Bungee Connect White Paper

Overview

Bungee Connect is an Eclipse-based application development framework targeted at Enterprise Java developers. It enables them to build or repurpose applications that run natively on the desktop, in the browser and on multiple mobile platforms from a single code base. Bungee Connect preserves and utilizes native behaviors and functionality specific to the target platform without losing visual or interactive fidelity as is the case with typical Cross-platform technologies. Bungee targets four browsers as well as iOS, Android, W8, BlackBerry, and Facebook. An application built with Bungee Connect for Android will also run natively on iOS iPhone or other targeted platforms. Adding a targeted platform takes just a few months initially depending on complexity.

Existing cross-platform development tool technologies ‘normalize’ (AKA least common denominator) platform-specific access. Cross-platform APIs result in applications that have greatly diminished user experience and product differentiation, and fundamentally lack the ability to address different form factors.

Bungee Connect enables Java developers to customize and adapt to each target platform, thus utilizing the full range of native look, feel and functionality. Bungee also supports the ability to tailor interaction for each specific form factor, even as completely different interactive mechanisms are required or preferred. Bungee’s unique Connection technology offers developers the ability to maintain a single set of application-specific logic where all multiple platform-specific assemblies are connected and configured separately through a simple design process.

For each platform and form factor “targeted” (supported) by Bungee Connect, Java developers can produce applications from the same Java code base, eliminating the need to rewrite or extensively modify applications in order to run on different platforms.

Problem to be solved

Enterprises face difficult decisions as they consider going Mobile with their customer or employee-facing internal applications. Numerous questions must be resolved:

- Which of the half dozen viable platforms should be supported?
- Which of the multiple form factors (iPhone/iPad) should be supported?
- Should we build native application or web applications?
- Is it more cost-effective to try to modify current applications for the new platforms and form factors or should we just start over?
- How much will it cost to maintain our Apps on numerous platforms and form factors?
- Can we get the performance and interactivity our applications need in order to be usable?
- How will we respond to an ever evolving array of platforms and form factors?
- Can we retool our internal skill set for Mobile without disruption, and how long will it take?
- Will this be a distraction from our central mission?

Unfortunate tradeoffs among the above concerns must be made. Re-writing and then maintaining corporate Apps on multiple Mobile platforms and form factors is expensive and time consuming, even if the company has the expertise in house. It is reasonable to believe that the Mobile landscape will change materially over the next few years, adding much uncertainty to the mix.

Bungee Connect was developed to address these questions.

Technology Description

The Model View Connection

All of today's mobile, web and desktop UI development architectures rely on some variant of the Model–View–Controller (or MVC) method for application construction. MVC defines a clean separation between the Model (application logic) and the View (user interface and interaction) by incorporating a Controller which coordinates state flow and interactive changes between the Model-View. Typically, however, the ability for the developer to control that flow leads to code that becomes interconnected resulting in applications that are difficult to move between platforms. MVC-based cross-platform APIs have emerged to solve this problem but they fall short when platform specific interactive control or behavior/look and feel is required or desired for optimal user experience (the LCD problem explained above).

Bungee introduces Connection technology that solves both the problem of clean Model-View separation as well as enabling direct and native access on every desired target platform or device. We call this approach the Model-View-Connection. The traditional Controller of MVC is replaced with a Connection framework that automates state changes, flow, and transformation between the Model and the View. Common elements of Controller behavior are now declarative with Connections in that they are expressed without direct reference to a particular type or specific access method on either the Model or View side of the connection. Connections are also extensible and configurable (as resource information) to allow for all the capability previously afforded by the Controller part of MVC, but without all the application and UI domain logic 'baked in' or managed inside the Controller. This unique Connection capability allows for a diverse set of connections to the Model without having to manage the separation or without having to compromise feature/function where traditional LCD API approaches fall short. In this white paper we will explain the anatomy of the Model-View-Connection, starting with the Model below.

Model

The Model manages the behavior and data of the application domain (see Model–View–Controller). The Bungee Connect product currently supports Java POJOs as the Model for its MVC implementation.

The Model can also notify observers when it's internal state or workflow activity changes. In Java, this is usually done with a listener or observer pattern. The Bungee Connect API provides a universal observer notification facility through the com.bungee.model.IModel interface. The com.bungee.model.Model class defines a set of static methods (or instance methods if the developer chooses to inherit from the

Model base type) that can be called from the Model (e.g. any developer defined POJO). The Model simply reports changes via a com.bungee.model.changed(...) or com.bungee.model.invoked(...) method call to the Bungee runtime and a notification is made to all interested observers.

In Bungee Connect, by far the most common observers are the Connection types. Connections automate the Controller part of the MVC UI design pattern. Connections are so integral to the Bungee Connect runtime that we refer to the standard MVC model as Model-View-***Connection***.

The Model's members (Java public class members) are all the Bungee Connect framework needs to discover how to connect to the Model at runtime. A Model definition consists of the following elements.

Fields

```
public int x;
public Person p;
public IModelArray<Person> list;
protected int y; // not a Model element since it is not accessible to outside observers
```

Properties (discovered on a Model by introspecting public method signatures of the Java class)

```
// a property called 'name'
private String name;
public String getName()
{
  return name;
}

public void setName(String name)
{
  this.name = name;
}

// a property with only 'getter' access since no 'setter' was defined on the class
private String id;
public String getID()
{
  return id;
}
```

Properties that Notify Observers

```
// change notifying property
private String name;
public String getName()
{
  return name;
}

public void setName(String name)
{
  Model.changed(this, "name", this.name, this.name = name);
}
```

Public Methods (including the capability of notifying Observers of Method Invocations)

```
// a Model method example without invocation notification
public void sendMessage(String msg)
{
  ...
}

// Model method with invocation notification
private List<Contact> contacts;
public void addContact(Contact contact)
{
  contacts.add(contact);

  Model.invoked(this, null, "addContact", (Object[]) contact);
}

// not a Model method since it is not public
private void calc()
{
}
```

View

The View operates on a Model making it into a suitable form for interaction. Views are typically GUI elements but can be non UI related as well. For example, a Database View contains information to be able to project on a subset of the data contained in a table.

Multiple Views can exist on a single Model; each can have a different purpose or may be intended for specific interaction in a particular application context. In Bungee Connect, each View contains a reference to the Model it represents as well as a View ID. Only one View can be designed for a particular View ID for a specific Model type. At runtime, when the application needs a View for a Model (Java Object instance) it looks up the View by View ID and the type of the Model instance. If the developer has designed a View for that type and View ID, then the View is instantiated and connected to the Model instance. If a View is not found on the Model type and View ID then Bungee attempts to find a View by searching the inherited hierarchy of the Model type.

A View contains one or many “View Fragments”. View Fragments are target-specific resource definitions that actualize the View on a particular device (see Figure 1).

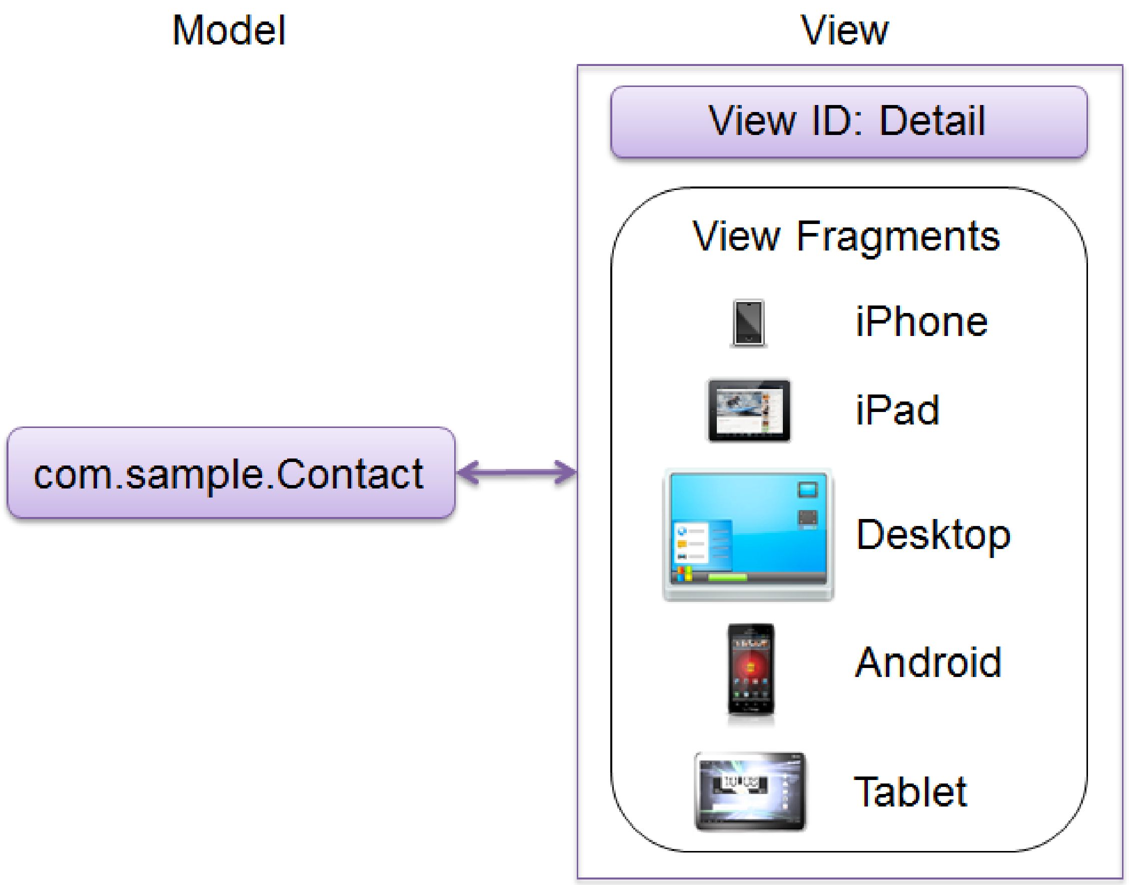


Figure 1 Example of the Bungee Connect Model-View Relationship

View Fragment

The View Fragment is a device-specific part of the View. There can be several View Fragments in a single View with at least one View Fragment for each kind of device target. For example, a view can have an Android View Fragment along with an iPad View Fragment, and an iPhone View Fragment. Both the iPad and iPhone devices can also have additional View Fragments for controlling platform specific interaction such as navigation and split view display, etc.

When an App is deployed on a particular target device only the target-compatible View Fragments are instantiated at runtime and the other View Fragments are dormant. View Fragment selection for a particular target is optimized during target generation so that only the parts of the View relevant and compatible for a target are generated for use on that target device at runtime (see Figure 2).

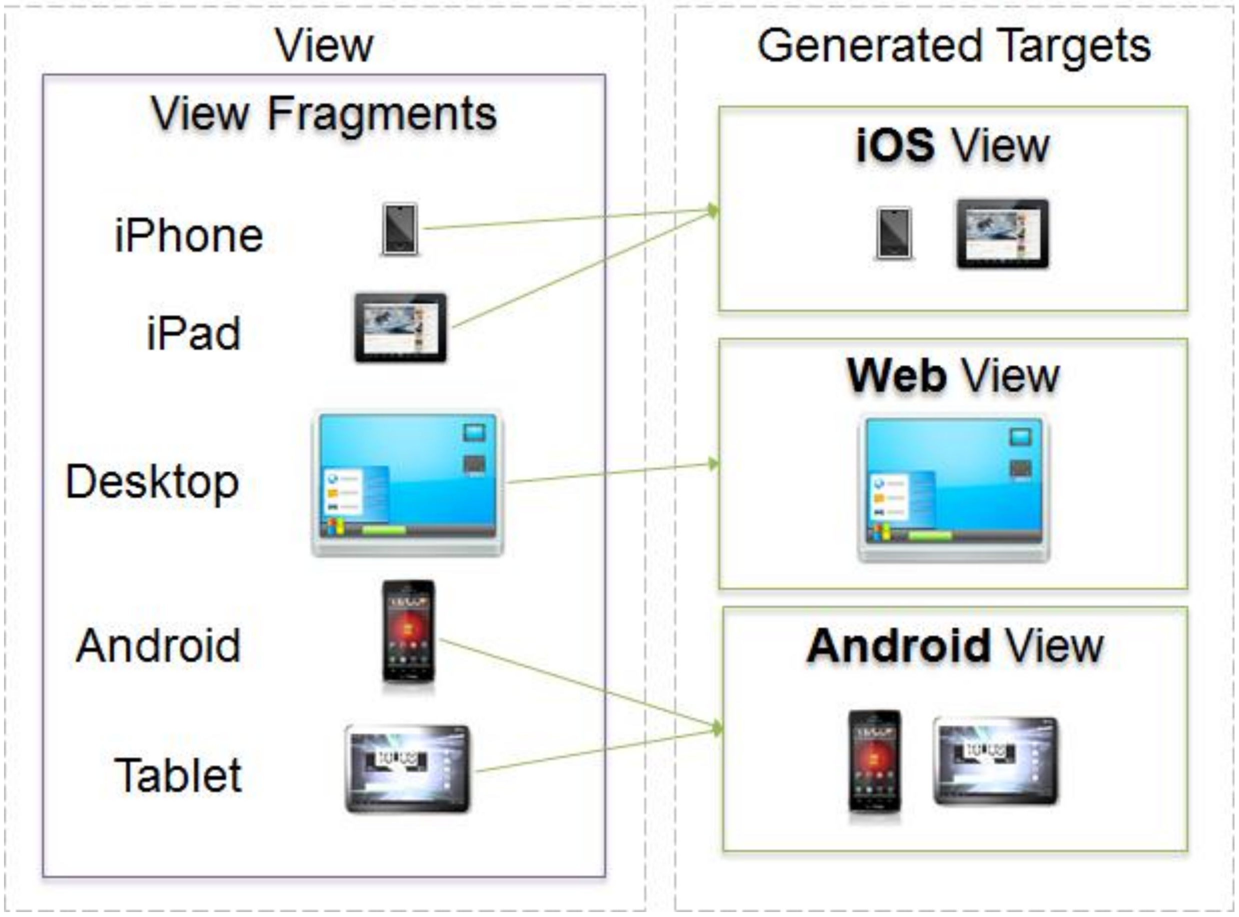


Figure 2 Example View with View Fragments and the resulting generated targets

Control

A View Contains View Fragments and View Fragments contain Controls. Both Views and View Fragments are Composite Controls. A Control is a special kind of Model that is "Connectable". Controls (Connectable Model Objects) are able to create a Connection between an external Model object instance and itself (Control.this) based on the View ID context. The View ID is used along with the type of the Model object to dynamically look up the appropriate Connection (that was designed for this Model and Control type). The Model is then bound to the Control with the aquired Connection (see Figure 3).

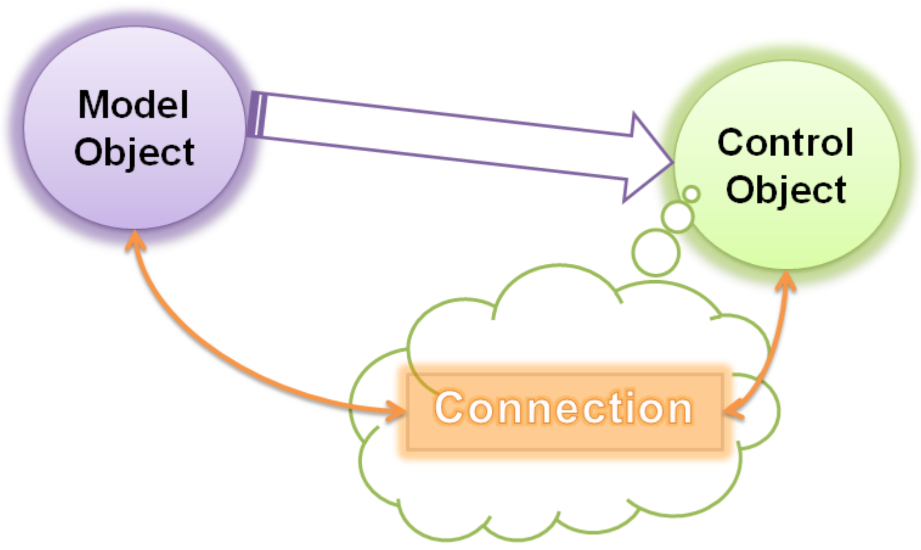


Figure 3 A Control object is able to connect itself to a Model object at runtime

Controls contain one or more “Facets”. Facets are concrete interfaces that a Connection can attach to on the Control side of the Model-View Connection. One or many Model-Control Connections can exist within a View Fragment Definition (see Figure 4).

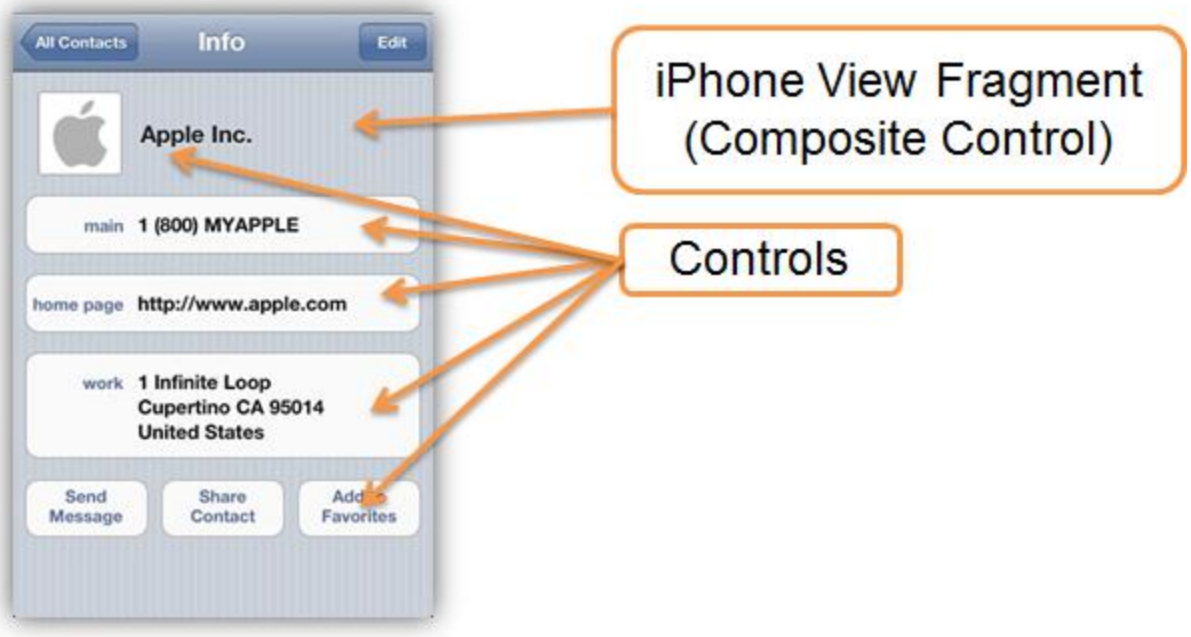
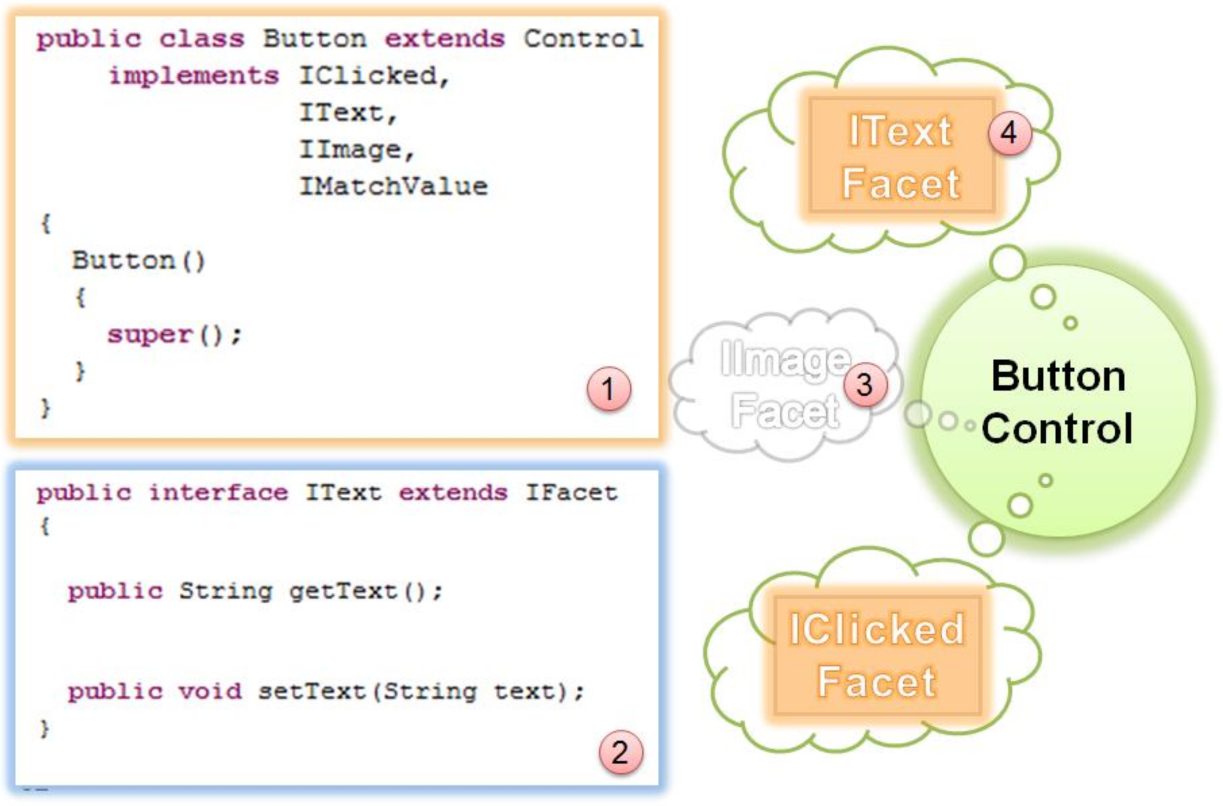


Figure 4 Example of a View Fragment which is a container for Controls and their Connection definitions

Facet

Controls contain one or more Facets. While Controls are Connectable Models, Facets are Connectable Interfaces. Since Facets are Interface types a Control can implement several Facets. Facets are ideal abstractions for enabling optimal portability of Control behavior and function across device and platform domains. This allows the developer to reuse Connections to a particular Facet that can be shared across many different Control types that run on many different device platforms. Therefore, the Control can still be unique and proprietary to a particular device (preserving all its native form and function) but still contain Facets (Interfaces) that are shared widely across many different Control types across many different platforms.

Facets are one of the key ingredients that enable Bungee Connect to give the developer the benefits of cross platform without compromising form and function that native Controls provide (see Figure 5).



```
public class Button extends Control
    implements IClicked,
               IText,
               IImage,
               IMatchValue
{
  Button()
  {
    super();
  }
}
```

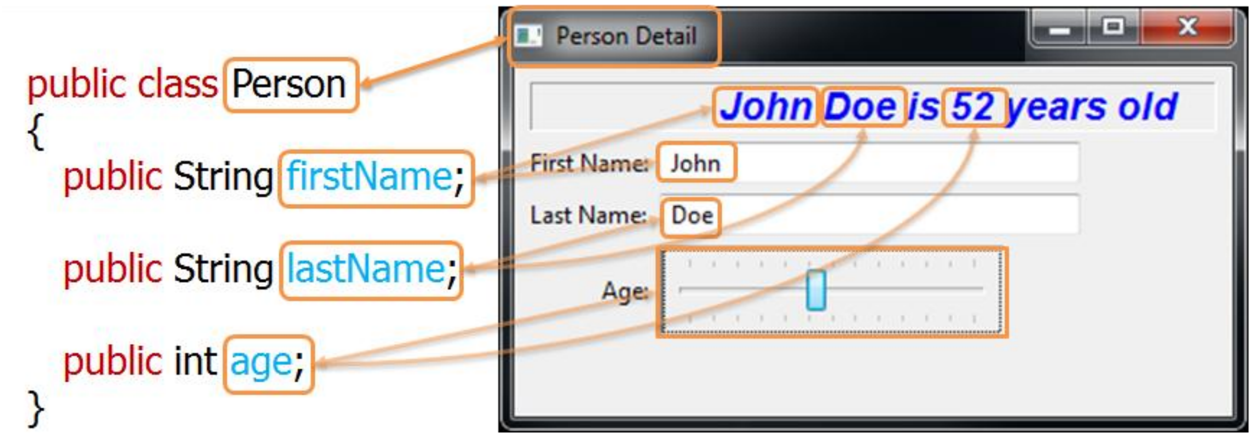
```
public interface IText extends IFacet
{

  public String getText();


  public void setText(String text);
}
```

Figure 5 Example Facet (see numbered explanations below)

1. Button Control example that currently implements four Facet Interface types
2. IText Facet Interface example shows a typical use pattern, in this case a property called "text"
3. Example of a Button Control that implements Text and Clicked Facets but does not have an implementation for Image (the example Button on that device or platform does not support Images). The Connection definition between the Model and the Control is preserved whenever the Control implements the Interface and the Model designs a connection to that Interface. See Tutorial 2 for an example Connection defined between a Model object and the IText Facet.
4. Shows that this Button Control on this device is able to implement the Text Facet

Connection

The Bungee Connection framework provides a powerful mechanism for connecting the Model to the View. The Connection replaces the traditional Controller design pattern commonly used in MVC frameworks. The Connection approach is more automated and better encapsulates common patterns of relationships typically encountered between the Model and View. Controller logic typically requires the developer to create code specific to the Model that they are coordinating to the View for each Model type for which they build a Controller.



```
public class Person
{
   public String firstName;

   public String lastName;

   public int age;
}
```

Person Detail

John Doe is 52 years old

First Name: John

Last Name: Doe

Age:

Figure 6 Model - View Connection Example

With Bungee, Connections are able to infer the relationships between the Model and the View via Model class reflection information and based on common usage patterns. For example, when the Model contains a field of type String, the default connection relationship between this field and a UI type-in box Control can be easily inferred (see Figure 6). The developer can then design and customize that initial inferred relationship through the use of other pre-packaged Connection types. Only when any existing Connection types fails to satisfy the desired custom 'controller logic' does the developer need to create custom logic (a custom Connection) to deal with that case. But even in those rare cases, the resulting custom Connection can be reused in other Model to View types since Connections typically focus on the type of Connection rather than the semantics of a particular Model type.

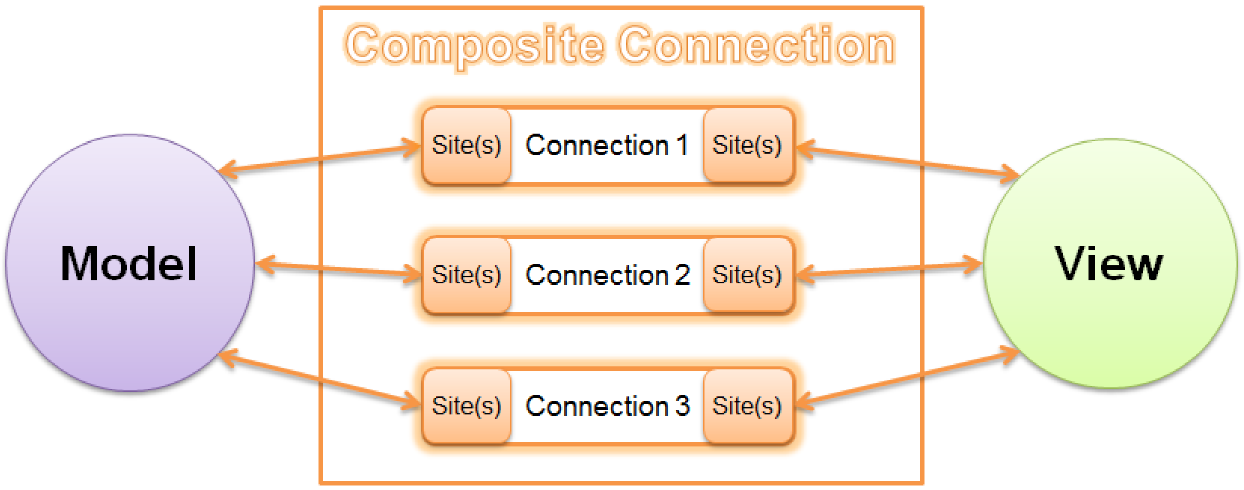


Figure 7 Connections Bind the Model to the View and replace traditional Controller Logic

Connection Types

There are three main types of Connections:

1. **Standard Connections**
   A set of default Connection types that can be considered 'built-in' as they are part of the Bungee Connect runtime. These include many of the common usage patterns ideal for building Model-View Connections.
2. **Composite Connections**
   Connections that contain one or more sub-connections given a particular Model-View context. Many times connection requirements between the Model and View are complex and require more than one connection to satisfy a specific Control workflow (e.g. A button Control connected to an Model Object that connects to the Button String title and the Action (clicked) method).
3. **Custom Connections**
   Developers can specialize existing Connection types by implementing their own Connection subclasses (e.g. a new type of String formatting Connection, or multiple Site aggregation, custom state transformation, etc.)

Connection Attributes and Capabilities

- **2 Way, Many-to-many**
  Connections can establish a two Way relationship between each side of the connection. This means that Connections can respond to state changes on either the View or the Model side of the Connection.
- **Dynamic and Polymorphic**
  Connections between the Model and View are determined at runtime and depend upon the Java object instance type discovered in a particular 'parent' Model object context and the View ID defined in the current View context. The dynamic connections are also polymorphic since a

Connection structure can change base specializations that can be made to a custom or composite connection based on the object instance type.

- **State Transformation and Flow Control**
  Connection types manage Sites (Sites in turn manage the physical location of the 'member of interest' and either side of the connection) and often perform transformation when state flows between the Model and View (and vice versa). For example, a Model integer property may be connected to a String Control property, when either side changes, it's the Connection that performs the conversion between the integer and String.
- **Cycle Reduction and Update Optimization**
  Changes in Model or View state can propagate, causing triggering cycles when connections respond and update localized state. Bungee standard Connections and Connection-base type implementations detect when a cycle occurs and reduces them to a single change rather than endlessly updating or hard failing. In this way, Model and View objects are updated only once in a given state or method-triggered context. These cases can become very complex or difficult to solve when Java Observer patterns are required to be managed by the developer in their own custom Controller logic context. With Connection technology, problems like these can be managed in a more general way that is reused throughout the system.

Summary

For Java developers using the Eclipse IDE, Bungee Connect is the best approach to go Mobile with Enterprise Apps. Bungee Connect offers a clear path to Apps that run natively on multiple Mobile platforms and form factors as well as four browsers and the desktop, all from a single code base. For example, an application built with Bungee Connect for Android will also run natively on iOS iPhone or other targeted platforms without the "Least Common Denominator" affect.

The use of Bungee Connect:

- Simplifies the decision of which platforms and form factors to support first
- Opens a pathway to build either native or web Apps from a single code base
- Eliminates costs associated with writing and maintaining multiple platforms by hand
- Produces Apps with no performance or usability limits
- Positions the company to respond to newly introduced platforms and form factors
- Helps the company to leverage its current investment in Java/Eclipse skills