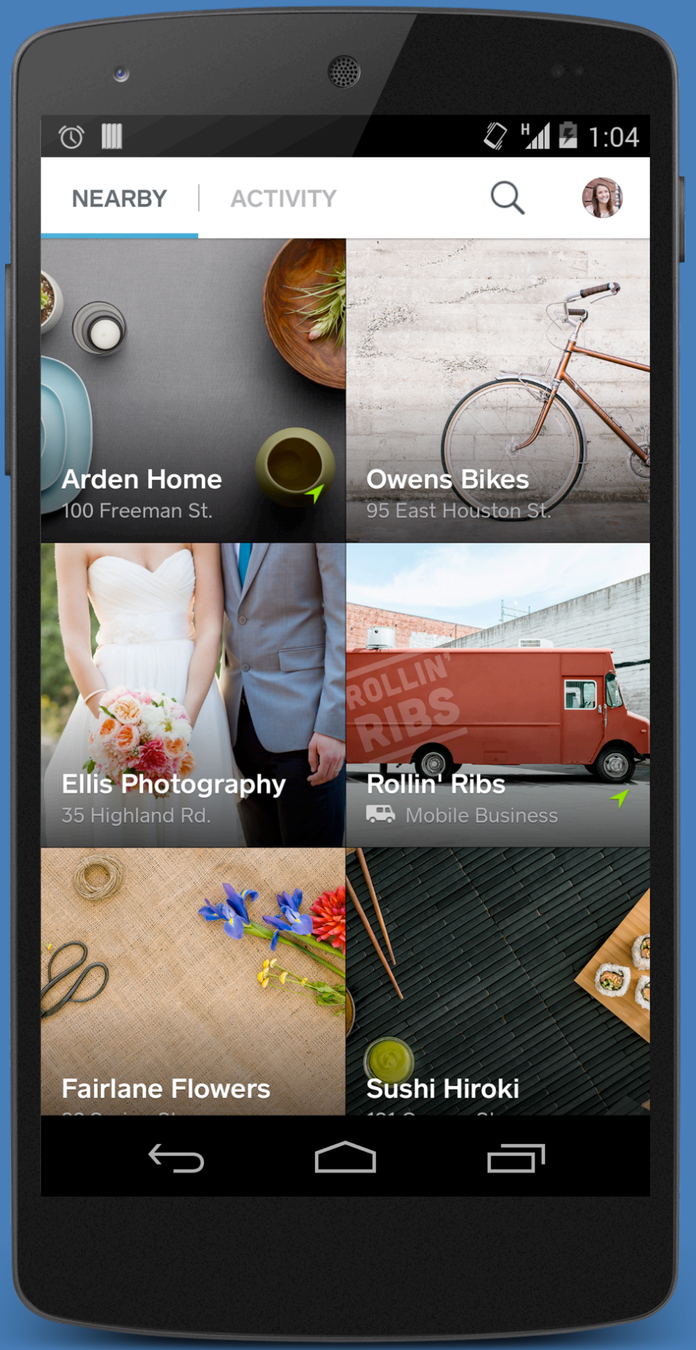
Mock Mode

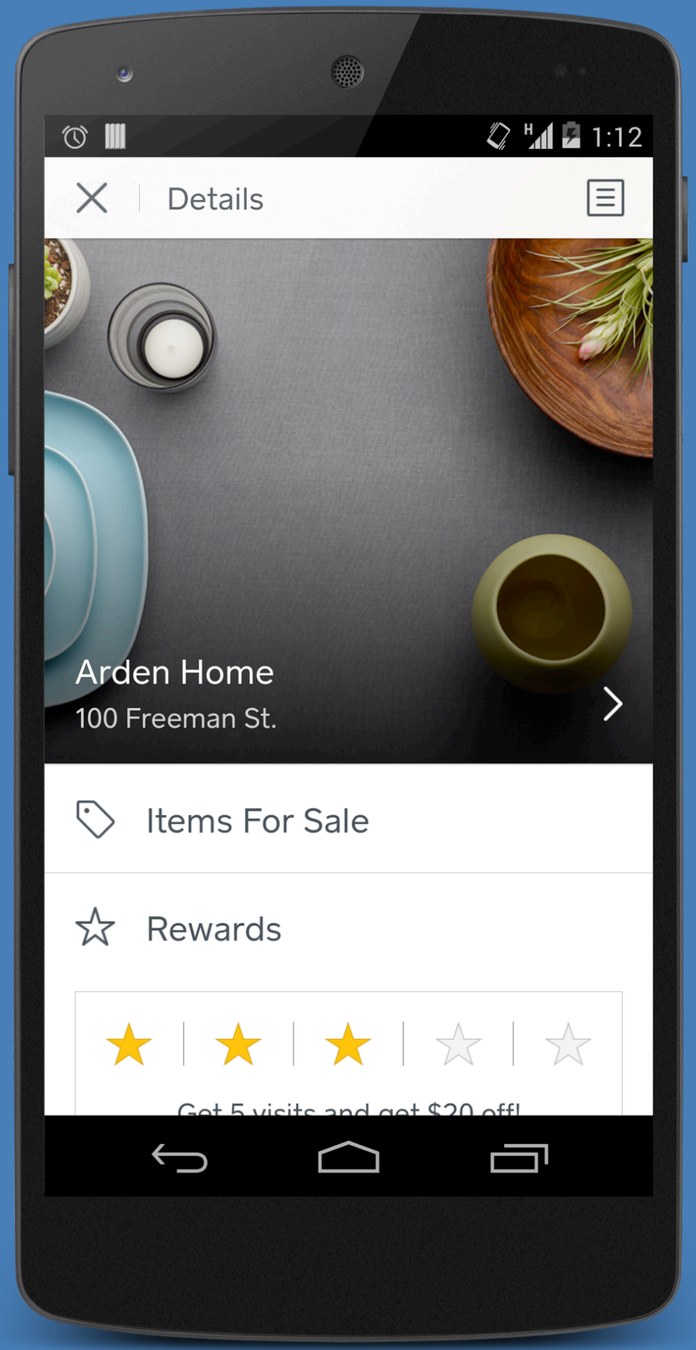
- Modules which override all network-calling classes
- Alternate implementations of network-calling classes which emulate a remote server but in-memory
- Fake images and data included in debug builds


1:13
Sign In
Login
Email Address
Password
Forgot password


1:13
Sign In
Login
amy@anderson.com
••••••••
Forgot password


1:13
Sign In
Login
amy@anderson.com
••••••••
Forgot password


1:04
NEARBY
ACTIVITY
Arden Home
100 Freeman St.
Owens Bikes
95 East Houston St.
Ellis Photography
35 Highland Rd.
Rollin' Ribs
Mobile Business
Fairlane Flowers
Sushi Hiroki


1:12
Details
Arden Home
100 Freeman St.
Items For Sale
Rewards

```
// src/release/java

final class Modules {
  static Object[] list() {
    return new Object[] {
      new WalletModule()
    };
  }
}
```

```
// src/release/java

final class Modules {
  static Object[] list() {
    return new Object[] {
      new WalletModule()
    };
  }
}
```

```
// src/debug/java

final class Modules {
  static Object[] list() {
    return new Object[] {
      new WalletModule(),
      new DebugWalletModule()
    };
  }
}
```

```
public class WalletApp extends Application {
  private ObjectGraph objectGraph;

  @Override public void onCreate() {
    super.onCreate();

    objectGraph = ObjectGraph.create(Modules.list());
  }

  public ObjectGraph getObjectGraph() {
    return objectGraph;
  }
}
```

Debug Drawer

Debug Drawer

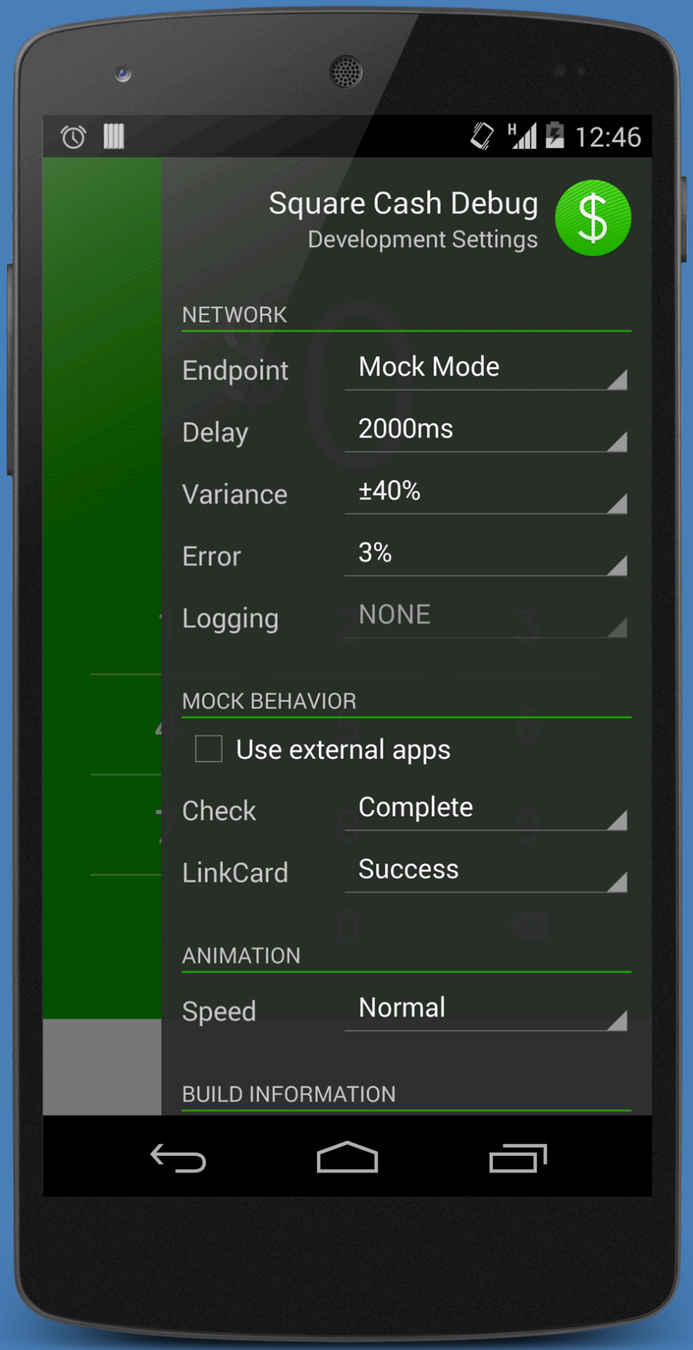
- Provides quick access to developer options and information

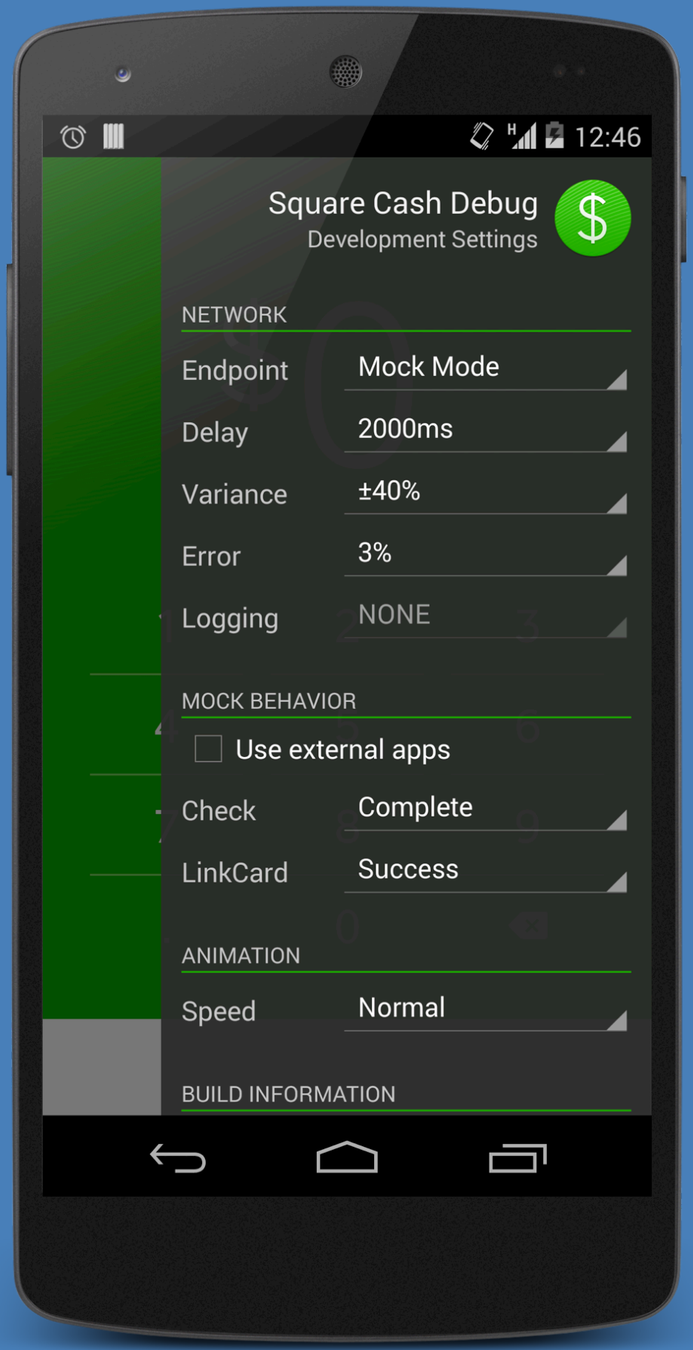
Debug Drawer

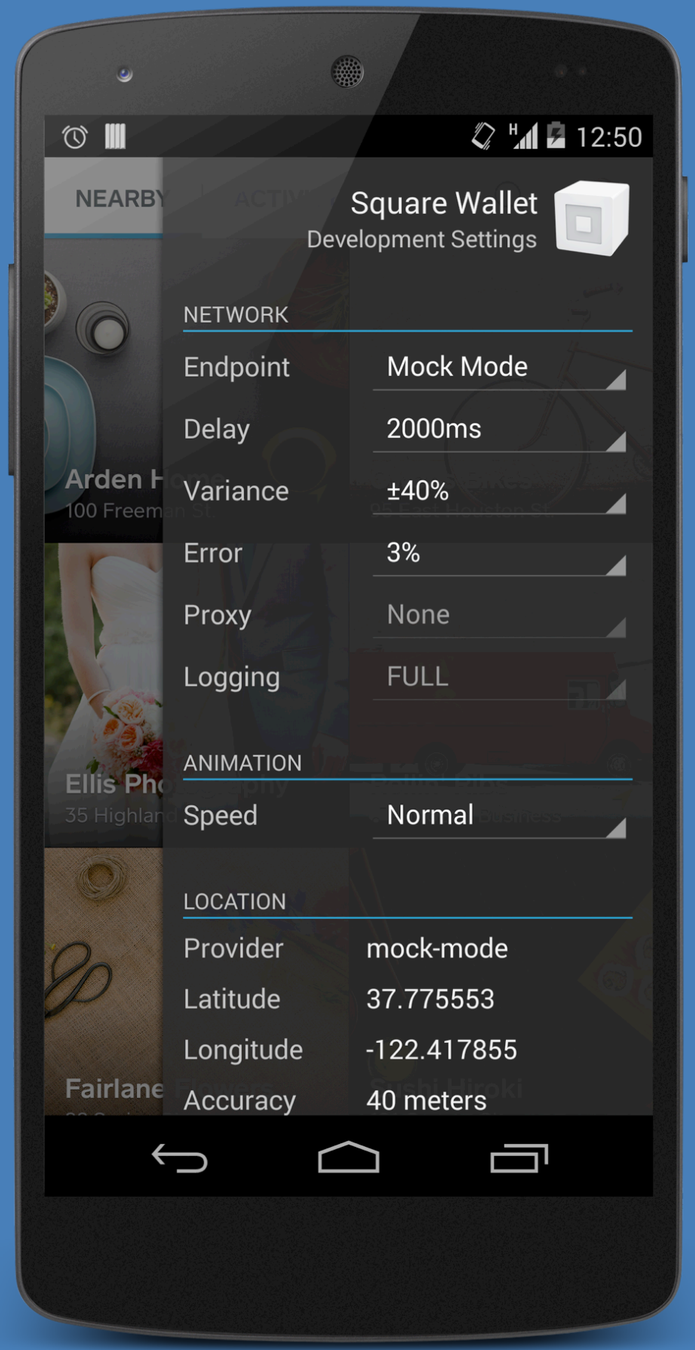
- Provides quick access to developer options and information
- Completely hidden from normal UI

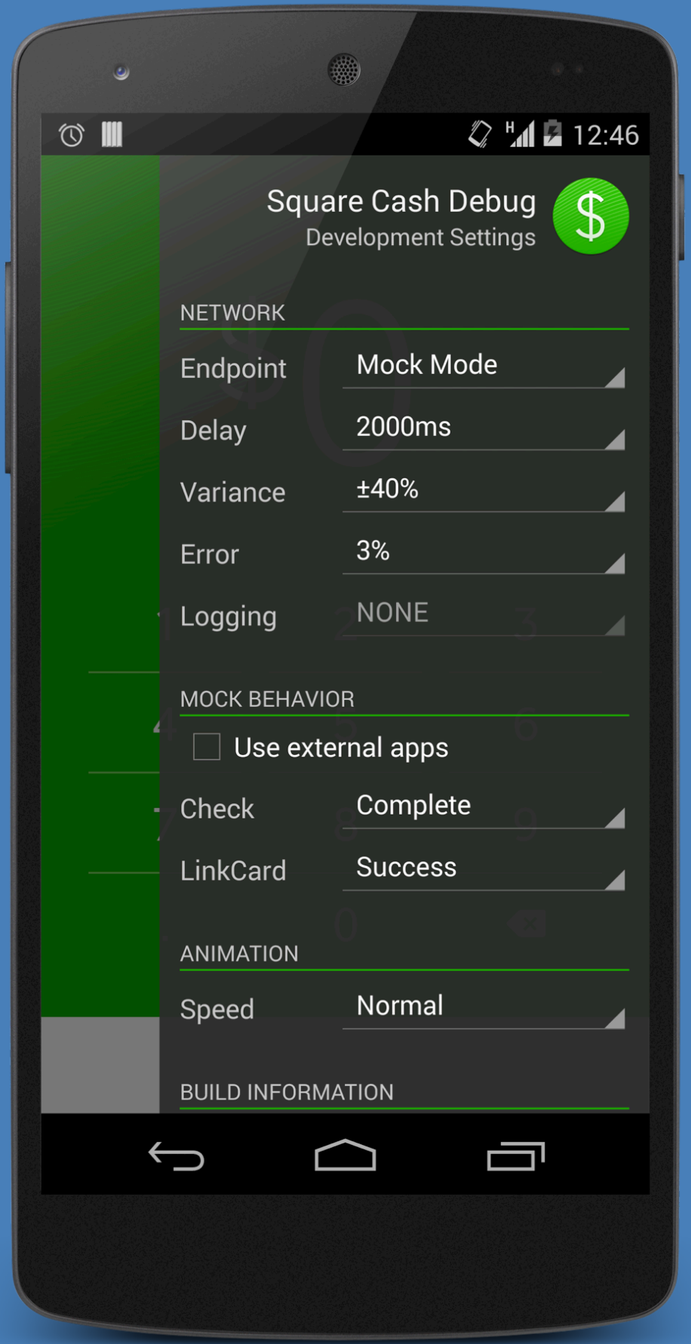
Debug Drawer

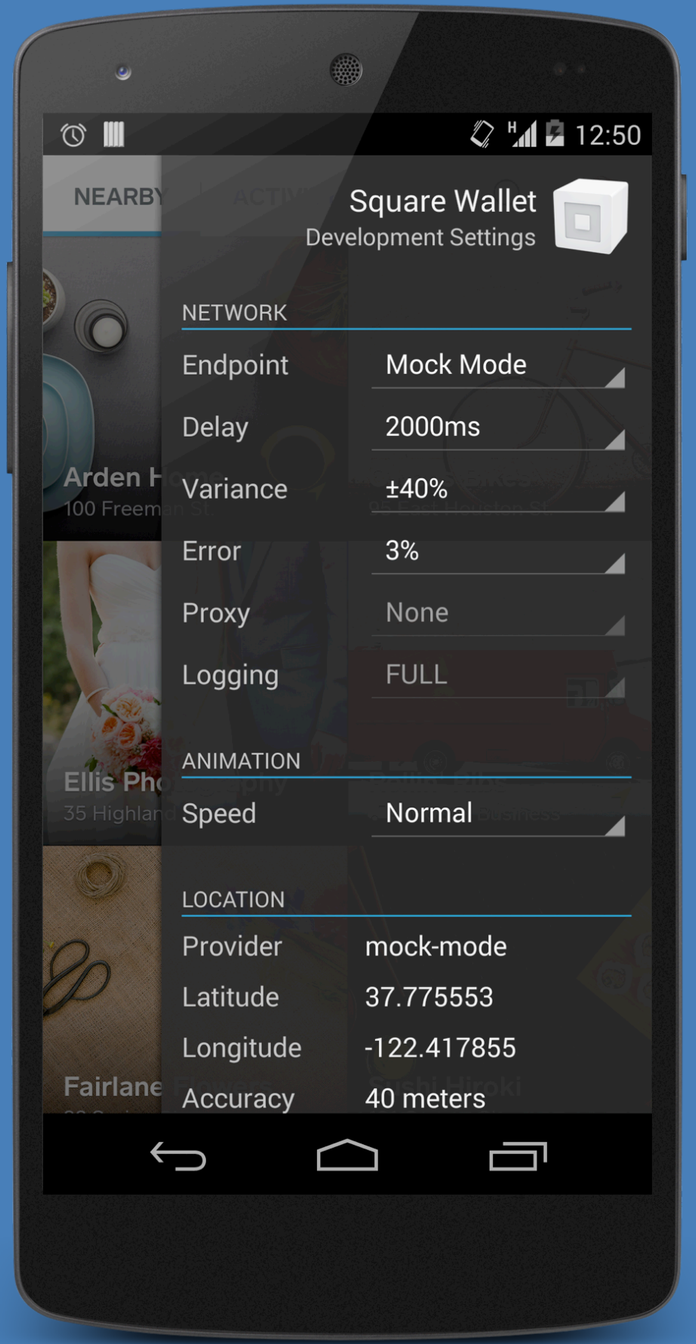
- Provides quick access to developer options and information
- Completely hidden from normal UI
- Contains controls for changing app behavior

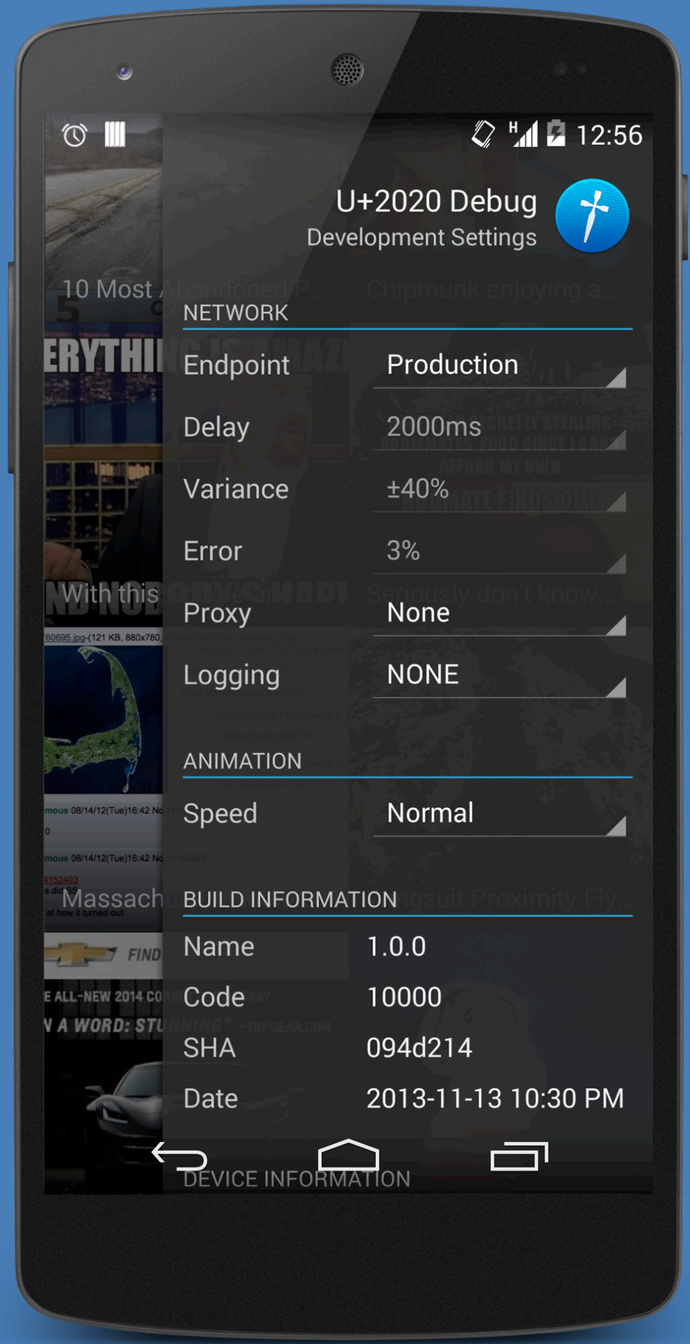

12:46
Square Cash Debug
Development Settings
NETWORK
Endpoint
Mock Mode
Delay
2000ms
Variance
±40%
Error
3%
Logging
NONE
MOCK BEHAVIOR
Use external apps
Check
Complete
LinkCard
Success
ANIMATION
Speed
Normal
BUILD INFORMATION


12:46
Square Cash Debug
Development Settings
NETWORK
Endpoint
Mock Mode
Delay
2000ms
Variance
±40%
Error
3%
Logging
NONE
MOCK BEHAVIOR
Use external apps
Check
Complete
LinkCard
Success
ANIMATION
Speed
Normal
BUILD INFORMATION


12:50
NEARBY
Square Wallet
Development Settings
NETWORK
Endpoint
Mock Mode
Delay
2000ms
Variance
±40%
Error
3%
Proxy
None
Logging
FULL
ANIMATION
Speed
Normal
LOCATION
Provider
mock-mode
Latitude
37.775553
Longitude
-122.417855
Accuracy
40 meters


12:46
Square Cash Debug
Development Settings
NETWORK
Endpoint
Mock Mode
Delay
2000ms
Variance
±40%
Error
3%
Logging
NONE
MOCK BEHAVIOR
Use external apps
Check
Complete
LinkCard
Success
ANIMATION
Speed
Normal
BUILD INFORMATION


12:50
Square Wallet
Development Settings
NETWORK
Endpoint
Mock Mode
Delay
2000ms
Variance
±40%
Error
3%
Proxy
None
Logging
FULL
ANIMATION
Speed
Normal
LOCATION
Provider
mock-mode
Latitude
37.775553
Longitude
-122.417855
Accuracy
40 meters


12:56
U+2020 Debug
Development Settings
NETWORK
Endpoint
Production
Delay
2000ms
Variance
±40%
Error
3%
Proxy
None
Logging
NONE
ANIMATION
Speed
Normal
BUILD INFORMATION
Name
1.0.0
Code
10000
SHA
094d214
Date
2013-11-13 10:30 PM
DEVICE INFORMATION

U+2020 Sample App

U+2020 Sample App

- Dagger
- Retrofit
- RxJava
- Picasso
- OkHttp
- Butter Knife
- Timber
- Build Flavors

U+2020 Sample App

- Dagger
- Retrofit
- RxJava
- Picasso
- OkHttp
- Butter Knife
- Timber
- Build Flavors

http://github.com/JakeWharton/u2020/

U+2020 Sample App

- Dagger
- Retrofit
- RxJava
- Picasso
- OkHttp
- Butter Knife
- Timber
- Build Flavors

#Soon

http://github.com/JakeWharton/u2020/

Dependency Injection

Dependency Injection

- Do NOT ignore the pattern

Dependency Injection

- Do NOT ignore the pattern
- Do NOT make *every* class use the pattern

Dependency Injection

- Do NOT ignore the pattern
- Do NOT make *every* class use the pattern
- Do NOT store dependencies as static fields

"Avoid Dependency Injection"

"Avoid Dependency Injection"

Using a dependency injection framework such as Guice or RoboGuice may be attractive because they can simplify the code you write and provide an adaptive environment that's useful for testing and other configuration changes. However, these frameworks tend to perform a lot of process initialization by scanning your code for annotations, which can require significant amounts of your code to be mapped into RAM even though you don't need it.

d.android.com/training/articles/memory.html#DependencyInjection

Dagger Performance

Dagger Performance

- Annotation processor generates code to fulfill dependencies

Dagger Performance

- Annotation processor generates code to fulfill dependencies
- Happens automatically inside of `javac`

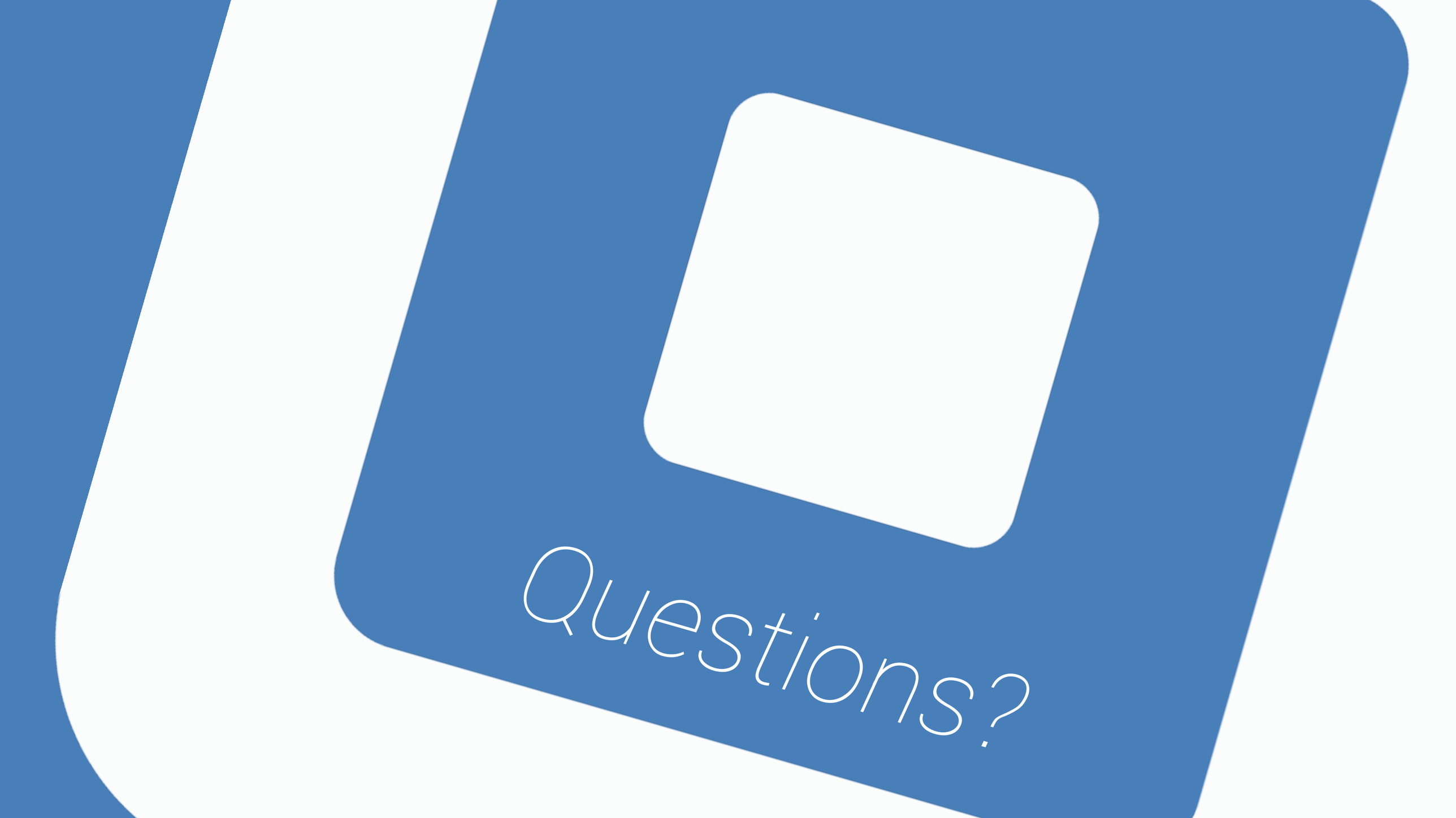
Dagger Performance

- Annotation processor generates code to fulfill dependencies
- Happens automatically inside of `javac`
- Zero reflection on methods or fields

Dagger Performance

- Annotation processor generates code to fulfill dependencies
- Happens automatically inside of `javac`
- Zero reflection on methods or fields
- Debugger and developer friendly

http://square.github.io/dagger/


Questions?

squareup.com/careers