Crimson: Bridging Rapid Application Development and Traditional Development Environments

Adam Govier

A REPORT SUBMITTED AS PART OF THE REQUIREMENTS FOR THE DEGREE OF MSCI COMPUTING SCIENCE AT THE SCHOOL OF COMPUTING, ENGINEERING AND TECHNOLOGY ROBERT GORDON UNIVERSITY ABERDEEN, SCOTLAND

April 2025

Supervisor Dr. John Isaacs

Abstract

Modern software development projects often overrun budgets and deadlines. While rapid application development tools such as low-code development platforms offer promising solutions to these challenges, they often result in a reduction of flexibility and the project often becomes restricted to only that vendor or tool, this is referred to as vendor lock-in. This dissertation introduces Crimson, a development tool that is used as an extension to a traditional software development environment. Crimson's goal is to retain the efficiency of other rapid application tools but refrain from flexibility limitations and vendor lock-in. Crimson dramatically reduces generic boiler plate for common functionality found in CRUD style web applications. This is done by providing the developer with a set of standard UI components that generate backend API routes based on a database table identifier specified as parameter to the referenced component. An early prototype was presented to industry experts, the feedback suggested that it poses strong potential as a tool to rapidly create prototypes or as a training aid but due to the prototypes limited functionality it wouldn't be suitable for full-scale project development yet. Crimson demonstrates a viable path towards combining the efficiency of rapid application development and robustness of traditional development processes.

Acknowledgements

I would like to offer my appreciation to my project supervisor for his fantastic guidance, and Dev4 Online for their in-depth industry feedback.

Declaration

I confirm that the work contained in this MSCI project report has been composed solely by myself and has not been accepted in any previous application for a degree. All sources of information have been specifically acknowledged and all verbatim extracts are distinguished by quotation marks.

Signed: Adam Govier Date: May 2024

Table of Contents

List of Abbreviations

LCDP(s)	Low-Code Development Platform(s)
RAD	Rapid Application Development

Chapter 1 – Introduction

Modern software development faces ongoing challenges, this includes the overrun of deadlines and budgets, and a shortage of skilled developers. Rapid Application Development (RAD) tools such as Low-Code Development Platforms (LCDP) have emerged as a potential solution. They often allow for faster delivery and reduced technical barriers. However, these tools frequently come with trade-offs such as limited flexibility and vendor lock-in.

This dissertation introduces Crimson, a framework that aims to bridge the gap between RAD and traditional development methodologies. Rather than replacing traditional development workflows, Crimson enhances them by driving automatic backend generation from referenced frontend component usage. Crimson, when fully implemented aims to reduce the trade-offs previously mentioned in existing RAD tools while retaining the efficiency improvement granted from said tools.

This report outlines the current state of RAD tools, the motivation behind Crimson, details the system's design and implementation, and evaluates its potential benefits and limitations.

Chapter 2 - Project Scope

2.1 Literature Review

2.1.1 Background

Developing software is frequently found to be time-consuming and costly, with a significant proportion of software projects running over budget and being delivered late. Estimates on the number of projects encountering budget overruns vary, with figures ranging from 30 to 70 per cent depending on the source (Acquaint Softtech, 2024; Bloch. Blumberg. & Laartz, 2012; Palumbo. Rehberg. & Li, 2024; The Standish Group International Inc, 2015). Similarly, missed deadlines are common. A study from 2003 on software effort estimation found that 60 to 80 per cent of software projects exceed their planned schedule or effort (Molokken, K. and Jorgensen, M., 2003). This aligns, with The Standish Group International Inc (2015), report findings that 60 per cent of projects are delivered late.

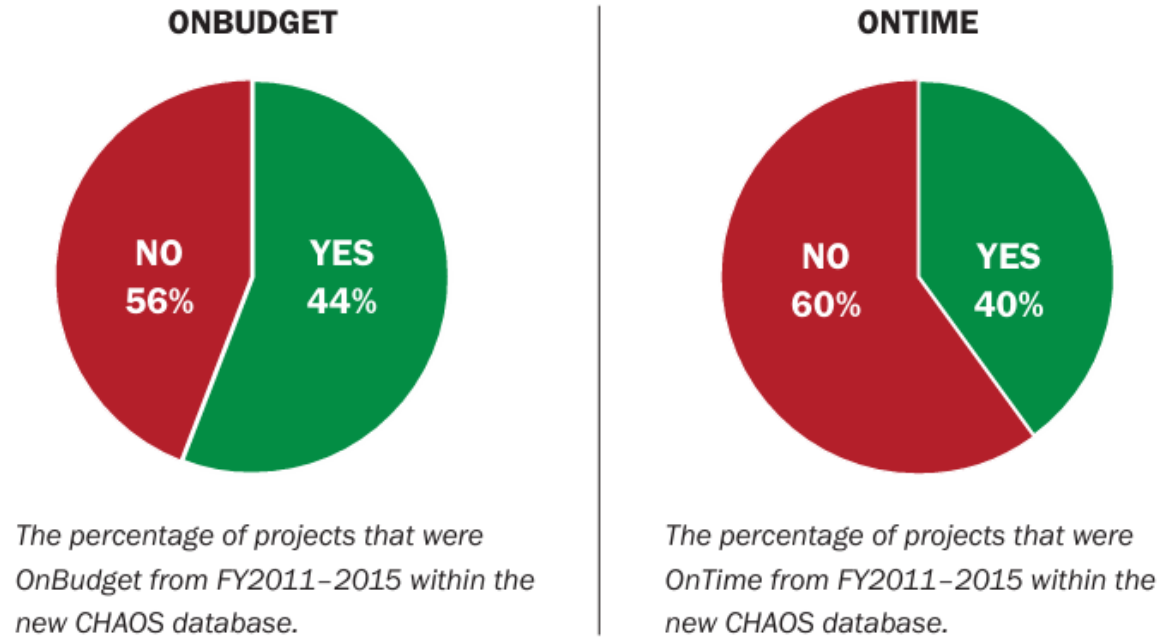

Figure 1, (Standish Group International Inc 2015), CHAOS Report 2015.

Despite recent economic challenges, such as COVID-19 which caused a slight decline in technology sector investment (Sherif, 2024; Whicher, 2024). The demand for software development remains strong, with strong growth predicted worldwide (Statista, 2024). Sources suggest that many organisations are experiencing skill shortages (Sherif, 2024; Whicher, 2024). The World Economic Forum's Executive Opinion Survey 2022, which recorded 12,036 responses from 126 economies, discovered the lack of skill and talent is the primary obstacle to market creation in technology. Although present across

economies, this talent shortage is most prominent in higher-income economies (World Economic Forum, 2024). With growing competition for skilled labour, and software development services in high demand, software development companies must look for new methods of delivering software faster and at less cost (Nan, & Harter, 2009).

In the next section on related research, this paper will explore what solutions have been proposed to solve the gap in talent, late project delivery, and the expensive process of modern software development.

2.1.2 Related Work

2.1.2.1 Rapid Application Development: Low-Code Development Tools

Low-code development is an example of an emerging trend (Rokis, & Kirikova, 2022; Luo, et al. 2021; Khorram, Mottu, & Sunyé, 2020; Bock & Frank, 2021; Sahay, et al., 2020). Low-code development platforms address the main cause of delayed deployment, which is the shortage of professional software developers (Rokis, & Kirikova, 2022; World Economic Forum, 2023). Although low-code development technology arose to the mainstream in around 2010 (Team Kissflow, 2024), foundations can be traced back much further to the mid-80s and early 90s with well-known software & tools, such as Microsoft Excel and Visual Basic (Formstack, n.d.; Darbyshire, 2023).

2.1.2.1.1 Low-Code Development Platforms by Definition

Literature provides varied definitions of low-code development, marked by some ambiguity yet unified by shared characteristics across descriptions. There is no clear definition of low code development. Terms like “low-code” and “no-code” are frequently used interchangeably, often resulting in confusion surrounding the concept (Luo, et al., 2021; Rokis, & Kirikova, 2022). Further criticism of the term “low-code” is provided by Bock & Frank, (2021), who highlight that the term ‘low-code’ is problematic saying that the range of platforms marketed under the term “low-code” is vastly heterogeneous.

However, these platforms often share similar characteristics. They frequently offer a visual approach to software development (Luo, et al., 2021; IBM, n.d.; Sahay, et al., 2020; Rokis, & Kirikova, 2022; Bock & Frank, 2021). Sahay, et al. (2020), enforce this idea by showing that every low-code development platform in their study included a graphical user interface designer. Bock & Frank, (2021), also describe an important part of low-code development platforms is their ability to contain many traditional software development tools in a single unified environment.

Another key concept, clearly derived from the term, is that these platforms allow users to develop software applications with little or no code (Luo, et al., 2021; Sahay, et al., 2020; Rokis, & Kirikova, 2022). Due to this, low-code development platforms are targeted at citizen developers, a citizen developer is a non-technical user with little or no programming experience (Sahay, et al., 2020). Khorram, Mottu, & Sunyé, (2020),

state that the primary user of these platforms are citizen developers. By 2026, It is estimated that 80 per cent of the user base for low-code development tools will be citizen developers (Onoufriou, 2024). Rokis, & Kirikova, (2022), agrees, mentioning that low-code development platforms enables citizen developers to be involved in the software development process.

2.1.2.1.2 Logic using Visual Abstraction

Low-code development platforms are not that new or innovative, claims Bock & Frank, (2021),

"To the contrary, what is actually provided in these environments are well-known tools and components for software development which have been used, in varying forms, for decades."

(Bock & Frank, 2021, p. 738). Bock & Frank, (2021), also claims that the design of low-code platforms has gained inspiration from several lines of research and that a prominent example of this is the concept of model-driven engineering.

Model-driven engineering allows domain experts (the user developing projects on the low code development platform) to use a higher-level syntax to define logic within the bounds of the low-code development platform. This higher-level syntax in Computer Science is referred to as a domain-specific language (DSL). A prime example of a DSL in low-code development is the Business Process Model and Notation (BPMN), used by platforms such as Mendix. BPMN allows domain experts to visually model an application's business logic, without the requirement of writing lower-level source code. Instead, these platforms utilise code generation engines to produce executable code from the DSLs Khorram, Mottu, & Sunyé, (2020). Please see Figure 2 as a demonstration of the DSL BPMN in a low-code development platform.

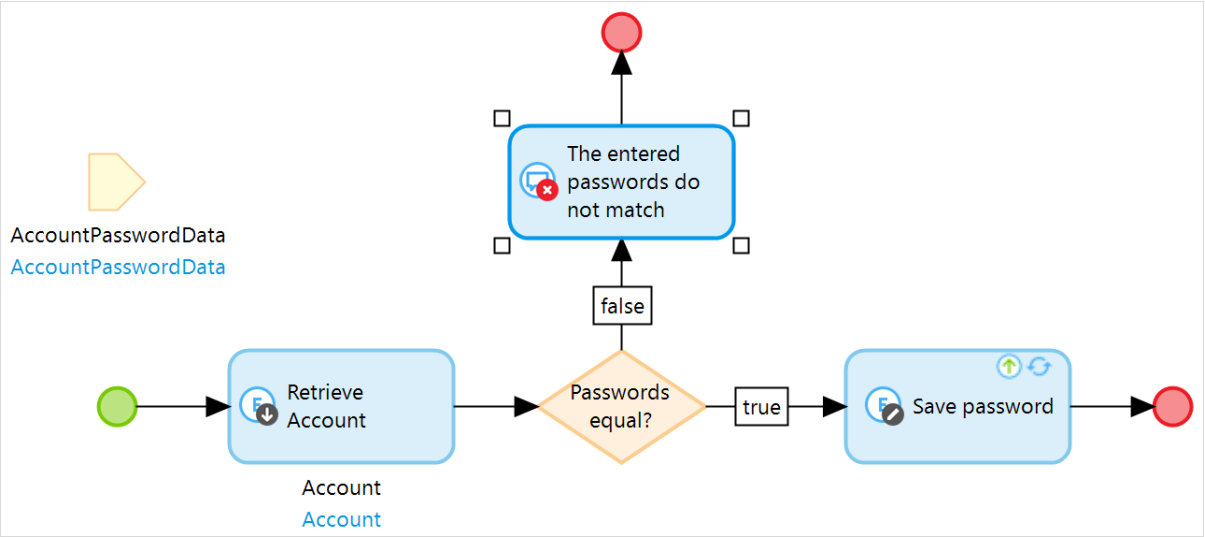

Figure 2, (Mendix, 2024), Mircoflows.

2.1.2.1.3 The Perception of Low-Code Development in Literature

Low-code development platforms are perceived in literature with a mixture of both positive and negative reception, with a collection of issues displayed across a multitude of sources. This discussion, highlights some of the key themes found whilst researching low-code development platforms.

Efficiency and Learning Curve

Research indicates that low-code development platforms are engineered in order to boost development speed and also lower the skill barrier, Luo, et al. (2021), argue that the availability of out-of-the-box components makes low-code development platforms easy to learn and use. This view is supported by Sahay, et al. (2020), and Khorram, Mottu, & Sunyé, (2020), who emphasise that low-code development platforms can bridge the gap between business and IT by enabling citizen developers to engage at a more concrete level in the software development lifecycle. Luo, et al. (2021), found that most developers agree that low-code development platforms result in faster development, decreasing the time to release. This is evident in the case of Martins, et al. (2020), who stated that if they used traditional development methods, their project would not have been ready to deliver on time. Bock & Frank, (2021), highlight a significant factor in reducing development efforts in low-code development platforms, is their ability to consolidate a range of development tools, including database management systems, deployment assistants, and various other tools, into a single, simplified development environment.

However, these platforms are not without their challenges, certain papers raise concerns that the end-users, or 'citizen developers' limited technical knowledge may inadvertently lead to the emergence of new difficulties (Khorram, Mottu, & Sunyé, 2020). A plethora of low-code development platforms do not include a sufficient amount of customisation features, this can require the end-user to develop custom code, thus, increasing the complexity and learning curve (Rokis, & Kirikova, 2022). Rokis, & Kirikova, (2022), also argue that multiple of low-code development platforms have a lack of documentation, lack of learning resources, nonintuitive interfaces, limited drag-and-drop capabilities, and they often require some level of knowledge in software development. Luo, et al. (2021), also state that most practitioners reflected that using these platforms are easier and faster than the traditional software development process, but it still has a steep learning curve to some degree.

Non-citizen developers can also face difficulties according to (Luo, et al., 2021). Luo, et al. (2021), state that low-code development platforms don't authorise access to source code. While this is generally true, it was found that this doesn't accurately describe all low-code development platforms. Findings from another paper Sahay, et al. (2020), discover that OutSystem's low-code platform does authorise the exportation of source code, this is confirmed by OutSystem's documentation (OutSystems, 2024). However,

Sahay, et al. (2020), revealed that, apart from OutSystems, none of the other studied low-code development platforms offer this feature. As a result, when an experienced developer needs to be involved in the development process, not having access to the source code may reduce the ease of use for the more technically inclined (Luo, et al., 2021).

Although not concretely agreed, the general-consensus between these source's findings is that low-code development platforms do speed up the development process, and generally require less technical skill than the traditional software development process. However, when applications are more complex or grow in complexity the platforms learning curve also grows, this may result in the requirement of a more experienced developer.

Vendor Lock-In

A recurring finding across several papers expresses the concern of vendor lock-in (Luo, et al. 2021; Bock & Frank, 2021; Khorram, Mottu, & Sunyé, 2020; Rokis, & Kirikova, 2022), vendor lock-in is when a project in one low-code platform cannot be used in another (Bock & Frank, 2021). Vendor lock-in poses a serious threat to the protection of investment, projects may have to be abandoned if the host platform stops support (Bock & Frank, 2021). Google App Maker, an example provided by (Bock & Frank, 2021), officially shut down in early 2021, taking down all existing apps with it (Google Workspace Updates, 2020). Rokis, & Kirikova, (2022), found this to be one of the leading reasons why companies are not adopting low code platforms. Sahay, et al. (2020), stated that there is a lack of technical standards across low-code development platforms, the researchers claim that this holds back collaboration amongst different engineers and developers. More research is required to evaluate the advantages of implementing a specification that would promote collaboration and interoperability across platforms.

Customisation and Reusability

Literature highlights significant concerns about the limitations of customisation within low code development platforms (Luo, et al., 2021; Rokis, & Kirikova, 2022; Sahay, et al., 2020). Due to the restrictive customisation options, many practitioners believe that low-code development platforms lack flexibility Luo, et al. (2021), and Rokis, & Kirikova, (2022), agreed with these claims, mentioning that some low-code development platforms suffer from limited flexibility particularly when solving more complex requirements. Both Rokis, & Kirikova, (2022), and Sahay, et al. (2020), agree that low-code development platforms often offer the ability to extend the core components and business logic of the application by writing custom code. However, as mentioned in the previous section "Ease of Use and Learning Curve", the authoring of custom code can add additional complexity.

Contrastingly, a few practitioners claim low-code development platforms to be very flexible even claiming that they are capable of doing almost everything (Luo, et al., 2021), yet it is not clear if this is unique to a specific platform or in general across low-code development platforms.

Reusability is also a drawback for many low-code platforms. Sahay, et al. (2020), claim that outside of the given library of non-domain-specific artefacts, these platforms rarely offer the ability to reuse artefacts created specifically for the project's domain. However, there is limited research on reusability in low-code platforms to reinforce this claim.

Contrary to the goal of low-code platforms, it is clear that code and some technical knowledge is often required. Bock & Frank, (2021), interestingly state that more research is needed to investigate how low-code development platforms can supplement rather than replace the traditional software development workflow.

Debugging

Practitioners from Luo, et al. (2021), study, and Rokis, & Kirikova, (2022), research agree that debugging is met with difficulty, Rokis, & Kirikova, (2022), elaborates this further by explaining that this is due to the graphical representation of software development kits.

Interestingly, Sahay, et al. (2020), present a different conclusion, stating the opposite that low-code development platforms simplify bug fixing, implying that debugging isn't a difficulty. This claim however, may be biased, as it is based on a single source – an article published by Mendix, a well-known low-code development platform. It would be in Mendix's interest to promote their platform.

Deployment and Publishing

Low-code development platforms typically come bundled with deployment assistants (Rokis, & Kirikova, 2022). Several sources argue that this deployment or publishing of an application is straight-forward when using low-code development platforms (Luo, et al., 2021; Sahay, et al., 2020; Rokis, & Kirikova, 2022).

"In most platforms, it is possible to quickly preview the developed application and deploy it with few clicks." (Sahay, et al., 2020, p. 173).

However, Rokis, & Kirikova, (2022), also state that it isn't always without difficulties. Issues including deployment configuration, accessibility, and performance (Rokis, & Kirikova, 2022). OWASP, (2022), also raise concerns about incorrect configuration stating that it may lead to data leakage or unauthorised modification. Interestingly, the primary risk to projects made with low-code development platforms are the developer themselves, either by misconfiguration or negligence (Onoufriou, 2024). However, this can also apply to traditional development environments. It could be argued that

misconfigurations are more likely to apply if projects are developed by citizen developers due to lack of domain expertise.

2.1.2.1.4 Overview

While low-code development platforms offer substantial benefits, including a faster development lifecycle and a lower skill threshold, even though there is prominent growth of these platforms, many companies chose to continue with the traditional software development route due to challenges present across many platform providers. The specialised expertise needed to manage complex applications, the potential for vendor-lock in, or even the possibility of a platform provider shutting down, leading to a loss of project access, often make low-code platforms a less favourable option.

2.1.2.2 Other forms of Rapid Application Development

2.1.2.2.1 UI Component Libraries

UI component libraries are extensive sets of pre-made UI components (Semma, 2023), for various development frameworks, for example, Telerik/Kendo UI which supports Blazor, Angular, React, Vue and many other frameworks (Telerik, 2023). See figure 3 for a demonstration of provided components by UI component library Sampo UI.

The general consensus is that UI libraries can speed up development (Motta, 2024; Andrzejewski, 2023; Studio by UXPin, 2023), and improve design consistency (Studio by UXPin, 2023; Andrzejewski, 2023).

However, sources (Studio by UXPin, 2023), and (Motta, 2024), are likely biased, as Studio by UXPin's blog article is promoting their own development tool labelled "Merge Component Manager", and Motta's blog post is posted on the official Telerik blog, a platform offering component library products. Interestingly, Andrzejewski, (2023), also mentioned that in their own experience with the UI component library Vuetify, they found it to be

"...more troublesome than it is worth" Andrzejewski, (2023),

Andrzejewski, (2023), state that UI libraries are provided to be generic as possible and often require a large amount of workarounds. In section 2.2.1.1, this paper discusses an innovative UI library which is more specialised in its domain.

Two papers uncovered some interesting drawbacks to UI libraries Semma, (2023), concluding that the studied UI libraries include notable performance differences between them but Semma, (2023), advises developers to also consider other factors such as usability when choosing a UI library. (Karlsson, 2021), studied 6 UI libraries and found that all 6 of them had one or more accessibility issues, totalling 50 issues overall. Karlsson, (2021), describes this as a major issue due to the growing popularity of UI

component libraries further stating that issues in UI libraries have the ability to affect hundreds of thousands of web products.

2.1.2.2.2 Sampo UI

Sampo UI is a specialized UI library focussing on providing a framework for building web applications that enable interactive data extrapolation from knowledge graphs within a user interface (Ikkala et al., 2022). This type of application is also known as a semantic portal. Sampo UI was created to reduce the difficulty of creating such portals and substantially reduce coding efforts, this is achieved by providing the developer with 120 domain specialised components (Ikkala et al., 2022). See Figure 3 for demonstration.

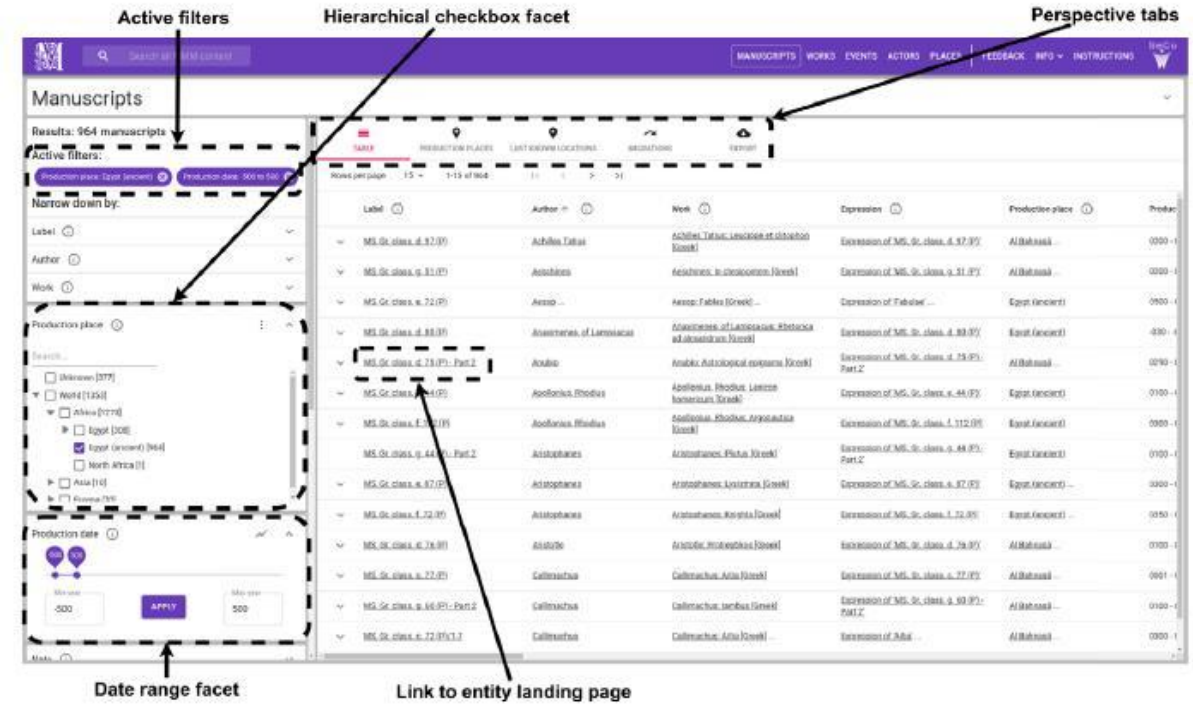

Figure 3, (Ikkala et al. 2022) SAMPO-UI components

What sets Sampo-UI apart from other discussed UI frameworks is its ability to communicate with a remote dataset without heavy involvement from the developer. Sampo UI's backend generates SPARQL queries based on incoming API requests and predefined queries, after executing the query it converts the SPARQL results into JSON for the UI to be able to understand (Ikkala et al., 2022). All Sampo UI faucet components are connected with a centralised state management system, this allows for the UI to listen for input on any of the components and also update the state of other components after results are received from the API (Ikkala et al., 2022). A simplified explanation of this is that the provided UI components are tightly coupled with the adaptive backend resulting in a dramatic reduction in required code written by the developer. Sampo UI has been received positively, at the time of publication 6 portals had already been published from various authors, and 5 more were underway. Notable

authors include “Ministry of Justice, Finland”, and the “National Archives of Finland” (Ikkala et al., 2022).

More research is needed to expand the technological concepts of Sampo UI to a broader class of applications. As mentioned earlier, Bock & Frank, (2021), pointed out that an investigation is needed into how low-code development platforms can supplement, rather than replace traditional software development environments. Sampo UI provides many similarities to low-code development platforms such as an expansive set of UI components and code generation. It would be interesting to research if Sampo UI’s concepts can be applied beyond semantic web portals to a broader range of web applications.

2.1.2.2.3 Mavo

Many people can author static web pages with HTML and CSS, however, face difficulty in making persistent interactive web applications (Verou, Zhang, & Karger, 2016). This can be related to early findings in this literature review which indicated that skill gaps are the primary setback for software development deadlines. Verou, Zhang, & Karger, (2016), claim that existing frameworks that aim to simplify software development, all require programmers of all levels to write a considerable amount of code. Verou, Zhang, & Karger, (2016), however, fails to compare low-code development platforms which have been primarily found to target citizen developers.

Mavo is presented to be a superset of HTML which aims to include all the required features to build interactive persistent applications without the required development experience of traditional methods (Verou, Zhang, & Karger, 2016). The results were promising, stating that even users with no programming experience were able to create Mavo applications and that the majority of users could easily mark up the editable portions of their mock-ups to create applications with complex hierarchical schemas (Verou, Zhang, & Karger, 2016). The results provided by the study were derived from a test group of 20 novice developers (Verou, Zhang, & Karger, 2016). Mavo may benefit a larger number of test participants, particularly those across different backgrounds. Verou, Zhang, & Karger, (2016), indicated that their primary target is novice developers, but mentioned they aim to target a broad population of users. This broader population of users are not included in this test group.

2.1.3 Conclusion

Two key research gaps were identified in the reviewed literature, the first gap identified by Bock & Frank, emphasises the need for further investigation into how low-code development platforms can supplement traditional development approaches rather than replace them (Bock & Frank, 2021). A project that shared similarities to the gap identified by Bock & Frank, was Sampo UI. Sampo UI shares some concepts with low-

code development platforms such as an expansive set of UI components and code generation, but Sampo UI is limited to semantic portals.

The research objective is to incorporate key concepts from low-code development platforms and other rapid application development tools such as Sampo UI, concepts including model driven development, libraries of abstract components, and generated business logic, into a traditional development environment. This integration seeks to enhance development efficiency while avoiding the limitations often associated with low-code platforms, and catering to a broader class of application than Sampo UI, thereby better meeting the needs of the industry.

The output artefact of this research, referred to as Crimson, will serve as the basis for addressing the research questions and requirements outlined in sections 4 and 5.

2.2 Research Questions

RQ1: Does integrating low-code development concepts inside a professional development environment improve the efficiency of trained developers?

RQ2: Can Crimson reduce timescales and software budgets?

RQ3: How does Crimson compare with existing rapid application development tools?

2.2.3 Requirements

2.2.3.1 Functional

1. Crimson must automatically generate backend API routes (unless overridden by the developer) based on used included UI components.

2. UI components must allow developers to specify database models in order to map out component logic to corresponding data sources and generate API routes.

3. Crimson must support full customisation of a component's default business logic.

4. Crimson must support a comprehensive role-based authentication system but allow for custom authentication rules for more specific scenarios.

2.2.3.2 Non-Functional

1. Crimson must avoid vendor-lock in, developers should be able to use the exported source code independently to Crimson, meaning exported source code must be clear, maintainable, and in an industry-leading language and/or framework.

2. Crimson should improve developer efficiency, specifically one developer should be able to produce more output than the same developer with traditional development tools.

3. Crimson should be easy to understand and have a minimal learning curve.

4. Crimson should securely handle sensitive data, such as personally identifiable information and credentials.

Chapter 3 – Design / Methodology

3.1 System Overview

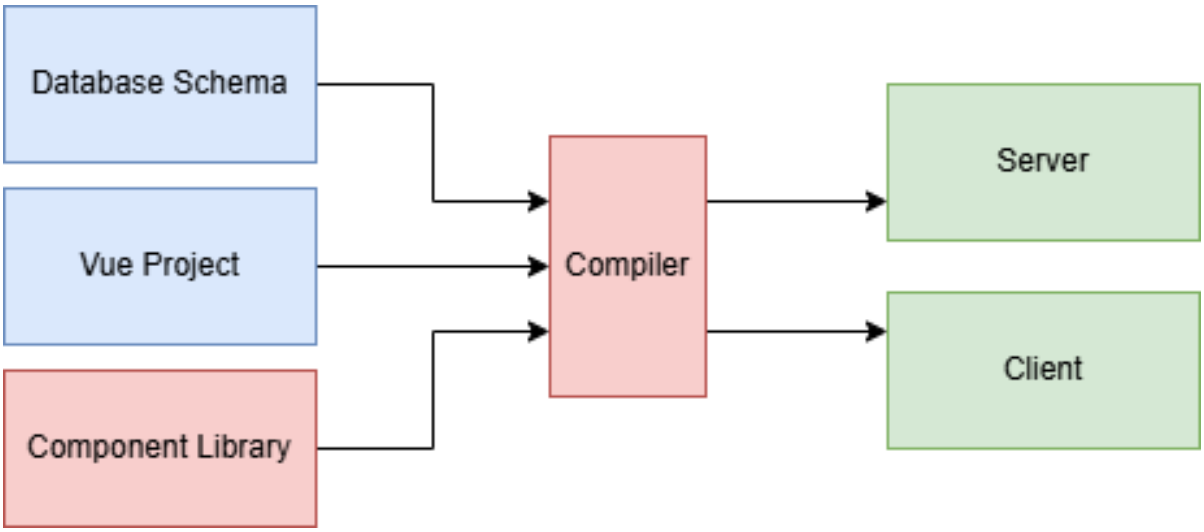

Figure 4, Complete System Structure

Crimson will act as an extension to Vue JS 3. Crimson will facilitate the automatic generation of backend code using a Crimson provided library of components. To support the generation of backend code, each Crimson Component will have its own predefined backend code not specific to any entity model. Each Crimson component will take in a property which specifies which entity model to use.

Entity models refer to a table like schema such as in a SQL style database. Crimson will likely use a MongoDB database; however, the concept still applies. MongoDB is a great fit due to its flexibility, meaning that you can add new properties whilst rarely running facing migration difficulties. To prioritise developer efficiency, Crimson will follow a database code-first style. Meaning that the developer will create entity models within TypeScript, and these changes will automatically be applied to the database without manual intervention, again the choice of MongoDB will do most of the groundwork for the code-first methodology.

Flexibility is of high importance; this is a prominent challenge of LCDPs. Flexibility will be possible by developer implemented lambda functions as component properties or allowing the developer to completely override controller classes.

Security is another important feature; components will need to have a property to specify which access levels can access which API routes. Without transformations, sensitive backend logic could be written into the client project. The client must be cleaned to contain only frontend code.

Figure 4 represents the planned system architecture. Blue items represent developer made directories, i.e., the directory of the entity models and the directory containing the

Crimson enabled Vue project. Red items represent the library of Crimson components and the compiler which will convert the left-hand side directories into two output projects. The output projects represented by the colour green, is a client server architecture, with the server acting as a web API.

Another key design choice is that any generated code must be understandable to a developer familiar with such tools (the chosen tech stack). This is to avoid vendor lock-in, another prominent issue of LCDPs.

Once Crimson is beyond the prototyping phase, comprehensive documentation should be created. Documentation should outline all available Crimson components, and what functionality they provide. Additional detail including required and optional properties, available Vue slots (used to provide custom template), and examples. Like many other technical documentations, a getting started guide would be a useful addition.

3.2 Component Structure

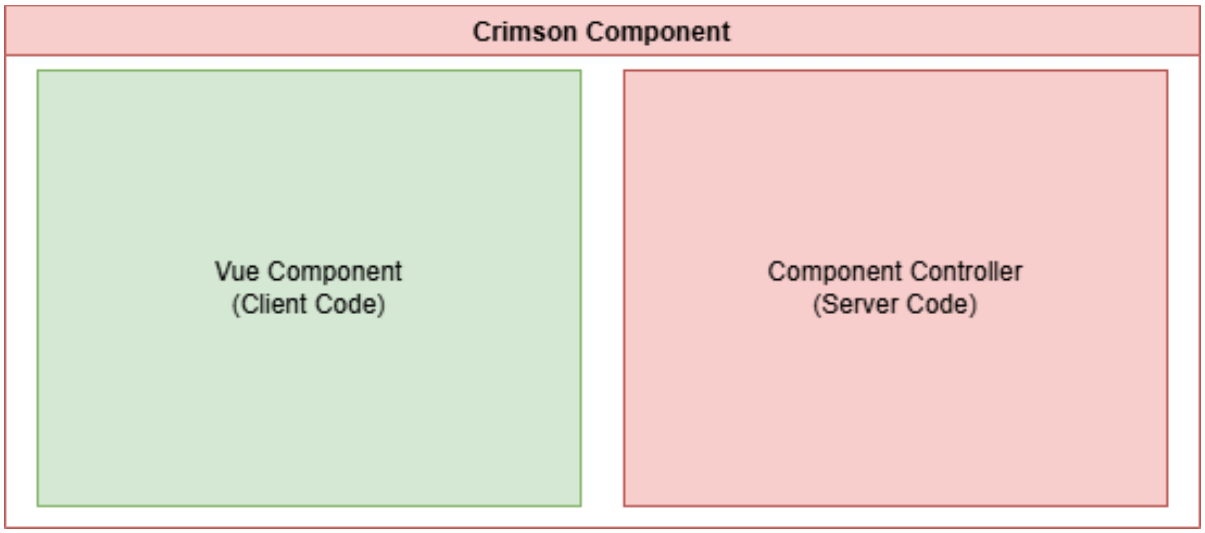

Figure 5 Crimson Component Structure

From a developer perspective, Crimson components should be able to be used almost exactly like they would any other Vue component. They would need to first import the component into the parent file and then reference it within the parents' template.

Each component will have unique properties the developer can use, for example, a table component might include a property specifying which columns the table should show. Components will also include a required property labelled "entity-model"; this will allow the developer to decide which entity model the component should run its backend operations on. Three other optional properties that will be available on all Crimson components are detailed in Appendix 3.

The next discussion will focus on how Crimson components will be developed rather than how developers will use them. Crimson components will be formed of two main parts (see Figure 5), a Vue component and a backend controller which specifies logic independent of specific entity models.

3.2.1 Component Controller

To generate backend logic specific to entity models, highly generalised routes must be created, the controller will follow similar concepts to a black-box environment, controllers must be made in a way that the logic does not know what type of entity model will be used with it.

A controller will likely consist of a bespoke class which extends a base controller class. The base class will give the controller access to various utility methods (see Appendix 4).

Now, methods are created and implemented within the controller. Each method will specify an API route. A method name should start with a lowercase HTTP method name (see HTTP/1.1: Method Definitions) such as “GET” or “POST” then a route name appended to it. ‘getTableItems’ would be a valid method name for example. Finally, the method body should contain the required server logic.

Figure 6 Example API client usage

A planned advantageous design choice would be to utilise the method signatures as a client API service too (see Figure 6 for example usage, and Appendix 5, for a more detailed explanation).

3.2.2 Crimson Component

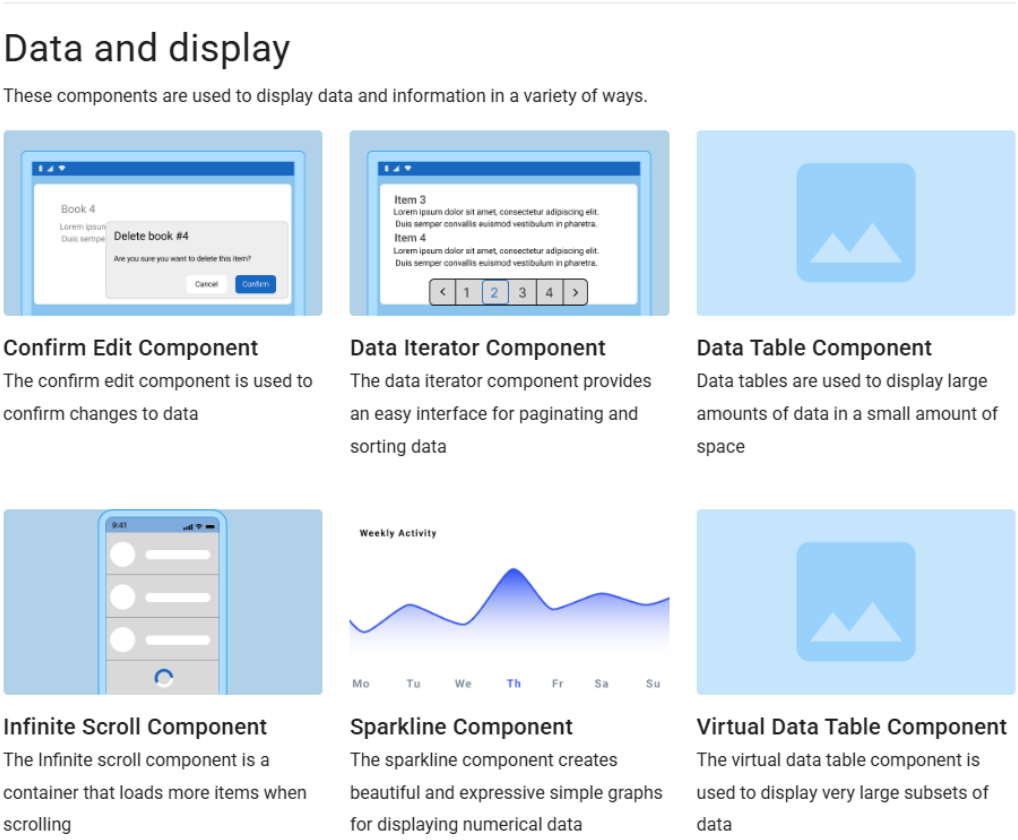

Figure 7 Example Vuetify components (Vuetify, 2025).

A component will be tied to its corresponding Crimson controller. The components template tag will consist of UI components provided by a third-party library like Vuetify (see Figure 7 for UI examples). This will reduce the implementation time of Crimson, due to not having to implement its own frontend UI library.

In the script section of the Vue component, required and optional properties will be declared (refer to 3.2). The component will also register the API client by specifying it's coupled Crimson controller. Then bespoke component logic is implemented, for example, firing API client methods on a UI event or loading data when the component is mounted.

3.3 Customisation and Extensibility

One of the major concerns of other RAD tools, primarily LCDPs is their lack of flexibility and customisation. Since Crimson is an addition to a traditional development environment, this gives room for extension by the means of custom code.

On the frontend with clever usage of preexisting UI component libraries, customisation properties could be passed to the Crimson component then passed to the underlying UI component. Also the powerful utility of Vue slots (insertion of custom template within components) would allow for a great deal of control from the frontend perspective.

Backend customisation will inherently be much more difficult. The proposal to tackle this, would be to provide developers with two logic customisation options.

3.3.1 Lambda Enabled Filtering

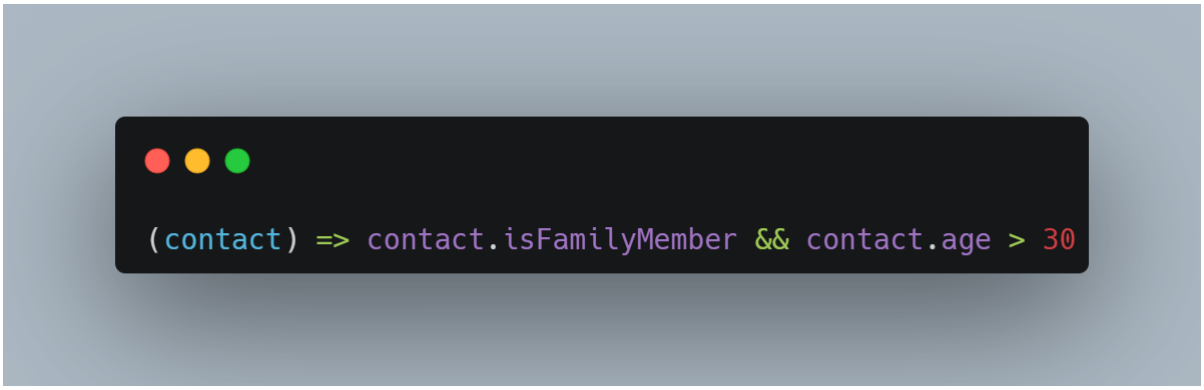

Figure 8 Example lambda filter function.

Developers could provide the component with a filter property; the value of this property would be a lambda function defined by the developer. The lambda would take an empty instance of the entity model as a parameter, allowing the developer to apply Boolean logic to rows (see Figure 8 for an example).

This function could then be applied to API routes by the compiler, possibly applying it to the end of the method as mixin or middleware. However, placement might be problematic, in logic such as server paging where the returned data is only a limited subset of the data, applying the lambda at the end would cause it to only filter data for that page. The position of the filtering might need to be specified by the controller.

3.3.2 Overriding Controllers

In cases where simple filtering is not enough, complete overrides should be possible. Developers should be able to create a new class which extends the component controller, then overriding methods to use custom logic. The developer will have access to custom methods from the top-level class (see Appendix 4).

3.4 Authentication and Authorisation

Authentication and authorisation are incredibly important for web applications. Lacking this functionality would make Crimson unsuitable for most applications. To address this a property could be provided to the Crimson component called authorise which a developer could use to specify which conditions is needed to allow a user to access

said resource. These conditions could be in the form of role keys such as “ADMIN”, or lambda functions for more specific tasks.

By using an authorisation framework like Identity for ASP.NET, policies could be generated by the compiler, and then applied as an authorise decorator on top of said methods.

More research is needed by Crimson, to investigate how to get around potential collisions/conflicts for when identical routes are used but have different authentication policies.

Standard routes would need to be generated for user management purposes such as creating a user, logging in etc

3.5 Compiler Architecture

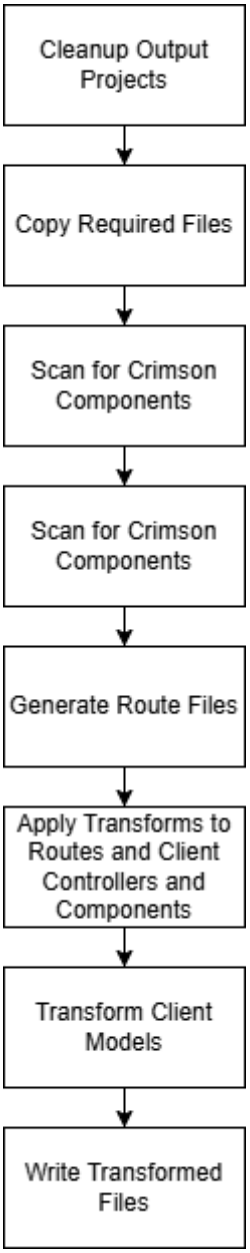

Figure 9 Simplified Compilation Pipeline

The Crimson compiler will follow a pipeline style methodology for simplicity of implementation, however, designing the compiler to run tasks in parallel would offer a performance gain. Refer to Figure 9 for a simplified pipeline diagram.

The proposed compilation process begins with a cleanup stage. To keep output projects tidy, rather than just overwriting files, they are completely deleted. This prevents the issue of unused files remaining in output projects if they no longer exist or no longer will be generated by the compiler. A beneficial addition to this design would be to preserve the 'node_modules' directory (see Appendix 6).

The compiler will then start to copy some necessary unprocessed files/directories. This primarily refers to copying the input project and Crimson component library directly to the client but also would include any required configuration files, and standard boilerplate. The unprocessed entity models and Crimson library will also be copied to the server project. A small transformation will be done on both output projects, to adjust any import paths to the new directory structure.

After cleanup and copying, the compiler will scan the input project for all files with a Vue extension. For each Vue file found, the compiler checks if any Crimson components have been referenced. If it does, a record will be kept for each Crimson, the record consisting of; the component, which controller the component uses (the default controller or one overridden by the developer), the chosen entity model, the lambda filter function (if present), and finally authorisation policies (if present).

These references will then be grouped by entity model. Each group's records will be filtered to only contain unique combinations of controller, lambda filter, and authorisation policies.

For each entity model, route controllers are generated based on the method signatures and bodies within all the controllers within the grouped records. The routes are then compiled into a single file labelled by the entity model identifier. Then transformations will be done on the route file to facilitate:

- Replacement of the get repository method with defined repository instances targeted at the specific entity model.
- Replacement of the get filter method with the provided lambda function.
- And placement of method decorators to facilitate provided authentication policies.
- The get context method will also be replaced with an object containing the values for request and response objects.

This file will then be written to the output server project under the routes folder.

Earlier in the proposed compilation steps, the Crimson component library containing the Crimson controllers was copied. These controllers will then be transformed based on the above generated routes to include API requests to said routes, also the response of the API will be the methods return value. A parameter containing the entity model identifier will be fed into the API request URL to call the correct resource. For the controller to be able to read this parameter it must first be sent by the Crimson Vue component, thus the Crimson Vue component must also be transformed.

Finally, the entity models will be stripped of any database specific code and then will be placed into the client output project under the directory models.

An additional planned feature of the compiler is hot-reload. Hot-reload allows developers to preview live changes to the code base, for more details on how hot-reload will be applied to Crimson refer to Appendix 7.

Chapter 4 – Implementation

4.1 Environment and Tooling Setup

The overall architecture remained mostly consistent with Figure 4. The only difference between was the project labelled “database schema” was moved into a directory called models within the project labelled “Vue Project”.

The output projects consist of these core technologies: Node.JS, Vue JS, Fastify, MongoDB, and TypeORM. For information on why these technologies were chosen please refer to Appendix 8.

The compiler runs on Node JS with TypeScript, this is to be compatible with the already implemented official Vue and TypeScript compilers, this guarantees the most recent updates available.

4.2 Crimson Library

4.2.1 Core Utilities

Two base classes have been implemented to assist in controller creation.

“CrimsonComponentController” – This is a placeholder class which contains three empty bodied methods such as “getFocusedRepository” (returns repository of selected entity model). Please refer to Appendix 9, for information on why empty body methods were used.

“**CrimsonEntity”** – This abstract class is extended by every entity model made by the developer. The entity contains a base database column called “_id” which represents the MongoDB Id assigned to each row. This allows controllers to do tasks such as deleting by ID, as the controller knows every generic has the property “_id”. “CrimsonEntity” could be expanded to include columns such as, when the entity model was created and when it was updated etc.

A range of interfaces have been made for specifying default component properties such as entity model, and to describe the controller request and response instances.

Another empty bodied function has been created to indicate to the compiler which controller a Crimson component uses.

4.2.2 Included Components

```
export default class Form<T> extends CrimsonComponent {
  async postCreate(payload: HttpPayload<T> | undefined = undefined): Promise<T> {
    const ctx = this.getContext();

    const data = this.getFocusedRepository();

    const entityInstance  = data.create(ctx.req?.body);

    const createdItem = await data.save(entityInstance);

    ctx.res.code(200);

    return createdItem;
  }
}
```

Figure 10 Form Controller

Three components have been implemented a Data Iterator component, a Data table component, and a Form component. These three components all contain bespoke logic relevant to the UI component such as the table containing a generic route for getting items that are server side paged and filtered.

For descriptions of currently implemented Crimson components refer to Appendix 10.

The concrete implementation of the Form controller can be viewed at Figure 10.

4.2.3 Generic difficulties

One of the biggest challenges I had when implementing the library was how TypeScript generis is handled in Vue.

One prominent example of struggles with generics I had, is that Crimson components have a "ref" object called data. Data is typed as an interface taking in a generic. With the generic being the type specified by the entity model property. An example interface for the reactive data is the 'CrimsonDataList' which contains a property called items which is an array of the generic. This all works as expected within the Crimson component, however when you give data back to the parent component (where the developer referenced the Crimson component) using something like slot properties (see Slots | Vue.js), the type is resolved as the base class 'CrimsonEntityModel' rather than the class extending 'CrimsonEntityModel' (the entity model originally provided as a property).

The first work around that I came up with is the developer explicitly marking the slot property as the type of the entity model. This was not ideal, as it would require the developer to first understand that the return data is the entity model and second it seems unnecessary for the developer to have to keep stating the types used.

After many hours of tinkering and research I found a post online (Stalker, 2021) which while not exactly similar, it explains that TypeScript is inferring the generic prop type as unknown, rather than the Product type they specified. One of the answers suggested that the generic should have a default value of an empty object, unfortunately they didn't leave an explanation on why. The implementation they provided isn't compatible with the way I setup my data object i.e., a reusable interface. I did some research on how I could apply the same concept inside my interface for the generic which should be inferred as a class extending the 'CrimsonEntity' class. I found this post (Eric, 2015) asking if they could tell TypeScript that a class can be initialised with the new keyword. In the answer they typed a constructer of a generic. I added this into my code to trick TypeScript into inferring that the generic is an instance of the generic, thus inferring that it must have a value i.e., at the very least an empty object. This resolved the type checking in the slot properties without the developer having to explicitly type each prop. I am unsure why component generic properties require the empty object; I think further research would need to be done to investigate the Vue and TypeScript compilers to understand why it was being resolved as the base class in the first place.

4.3 Component Scanning

To pick up all Vue files within the root of the input project or sub directories, Glob is used, which allows the use of patterns to find files. Within each found Vue file the template is scanned for component references using the property "entity-model". The script tag is scanned to find import paths for every component reference identified. Each Crimson component is then loaded to find the controller referenced with the call to the "useController" method. Finally, a list of records is kept containing the entity model, Crimson component, and controller.

The records are then grouped by entity model and filtered to only include unique records in each group – this prevents the creation of the same route multiple times. These groups will be referred to as the entity controller map.

4.4 Transformations

Transformations are now applied to the entity controller map to create and modify various source files. Transforms are made possible by the Factory functionality of the TypeScript compiler, and in many places, standard templating or replacements are used.

4.4.1 Example Transformations

Many transformations are run, from simply changing paths of imports to complete route generation. Below are two examples of transformations done by the Crimson compiler.

4.4.1.1 Route Generation

```
export function createFastifyRoute(blueprint: RouteBlueprint) {
  const route = `
    fastify.${blueprint.getHttpMethod()}("/${blueprint.getRouteName()}", {
      async handler (req: FastifyRequest<{Body: ${blueprint.getEntityModel().getIdentifier()}}>, res:
FastifyReply) {
      ${blueprint.getCodeBlock()}
      }
    });
  `;

  return route;
}
```

Figure 11 Fastify Route Creation

```
export function createFastifyRoutePlugin(name: string, routes: string[], entityModel:
TypescriptSourceFile) {
  const routePlugin = `
    import { FastifyInstance, FastifyRequest, FastifyReply } from 'fastify';
    import ${entityModel.getIdentifier()} from '../../server/models/${entityModel.getIdentifier()}';
    import AppDataSource from '../crimson/AppDataSource';

    export default async function ${name}Routes(fastify: FastifyInstance) {
      ${routes.join("\r\n")}
    }
  `;

  return replaceCrimsonKeywordsInRoutes(routePlugin, entityModel);
}
```

Figure 12 Fastify Plugin Wrapper

Initially, controllers need to be parsed into a format that can be converted to Fastify routes. This is handled by the route parsing method, it scans each method inside of a controller by looping through the TypeScript AST, identifying:

- **HTTP method** (the beginning of the method name)
- **Route name** (from the remainder of the method name)
- **Method Body** (contains the route logic)

The identified information is stored in an object called a route blueprint; the parsing method returns a list of these route blueprints.

Each entity model has a master route blueprint list. As controllers are processed for each entity model, their route blueprints are merged into the master list. This allows for multiple controllers to contribute routes to the same entity model.

For each blueprint in a master list, a Fastify route is generated (see Figure 10). For Fastify to recognise external route files, they must be in the format of a Fastify plugin. The list of Fastify routes for the given entity is wrapped in a templated plugin function (see Figure 11). For the route to understand the identifier of the entity model, the entity model is imported at the top of the route file.

Finally, before the route file is written to the routes folder within the server project, a replacement function swaps out placeholders such as generic repository getters with the concrete repository of the selected entity model.

4.4.1.2 Client Controller Transformation

Early in the compilation process the Crimson library is copied to the client project, the library includes the Crimson components. The copied controllers of the Crimson components must be transformed into API services, i.e., to perform API requests and parse and return responses.

Each method inside controllers is processed as follows:

1. The method body is cleared leaving an empty method signature.
2. A new parameter is prepended to specify which sub path to request the resource on, this parameter is set to the name of the entity model identifier inside the Crimson Vue component (note the component is also transformed to provide this value from the name of the generic).
3. A new method body is generated.

See Appendix 11, for a high-level explanation of how client API service method bodies are generated.

For a more detailed explanation of how TypeScript code can be generated using the TypeScript Factory see the below section 4.4.2.

4.4.2 TypeScript Compiler Challenges

For most transformations, the Factory functionality of the TypeScript compiler is used. The tools available in the TypeScript compiler API are fantastic, with premade methods for most challenges – anything from creating new import statements to the modification of method declarations. However, one difficulty became prominent deep into the implementation of the compiler. Outside of simple creation and modification, the logic can become very complex and hard to understand without thorough read through of code, this will hinder the maintainability of the compiler.

A good example of this is the client project transformations. In particular converting the API controllers into API communication services as referenced earlier. To do this a block representing the fetch request must be programmatically constructed with various helper methods. The block must contain sub blocks including the URI plus query parameters, options such as the body (if present), and headers. Finally, another block is needed to parse the response into a JSON object.

To support the reader in understanding the difficulty of using the TypeScript Factory API, see Appendix 12, for an in-depth walkthrough in the creation of the headers option of a Fetch request.

Partway through implementing the compiler when this problem became more apparent. I discovered a third-party module aiming to make this process simpler (see Appendix 13) called TS-Morph. I made the conscious design choice due to being so far into the implementation of the prototype Crimson compiler to continue with the standalone TypeScript compiler API for two reasons:

1.) To keep a standardised method of applying transforms without having to refactor the codebase.

2.) To get a deeper understanding of how the TypeScript compiler works.

The main thought process behind not refactoring was the limited time remaining to implement a working prototype.

Another difficulty I had with using the TypeScript compiler was the usage of experimental features, Crimson uses class and method decorators as per TypeORM guidance to mark TypeScript classes as database entities and properties as columns. The decorators are separate to the class declaration or method declaration rather than being a property of said class/method. When iterating through nodes however, it didn't seem to be classified as a node either. Experimental features are not referenced in the TypeScript transformer handbook (itsdouges, 2019), meaning that the decorator properties lacked usage guidelines, obviously plenty of documentation is available to anyone using TypeScript as a language but for metaprogramming it was severely limited. In the end, I ended up writing REGEX queries to remove database syntax from client models. I am confident that the functionality is within the TypeScript compiler API to remove decorators, but I was not able to do it without the required expertise or specific examples.

Chapter 5 – Project Evaluation

Crimson is still in its early stages of implementation, the proof of concept developed in the timeframe given for this project lacks feature completeness required for empirical evaluation. However, key observations have been made by Crimson's researchers, and informal feedback from third parties have been received.

5.1 Researchers Observations

The following observations aim to summarise findings and touch on each aspect of the functional and non-functional requirements.

- Crimson can successfully generate backend API routes based on referenced components.
- Database collections can simply be mapped within the project's source code without the requirement for the need to create schemas on a database level or iterative migrations when models are modified. These models naturally integrate into components due to their TypeScript representation.
- Flexibility is extremely limited, and backend modification isn't available to the developer. Unfortunately, due to the limited time frame this wasn't something that was implemented. However, this is an extremely important functional requirement, not to fall to the same short falls as RAD tools. Moving forward, the design section outlines a clear methodology to allow developers to extend the framework. This is something that would be implemented in future versions of Crimson.
- Vendor lock-in is another key concern of other RAD tools. Crimsons output is easily accessible and in the format of the well-known client-server architecture. Output code is human readable, and imitates code written by a human developer.
- Developer efficiency is a difficult metric to measure at this stage, the prototype lacks enough premade components, flexibility, and security requirements to be useful to real-world projects. However, as a proof of concept it demonstrates how rapidly a simple create (using the Form component) and display (using the Data Iterator or Data Table component) application can be developed. While not empirically evaluated, this observation shows potential moving forward.
- Ease of use is a clear strength of Crimson. Crimson targets software developers not domain experts. Crimson slots fluently into a Vue environment, developers with experience in Vue JS, Node JS, and as an extension a third-party UI library like Vuetify should find the experience almost identical but have the benefit of not needing to implement most of the API logic manually.
- Work is required to adhere to proper security principles such as the secure handling of sensitive data, and to include a robust authentication and

authorisation system, a high-level overview of how authentication and authorisation is covered within the design section. Moving forward, Crimson would do further research into security requirements and implement them.

5.2 Informal Feedback

To complement the previous section. Feedback was collected during a demonstration of Crimson delivered to Aberdeen based software development company Dev4 Online. The session included a brief discussion of what Crimson is and what it aims to achieve, its current features, limitations, and finally a demonstration of using the development tool to build a simple web application.

The consensus was that the current implementation is a good proof of concept. Gail Henderson CEO, introduced that Crimson could be a powerful tool for prototyping web applications, mentioning that for a very small cost, Dev4 Online could demonstrate to its clients effectively what their deliverable might look like and how it might function, and use it as a blueprint to further iterate on the client's ever-changing requirements. Gail was also particularly interested in how Crimson could be applied to a mobile environment. In future, mobile support could be added using a tool like Capacitor which makes web-based applications run as if they were native mobile applications.

The researchers asked Dev4 Online, if Crimson was fully featured, could it be beneficial for full-scale projects rather than just focussing on prototypes.

Gail mentioned that if it speeds up the development process, and is inherently flexible to more bespoke use cases, it will provide benefit.

Janos Mudri, a senior developer at Dev4 Online, discussed what Crimson would need to provide benefit to the software development process at a deeper technical level. He mentions that he likes the concept and iterates on the importance of flexibility which the current prototype doesn't have. He also mentions that Crimson needs to provide first party support for validation, both on the server and as visual feedback on the client. He also mentions that the authentication system needs to offer a high level of flexibility to be able to work with external APIs. Testing is another key factor outlined by Janos which was not mentioned in my design or implementation stage, mentioning that, developers should be able to apply unit tests to generated code, and other parts within the application.

Janos mentions that the generated backend is too tightly coupled with the frontend, explaining that developers need fine grained control of the backend and manual modifications to the backend project is a mandatory requirement of his. He says it's a great tool to kickstart your project, but the lack of backward compatibility if a modification is made to one of the output projects is critical. When the developer

makes a change within the input project any manual changes in the output project will be completely overwritten.

Gail agrees, mentioning that more experienced developers might want full control of the backend, and suggests that a migration tool would be hugely beneficial to Crimson. Like version control, if a change is made to the output project, the input project would need to merge these changes before recompiling. This is an extremely interesting proposal, it would require significant work to achieve, but necessary for Crimson to act as more than just a prototyping tool, or a project kick-starter.

Gail also potentially proposes Crimson as a training tool – something not considered previously during design or implementation. Particularly for frontend developers, Crimson could provide very specific examples on how to build APIs to serve the web applications frontend.

Janos also brings up a security concern, saying that the tightly coupled application grants frontend developers to have constraint-less access to databases and core infrastructure. The importance of separated environments here is key, also for larger scale applications, there should be enforced policies on what individual developers or groups can interact with.

John Basil, a software developer at Dev4 Online, agrees with all the comments outlined by Gail and Janos. Further enforcing the value of the feedback.

5.3 Research Questions

The current prototype is not yet sufficient to be able to answer the research questions set by this document. However, with future development, issues highlighted in the above evaluations could be resolved, allowing for an empirically study, comparing the development lifecycle between traditional development and traditional development assisted by Crimson, or the comparison of Crimson to other RAD tools such as LCDPs.

Chapter 7 – Conclusion

This dissertation investigated the conceptualisation, implementation, and informal evaluation of Crimson – a framework aiming to bridge the gap between RAD and traditional software development practices. While LCDPs offer faster development and often no software development experience, they often lack flexibility and introduce vendor lock-in, which can be problematic particularly if a platform was to shut down. Crimson, if fully featured, addresses these issues by integrating core concepts of LCDPs and other RAD tools into a traditional development environment.

The proof-of-concept successfully demonstrates component-driven route generation, research observations and external feedback highlights its potential for rapid prototyping or as a training tool for frontend developers looking for an introduction into backend development. However, current limitations such as limited range of components, lack of extensibility to backend logic, minimal security features, and absence of testing must be addressed for Crimson to be applicable as a primary software development tool.

Crimson, with further development, could become a powerful tool for increasing development efficiency without restricting control in the application of both prototyping and production of real-world web applications.

Chapter 8 – References

Vuetify. (2025). *All Vuetify Components — Vuetify*. [online] Available at: https://vuetifyjs.com/en/components/all/#data-and-display [Accessed 17 Apr. 2025].

Stalker, N. (2021). *Vue 3 PropType doesn't see the interface*. [online] Stack Overflow. Available at: https://stackoverflow.com/a/74127859.

Eric (2015). *Can I tell TypeScript that a Generic type has a constructor?* [online] Stack Overflow. Available at: https://stackoverflow.com/questions/33394791/can-i-tell-typescript-that-a-generic-type-has-a-constructor.

Acquaint Softtech, 2024. Eliminate Software Budget Overrun: #14 Facts & Statistics. [Online] Available at: https://acquaintsoft.com/blog/software-development-budget-overruns-facts statistics [Accessed 20th October 2024].

Andrzejewski, J., 2023. Component Libraries - Should you use them?. [Online] Available at: https://dev.to/jacobandrewsky/component-librarires-should-you-use-them-4ff7 [Accessed 28 October 2024].

Bloch, M., Blumberg, S. and Laartz, J., 2012. Delivering large-scale IT projects on time, on budget, and on value. Harvard Business Review, 5(1), pp. 1-6.

Bock, A.C. and Frank, U., 2021. Low-code platform. Business & Information Systems Engineering, 63, pp.733-740.

Darbyshire, 2023. Day 1: The History of No-Code and Low-Code Products. [Online] Available at: https://www.smartsuite.com/blog/history-no-code-low-code-products [Accessed 21st October 2024].

Figure 1, The Standish Group International Inc, 2015, CHAOS Report 2015. [Online] Available at: https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf [Accessed 20th October 2024]. pp. 1-13.

Figure 2, Mendix, 2024, Mircoflows. [Online] Available at: https://docs.mendix.com/refguide/microflows/ [Accessed 27th October 2024].

Figure 3, Ikkala, E., Hyvönen, E., Rantala, H. and Koho, M., 2022. Sampo-UI: A full stack JavaScript framework for developing semantic portal user interfaces. Semantic Web, 13(1), pp.69-84.

Formstack, n.d. The Rise of the No-Code Economy. [Online] Available at: https://resources.formstack.com/reports/rise-of-the-no-code-economy/history [Accessed 21st October 2024].

Google Workspace Updates, 2020. Google App Maker will be shut down on January 19, 2021. [Online] Available at: https://workspaceupdates.googleblog.com/2020/01/app-maker update.html [Accessed 23 October 2024].

IBM, n.d. What is low-code? [Online] Available at: https://www.ibm.com/topics/low-code [Accessed 21st October 2024].

Ikkala, E., Hyvönen, E., Rantala, H. and Koho, M., 2022. Sampo-UI: A full stack JavaScript framework for developing semantic portal user interfaces. Semantic Web, 13(1), pp.69-84.

Karlsson, A., 2021. Evaluating the State of Accessibility in React UI Component Libraries. Independent thesis Basic level (degree of Bachelor). pp. 1-39.

Khorram, F., Mottu, J.M. and Sunyé, G., 2020. Challenges & opportunities in low-code testing. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 1-10.

Luo, Y., Liang, P., Wang, C., Shahin, M. and Zhan, J., 2021. Characteristics and challenges of low-code development: the practitioners' perspective. In Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM). pp. 1-11.

Martins, R., Caldeira, F., Sa, F., Abbasi, M. and Martins, P., 2020. An overview on how to develop a low-code application using OutSystems. In 2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE) (pp. 395-401).

Molokken, K. and Jorgensen, M., 2003. A Review of Surveys on Software Effort Estimation. In 2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings. pp. 223-230.

Motta, J. S., 2024. How Telerik Transformed My Work and Why You Should Adopt It Too. [Online] Available at: https://www.telerik.com/blogs/how-telerik-transformed-work-why you-should-adopt-too [Accessed 28 October 2024].

Nan, N. and Harter, D.E., 2009. Impact of budget and schedule pressure on software development cycle time and effort. IEEE Transactions on Software Engineering, 35(5), pp. 624-637.

Onoufriou, A., 2024. Safety Risks and Security Threats in Low-code Software Development (Bachelor's thesis, University of Twente). pp.1-17.

OutSystems, 2024. Download source code. [Online] Available at: https://success.outsystems.com/documentation/11/reference/outsystems_apis/lifetime_api _v2/lifetime_api_examples/download_source_code/ [Accessed 22 October 2024].

OWASP, 2022. LCNC-SEC-05: Security Misconfiguration. [Online] Available at: https://owasp.org/www-project-top-10-low-code-no-code-security risks/content/2022/en/LCNC-SEC-05-Security-Misconfiguration [Accessed 26 October 2024].

Palumbo, S., Rehberg, B., and Li, H., 2024. Software Projects Don't Have to Be Late, Costly, and Irrelevant. [Online] Available at: https://www.bcg.com/publications/2024/software projects-dont-have-to-be-late-costly-and-irrelevant [Accessed 20th October 2024].

Rokis, K. and Kirikova, M., 2022, September. Challenges of low-code/no-code software development: A literature review. In International Conference on Business Informatics Research (pp. 3-17). Cham: Springer International Publishing.

Sahay, A., Indamutsa, A., Di Ruscio, D. and Pierantonio, A., 2020. Supporting the understanding and comparison of low-code development platforms. In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 171-178).

Semma, A., 2023. Speed Matters: Measuring Time to First Byte and Other Key Metrics in VueJS Components Frameworks: Array. Indonesian Journal of Informatics and Research, 4(1), pp.1-7.

Sherif, A., 2024. Tech organizations experiencing skills shortage worldwide 2015-2023. [Online] Available at: https://www.statista.com/statistics/1269776/worldwide-organizations talent-shortage-skills-tech [Accessed 20th October 2024].

Statista, 2024. Software – Worldwide. [Online] Available at: https://www.statista.com/outlook/tmo/software/worldwide [Accessed 20th October 2024].

Studio by UXPin, 2023. What is a Component Library, and Why Should You Use One for UI Development?. [Online] Available at: https://www.uxpin.com/studio/blog/ui-component library/ [Accessed 28 October 2024].

Team Kissflow, 2024. The History of Low-Code Platforms: How Development Changed. [Online] Available at: https://kissflow.com/low-code/history-of-low-code-development platforms/ [Accessed 21st October 2024].

Telerik, 2024. Modern UI Made Easy. [Online] Available at: https://www.telerik.com/ [Accessed 28 October 2024].

The Standish Group International Inc, 2015. CHAOS Report 2015. [Online] Available at: https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf [Accessed 20th October 2024]. pp. 1-13.

Verou, L., Zhang, A.X. and Karger, D.R., 2016. Mavo: creating interactive data-driven web applications by authoring HTML. In Proceedings of the 29th Annual Symposium on User Interface Software and Technology. pp. 483-496.

Whicher, A., 2024. Top 10 most in-demand tech jobs for 2024. [Online] Available at: https://www.hays.co.uk/career-advice/article/top-10-most-in-demand-tech-jobs-for-2024 [Accessed 20th October 2024].

World Economic Forum, 2023. Markets of Tomorrow: Turning Technologies into New Sources of Global Growth. [Online] Available at: https://www3.weforum.org/docs/WEF_Markets_of_Tomorrow_2023.pdf [Accessed 20 October 2024]. pp. 1-55.

itsdouges (2019). *GitHub - itsdouges/typescript-transformer-handbook: 📘 A comprehensive handbook on how to create transformers for TypeScript with code examples*. [online] GitHub. Available at: https://github.com/itsdouges/typescript-transformer-handbook [Accessed 21 Apr. 2025].

Appendices

1. Project Log

Date	**Progress**
June 2024	Met with my supervisor to discuss two potential dissertation ideas.
September 2024	Finalised project idea and wrote the Ethics Form and Project Proposal.
October 2024	Dedicated many hours to reviewing literature in relevant topics. Wrote and submitted the Project Scope containing found research in the form of a literature review, and relevant requirements and research questions.
December 2024	Investigated how to parse and transform source files and persist them.
January 2025	Designed how the developer would interact with the framework and experimented with generics in TypeScript at a more in-depth level, particularly focusing on how TypeORM models can interact with Vue components.
February 2025	Started implementing the core Crimson library, mostly focussing on utility and one skeleton component to define syntax

	(this skeleton component was later made into the Data Iterator component).
March 2025	Work was focussed on the compiler, to be able to convert the input project with the skeleton component into output projects.
April 2025	Finished the compiler, implemented 3 Crimson components. Demonstrated Crimson to DEV4 Online. Wrote the dissertation.
May 2025	Demonstrated Crimson to the university and public at the degree show.

2. Source Code

Source code is available for Crimson at the following GitHub link:
https://github.com/AdamGovier/Project-Crimson

3. Optional Component Parameters

Three other optional properties will be available on all Crimson components, a property allowing the developer to specify which authorisation levels can access the API routes, a property specifying a lambda function for filtering, and a property to specify if they wish to use a custom controller which overrides the default Crimson controller.

4. Crimson Controller Base Class Planned Methods

The base class will give the controller access to various utility methods:

- A method for getting a repository of the selected entity model, this again is of an unknown type to the library. The repository will allow for standard CRUD operations on the unknown entity model. Also, by extension another method for getting a repository of a concrete reference to an entity model for overridden controllers.
- A get context method, this method will allow interaction with the request and response object, for example to set a response code or read the request body.
- A get filter method, this method could be used to apply the lambda filtering function if present. The reason that this is needed and not automatically added by the compiler is that depending on where the filtering is placed, it

could dramatically change the results. For example, if results are paged, applying the filter at the end would only filter results on said page.

5. Client API Service Design

A planned advantageous design choice would be to utilise the method signatures as a client API service too. The compiler will swap out the server logic on the client with API request functionality and response parsing. Controllers would need to mark the expected return type with TypeScript for the IDE to understand what data is returned. Within the template there will be a method to specify which controller to use, the returned instance of the class could be stored as a variable (see Figure 6 for example usage of the API client).

6. Preserving Node Modules

A beneficial addition to the cleanup stage would be to preserve the 'node_modules' directory. The 'node_modules' directory which contains large amounts of external modules from third-party developers, takes up a large amount of space, reinstalling the modules each time could be inefficient. However, preserving 'node_modules' would increase complexity due to the need of tracking whether modules have been removed or added since last compilation.

7. Hot Reload

Every time the input project is changed, the compiler will be rerun and then the output client will be automatically started as well as the server. This will likely be implemented as a node package manager script in the input projects 'package.json' file. The script will likely use the tool Nodemon to keep track of changes.

8. Output Project's Core Technologies

- Node.JS with TypeScript – for running server-side JavaScript.
- Vue JS with TypeScript – a reactive componentised frontend framework for web applications.
- Fastify – A web framework like the more well-known Express JS. Fastify enables the routing of the API and interaction between request and responses.
- MongoDB – The chosen database due to the previously discussed all round flexibility.
- TypeORM – A database driver, handles connections and management of data operations and schema within the server.

9. Empty Methods

The reason they are empty is that they are purely placeholders. When a controller is compiled into a route, base method calls are replaced with required logic. For example,

for a Contact API route “this.getFocusedRepository()” would be replaced with “AppDataSource.getRepository(Contact)”. The reason that there are methods in the first place is to prevent the IDE from throwing undefined errors when implementing a Crimson component controller.

10. Included Crimson Components

The Data Iterator component simply gets a list of entity models back from the database, and allows the developer to provide their own templating, this could be used to create a grid of cards etc. The Data Table component is simple table which has pagination at the bottom and support for basic filtering, this has been implemented to perform this filtering on the server side. All the developer must do is provide which columns they’d like to display. The Data Table also has a built-in delete button at the end of each row. The form component acts as a form container with a slot to allow the developer to place form inputs, the form inputs value can then be bound to a sub property of the entity model. On submit, the form automatically handled the creation of an entity.

11. Client API Service Method Body Generation

Using the TypeScript compiler factory methods, 2 new statements are created.

Fetch Request

A call expression is created to built-in Fetch method, used to perform web requests. The Fetch method takes in two parameters, the URI and the request options. The URI is built from the host address, a dynamic sub path provided from the new parameter available on the controller method, and finally the route name which is identified from the remaining string of the method name (after the prepended HTTP method). The controller method also has an existing parameter called payload which contains properties such as the request body and query parameters, if query parameters are present these must also be appended to the URI. Next the options object is created, the body property is included if present within the payload, and another property is created for headers, the only header present currently is the content type which is always “application/json”. The response of the Fetch method is awaited and stored in a read only variable called response.

Response Parsing

After the fetch request, a statement is created to call the JSON method of the response object. This response is then used as the return value of the method.

12. Headers Object Generation using TypeScript Factory

To explain the complexity of the task, here is what is needed just to construct the headers object of a given request. A property is created within the options object using the “createPropertyAssignment” method, the first parameter of the method is the property key, the property key is created by calling the “createIdentifier” method with a string parameter given called “headers”. The next parameter of the “createPropertyAssignment” method is the property value, the value for the header’s property must be another object, to do this an object is created using the “createObjectLiteralExpression” method, this takes in array of properties. To add the property for content type you must call the “createPropertyAssignment” method which again takes in two parameters the key which is created with the “createStringLiteral” method which is given the parameter "Content-Type". The reason “createStringLiteral” is used here not “createIdentifier” is that "Content-Type" is not a valid JavaScript variable name and needs to be wrapped with double quotes as shown in JSON format. Then the second parameter of “createPropertyAssignment” is set to the output value of “createStringLiteral” which takes in the string “application/json”. As you can see for the creation a simple JavaScript object, a large amount of code is needed in the compiler, the creation of the whole fetch request is an order of magnitude bigger. I think this approach is rather unmaintainable, and not clearly understandable at first glance.

13. TS-Morph Simplification

TS-Morph acts as a wrapper to the TypeScript compiler which abstracts a lot of the requirements for builder methods at every single step of the creation or modification of blocks. For example, rather than using all these builder methods to create the content type header, I could just give it the object in standard JavaScript format directly.