Developing for KEITH or LS

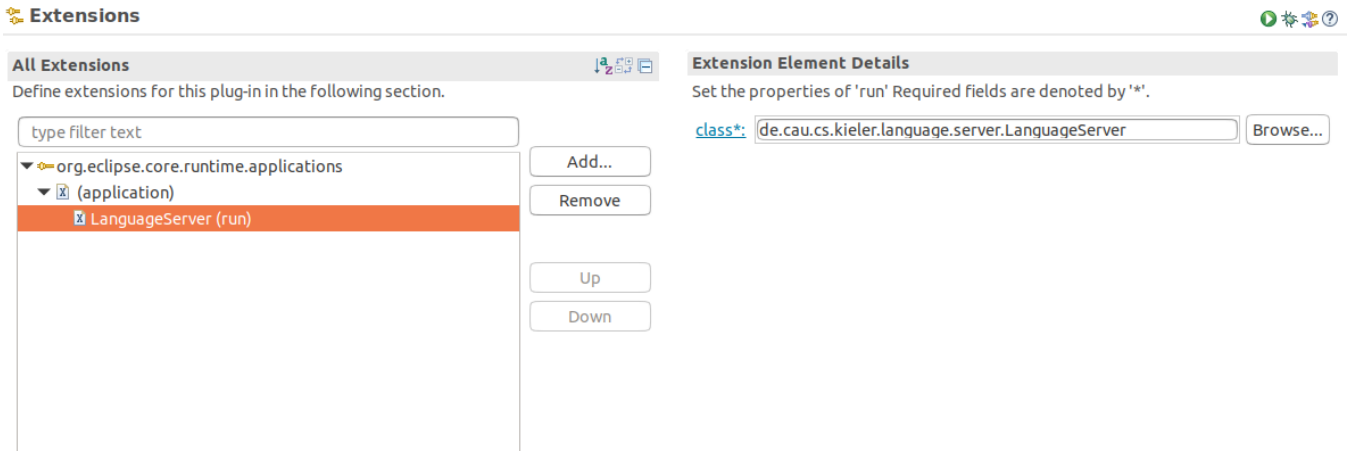
One the server side xtext is used to provide a language server. On the client side the Theia framework is used to communicate with it. This guide is for KIELER developers who want to build functinality in form of language server extensions for KEITH and frontend developers for KEITH that want to know how to communicate with the LS.

Starting the Language Server (LS)

This part of the guide covers the server side development for KEITH, the language server (LS). It is adviced to have a look at the existing implementation in the language.server plugin.

Register start hook (Eclipse)

Currently the LS is an eclipse application and it has to be started as one. To do this the Manifest.mf in the META-INF folder of your language server plugin has to be changed. In the extensions tab a new extension point for org.eclipse.core.runtime.applications has to be created. This extension points maps to your start class of your language server, which has to be an IApplication that implements a start method.



This start class should somehow distinguish between connecting via socket and connecting via stdin/out.

Language Registration

Language that are registered here are always xtext languages.

Register languages that are defined in the semantic

Since we are in the semantics repository we can use java ServiceLoader to add new ILSSetups, which register a language.

```
interface ILSSetup {
    def Injector doLSSetup()
}

class SCTXLSSetup implements ILSSetup {
    override doLSSetup() {
        return SCTXIdeSetup.doSetup()
    }
}
```

A language that wants to be included in the LS can implement this interface. Registering SCTXLSSetup via ServiceLoader allows to register all available languages like this:

```
for (contribution: KielerServiceLoader.load(ILSSetupContribution)) {
        contribution.LSSetup.doLSSetup()
}
```

Register that are defined outside of the semantic

Have a look at one of the LSSetups defined in the semantic.

Bindings

Bindings for the injector created by createLSModules and why they are needed:

- the KeithServerModule bind all standard stuff
- if we start via stdin/out, the ServerLauncher has to be bound to the LanguageServerLauncher
- we have to bind a ResourceRegistry and our own WorkspaceConfigFactory that allows to open a folder without xtext getting involved
- all bindings for the KGraphDiagramModule and KGraphDiagramServerModule are done to have a working diagram server

Starting and connecting

The following things are done via the LanguageServer class on startup of the LS:

- port and host argument are read. This decides whether the LS starts via socket or stdin/out
- socket
    - call bindAndRegisterLanguages (this loads all languages, languages defined in the pragmatic have to be added manually, all other ones are added via the KielerServiceLoader (see ILSSetup))
    - injector is created via createLSModules
    - a new socket is opened, buildAndStartLS is called (gets all ILanguageServerContribution via KielerServiceLoader and starts LS)
- stdin/out
    - main of LanguageServerLauncher is called
    - this calls bindAndRegisterLanguages
    - launch of ServerLauncher is called with injector created by createLSModules
    - this calls the start method which calls buildAndStartLS

Connection via socket

You have to specify a port and an optional host as VM arguments to start the LS via socket.

Currently the LS is an eclipse application, therefore you create a new Eclipse Application run configuration in eclipse.

In the `Main` tab, section `product to run` select Run an application and select `de.cau.cs.kieler.language.server.LanguageServer`.

In the `arguments` tab, section `VM arguments` add `-Dhost=localhost -Dport=5007` to the arguments. You might want to specify `Xmx` here too, if you plan to use bigger models (such as the railway controller).

Running this eclipse application requires a Theia client that tries to connect via socket to an LS running on port 5007.

Connection via stdin/stdout

Is only relevant for the product. Requeires to start to LS without an host or port Vm argument. Cannot be debugged.

Developing a LS extension

We use java ServiceLoader to register stuff. Here is a small example how a LanguageServerExtension is registered via a ServiceLoader and how it is used:

Register LanguageServerExtensions (ServiceLoader Example)

This is a LanguageServerExtension. It has to be used in the de.cau.cs.kieler.language.server plugin. Since the language-server-plugin should not have dependencies to all plugins that define a language server extension dependency inversion is used to prevent that. A ServiceLoader via dependency inversion does exactly that.

Here is such an example extension, the KiCoolLanguageServerExtension:

```
package de.cau.cs.kieler.kicool.ide.language.server


/**
 * @author really fancy name
 *
 */
@Singleton
class KiCoolLanguageServerExtension implements ILanguageServerExtension, CommandExtension,
ILanguageClientProvider {
        // fancy extension stuff

        KeithLanguageClient client
        // A language server extension must implement the initialize method,
        // it is however only called if the extension is registered via a language.
        // This should never be the case, so this is never called.
    override initialize(ILanguageServerAccess access) {
        this.languageServerAccess = access
    }

        // implement ILanguageClientProvider
    override setLanguageClient(LanguageClient client) {
        this.client = client as KeithLanguageClient
    }

        // implement ILanguageClientProvider
    override getLanguageClient() {
        return this.client
    }

}
```

The CommandExtension defines all commands (requests or notifications) that are send from client to server. An example how this looks like can be seen in the code snippet Example CommandExtension is an example how to define a server side extension interface.

The ILanguageClientProvider should be implemented by an extension that plans to send messages from the server to the client.

This language server extension is provided by a corresponding contribution, which is later used to access it:

```
package de.cau.cs.kieler.kicool.ide.language.server

import com.google.inject.Injector
import de.cau.cs.kieler.language.server.ILanguageServerContribution

/**
 * @author really fancy name
 *
 */
class KiCoolLanguageServerContribution implements ILanguageServerContribution {

    override getLanguageServerExtension(Injector injector) {
        return injector.getInstance(KiCoolLanguageServerExtension)
    }
}
```

Create a file called de.cau.cs.kieler.language.server.ILanguageServerContribution in <plugin>/META-INF/services/ (in this example this is de.cau.cs.kieler.kicool.ide). The name of the file refers to the contribution interface that should be used to provide the contribution. The content of the file is the following:

```
de.cau.cs.kieler.kicool.ide.language.server.KiCoolLanguageServerContribution
```

This is the fully qualified name of the contribution written earlier.

The language server uses all LanguageServerExtensions like this:

```
var iLanguageServerExtensions = <Object>newArrayList(languageServer) // list of all language server
extensions
for (lse : KielerServiceLoader.load(ILanguageServerContribution)) { // dynamically load all contributions to
add LS extensions
        iLanguageServerExtensions.add(lse.getLanguageServerExtension(injector))
}
```

The resulting list of implementions is used to add the extensions to the language server.

The interfaces used for dynamic registration are in the semantics repository. If you define a pragmatics LS extension you have to statically add these extensions to your list.

Register an extension (on server side)

See example above for ServiceLoader and initial stuff.

What is still missing are the contents of the CommandExtension implemented by the KiCoolLanguageServerExtension. This is an interface defining all additional commands. The CommandExtension looks like this.

Example CommandExtension

```
package de.cau.cs.kieler.kicool.ide.language.server

import java.util.concurrent.CompletableFuture
import org.eclipse.lsp4j.jsonrpc.services.JsonRequest
import org.eclipse.lsp4j.jsonrpc.services.JsonSegment

/**
 * Interface to the LSP extension commands
 *
 * @author really fancy name
 *
 */
@JsonSegment('keith/kicool')
interface CommandExtension {

    /**
     * Compiles file given by uri with compilationsystem given by command.
     */
    @JsonRequest('compile')
    def CompletableFuture<CompilationResults> compile(String uri, String clientId, String command, boolean
inplace);

    /**
     * Build diagram for snapshot with id index for file given by uri. Only works, if the file was already
compiled.
     */
    @JsonRequest('show')
    def CompletableFuture<String> show(String uri, String clientId, int index)

    /**
     * Returns all compilation systems which are applicable for the file at given uri.
     *
     * @param uri URI as string to get compilation systems for
     * @param filter boolean indicating whether compilation systems should be filtered
     */
    @JsonRequest('get-systems')
    def CompletableFuture<Object> getSystems(String uri, boolean filterSystems)
}
```

This defines three json-rpc commands: "keith/kicool/compile", "keith/kicool/show", "keith/kicool/get-systems". These are implemented in KiCoolLanguageServerExtension.

Server Client communication interface

Not only messages from client to server but rather messages from server client might be needed.

Messages that can be send from server to client are defined in the KeithLanguageClient:

Example KeithLanguageLCient

```
/**
 * LanguageClient that implements additional methods necessary for server client communication in KEITH.
 *
 * @author really fancy name
 *
 */
 @JsonSegment("keith")
interface KeithLanguageClient extends LanguageClient {

    @JsonNotification("kicool/compile")
    def void compile(Object results, String uri, boolean finished);

    @JsonNotification("kicool/cancel-compilation")
    def void cancelCompilation(boolean success);

        // Not only notifications, but also server client requests should be possible, but currently there
is no use case for that.
}
```

These messages can be caught on the client side by defining the message that is caught like this:

Client side message definition

```
export const snapshotDescriptionMessageType = new NotificationType<CodeContainer, void>('keith/kicool
/compile');
```

This message type is bound to a method that should be called whenever the client receives such a message.

Client side message registration

```
const lClient: ILanguageClient = await this.client.languageClient
lClient.onNotification(snapshotDescriptionMessageType, this.handleNewSnapshotDescriptions.bind(this))
```

The method should receive all parameters specific in the KeithLanguageClient interface on the serevr side.

Such a notification from server to client is send like this:

Server side message sending

```
future.thenAccept([
        // client is the KeithLanguageClient registered in a LanguageServerExtension that implements a
ILanguageClientProvider
        // compile is the command defined in the KeithLanguageClientInterface
        client.compile(new CompilationResults(this.snapshotMap.get(uri)), uri, finished)
])
```

Register and calling an extension (on client side)

Language server extension do not have to be registered on the client side. It is just called.

You can send a request or a notification to the language server like this:

Client side message sending

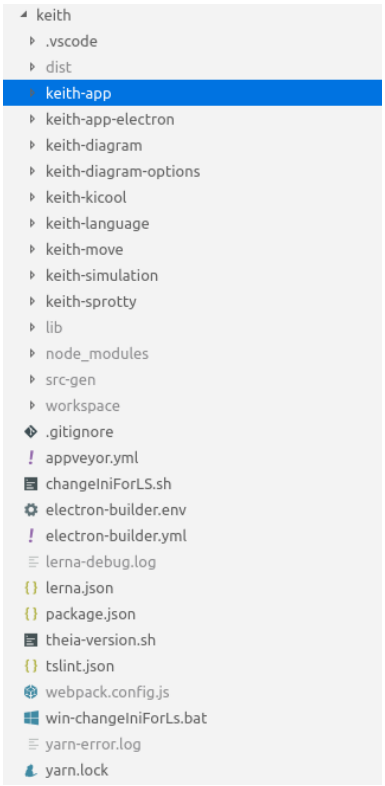
```
const lclient = await this.client.languageClient
const snapshotsDescriptions: CodeContainer = await lclient.sendRequest("keith/kicool/compile", [uri,
KeithDiagramManager.DIAGRAM_TYPE + '_sprotty', command,
        this.compilerWidget.compileInplace]) as CodeContainer
// or via a thenable
client.languageClient.then(lClient => {
lClient.sendRequest("keith/kicool/compile").then((snapshotsDescriptions: CodeContainer) => {
        // very important stuff
}
// await is preferred, since it is shorter.
```

In this example client is an instance of a language client. It is usually injected like this:

Client side LanguageClientContribution injection

```
@inject(KeithLanguageClientContribution) public readonly client: KeithLanguageClientContribution
constructor(
        // other injected classes that are relevant for the constructor
    ) {
        // constructor stuff
}
```

How to make a new package for KEITH



Clone the KEITH repository.

Open the keith folder in VSCode. You are know in the keith directory in VSCode (see picture an the right).

Create a new folder called keith-<your extension name>.

Copy a package.json, a tslint.json, a tsconfig.json, and a src folder into the folder.

Add keith-<your extension name> to workspaces in the top level package.json.

Add "keith-<your extension name>": "<your-version (e.g. 0.1.0)>" to the dependencies in the top level package.json and the product package.json files (e.g. the package.json in keith-app).

What is in the src directory?

The source directory has three optional subfolders.

- node: Holds all backend related classes. This does currently only exist in the keith-language package.
- common: Holds general helper methods, string constants and some data classes
- browser: Holds all widgets, contribution for commands, menus, and widgets, and the frontend-extension.

The frontend-extension

This binds all necessary classes. Look at existing frontend extension in KEITH or Theia to see how this is done.

More examples for stuff

See Theia examples.

How to write a widget

There are different kinds of widgets that are commonly used in KEITH or in existing Theia packages.

- BaseWidget: Very basic
- ReactWidget: A render method has to be implemented that redraws the widget on demand. Additionally several on* event methods can beimplemented.
- TreeWidget: Extends the ReactWidget and draws the contents of the widget in a tree view.

If a widget has a state it should implement the StatefulWidget interface, which allows to imlement a store and restore method.

Look at examples in KEITH or Theia to see how this is done.

How to make a new module for sprotty (see actionModule, ...)

WIP

How to use (Kieler)ServiceLoader

Classes that are provided via a ServiceLoader have a contribution that provides them. This contributions share a common interface lets call it `IStuffContribution`. A specific StuffContribution (it provides stuff) is the ImportantStuffContribution.

```
// defined in some common package
interface IStuffContribution {

    abstract def IStuff getStuff()
}
// defined in the package that holds ImportantStuff
class ImportantStuffContribution implements IStuffContribution {
    override getStuff() {
        return new ImportantStuff()
    }
}
```

The ServiceLoader has to know that some plugin provides an implementation for IStuffContribution. Therefore, a folder named services is added next to the corresponding MANIFEST.MF.

In the services folder a file named the same as the fully qualified name of the implemented interface is added. Here this one is called `de.cau.cs.kieler.basic.package.IStuffContribution`.

The file holds the fully qualified names of all implementations of this interface:

```
de.cau.cs.kieler.importatn.package.ImportantStuffContribution
```

Now you can access implementation of IStuff as follows:

```
var stuffList = newArrayList
for (stuffContribution : KielerServiceLoader.load(IStuffContribution)) { // dynamically load all
contributions that provide stuff
        // Add all stuff to a list of stuff
        stuffList.add(stuffContribution.getStuff())
}
```