Rivet: Dependency Injection in Rust

CS242 Final Report

Michael Diamond
diamondm@stanford.edu

Matthew Vilim
mvilim@stanford.edu

1 SUMMARY

Rust's strict compile-time semantics make approaches that are often straightforward in other modern languages significantly more difficult or less intuitive. Rust's concepts of ownership and data lifetimes, along with its limited runtime reflection support constrain users hoping to design loosely coupled and pluggable applications. We explored various approaches to the strategy pattern[7] and dependency injection[6] (DI) in Rust, providing a qualitative evaluation of each, discussing their strengths and weaknesses. We rank each approach explored in terms or several metrics: safety, expressiveness, flexibility, maintainability, and debugability. We created a pluggable web server framework, called Rivet[8], as a testbed to explore and demonstrate these different approaches, and identified several promising strategies and areas for future research.

2 BACKGROUND

As software applications grow in size, tightly-coupled dependencies between conceptually isolated components introduce cognitive overhead and unnecessary complexity, forcing developers to understand large swaths of the application in order to safely make further changes to any part of the system.

The strategy pattern is a useful tool for combating tight coupling; by registering distinct components with a centralized dispatcher each component can be developed in isolation. The dispatcher takes responsibility for application-wide tasks such as managing resource contention, but routes units of work to the appropriate strategy for processing. This allows the dispatcher to remain decoupled from the actual work that needs to be done, and the individual strategies do not require any awareness of each other.

However, this pattern introduces two significant issues. First, it requires all strategies implement the same API(s) in order for the dispatcher to be able to invoke them (e.g. Java's `Runnable` interface), which constrains the flexibility of the plugin's APIs and often requires the use of a monolithic "`StrategyContext`" type that contains any information a strategy might possibly need. Second, it doesn't resolve the issue of tight-coupling in the form of global state. Any data or resources not provided by the dispatcher generally can only be retrieved from the application's global state. This leads to brittle, difficult to test code that cannot be decoupled from the application's full behavior.

Dependency injection is a technique widely employed to address both of these concerns. At a high level dependency injection is a type of inversion of control (IoC) — application code is *given* the resources and data it depends on, rather than constructing or owning it directly.

A simple example of code not using dependency injection is shown in Listing 1. A corresponding implementation using DI is shown in Figure 2.

Listing 1: Dependency is tightly-coupled to its usage

```
class DatabaseUpdater() {
  private final Database db =
      new Database(USERNAME, PASSWORD);

  public updateDatabase(...) { db.write(...); }
}
```

Listing 2: Loosely-coupled, dependencies are passed in

```
class DatabaseUpdater() {
  private final Database db;

  public DatabaseUpdater(Database db) { this.db = db; }

  public updateDatabase(...) { db.write(...); }
}
```

Decoupling resource construction from where it's used makes `DatabaseUpdater` simpler to work with, but it has only offloaded the task of actually constructing the `Database` to some other part of the application. Manually constructing all necessary dependencies and passing them in is possible but can quickly become tedious and error-prone. In order to address this issue, dependency injection frameworks exist to support recursive dependency resolution. The dependency "factory" knows how to generate an object and all its dependencies, limiting the dependency construction information to single place far away from where the dependencies will be used.

2.1 Webservers

Developing a web server (or any sort of RPC server) is a classic use case for a strategy pattern as it enables developers to separate the role of the server framework (receiving requests and sending responses) from how those individual requests are handled. Individual request paths (e.g. `/foo.html` and `/bar?baz=true`) can be processed by entirely isolated units of code, which have no knowledge of each other or any other paths that the server is able to handle. These requests may also be structurally different and require different dependencies in order to be properly completed; therefore, a flexible dependency injection pattern allows individual strategies to work with just the dependencies they need.

Many modern open source web frameworks take this approach[1], including Python's Django and web.py projects, which allow individual plugins to separately specify the data they require from the request, rather than conforming to a single (or finite) API. For example, web.py allows arbitrary URL "chunks" to be passed as arguments to a plugin's handler function.

There are also several existing web frameworks for Rust[9] (in different stages of development), notably including Rocket[5] which

[1] Contrast this with languages like PHP which take a one-script-per-page approach, or lower-level tools that simply accept requests and return responses, without any meaningful dispatching, decoupling, or type-safety.

we investigated as it implements an elegant strategy and dependency injection pattern.

2.2 Prior Art

Several dependency injection approaches have already been created for Rust [4] [1] [2], however these projects all appear to be experimental with varying robustness. To our knowledge there is not a general-purpose DI library developers can incorproate into their applications today.

- **rust-ioc** This project provides a factory container with support for resolving a dependency graph at compile time; this crate is the most well-maintained and documented. A full summary of its features is available at https://github.com/KodrAus/rust-ioc/blob/master/README.md.
- **di-rs** This repository provides DI with a slight twist inspired by a JavaScript framework[3]; it supports restricted lifetimes with scoping as discussed in Section 3.6 and provides multi-threaded support.
- **hypospray** This project looked the most promising in terms of features and expressibility and is the only of the three to use code generation; however, the project is broken[2] and no longer seems to be maintained. Dependency graphs are checked at compile time, and it supports injecting up to five dependencies at a time[3].

Rocket is notable also as it provides a generic plugin mechanism that relies on code generation to automatically create the coupling between the user's decoupled components. Rocket also provides a DI factory API they call *Managed State*[4], an example of which is shown in Listing 3. This example indicates the decoupling that Rocket's code generation provides; the `index` and `count` functions are injected with a user-defined structure `HitCount`, receiving only the data they need to complete the request.

Listing 3: Example of Rocket's *Managed State* interface

```
struct HitCount(AtomicUsize);

#[get("/")]
fn index(hit_count: State<HitCount>) -> &'static str {
  hit_count.0.fetch_add(1, Ordering::Relaxed);
  "Your visit has been recorded!"
}

#[get("/count")]
fn count(hit_count: State<HitCount>) -> String {
  hit_count.0.load(Ordering::Relaxed).to_string()
}

fn main() {
  rocket::ignite()
    .mount("/", routes![index, count])
    .manage(HitCount(AtomicUsize::new(0)))
    .launch()
}
```

As a concrete example of where Rust developers might need Dependency Injection, Rocket's lead developer Sergio Benitez directed us to an open issue[5] tracking implementing some sort of solution for dependencies which require configuration (such as database connections). Presently in Rust and Rocket the experience here is fairly poor because existing approaches rely heavily on code generation, which means in order to reconfigure such resources (e.g. update the database connection information) the application must be recompiled. At this time this remains an open feature request for Rocket, but the specified behavior boils down to runtime dependency injection using data that is not known at compile time.

[2]https://github.com/jonysy/hypospray/issues/3
[3]https://github.com/jonysy/hypospray/blob/2de8cb698/src/graph/ext.rs#L130
[4]https://rocket.rs/guide/state/
[5]https://github.com/SergioBenitez/Rocket/issues/167

3 APPROACH

Using the `tiny-http`[10] Rust HTTP library as a starting point we implemented a server meta-framework that allowed us to explore different strategy patterns and dependency injection mechanisms. `tiny-http` represents each HTTP request and response via a pair of structs appropriately named `Request` and `Response`. These encapsulate everything the server developer might need to inspect or mutate, but they're fairly difficult to work with in practice unless all valid requests are homogeneous (e.g. serving static content from disk, based solely on the URL). Heterogeneous request handling (e.g. different "sections" of a website serving separate types of content) calls for a more elegant decoupling.

We defined a simple `Responder` trait, shown in Listing 4, which our different approaches would implement, in order to first decouple the server's low-level request processing from the approaches that would handle individual requests.

Listing 4: The meta-framework's `Responder` trait

```
use tiny_http::{Request, ResponseBox};

pub trait Responder {
    fn handle(& self, &Request) -> ResponseBox;
}
```

A simple routing mechanism, keyed off the first `/`-delimited segment of the requested URL, was used to dispatch requests to individual responders. These responders, in turn, exposed some *different* (and ideally better) API than the `Request` and `Response` structs for handling the requests routed to them.

For the sake of simplicity, our examples primarily looked at ways of providing the URL's path and query parameters more cleanly. Our approaches could be expanded to any other request/response data (HTTP method, hostname, cookies, headers, etc.) or other dependencies (such as database hooks) by simply replicating what was done for the path and query data. Where that isn't the case is called out below as a disadvantage.

3.1 Qualitative Metrics

Fundamentally our investigation is subjective — which approaches make for a better development experience than simply processing raw requests, while also working within Rust's constraints and following its best practices.

In order to be somewhat qualitative and offer a more meaningful result than simply "better" or "worse" we identified five axes on which our approaches will be compared. In some cases these axes are at odds with each other — for example an API that offers greater flexibility will generally sacrifice safety or maintainability in order to do so.

- **Safety** How error-prone is the approach? (e.g whether issues cause compile-time failures or runtime failures, or introduce overhead such as suboptimal memory lifecycles)
- **Expressiveness** How easy is it to work within the framework? (e.g. minimal boilerplate or linguistic constraints)
- **Flexibility** How easy is it to add new dependencies with minimal change? Are there constraints on the types of dependencies or their lifetimes?
- **Maintainability** The ability to make changes to certain code paths without fear of affecting other paths
- **Debug-ability** Being able to clearly diagnose failures and their root causes

A summary of our results in terms of these axes is depicted in Table 1.

3.2 Simplified

Our first step was to simply hide the `Request` object, as it's much larger and more cumbersome than many applications need. For simple use-cases it could be sufficient to simply pull out the basic information the developer needs and provide it to them via a more expressive API — specifically:

Listing 5: A simpler API

```
fn respond(url_parts: &Vec<String>,
           query_params: &HashMap<String, String>) -> String
```

Hiding the `Request` and `Response` types like this gives the user a smaller API surface — they don't have to worry about the HTTP specification or other complexities of a general-purpose server. Instead, they simply process the URL and generate a string to send as the response, which the framework then converts into a proper HTTP response.

However, it offers little type-safety beyond what `Request` already provided, and notably is a far more severely rigid API. There's no mechanism to expose additional data or dependencies (such as the request cookies) short of redefining the function's API. In addition to being inflexible, it provides no way for the user to compartmentalize different request types and requires the user to write boilerplate (such as an if-else chain) to handle different request types separately, leading to difficult to maintain and overly-verbose code. Any sort of URL parsing (e.g. extracting a numeric identifier from the path) must be done manually, which is both tedious and error-prone.

While broadly unsuitable, this example demonstrates the general principle of hiding the HTTP layer from the user, and there are low-hanging-fruit to resolve some of the concerns above.

3.3 Pattern

The first improvement is to make request routing a feature of the framework, rather than something the user is responsible for. To do so, we created a registry of regular expressions matching different URL patterns and paired this with a callback function `fn(&regex::Captures, &HashMap<String, String>) -> String` which would be invoked if the request URL matched the given pattern.

Listing 6: Using patterns to route heterogenous requests

```
lazy_static! {
    // Note that the order matters -
    // the first matched pattern will be used
    static ref ROUTES: Vec<Route> = vec![
        Route::new("/foo/([^/]*)", handle_foo),
        Route::new("/bar/(\\d+)", handle_bar),
        Route::new("", handle_root)
    ];
}

struct Route {
    path: regex::Regex,
    callback: fn(&regex::Captures, &HashMap<String, String>)
        -> String
}

fn handle_foo(path_captures: &regex::Captures,
        query_params: &HashMap<String, String>) -> String {
    format!("Captures: {:?}\nQuery Params: {:?}",
        url_captures, url_params)
}
```

While this has a type signature similar to the "Simplified" approach, it's far more expressive and maintainable, as users can specify arbitrary segmenting of the request space to be sent to different callbacks. These callbacks do not need to be aware of one another or interact in any way, making ongoing maintenance much simpler and safer. Furthermore, by decoupling request routing from the request handling functions users have *some* flexibility in how their code is invoked. While adding additional dependencies to this callback API is non-trivial — every callback function would need to be updated with an additional parameter — specifying a closure as the callback function (e.g. `|captures, _params| simple_handler(captures)`) allows the `simple_handler` function to only specify the data it actually requires, not the whole request.

Similarly, using capturing groups of a regular expression rather than the raw URL path gives the developer more confidence that the requests they're handling are well-formed and can be safely processed. For instance, the `handle_bar` handler above can be confident that it will always be invoked with a capture parameter containing exactly one group consisting of digits. This could be made even more type-safe and expressive with a higher-level pattern matching abstraction that validates and parses the captured inputs into the desired types before invoking the user's callback.

By routing requests and validating them before invoking the user's callback, the framework is easier to debug as well. While users are still responsible for some amount of input processing, the majority is handled by the framework, and runtime errors (such as failing to parse an invalid input that makes it past the regular expression) can be handled with standard Rust idioms such as `match`'ing the parse results.

3.4 Closure

From here we clearly needed to investigate more flexible options — dependency injection. We started by looking at an existing pattern[6] implemented by rust-ioc (the "ioc" referring to "inversion of control"), which they achieve by using closures to separate resource construction from invocation. Conceptually, their pattern is a form of partial application; associating a resource with a stateful type that then accepts the data the resource should operate on. It also

[6] https://github.com/KodrAus/rust-ioc/blob/master/factories

Table 1: Qualitative comparison of differing mechanisms investigated

Approach	Safety	Expressiveness	Flexibility	Maintainability	Debugability
Simplified	low	low	none	low	moderate
Pattern	moderate	moderate → high	low	high	high
Closure	high	low → moderate	moderate	high	moderate
Factory	moderate	low	low	moderate	low
Scope	high	low	low	moderate	low
Traits	high	high	moderate	high	low → moderate

takes advantage of the `impl Trait`[7] syntax, which we did not explore here.

By emulating this approach we can similarly decouple the user's function(s) from the callback API used by the framework itself:

Listing 7: Passing dependencies via closures

```
fn handle(&self, request: &Request) -> ResponseBox {
    let path = ...;
    let query = ...;

    let cb: Box<Fn() -> ResponseBox> = match path.first() {
        Some(part) => match part.as_ref() {
            "path" => Box::new(
                || util::success(&params_only(path))),
            "query" => Box::new(
                || util::success(&query_only(query))),
            "both" => Box::new(
                || util::success(&both(path, query))),
             _ => Box::new(|| util::fail404("Not found!"))
        },
        None => Box::new(|| util::success(&root()))
    };

    // framework invokes the callback later
    cb()
}

fn params_only(params: &Vec<String>) -> String { ... }

fn query_only(query: &HashMap<String, String>)
    -> String { ... }
```

The `handle` function is responsible for pulling out any data or other resources from the request, then dispatching[8] the request to the appropriate callback. With this technique we're able to use a single simple callback signature (taking no explicit arguments) but pass in whatever parameters the user expects by closing over them. This is far more flexible than our prior approaches. Not only can the user cleanly define whatever function signature(s) they'd like to use, but furthermore, there are no changes required to the existing functions if additional data is made available in the future.

The benefit of using closures (vs. just directly invoking the user's functions) is the inversion of control demonstrated in the rust-ioc project. The framework gets a `cb` callback it can invoke when it's ready to do so, without concern for which dependencies the user's code requires.

In this implementation the dense closure boilerplate somewhat impacts the expressiveness of the approach, but this block could conceptually be generated by a script, macro, codegen, or other tool, as each line takes the same form based on the arguments of the function being called. Instead of manually listing all matches in one place each function could be annotated with the path or pattern it's intended to be invoked with, and the `handle` function would be generated based on the annotation and function parameters.

This approach would come with some tradeoffs, notably if the dispatching logic is being generated, it will be difficult for users to trace. Additionally, it could constrain what types could be dynamically injected, since the code inspection would need to be able to determine which instances should be injected based solely on the function signature. If two values with the same type needed to be injected, some more cumbersome mechanism to differentiate them would need to be introduced, which would both complicate the user experience and potentially lead to confusing debugging experiences.

Despite these issues, this closure-based approach shows general promise as a way to work within Rust's constraints while enabling dynamic and flexible user-facing APIs.

3.5 Factory

This method of DI works as described above by providing a single location where all dependencies are defined — essentially associating instances with a unique key they can later be retrieved with. A dependency can be added with the `add` function by tagging it with a unique string; to resolve the dependencies, the same string can be provided to `resolve` to construct a new instance of that object. The definition of the container is shown in Listing 8; internally, dependencies are tracked in a `HashMap<String, Box<Any>>` and downcast at resolution time. Shared dependencies can be stored in the map as `Rc<_>`, and even mutable shared references can be stored as `Rc<RefCell<_>>` using Rust's concept of interior mutability and runtime borrowing.

[7] https://github.com/rust-lang/rfcs/blob/master/text/1522-conservative-impl-trait.md

[8] for brevity here we simpy routed requests based on the first segment of the URL path, but a proper implementation could take advantage of the more powerful pattern-based dispatching described above.

Listing 8: Factory container definition

```
struct Container {
    constructors: HashMap<String, Box<Any>>,
}

impl Container {
    fn new() -> Container {
        Container { constructors: HashMap::new() }}

    fn add<T: Constructors<T> + 'static>
      (&mut self, s: &str, value: T) {
        self.constructors.insert(s.to_string(),
          Box::new(value.construct()) as Box<Any>);
    }

    fn resolve<T: Clone + 'static>(&self, s: &str) -> T {
        let item = self.constructors.get(s).unwrap();
        let construct =
          item.downcast_ref::<Construct<T>>().unwrap();
        construct.c()
    }
}
```

The actual construction itself takes place through the following traits shown in Listing 9. An example of usage within a web server plugin in shown in Listing 10. While this approach does provide a DI factory scheme, it suffers from a number of problems:

- All dependencies registered in the map must implement the `Clone` trait. This limitation is a result of the fact that at the time of dependency creation, the constructor is unknown. Thus, dependencies are required to provide a method of duplication. In the case of shared dependencies using `Rc<_>`, this `Clone` serves simply to increment the reference count. An alternate DI scheme that instantiates dependencies on the fly through provided constructors is explored in Section 3.6.
- Dependency lifetimes are static and will last as long as the factory container exists. The brittle nature of dependency declarations in this scheme entails that dependencies' lifetimes are not simply limited to their natural lifetimes. A more ideal solution would tie dependency lifetimes to a scope associated with that of their parents as explored in 3.6.
- This method also suffers from runtime safety issues. The user must explicitly annotate the type of the dependency to be resolved as shown in Listing 10. An error by the user in specifying the type to be retrieved would result in a run-time error.
- There is also a fair amount of boilerplate involved in wiring dependencies as shown in the example; reducing boilerplate would require code generation or macros.
- As the dependency graph grows, the origin of runtime failures discussed above become difficult to pinpoint.

Listing 9: Factory constructor traits

```
struct Construct<'a, T> { build: Box<Builder<T> + 'a> }

impl<'a, T> Construct<'a, T> {
    fn c(&self) -> T { self.build.c() }
}

trait Builder<T> { fn c(&self) -> T; }

impl<T: Clone> Builder<T> for T {
    fn c(&self) -> T { self.clone() }
}
```

Listing 10: Factory usage in plugin

```
pub struct Factory { container: Container }

impl responders::Responder for Factory {
    fn new() -> Factory {
      let mut c = Container::new();
      let count = Rc::new(RefCell::new(0));
      c.add("count", count);
      Factory { container: c }
    }

    fn handle(&mut self, request: &tiny_http::Request)
      -> tiny_http::ResponseBox {
        let url_parts =
          util::strip_url_prefix(request.url(), "/factory");

        self.container.add("url_parts", url_parts);
        let count: Rc<RefCell<i32>> =
          self.container.resolve("count");
        *count.borrow_mut() += 1;
        util::success(&format!("Count {:?}", count))
    }
}
```

3.6 Scope

One of the principle shortcomings of the method discussed in Section 3.5 is the inability to tie dependent objects' lifetimes to that of their parents. The following method based on [1] solves this issue by providing a way to scope object lifetimes. The method is based on a slightly altered method of DI described by the author of [1] [3]. Rather than pulling dependencies from a factory container, the entire dependency tree is constructed on demand and tied to the parent. The `Scope<T>` struct shown in Listing 11 is used to hold an object and all of its constructed dependencies. The `Dependencies` struct in Listing 12 maintains a `HashMap` mapping each dependency tagged by the user with a unique string to a set of closures that are constructors for its dependent objects. When a dependency is resolved by calling `resolve`, each constructor is called, and the dependent objects are returned. These closures themselves may contain calls to resolve dependencies. The dependency graph is defined by calling `add`, providing the constructor for that dependency.

Listing 11: Dependency scoping

```
pub struct Scope<T> {
    pub parent: T,
    children: Vec<Box<Any>>,
}

pub trait Resolve<T> {
    fn resolve(self, s: &str, deps: &Dependencies)
      -> Scope<T>;
}

impl<T: Any> Resolve<T> for T {
    fn resolve(self, s: &str, deps: &Dependencies)
      -> Scope<T> {
        deps.run_constructors(s, self)
    }
}
```

This method does provide advantages in limiting the lifetimes of dependent objects in comparison to the factory container, but it also has drawbacks of its own. Due to the fixed nature of the closure function signature, constructed objects can only be injected with at most one dependency. This restriction is severely limiting in cases where an object may depend on more than one object. Later

Listing 12: Scoped dependency resolution and construction

```
pub struct Dependencies { constructors: HashMap<String, Vec<Box<Fn(&Dependencies, &Any) -> Box<Any>>>> }

impl Dependencies {
    pub fn new() -> Dependencies { Dependencies { constructors: HashMap::new() }}

    pub fn run_constructors<P: Any>(&self, s: &str, parent: P) -> Scope<P> {
        match self.constructors.get(s) {
            Some(list) => {
                let deps: Vec<_> = list.iter() .map(|construct| construct(&self, &parent)) .collect();
                Scope { parent: parent, children: deps }
            },
            None => Scope { parent: parent, children: vec![] },
        }}

    pub fn add<P, C, F>(&mut self, s: &str, constructor: F)
        where P: 'static + Any, C: 'static + Any, F: for<'r> Fn(&'r Dependencies, &P) -> C + 'static {
        match self.constructors.entry(s.to_string()) {
            Entry::Occupied(mut list) => { list.get_mut().push(box_constructor(constructor)); },
            Entry::Vacant(e) =>          { e.insert(vec![box_constructor(constructor)]); },
        };}
}

fn box_constructor<P, C, F>(constructor: F) -> Box<Fn(&Dependencies, &Any) -> Box<Any>>
  where F: for<'r> Fn(&'r Dependencies, &P) -> C + 'static, P: 'static + Any, C: 'static + Any {
    Box::new(move |deps: &Dependencies, parent: &Any| -> Box<Any> {
        let concrete_parent = parent.downcast_ref::<P>().unwrap();
        Box::new(constructor(deps, concrete_parent))
    })
}
```

versions of [1] overcome this limitation by providing hard-coded functions for one dependency, two dependencies, etc. This method also suffers from the same shortfalls in terms of expressiveness, flexibility, and debugability discussed in Section 3.5.

3.7 Traits

Inspired by the factory approach discussed in 3.5, we wanted to provide the same heterogenous storage pattern, but with more robust type safety. Unlike many other object-oriented languages Rust's traits do not require unique method names in order to be implemented together, which meant it might be possible to use a set of traits to expose type-safe views into the values of a map of `Any` objects.

Listing 13: Traits provide type-safe views

```
struct DepStore { store: HashMap<String, Box<Any>> }

trait PathParts { fn get(&self) -> &Vec<String>; }
impl PathParts for DepStore {
    fn get(&self) -> &Vec<String> {
        self.store.get("PathParts").unwrap()
            .downcast_ref::<Vec<String>>().unwrap()
    }
}

trait Query {
    fn get(&self) -> &HashMap<String, String>;
}
impl Query for DepStore {
    fn get(&self) -> &HashMap<String, String> {
        self.store.get("Query").unwrap()
            .downcast_ref::<HashMap<String, String>>()
            .unwrap()
    }
}
```

This allows us to hide the boilerplate of retrieving values from the backing store behind traits, enabling users to declare function signatures that take any number of different traits, and the framework can pass in the same instance (the map struct) for all of them:

Listing 14: The backing store can be passed as different traits

```
fn handle(&self, request: &Request) -> ResponseBox {
    let deps = DepStore::new(...);
    util::success(&dispatch(&deps, &deps))
}

fn dispatch<P: PathParts, Q: Query>
        (path: &P, query: &Q) -> String {
    let path = path.get();
    let query = query.get();
    ...
}
```

This pattern provides a number of benefits in terms of both safety and flexibility, but actually implementing it would be quite verbose — users would need to declare a new `trait` and `impl` for each type they want to inject, and might need to do so repeatedly if they need to `impl` a trait for multiple variants of the backing store (e.g. if certain objects should only be available to a subset of requests). To reduce this excessive boilerplate, we introduced a set of macros to generate these types automatically. These macros are included in Listing 15.

By hiding the boilerplate behind macros we end up with a concise yet highly flexible API for type-safe dependency injection:

Listing 15: Macros for trait-based dependency injection

```
/// Constructs a "binder", a struct that can hold arbitrary types, installed via the bind! macro.
///   Usage: binder!(BinderTypeName)
///     BinderTypeName:  Name of the struct to define.
macro_rules! binder {
    ($store:ident) => {
        struct $store { store: HashMap<String, Box<Any>> }
        impl $store {
            fn new() -> $store { $store { store: HashMap::new() } }
        }}}

/// Binds a value to to a binder instance - effectively just a wrapper for
///   BindingTrait::put(&mut binder, value)
/// but can be used for consistency with the other macro APIs
///   Usage: bind!(store, BindingTrait, Binding)
///     BinderInstance:  A Binder instance, where the binding will be stored
///     BindingTrait:    Trait which will provide Binding
///     Binding:         Instance to bind to the BindingTrait
macro_rules! bind {
    ($map:ident, $bnd:ident, $value:expr) => {
        $bnd::put(&mut $map, $value);
    }}

/// Registers a binding, creating a trait with the given name
///   Usage: binding!(BinderType, BindingTraitName, BindingType)
///     BinderType:        A binder type, created by binder!()
///     BindingTraitName:  Trait to create that will provide the given binding
///     BindingType:       Type that BindingTrait will provide
macro_rules! binding {
    ($store:ident, $name:ident, $ty:ty) => {
        trait $name { fn get(&self) -> &$ty; fn put(&mut self, value: $ty); }

        impl $name for $store {
            fn get(&self) -> &$ty {
                match self.store.get(stringify!($name).into()) {
                    Some(dep) => { match dep.downcast_ref::<$ty>() {
                        Some(dep) => dep,
                        None => panic!("Could not downcast {} to {} - wrong binding! type?",
                            stringify!($name), stringify!($ty))
                    }},
                    None => panic!("{} has no binding for {}!\n\tBound types: {:?}\n",
                         stringify!($store), stringify!($name), self.store.keys())
                }
            }
            fn put(&mut self, value: $ty) {
                match self.store.entry(stringify!($name).into()) {
                    Entry::Occupied(entry) => {
                        let existing: &$ty = entry.get().downcast_ref::<$ty>().unwrap();
                        panic!("Conflicting binding for {}; cannot bind to {:?} already bound to {:?}",
                            stringify!($bnd), value, existing)
                    },
                    Entry::Vacant(entry) => {
                        entry.insert(Box::new(value) as Box<Any>);
                    }
                }
            }}}}

// Registers a provider of a binding, introducing a recursive dependency on another binding
// Note this can only provide references, not owned types (because the closure would be the owner,
// and it goes out of scope upon returning).
//   Usage: provider!(BinderType, ProviderTraitName, ProviderType, DependantTrait, Closure)
///     BinderType:         A binder type, created by binder!()
///     ProviderTraitName:  Trait to create that will provide the given binding
///     ProviderType:       Type that ProviderTrait will provide
///     DependantTrait:     Binding trait that the provider depends on
///     Closure:            A closure of the form |d: &'a DependantTrait| ... that returns a &ProviderType
macro_rules! provider {
    ($store:ident, $name:ident, $ty:ty, $dep:ty, $provider_fn:expr) => {
        trait $name { fn get(&self) -> &$ty; }

        impl $name for $store {
            fn get<'a>(&'a self) -> &$ty {
                &$provider_fn(self as &$dep)
            }}}}
```

Listing 16: Using macros to condense the API

```
binder!(DepStore);
binding!(DepStore, UrlParts, util::UrlParts);
provider!(DepStore, PathParts, Vec<String>,
    UrlParts, |d: &'a UrlParts| d.get().path_components());
provider!(DepStore, UrlParams, HashMap<String, String>,
    UrlParts, |d: &'a UrlParts| d.get().query());

fn handle(&self, request: &Request) -> ResponseBox {
    // struct containing both the path and the query
    let url_parts = ...;

    let mut deps = DepStore::new();
    bind!(deps, UrlParts, url_parts);

    user_func(&deps, ...)
}
```

This example introduces several interconnected macros:

- **`binder!`** Creates the struct containing the actual hashmap data store.
- **`binding!`** Creates a `trait` and `impl` enabling the listed type to be retrieved from the binder type via the given trait.
- **`provider!`** Creates a `trait` and `impl` that recursively depend on a different trait in the store, rather than on its own data.
- **`bind!`** Adds (or binds) a value to the store's map. If a trait is used without a value having been bound the user will see a panic at runtime.

An additional macro, `inject!`, is provided to transform a function that takes *n* binding traits into one that takes exactly one such argument, which enables the framework to support callbacks of any number of dependency-injected arguments. This macro proved to be one of the key limiting factors of this approach, as (it seems[9]) the only way to implement such a macro is to enumerate the cases — one parameter maps to one argument, two parameters map to two arguments, and so on. The Hypospray project appears to have run into the same issue, as mentioned in 2.2.

Functionally this is identical to the initial hard-coded trait approach, but with almost no boilerplate beyond actually specifying the desired type names. Users get type safety, compartmentalization, and a fair bit of flexibility in a concise (albeit bespoke) syntax.

Due to the somewhat intricate relationships between the different macros there are certain patterns that result in surprising errors or bugs (such as unexpectedly missing bindings due to implicit dereferencing), but improvements in the macros' implementations has helped reduce their likelyhood and confusion. Debugability still remains the least-well supported aspect of this approach, but many common errors, such as unbound types, are able to trigger clear and detailed panics.

4 RESULTS

4.1 Summary

The design patterns we explored are quite common in other languages yet are difficult given Rust's design philosophy as a system's language with a preference for doing as much work as possible at compile-time. Finding ways to bridge that gap, while still taking advantage of Rust's safety and performance, is a fine needle to thread. While no single one of our approaches scored highly in all the evaluation criteria, the macro-based traits pattern shows the most promise for providing users with a type-safe and expressive yet low-boilerplate interface. Pulling in more functionality from the other approaches could further improve its utility, as would enhancing it's error-detection behavior. Discounting the motivating approaches, the factory-based method in particular suffered from low safety, limited flexibility, and verbosity.

4.2 Future Work

- Expand the behavior provided by the trait approach to support more common Rust patterns, such as enabling mutation or ownership passing of values held by the backing map. This requires finding a workable balance of flexibility and safety, and different balances may be appropriate in different cases. For example supporting ownership passing would effectively mean removing the data from the backing map, which would cause future attempts to invoke the trait's `get` method to fail at runtime.
- Investigate further metaprogramming and code generation approaches. Most existing dependency injection style applications in Rust today rely heavily on code generation, as it's the most flexible way to emulate reflection in other languages. Frameworks like Rocket provide even more elegant APIs than those explored here, essentially all thanks to compile-time code generation. This was an area we'd hoped to explore more, but it proved to be a larger space than we had time to dive into fully.
- Dynamically-typed return types, not just function inputs. For simplicity, every approach we explored assumed the data returned to the client would be a `String`, but webservers often return binary data such as images, and users may prefer to work with other higher-level types such as JSON and want the framework to be responsible for serializing the data they produce.

5 CONTRIBUTIONS

Michael implemented the Rivet server skeleton and meta-framework and the Simplified, Pattern, Closure, and Traits aproaches. Matthew implemented the Factory and Scope patterns.

REFERENCES

[1] [n. d.]. di-rs. ([n. d.]). https://github.com/Nercury/di-rs.
[2] [n. d.]. hypospray. ([n. d.]). https://github.com/jonysy/hypospray.
[3] [n. d.]. Ownership-driven dependency injection for JavaScript. ([n. d.]). https://www.npmjs.com/package/inceptor.
[4] [n. d.]. rust-ioc. ([n. d.]). https://github.com/KodrAus/rust-ioc.
[5] Sergio Benitez and the Rocket authors. [n. d.]. Rocket - Simple, Fast, Type-Safe Web Framework for Rust. ([n. d.]). https://rocket.rs/.
[6] Wikipedia contributors. 2017. Dependency injection - Wikipedia. (2017). https://en.wikipedia.org/wiki/Dependency_injection, accessed 2017-12-13.
[7] Wikipedia contributors. 2017. Strategy pattern - Wikipedia. (2017). https://en.wikipedia.org/wiki/Strategy_pattern, accessed 2017-12-13.
[8] Michael Diamond and Matthew Vilim. [n. d.]. Rivet Framework Source Code. ([n. d.]). https://bitbucket.org/dimo414/rivet.
[9] Markus Kohlhase. 2017. Rust web framework comparison. (2017). https://github.com/flosse/rust-web-framework-comparison, accessed 2017-11-1.
[10] Pierre Krieger and tiny-http contributors. [n. d.]. tiny-http - Low level HTTP server library in Rust. ([n. d.]). https://github.com/tiny-http/tiny-http.

[9] https://stackoverflow.com/q/47767910/113632