Adapting JDT to the Cloud

Alex Boyko – Pivotal

Jay Arthanareeswaran - IBM

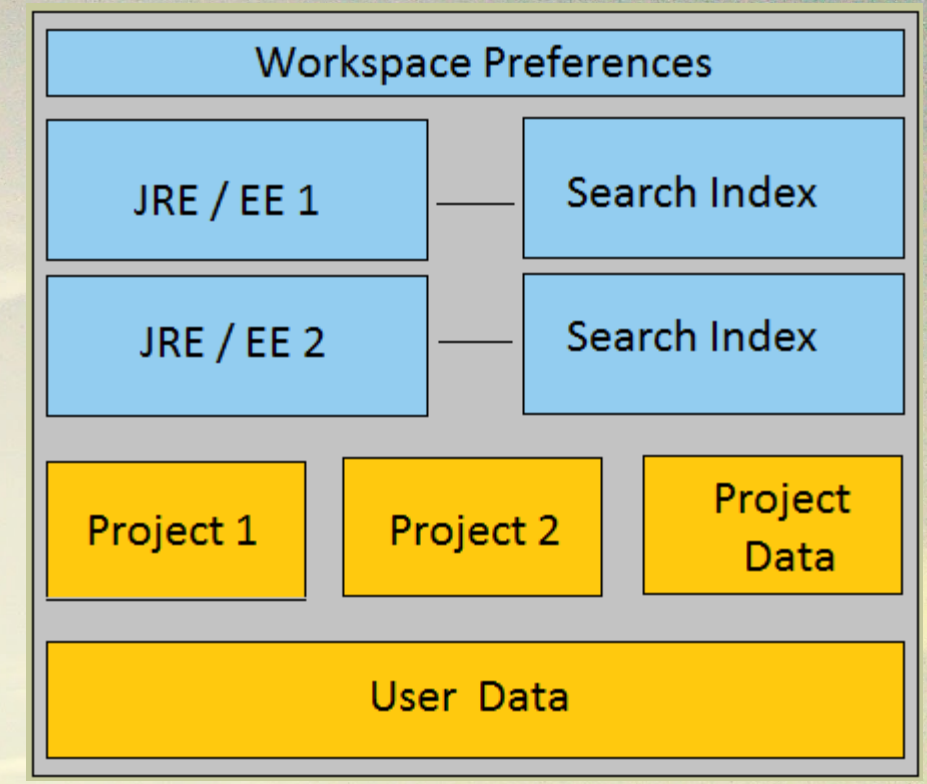
John Arthorne - IBM

Topics

- Background and motivation
- Adapting JDT code base to run in cloud
- Incorporating Java tooling in Web IDEs
- Demo
- Conclusions and next steps

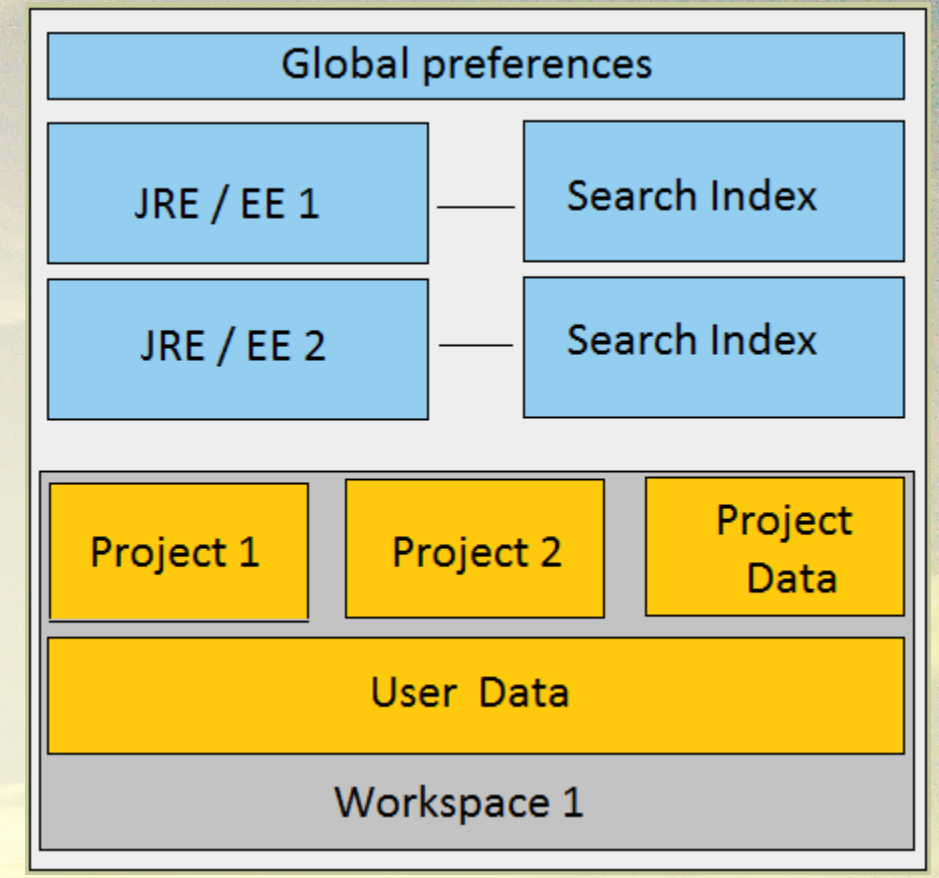
JDT in Eclipse

- One User – One Workspace
- Not scalable



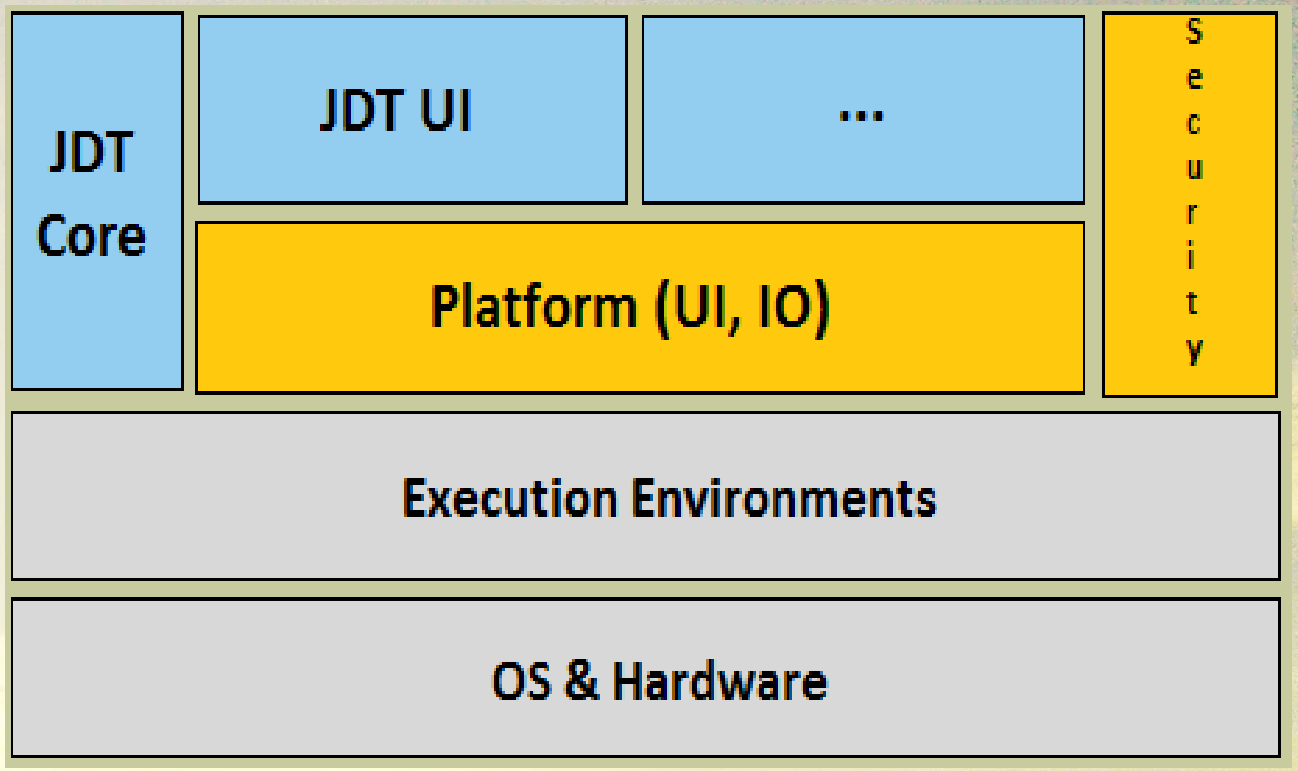
JDT in Cloud

- Want something like this
- Share resources
- Scale better



JDT in Eclipse

- Some editor features are tied to UI
  - Code completion
  - Quick Fix
  - Refactoring
  - Hover/Javadoc
  - Code Formatter



JDT in Eclipse

- Some features are designed for IDE
- Maintain state of some sort
- Flow needs to be broken down

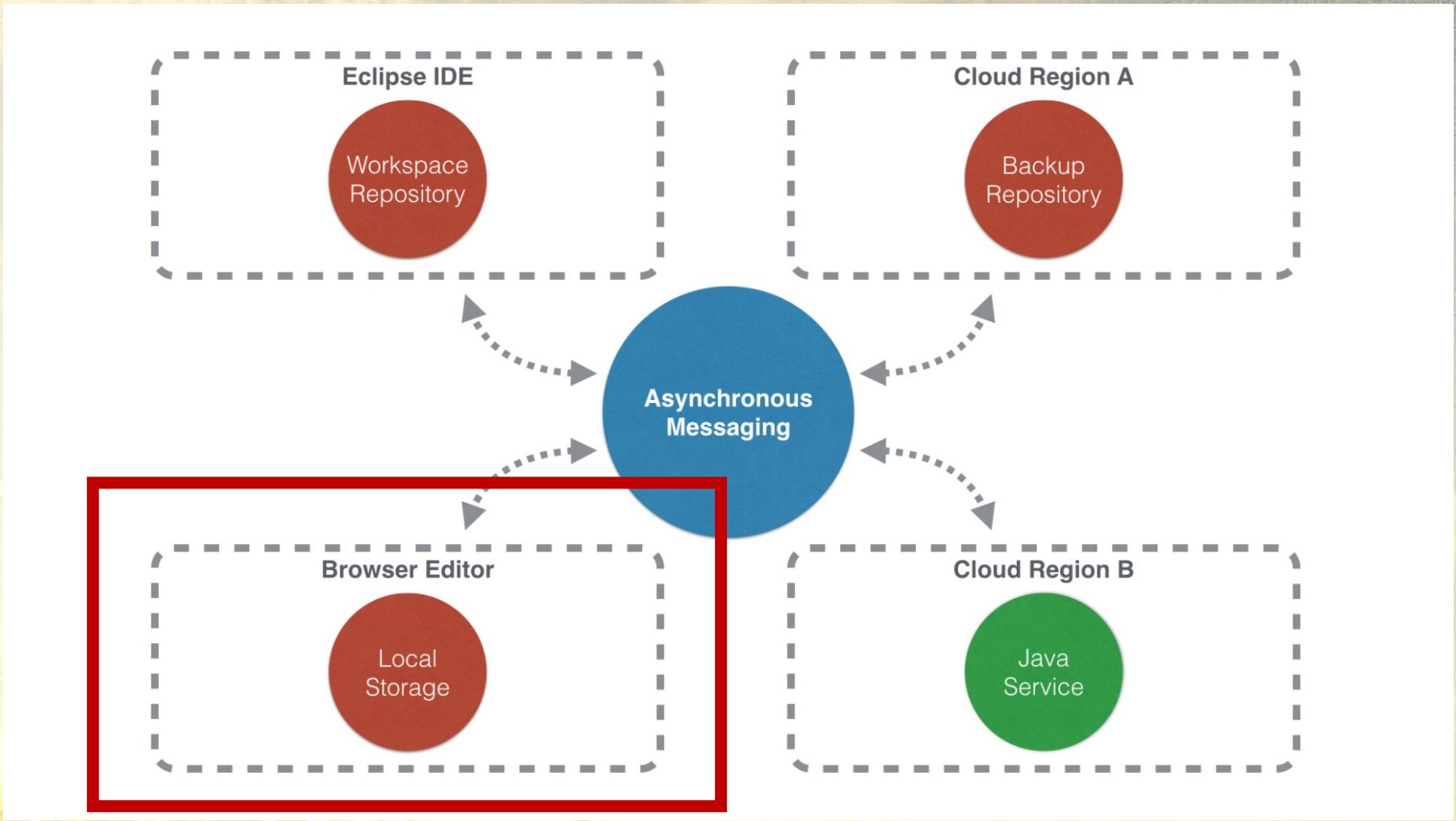
JDT in Eclipse

- In-memory User Cache
  - Java Element cache
  - Delta cache
  - Javadoc cache
- Very basic cache management
- Not scalable for multi-user set up
- Some can be reused across users

Flux project overview

- Flux project is exploring new architectures for Cloud-based development tools
- Goal is to connect and integrate tools across servers, browsers, and rich clients
- Using an architecture of loosely coupled tooling micro-services connected by an asynchronous message bus
- Prototype uses JDT service running in cloud, connected to browser-based editor

Flux prototype architecture



Web IDE integration

- Considered both Orion and Che as possible web editor integration examples
- Selected Orion because its plug-in model is well suited to the micro-service approach
- Good correspondence between Orion plug-ins and JDT API
- Added socket.io to Orion for communication with Flux message bus

Flux Orion client features

- Syncing of resources between Orion file system and resource stored in the cloud by Flux (Mongo DB for example)
- Syncing of editing changes between Orion editor and other editors in the cloud via Flux messaging system
- Send messages to Flux to perform various Java IDE related actions such as invoke content assist or navigate to definition
- Receive messages from Flux and react appropriately in the UI to display current problem markers, apply the content assist proposal etc.

Orion plugins used

- **orion.core.file** plugin is used to sync the Orion file system to file system in Flux. It tracks files and folder create/update/delete changes and applies them in Orion file system or broadcast the same changes done in Orion locally to Flux.
- **orion.edit.model** plugin to broadcast edit changes from Orion editor to Flux
- **orion.edit.live** plugin to react in Orion editor to edit changes performed elsewhere (in the cloud) on the same resource. (For example a name of a method has been changed and Flux JDT service in the cloud broadcasted new problem markers for the resource. This plugin will allow us to set new errors and warnings on editor contents). In other words it reacts to any messages directed to currently edited resource in terms of editor UI.

Orion plugins used (cont)

- **orion.edit.validator** plugin updates errors and warnings in the editor when Orion needs to update them. Errors and warnings are obtained by sending a problem markers request to Flux and collecting replies from Flux
- **orion.edit.contentAsisst** plugin collects data for content assist proposals at a given position within the resource opened in the editor. The data is requested with a Flux message and collected from reply messages
- **orion.edit.hover** plugin for providing hover tooltip data at the current offset. The data is requested with a Flux message and collected from reply messages and then forwarded to Orion.
- **orion.edit.command** plugin for performing various IDE related actions. The message is sent out to Flux that encapsulates the UI action. The reply messages can either be collected to forward data to Orion editor or action can be applied directly to the resource in the Cloud. For example organize Java imports can be performed by Flux JDT Service and changes can be synchronized by all interested Flux messaging system participants.

Problem #1: Orion plugins have one response

- For security reasons Orion plugins are very constrained in how they communicate with main page
- Orion requests data from plugins in REST kind of way. It asks the question and expects a single answer. This transforms into a Flux request message, but Flux can reply with a number of messages and replies are spread over time.
- Collecting reply messages for a short period of time and then combine them into a single data structure and present that to Orion is not a good way to solve this.

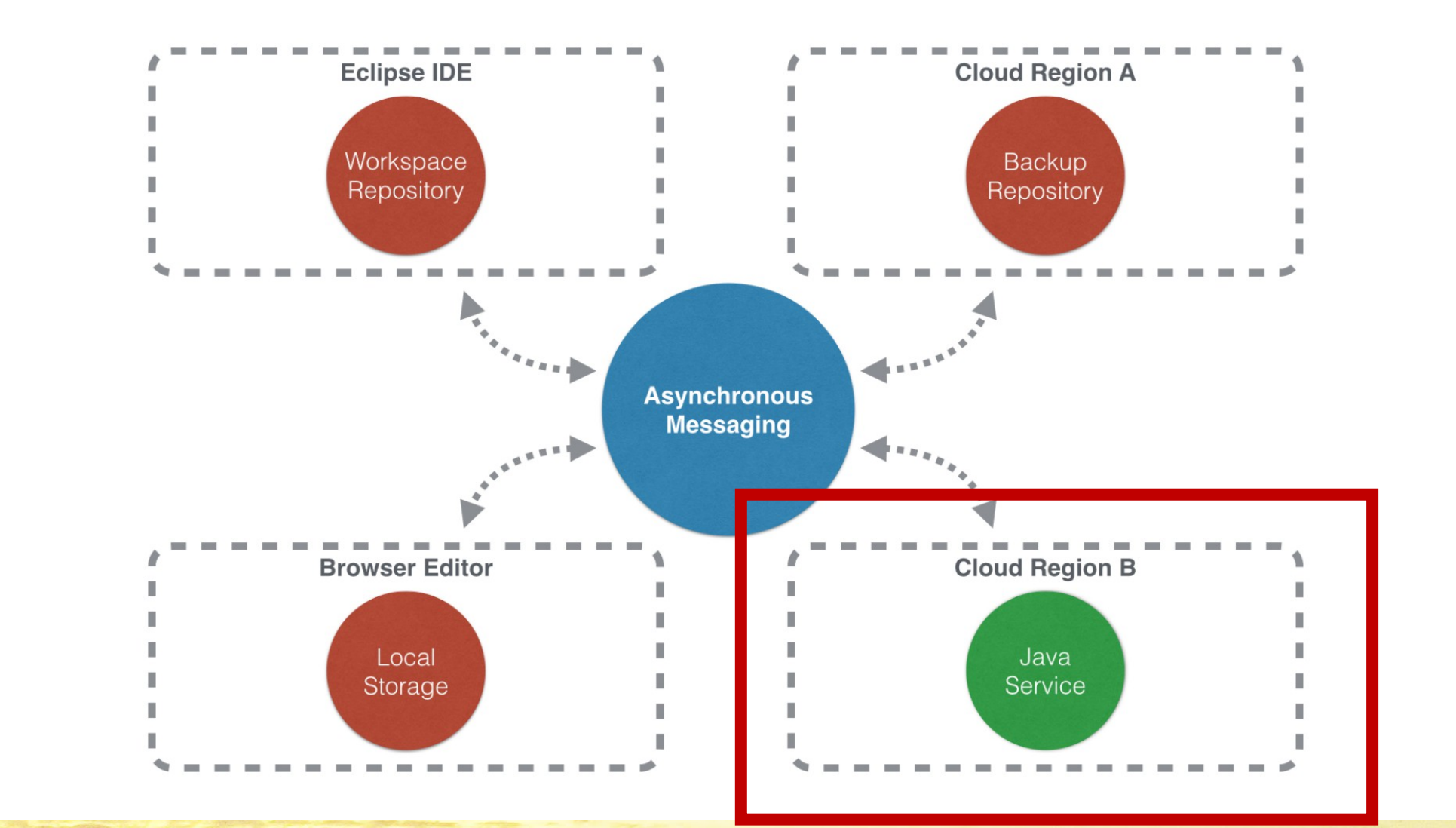
Problem #2: Don’t call us, we’ll call you

- Orion will ask a plugin for data when it needs it, but a plugin cannot pass the data to Orion whenever the data is ready.
- For example “orion.core.file” plugin may be asked to move a file if it’s moved in Orion. However if a file has been moved in the cloud, Flux message would arrive to “orion.core.file” plugin (to our Flux integration), but the plugin cannot signal the Orion UI to update for this change
- Orion needs an asynchronous message story!

Problem #3: where to apply edits

- Two approaches to tooling services that perform edits:
  - Apply edit directly in JDT service and just sync file change
  - Send edit description back to client for insertion (prototype approach)
- Examples: organize imports, quick fix, and content assist
- What if edit crosses resource boundaries? E.g., Spring IDE content assist may add a JAR to classpath
- Content assist might be split into enumerated proposals data and application would a Flux message to apply CA proposal

Flux prototype architecture



Flux JDT Service

- Headless Eclipse app running in cloud, made of:
  - Eclipse Workspace
  - Eclipse JDT core plugins
  - Flux message bus connection plugin
- Uses Flux file sync to keep workspace always in sync
- Handles Java tooling specific events from Flux bus:
  - Problem markers (errors/warnings)
  - Content assist
  - Jump to declaration
  - Javadoc hovers
  - Quick fixes

Problem #1: The singleton problem

- Eclipse workspace and JDT are full of singleton assumptions (one workspace, one user, one project with a given name, etc)
- Solution: one instance of JDT service per active user
- Each user has separate topic channel on Flux message bus

Questions:

- How to decide when to start JDT service for user?
- How long does it take to start JDT service in the Cloud?
- When does JDT Service need to be shut down?

Problem #2: Connecting services to users

- Deploying and starting a JDT services takes 1+ minutes
- When there are many users, how do we connect the right service to the right user

Blank JDT services

- Fully started JDT service
- Not connected to any user channel
- Connected to “service” channel (receives request messages from all user channels)
- Ignores all JDT tooling requests:
  - No file system syncing == blank workspace
  - No edit changes syncing
  - No replies for JDT specific messages

Flux service channel

Background:

- Communication in Flux is done via channels.
- Messages in a channel are generally not visible to other channels
- Request message is broadcasted on the user channel
- Reply messages come directly to the requester

- Service channel properties:
  - All request (broadcast) messages from user channels are also routed to Service channel
  - Used for communication between services where user is not participating
  - Admin credentials required to connect to this channel

Details on service lifecycle

- Client asks if there is a Java service available for them
- If there is an instance already loaded for that user, it says “service ready”
- Blank Java services will all respond with “service available”
- Client will periodically ask for service to prevent it shutting down
- JDT Service Provider service to maintain the size of the pool of “blank” services via Flux messages

Problem #3: Statefull Java tools

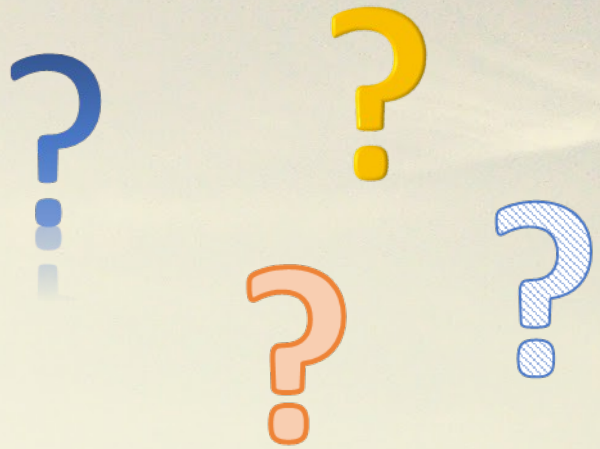
- Cloud Foundry treats app instances as interchangeable, ephemeral, and discardable
- CF instances designed for homogeneous scale-out, not per-user
- If JDT service pool is reduced, it may decide to shoot an active JDT instance rather than a blank/idle instance
- Fabric may decide to migrate an app instance to a different cluster
- To fit in this world JDT needs to **externalize state**
- For example move caches and Java model to external database

Future directions

- More refactoring needed in JDT to tease apart headless services from UI parts
- JDT moves so fast that forking it will too difficult to maintain
- Major decision point:
  - Do we peel off individual Java tooling services to make them multi-user
  - Continue on the path of instance per user
- Explore approach taken by Eclipse Che and look for ways to collaborate

Questions?

Slides are attached to session at eclipsecon.org



Please rate the talk to let us know if you found it valuable!