Blue-Green Application Deployment w/ GemFire

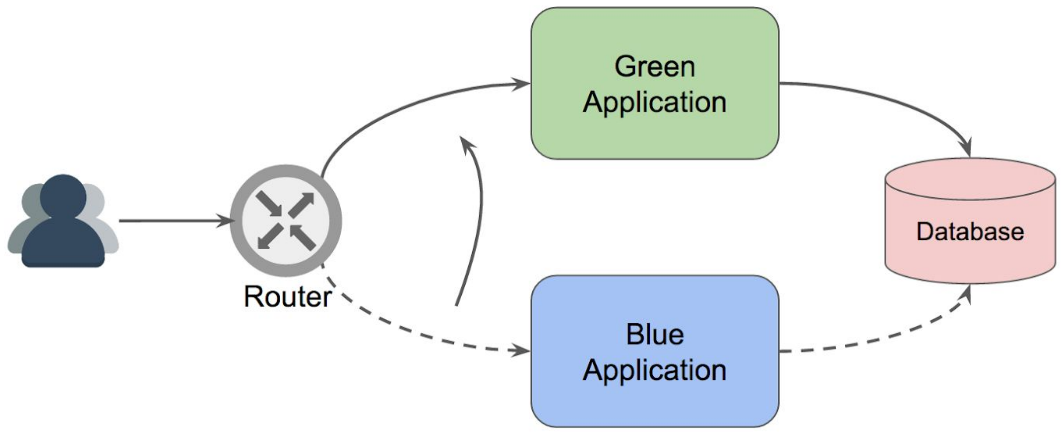
Authors: Aditya Padhye, Gideon Low

Motivation

Blue-green deployments are synonymous with agile programming and incremental feature upgrades. The idea to introduce a new version of an application into production with no downtime is not new. Most companies implement blue-green deployments in one form or another if they wish to deliver updates at a regular cadence. Gone are the days when enterprises had to spam their users notifying them about application downtime (well not really, but we hope to get there someday). This post is a small step in that direction; to achieve blue-green deployments when GemFire/Geode is being used as the persistence layer.

Blue-Green deployments:
The idea of blue-green deployments was formalized by Martin Fowler a while ago. The theory behind the idea is pretty straightforward:

1. Blue application (version 1) is in production.
2. Deploy green application (version 2) to production.
3. Switch traffic from blue to green application.
4. Delete blue application.



This not only allows for continuous and fast delivery of application updates but also the possibility to rollback to the last known good state in case things go haywire.

Simple, right? Well, it gets ‘slightly’ more complicated when the changes to the application involve modifications to the back-end database schemas:

As Mr. Fowler explains in his blog:

> “Databases can often be a challenge with this technique, particularly when you need to change the schema to support a new version of the software. The trick is to separate the deployment of schema changes from application upgrades. So first apply a database refactoring to change the schema to support both the new and old version of the application, deploy that, check everything is working fine so you have a rollback point, then deploy the new version of the application. (And when the upgrade has bedded down remove the database support for the old version.)”

What this boils-down to is the need to enable both backward and forward compatibility between application versions within the data management layer, so that sequential versions of an application can interoperate seamlessly and without downtime during the upgrade procedure.

Let’s look at how we can apply these ‘refactoring’ changes to data in a GemFire grid to support both the old and new version of an application for an example use-case.

“Schema” changes in GemFire

GemFire stores data in the form of serialized Java objects. Although there is no schema to change per se, the structure of the object itself may change with the addition, deletion or modification of attributes.

Let us consider an example in which a Person class undergoes a change. The original version uses a single attribute (“name”) to capture a person’s full name. The new version of the application instead captures a person’s first and last name separately, using the attributes *firstName* and *lastName*.
For example: ‘John Doe’ becomes ‘John’ and ‘Doe’ in the new version of the application.

Let’s assume we have a REST application that has the following attributes for Person. Let’s call this the blue-person-class.

```
class Person {
      String id;
      String name;
      String age;
}
```

Below is the class for the new version of the application, with the *‘name’* field split into *‘firstName’* and *‘lastName’*. Let’s call this the green-person-class.

```
class Person {
      String id;
      String firstName;
      String lastName;
      String age;
}
```

In order for the objects on the server to be compatible with both versions of the application, we’ll be using an intermediate class with all three fields: *name*, *firstName* and *lastName*. Let’s call this the intermediate-person-class.

```
class Person {
      String id;
      String name;
      String firstName;
      String lastName;
      String age;
}
```

Approach

Before we are ready to deploy the green-application, we need to refactor the data stored in GemFire to make it compatible with both versions of the applications. We also need to make sure that objects created or updated by the blue-application during this transition phase correspond to the type intermediate-person-object.

Here are the GemFire tools/features used to implement the necessary pattern:

PDX Serialization and Metadata Repository

PDX is GemFire’s native/optimized storage mechanism for objects. We rely on PDX’s ability to manage different versions of an object type (or class) as a combined super-set, including the attributes defined in both older and newer versions. Client applications only materialize the attributes included in their local class definition, and transparently piggy-back attribute data that doesn’t.

Typically, the GemFire PDX metadata repository is built-up via use of the ReflectionBasedAutoSerializer. During client application start-up, domain object class definitions targeted for GemFire server storage get fed into the ReflectionBasedAutoSerializer, which creates a PDX type for each uniquely named class, and a PDX attribute for each of the PDX type’s defined attributes. These PDX types and PDX attributes are sent to the server-side and used to construct a tabular (or columnar) style object storage structure. The PDX management API usefully provides methods to dynamically add and programmatically manipulate new class attributes we anticipate coming from new versions of an application. We will use this feature to help migrate in-place “blue” data already populated from the blue version of the application.

Server-side Function

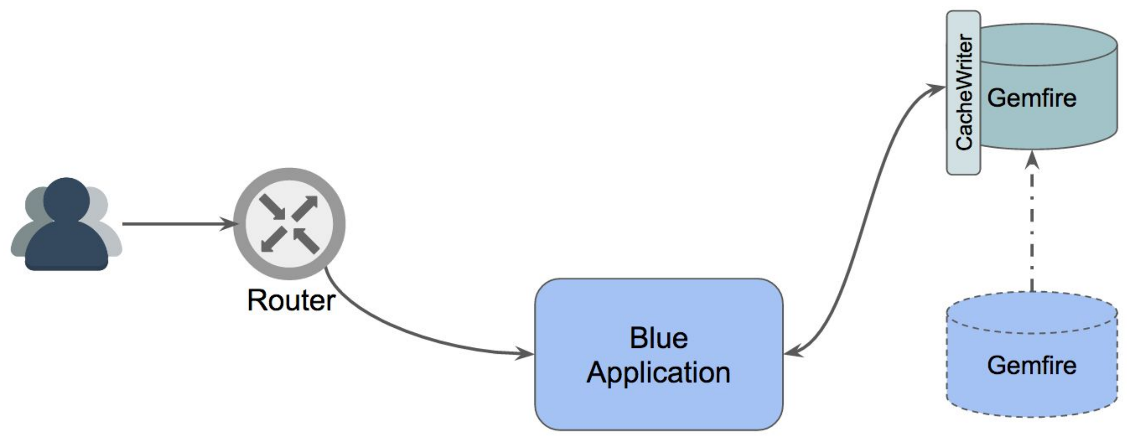
GemFire Server-side Functions provide a way to execute user code directly on the server-side in-memory data (analogous to a database Stored Procedure). Our custom, one-time, batch data migration logic is invoked via a server-side function.

Cache-Writer

A user plug-in that executes code in response to changes to data (analogous to a database table “before” trigger). We use a custom cache-writer to transparently morph *blue* and *green* class instances received from client applications into our *intermediate* version.

Let's go through this step by step:

1) Initially, the blue-app is in production and the objects in GemFire are of the type blue-person-object.



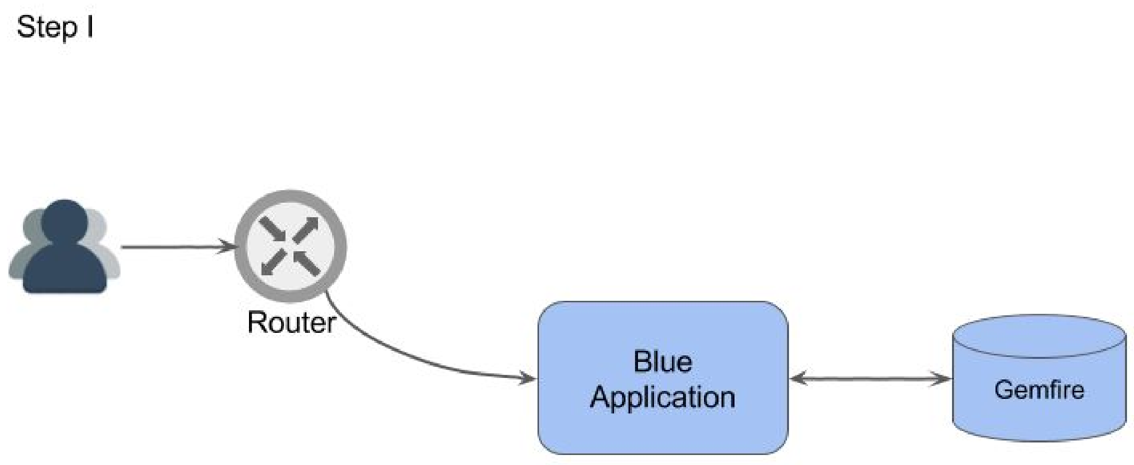
2) Before we can deploy the green-app into production, we need to do the following:
    a) Make sure any new objects created or modified by the blue application version are of type intermediate-person-object. We do this using a Cache-Writer. When an object is created or updated, it is intercepted by the CacheWriter, and the *name* attribute is copied, split, and added into *firstName* and *lastName,* and finally an object of type intermediate-person-object is stored. In GemFire's PDX object storage model, adding these two fields automatically and dynamically modifies the class definition to correspond with the intermediate version of the class.
    b) Migrate existing object instances stored in the cache from type blue-person-object to type intermediate-person-object. A server-side function can iterate over all data present in the region to split the *name* field into *firstName* and *lastName*.
3) At this point, the green-app can be deployed to production. The cache-writer is still intercepting any create/update events to make sure all objects being stored are of type intermediate-person-object.
4) Once deployed to production, the traffic can be routed to the green-app.

5) After making sure everything works fine with the green-app, the CacheWriter can be disabled. As a last step, another server-side function can be used to get rid of the *name* field from all the objects to remove the backward compatibility with the blue-app. Done!

Result

The result is that of having deployed a new version of the application without having to bring the old one down. In the back we have a persistence layer that is now completely compatible with the changes in the application.

Since no blog should ever be complete without a gif, here’s one of what the step-by-step process looks like:


Step I
Router
Blue Application
Gemfire